1

HELLO, WORLD OF ASSEMBLY LANGUAGE

This chapter is a "quick-start" chapter that lets you start writing basic assembly language programs as rapidly as possible. This chapter does the following:

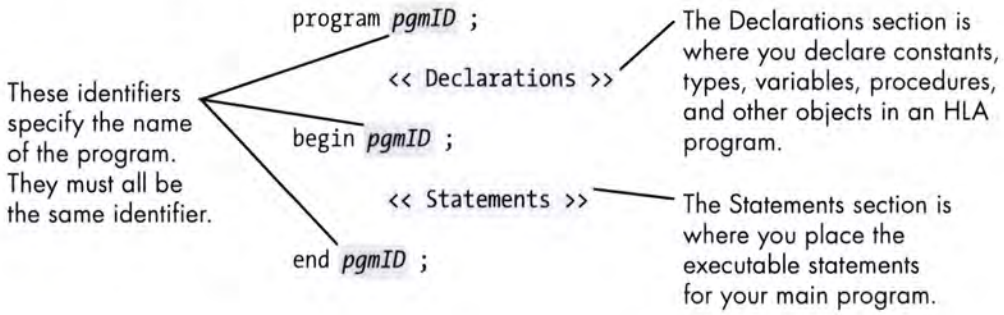
- Presents the basic syntax of an HLA (High Level Assembly) program
- Introduces you to the Intel CPU architecture
- Provides a handful of data declarations, machine instructions, and high-level control statements
- Describes some utility routines you can call in the HLA Standard Library
- Shows you how to write some simple assembly language programs

By the conclusion of this chapter, you should understand the basic syntax of an HLA program and should understand the prerequisites that are needed to start learning new assembly language features in the chapters that follow.

PETITIONER'S
Exhibit No.:
1021
26-Feb-25
[Kara Miller]

1.1 The Anatomy of an HLA Program

A typical HLA program takes the form shown in Figure 1-1.



program, begin, and end are HLA reserved words that delineate the program. Note the placement of the semicolons in this program.

Figure 1-1: Basic HLA program

pgmID in the template above is a user-defined program identifier. You must pick an appropriate descriptive name for your program. In particular, *pgmID* would be a horrible choice for any real program. If you are writing programs as part of a course assignment, your instructor will probably give you the name to use for your main program. If you are writing your own HLA program, you will have to choose an appropriate name for your project.

Identifiers in HLA are very similar to identifiers in most high-level languages. HLA identifiers may begin with an underscore or an alphabetic character and may be followed by zero or more alphanumeric or underscore characters. HLA's identifiers are *case neutral.* This means that the identifiers are case sensitive insofar as you must always spell an identifier exactly the same way in your program (even with respect to upper- and lowercase). However, unlike in case-sensitive languages such as C/C++, you may not declare two identifiers in the program whose name differs only by alphabetic case.

A traditional first program people write, popularized by Kernighan and Ritchie's *The C Programming Language*, is the "Hello, world!" program. This program makes an excellent concrete example for someone who is learning a new language. Listing 1-1 presents the HLA *helloWorld* program.

```
program helloWorld;
#include( "stdlib.hhf" );

begin helloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end helloWorld;
```

Listing 1-1: The helloWorld *program*

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000002

The `#include` statement in this program tells the HLA compiler to include a set of declarations from the *stdlib.hhf* (standard library, HLA Header File). Among other things, this file contains the declaration of the `stdout.put` code that this program uses.

The `stdout.put` statement is the print statement for the HLA language. You use it to write data to the standard output device (generally the console). To anyone familiar with I/O statements in a high-level language, it should be obvious that this statement prints the phrase `Hello, World of Assembly Language`. The `nl` appearing at the end of this statement is a constant, also defined in *stdlib.hhf*, that corresponds to the newline sequence.

Note that semicolons follow the `program`, `begin`, `stdout.put`, and `end` statements. Technically speaking, a semicolon does not follow the `#include` statement. It is possible to create include files that generate an error if a semicolon follows the `#include` statement, so you may want to get in the habit of not putting a semicolon here.

The `#include` is your first introduction to HLA declarations. The `#include` itself isn't actually a declaration, but it does tell the HLA compiler to substitute the file *stdlib.hhf* in place of the `#include` directive, thus inserting several declarations at this point in your program. Most HLA programs you will write will need to include one or more of the HLA Standard Library header files (*stdlib.hhf* actually includes all the standard library definitions into your program).

Compiling this program produces a *console* application. Running this program in a command window prints the specified string, and then control returns to the command-line interpreter (or *shell* in Unix terminology).

HLA is a free-format language. Therefore, you may split statements across multiple lines if this helps to make your programs more readable. For example, you could write the `stdout.put` statement in the *helloWorld* program as follows:

```
stdout.put
(
    "Hello, World of Assembly Language",
    nl
);
```

Another construction you'll see appearing in example code throughout this text is that HLA automatically concatenates any adjacent string constants it finds in your source file. Therefore, the statement above is also equivalent to

```
stdout.put
(
    "Hello, "
    "World of Assembly Language",
    nl
);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000003

Indeed, `nl` (the newline) is really nothing more than a string constant, so (technically) the comma between the `nl` and the preceding string isn't necessary. You'll often see the above written as

```
stdout.put( "Hello, World of Assembly Language" nl );
```

Notice the lack of a comma between the string constant and `nl`; this turns out to be legal in HLA, though it applies only to certain constants; you may not, in general, drop the comma. Chapter 4 explains in detail how this works. This discussion appears here because you'll probably see this "trick" employed by sample code prior to the formal explanation.

1.2 Running Your First HLA Program

The whole purpose of the "Hello, world!" program is to provide a simple example by which someone who is learning a new programming language can figure out how to use the tools needed to compile and run programs in that language. True, the *helloWorld* program in Section 1.1 helps demonstrate the format and syntax of a simple HLA program, but the real purpose behind a program like *helloWorld* is to learn how to create and run a program from beginning to end. Although the previous section presents the layout of an HLA program, it did not discuss how to edit, compile, and run that program. This section will briefly cover those details.

All of the software you need to compile and run HLA programs can be found at *http://www.artofasm.com/* or at *http://webster.cs.ucr.edu/*. Select **High Level Assembly** from the Quick Navigation Panel and then the Download HLA link from that page. HLA is currently available for Windows, Mac OS X, Linux, and FreeBSD. Download the appropriate version of the HLA software for your system. From the Download HLA web page, you will also be able to download all the software associated with this book. If the HLA download doesn't include them, you will probably want to download the HLA reference manual and the HLA Standard Library reference manual along with HLA and the software for this book. This text does not describe the entire HLA language, nor does it describe the entire HLA Standard Library. You'll want to have these reference manuals handy as you learn assembly language using HLA.

This section will not describe how to install and set up the HLA system because those instructions change over time. The HLA download page for each of the operating systems describes how to install and use HLA. Please consult those instructions for the exact installation procedure.

Creating, compiling, and running an HLA program is very similar to the process you'd use when creating, compiling, or running a program in any computer language. First, because HLA is not an *integrated development environment (IDE)* that allows you to edit, compile, test and debug, and run your application all from within the same program, you'll create and edit HLA programs using a text editor.[1]

[1] HIDE (HLA Integrated Development Environment) is an IDE available for Windows users. See the High Level Assembly web page for details on downloading HIDE.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000004

Windows, Mac OS X, Linux, and FreeBSD offer many text editor options. You can even use the text editor provided with other IDEs to create and edit HLA programs (such as those found in Visual C++, Borland's Delphi, Apple's Xcode, and similar languages). The only restriction is that HLA expects ASCII text files, so the editor you use must be capable of manipulating and saving text files. Under Windows you can always use Notepad to create HLA programs. If you're working under Linux and FreeBSD you can use joe, vi, or emacs. Under Mac OS X you can use XCode or Text Wrangler or another editor of your preference.

The HLA compiler[2] is a traditional *command-line compiler*, which means that you need to run it from a Windows *command-line prompt* or a Linux/FreeBSD/Mac OS X *shell*. To do so, enter something like the following into the command-line prompt or shell window:

```
hla hw.hla
```

This command tells HLA to compile the *hw.hla* (*helloWorld*) program to an executable file. Assuming there are no errors, you can run the resulting program by typing the following command into your command prompt window (Windows):

```
hw
```

or into the shell interpreter window (Linux/FreeBSD/Mac OS X):

```
./hw
```

If you're having problems getting the program to compile and run properly, please see the HLA installation instructions on the HLA download page. These instructions describe in great detail how to install, set up, and use HLA.

1.3 Some Basic HLA Data Declarations

HLA provides a wide variety of constant, type, and data declaration statements. Later chapters will cover the declaration sections in more detail, but it's important to know how to declare a few simple variables in an HLA program.

HLA predefines several different signed integer types including int8, int16, and int32, corresponding to 8-bit (1-byte) signed integers, 16-bit (2-byte) signed integers, and 32-bit (4-byte) signed integers, respectively.[3] Typical variable declarations occur in the HLA *static variable section*. A typical set of variable declarations takes the form shown in Figure 1-2.

[2] Traditionally, programmers have always called translators for assembly languages *assemblers* rather than *compilers*. However, because of HLA's high-level features, it is more proper to call HLA a compiler rather than an assembler.

[3] A discussion of bits and bytes will appear in Chapter 2 for those who are unfamiliar with these terms.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000005

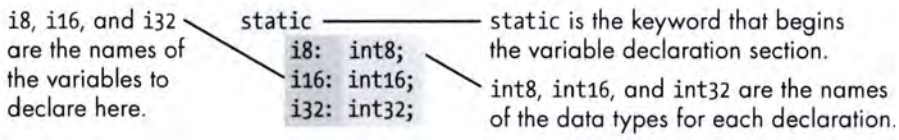


Figure 1-2: Static variable declarations

Those who are familiar with the Pascal language should be comfortable with this declaration syntax. This example demonstrates how to declare three separate integers: i8, i16, and i32. Of course, in a real program you should use variable names that are more descriptive. While names like *i8* and *i32* describe the type of the object, they do not describe its purpose. Variable names should describe the purpose of the object.

In the *static declaration section*, you can also give a variable an initial value that the operating system will assign to the variable when it loads the program into memory. Figure 1-3 provides the syntax for this.

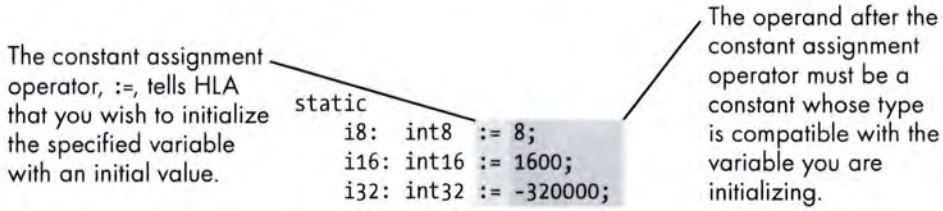


Figure 1-3: Static variable initialization

It is important to realize that the expression following the assignment operator (:=) must be a constant expression. You cannot assign the values of other variables within a static variable declaration.

Those familiar with other high-level languages (especially Pascal) should note that you can declare only one variable per statement. That is, HLA does not allow a comma-delimited list of variable names followed by a colon and a type identifier. Each variable declaration consists of a single identifier, a colon, a type ID, and a semicolon.

Listing 1-2 provides a simple HLA program that demonstrates the use of variables within an HLA program.

```
Program DemoVars;
#include( "stdlib.hhf" )

static
    InitDemo:       int32 := 5;
    NotInitialized: int32;

begin DemoVars;

    // Display the value of the pre-initialized variable:

    stdout.put( "InitDemo's value is ", InitDemo, nl );

    // Input an integer value from the user and display that value:
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000006

```
    stdout.put( "Enter an integer value: " );
    stdin.get( NotInitialized );
    stdout.put( "You entered: ", NotInitialized, nl );

end DemoVars;
```

Listing 1-2: Variable declaration and use

In addition to static variable declarations, this example introduces three new concepts. First, the `stdout.put` statement allows multiple parameters. If you specify an integer value, `stdout.put` will convert that value to its string representation on output.

The second new feature introduced in Listing 1-2 is the `stdin.get` statement. This statement reads a value from the standard input device (usually the keyboard), converts the value to an integer, and stores the integer value into the `NotInitialized` variable. Finally, Listing 1-2 also introduces the syntax for (one form of) HLA comments. The HLA compiler ignores all text from the `//` sequence to the end of the current line. (Those familiar with Java, C++, and Delphi should recognize these comments.)

1.4 Boolean Values

HLA and the HLA Standard Library provide limited support for boolean objects. You can declare boolean variables, use boolean literal constants, use boolean variables in boolean expressions, and you can print the values of boolean variables.

Boolean literal constants consist of the two predefined identifiers `true` and `false`. Internally, HLA represents the value true using the numeric value `1`; HLA represents false using the value `0`. Most programs treat `0` as false and anything else as true, so HLA's representations for true and false should prove sufficient.

To declare a boolean variable, you use the `boolean` data type. HLA uses a single byte (the least amount of memory it can allocate) to represent boolean values. The following example demonstrates some typical declarations:

```
static
    BoolVar:     boolean;
    HasClass:    boolean := false;
    IsClear:     boolean := true;
```

As this example demonstrates, you can initialize boolean variables if you desire.

Because boolean variables are byte objects, you can manipulate them using any instructions that operate directly on 8-bit values. Furthermore, as long as you ensure that your boolean variables only contain `0` and `1` (for false and true, respectively), you can use the 80x86 `and`, `or`, `xor`, and `not` instructions to manipulate these boolean values (these instructions are covered in Chapter 2).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000007

You can print boolean values by making a call to the `stdout.put` routine. For example:

```
stdout.put( BoolVar )
```

This routine prints the text `true` or `false` depending upon the value of the boolean parameter (`0` is false; anything else is true). Note that the HLA Standard Library does not allow you to read boolean values via `stdin.get`.

1.5 Character Values

HLA lets you declare 1-byte ASCII character objects using the `char` data type. You may initialize character variables with a literal character value by surrounding the character with a pair of apostrophes. The following example demonstrates how to declare and initialize character variables in HLA:

```
static
    c: char;
    LetterA: char := 'A';
```

You can print character variables use the `stdout.put` routine, and you can read character variables using the `stdin.get` procedure call.

1.6 An Introduction to the Intel 80x86 CPU Family

Thus far, you've seen a couple of HLA programs that will actually compile and run. However, all the statements appearing in programs to this point have been either data declarations or calls to HLA Standard Library routines. There hasn't been any *real* assembly language. Before we can progress any further and learn some real assembly language, a detour is necessary; unless you understand the basic structure of the Intel 80x86 CPU family, the machine instructions will make little sense.

The Intel CPU family is generally classified as a *Von Neumann Architecture Machine.* Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices.* These three components are interconnected using the *system bus* (consisting of the address, data, and control buses). The block diagram in Figure 1-4 shows this relationship.

The CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select one of the memory locations or I/O device port locations, each of which has a unique binary numeric *address.* Then the CPU, memory, and I/O devices pass data among themselves by placing the data on the data bus. The control bus contains signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000008

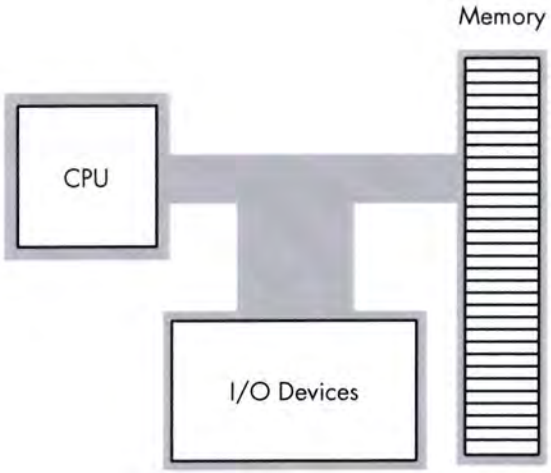


Figure 1-4: Von Neumann computer system block diagram

The 80x86 CPU registers can be broken down into four categories: general-purpose registers, special-purpose application-accessible registers, segment registers, and special-purpose kernel-mode registers. Because the segment registers aren't used much in modern 32-bit operating systems (such as Windows, Mac OS X, FreeBSD, and Linux) and because this text is geared to writing programs written for 32-bit operating systems, there is little need to discuss the segment registers. The special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools. Such software construction is well beyond the scope of this text.

The 80x86 (Intel family) CPUs provide several general-purpose registers for application use. These include eight 32-bit registers that have the following names: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP.

The *E* prefix on each name stands for *extended.* This prefix differentiates the 32-bit registers from the eight 16-bit registers that have the following names: AX, BX, CX, DX, SI, DI, BP, and SP.

Finally, the 80x86 CPUs provide eight 8-bit registers that have the following names: AL, AH, BL, BH, CL, CH, DL, and DH.

Unfortunately, these are not all separate registers. That is, the 80x86 does not provide 24 independent registers. Instead, the 80x86 overlays the 32-bit registers with the 16-bit registers, and it overlays the 16-bit registers with the 8-bit registers. Figure 1-5 shows this relationship.

The most important thing to note about the general-purpose registers is that they are not independent. Modifying one register may modify as many as three other registers. For example, modification of the EAX register may very well modify the AL, AH, and AX registers. This fact cannot be overemphasized here. A very common mistake in programs written by beginning assembly language programmers is register value corruption because the programmer did not completely understand the ramifications of the relationship shown in Figure 1-5.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000009

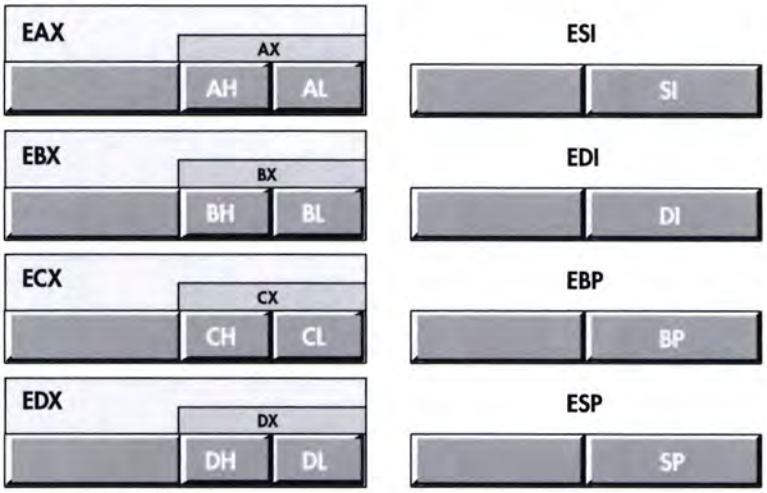


Figure 1-5: 80x86 (Intel CPU) general-purpose registers

The EFLAGS register is a 32-bit register that encapsulates several single-bit boolean (true/false) values. Most of the bits in the EFLAGS register are either reserved for kernel mode (operating system) functions or are of little interest to the application programmer. Eight of these bits (or *flags*) are of interest to application programmers writing assembly language programs. These are the overflow, direction, interrupt disable,[4] sign, zero, auxiliary carry, parity, and carry flags. Figure 1-6 shows the layout of the flags within the lower 16 bits of the EFLAGS register.

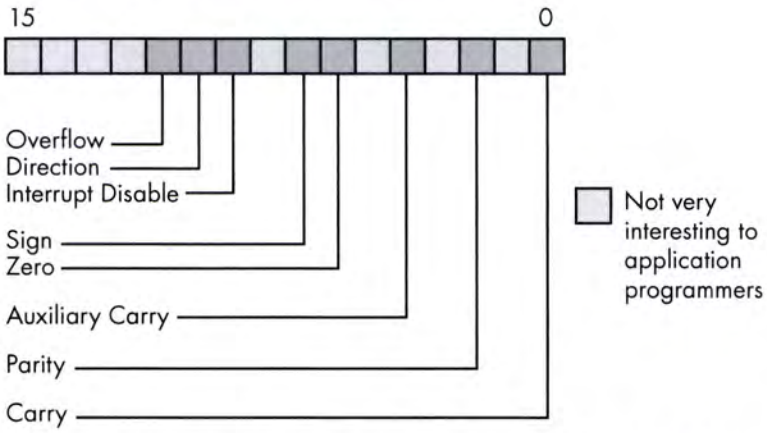


Figure 1-6: Layout of the FLAGS register (lower 16 bits of EFLAGS)

Of the eight flags that are of interest to application programmers, four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags. Collectively, we will call these four flags the *condition codes*.[5] The state of these flags lets you test the result of previous computations. For example, after comparing two values, the condition code flags will tell you whether one value is less than, equal to, or greater than a second value.

[4] Application programs cannot modify the interrupt flag, but we'll look at this flag in Chapter 2; hence the discussion of this flag here.

[5] Technically the parity flag is also a condition code, but we will not use that flag in this text.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000010

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the 80x86 CPU involve a register. For example, to add two variables together, storing the sum into a third variable, you must load one of the variables into a register, add the second operand to the value in the register, and then store the register away in the destination variable. Registers are a middleman in nearly every calculation. Therefore, registers are very important in 80x86 assembly language programs.

Another thing you should be aware of is that although the registers have the name "general purpose," you should not infer that you can use any register for any purpose. All the 80x86 registers have their own special purposes that limit their use in certain contexts. The SP/ESP register pair, for example, has a very special purpose that effectively prevents you from using it for anything else (it's the *stack pointer*). Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general-purpose register. For the time being, you should avoid the use of the ESP and EBP registers for generic calculations; also, keep in mind that the remaining registers are not completely interchangeable in your programs.

1.7 The Memory Subsystem

A typical 80x86 processor running a modern 32-bit OS can access a maximum of 2^{32} different memory locations, or just over 4 billion bytes. A few years ago, 4 gigabytes of memory would have seemed like infinity; modern machines, however, exceed this limit. Nevertheless, because the 80x86 architecture supports a maximum 4GB address space when using a 32-bit operating system like Windows, Mac OS X, FreeBSD, or Linux, the following discussion will assume the 4GB limit.

Of course, the first question you should ask is, "What exactly is a memory location?" The 80x86 supports *byte-addressable memory*. Therefore, the basic memory unit is a byte, which is sufficient to hold a single character or a (very) small integer value (we'll talk more about that in Chapter 2).

Think of memory as a linear array of bytes. The address of the first byte is 0 and the address of the last byte is $2^{32}-1$. For an 80x86 processor, the following pseudo-Pascal array declaration is a good approximation of memory:

```
Memory: array [0..4294967295] of byte;
```

C/C++ and Java users might prefer the following syntax:

```
byte Memory[4294967296];
```

To execute the equivalent of the Pascal statement `Memory [125] := 0;` the CPU places the value 0 on the data bus, places the address 125 on the address bus, and asserts the write line (this generally involves setting that line to 0), as shown in Figure 1-7.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000011

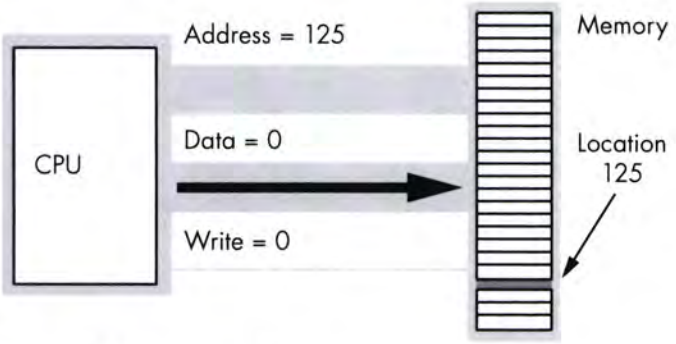


Figure 1-7: Memory write operation

To execute the equivalent of `CPU := Memory [125];` the CPU places the address 125 on the address bus, asserts the read line (because the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1-8).

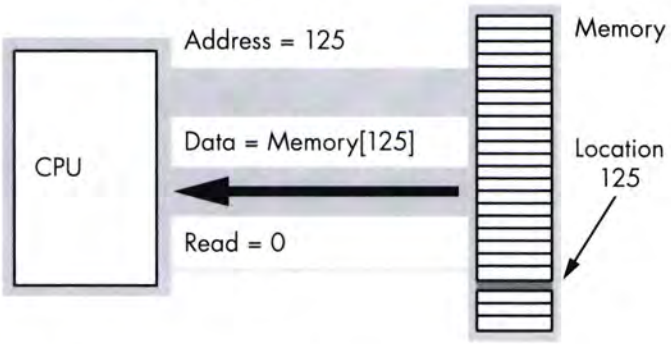


Figure 1-8: Memory read operation

This discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Because memory consists of an array of bytes, how can we possibly deal with values larger than a single byte? Easy—to store larger values, the 80x86 uses a sequence of consecutive memory locations. Figure 1-9 shows how the 80x86 stores bytes, words (2 bytes), and double words (4 bytes) in memory. The memory address of each of these objects is the address of the first byte of each object (that is, the lowest address).

Modern 80x86 processors don't actually connect directly to memory. Instead, there is a special memory buffer on the CPU known as the *cache* (pronounced "cash") that acts as a high-speed intermediary between the CPU and main memory. Although the cache handles the details automatically for you, one fact you should know is that accessing data objects in memory is sometimes more efficient if the address of the object is an even multiple of the object's size. Therefore, it's a good idea to *align* 4-byte objects (double words) on addresses that are multiples of 4. Likewise, it's most

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000012

efficient to align 2-byte objects on even addresses. You can efficiently access single-byte objects at any address. You'll see how to set the alignment of memory objects in Section 3.4.

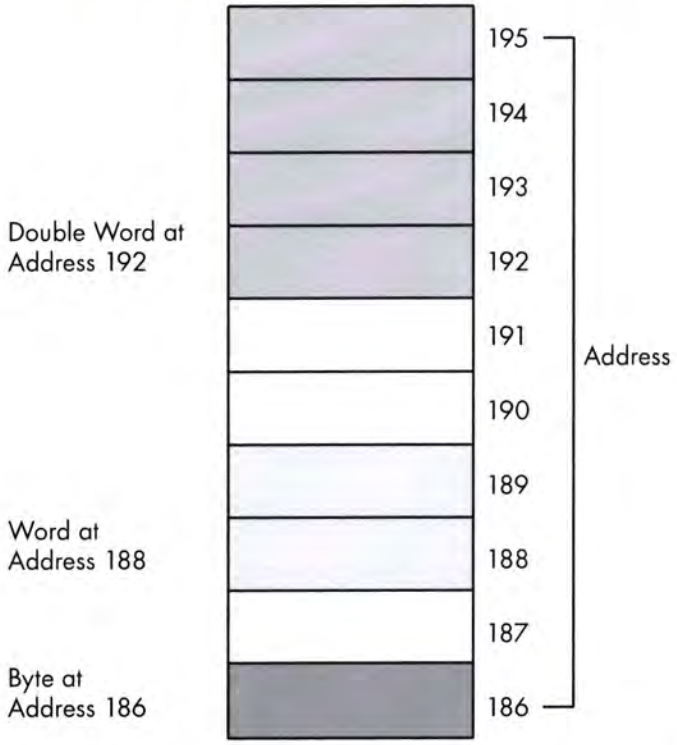


Figure 1-9: Byte, word, and double-word storage in memory

Before leaving this discussion of memory objects, it's important to understand the correspondence between memory and HLA variables. One of the nice things about using an assembler/compiler like HLA is that you don't have to worry about numeric memory addresses. All you need to do is declare a variable in HLA, and HLA takes care of associating that variable with some unique set of memory addresses. For example, if you have the following declaration section:

```
static
    i8          :int8;
    i16         :int16;
    i32         :int32;
```

HLA will find some unused 8-bit byte in memory and associate it with the `i8` variable; it will find a pair of consecutive unused bytes and associate `i16` with them; finally, HLA will find 4 consecutive unused bytes and associate the value of `i32` with those 4 bytes (32 bits). You'll always refer to these variables by their name. You generally don't have to concern yourself with their numeric address. Still, you should be aware that HLA is doing this for you behind your back.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000013

1.8 Some Basic Machine Instructions

The 80x86 CPU family provides from just over a hundred to many thousands of different machine instructions, depending on how you define a machine instruction. Even at the low end of the count (greater than 100), it appears as though there are far too many machine instructions to learn in a short time. Fortunately, you don't need to know all the machine instructions. In fact, most assembly language programs probably use around 30 different machine instructions.[6] Indeed, you can certainly write several meaningful programs with only a few machine instructions. The purpose of this section is to provide a small handful of machine instructions so you can start writing simple HLA assembly language programs right away.

Without question, the `mov` instruction is the most oft-used assembly language statement. In a typical program, anywhere from 25 percent to 40 percent of the instructions are `mov` instructions. As its name suggests, this instruction moves data from one location to another.[7] The HLA syntax for this instruction is:

```
mov( source_operand, destination_operand );
```

The *`source_operand`* can be a register, a memory variable, or a constant. The *`destination_operand`* may be a register or a memory variable. Technically the 80x86 instruction set does not allow both operands to be memory variables. HLA, however, will automatically translate a `mov` instruction with two-word or double-word memory operands into a pair of instructions that will copy the data from one location to another. In a high-level language like Pascal or C/C++, the `mov` instruction is roughly equivalent to the following assignment statement:

```
destination_operand = source_operand ;
```

Perhaps the major restriction on the `mov` instruction's operands is that they must both be the same size. That is, you can move data between a pair of byte (8-bit) objects, word (16-bit) objects, or double-word (32-bit) objects; you may not, however, mix the sizes of the operands. Table 1-1 lists all the legal combinations for the `mov` instruction.

You should study this table carefully because most of the general-purpose 80x86 instructions use this syntax.

[6] Different programs may use a different set of 30 instructions, but few programs use more than 30 distinct instructions.

[7] Technically, `mov` actually copies data from one location to another. It does not destroy the original data in the source operand. Perhaps a better name for this instruction would have been `copy`. Alas, it's too late to change it now.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000014

Table 1-1: Legal 80x86 mov Instruction Operands

Source	Destination
Reg_8*	Reg_8
Reg_8	Mem_8
Mem_8	Reg_8
Constant†	Reg_8
Constant	Mem_8
Reg_{16}	Reg_{16}
Reg_{16}	Mem_{16}
Mem_{16}	Reg_{16}
Constant	Reg_{16}
Constant	Mem_{16}
Reg_{32}	Reg_{32}
Reg_{32}	Mem_{32}
Mem_{32}	Reg_{32}
Constant	Reg_{32}
Constant	Mem_{32}

* The suffix denotes the size of the register or memory location.

† The constant must be small enough to fit in the specified destination operand.

The 80x86 add and sub instructions let you add and subtract two operands. Their syntax is nearly identical to the mov instruction:

```
add( source_operand, destination_operand );
sub( source_operand, destination_operand );
```

The add and sub operands take the same form as the mov instruction.[8] The add instruction does the following:

```
destination_operand = destination_operand + source_operand ;
destination_operand += source_operand;  // For those who prefer C syntax.
```

The sub instruction does the calculation:

```
destination_operand = destination_operand - source_operand ;
destination_operand -= source_operand ;  // For C fans.
```

With nothing more than these three instructions, plus the HLA control structures that the next section discusses, you can actually write some sophisticated programs. Listing 1-3 provides a sample HLA program that demonstrates these three instructions.

[8] Remember, though, that add and sub do not support memory-to-memory operations.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000015

```
program DemoMOVaddSUB;

#include( "stdlib.hhf" )

static
    i8:     int8    := -8;
    i16:    int16   := -16;
    i32:    int32   := -32;

begin DemoMOVaddSUB;

    // First, print the initial values
    // of our variables.

    stdout.put
    (
        nl,
        "Initialized values: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );

    // Compute the absolute value of the
    // three different variables and
    // print the result.
    // Note: Because all the numbers are
    // negative, we have to negate them.
    // Using only the mov, add, and sub
    // instructions, we can negate a value
    // by subtracting it from zero.

    mov( 0, al );   // Compute i8 := -i8;
    sub( i8, al );
    mov( al, i8 );

    mov( 0, ax );   // Compute i16 := -i16;
    sub( i16, ax );
    mov( ax, i16 );

    mov( 0, eax );  // Compute i32 := -i32;
    sub( i32, eax );
    mov( eax, i32 );

    // Display the absolute values:

    stdout.put
    (
        nl,
        "After negation: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000016

```
    );

    // Demonstrate add and constant-to-memory
    // operations:

    add( 32323200, i32 );
    stdout.put( nl, "After add: i32=", i32, nl );


end DemoMOVaddSUB;
```

Listing 1-3: Demonstration of the mov, add, and sub instructions

1.9 Some Basic HLA Control Structures

The mov, add, and sub instructions, while valuable, aren't sufficient to let you write meaningful programs. You will need to complement these instructions with the ability to make decisions and create loops in your HLA programs before you can write anything other than a simple program. HLA provides several high-level control structures that are very similar to control structures found in high-level languages. These include if..then..elseif..else..endif, while..endwhile, repeat..until, and so on. By learning these statements you will be armed and ready to write some real programs.

Before discussing these high-level control structures, it's important to point out that these are not real 80x86 assembly language statements. HLA compiles these statements into a sequence of one or more real assembly language statements for you. In Chapter 7, you'll learn how HLA compiles the statements, and you'll learn how to write pure assembly language code that doesn't use them. However, there is a lot to learn before you get to that point, so we'll stick with these high-level language statements for now.

Another important fact to mention is that HLA's high-level control structures are *not* as high level as they first appear. The purpose behind HLA's high-level control structures is to let you start writing assembly language programs as quickly as possible, not to let you avoid the use of assembly language altogether. You will soon discover that these statements have some severe restrictions associated with them, and you will quickly outgrow their capabilities. This is intentional. Once you reach a certain level of comfort with HLA's high-level control structures and decide you need more power than they have to offer, it's time to move on and learn the real 80x86 instructions behind these statements.

Do not let the presence of high-level-like statements in HLA confuse you. Many people, after learning about the presence of these statements in the HLA language, erroneously come to the conclusion that HLA is just some special high-level language and not a true assembly language. This isn't true. HLA is a full low-level assembly language. HLA supports all the same machine instructions as any other 80x86 assembler. The difference is that HLA has some *extra* statements that allow you to do *more* than is possible with those other 80x86 assemblers. Once you learn 80x86 assembly

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000017

language with HLA, you may elect to ignore all these extra (high-level) statements and write only low-level 80x86 assembly language code if this is your desire.

The following sections assume that you're familiar with at least one high-level language. They present the HLA control statements from that perspective without bothering to explain how you actually use these statements to accomplish something in a program. One prerequisite this text assumes is that you already know how to use these generic control statements in a high-level language; you'll use them in HLA programs in an identical manner.

1.9.1 Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the `if`, `while`, and `repeat..until` statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high-level control structures. This is one area where your familiarity with a high-level language will work against you—you'll want to use the fancy expressions you use in a high-level language, yet HLA supports only some basic forms.

HLA boolean expressions take the following forms:[9]

```
flag_specification
!flag_specification
register
!register
Boolean_variable
!Boolean_variable
mem_reg relop mem_reg_const
register in LowConst..HiConst
register not in LowConst..HiConst
```

A `flag_specification` may be one of the symbols that are described in Table 1-2.

Table 1-2: Symbols for `flag_specification`

Symbol	Meaning	Explanation
@c	Carry	True if the carry is set (1); false if the carry is clear (0).
@nc	No carry	True if the carry is clear (0); false if the carry is set (1).
@z	Zero	True if the zero flag is set; false if it is clear.
@nz	Not zero	True if the zero flag is clear; false if it is set.
@o	Overflow	True if the overflow flag is set; false if it is clear.
@no	No overflow	True if the overflow flag is clear; false if it is set.
@s	Sign	True if the sign flag is set; false if it is clear.
@ns	No sign	True if the sign flag is clear; false if it is set.

[9] There are a few additional forms that we'll cover in Chapter 6.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000018

The use of the flag values in a boolean expression is somewhat advanced. You will begin to see how to use these boolean expression operands in the next chapter.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general-purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a nonzero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or nonzero (true). Because HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires such variables be of type `boolean`. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. Table 1-3 lists the legal combinations.

Table 1-3: Legal Boolean Expressions

Left Operand	Relational Operator	Right Operand
Memory variable or register	= or == <> or != < <= > >=	Variable, register, or constant

Note that both operands cannot be memory operands. In fact, if you think of the *right operand* as the source operand and the *left operand* as the destination operand, then the two operands must be the same that `add` and `sub` allow.

Also like the `add` and `sub` instructions, the two operands must be the same size. That is, they must both be byte operands, they must both be word operands, or they must both be double-word operands. If the right operand is a constant, its value must be in the range that is compatible with the left operand.

There is one other issue: if the left operand is a register and the right operand is a positive constant or another register, HLA uses an *unsigned* comparison. The next chapter will discuss the ramifications of this; for the time being, do not compare negative values in a register against a constant or another register. You may not get an intuitive result.

The `in` and `not in` operators let you test a register to see if it is within a specified range. For example, the expression `eax in 2000..2099` evaluates true if the value in the EAX register is between 2,000 and 2,099 (inclusive). The `not in` (two words) operator checks to see if the value in a register is outside the specified range. For example, `al not in 'a'..'z'` evaluates true if the character in the AL register is not a lowercase alphabetic character.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000019

```
    stdout.put( "Enter an integer between 0 and 127: " );
    stdin.get( PosValue );

    stdout.put( nl, "Value in hexadecimal: $" );
    stdout.puth8( PosValue );

    mov( PosValue, al );
    not( al );
    stdout.put( nl, "Invert all the bits: $", al, nl );
    add( 1, al );
    stdout.put( "Add one: $", al, nl );
    mov( al, NegValue );
    stdout.put( "Result in decimal: ", NegValue, nl );

    stdout.put
    (
        nl,
        "Now do the same thing with the NEG instruction: ",
        nl
    );
    mov( PosValue, al );
    neg( al );
    mov( al, NegValue );
    stdout.put( "Hex result = $", al, nl );
    stdout.put( "Decimal result = ", NegValue, nl );

end twosComplement;
```

Listing 2-6: twosComplement example

As you've seen previously, you use the `int8`, `int16`, `int32`, `int64`, and `int128` data types to reserve storage for signed integer variables. You've also seen routines like `stdout.puti8` and `stdin.geti32` that read and write signed integer values. Because this section has made it abundantly clear that you must differentiate signed and unsigned calculations in your programs, you should probably be asking yourself, "How do I declare and use unsigned integer variables?"

The first part of the question, "How do I declare unsigned integer variables," is the easiest to answer. You simply use the `uns8`, `uns16`, `uns32`, `uns64`, and `uns128` data types when declaring the variables. For example:

```
static
    u8:         uns8;
    u16:        uns16;
    u32:        uns32;
    u64:        uns64;
    u128:       uns128;
```

As for using these unsigned variables, the HLA Standard Library provides a complementary set of input/output routines for reading and displaying unsigned variables. As you can probably guess, these routines include `stdout.putu8`, `stdout.putu16`, `stdout.putu32`, `stdout.putu64`, `stdout.putu128`,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000021

stdout.putu8Size, stdout.putu16Size, stdout.putu32Size, stdout.putu64Size, stdout.putu128Size, stdin.getu8, stdin.getu16, stdin.getu32, stdin.getu64, and stdin.getu128. You use these routines just as you would use their signed integer counterparts except you get to use the full range of the unsigned values with these routines. The source code in Listing 2-7 demonstrates unsigned I/O as well as demonstrates what can happen if you mix signed and unsigned operations in the same calculation.

```
program UnsExample;
#include( "stdlib.hhf" )

static
    UnsValue:   uns16;

begin UnsExample;

    stdout.put( "Enter an integer between 32,768 and 65,535: " );
    stdin.getu16();
    mov( ax, UnsValue );

    stdout.put
    (
        "You entered ",
        UnsValue,
        ".  If you treat this as a signed integer, it is "
    );
    stdout.puti16( UnsValue );
    stdout.newln();

end UnsExample;
```

Listing 2-7: Unsigned I/O

2.9 Sign Extension, Zero Extension, Contraction, and Saturation

Because two's complement format integers have a fixed length, a small problem develops. What happens if you need to convert an 8-bit two's complement value to 16 bits? This problem and its converse (converting a 16-bit value to 8 bits) can be accomplished via *sign extension* and *contraction* operations.

Consider the value −64. The 8-bit two's complement value for this number is $C0. The 16-bit equivalent of this number is $FFC0. Now consider the value +64. The 8- and 16-bit versions of this value are $40 and $0040, respectively. The difference between the 8- and 16-bit numbers can be described by the rule, "If the number is negative, the H.O. byte of the 16-bit number contains $FF; if the number is positive, the H.O. byte of the 16-bit quantity is 0."

To extend a signed value from some number of bits to a greater number of bits is easy; just copy the sign bit into all the additional bits in the new

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000022

format. For example, to sign extend an 8-bit number to a 16-bit number, simply copy bit 7 of the 8-bit number into bits 8..15 of the 16-bit number. To sign extend a 16-bit number to a double word, simply copy bit 15 into bits 16..31 of the double word.

You must use sign extension when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division, in particular) may require a sign extension to 32 bits:

```
Sign Extension:
8 Bits      16 Bits      32 Bits

$80         $FF80        $FFFF_FF80
$28         $0028        $0000_0028
$9A         $FF9A        $FFFF_FF9A
$7F         $007F        $0000_007F
            $1020        $0000_1020
            $8086        $FFFF_8086
```

To extend an unsigned value to a larger one, you must zero extend the value. Zero extension is very easy—just store a 0 into the H.O. byte(s) of the larger operand. For example, to zero extend the 8-bit value $82 to 16 bits, you simply add a 0 to the H.O. byte, yielding $0082.

```
Zero Extension:
8 Bits      16 Bits      32 Bits

$80         $0080        $0000_0080
$28         $0028        $0000_0028
$9A         $009A        $0000_009A
$7F         $007F        $0000_007F
            $1020        $0000_1020
            $8086        $0000_8086
```

The 80x86 provides several instructions that will let you sign or zero extend a smaller number to a larger number. Table 2-6 lists a group of instructions that will sign extend the AL, AX, or EAX register.

Table 2-6: Instructions for Extending AL, AX, and EAX

Instruction	Explanation
cbw();	Converts the byte in AL to a word in AX via sign extension.
cwd();	Converts the word in AX to a double word in DX:AX via sign extension.
cdq();	Converts the double word in EAX to the quad word in EDX:EAX via sign extension.
cwde();	Converts the word in AX to a double word in EAX via sign extension.

Note that the cwd (convert word to double word) instruction does not sign extend the word in AX to the double word in EAX. Instead, it stores the

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000023

H.O. word of the sign extension into the DX register (the notation DX:AX tells you that you have a double-word value with DX containing the upper 16 bits and AX containing the lower 16 bits of the value). If you want the sign extension of AX to go into EAX, you should use the `cwde` (convert word to double word, extended) instruction.

The four instructions above are unusual in the sense that these are the first instructions you've seen that do not have any operands. These instructions' operands are *implied* by the instructions themselves.

Within a few chapters you will discover just how important these instructions are and why the `cwd` and `cdq` instructions involve the DX and EDX registers. However, for simple sign extension operations, these instructions have a few major drawbacks—you do not get to specify the source and destination operands, and the operands must be registers.

For general sign extension operations, the 80x86 provides an extension of the `mov` instruction, `movsx` (move with sign extension), that copies data and sign extends the data while copying it. The `movsx` instruction's syntax is very similar to the `mov` instruction:

```
movsx( source, dest );
```

The big difference in syntax between this instruction and the `mov` instruction is the fact that the destination operand must be larger than the source operand. That is, if the source operand is a byte, the destination operand must be a word or a double word. Likewise, if the source operand is a word, the destination operand must be a double word. Another difference is that the destination operand has to be a register; the source operand, however, can be a memory location.[3] The `movsx` instruction does not allow constant operands.

To zero extend a value, you can use the `movzx` instruction. It has the same syntax and restrictions as the `movsx` instruction. Zero extending certain 8-bit registers (AL, BL, CL, and DL) into their corresponding 16-bit registers is easily accomplished without using `movzx` by loading the complementary H.O. register (AH, BH, CH, or DH) with 0. Obviously, to zero extend AX into DX:AX or EAX into EDX:EAX, all you need to do is load DX or EDX with 0.[4]

The sample program in Listing 2-8 demonstrates the use of the sign extension instructions.

```
program signExtension;
#include( "stdlib.hhf" )

static
    i8:     int8;
    i16:    int16;
    i32:    int32;
```

[3] This doesn't turn out to be much of a limitation because sign extension almost always precedes an arithmetic operation that must take place in a register.

[4] Zero extending into DX:AX or EDX:EAX is just as necessary as the CWD and CDQ instructions, as you will eventually see.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000024

```
begin signExtension;

    stdout.put( "Enter a small negative number: " );
    stdin.get( i8 );

    stdout.put( nl, "Sign extension using CBW and CWDE:", nl, nl );

    mov( i8, al );
    stdout.put( "You entered ", i8, " ($", al, ")", nl );

    cbw();
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    cwde();
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

    stdout.put( nl, "Sign extension using MOVSX:", nl, nl );

    movsx( i8, ax );
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    movsx( i8, eax );
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

end signExtension;
```

Listing 2-8: Sign extension instructions

Sign *contraction*, converting a value with some number of bits to the identical value with a fewer number of bits, is a little more troublesome. Sign extension never fails. Given an m-bit signed value, you can always convert it to an n-bit number (where $n > m$) using sign extension. Unfortunately, given an n-bit number, you cannot always convert it to an m-bit number if $m < n$. For example, consider the value −448. As a 16-bit signed number, its hexadecimal representation is $FE40. Unfortunately, the magnitude of this number is too large for an 8-bit value, so you cannot sign contract it to 8 bits. This is an example of an overflow condition that occurs upon conversion.

To properly sign contract a value, you must look at the H.O. byte(s) that you want to discard. The H.O. bytes must all contain either 0 or $FF. If you encounter any other values, you cannot contract it without overflow. Finally, the H.O. bit of your resulting value must match *every* bit you've removed from the number. Here are some examples (16 bits to 8 bits):

```
$FF80 can be sign contracted to $80.
$0040 can be sign contracted to $40.
$FE40 cannot be sign contracted to 8 bits.
$0100 cannot be sign contracted to 8 bits.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000025

Another way to reduce the size of an integer is by *saturation*. Saturation is useful in situations where you must convert a larger object to a smaller object, and you're willing to live with possible loss of precision. To convert a value via saturation you simply copy the larger value to the smaller value if it is not outside the range of the smaller object. If the larger value is outside the range of the smaller value, then you *clip* the value by setting it to the largest (or smallest) value within the range of the smaller object.

For example, when converting a 16-bit signed integer to an 8-bit signed integer, if the 16-bit value is in the range −128..+127, you simply copy the L.O. byte of the 16-bit object to the 8-bit object. If the 16-bit signed value is greater than +127, then you clip the value to +127 and store +127 into the 8-bit object. Likewise, if the value is less than −128, you clip the final 8-bit object to −128. Saturation works the same way when clipping 32-bit values to smaller values. If the larger value is outside the range of the smaller value, then you simply set the smaller value to the value closest to the out-of-range value that you can represent with the smaller value.

Obviously, if the larger value is outside the range of the smaller value, then there will be a loss of precision during the conversion. While clipping the value to the limits the smaller object imposes is never desirable, sometimes this is acceptable because the alternative is to raise an exception or otherwise reject the calculation. For many applications, such as audio or video processing, the clipped result is still recognizable, so this is a reasonable conversion.

2.10 Shifts and Rotates

Another set of logical operations that apply to bit strings is the *shift* and *rotate* operations. These two categories can be further broken down into *left shifts, left rotates, right shifts,* and *right rotates*. These operations turn out to be extremely useful.

The left-shift operation moves each bit in a bit string one position to the left (Figure 2-8 provides an example of an 8-bit shift).

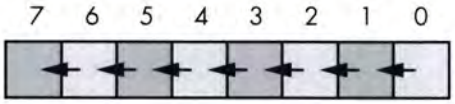


Figure 2-8: Shift-left operation

Bit 0 moves into bit position 1, the previous value in bit position 1 moves into bit position 2, and so on. There are, of course, two questions that naturally arise: "What goes into bit 0?" and "Where does the high-order bit go?" We'll shift a 0 into bit 0, and the previous value of the high-order bit will become the *carry* out of this operation.

The 80x86 provides a shift-left instruction, `shl`, that performs this useful operation. The syntax for the `shl` instruction is:

```
shl( count, dest );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000026

The *count* operand is either CL or a constant in the range 0..*n*, where *n* is one less than the number of bits in the destination operand (for example, $n = 7$ for 8-bit operands, $n = 15$ for 16-bit operands, and $n = 31$ for 32-bit operands). The *dest* operand is a typical destination operand. It can be either a memory location or a register.

When the *count* operand is the constant 1, the `shl` instruction does the operation shown in Figure 2-9.

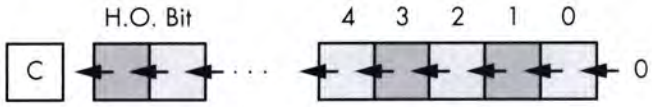


Figure 2-9: Shift-left operation

In Figure 2-9, the *C* represents the carry flag. That is, the H.O. bit shifted out of the operand moves into the carry flag. Therefore, you can test for overflow after a `shl( 1, dest );` instruction by testing the carry flag immediately after executing the instruction (e.g., by using `if( @c ) then...` or `if( @nc ) then...`).

Intel's literature suggests that the state of the carry flag is undefined if the shift count is a value other than 1. Usually, the carry flag contains the last bit shifted out of the destination operand, but Intel doesn't seem to guarantee this.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a 0 to the right of the number) effectively multiplies it by 10 (the radix):

```
1234 shl 1 = 12340
```

(`shl 1` means shift one digit position to the left.)

Because the radix of a binary number is 2, shifting it left multiplies it by 2. If you shift a binary value to the left twice, you multiply it by 2 twice (that is, you multiply it by 4). If you shift a binary value to the left three times, you multiply it by 8 (2*2*2). In general, if you shift a value to the left *n* times, you multiply that value by 2^n.

A right-shift operation works the same way, except we're moving the data in the opposite direction. For a byte value, bit 7 moves into bit 6, bit 6 moves into bit 5, bit 5 moves into bit 4, and so on. During a right shift, we'll move a 0 into bit 7, and bit 0 will be the carry out of the operation (see Figure 2-10).

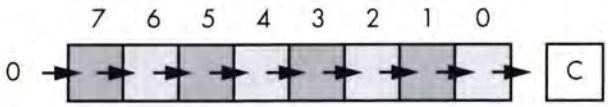


Figure 2-10: Shift-right operation

As you would probably expect, the 80x86 provides a `shr` instruction that will shift the bits to the right in a destination operand. The syntax is the same as the `shl` instruction except, of course, you specify `shr` rather than `shl`:

```
shr( count, dest );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000027

This instruction shifts a 0 into the H.O. bit of the destination operand, it shifts the other bits one place to the right (that is, from a higher bit number to a lower bit number). Finally, bit 0 is shifted into the carry flag. If you specify a count of 1, the shr instruction does the operation shown in Figure 2-11.

Figure 2-11: Shift-right operation

Once again, Intel's documents suggest that shifts of more than 1 bit leave the carry in an undefined state.

Because a left shift is equivalent to a multiplication by 2, it should come as no surprise that a right shift is roughly comparable to a division by 2 (or, in general, a division by the radix of the number). If you perform n right shifts, you will divide that number by 2^n.

There is one problem with shift rights with respect to division: A shift right is only equivalent to an *unsigned* division by 2. For example, if you shift the unsigned representation of 254 ($FE) one place to the right, you get 127 ($7F), exactly what you would expect. However, if you shift the binary representation of −2 ($FE) to the right one position, you get 127 ($7F), which is *not* correct. This problem occurs because we're shifting a 0 into bit 7. If bit 7 previously contained a 1, we're changing it from a negative to a positive number. Not a good thing to do when dividing by 2.

To use the shift right as a division operator, we must define a third shift operation: arithmetic shift right.[5] An arithmetic shift right works just like the normal shift-right operation (a logical shift right) with one exception: Instead of shifting a 0 into the high-order bit, an arithmetic shift-right operation copies the high-order bit back into itself; that is, during the shift operation it does not modify the high-order bit, as Figure 2-12 shows.

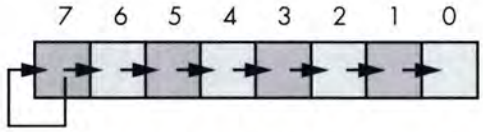


Figure 2-12: Arithmetic shift-right operation

An arithmetic shift right generally produces the result you expect. For example, if you perform the arithmetic shift-right operation on −2 ($FE), you get −1 ($FF). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer that is *less than or equal to the actual result.* Based on experiences with high-level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates toward 0. But this simply isn't the case. For example, if you apply the arithmetic shift-right operation on −1 ($FF), the result is −1, not 0. Because −1 is less than 0,

[5] There is no need for an arithmetic shift left. The standard shift-left operation works for both signed and unsigned numbers, assuming no overflow occurs.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000028

the arithmetic shift-right operation rounds toward −1. This is not a bug in the arithmetic shift-right operation; it just uses a different (though valid) definition of integer division.

The 80x86 provides an arithmetic shift-right instruction, `sar` (shift arithmetic right). This instruction's syntax is nearly identical to `shl` and `shr`. The syntax is:

```
sar( count, dest );
```

The usual limitations on the count and destination operands apply. This instruction operates as shown in Figure 2-13 if the count is 1.

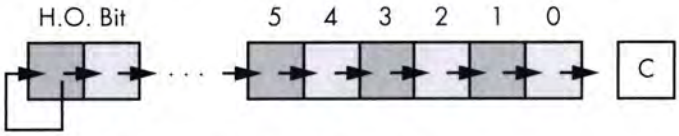


Figure 2-13: `sar( 1, dest )` operation

Once again, Intel's documents suggest that shifts of more than 1 bit leave the carry in an undefined state.

Another pair of useful operations are *rotate left* and *rotate right*. These operations behave like the shift-left and shift-right operations with one major difference: The bit shifted out from one end is shifted back in at the other end. Figure 2-14 diagrams these operations.

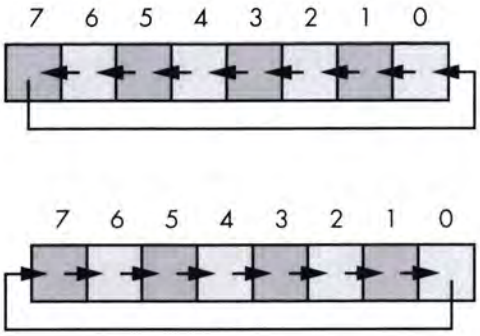


Figure 2-14: Rotate-left and rotate-right operations

The 80x86 provides `rol` (rotate left) and `ror` (rotate right) instructions that do these basic operations on their operands. The syntax for these two instructions is similar to the shift instructions:

```
rol( count, dest );
ror( count, dest );
```

Once again, these instructions provide a special behavior if the shift count is 1. Under this condition these two instructions also copy the bit shifted out of the destination operand into the carry flag as Figures 2-15 and 2-16 show.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000029

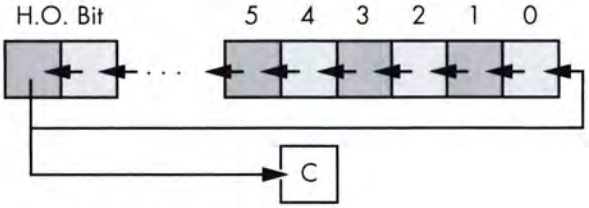


Figure 2-15: `rol( 1, dest )` operation

Note that Intel's documents suggest that rotates of more than 1 bit leave the carry in an undefined state.

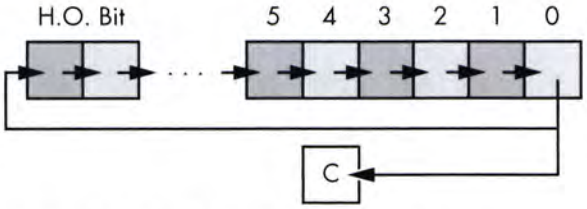


Figure 2-16: `ror( 1, dest )` operation

It is often more convenient for the rotate operation to shift the output bit through the carry and shift the previous carry value back into the input bit of the shift operation. The 80x86 `rcl` (rotate through carry left) and `rcr` (rotate through carry right) instructions achieve this for you. These instructions use the following syntax:

```
rcl( count, dest );
rcr( count, dest );
```

As is true for the other shift and rotate instructions, the *`count`* operand is either a constant or the CL register, and the *`dest`* operand is a memory location or register. The *`count`* operand must be a value that is less than the number of bits in the *`dest`* operand. For a count value of 1, these two instructions do the rotation shown in Figure 2-17.

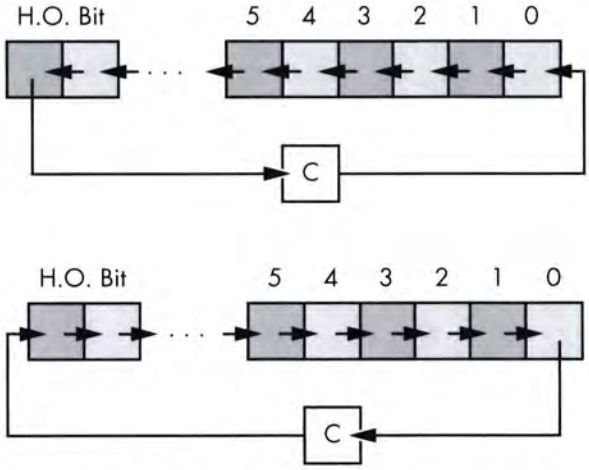


Figure 2-17: `rcl( 1, dest )` and `rcr( 1, dest )` operations

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000030

Again, Intel's documents suggest that rotates of more than 1 bit leave the carry in an undefined state.

2.11 Bit Fields and Packed Data

Although the 80x86 operates most efficiently on `byte`, `word`, and `dword` data types, occasionally you'll need to work with a data type that uses some number of bits other than 8, 16, or 32. For example, consider a date of the form 04/02/01. It takes three numeric values to represent this date: month, day, and year values. Months, of course, take on the values 1..12. It will require at least 4 bits (maximum of 16 different values) to represent the month. Days range between 1 and 31. So it will take 5 bits (maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires 7 bits (that can be used to represent up to 128 different values). 4 + 5 + 7 = 16 bits, or 2 bytes. In other words, we can pack our date data into 2 bytes rather than the 3 that would be required if we used a separate byte for each of the month, day, and year values. This saves 1 byte of memory for each date stored, which could be a substantial saving if you need to store many dates. The bits could be arranged as shown in Figure 2-18.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	M	M	M	D	D	D	D	D	Y	Y	Y	Y	Y	Y	Y

Figure 2-18: Short packed date format (2 bytes)

MMMM represents the 4 bits making up the month value, DDDDD represents the 5 bits making up the day, and YYYYYYY is the 7 bits composing the year. Each collection of bits representing a data item is a *bit field.* For example, April 2, 2001, would be represented as $4101:

```
        0100      00010   0000001          = %0100_0001_0000_0001 or $4101
         4          2       01
```

Although packed values are *space efficient* (that is, very efficient in terms of memory usage), they are computationally *inefficient* (slow!). The reason? It takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything. The sample program in Listing 2-9 demonstrates the effort that must go into packing and unpacking this 16-bit date format.

```
program dateDemo;

#include( "stdlib.hhf" )

static
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000031

```
    day:        uns8;
    month:      uns8;
    year:       uns8;

    packedDate: word;

begin dateDemo;

    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    //  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    //   m  m  m  m  d  d  d  d  d  y  y  y  y  y  y  y

    mov( 0, ax );
    mov( ax, packedDate );  // Just in case there is an error.
    if( month > 12 ) then

        stdout.put( "Month value is too large", nl );

    elseif( month = 0 ) then

        stdout.put( "Month value must be in the range 1..12", nl );

    elseif( day > 31 ) then

        stdout.put( "Day value is too large", nl );

    elseif( day = 0 ) then

        stdout.put( "Day value must be in the range 1..31", nl );

    elseif( year > 99 ) then

        stdout.put( "Year value must be in the range 0..99", nl );

    else

        mov( month, al );
        shl( 5, ax );
        or( day, al );
        shl( 7, ax );
        or( year, al );
        mov( ax, packedDate );

    endif;

    // Okay, display the packed value:

    stdout.put( "Packed data = $", packedDate, nl );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000032

```
    // Unpack the date:

    mov( packedDate, ax );
    and( $7f, al );         // Retrieve the year value.
    mov( al, year );

    mov( packedDate, ax );  // Retrieve the day value.
    shr( 7, ax );
    and( %1_1111, al );
    mov( al, day );

    mov( packedDate, ax );  // Retrieve the month value.
    rol( 4, ax );
    and( %1111, al );
    mov( al, month );

    stdout.put( "The date is ", month, "/", day, "/", year, nl );


end dateDemo;
```

Listing 2-9: Packing and unpacking date data

Of course, having gone through the problems with Y2K (Year 2000), you know that using a date format that limits you to 100 years (or even 127 years) would be quite foolish at this time. If you are concerned about your software running 100 years from now, perhaps it would be wise to use a 3-byte date format rather than a 2-byte format. As you will see in the chapter on arrays, however, you should always try to create data objects whose length is an even power of 2 (1 byte, 2 bytes, 4 bytes, 8 bytes, and so on) or you will pay a performance penalty. Hence, it is probably wise to go ahead and use 4 bytes and pack this data into a double-word variable. Figure 2-19 shows one possible data organization for a 4-byte date.

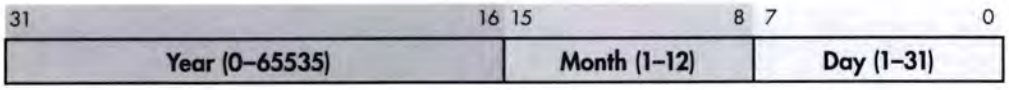


Figure 2-19: Long packed date format (4 bytes)

In this long packed date format we made several changes beyond simply extending the number of bits associated with the year. First, because there are extra bits in a 32-bit double-word variable, this format allocates extra bits to the month and day fields. Because these two fields now consist of 8 bits each, they can be easily extracted as a byte object from the double word. This leaves fewer bits for the year, but 65,536 years is probably sufficient; you can probably assume without too much concern that your software will not still be in use 63,000 years from now when this date format will no longer work.

Of course, you could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into 1 byte each and one that should probably have at least 2 bytes. Because this "packed" date format consumes the same 4 bytes as the unpacked version,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000033

what is so special about this format? Well, another difference you will note between this long packed date format and the short date format appearing in Figure 2-18 is the fact that this long date format rearranges the bits so the `Year` field is in the H.O. bit positions, the `Month` field is in the middle bit positions, and the `Day` field is in the L.O. bit positions. This is important because it allows you to very easily compare two dates to see if one date is less than, equal to, or greater than another date. Consider the following code:

```
mov( Date1, eax );          // Assume Date1 and Date2 are dword variables
if( eax > Date2 ) then      // using the Long Packed Date format.

    << Do something if Date1 > Date2 >>

endif;
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare *Date1* and *Date2* in such an easy fashion. Therefore, this example demonstrates another reason for packing data even if you don't realize any space savings—it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Examples of practical packed data types abound. You could pack eight boolean values into a single byte, you could pack two BCD digits into a byte, and so on. Of course, a classic example of packed data is the EFLAGS register (see Figure 2-20). This register packs nine important boolean objects (along with seven important system flags) into a single 16-bit register. You will commonly need to access many of these flags. For this reason, the 80x86 instruction set provides many ways to manipulate the individual bits in the EFLAGS register. Of course, you can test many of the condition code flags using the HLA pseudo-boolean variables such as `@c`, `@nc`, `@z`, and `@nz` in an `if` statement or other statement using a boolean expression.

In addition to the condition codes, the 80x86 provides instructions that directly affect certain flags (Table 2-7).

Table 2-7: Instructions That Affect Certain Flags

Instruction	Explanation
`cld();`	Clears (sets to 0) the direction flag.
`std();`	Sets (to 1) the direction flag.
`cli();`	Clears the interrupt disable flag.
`sti();`	Sets the interrupt disable flag.
`clc();`	Clears the carry flag.
`stc();`	Sets the carry flag.
`cmc();`	Complements (inverts) the carry flag.
`sahf();`	Stores the AH register into the L.O. 8 bits of the EFLAGS register.
`lahf();`	Loads AH from the L.O. 8 bits of the EFLAGS register.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000034

There are other instructions that affect the EFLAGS register as well; these instructions, however, demonstrate how to access several of the packed boolean values in the EFLAGS register. The `lahf` and `sahf` instructions, in particular, provide a convenient way to access the L.O. 8 bits of the EFLAGS register as an 8-bit byte (rather than as eight separate 1-bit values). See Figure 2-20 for a layout of the EFLAGS register.

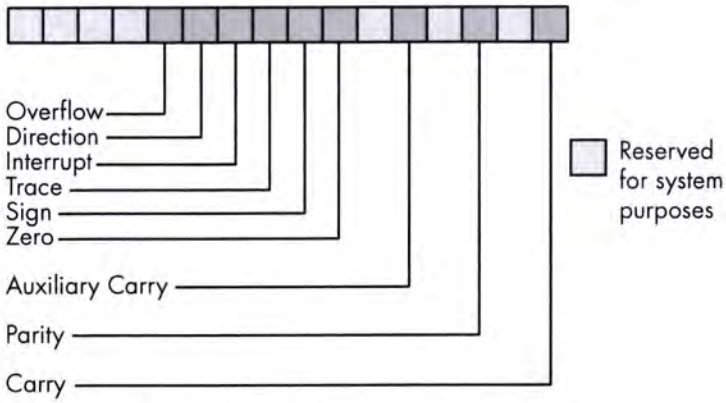


Figure 2-20: EFLAGS register as packed boolean data

The `lahf` (load AH with the L.O. 8 bits of the EFLAGS register) and the `sahf` (store AH into the L.O. byte of the EFLAGS register) use the following syntax:

```
lahf();
sahf();
```

2.12 An Introduction to Floating-Point Arithmetic

Integer arithmetic does not let you represent fractional numeric values. Therefore, modern CPUs support an approximation of *real* arithmetic: floating-point arithmetic. A big problem with floating-point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating-point arithmetic. This is a source of defects in many programs. One of the primary goals of this section is to describe the limitations of floating-point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinite precision* arithmetic. Consider the simple statement $x := x + 1$, where x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur.* That is, this statement is valid only for certain values of x ($minint <= x < maxint$). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., 5/2 does not equal 2.5).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating-point values suffer from this same problem, only worse. After

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000035

all, the integers are a subset of the real numbers. Therefore, the floating-point values must represent the same infinite set of integers. However, there are an infinite number of real values between any two integer values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges either.

To represent real numbers, most floating-point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating-point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating-point arithmetic operates. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating-point format for our examples. Our floating-point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values, as shown in Figure 2-21.

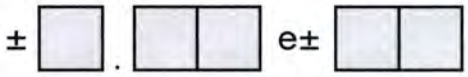


Figure 2-21: A floating-point format

When adding and subtracting two numbers in scientific notation, we must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, we must adjust the values so they have the same exponent. One way to do this is to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating-point calculation had been limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1, a value that is even less accurate. To improve the accuracy of floating-point calculations, it is necessary to add extra digits for use during the calculation. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value that is the result of a sequence of floating-point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 to 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000036

maintain only three significant digits, so adding in a small value shouldn't affect the result at all. However, suppose we were to add 1.00e0 to 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth . . . and tenth times we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself 10 times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is an important thing to know about limited-precision arithmetic:

The order of evaluation can affect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another when adding and subtracting floating-point values. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 − 1.22e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e − 2, this latter form suggests that the last two digits are exactly 0. Unfortunately, we have only a single significant digit at this time. Indeed, some floating-point unit (FPU) software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

When subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating-point format.

Multiplication and division do not suffer from the same problems as addition and subtraction because you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error that already exists in a value. For example, if you multiply 1.23e0 by 2, when you should be multiplying 1.24e0 by 2, the result is even less accurate. This brings up a third important rule when working with limited-precision arithmetic:

When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute $x * (y + z)$. Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform $x * (y + z)$ to get $x * y + x * z$ and compute the result by performing the multiplications first.[6]

[6] Of course, the drawback is that you must now perform two multiplications rather than one, so the result may be slower.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000037

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

Comparing floating-point numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating-point value), you should *never* compare two floating-point values to see if they are equal. In a binary floating-point format, different computations that produce the same (mathematical) result may differ in their least significant bits. For example, 1.31e0 + 1.69e0 should produce 3.00e0. Likewise, 1.50e0 + 1.50e0 should produce 3.00e0. However, if you were to compare (1.31e0 + 1.69e0) against (1.50e0 + 1.50e0), you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Because this is not necessarily true after two different floating-point computations that should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating-point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The usual way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then ...
```

Another common way to handle this same comparison is to use a statement of the form

```
if abs(Value1-Value2) <= error then ...
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error that will creep into your computations. The exact value will depend upon the particular floating-point format you use, but more on that a little later. Here is the final rule we will state in this section:

When comparing two floating-point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.

There are many other little problems that can occur when using floating-point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating-point arithmetic like real arithmetic—the inaccuracies present in limited-precision arithmetic can get you into trouble if you are not careful. A

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000038

good text on numerical analysis or even scientific computing can help fill in the details that are beyond the scope of this text. If you are going to be working with floating-point arithmetic, *in any language*, you should take the time to study the effects of limited-precision arithmetic on your computations.

HLA's `if` statement does not support boolean expressions involving floating-point operands. Therefore, you cannot use statements like `if( x < 3.141) then...` in your programs. Chapter 6 will teach you how to do floating-point comparisons.

2.12.1 IEEE Floating-Point Formats

When Intel planned to introduce a floating-point unit for its new 8086 microprocessor, it was smart enough to realize that the electrical engineers and solid-state physicists who design chips were probably not the best people to pick the best possible binary representation for a floating-point format. So Intel went out and hired the best numerical analyst it could find to design a floating-point format for its 8087 FPU. That person then hired two other experts in the field, and the three of them (Kahn, Coonan, and Stone) designed Intel's floating-point format. They did such a good job designing the KCS Floating-Point Standard that the IEEE organization adopted this format for the IEEE floating-point format.[7]

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating-point formats: single-precision, double-precision, and extended-precision. The single- and double-precision formats corresponded to C's float and double types or FORTRAN's real and double-precision types. Intel intended to use extended-precision for long chains of computations. Extended-precision contains 16 extra bits that the calculations could use as guard bits before rounding down to a double-precision value when storing the result.

The single-precision format uses a *one's complement 24-bit mantissa* and an *8-bit excess-127 exponent*. The mantissa usually represents a value from 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be 1 and represents a value just to the left of the *binary point*.[8] The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value

```
1.mmmmmmmm mmmmmmmmm mmmmmmmmm
```

The `mmmm` characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (0 or 1) times a successive negative power of 2. The implied 1 bit is always multiplied by 2^0, which is 1. This is why the mantissa is always greater than or equal to 1. Even if the other

[7] There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.

[8] The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000039

mantissa bits are all 0, the implied 1 bit always gives us the value 1[9]. Of course, even if we had an almost infinite number of 1 bits after the binary point, they still would not add up to 2. This is why the mantissa can represent values in the range 1 to just under 2.

Although there are an infinite number of values between 1 and 2, we can only represent 8 million of them because we use a 23-bit mantissa (the 24th bit is always 1). This is the reason for inaccuracy in floating-point arithmetic—we are limited to 23 bits of precision in computations involving single-precision floating-point values.

The mantissa uses a one's complement format rather than two's complement. This means that the 24-bit value of the mantissa is simply an unsigned binary number, and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for 0 (with the sign bit set or clear). Generally, this is important only to the person designing the floating-point software or hardware system. We will assume that the value 0 always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating-point format comes into play. The floating-point format raises 2 to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is 8 bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 ($7F). Therefore, to convert an exponent to excess-127 format, simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating-point values. The single-precision floating-point format takes the form shown in Figure 2-22.

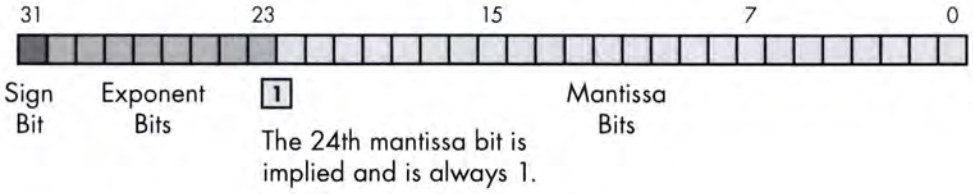


Figure 2-22: Single-precision (32-bit) floating-point format

With a 24-bit mantissa, you will get approximately 6 ½ digits of precision (½ digit of precision means that the first six digits can all be in the range 0..9, but the seventh digit can only be in the range 0..x, where $x < 9$ and is generally close to 5). With an 8-bit excess-127 exponent, the dynamic range of single-precision floating-point numbers is approximately 2 ± 128 or about 10 ± 38.

Although single-precision floating-point numbers are perfectly suitable for many applications, the dynamic range is somewhat limited and is unsuitable for many financial, scientific, and other applications. Furthermore, during long chains of computations, the limited accuracy of the single-precision format may introduce serious error.

[9] Actually, this isn't necessarily true. The IEEE floating-point format supports *denormalized* values where the H.O. bit is not 0. However, we will ignore denormalized values in our discussion.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000040

The double-precision format helps overcome the problems of single-precision floating-point. Using twice the space, the double-precision format has an 11-bit excess-1023 exponent and a 53-bit mantissa (with an implied H.O. bit of 1) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$ and 14 ½ digits of precision, sufficient for most applications. Double-precision floating-point values take the form shown in Figure 2-23.

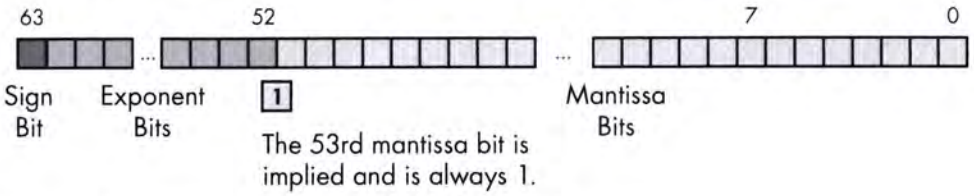


Figure 2-23: 64-bit double-precision floating-point format

In order to help ensure accuracy during long chains of computations involving double-precision floating-point numbers, Intel designed the extended-precision format. The extended-precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa and four of the additional bits are appended to the end of the exponent. Unlike the single- and double-precision values, the extended-precision format's mantissa does not have an implied H.O. bit, which is always 1. Therefore, the extended-precision format provides a 64-bit mantissa, a 15-bit excess-16383 exponent, and a 1-bit sign. The format for the extended-precision floating-point value is shown in Figure 2-24.

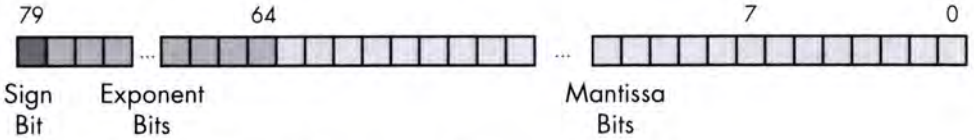


Figure 2-24: 80-bit extended-precision floating-point format

On the FPUs all computations are done using the extended-precision format. Whenever you load a single or double-precision value, the FPU automatically converts it to an extended-precision value. Likewise, when you store a single or double-precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended-precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating-point value is one whose H.O. mantissa bit contains 1. Almost any nonnormalized value can be normalized; shift the mantissa bits to the left and decrement the exponent until a 1 appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating-point value by 2. Likewise, whenever you decrement the exponent, you divide the floating-point value by 2. By the same token, shifting the mantissa to the left one bit position multiplies the floating-point value by 2; likewise, shifting the mantissa

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000041

to the right divides the floating-point value by 2. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating-point number at all.

Keeping floating-point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all 0, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating-point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating-point number cannot be normalized. Zero is one of these special cases. Obviously it cannot be normalized because the floating-point representation for 0 has no 1 bits in the mantissa. This, however, is not a problem because we can exactly represent the value 0 with only a single bit.

The second case is when we have some H.O. bits in the mantissa that are 0 but the biased exponent is also 0 (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are 0 (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values.[10] Although the use of denormalized values allows IEEE floating-point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision.

2.12.2 HLA Support for Floating-Point Values

HLA provides several data types and library routines to support the use of floating-point data in your assembly language programs. These include built-in types to declare floating-point variables as well as routines that provide floating-point input, output, and conversion.

Perhaps the best place to start when discussing HLA's floating-point facilities is with a description of floating-point literal constants. HLA floating-point constants allow the following syntax:

- An optional + or - symbol, denoting the sign of the mantissa (if this is not present, HLA assumes that the mantissa is positive)
- Followed by one or more decimal digits
- Optionally followed by a decimal point and one or more decimal digits
- Optionally followed by an e or E, optionally followed by a sign (+ or -) and one or more decimal digits

Note that the decimal point or the e/E must be present in order to differentiate this value from an integer or unsigned literal constant. Here are some examples of legal literal floating-point constants:

```
1.234  3.75e2  -1.0  1.1e-1  1e+4  0.1  -123.456e+789  +25e0
```

[10] The alternative would be to underflow the values to 0.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000042

Notice that a floating-point literal constant cannot begin with a decimal point; it must begin with a decimal digit, so you must use 0.1 to represent .1 in your programs.

HLA also allows you to place an underscore character (_) between any two consecutive decimal digits in a floating-point literal constant. You may use the underscore character in place of a comma (or other language-specific separator character) to help make your large floating-point numbers easier to read. Here are some examples:

```
1_234_837.25  1_000.00  789_934.99  9_999.99
```

To declare a floating-point variable you use the `real32`, `real64`, or `real80` data types. Like their integer and unsigned brethren, the number at the end of these data type declarations specifies the number of bits used for each type's binary representation. Therefore, you use `real32` to declare single-precision real values, `real64` to declare double-precision floating-point values, and `real80` to declare extended-precision floating-point values. Other than the fact that you use these types to declare floating-point variables rather than integers, their use is nearly identical to that for `int8`, `int16`, `int32`, and so on. The following examples demonstrate these declarations and their syntax:

```
static

        fltVar1:      real32;
        fltVar1a:     real32 := 2.7;
        pi:           real32 := 3.14159;
        DblVar:       real64;
        DblVar2:      real64 := 1.23456789e+10;
        XPVar:        real80;
        XPVar2:       real80 := -1.0e-104;
```

To output a floating-point variable in ASCII form, you would use one of the `stdout.putr32`, `stdout.putr64`, or `stdout.putr80` routines. These procedures display a number in decimal notation, that is, a string of digits, an optional decimal point, and a closing string of digits. Other than their names, these three routines use exactly the same calling sequence. Here are the calls and parameters for each of these routines:

```
stdout.putr80( r:real80; width:uns32; decpts:uns32 );
stdout.putr64( r:real64; width:uns32; decpts:uns32 );
stdout.putr32( r:real32; width:uns32; decpts:uns32 );
```

The first parameter to these procedures is the floating-point value you wish to print. The size of this parameter must match the procedure's name (e.g., the `r` parameter must be an 80-bit extended-precision floating-point variable when calling the `stdout.putr80` routine). The second parameter specifies the field width for the output text; this is the number of print

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000043

positions the number will require when the procedure displays it. Note that this width must include print positions for the sign of the number and the decimal point. The third parameter specifies the number of print positions after the decimal point. For example:

```
stdout.putr32( pi, 10, 4 );
```

displays the value

```
_ _ _ _ 3.1416
```

(underscores represent leading spaces in this example).

Of course, if the number is very large or very small, you will want to use scientific notation rather than decimal notation for your floating-point numeric output. The HLA Standard Library `stdout.pute32`, `stdout.pute64`, and `stdout.pute80` routines provide this facility. These routines use the following procedure prototypes:

```
stdout.pute80( r:real80; width:uns32 );
stdout.pute64( r:real64; width:uns32 );
stdout.pute32( r:real32; width:uns32 );
```

Unlike the decimal output routines, these scientific notation output routines do not require a third parameter specifying the number of digits after the decimal point to display. The `width` parameter indirectly specifies this value because all but one of the mantissa digits always appear to the right of the decimal point. These routines output their values in decimal notation, similar to the following:

```
1.23456789e+10  -1.0e-104  1e+2
```

You can also output floating-point values using the HLA Standard Library `stdout.put` routine. If you specify the name of a floating-point variable in the `stdout.put` parameter list, the `stdout.put` code will output the value using scientific notation. The actual field width varies depending on the size of the floating-point variable (the `stdout.put` routine attempts to output as many significant digits as possible, in this case). Here's an example:

```
stdout.put( "XPVar2 = ", XPVar2 );
```

If you specify a field width, by using a colon followed by a signed integer value, then the `stdout.put` routine will use the appropriate `stdout.puteXX` routine to display the value. That is, the number will still appear in scientific notation, but you get to control the field width of the output value. Like the field width for integer and unsigned values, a positive field width right justifies the number in the specified field, and a negative number left justifies the value.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000044

Here is an example that prints the XPVar2 variable using 10 print positions:

```
stdout.put( "XPVar2 = ", XPVar2:10 );
```

If you wish to use stdout.put to print a floating-point value in decimal notation, you need to use the following syntax:

```
Variable_Name : Width : DecPts
```

Note that the *DecPts* field must be a nonnegative integer value.

When stdout.put contains a parameter of this form, it calls the corresponding stdout.putr*XX* routine to display the specified floating-point value. As an example, consider the following call:

```
stdout.put( "Pi = ", pi:5:3 );
```

The corresponding output is:

```
3.142
```

The HLA Standard Library provides several other useful routines you can use when outputting floating-point values. Consult the HLA Standard Library reference manual for more information on these routines.

The HLA Standard Library provides several routines to let you display floating-point values in a wide variety of formats. In contrast, the HLA Standard Library provides only two routines to support floating-point input: stdin.getf() and stdin.get(). The stdin.getf() routine requires the use of the 80x86 FPU stack, a hardware component that this chapter doesn't cover. Therefore, we'll defer the discussion of the stdin.getf() routine until Chapter 6. Because the stdin.get() routine provides all the capabilities of the stdin.getf() routine, this deferral will not be a problem.

You've already seen the syntax for the stdin.get() routine; its parameter list simply contains a list of variable names. The stdin.get() function reads appropriate values for the user for each of the variables appearing in the parameter list. If you specify the name of a floating-point variable, the stdin.get() routine automatically reads a floating-point value from the user and stores the result into the specified variable. The following example demonstrates the use of this routine:

```
stdout.put( "Input a double-precision floating-point value: " );
stdin.get( DblVar );
```

WARNING *This section discussed how you would declare floating-point variables and how you would input and output them. It did not discuss arithmetic. Floating-point arithmetic is different from integer arithmetic; you cannot use the 80x86* add *and* sub *instructions to operate on floating-point values. Floating-point arithmetic will be the subject of Chapter 6.*

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000045

2.13 Binary-Coded Decimal Representation

Although the integer and floating-point formats cover most of the numeric needs of an average program, there are some special cases where other numeric representations are convenient. In this section we'll discuss the binary-coded decimal format because the 80x86 CPU provides a small amount of hardware support for this data representation.

BCD values are a sequence of nibbles, with each nibble representing a value in the range 0..9. Of course you can represent values in the range 0..15 using a nibble; the BCD format, however, uses only 10 of the possible 16 different values for each nibble.

Each nibble in a BCD value represents a single decimal digit. Therefore, with a single byte (i.e., two digits) we can represent values containing two decimal digits, or values in the range 0..99 (see Figure 2-25). With a word, we can represent values having four decimal digits, or values in the range 0..9,999. Likewise, with a double word we can represent values with up to eight decimal digits (because there are eight nibbles in a double-word value).

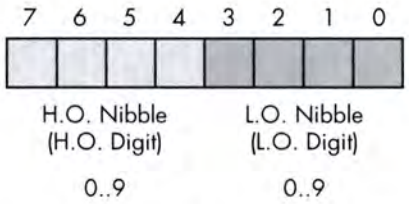


Figure 2-25: CD data representation in memory

As you can see, BCD storage isn't particularly memory efficient. For example, an 8-bit BCD variable can represent values in the range 0..99 while that same 8 bits, when holding a binary value, can represent values in the range 0..255. Likewise, a 16-bit binary value can represent values in the range 0..65,535, while a 16-bit BCD value can represent only about one-sixth of those values (0..9,999). Inefficient storage isn't the only problem. BCD calculations tend to be slower than binary calculations.

At this point, you're probably wondering why anyone would ever use the BCD format. The BCD format does have two saving graces: It's very easy to convert BCD values between the internal numeric representation and their string representation; also, it's very easy to encode multidigit decimal values in hardware (e.g., using a thumb wheel or dial) using BCD. For these two reasons, you're likely to see people using BCD in embedded systems (such as toaster ovens, alarm clocks, and nuclear reactors) but rarely in general-purpose computer software.

A few decades ago people mistakenly thought that calculations involving BCD (or just decimal) arithmetic were more accurate than binary calculations. Therefore, they would often perform important calculations, like those involving dollars and cents (or other monetary units) using decimal-based arithmetic. While it is true that certain calculations can produce more accurate results in BCD, this statement is not true in general. Indeed, for most calculations (even those involving fixed-point decimal arithmetic), the

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000046

binary representation is more accurate. For this reason, most modern computer programs represent all values in a binary form. For example, the Intel 80x86 floating-point unit supports a pair of instructions for loading and storing BCD values. Internally, however, the FPU converts these BCD values to binary and performs all calculations in binary. It uses BCD only as an external data format (external to the FPU, that is). This generally produces more accurate results and requires far less silicon than having a separate coprocessor that supports decimal arithmetic.

2.14 Characters

Perhaps the most important data type on a personal computer is the character data type. The term *character* refers to a human or machine-readable symbol that is typically a nonnumeric entity. In general, the term *character* refers to any symbol that you can normally type on a keyboard (including some symbols that may require multiple key presses to produce) or display on a video display. Many beginners often confuse the terms *character* and *alphabetic character*. These terms are not the same. Punctuation symbols, numeric digits, spaces, tabs, carriage returns (enter), other control characters, and other special symbols are also characters. When this text uses the term *character* it refers to any of these characters, not just the alphabetic characters. When this text refers to alphabetic characters, it will use phrases like "alphabetic characters," "uppercase characters," or "lowercase characters."

Another common problem beginners have when they first encounter the character data type is differentiating between numeric characters and numbers. The character `1` is different from the value 1. The computer (generally) uses two different internal representations for numeric characters (`0`, `1`, ..., `9`) versus the numeric values 0..9. You must take care not to confuse the two.

Most computer systems use a 1- or 2-byte sequence to encode the various characters in binary form. Windows, Mac OS X, FreeBSD, and Linux certainly fall into this category, using either the ASCII or Unicode encodings for characters. This section will discuss the ASCII character set and the character declaration facilities that HLA provides.

2.14.1 The ASCII Character Encoding

The ASCII (American Standard Code for Information Interchange) character set maps 128 textual characters to the unsigned integer values 0..127 ($0..$7F). Internally, of course, the computer represents everything using binary numbers, so it should come as no surprise that the computer also uses binary values to represent nonnumeric entities such as characters. Although the exact mapping of characters to numeric values is arbitrary and unimportant, it is important to use a standardized code for this mapping because you will need to communicate with other programs and peripheral devices and you need to talk the same "language" as these other programs and devices. This is where the ASCII code comes into play; it is a standardized code that nearly everyone has agreed on. Therefore, if you use the

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000047

ASCII code 65 to represent the character 'A', then you know that some peripheral device (such as a printer) will correctly interpret this value as the character 'A' whenever you transmit data to that device.

You should not get the impression that ASCII is the only character set in use on computer systems. IBM uses the EBCDIC character set family on many of its mainframe computer systems. Another common character set in use is the Unicode character set. Unicode is an extension to the ASCII character set that uses 16 bits rather than 7 bits to represent characters. This allows the use of 65,536 different characters in the character set, allowing the inclusion of most symbols in the world's different languages into a single unified character set.

Because the ASCII character set provides only 128 different characters and a byte can represent 256 different values, an interesting question arises: "What do we do with the values 128..255 that one could store into a byte?" One answer is to ignore those extra values. That will be the primary approach of this text. Another possibility is to extend the ASCII character set and add an additional 128 characters to it. Of course, this would tend to defeat the whole purpose of having a standardized character set unless you could get everyone to agree on the extensions. That is a difficult task.

When IBM first created its IBM-PC, it defined these extra 128 character codes to contain various non-English alphabetic characters, some line-drawing graphics characters, some mathematical symbols, and several other special characters. Because IBM's PC was the foundation for what we typically call a PC today, that character set has become a pseudo-standard on all IBM-PC compatible machines. Even on modern machines, which are not IBM-PC compatible and cannot run early PC software, the IBM extended character set survives. Note, however, that this PC character set (an extension of the ASCII character set) is not universal. Most printers will not print the extended characters when using native fonts, and many programs (particularly in non-English-speaking countries) do not use those characters for the upper 128 codes in an 8-bit value. For these reasons, this text will generally stick to the standard 128-character ASCII character set.

Despite the fact that it is a standard, simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it's true that an 'A' on one machine is most likely an 'A' on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported—backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. *End of line* is a particularly troublesome example. Windows, MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Older Apple Macintosh computers (Mac OS 9 and earlier) and many other systems mark the end of a line by a single CR character. Linux, Mac OS X, FreeBSD, and other Unix systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000048

your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Because you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., for `'0'`, `'A'`, `'a'`, etc.).

The ASCII character set is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0..$1F (31), form a special set of nonprinting characters, the *control characters*. We call them control characters because they perform various printer/display control operations rather than display symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters;[11] line feed, which moves the cursor down one line on the output device; and backspace, which moves the cursor back one position to the left. Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes contains various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code $20) and the numeric digits (ASCII codes $30..$39).

The third group of 32 ASCII characters contains the uppercase alphabetic characters. The ASCII codes for the characters `'A'..'Z'` lie in the range $41..$5A (65..90). Because there are only 26 different alphabetic characters, the remaining 6 codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes represents the lowercase alphabetic symbols, 5 additional special symbols, and another control character (delete). Note that the lowercase character symbols use the ASCII codes $61..$7A. If you convert the codes for the upper- and lowercase characters to binary, you will notice that the uppercase symbols differ from their lowercase equivalents in exactly one bit position. For example, consider the character codes for `'E'` and `'e'` appearing in Figure 2-26.

	7	6	5	4	3	2	1	0
E	0	1	0	0	0	1	0	1

	7	6	5	4	3	2	1	0
e	0	1	1	0	0	1	0	1

Figure 2-26: ASCII codes for E and e

[11] Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left-hand side of the paper.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000049

The only place these two codes differ is in bit 5. Uppercase characters always contain a 0 in bit 5; lowercase alphabetic characters always contain a 1 in bit 5. You can use this fact to quickly convert between upper- and lowercase. If you have an uppercase character, you can force it to lowercase by setting bit 5 to 1. If you have a lowercase character and you wish to force it to uppercase, you can do so by setting bit 5 to 0. You can toggle an alphabetic character between upper- and lowercase by simply inverting bit 5.

Indeed, bits 5 and 6 determine which of the four groups in the ASCII character set you're in, as Table 2-8 shows.

Table 2-8: ASCII Groups

Bit 6	Bit 5	Group
0	0	Control characters
0	1	Digits and punctuation
1	0	Uppercase and special
1	1	Lowercase and special

So you could, for instance, convert any upper- or lowercase (or corresponding special) character to its equivalent control character by setting bits 5 and 6 to 0.

Consider, for a moment, the ASCII codes of the numeric digit characters appearing in Table 2-9.

Table 2-9: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
0	48	$30
1	49	$31
2	50	$32
3	51	$33
4	52	$34
5	53	$35
6	54	$36
7	55	$37
8	56	$38
9	57	$39

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important—the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to 0) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to 3. Note that you can use the logical and operation

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000050

to force the H.O. bits to 0; likewise, you can use the logical or operation to force the H.O. bits to %0011 (3).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 ($31 $32 $33) in this fashion yields 3 bytes: $010203; the correct value for 123 is $7B. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

2.14.2 HLA Support for ASCII Characters

Although you could easily store character values in byte variables and use the corresponding numeric equivalent ASCII code when using a character literal in your program, such agony is unnecessary. HLA provides support for character variables and literals in your assembly language programs.

Character literal constants in HLA take one of two forms: a single character surrounded by apostrophes or a hash mark (#) followed by a numeric constant in the range 0..127 (specifying the ASCII code of the character). Here are some examples:

```
'A'    #65    #$41    #%0100_0001
```

Note that these examples all represent the same character ('A') because the ASCII code of 'A' is 65.

With one exception, only a single character may appear between the apostrophes in a literal character constant. That single exception is the apostrophe character itself. If you wish to create an apostrophe literal constant, place four apostrophes in a row (i.e., double up the apostrophe inside the surrounding apostrophes):

```
''''
```

The hash mark operator (#) must precede a legal HLA numeric constant (either decimal, hexadecimal, or binary, as the examples above indicate). In particular, the hash mark is not a generic character conversion function; it cannot precede registers or variable names, only constants.

As a general rule, you should always use the apostrophe form of the character literal constant for graphic characters (that is, those that are printable or displayable). Use the hash mark form for control characters (that are invisible or do funny things when you print them) or for extended ASCII characters that may not display or print properly within your source code.

Notice the difference between a character literal constant and a string literal constant in your programs. Strings are sequences of zero or more characters surrounded by quotation marks; characters are surrounded by apostrophes.

It is especially important to realize that

```
'A' ≠ "A"
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000051

The character constant 'A' and the string containing the single character A have two completely different internal representations. If you attempt to use a string containing a single character where HLA expects a character constant, HLA will report an error. Strings and string constants are the subject of Chapter 4.

To declare a character variable in an HLA program, you use the char data type. For example, the following declaration demonstrates how to declare a variable named UserInput:

```
static
    UserInput:          char;
```

This declaration reserves 1 byte of storage that you could use to store any character value (including 8-bit extended ASCII characters). You can also initialize character variables as the following example demonstrates:

```
static

    TheCharA:           char := 'A';
    ExtendedChar:       char := #128;
```

Because character variables are 8-bit objects, you can manipulate them using 8-bit registers. You can move character variables into 8-bit registers, and you can store the value of an 8-bit register into a character variable.

The HLA Standard Library provides a handful of routines that you can use for character I/O and manipulation; these include stdout.putc, stdout.putcSize, stdout.put, stdin.getc, and stdin.get.

The stdout.putc routine uses the following calling sequence:

```
stdout.putc( charvar );
```

This procedure outputs the single-character parameter passed to it as a character to the standard output device. The parameter may be any char constant or variable, or a byte variable or register.[12]

The stdout.putcSize routine provides output width control when displaying character variables. The calling sequence for this procedure is

```
stdout.putcSize( charvar, widthInt32, fillchar );
```

This routine prints the specified character (parameter c) using at least *widthInt32* print positions.[13] If the absolute value of *widthInt32* is greater than 1, then stdout.putcSize prints the *fillchar* character as padding. If the value of *widthInt32* is positive, then stdout.putcSize prints the character right justified

[12] If you specify a byte variable or a byte-sized register as the parameter, the stdout.putc routine will output the character whose ASCII code appears in the variable or register.

[13] The only time stdout.putcSize uses more print positions than you specify is when you specify 0 as the width; then this routine uses exactly one print position.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000052

in the print field; if *widthInt32* is negative, then stdout.putcSize prints the character left justified in the print field. Because character output is usually left justified in a field, the *widthInt32* value will normally be negative for this call. The space character is the most common *fillchar* value.

You can also print character values using the generic stdout.put routine. If a character variable appears in the stdout.put parameter list, then stdout.put will automatically print it as a character value. For example:

```
stdout.put( "Character c = '", c, "'", nl );
```

You can read characters from the standard input using the stdin.getc and stdin.get routines. The stdin.getc routine does not have any parameters. It reads a single character from the standard input buffer and returns this character in the AL register. You may then store the character value away or otherwise manipulate the character in the AL register. The program in Listing 2-10 reads a single character from the user, converts it to uppercase if it is a lowercase character, and then displays the character.

```
program charInputDemo;
#include( "stdlib.hhf" )
begin charInputDemo;

    stdout.put( "Enter a character: " );
    stdin.getc();
    if( al >= 'a' ) then

        if( al <= 'z' ) then

            and( $5f, al );

        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,
        "converted to uppercase, was '"
    );
    stdout.putc( al );
    stdout.put( "'", nl );

end charInputDemo;
```

Listing 2-10: Character input sample

You can also use the generic stdin.get routine to read character variables from the user. If a stdin.get parameter is a character variable, then the stdin.get routine will read a character from the user and store the character value into the specified variable. Listing 2-11 is a rewrite of Listing 2-10 using the stdin.get routine.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000053

```
program charInputDemo2;
#include( "stdlib.hhf" )
static
    c:char;

begin charInputDemo2;

    stdout.put( "Enter a character: " );
    stdin.get(c);
    if( c >= 'a' ) then

        if( c <= 'z' ) then

            and( $5f, c );

        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,
        "converted to uppercase, was '",
        c,
        "'", nl
    );

end charInputDemo2;
```

Listing 2-11: `stdin.get` character input sample

As you may recall from the last chapter, the HLA Standard Library buffers its input. Whenever you read a character from the standard input using `stdin.getc` or `stdin.get`, the library routines read the next available character from the buffer; if the buffer is empty, then the program reads a new line of text from the user and returns the first character from that line. If you want to guarantee that the program reads a new line of text from the user when you read a character variable, you should call the `stdin.flushInput` routine before attempting to read the character. This will flush the current input buffer and force the input of a new line of text on the next input (probably a `stdin.getc` or `stdin.get` call).

The end of line is problematic. Different operating systems handle the end of line differently on output versus input. From the console device, pressing the ENTER key signals the end of a line; however, when reading data from a file, you get an end-of-line sequence that is a linefeed or a carriage return/line feed pair (under Windows) or just a line feed (under Linux/Mac OS X/FreeBSD). To help solve this problem, HLA's Standard Library provides an "end of line" function. This procedure returns true (1) in the AL register if all the current input characters have been exhausted; it returns false (0) otherwise. The sample program in Listing 2-12 demonstrates the `stdin.eoln` function.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000054

```
program eolnDemo;
#include( "stdlib.hhf" )
begin eolnDemo;

    stdout.put( "Enter a short line of text: " );
    stdin.flushInput();
    repeat

        stdin.getc();
        stdout.putc( al );
        stdout.put( "=$", al, nl );

    until( stdin.eoln() );

end eolnDemo;
```

Listing 2-12: Testing for end of line using `stdin.eoln`

The HLA language and the HLA Standard Library provide many other procedures and additional support for character objects. Chapters 4 and 11, as well as the HLA reference documentation, describe how to use these features.

2.15 The Unicode Character Set

Although the ASCII character set is, unquestionably, the most popular character representation on computers, it is certainly not the only format around. For example, IBM uses the EBCDIC code on many of its mainframe and minicomputer lines. Because EBCDIC appears mainly on IBM's big iron and you'll rarely encounter it on personal computer systems, we will not consider that character set in this text. Another character representation that is becoming popular on small computer systems (and large ones, for that matter) is the Unicode character set. Unicode overcomes two of ASCII's greatest limitations: the limited character space (i.e., a maximum of 128/256 characters in an 8-bit byte) and the lack of international (beyond the United States) characters.

Unicode uses a 16-bit word to represent a single character. Therefore, Unicode supports up to 65,536 different character codes. This is obviously a huge advance over the 256 possible codes we can represent with an 8-bit byte. Unicode is upward compatible from ASCII. Specifically, if the H.O. 9 bits of a Unicode character contain 0, then the L.O. 7 bits represent the same character as the ASCII character with the same character code. If the H.O. 9 bits contain some nonzero value, then the character represents some other value. If you're wondering why so many different character codes are necessary, simply note that certain Asian character sets contain 4,096 characters (at least their Unicode subset does).

This text will stick to the ASCII character set except for a few brief mentions of Unicode here and there. Eventually, this text may have to eliminate the discussion of ASCII in favor of Unicode because many new operating systems are using Unicode internally (and converting to ASCII as

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000055

necessary). Unfortunately, many string algorithms are not as conveniently written for Unicode as for ASCII (especially character set functions), so we'll stick with ASCII in this text as long as possible.

2.16 For More Information

The electronic edition of this book (on Webster at *http://webster.cs.ucr.edu/* or *http://artofasm.com/*) contains some additional information on data representation you may find useful. For general information about data representation, you should consider reading my book *Write Great Code, Volume 1* (No Starch Press, 2004), or a textbook on data structures and algorithms (available at any bookstore).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000056

3

MEMORY ACCESS AND ORGANIZATION

Chapters 1 and 2 show you how to declare and access simple variables in an assembly language program. This chapter fully explains 80x86 memory access. You will learn how to efficiently organize your variable declarations to speed up access to their data. This chapter will teach you about the 80x86 stack and how to manipulate data on the stack. Finally, this chapter will teach you about dynamic memory allocation and the *heap*.

This chapter discusses several important concepts, including:

- 80x86 memory addressing modes
- Indexed and scaled-indexed addressing modes
- Memory organization
- Memory allocation by program
- Data type coercion

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000057

- The 80x86 stack
- Dynamic memory allocation

This chapter will teach to you make efficient use of your computer's memory resources.

3.1 The 80x86 Addressing Modes

The 80x86 processors let you access memory in many different ways. Until now, you've seen only a single way to access a variable, the so-called *displacement-only* addressing mode. In this section you'll see some additional ways your programs can access memory using 80x86 *memory addressing modes*. The 80x86 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step toward mastering 80x86 assembly language.

When Intel designed the original 8086 processor, it provided the processor with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. However, in 32-bit environments like Windows, Mac OS X, FreeBSD, and Linux, these earlier addressing modes are not very useful; indeed, HLA doesn't even support the use of these older, 16-bit-only addressing modes. Fortunately, anything you can do with the older addressing modes can be done with the new addressing modes. Therefore, you won't need to bother learning the old 16-bit addressing modes when writing code for today's high-performance operating systems. Do keep in mind, however, that if you intend to work under MS-DOS or some other 16-bit operating system, you will need to study up on those old addressing modes (see the 16-bit edition of this book at *http://webster.cs.ucr.edu/* for details).

3.1.1 80x86 Register Addressing Modes

Most 80x86 instructions can operate on the 80x86's general-purpose register set. By specifying the name of the register as an operand to the instruction, you can access the contents of that register. Consider the 80x86 `mov` (move) instruction:

```
mov( source, destination );
```

This instruction copies the data from the *source* operand to the *destination* operand. The 8-bit, 16-bit, and 32-bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 80x86 `mov` instructions:

```
mov( bx, ax );          // Copies the value from bx into ax
mov( al, dl );          // Copies the value from al into dl
mov( edx, esi );        // Copies the value from edx into esi
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000058

```
mov( bp, sp );          // Copies the value from bp into sp
mov( cl, dh );          // Copies the value from cl into dh
mov( ax, ax );          // Yes, this is legal!
```

The registers are the best place to keep variables. Instructions using the registers are shorter and faster than those that access memory. Of course, most computations require at least one register operand, so the register addressing mode is very popular in 80x86 assembly code.

3.1.2 80x86 32-Bit Memory Addressing Modes

The 80x86 provides hundreds of different ways to access memory. This may seem like quite a lot at first, but fortunately most of the addressing modes are simple variants of one another, so they're very easy to learn. And learn them you should! The key to good assembly language programming is the proper use of memory addressing modes.

The addressing modes provided by the 80x86 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide all the different addressing modes on the 80x86. See, from hundreds down to five. It's not so bad after all!

3.1.2.1 The Displacement-Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location. Assuming that variable `j` is an `int8` variable appearing at address $8088, the instruction `mov( j, al );` loads the AL register with a copy of the byte at memory location $8088. Likewise, if `int8` variable `k` is at address $1234 in memory, then the instruction `mov( dl, k );` stores the value in the DL register to memory location $1234 (see Figure 3-1).

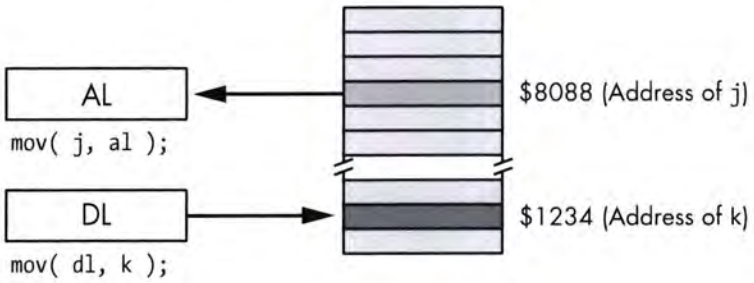


Figure 3-1: Displacement-only (direct) addressing mode

The displacement-only addressing mode is perfect for accessing simple scalar variables. This is named the displacement-only addressing mode because a 32-bit constant (displacement) follows the `mov` opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address 0). The examples in this chapter often access

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000059

bytes in memory. Don't forget, however, that you can also access words and double words on the 80x86 processors by specifying the address of their first byte (see Figure 3-2).

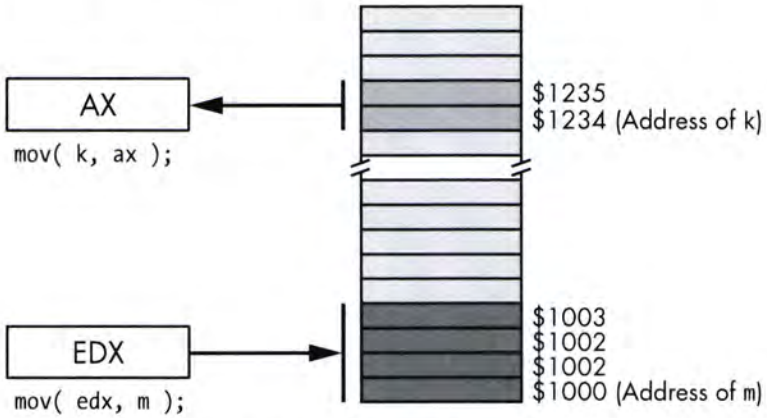


Figure 3-2: Accessing a word or dword using the displacement-only addressing mode

3.1.2.2 The Register-Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the *register-indirect* addressing modes. The term *indirect* means that the operand is not the actual address, but rather the operand's value specifies the memory address to use. In the case of the register-indirect addressing modes, the value held in the register is the address of the memory location to access. For example, the instruction `mov( eax, [ebx] );` tells the CPU to store EAX's value at the location whose address is in EBX (the square brackets around EBX tell HLA to use the register-indirect addressing mode).

There are eight forms of this addressing mode on the 80x86. The following instructions are examples of these eight forms:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

These eight addressing modes reference the memory location at the offset found in the register enclosed by brackets (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP, respectively).

Note that the register-indirect addressing modes require a 32-bit register. You cannot specify a 16-bit or 8-bit register when using an indirect addressing mode.[1] Technically, you could load a 32-bit register with an

[1] Actually, the 80x86 does support addressing modes involving certain 16-bit registers, as mentioned earlier. However, HLA does not support these modes and they are not useful under 32-bit operating systems.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000060

arbitrary numeric value and access that location indirectly using the register-indirect addressing mode:

```
mov( $1234_5678, ebx );
mov( [ebx], al );     // Attempts to access location $1234_5678.
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause the operating system to generate a protection fault because it's not always legal to access arbitrary memory locations. As it turns out, there are better ways to load the address of some object into a register; you'll see how to do this shortly.

The register-indirect addressing modes have many uses. You can use them to access data referenced by a pointer, you can use them to step through array data, and, in general, you can use them whenever you need to modify the address of a variable while your program is running.

The register-indirect addressing mode provides an example of an *anonymous* variable. When using a register-indirect addressing mode, you refer to the value of a variable by its numeric memory address (e.g., the value you load into a register) rather than by the name of the variable—hence the phrase *anonymous variable.*

HLA provides a simple operator that you can use to take the address of a `static` variable and put this address into a 32-bit register. This is the `&` (address-of) operator (note that this is the same symbol that C/C++ uses for the address-of operator). The following example loads the address of variable `j` into EBX and then stores EAX's current value into `j` using a register-indirect addressing mode:

```
mov( &j, ebx );              // Load address of j into ebx.
mov( eax, [ebx] );           // Store eax into j.
```

Of course, it would have been easier to store EAX's value directly into `j` rather than using two instructions to do this indirectly. However, you can easily imagine a code sequence where the program loads one of several different addresses into EBX prior to the execution of the `mov( eax, [ebx]);` statement, thus storing EAX into one of several different locations depending on the execution path of the program.

WARNING *The & (address-of) operator is not a general address-of operator like the & operator in C/C++. You may apply this operator only to static variables.*[2] *You cannot apply it to generic address expressions or other types of variables. In Section 3.13, you will learn about the* load effective address *instruction that provides a general solution for obtaining the address of some variable in memory.*

[2] The term *static* here indicates a `static`, `readonly`, or `storage` object.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000061

3.1.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

```
mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );
```

VarName is the name of some variable in your program.

The indexed addressing modes compute an effective address[3] by adding the address of the variable to the value of the 32-bit register appearing inside the square brackets. Their sum is the actual memory address the instruction accesses. So if *VarName* is at address $1100 in memory and EBX contains 8, then `mov(VarName[ ebx ], al);` loads the byte at address $1108 into the AL register (see Figure 3-3).

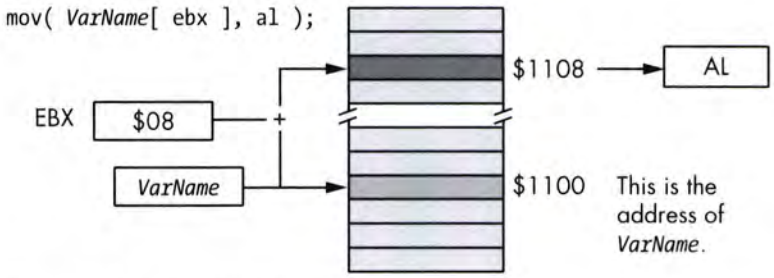


Figure 3-3: Indexed addressing mode

The indexed addressing modes are really handy for accessing elements of arrays. You will see how to use these addressing modes for that purpose in Chapter 4.

3.1.2.4 Variations on the Indexed Addressing Mode

There are two important syntactical variations of the indexed addressing mode. Both forms generate the same basic machine instructions, but their syntax suggests other uses for these variants.

The first variant uses the following syntax:

```
mov( [ ebx + constant ], al );
mov( [ ebx - constant ], al );
```

These examples use only the EBX register. However, you can use any of the other 32-bit general-purpose registers in place of EBX. This form computes its effective address by adding the value in EBX to the specified constant or subtracting the specified constant from EBX (see Figures 3-4 and 3-5).

[3] The effective address is the ultimate address in memory that an instruction will access, once all the address calculations are complete.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000062

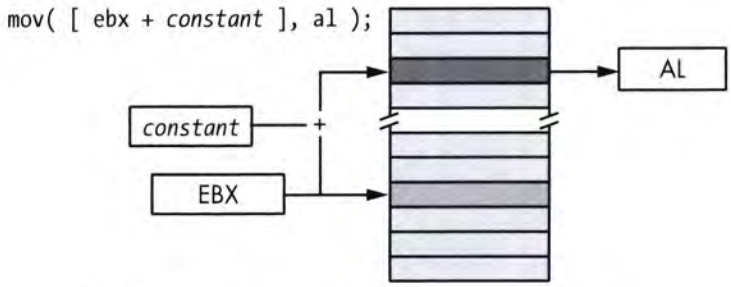


Figure 3-4: Indexed addressing mode using a register plus a constant

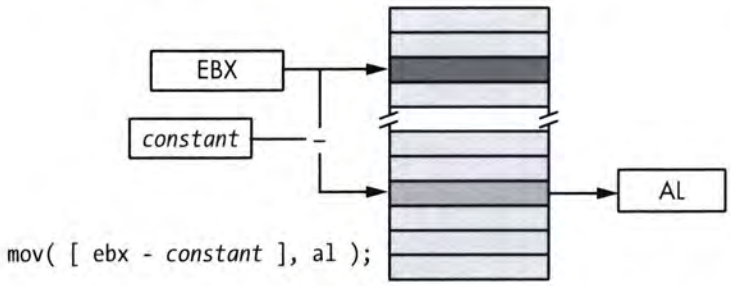


Figure 3-5: Indexed addressing mode using a register minus a constant

This particular variant of the addressing mode is useful if a 32-bit register contains the base address of a multibyte object and you wish to access a memory location some number of bytes before or after that location. One important use of this addressing mode is accessing fields of a record (or structure) when you have a pointer to the record data. This addressing mode is also invaluable for accessing automatic (local) variables in procedures (see Chapter 5 for more details).

The second variant of the indexed addressing mode is actually a combination of the previous two forms. The syntax for this version is the following:

```
mov( VarName[ ebx + constant ], al );
mov( VarName[ ebx - constant ], al );
```

Once again, this example uses only the EBX register. You may substitute any of the 32-bit general-purpose registers in lieu of EBX in these two examples. This particular form is useful when accessing elements of an array of records (structures) in an assembly language program (more on that in Chapter 4).

These instructions compute their effective address by adding or subtracting the *constant* value from *VarName*'s address and then adding the value in EBX to this result. Note that HLA, not the CPU, computes the sum or difference of *VarName*'s address and *constant*. The actual machine instructions above contain a single constant value that the instructions add to the value in EBX at runtime. Because HLA substitutes a constant for *VarName*, it can reduce an instruction of the form

```
mov( VarName[ ebx + constant], al );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000063

to an instruction of the form

```
mov( constant1[ ebx + constant2], al );
```

Because of the way these addressing modes work, this is semantically equivalent to

```
mov( [ebx + (constant1 + constant2)], al );
```

HLA will add the two constants together at compile time, effectively producing the following instruction:

```
mov( [ebx + constant_sum], al );
```

Of course, there is nothing special about subtraction. You can easily convert the addressing mode involving subtraction to addition by simply taking the two's complement of the 32-bit constant and then adding this complemented value (rather than subtracting the original value).

3.1.2.5 Scaled-Indexed Addressing Modes

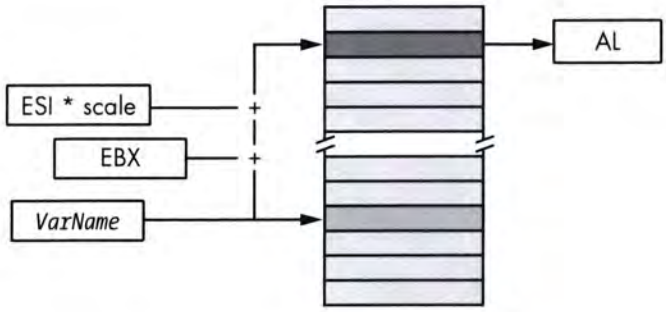
The scaled-indexed addressing modes are similar to the indexed addressing modes with two differences: (1) The scaled-indexed addressing modes allow you to combine two registers plus a displacement, and (2) the scaled-indexed addressing modes let you multiply the index register by a (scaling) factor of 1, 2, 4, or 8. The syntax for these addressing modes is

```
VarName[ IndexReg32*scale ]
VarName[ IndexReg32*scale + displacement ]
VarName[ IndexReg32*scale - displacement ]

[ BaseReg32 + IndexReg32*scale ]
[ BaseReg32 + IndexReg32*scale + displacement ]
[ BaseReg32 + IndexReg32*scale - displacement ]

VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

In these examples, *BaseReg32* represents any general-purpose 32-bit register, *IndexReg32* represents any general-purpose 32-bit register except ESP, and *scale* must be one of the constants 1, 2, 4, or 8.

The primary difference between the scaled-indexed addressing modes and the indexed addressing modes is the inclusion of the *IndexReg32*scale* component. These modes compute the effective address by adding in the value of this new register multiplied by the specified scaling factor (see Figure 3-6 for an example involving EBX as the base register and ESI as the index register).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000064




```
mov( VarName[ ebx + esi*scale ], al );
```

Figure 3-6: Scaled-indexed addressing mode

In Figure 3-6, suppose that EBX contains $100, ESI contains $20, and *VarName* is at base address $2000 in memory; then the following instruction

```
mov( VarName[ ebx + esi*4 + 4 ], al );
```

will move the byte at address $2184 ($100 + $20*4 + 4) into the AL register.

The scaled-indexed addressing modes are useful for accessing elements of arrays whose elements are 2, 4, or 8 bytes each. These addressing modes are also useful for access elements of an array when you have a pointer to the beginning of the array.

3.1.2.6 Addressing Mode Wrap-up

Well, believe it or not, you've just learned several hundred addressing modes! That wasn't hard now, was it? If you're wondering where all these modes came from, just note that the register-indirect addressing mode isn't a single addressing mode but eight different addressing modes (involving the eight different registers). Combinations of registers, constant sizes, and other factors multiply the number of possible addressing modes on the system. In fact, you need only memorize about two dozen forms and you've got it made. In practice, you'll use less than half the available addressing modes in any given program (and many addressing modes you may never use at all). So learning all these addressing modes is actually much easier than it sounds.

3.2 Runtime Memory Organization

An operating system like Mac OS X, FreeBSD, Linux, or Windows tends to put different types of data into different sections (or segments) of memory. Although it is possible to reconfigure memory to your choice by running the linker and specifying various parameters, by default Windows loads an HLA program into memory using the organization appearing in Figure 3-7 (Linux, Mac OS X, and FreeBSD are similar, though they rearrange some of the sections).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000065

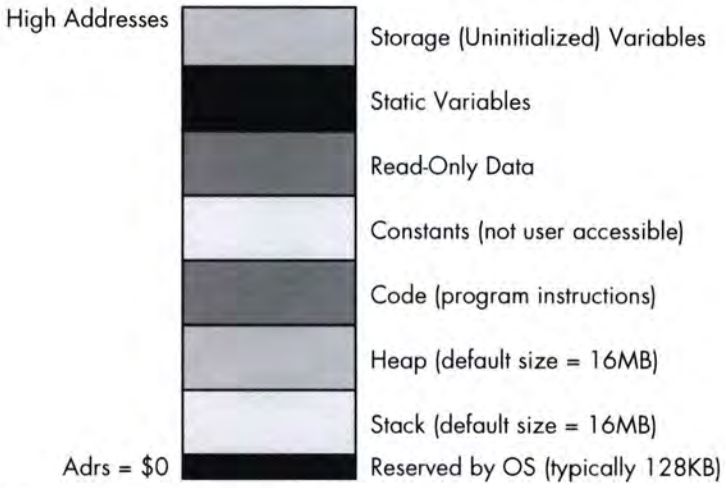


Figure 3-7: HLA typical runtime memory organization

The operating system reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at these low addresses. One reason the operating system reserves this space is to help trap NULL pointer references. If you attempt to access memory location 0, the operating system will generate a general protection fault, meaning you've accessed a memory location that doesn't contain valid data. Because programmers often initialize pointers to NULL (0) to indicate that the pointer is not pointing anywhere, an access of location 0 typically means that the programmer has made a mistake and has not properly initialized a pointer to a legal (non-NULL) value.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the `stack` section, the `heap` section, the `code` section, the `readonly` section, the `static` section, and the `storage` section. Each of these memory sections correspond to some type of data you can create in your HLA programs. Each section is discussed in detail below.

3.2.1 The code Section

The `code` section contains the machine instructions that appear in an HLA program. HLA translates each machine instruction you write into a sequence of one or more byte values. The CPU interprets these byte values as machine instructions during program execution.

By default, when HLA links your program it tells the system that your program can execute instructions in the code segment and you can read data from the code segment. Note, specifically, that you cannot write data to the code segment. The operating system will generate a general protection fault if you attempt to store any data into the code segment.

Remember, machine instructions are nothing more than data bytes. In theory, you could write a program that stores data values into memory and then transfers control to the data it just wrote, thereby producing a program that writes itself as it executes. This possibility produces romantic visions of *Artificial Intelligence* programs that modify themselves to produce some desired

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000066

result. In real life, the effect is somewhat less glamorous. Generally, self-modifying programs are very difficult to debug because the instructions are constantly changing behind the programmer's back. Because most modern operating systems make it very difficult to write self-modifying programs, we will not consider them any further in this text.

HLA automatically stores the data associated with your machine code into the code section. In addition to machine instructions, you can also store data into the code section by using the following pseudo-opcodes:[4]

```
byte      int8
word      int16
dword     in32
uns8      boolean
uns16     char
uns32
```

The following `byte` statement exemplifies the syntax for each of these pseudo-opcodes:

```
byte comma_separated_list_of_byte_constants ;
```

Here are some examples:

```
boolean     true;
char        'A';
byte        0, 1, 2;
byte        "Hello", 0
word        0, 2;
int8        -5;
uns32       356789, 0;
```

If more than one value appears in the list of values after the pseudo-opcode, HLA emits each successive value to the code stream. So the first `byte` statement above emits 3 bytes to the code stream, the values 0, 1, and 2. If a string appears within a `byte` statement, HLA emits 1 byte of data for each character in the string. Therefore, the second byte statement above emits 6 bytes: the characters `H`, `e`, `l`, `l`, and `o`, followed by a `0` byte.

Keep in mind that the CPU will attempt to treat data you emit to the code stream as machine instructions unless you take special care not to allow the execution of the data. For example, if you write something like the following:

```
mov( 0, ax );
byte 0,1,2,3;
add( bx, cx );
```

[4] This isn't a complete list. HLA generally allows you to use any scalar data type name as a statement to reserve storage in the code section. You'll learn more about the available data types in Chapter 4.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000067

your program will attempt to execute the 0, 1, 2, and 3 byte values as machine instructions after executing the mov. Unless you know the machine code for a particular instruction sequence, sticking such data values into the middle of your code will generally crash your program. Typically when you place such data in your programs, you'll execute some code that transfers control around the data.

3.2.2 The static Section

The static section is where you will typically declare your variables. Although the static section syntactically appears as part of a program or procedure, keep in mind that HLA moves all static variables to the static section in memory. Therefore, HLA does not sandwich the variables you declare in the static section between procedures in the code section.

In addition to declaring static variables, you can also embed lists of data into the static declaration section. You use the same technique to embed data into your static section that you use to embed data into the code section: You use the byte, word, dword, uns32, and so on pseudo-opcodes. Consider the following example:

```
static
    b:   byte := 0;
         byte 1,2,3;

    u:   uns32 := 1;
         uns32 5,2,10;

    c:   char;
         char 'a', 'b', 'c', 'd', 'e', 'f';

    bn:  boolean;
         boolean true;
```

Data that HLA writes to the static memory segment using these pseudo-opcodes is written to the segment after the preceding variables. For example, the byte values 1, 2, and 3 are emitted to the static section after b's 0 byte. Because there aren't any labels associated with these values, you do not have direct access to these values in your program. You can use the indexed addressing modes to access these extra values (examples appear in Chapter 4).

In the examples above, note that the c and bn variables do not have an (explicit) initial value. However, if you don't provide an initial value, HLA will initialize the variables in the static section to all 0 bits, so HLA assigns the NUL character (ASCII code 0) to c as its initial value. Likewise, HLA assigns false as the initial value for bn. In particular, you should note that your variable declarations in the static section always consume memory, even if you haven't assigned them an initial value.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000068

3.2.3 The readonly Data Section

The readonly data section holds constants, tables, and other data that your program cannot change during execution. You create read-only objects by declaring them in the readonly declaration section. The readonly section is very similar to the static section with three primary differences:

- The readonly section begins with the reserved word readonly rather than static.
- All declarations in the readonly section generally have an initializer.
- The system does not allow you to store data into a readonly object while the program is running.

Here's an example:

```
readonly
    pi:         real32 := 3.14159;
    e:          real32 := 2.71;
    MaxU16:     uns16 := 65_535;
    MaxI16:     int16 := 32_767;
```

All readonly object declarations must have an initializer because you cannot initialize the value under program control.[5] For all intents and purposes, you can think of readonly objects as constants. However, these constants consume memory, and other than the fact that you cannot write data to readonly objects, they behave like static variables. Because they behave like static objects, you cannot use a readonly object everywhere a constant is allowed; in particular, readonly objects are memory objects, so you cannot supply a readonly object (which you are treating like a constant) and some other memory object as the operands to an instruction.

As with the static section, you may embed data values in the readonly section using the byte, word, dword, and so on data declarations. For example:

```
readonly
    roArray: byte := 0;
             byte 1, 2, 3, 4, 5;
    qwVal:   qword := 1;
             qword 0;
```

3.2.4 The storage Section

The readonly section requires that you initialize all objects you declare. The static section lets you optionally initialize objects (or leave them uninitialized, in which case they have the default initial value of 0). The storage section completes the initialization coverage: you use it to declare variables

[5] There is one exception you'll see in Chapter 5.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000069

that are always uninitialized when the program begins running. The storage section begins with the storage reserved word and contains variable declarations without initializers. Here is an example:

```
storage
    UninitUns32:    uns32;
    i:              int32;
    character:      char;
    b:              byte;
```

Linux, FreeBSD, Mac OS X, and Windows will initialize all storage objects to 0 when they load your program into memory. However, it's probably not a good idea to depend on this implicit initialization. If you need an object initialized with 0, declare it in a static section and explicitly set it to 0.

Variables you declare in the storage section may consume less disk space in the executable file for the program. This is because HLA writes out initial values for readonly and static objects to the executable file, but it may use a compact representation for uninitialized variables you declare in the storage section; note, however, that this behavior is OS- and object-module-format dependent.

Because the storage section does not allow initialized values, you *cannot* put unlabeled values in the storage section using the byte, word, dword, and so on pseudo-opcodes.

3.2.5 The @nostorage Attribute

The @nostorage attribute lets you declare variables in the static data declaration sections (i.e., static, readonly, and storage) without actually allocating memory for the variable. The @nostorage option tells HLA to assign the current address in a declaration section to a variable but not to allocate any storage for the object. That variable will share the same memory address as the next object appearing in the variable declaration section. Here is the syntax for the @nostorage option:

```
    variableName: varType; @nostorage;
```

Note that you follow the type name with @nostorage; rather than some initial value or just a semicolon. The following code sequence provides an example of using the @nostorage option in the readonly section:

```
readonly
    abcd: dword; nostorage;
          byte 'a', 'b', 'c', 'd';
```

In this example, abcd is a double word whose L.O. byte contains 97 ('a'), byte 1 contains 98 ('b'), byte 2 contains 99 ('c'), and the H.O. byte contains 100 ('d'). HLA does not reserve storage for the abcd variable, so HLA associates the following 4 bytes in memory (allocated by the byte directive) with abcd.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000070

Note that the `@nostorage` attribute is legal only in the `static`, `storage`, and `readonly` sections (the so-called *static* declarations sections). HLA does not allow its use in the `var` section that you'll read about next.

3.2.6 The var Section

HLA provides another variable declaration section, the `var` section, that you can use to create *automatic* variables. Your program will allocate storage for automatic variables whenever a program unit (i.e., main program or procedure) begins execution, and it will deallocate storage for automatic variables when that program unit returns to its caller. Of course, any automatic variables you declare in your main program have the same *lifetime*[6] as all the `static`, `readonly`, and `storage` objects, so the automatic allocation feature of the `var` section is wasted in the main program. In general, you should use automatic objects only in procedures (see Chapter 5 for details). HLA allows them in your main program's declaration section as a generalization.

Because variables you declare in the `var` section are created at runtime, HLA does not allow initializers on variables you declare in this section. So the syntax for the `var` section is nearly identical to that for the `storage` section; the only real difference in the syntax between the two is the use of the `var` reserved word rather than the `storage` reserved word.[7] The following example illustrates this:

```
var
    vInt:     int32;
    vChar:    char;
```

HLA allocates variables you declare within the `var` section within the `stack` memory section. HLA does not allocate `var` objects at fixed locations; instead, it allocates these variables in an activation record associated with the current program unit. Chapter 5 discusses activation records in greater detail; for now it is important only to realize that HLA programs use the EBP register as a pointer to the current activation record. Therefore, whenever you access a `var` object, HLA automatically replaces the variable name with `[EBP±displacement]`. Displacement is the offset of the object within the activation record. This means that you cannot use the full scaled-indexed addressing mode (a base register plus a scaled index register) with `var` objects because `var` objects already use the EBP register as their base register. Although you will not directly use the two register addressing modes often, the fact that the `var` section has this limitation is a good reason to avoid using the `var` section in your main program.

[6] The lifetime of a variable is the point from which memory is first allocated to the point the memory is deallocated for that variable.

[7] Actually, there are a few other, minor, differences, but we won't deal with those differences in this text. See the HLA language reference manual for more details.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000071

3.2.7 Organization of Declaration Sections Within Your Programs

The static, readonly, storage, and var sections may appear zero or more times between the program header and the associated begin for the main program. Between these two points in your program, the declaration sections may appear in any order, as the following example demonstrates:

```
program demoDeclarations;

static
    i_static:     int32;

var
    i_auto:       int32;

storage
    i_uninit:     int32;

readonly
    i_readonly:   int32 := 5;

static
    j:            uns32;

var
    k:            char;

readonly
    i2:           uns8 := 9;

storage
    c:            char;

storage
    d:            dword;

begin demoDeclarations;

    << Code goes here. >>

end demoDeclarations;
```

In addition to demonstrating that the sections may appear in an arbitrary order, this section also demonstrates that a given declaration section may appear more than once in your program. When multiple declaration sections of the same type (for example, the three storage sections above) appear in a declaration section of your program, HLA combines them into a single group.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000072

3.3 How HLA Allocates Memory for Variables

As you've seen, the 80x86 CPU doesn't deal with variables that have names like `I`, `Profits`, and `LineCnt`. The CPU deals strictly with numeric addresses it can place on the address bus like $1234_5678, $0400_1000, and $8000_CC00. HLA, on the other hand, does not force to you refer to variable objects by their addresses (which is nice, because names are so much easier to remember). This is good, but it does obscure what is really going on. In this section, we'll take a look at how HLA associates numeric addresses with your variables so you'll understand (and appreciate) the process that is taking place behind your back.

Take another look at Figure 3-7. As you can see, the various memory sections tend to be adjacent to one another. Therefore, if the size of one memory section changes, then this affects the starting address of all the following sections in memory. For example, if you add a few additional machine instructions to your program and increase the size of the `code` section, this may affect the starting address of the `static` section in memory, thus changing the addresses of all your static variables. Keeping track of variables by their numeric address (rather than by their names) is difficult enough; imagine how much worse it would be if the addresses are constantly shifting around as you add and remove machine instructions in your program! Fortunately, you don't have to keep track of variable addresses; HLA does that bookkeeping for you.

HLA associates a current *location counter* with each of the three static declaration sections (`static`, `readonly`, and `storage`). These location counters initially contain 0, and whenever you declare a variable in one of the static sections, HLA associates the current value of that section's location counter with the variable; HLA also bumps up the value of that location counter by the size of the object you're declaring. As an example, assume that the following is the only `static` declaration section in a program:

```
static
    b     :byte;                // Location counter = 0, size = 1
    w     :word;                // Location counter = 1, size = 2
    d     :dword;               // Location counter = 3, size = 4
    q     :qword;               // Location counter = 7, size = 8
    l     :lword;               // Location counter = 15, size = 16
                                // Location counter is now 31.
```

Of course, the runtime address of each of these variables is not the value of the location counter. First of all, HLA adds in the base address of the `static` memory section to each of these location counter values (which we call *displacements* or *offsets*). Second, there may be other static objects in modules that you link with your program (e.g., from the HLA Standard Library) or even additional `static` sections in the same source file, and the linker has to merge the `static` sections together. Hence, these offsets may have very little bearing on the final address of these variables in memory.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000073

Nevertheless, one important fact remains: HLA allocates variables you declare in a single static declaration section in contiguous memory locations. That is, given the declaration above, w will immediately follow b in memory, d will immediately follow w in memory, q will immediately follow d, and so on. Generally, it's not good coding style to assume that the system allocates variables this way, but sometimes it's convenient to do so.

Note that HLA allocates memory objects you declare in readonly, static, and storage sections in completely different regions of memory. Therefore, you cannot assume that the following three memory objects appear in adjacent memory locations (indeed, they probably will not):

```
static
    b       :byte;
readonly
    w       :word := $1234;
storage
    d       :dword;
```

In fact, HLA will not even guarantee that variables you declare in separate static (or whatever) sections are adjacent in memory, even if there is nothing between the declarations in your code (for example, you cannot assume that b, w, and d are in adjacent memory locations in the following declarations, nor can you assume that they *won't* be adjacent in memory):

```
static
    b       :byte;
static
    w       :word := $1234;
static
    d       :dword;
```

If your code requires these variables to consume adjacent memory locations, you must declare them in the same static section.

Note that HLA handles variables you declare in the var section a little differently than the variables you declare in one of the static sections. We'll discuss the allocation of offsets to var objects in Chapter 5.

3.4 HLA Support for Data Alignment

In order to write fast programs, you need to ensure that you properly align data objects in memory. Proper alignment means that the starting address for an object is a multiple of some size, usually the size of an object if the object's size is a power of 2 for values up to 16 bytes in length. For objects greater than 16 bytes, aligning the object on an 8-byte or 16-byte address boundary is probably sufficient. For objects less than 16 bytes, aligning the object at an address that is the next power of 2 greater than the object's size is usually fine. Accessing data that is not aligned at an appropriate address may require extra time; so if you want to ensure that your program runs as rapidly as possible, you should try to align data objects according to their size.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000074

Data becomes misaligned whenever you allocate storage for different-sized objects in adjacent memory locations. For example, if you declare a byte variable, it will consume 1 byte of storage, and the next variable you declare in that declaration section will have the address of that byte object plus 1. If the byte variable's address happens to be an even address, then the variable following that byte will start at an odd address. If that following variable is a word or double-word object, then its starting address will not be optimal. In this section, we'll explore ways to ensure that a variable is aligned at an appropriate starting address based on that object's size.

Consider the following HLA variable declarations:

```
static
    dw:     dword;
    b:      byte;
    w:      word;
    dw2:    dword;
    w2:     word;
    b2:     byte;
    dw3:    dword;
```

The first static declaration in a program (running under Windows, Mac OS X, FreeBSD, Linux, and most 32-bit operating systems) places its variables at an address that is an even multiple of 4,096 bytes. Whatever variable first appears in the static declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables plus the starting address of that static section. Therefore, assuming HLA allocates the variables in the previous example at a starting address of 4096, HLA will allocate them at the following addresses:

```
                        //  Start Adrs          Length
    dw:     dword;      //     4096                4
    b:      byte;       //     4100                1
    w:      word;       //     4101                2
    dw2:    dword;      //     4103                4
    w2:     word;       //     4107                2
    b2:     byte;       //     4109                1
    dw3:    dword;      //     4110                4
```

With the exception of the first variable (which is aligned on a 4KB boundary) and the byte variables (whose alignment doesn't matter), all of these variables are misaligned. The w, w2, and dw2 variables start at odd addresses, and the dw3 variable is aligned on an even address that is not a multiple of 4.

An easy way to guarantee that your variables are aligned properly is to put all the double-word variables first, the word variables second, and the byte variables last in the declaration, as shown here:

```
static
    dw:     dword;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000075

```
dw2:    dword;
dw3:    dword;
w:      word;
w2:     word;
b:      byte;
b2:     byte;
```

This organization produces the following addresses in memory:

```
                    //  Start Adrs          Length
dw:     dword;      //     4096                4
dw2:    dword;      //     4100                4
dw3:    dword;      //     4104                4
w:      word;       //     4108                2
w2:     word;       //     4110                2
b:      byte;       //     4112                1
b2:     byte;       //     4113                1
```

As you can see, these variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While there are many technical reasons that make this alignment impossible, a good practical reason for not doing this is that it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another regardless of their size).

To resolve this problem, HLA provides the `align` directive. The `align` directive uses the following syntax:

```
align( integer_constant );
```

The integer constant must be one of the following small unsigned integer values: 1, 2, 4, 8, or 16. If HLA encounters the `align` directive in a `static` section, it will align the very next variable on an address that is an even multiple of the specified alignment constant. The previous example could be rewritten, using the `align` directive, as follows:

```
static
    align( 4 );
    dw:     dword;
    b:      byte;
    align( 2 );
    w:      word;
    align( 4 );
    dw2:    dword;
    w2:     word;
    b2:     byte;
    align( 4 );
    dw3:    dword;
```

If you're wondering how the `align` directive works, it's really quite simple. If HLA determines that the current address (location counter value) is not

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000076

an even multiple of the specified value, HLA will quietly emit extra bytes of padding after the previous variable declaration until the current address in the `static` section is an even multiple of the specified value. This has the effect of making your program slightly larger (by a few bytes) in exchange for faster access to your data. Given that your program will grow by only a few bytes when you use this feature, this is probably a good trade-off.

As a general rule, if you want the fastest possible access, you should choose an alignment value that is equal to the size of the object you want to align. That is, you should align words to even boundaries using an `align(2);` statement, double words to 4-byte boundaries using `align(4);`, quad words to 8-byte boundaries using `align(8);`, and so on. If the object's size is not a power of 2, align it to the next higher power of 2 (up to a maximum of 16 bytes). Note, however, that you need only align `real80` (and `tbyte`) objects on an 8-byte boundary.

Note that data alignment isn't always necessary. The cache architecture of modern 80x86 CPUs actually handles most misaligned data. Therefore, you should use the alignment directives only with variables for which speedy access is absolutely critical. This is a reasonable space/speed trade-off.

3.5 Address Expressions

Earlier, this chapter points out that addressing modes take a couple generic forms, including the following:

```
VarName[ Reg32 ]
VarName[ Reg32 + offset ]
VarName[ RegNotESP32*scale ]
VarName[ Reg32 + RegNotESP32*scale ]
VarName[ RegNotESP32*scale + offset ]
VarName[ Reg32 + RegNotESP32*scale + offset ]
```

Another legal form, which isn't actually a new addressing mode but simply an extension of the displacement-only addressing mode, is:

```
VarName[ offset ]
```

This latter example computes its effective address by adding the constant offset within the brackets to the variable's address. For example, the instruction `mov(Address[3], al);` loads the AL register with the byte in memory that is 3 bytes beyond the `Address` object (see Figure 3-8).

Always remember that the *offset* value in these examples must be a constant. If `Index` is an `int32` variable, then `Variable[Index]` is not a legal address expression. If you wish to specify an index that varies at runtime, then you must use one of the indexed or scaled-indexed addressing modes.

Another important thing to remember is that the offset in `Address[`*offset*`]` is a byte address. Despite the fact that this syntax is reminiscent of array indexing in a high-level language like C/C++ or Pascal, this does not properly index into an array of objects unless `Address` is an array of bytes.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000077

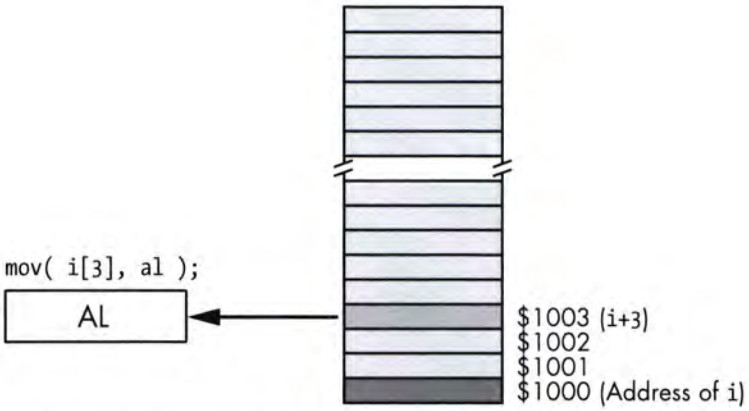


Figure 3-8: Using an address expression to access data beyond a variable

This text will consider an *address expression* to be any legal 80x86 addressing mode that includes a displacement (i.e., variable name) or an offset. In addition to the above forms, the following are also address expressions:

```
[ Reg32 + offset ]
[ Reg32 + RegNotESP32*scale + offset ]
```

This book will *not* consider the following to be address expressions because they do not involve a displacement or offset component:

```
[ Reg32 ]
[ Reg32 + RegNotESP32*scale ]
```

Address expressions are special because those instructions containing an address expression always encode a displacement constant as part of the machine instruction. That is, the machine instruction contains some number of bits (usually 8 or 32) that hold a numeric constant. That constant is the sum of the displacement (i.e., the address or offset of the variable) plus the offset. Note that HLA automatically adds these two values together for you (or subtracts the offset if you use the - rather than + operator in the addressing mode).

Until this point, the offset in all the addressing mode examples has always been a single numeric constant. However, HLA also allows a *constant expression* anywhere an offset is legal. A constant expression consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, modulo, and a wide variety of others. Most address expressions, however, will involve only addition, subtraction, multiplication, and sometimes division. Consider the following example:

```
mov( X[ 2*4+1 ], al );
```

This instruction will move the byte at address `X+9` into the AL register.

The value of an address expression is always computed at compile time, never while the program is running. When HLA encounters the instruction

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000078

above, it calculates 2 * 4 + 1 on the spot and adds this result to the base address of X in memory. HLA encodes this single sum (base address of X plus 9) as part of the instruction; HLA does not emit extra instructions to compute this sum for you at runtime (which is good, because doing so would be less efficient). Because HLA computes the value of address expressions at compile time, all components of the expression must be constants because HLA cannot know the runtime value of a variable while it is compiling the program.

Address expressions are useful for accessing the data in memory beyond a variable, particularly when you've used the byte, word, dword, and so on statements in a static or readonly section to tack on additional bytes after a data declaration. For example, consider the program in Listing 3-1.

```
program adrsExpressions;
#include( "stdlib.hhf" )
static
  i: int8; @nostorage;
     byte 0, 1, 2, 3;

begin adrsExpressions;

  stdout.put
  (
    "i[0]=", i[0], nl,
    "i[1]=", i[1], nl,
    "i[2]=", i[2], nl,
    "i[3]=", i[3], nl
  );

end adrsExpressions;
```

Listing 3-1: Demonstration of address expressions

The program in Listing 3-1 will display the four values 0, 1, 2, and 3 as though they were array elements. This is because the value at the address of i is 0 (this program declares i using the @nostorage option, so i is the address of the next object in the static section, which just happens to be the value 0 appearing as part of the byte statement). The address expression i[1] tells HLA to fetch the byte appearing at i's address plus 1. This is the value 1, because the byte statement in this program emits the value 1 to the static segment immediately after the value 0. Likewise for i[2] and i[3], this program displays the values 2 and 3.

3.6 Type Coercion

Although HLA is fairly loose when it comes to type checking, HLA does ensure that you specify appropriate operand sizes to an instruction. For example, consider the following (incorrect) program:

```
program hasErrors;
static
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000079

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of x until it computes the product of y and x. As your expressions increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

```
w := (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (that is, the two subexpressions with the highest precedence) first and set their values aside. When you've computed the values for both subexpressions, you can compute their sum. One way to deal with a complex expression like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, you can convert the single expression above into the following sequence:

```
temp1 := a + b;
temp2 := y + z;
w := temp1 * temp2;
```

Because converting simple expressions to assembly language is quite easy, it's now a snap to compute the former complex expression in assembly. The code is:

```
mov( a, eax );
add( b, eax );
mov( eax, temp1 );
mov( y, eax );
add( z, eax );
mov( eax, temp2 );
mov( temp1, eax );
intmul( temp2, eax );
mov( eax, w );
```

Of course, this code is grossly inefficient, and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results, this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000081

Here's yet another example:

```
x := (y + z) * (a - b) / 10;
```

This can be converted to a set of four simple expressions:

```
temp1 := (y + z)
temp2 := (a - b)
temp1 := temp1 * temp2
X := temp1 / 10
```

You can convert these four simple expressions into the following assembly language statements:

```
mov( y, eax );          // Compute eax = y + z
add( z, eax );
mov( a, ebx );          // Compute ebx = a - b
sub( b, ebx );
imul( ebx, eax );       // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that you should attempt to keep temporary values in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the result. Because you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly simple. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

6.2.4 Commutative Operators

If *op* represents some operator, that operator is *commutative* if the following relationship is always true:

```
(A op B) = (B op A)
```

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial, and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000082

always check to see if there is a better sequence you can use to improve the size or speed of your code. Tables 6-6 and 6-7, respectively, list the commutative and noncommutative operators you typically find in high-level languages.

Table 6-6: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
and	&& or &	Logical or bitwise and
or	|| or |	Logical or bitwise or
xor	^	(Logical or) bitwise exclusive-or
=	==	Equality
<>	!=	Inequality

Table 6-7: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or div	/	Division
mod	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

6.3 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

```
b := ((x = y) and (a <= c)) or ((z - a) <> 5);
```

b is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose (thus, HLA also allocates a whole byte for a boolean variable). With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like C) choose 0 to represent false and anything else to represent true. Some people prefer to represent true and false with 1 and 0 (respectively) and not allow any other values. Others select all

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000083

1 bits ($FFFF_FFFF, $FFFF, or $FF) for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their advantages and drawbacks.

Using only 0 and 1 to represent false and true offers two very big advantages: (1) The setcc instructions produce these results, so this scheme is compatible with those instructions; (2) the 80x86 logical instructions (and, or, xor, and, to a lesser extent, not) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

```
// c = a AND b;

     mov( a, al );
     and( b, al );
     mov( al, c );

// c = a OR b;

     mov( a, al );
     or( b, al );
     mov( al, c );

// c = a XOR b;

     mov( a, al );
     xor( b, al );
     mov( al, c );

// b = NOT a;

     mov( a, al );     // Note that the NOT instruction does not
     not( al );        // properly compute al = NOT al by itself.
     and( 1, al );     // I.e., (NOT 0) does not equal one. The AND
     mov( al, b );     // instruction corrects this problem.

     mov( a, al );     // Another way to do b = NOT a;
     xor( 1, al );     // Inverts bit 0.
     mov( al, b );
```

Note, as pointed out above, that the not instruction will not properly compute logical negation. The bitwise not of 0 is $FF and the bitwise not of 1 is $FE. Neither result is 0 or 1. However, by anding the result with 1 you get the proper result. Note that you can implement the not operation more efficiently using the xor(1, ax); instruction because it affects only the L.O. bit.

As it turns out, using 0 for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: You cannot use the 80x86 and, or, xor, and not instructions to implement the boolean operations of the same name. Consider the two values $55 and $AA. They're both nonzero so they both represent the value true. However, if you logically and $55 and $AA together using the 80x86

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000084

and instruction, the result is 0. True and true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing boolean operations.

A system that uses nonzero values to represent true and 0 to represent false is an *arithmetic logical system.* A system that uses the two distinct values like 0 and 1 to represent false and true is called a *boolean logical system,* or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression

```
b := ((x = y) and (a <= d)) or ((z - a) <> 5);
```

The simple expressions resulting from this expression might be:

```
mov( x, eax );
cmp( y, eax );
sete( al );        // al := x = y;

mov( a, ebx );
cmp( ebx, d );
setle( bl );      // bl := a <= d;
and( al, bl );    // bl := (x = y) and (a <= d);

mov( z, eax );
sub( a, eax );
cmp( eax, 5 );
setne( al );
or( bl, al );      // al := ((x = y) and (a <= d)) or ((z - a) <> 5);
mov( al, b );
```

When working with boolean expressions don't forget that you might be able to optimize your code by simplifying those boolean expressions. You can use algebraic transformations to help reduce the complexity of an expression. In the chapter on control structures, you'll also see how to use control flow to calculate a boolean result. This is generally quite a bit more efficient than using *complete boolean evaluation* as the examples in this section teach.

6.4 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well-written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well known and commonly found in assembly language programs. Some of them are little more than tricks, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000085

6.4.1 Multiplying without mul, imul, or intmul

When multiplying by a constant, you can sometimes write faster code by using shifts, additions, and subtractions in place of multiplication instructions.

Remember, a shl instruction computes the same result as multiplying the specified operand by 2. Shifting to the left two bit positions multiplies the operand by 4. Shifting to the left three bit positions multiplies the operand by 8. In general, shifting an operand to the left n bits multiplies it by 2^n. You can multiply any value by some constant using a series of shifts and additions or shifts and subtractions. For example, to multiply the AX register by 10, you need only multiply it by 8 and then add in two times the original value. That is, 10 * ax = 8 * ax + 2 * ax. The code to accomplish this is:

```
shl( 1, ax );          // Multiply ax by two.
mov( ax, bx);          // Save 2*ax for later.
shl( 2, ax );          // Multiply ax by eight (*4 really,
                       // but ax contains *2).
add( bx, ax );         // Add in ax*2 to ax*8 to get ax*10.
```

Many x86 processors can multiply the AX register (or just about any register, for that matter) by various constant values much faster by using shl than by using the mul instruction. This may seem hard to believe because it takes only one instruction to compute this product:

```
intmul( 10, ax );
```

However, if you look at the instruction timings, the shift and add example above requires fewer clock cycles on many processors in the 80x86 family than the mul instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by 7:

```
mov( eax, ebx );          // Save eax * 1
shl( 3, eax );            // eax = eax * 8
sub( ebx, eax );          // eax * 8 - eax * 1 is eax * 7
```

A common error beginning assembly language programmers make is subtracting or adding 1 or 2 rather than eax * 1 or eax * 2. The following does not compute eax * 7:

```
shl( 3, eax );
sub( 1, eax );
```

It computes (8 * eax) - 1, something entirely different (unless, of course, EAX = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000086

You can also use the lea instruction to compute certain products. The trick is to use the scaled index addressing modes. The following examples demonstrate some simple cases:

```
lea( eax, [ecx][ecx] );        // eax := ecx * 2
lea( eax, [eax][eax*2] );      // eax := eax * 3
lea( eax, [eax*4] );           // eax := eax * 4
lea( eax, [ebx][ebx*4] );      // eax := ebx * 5
lea( eax, [eax*8] );           // eax := eax * 8
lea( eax, [edx][edx*8] );      // eax := edx * 9
```

6.4.2 Division Without div or idiv

Just as the shl instruction is useful for simulating a multiplication by a power of 2, the shr and sar instructions can simulate a division by a power of 2. Unfortunately, you cannot easily use shifts, additions, and subtractions to perform a division by an arbitrary constant. Therefore, keep in mind that this trick is useful only when dividing by powers of 2. Also, don't forget that the sar instruction rounds towards negative infinity rather than toward 0; this is not the way the idiv instruction operates (it rounds toward 0).

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. Because the multiply instruction is faster than the divide instruction, multiplying by a reciprocal is usually faster than division.

Now you're probably wondering, "How does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by 1/10, there is no way you can load the value 1/10 into an 80x86 integer register prior to performing the multiplication. However, we could multiply 1/10 by 10, perform the multiplication, and then divide the result by 10 to get the final result. Of course, this wouldn't buy you anything; in fact, it would make things worse because you're now doing a multiplication by 10 as well as a division by 10. However, suppose you multiply 1/10 by 65,536 (6,553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation, and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides AX by 10:

```
mov( 6554, dx );          // 6,554 = round( 65,536/10 )
mul( dx, ax );
```

This code leaves AX/10 in the DX register.

To understand how this works, consider what happens when you multiply AX by 65,536 ($1_0000). This simply moves AX into DX and sets AX to 0 (a multiply by $1_0000 is equivalent to a shift left by 16 bits). Multiplying by 6,554 (65,536 divided by 10) puts AX divided by 10 into the DX register. Because mul is faster than div, this technique runs a little faster than using a division.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000087

Multiplying by a reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal pays off only if you perform the division many, many times (by the same value).

6.4.3 Implementing Modulo-N Counters with and

If you want to implement a counter variable that counts up to $2^n - 1$ and then resets to 0, simply use the following code:

```
inc( CounterVar );
and( nBits, CounterVar );
```

where nBits is a binary value containing n bits containing ones right justified in the number. For example, to create a counter that cycles between 0 and 15 ($2^4 - 1$), you could use the following:

```
inc( CounterVar );
and( %00001111, CounterVar );
```

6.5 Floating-Point Arithmetic

When the 8086 CPU first appeared in the late 1970s, semiconductor technology was not to the point where Intel could put floating-point instructions directly on the 8086 CPU. Therefore, Intel devised a scheme whereby it could use a second chip to perform the floating-point calculations—the floating-point unit (or FPU).[1] By the release of the Intel Pentium chip, semiconductor technology had advanced to the point that the FPU was fully integrated onto the 80x86 CPU. Therefore, almost all modern 80x86 CPU devices fully support floating-point arithmetic directly on the CPU.

6.5.1 FPU Registers

The 80x86 FPUs add 13 registers to the 80x86: eight floating-point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general-purpose register set insofar as all floating-point calculations take place in these registers. The control register contains bits that let you decide how the FPU handles certain degenerate cases like rounding of inaccurate computations; it also contains bits that control precision and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating-point flags that describe the state of the FPU. The tag register contains several groups of bits that determine the state of the value in each of the eight floating-point data registers. The instruction and data pointer registers

[1] Intel has also referred to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000088

contain certain state information about the last floating-point instruction executed. We will not consider the last three registers here; see the Intel documentation for more details.

6.5.1.1 FPU Data Registers

The FPUs provide eight 80-bit data registers organized as a stack. This is a significant departure from the organization of the general-purpose registers on the 80x86 CPU. HLA refers to these registers as ST0, ST1, . . . ST7.

The biggest difference between the FPU register set and the 80x86 register set is the stack organization. On the 80x86 CPU, the AX register is always the AX register, no matter what happens. On the FPU, however, the register set is an eight-element stack of 80-bit floating-point values (see Figure 6-1).

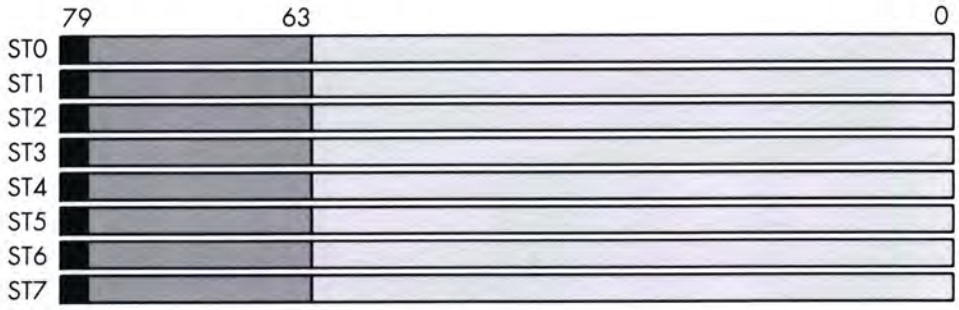


Figure 6-1: FPU floating-point register stack

ST0 refers to the item on the top of the stack, ST1 refers to the next item on the stack, and so on. Many floating-point instructions push and pop items on the stack; therefore, ST1 will refer to the previous contents of ST0 after you push something onto the stack. It will take some thought and practice to get used to the fact that the register numbers change, but this is an easy problem to overcome.

6.5.1.2 The FPU Control Register

When Intel designed the 80x87 (and, essentially, the IEEE floating-point standard), there were no standards in floating-point hardware. Different (mainframe and mini) computer manufacturers all had different and incompatible floating-point formats. Unfortunately, several applications had been written taking into account the idiosyncrasies of these different floating-point formats. Intel wanted to design an FPU that could work with the majority of the software out there (keep in mind that the IBM-PC was three to four years away when Intel began designing the 8087, so Intel couldn't rely on that "mountain" of software available for the PC to make its chip popular). Unfortunately, many of the features found in these older floating-point formats were mutually incompatible. For example, in some floating-point systems rounding would occur when there was insufficient precision; in others, truncation would occur. Some applications would work with one floating-point system but not with the other. Intel wanted as many applications as possible to

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000089

work with as few changes as possible on its 80x87 FPUs, so it added a special register, the FPU *control register*, that lets the user choose one of several possible operating modes for the FPU.

The 80x87 control register contains 16 bits organized as shown in Figure 6-2.

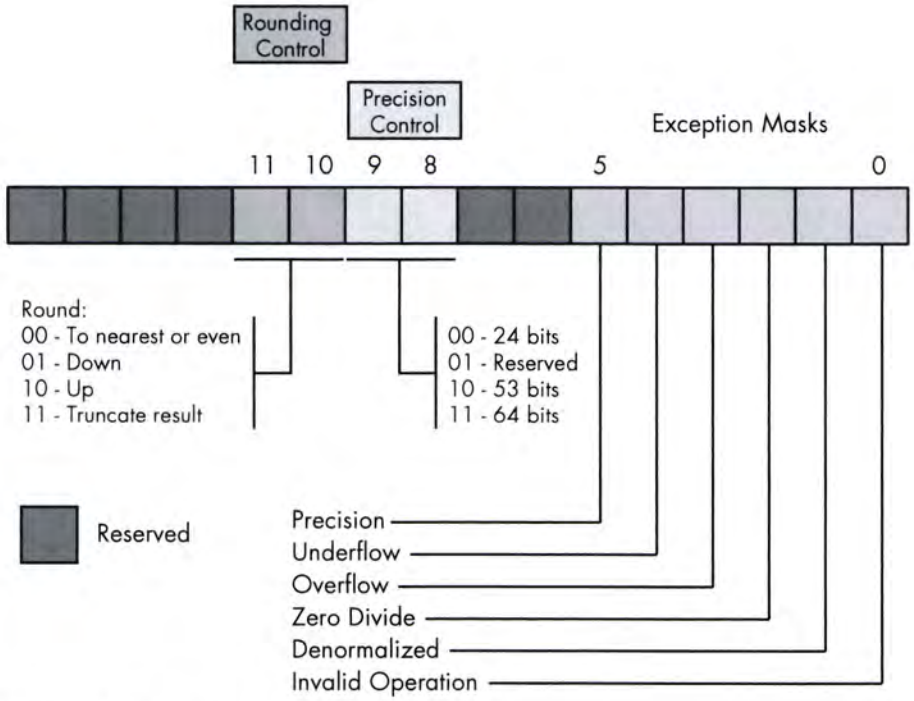


Figure 6-2: FPU control register

Bits 10 and 11 of the FPU control register provide rounding control according to the values appearing in Table 6-8.

Table 6-8: Rounding Control

Bits 10 & 11	Function
00	To nearest or even
01	Round down
10	Round up
11	Truncate

The 00 setting is the default. The FPU rounds up values above one-half of the least significant bit. It rounds down values below one-half of the least significant bit. If the value below the least significant bit is exactly one-half of the least significant bit, then the FPU rounds the value toward the value whose least significant bit is 0. For long strings of computations, this provides a reasonable, automatic way to maintain maximum precision.

The round-up and round-down options are present for those computations where it is important to keep track of the accuracy during a computation. By setting the rounding control to round down and performing the operation,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000090

then repeating the operation with the rounding control set to round up, you can determine the minimum and maximum ranges between which the true result will fall.

The truncate option forces all computations to truncate any excess bits during the computation. You will rarely use this option if accuracy is important to you. However, if you are porting older software to the FPU, you might use this option to help when porting the software. One place where this option is extremely useful is when converting a floating-point value to an integer. Because most software expects floating-point-to-integer conversions to truncate the result, you will need to use the truncation/rounding mode to achieve this.

Bits 8 and 9 of the control register specify the precision during computation. This capability is provided to allow compatibility with older software as required by the IEEE 754 standard. The precision control bits use the values in Table 6-9.

Table 6-9: Mantissa Precision Control Bits

Bits 8 & 9	Precision Control
00	24 bits
01	Reserved
10	53 bits
11	64 bits

Some CPUs may operate faster with floating-point values whose precision is 53 bits (i.e., 64-bit floating-point format) rather than 64 bits (i.e., 80-bit floating-point format). Please see the documentation for your specific processor for details. Generally, the CPU defaults these bits to %11 to select the 64-bit mantissa precision.

Bits 0..5 are the *exception masks*. These are similar to the interrupt enable bit in the 80x86's flags register. If these bits contain a 1, the corresponding condition is ignored by the FPU. However, if any bit contains 0, and the corresponding condition occurs, then the FPU immediately generates an interrupt so the program can handle the degenerate condition (typically, this would wind up raising an HLA exception; see the *excepts.hhf* header file for the exception values).

Bit 0 corresponds to an invalid operation error. This generally occurs as the result of a programming error. Situations that raise the invalid operation exception (`ex.fInvalidOperation`) include pushing more than eight items onto the stack or attempting to pop an item off an empty stack, taking the square root of a negative number, or loading a nonempty register.

Bit 1 masks the *denormalized* interrupt that occurs whenever you try to manipulate denormalized values. Denormalized exceptions occur when you load arbitrary extended-precision values into the FPU or work with very small numbers just beyond the range of the FPU's capabilities. Normally, you would probably *not* enable this exception. If you enable this exception and the FPU generates this interrupt, the HLA runtime system raises the `ex.fDenormal` exception.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000091

Bit 2 masks the *zero divide* exception. If this bit contains 0, the FPU will generate an interrupt if you attempt to divide a nonzero value by 0. If you do not enable the zero division exception, the FPU will produce `NaN` (not a number) whenever you perform a zero division. It's probably a good idea to enable this exception by programming a 0 into this bit. Note that if your program generates this interrupt, the HLA runtime system will raise the `ex.fDivByZero` exception.

Bit 3 masks the *overflow* exception. The FPU will raise the overflow exception if a calculation overflows or if you attempt to store a value that is too large to fit into the destination operand (for example, storing a large extended-precision value into a single-precision variable). If you enable this exception and the FPU generates this interrupt, the HLA runtime system raises the `ex.fOverflow` exception.

Bit 4, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the destination operand. Like overflow, this exception can occur whenever you store a small extended-precision value into a smaller variable (single or double precision) or when the result of a computation is too small for extended precision. If you enable this exception and the FPU generates this interrupt, the HLA runtime system raises the `ex.fUnderflow` exception.

Bit 5 controls whether the *precision* exception can occur. A precision exception occurs whenever the FPU produces an imprecise result, generally the result of an internal rounding operation. Although many operations will produce an exact result, many more will not. For example, dividing 1 by 10 will produce an inexact result. Therefore, this bit is usually 1 because inexact results are very common. If you enable this exception and the FPU generates this interrupt, the HLA runtime system raises the `ex.InexactResult` exception.

Bits 6..7 and 12..15 in the control register are currently undefined and reserved for future use (bits 7 and 12 were valid on older FPUs but are no longer used).

The FPU provides two instructions, `fldcw` (load control word) and `fstcw` (store control word), that let you load and store the contents of the control register. The single operand to these instructions must be a 16-bit memory location. The `fldcw` instruction loads the control register from the specified memory location. `fstcw` stores the control register into the specified memory location. The syntax for these instructions is:

```
fldcw( mem16 );
fstcw( mem16 );
```

Here's some example code that sets the rounding control to "truncate result" and sets the rounding precision to 24 bits:

```
static
    fcw16: word;
        .
        .
        .
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000092

```
fstcw( fcw16 );
mov( fcw16, ax );
and( $f0ff, ax );        // Clears bits 8-11.
or( $0c00, ax );         // Rounding control=%11, Precision = %00.
mov( ax, fcw16 );
fldcw( fcw16 );
```

6.5.1.3 The FPU Status Register

The FPU status register provides the status of the FPU at the instant you read it. The `fstsw` instruction stores the16-bit floating-point status register into a word variable. The status register is a 16-bit register; its layout appears in Figure 6-3.

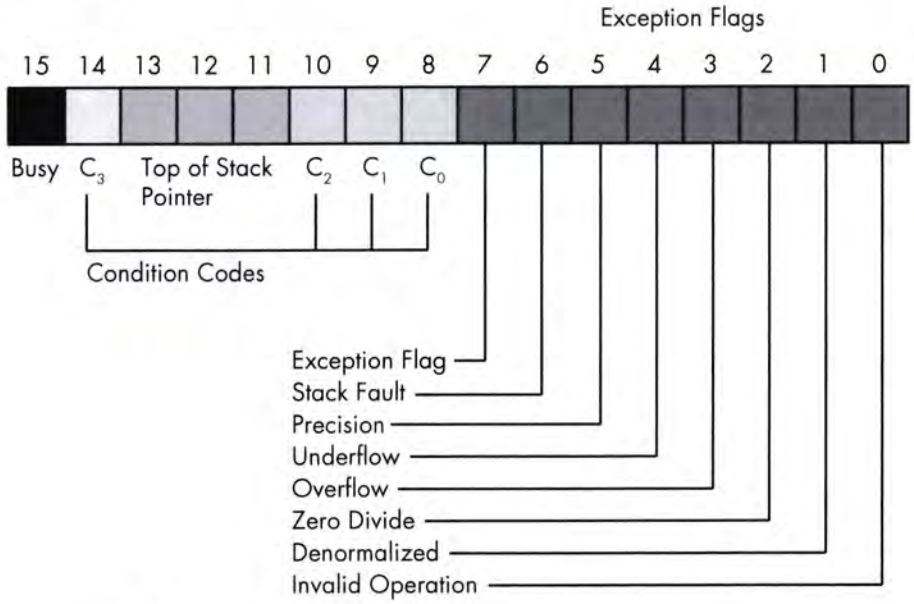


Figure 6-3: The FPU status register

Bits 0 through 5 are the exception flags. These bits appear in the same order as the exception masks in the control register. If the corresponding condition exists, then the bit is set. These bits are independent of the exception masks in the control register. The FPU sets and clears these bits regardless of the corresponding mask setting.

Bit 6 indicates a *stack fault.* A stack fault occurs whenever there is a stack overflow or underflow. When this bit is set, the C_1 condition code bit determines whether there was a stack overflow ($C_1 = 1$) or stack underflow ($C_1 = 0$) condition.

Bit 7 of the status register is set if *any* error condition bit is set. It is the logical or of bits 0 through 5. A program can test this bit to quickly determine if an error condition exists.

Bits 8, 9, 10, and 14 are the coprocessor condition code bits. Various instructions set the condition code bits, as shown in Tables 6-10 and 6-11, respectively.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000093

Table 6-10: FPU Condition Code Bits (X = "Don't care")

Instruction	Condition Code Bits				Condition
	C_3	C_2	C_1	C_0	
fcom	0	0	X	0	ST > source
fcomp	0	0	X	1	ST < source
fcompp	1	0	X	0	ST = source
ficom ficomp	1	1	X	1	ST or source undefined
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is 0 (+ or –)
	1	1	X	1	ST is uncomparable
fxam	0	0	0		+ Unnormalized
	0	0	1	0	– Unnormalized
	0	1	0	0	+ Normalized
	0	1	1	0	– Normalized
	1	0	0	0	+ 0
	1	0	1	0	– 0
	1	1	0	0	+ Denormalized
	1	1	1	0	– Denormalized
	0	0	0	1	+ NaN
	0	0	1	1	– NaN
	0	1	0	1	+ Infinity
	0	1	1	1	– Infinity
	1	X	X	1	Empty register
fucom	0	0	X	0	ST > source
fucomp	0	0	X	1	ST < source
fucompp	1	0	X	0	ST = source
	1	1	X	1	Unordered

Table 6-11: Condition Code Interpretations (X = "Don't care")

Instruction	Condition Code Bits				Condition
	C_3	C_2	C_1	C_0	
fcom	0	0	X	0	ST > source
fcomp	0	0	X	1	ST < source
fcompp	1	0	X	0	ST = source
ficom ficomp	1	1	X	1	ST or source undefined
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is 0 (+ or –)
	1	1	X	1	ST is uncomparable

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000094

Table 6-11: Condition Code Interpretations (X = "Don't care") (continued)

Instruction	Condition Code Bits				Condition
	C_3	C_2	C_1	C_0	
fxam	0	0	0	1	+ Unnormalized
	0	0	1	0	– Unnormalized
	0	1	0	0	+ Normalized
	0	1	1	0	– Normalized
	1	0	0	0	+ 0
	1	0	1	0	– 0
	1	1	0	0	+ Denormalized
	1	1	1	0	– Denormalized
	0	0	0	1	+ NaN
	0	0	1	1	– NaN
	0	1	0	1	+ Infinity
	0	1	1	1	– Infinity
	1	X	X	1	Empty register
fucom	0	0	X	0	ST > source
fucomp	0	0	X	1	ST < source
fucompp	1	0	X	0	ST = source
	1	1	X	1	Unordered

Bits 11–13 of the FPU status register provide the register number of the top of stack. During computations, the FPU adds (modulo-8) the *logical* register numbers supplied by the programmer to these three bits to determine the *physical* register number at runtime.

Bit 15 of the status register is the *busy* bit. It is set whenever the FPU is busy. This bit is a historical artifact from the days when the FPU was a separate chip; most programs will have little reason to access this bit.

6.5.2 FPU Data Types

The FPU supports seven different data types: three integer types, a packed decimal type, and three floating-point types. The integer type supports 64-bit integers, although it is often faster to do the 64-bit arithmetic using the integer unit of the CPU (see Chapter 8). Certainly it is faster to do 16-bit and 32-bit integer arithmetic using the standard integer registers. The packed decimal type provides a 17-digit signed decimal (BCD) integer. The primary purpose of the BCD format is to convert between strings and floating-point values. The remaining three data types are the 32-bit, 64-bit, and 80-bit floating-point data types. The 80x87 data types appear in Figures 6-4, 6-5, and 6-6.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000095

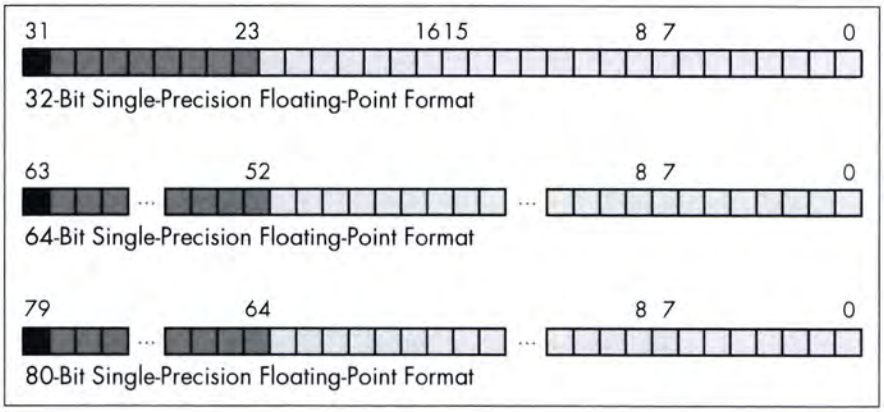


Figure 6-4: FPU floating-point formats

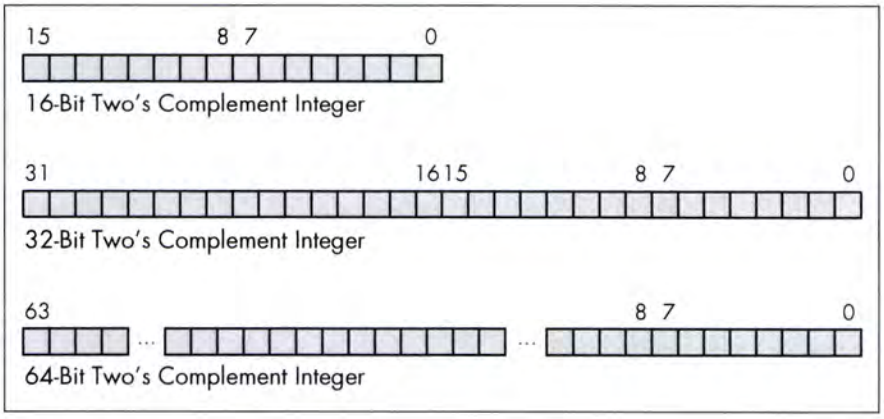


Figure 6-5: FPU integer formats

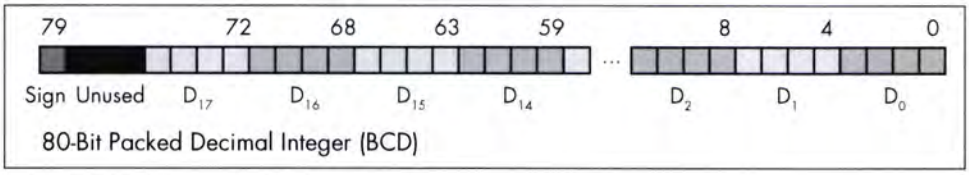


Figure 6-6: FPU packed decimal format

The FPU generally stores values in a *normalized* format. When a floating-point number is normalized, the H.O. bit of the mantissa is always 1. In the 32- and 64-bit floating-point formats, the FPU does not actually store this bit; the FPU always assumes that it is 1. Therefore, 32- and 64-bit floating-point numbers are always normalized. In the extended-precision 80-bit floating-point format, the FPU does *not* assume that the H.O. bit of the mantissa is 1; the H.O. bit of the mantissa appears as part of the string of bits.

Normalized values provide the greatest precision for a given number of bits. However, there are a large number of nonnormalized values that we *cannot* represent with the 80-bit format. These values are very close to 0 and represent the set of values whose mantissa H.O. bit is not 0. The FPUs support a special 80-bit form known as *denormalized* values. Denormalized values allow the FPU to encode very small values it cannot encode using normalized values, but denormalized values offer fewer bits of precision than normalized

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000096

values. Therefore, using denormalized values in a computation may introduce some slight inaccuracy into a computation. Of course, this is always better than underflowing the denormalized value to 0 (which could make the computation even less accurate), but you must keep in mind that if you work with very small values you may lose some accuracy in your computations. Note that the FPU status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

6.5.3 The FPU Instruction Set

The FPU adds many instructions to the 80x86 instruction set. We can classify these instructions as data movement instructions, conversions, arithmetic instructions, comparisons, constant instructions, transcendental instructions, and miscellaneous instructions. The following sections describe each of the instructions in these categories.

6.5.4 FPU Data Movement Instructions

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are `fld`, `fst`, `fstp`, and `fxch`. The `fld` instruction always pushes its operand onto the floating-point stack. The `fstp` instruction always pops the top of stack after storing the top of stack (TOS). The remaining instructions do not affect the number of items on the stack.

6.5.4.1 The fld Instruction

The `fld` instruction loads a 32-bit, 64-bit, or 80-bit floating-point value onto the stack. This instruction converts 32- and 64-bit operands to an 80-bit extended-precision value before pushing the value onto the floating-point stack.

The `fld` instruction first decrements the TOS pointer (bits 11–13 of the status register) and then stores the 80-bit value in the physical register specified by the new TOS pointer. If the source operand of the FLD instruction is a floating-point data register, `sti`, then the actual register the FPU uses for the load operation is the register number *before* decrementing the TOS pointer. Therefore, `fld( st0 );` duplicates the value on the top of the stack.

The `fld` instruction sets the stack fault bit if stack overflow occurs. It sets the denormalized exception bit if you load an 80-bit denormalized value. It sets the invalid operation bit if you attempt to load an empty floating-point register onto the top of stack (or perform some other invalid operation).

Here are some examples:

```
fld( st1 );
fld( real32_variable );
fld( real64_variable );
fld( real80_variable );
fld( (type real64 [ebx]) );
fld( real_constant );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000097

Note that there is no way to directly load a 32-bit integer register onto the floating-point stack, even if that register contains a real32 value. To accomplish this, you must first store the integer register into a memory location; then you can push that memory location onto the FPU stack using the fld instruction. For example:

```
mov( eax, tempReal32 );     // Save real32 value in eax to memory.
fld( tempReal32 );          // Push that real value onto the FPU stack.
```

Note that loading a constant via fld is actually an HLA extension. The FPU doesn't support this instruction type. HLA creates a real80 object in the constants segment and uses the address of this memory object as the true operand for fld.

6.5.4.2 The fst and fstp Instructions

The fst and fstp instructions copy the value on the top of the floating-point stack to another floating-point register or to a 32-, 64-, or 80-bit memory variable. When copying data to a 32- or 64-bit memory variable, the FPU rounds the 80-bit extended-precision value on the top of stack to the smaller format as specified by the rounding control bits in the FPU control register.

The fstp instruction pops the value off the top of the stack when moving it to the destination location. It does this by incrementing the TOS pointer in the status register after accessing the data in ST0. If the destination operand is a floating-point register, the FPU stores the value at the specified register number *before* popping the data off the top of the stack.

Executing an fstp(st0); instruction effectively pops the data off the top of stack with no data transfer. Here are some examples:

```
fst( real32_variable );
fst( real64_variable );
fst( realArray[ ebx*8 ] );
fst( st2 );
fstp( st1 );
```

The last example above effectively pops ST1 while leaving ST0 on the top of stack.

The fst and fstp instructions will set the stack exception bit if a stack underflow occurs (attempting to store a value from an empty register stack). They will set the precision bit if there is a loss of precision during the store operation (this will occur, for example, when storing an 80-bit extended-precision value into a 32- or 64-bit memory variable and some bits are lost during conversion). They will set the underflow exception bit when storing an 80-bit value into a 32- or 64-bit memory variable, but the value is too small to fit into the destination operand. Likewise, these instructions will set the

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000098

overflow exception bit if the value on the top of stack is too big to fit into a 32- or 64-bit memory variable. The `fst` and `fstp` instructions set the denormalized flag when you try to store a denormalized value into an 80-bit register or variable.[2] They set the invalid operation flag if an invalid operation (such as storing into an empty register) occurs. Finally, these instructions set the C_1 condition bit if rounding occurs during the store operation (this occurs only when storing into a 32- or 64-bit memory variable and you have to round the mantissa to fit into the destination).

NOTE *Because of an idiosyncrasy in the FPU instruction set related to the encoding of the instructions, you cannot use the `fst` instruction to store data into a `real80` memory variable. You may, however, store 80-bit data using the `fstp` instruction.*

6.5.4.3 The fxch Instruction

The `fxch` instruction exchanges the value on the top of stack with one of the other FPU registers. This instruction takes two forms: one with a single FPU register as an operand and the second without any operands. The first form exchanges the top of stack with the specified register. The second form of `fxch` swaps the top of stack with ST1.

Many FPU instructions, for example, `fsqrt`, operate only on the top of the register stack. If you want to perform such an operation on a value that is not on the top of stack, you can use the `fxch` instruction to swap that register with TOS, perform the desired operation, and then use the `fxch` to swap the TOS with the original register. The following example takes the square root of ST2:

```
fxch( st2 );
fsqrt();
fxch( st2 );
```

The `fxch` instruction sets the stack exception bit if the stack is empty. It sets the invalid operation bit if you specify an empty register as the operand. This instruction always clears the C_1 condition code bit.

6.5.5 Conversions

The FPU performs all arithmetic operations on 80-bit real quantities. In a sense, the `fld` and `fst/fstp` instructions are conversion instructions because they automatically convert between the internal 80-bit real format and the 32- and 64-bit memory formats. Nonetheless, we'll simply classify them as data movement operations, rather than conversions, because they are moving real values to and from memory. The FPU provides six other instructions that convert to or from integer or binary-coded decimal (BCD) format when moving data. These instructions are `fild`, `fist`, `fistp`, `fisttp`, `fbld`, and `fbstp`.

[2] Storing a denormalized value into a 32- or 64-bit memory variable will always set the underflow exception bit.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000099

6.5.5.1 The fild Instruction

The fild (integer load) instruction converts a 16-, 32-, or 64-bit two's complement integer to the 80-bit extended-precision format and pushes the result onto the stack. This instruction always expects a single operand. This operand must be the address of a word, double-word, or quad-word integer variable. You cannot specify one of the 80x86's 16- or 32-bit general-purpose registers. If you want to push the value of an 80x86 general-purpose register onto the FPU stack, you must first store it into a memory variable and then use fild to push that memory variable.

The fild instruction sets the stack exception bit and C_1 (accordingly) if stack overflow occurs while pushing the converted value. Look at these examples:

```
fild( word_variable );
fild( dword_val[ ecx*4 ] );
fild( qword_variable );
fild( (type int64 [ebx]) );
```

6.5.5.2 The fist, fistp, and fisttp Instructions

The fist, fistp, and fisttp instructions convert the 80-bit extended-precision variable on the top of stack to a 16-, 32-, or 64-bit integer and store the result away into the memory variable specified by the single operand. The fist and fistp instructions convert the value on TOS to an integer according to the rounding setting in the FPU control register (bits 10 and 11). The fisttp instruction always does the conversion using the truncation mode. As for the fild instruction, the fist, fistp, and fisttp instructions will not let you specify one of the 80x86's general-purpose 16- or 32-bit registers as the destination operand.

The fist instruction converts the value on the top of stack to an integer and then stores the result; it does not otherwise affect the floating-point register stack. The fistp and fisttp instructions pop the value off the floating-point register stack after storing the converted value.

These instructions set the stack exception bit if the floating-point register stack is empty (this will also clear C_1). They set the precision (imprecise operation) and C_1 bits if rounding occurs (that is, if there is any fractional component to the value in ST0). These instructions set the underflow exception bit if the result is too small (that is, less than 1 but greater than 0 or less than 0 but greater than −1). Here are some examples:

```
fist( word_var[ ebx*2 ] );
fist( qword_var );
fisttp( dword_var );
fistp( dword_var );
```

Don't forget that the fist and fistp instructions use the rounding control settings to determine how they will convert the floating-point data to an integer during the store operation. Be default, the rounding control is usually set

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000100

to "round" mode; yet most programmers expect fist/fistp to truncate the decimal portion during conversion. If you want fist/fistp to truncate floating-point values when converting them to an integer, you will need to set the rounding control bits appropriately in the floating-point control register (or use the fisttp instruction to truncate the result regardless of the rounding control bits). Here's an example:

```
static
    fcw16:          word;
    fcw16_2:        word;
    IntResult:      int32;
        .
        .
        .
        fstcw( fcw16 );
        mov( fcw16, ax );
        or( $0c00, ax );        // Rounding control=%11 (truncate).
        mov( ax, fcw16_2 );     // Store into memory and reload the ctrl word.
        fldcw( fcw16_2 );

        fistp( IntResult );     // Truncate ST0 and store as int32 object.

        fldcw( fcw16 );         // Restore original rounding control.
```

6.5.5.3 The fbld and fbstp Instructions

The fbld and fbstp instructions load and store 80-bit BCD values. The fbld instruction converts a BCD value to its 80-bit extended-precision equivalent and pushes the result onto the stack. The fbstp instruction pops the extended-precision real value on TOS, converts it to an 80-bit BCD value (rounding according to the bits in the floating-point control register), and stores the converted result at the address specified by the destination memory operand. Note that there is no fbst instruction.

The fbld instruction sets the stack exception bit and C_1 if stack overflow occurs. It sets the invalid operation bit if you attempt to load an invalid BCD value. The fbstp instruction sets the stack exception bit and clears C_1 if stack underflow occurs (the stack is empty). It sets the underflow flag under the same conditions as fist and fistp. Look at these examples:

```
// Assuming fewer than 8 items on the stack, the following
// code sequence is equivalent to an fbst instruction:

        fld( st0 );
        fbstp( tbyte_var );

// The following example easily converts an 80-bit BCD value to
// a 64-bit integer:

        fbld( tbyte_var );
        fist( qword_var );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000101

These two instructions are especially useful for converting between string and floating-point formats. See the floating-point-to-string and string-to-floating-point conversion routines in the HLA Standard Library for more details.

6.5.6 Arithmetic Instructions

The arithmetic instructions make up a small but important subset of the FPU's instruction set. These instructions fall into two general categories: those that operate on real values and those that operate on a real and an integer value.

6.5.6.1 The fadd and faddp Instructions

These two instructions take the following forms:

```
fadd()
faddp()
fadd( st0, sti );
fadd( sti, st0 );
faddp( st0, sti );
fadd( mem_32_64 );
fadd( real_constant );
```

The `fadd` instruction, with no operands, adds the value in ST0 to the value in ST1 and stores the result into ST1. The `faddp` instruction (with no operands) pops the two values on the top of stack, adds them, and pushes their sum back onto the stack.

The next two forms of the `fadd` instruction, those with two FPU register operands, behave like the 80x86's `add` instruction. They add the value in the source register operand to the value in the destination register operand. Note that one of the register operands must be ST0.

The `faddp` instruction with two operands adds ST0 (which must always be the source operand) to the destination operand and then pops ST0. The destination operand must be one of the other FPU registers.

The last form above, `fadd` with a memory operand, adds a 32- or 64-bit floating-point variable to the value in ST0. This instruction will convert the 32- or 64-bit operands to an 80-bit extended-precision value before performing the addition. Note that this instruction does *not* allow an 80-bit memory operand.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Like `fld( real_constant)`, the `fadd( real_constant )` instruction is an HLA extension. Note that it creates a 64-bit variable holding the constant value and emits the `fadd( mem64 )` instruction, specifying the read-only object it creates in the constants segment.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000102

6.5.6.2 The fsub, fsubp, fsubr, and fsurpb Instructions

These four instructions take the following forms:

```
fsub()
fsubp()
fsubr()
fsubrp()

fsub( st0, sti )
fsub( sti, st0 );
fsubp( st0, sti );
fsub( mem_32_64 );
fsub( real_constant );

fsubr( st0, sti )
fsubr( sti, st0 );
fsubrp( st0, sti );
fsubr( mem_32_64 );
fsubr( real_constant );
```

With no operands, the `fsub` instruction subtracts ST0 from ST1 and leaves the result in ST1. With no operands the `fsubp` instruction pops ST0 and ST1 from the register stack, computes `st1 - st0` and then pushes the difference back onto the stack. The `fsubr` and `fsubrp` instructions (reverse subtraction) operate in an almost identical fashion except they compute `st0 - st1`.

With two register operands (*source*, *destination*) the `fsub` instruction computes *destination* := *destination* - *source*. One of the two registers must be ST0. With two registers as operands, the `fsubp` also computes *destination* := *destination* - *source*, and then it pops ST0 off the stack after computing the difference. For the `fsubp` instruction, the source operand must be ST0.

With two register operands, the `fsubr` and `fsubrp` instructions work in a similar fashion to `fsub` and `fsubp`, except they compute *destination* := *source* - *destination*.

The `fsub(` *mem* `)` and `fsubr(` *mem* `)` instructions accept a 32- or 64-bit memory operand. They convert the memory operand to an 80-bit extended-precision value and subtract this from ST0 (`fsub`) or subtract ST0 from this value (`fsubr`) and store the result back into ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

NOTE *The instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. HLA generates a constant segment memory object initialized with the constant's value.*

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000103

6.5.6.3 The fmul and fmulp Instructions

The `fmul` and `fmulp` instructions multiply two floating-point values. These instructions allow the following forms:

```
fmul()
fmulp()

fmul( sti, st0 );
fmul( st0, sti );
fmul( mem_32_64 );
fmul( real_constant );

fmulp( st0, sti );
```

With no operands, `fmul` will compute `st0 * st1` and store the product into ST1. The `fmulp` instruction, with no operands, will pop ST0 and ST1, multiply these values, and push their product back onto the stack. The `fmul` instructions with two register operands compute *`destination := destination * source`*. One of the registers (source or destination) must be ST0.

The `fmulp( st0, sti )` instruction computes `sti := sti * st0` and then pops ST0. This instruction uses the value for ST*i* before popping ST0. The `fmul(` *`mem`* `)` instruction requires a 32- or 64-bit memory operand. It converts the specified memory variable to an 80-bit extended-precision value and then multiplies ST0 by this value.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

NOTE *The instruction that has a real constant as its operand isn't a true FPU instruction. It is an extension provided by HLA (see the note at the end of Section 6.5.6.2 for details).*

6.5.6.4 The fdiv, fdivp, fdivr, and fdivrp Instructions

These four instructions allow the following forms:

```
fdiv()
fdivp()
fdivr()
fdivrp()

fdiv( sti, st0 );
fdiv( st0, sti );
fdivp( st0, sti );

fdivr( sti, st0 );
fdivr( st0, sti );
fdivrp( st0, sti );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000104

```
fdiv( mem_32_64 );
fdivr( mem_32_64 );
fdiv( real_constant );
fdivr( real_constant );
```

With no operands, the fdivp instruction pops ST0 and ST1, computes st1/st0, and pushes the result back onto the stack. The fdiv instruction with no operands computes st1 := st1/st0. The fdivr and fdivrp instructions work in a similar fashion to fdiv and fdivp except that they compute st0/st1 rather than st1/st0.

With two register operands, these instructions compute the following quotients:

```
fdiv( sti, st0 );          // st0 := st0/sti
fdiv( st0, sti );          // sti := sti/st0
fdivp( st0, sti );         // sti := sti/st0 then pop st0
fdivr( st0, sti );         // st0 := st0/sti
fdivrp( st0, sti );        // sti := st0/sti then pop st0
```

The fdivp and fdivrp instructions also pop ST0 after performing the division operation. The value for *i* in these two instructions is computed before popping ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, zero divide, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Note that the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA.

6.5.6.5 The fsqrt Instruction

The fsqrt routine does not allow any operands. It computes the square root of the value on top of stack (TOS) and replaces ST0 with this result. The value on TOS must be 0 or positive; otherwise fsqrt will generate an invalid operation exception.

This instruction can raise the stack, precision, denormalized, and invalid operation exceptions, as appropriate. If rounding occurs during the computation, fsqrt sets the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Here's an example:

```
// Compute z := sqrt(x**2 + y**2);

        fld( x );                  // Load x.
        fld( st0 );                // Duplicate x on TOS.
        fmulp();                   // Compute x**2.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000105

```
        fld( y );               // Load y.
        fld( st0 );             // Duplicate y.
        fmul();                 // Compute y**2.

        faddp();                // Compute x**2 + y**2.
        fsqrt();                // Compute sqrt( x**2 + y**2 ).
        fstp( z );              // Store result away into z.
```

6.5.6.6 The fprem and fprem1 Instructions

The fprem and fprem1 instructions compute a *partial remainder*. Intel designed the fprem instruction before the IEEE finalized its floating-point standard. In the final draft of the IEEE floating-point standard, the definition of fprem was a little different than Intel's original design. Unfortunately, Intel needed to maintain compatibility with the existing software that used the fprem instruction, so it designed a new version to handle the IEEE partial remainder operation, fprem1. You should always use fprem1 in new software; therefore we will discuss only fprem1 here, although you use fprem in an identical fashion.

fprem1 computes the *partial* remainder of st0/st1. If the difference between the exponents of ST0 and ST1 is less than 64, fprem1 can compute the exact remainder in one operation. Otherwise you will have to execute the fprem1 two or more times to get the correct remainder value. The C_2 condition code bit determines when the computation is complete. Note that fprem1 does *not* pop the two operands off the stack; it leaves the partial remainder in ST0 and the original divisor in ST1 in case you need to compute another partial product to complete the result.

The fprem1 instruction sets the stack exception flag if there aren't two values on the top of stack. It sets the underflow and denormal exception bits if the result is too small. It sets the invalid operation bit if the values on TOS are inappropriate for this operation. It sets the C_2 condition code bit if the partial remainder operation is not complete. Finally, it loads C_3, C_1, and C_0 with bits 0, 1, and 2 of the quotient, respectively.

An example follows:

```
// Compute z := x mod y

        fld( y );
        fld( x );
        repeat

            fprem1();
            fstsw( ax );    // Get condition code bits into ax.
            and( 1, ah );   // See if C2 is set.

        until( @z );        // Repeat until C2 is clear.
        fstp( z );          // Store away the remainder.
        fstp( st0 );        // Pop old y value.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000106

6.5.6.7 The frndint Instruction

The frndint instruction rounds the value on the top of stack (TOS) to the nearest integer using the rounding algorithm specified in the control register.

This instruction sets the stack exception flag if there is no value on the TOS (it will also clear C_1 in this case). It sets the precision and denormal exception bits if there was a loss of precision. It sets the invalid operation flag if the value on the TOS is not a valid number. Note that the result on TOS is still a floating-point value; it simply does not have a fractional component.

6.5.6.8 The fabs Instruction

fabs computes the absolute value of ST0 by clearing the mantissa sign bit of ST0. It sets the stack exception bit and invalid operation bits if the stack is empty.

Here's an example:

```
// Compute x := sqrt(abs(x));

        fld( x );
        fabs();
        fsqrt();
        fstp( x );
```

6.5.6.9 The fchs Instruction

fchs changes the sign of ST0's value by inverting the mantissa sign bit (that is, this is the floating-point negation instruction). It sets the stack exception bit and invalid operation bits if the stack is empty.

Look at this example:

```
// Compute x := -x if x is positive, x := x if x is negative.
// That is, force x to be a negative value.

        fld( x );
        fabs();
        fchs();
        fstp( x );
```

6.5.7 Comparison Instructions

The FPU provides several instructions for comparing real values. The fcom, fcomp, and fcompp instructions compare the two values on the top of stack and set the condition codes appropriately. The ftst instruction compares the value on the top of stack with 0.

Generally, most programs test the condition code bits immediately after a comparison. Unfortunately, there are no FPU instructions that test the FPU condition codes. Instead, you use the fstsw instruction to copy the floating-point status register into the AX register; then you can use the sahf instruction to copy the AH register into the 80x86's condition code bits. After doing this,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000107

you can test the standard 80x86 flags to check for some condition. This technique copies C_0 into the carry flag, C_2 into the parity flag, and C_3 into the zero flag. The sahf instruction does not copy C_1 into any of the 80x86's flag bits.

Because the sahf instruction does not copy any FPU status bits into the sign or overflow flags, you cannot use signed comparison instructions. Instead, use unsigned operations (e.g., seta, setb) when testing the results of a floating-point comparison. *Yes, these instructions normally test unsigned values, and floating-point numbers are signed values.* However, use the unsigned operations anyway; the fstsw and sahf instructions set the 80x86 flags register as though you had compared unsigned values with the cmp instruction.

The Pentium II and (upward) compatible processors provide an extra set of floating-point comparison instructions that directly affect the 80x86 condition code flags. These instructions circumvent having to use fstsw and sahf to copy the FPU status into the 80x86 condition codes. These instructions include fcomi and fcomip. You use them just like the fcom and fcomp instructions, except, of course, you do not have to manually copy the status bits to the FLAGS register.

6.5.7.1 The fcom, fcomp, and fcompp Instructions

The fcom, fcomp, and fcompp instructions compare ST0 to the specified operand and set the corresponding FPU condition code bits based on the result of the comparison. The legal forms for these instructions are:

```
fcom()
fcomp()
fcompp()

fcom( sti )
fcomp( sti )

fcom( mem_32_64 )
fcomp( mem_32_64 )
fcom( real_constant )
fcomp( real_constant )
```

With no operands, fcom, fcomp, and fcompp compare ST0 against ST1 and set the FPU flags accordingly. In addition, fcomp pops ST0 off the stack and fcompp pops both ST0 and ST1 off the stack.

With a single-register operand, fcom and fcomp compare ST0 against the specified register. fcomp also pops ST0 after the comparison.

With a 32- or 64-bit memory operand, the fcom and fcomp instructions convert the memory variable to an 80-bit extended-precision value and then compare ST0 against this value, setting the condition code bits accordingly. fcomp also pops ST0 after the comparison.

These instructions set C_2 (which winds up in the parity flag) if the two operands are not comparable (e.g., NaN). If it is possible for an illegal floating-point value to wind up in a comparison, you should check the parity flag for an error before checking the desired condition (e.g., using HLA's @p and @np conditions, or by using the setp/setnp instructions).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000108

These instructions set the stack fault bit if there aren't two items on the top of the register stack. They set the denormalized exception bit if either or both operands are denormalized. They set the invalid operation flag if either or both operands are quiet NaNs. These instructions always clear the C_1 condition code.

Note that the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. When HLA encounters such an instruction, it creates a real64 read-only variable in the constants segment and initializes this variable with the specified constant. Then HLA translates the instruction to one that specifies a real64 memory operand.

NOTE *Because of the precision differences (64 bits versus 80 bits), if you use a constant operand in a floating-point instruction you may not get results that are as precise as you would expect.*

Let's look at an example of a floating-point comparison:

```
fcompp();
fstsw( ax );
sahf();
setb( al );   // al = true if st1 < st0.
     .
     .
     .
```

Note that you cannot compare floating-point values in an HLA runtime boolean expression (e.g., within an if statement). You may, however, test the conditions in such statements after a floating-point comparison like the sequence above. For example:

```
fcompp();
fstsw( ax );
sahf();
if( @b ) then

     << Code that executes if st1 < st0 >>

endif;
```

6.5.7.2 The fcomi and fcomip Instructions

The fcomi and fcomip instructions compare ST0 to the specified operand and set the corresponding EFLAG condition code bits based on the result of the comparison. You use these instructions in a similar manner to fcom and fcomp except you can test the CPU's flag bits directly after the execution of these instructions without first moving the FPU status bits into the EFLAGS register. The legal forms for these instructions are as follows:

```
fcomi()
fcomip()
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000109

```
fcomi( sti )
fcomip( sti )

fcomi( mem_32_64 )
fcomip( mem_32_64 )
fcomi( real_constant )
fcomip( real_constant )
```

6.5.7.3 The ftst Instruction

The ftst instruction compares the value in ST0 against 0.0. It behaves just like the fcom instruction would if ST1 contained 0.0. Note that this instruction does not differentiate −0.0 from +0.0. If the value in ST0 is either of these values, ftst will set C_3 to denote equality. This instruction does *not* pop ST0 off the stack.

Here's an example:

```
ftst();
fstsw( ax );
sahf();
sete( al );                    // Set al to 1 if TOS = 0.0
```

6.5.8 Constant Instructions

The FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid operation, and C_1 flags if a stack overflow occurs; they do not otherwise affect the FPU flags. The specific instructions in this category include the following:

```
fldz()          // Pushes +0.0.
fld1()          // Pushes +1.0.
fldpi()         // Pushes pi.
fldl2t()        // Pushes log2(10).
fldl2e()        // Pushes log2(e).
fldlg2()        // Pushes log10(2).
fldln2()        // Pushes ln(2).
```

6.5.9 Transcendental Instructions

The FPU provides eight transcendental (logarithmic and trigonometric) instructions to compute sine, cosine, partial tangent, partial arctangent, $2x - 1$, $y * \log_2(x)$, and $y * \log_2(x + 1)$. Using various algebraic identities, it is easy to compute most of the other common transcendental functions using these instructions.

6.5.9.1 The f2xm1 Instruction

f2xm1 computes $2^{ST0} - 1$. The value in ST0 must be in the range −1.0..ST0..+1.0. If ST0 is out of range, f2xm1 generates an undefined result but raises no exceptions. The computed value replaces the value in ST0.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000110

Here's an example computing 10^x using the identity $10^x = 2^{x*\log2(10)}$. This is only useful for a small range of x that doesn't put ST0 outside of the previously mentioned valid range.

```
fld( x );
fldl2t();
fmul();
f2xm1();
fld1();
fadd();
```

Note that `f2xm1` computes $2x - 1$, which is why the code above adds 1.0 to the result at the end of the computation.

6.5.9.2 The fsin, fcos, and fsincos Instructions

These instructions pop the value off the top of the register stack and compute the sine, cosine, or both and push the result(s) back onto the stack. The `fsincos` instruction pushes the sine followed by the cosine of the original operand; hence it leaves cos(ST0) in ST0 and sin(ST0) in ST1.

These instructions assume ST0 specifies an angle in radians and this angle must be in the range $-2^{63} < ST0 < +2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave ST0 unchanged. You can use the `fprem1` instruction, with a divisor of 2π, to reduce the operand to a reasonable range.

These instructions set the stack fault/C_1, precision, underflow, denormalized, and invalid operation flags according to the result of the computation.

6.5.9.3 The fptan Instruction

`fptan` computes the tangent of ST0 and pushes this value, and then it pushes 1.0 onto the stack. Like the `fsin` and `fcos` instructions, the value of ST0 must be in radians and in the range $-2^{63} < ST0 < +2^{63}$. If the value is outside this range, `fptan` sets C_2 to indicate that the conversion did not take place. As with the `fsin`, `fcos`, and `fsincos` instructions, you can use the `fprem1` instruction to reduce this operand to a reasonable range using a divisor of 2π.

If the argument is invalid (i.e., zero or π radians, which causes a division by 0), the result is undefined and this instruction raises no exceptions. `fptan` will set the stack fault, precision, underflow, denormal, invalid operation, C_2, and C_1 bits as required by the operation.

6.5.9.4 The fpatan Instruction

This instruction expects two values on the top of stack. It pops them and computes $ST0 = \tan^{-1}(ST1/ST0)$.

The resulting value is the arctangent of the ratio on the stack expressed in radians. If you have a value you wish to compute the tangent of, use `fld1` to create the appropriate ratio and then execute the `fpatan` instruction.

This instruction affects the stack fault/C_1, precision, underflow, denormal, and invalid operation bits if a problem occurs during the computation. It sets the C_1 condition code bit if it has to round the result.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000111

6.5.9.5 The fyl2x Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes ST0 = ST1 * $\log_2$(ST0).

This instruction has no operands (to the instruction itself). The instruction uses the following syntax:

```
fyl2x();
```

Note that this instruction computes the base-2 logarithm. Of course, it is a trivial matter to compute the log of any other base by multiplying by the appropriate constant.

6.5.9.6 The fyl2xp1 Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes ST0 = ST1 * $\log_2$(ST0 + 1.0).

The syntax for this instruction is:

```
fyl2xp1();
```

Otherwise, the instruction is identical to `fyl2x`.

6.5.10 Miscellaneous Instructions

The FPU includes several additional instructions that control the FPU, synchronize operations, and let you test or set various status bits. These instructions include `finit/fninit`, `fldcw`, `fstcw`, `fclex/fnclex`, and `fstsw`.

6.5.10.1 The finit and fninit Instructions

The `finit` instruction initializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction initializes the control register to $37F, the status register to 0, and the tag word to $FFFF. The other registers are unaffected.

Here are some examples:

```
finit();
fninit();
```

The difference between `finit` and `fninit` is that `finit` first checks for any pending floating-point exceptions before initializing the FPU; `fninit` does not.

6.5.10.2 The fldcw and fstcw Instructions

The `fldcw` and `fstcw` instructions require a single 16-bit memory operand:

```
fldcw( mem16 );
fstcw( mem16 );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000112

These two instructions load the control register from a memory location (fldcw) or store the control word to a 16-bit memory location (fstcw).

When using the fldcw instruction to turn on one of the exceptions, if the corresponding exception flag is set when you enable that exception, the FPU will generate an immediate interrupt before the CPU executes the next instruction. Therefore, you should use the fclex instruction to clear any pending interrupts before changing the FPU exception enable bits.

6.5.10.3 The fclex and fnclex Instructions

The fclex and fnclex instructions clear all exception bits, the stack fault bit, and the busy flag in the FPU status register.

Here are some examples:

```
fclex();
fnclex();
```

The difference between these instructions is the same as between finit and fninit.

6.5.10.4 The fstsw and fnstsw Instructions

These instructions store the FPU status register into a 16-bit memory location or the AX register.

```
fstsw( ax );
fnstsw( ax );
fstsw( mem16 );
fnstsw( mem16 );
```

These instructions are unusual in the sense that they can copy an FPU value into one of the 80x86 general-purpose registers (specifically, AX). Of course, the whole purpose behind allowing the transfer of the status register into AX is to allow the CPU to easily test the condition code register with the sahf instruction. The difference between fstsw and fnstsw is the same as for fclex and fnclex.

6.5.11 Integer Operations

The FPU provides special instructions that combine integer-to-extended-precision conversion with various arithmetic and comparison operations. These instructions are the following:

```
fiadd( int_16_32 );
fisub( int_16_32 );
fisubr( int_16_32 );
fimul( int_16_32 );
fidiv( int_16_32 );
fidivr( int_16_32 );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000113

```
ficom( int_16_32 );
ficomp( int_16_32 );
```

These instructions convert their 16- or 32-bit integer operands to an 80-bit extended-precision floating-point value and then use this value as the source operand for the specified operation. These instructions use ST0 as the destination operand.

6.6 Converting Floating-Point Expressions to Assembly Language

Because the FPU register organization is different than the 80x86 integer register set, translating arithmetic expressions involving floating-point operands is a little different than the techniques for translating integer expressions. Therefore, it makes sense to spend some time discussing how to manually translate floating-point expressions into assembly language.

In one respect, it's actually easier to translate floating-point expressions into assembly language. The stack architecture of the Intel FPU eases the translation of arithmetic expressions into assembly language. If you've ever used a Hewlett-Packard calculator, you'll be right at home on the FPU because, like the HP calculator, the FPU uses *postfix notation* (also called *Reverse Polish notation*, or *RPN*), for arithmetic operations. Once you get used to using postfix notation, it's actually a bit more convenient for translating expressions because you don't have to worry about allocating temporary variables—they always wind up on the FPU stack.

Postfix notation, as opposed to standard *infix notation*, places the operands before the operator. The following examples give some simple examples of infix notation and the corresponding postfix notation:

infix notation	postfix notation
5 + 6	5 6 +
7 - 2	7 2 -
x * y	x y *
a / b	a b /

A postfix expression like `5 6 +` says, "push 5 onto the stack, push 6 onto the stack, and then pop the value off the top of stack (6) and add it to the new top of stack." Sound familiar? This is exactly what the `fld` and `fadd` instructions do. In fact, you can calculate this using the following code:

```
fld( 5.0 );
fld( 6.0 );
fadd();                    // 11.0 is now on the top of the FPU stack.
```

As you can see, postfix is a convenient notation because it's very easy to translate this code into FPU instructions.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000114

One advantage to postfix notation is that it doesn't require any parentheses. The following examples demonstrate some slightly more complex infix-to-postfix conversions:

infix notation	postfix notation
(x + y) * 2	x y + 2 *
x * 2 - (a + b)	x 2 * a b + -
(a + b) * (c + d)	a b + c d + *

The postfix expression x y + 2 * says, "Push x, then push y; next, add those values on the stack (producing x + y on the stack). Next, push 2 and then multiply the two values (2 and x + y) on the stack to produce two times the quantity x + y." Once again, we can translate these postfix expressions directly into assembly language. The following code demonstrates the conversion for each of the above expressions:

```
//          x y + 2 *

          fld( x );
          fld( y );
          fadd();
          fld( 2.0 );
          fmul();

//          x 2 * a b + -

          fld( x );
          fld( 2.0 );
          fmul();
          fld( a );
          fld( b );
          fadd();
          fsub();

//          a b + c d + *

          fld( a );
          fld( b );
          fadd();
          fld( c );
          fld( d );
          fadd();
          fmul();
```

6.6.1 Converting Arithmetic Expressions to Postfix Notation

Because the process of translating arithmetic expressions into assembly language involves postfix notation (RPN), converting arithmetic expressions into postfix notation seems like a good place to begin our discussion of floating-point expression conversion. This section will concentrate on postfix conversion.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000115

For simple expressions, those involving two operands and a single expression, the translation is trivial. Simply move the operator from the infix position to the postfix position (that is, move the operator from between the operands to after the second operand). For example, `5 + 6` becomes `5 6 +`. Other than separating your operands so you don't confuse them (i.e., is it 5 and 6 or 56?), converting simple infix expressions into postfix notation is straightforward.

For complex expressions, the idea is to convert the simple subexpressions into postfix notation and then treat each converted subexpression as a single operand in the remaining expression. The following discussion surrounds completed conversions with square brackets so it is easy to see which text needs to be treated as a single operand in the conversion.

As for integer expression conversion, the best place to start is in the innermost parenthetical subexpression and then work your way outward considering precedence, associativity, and other parenthetical subexpressions. As a concrete working example, consider the following expression:

```
x = ((y - z) * a) - ( a + b * c ) / 3.14159
```

A possible first translation is to convert the subexpression `(y - z)` into postfix notation:

```
x = ([y z -] * a) - ( a + b * c ) / 3.14159
```

Square brackets surround the converted postfix code just to separate it from the infix code. These exist only to make the partial translations more readable. Remember, for the purposes of conversion we will treat the text inside the square brackets as a single operand. Therefore, you would treat `[y z -]` as though it were a single variable name or constant.

The next step is to translate the subexpression `([y z -] * a )` into postfix form. This yields the following:

```
x = [y z - a *] - ( a + b * c ) / 3.14159
```

Next, we work on the parenthetical expression `( a + b * c )`. Because multiplication has higher precedence than addition, we convert `b * c` first:

```
x = [y z - a *] - ( a + [b c *]) / 3.14159
```

After converting `b * c` we finish the parenthetical expression:

```
x = [y z - a *] - [a b c * +] / 3.14159
```

This leaves only two infix operators: subtraction and division. Because division has the higher precedence, we'll convert that first:

```
x = [y z - a *] - [a b c * + 3.14159 /]
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000116

Finally, we convert the entire expression into postfix notation by dealing with the last infix operation, subtraction:

```
x = [y z - a *] [a b c * + 3.14159 /] -
```

Removing the square brackets to give us true postfix notation yields the following postfix expression:

```
x = y z - a * a b c * + 3.14159 / -
```

The following steps demonstrate another infix-to-postfix conversion for the expression:

```
a = (x * y - z + t) / 2.0
```

1. Work inside the parentheses. Because multiplication has the highest precedence, convert that first:

   ```
   a = ( [x y *] - z + t) / 2.0
   ```

2. Still working inside the parentheses, we note that addition and subtraction have the same precedence, so we rely on associativity to determine what to do next. These operators are left associative, so we must translate the expressions in a left-to-right order. This means translate the subtraction operator first:

   ```
   a = ( [x y * z -] + t) / 2.0
   ```

3. Now translate the addition operator inside the parentheses. Because this finishes the parenthetical operators, we can drop the parentheses:

   ```
   a = [x y * z - t +] / 2.0
   ```

4. Translate the final infix operator (division). This yields the following:

   ```
   a = [x y * z - t + 2.0 / ]
   ```

5. Drop the square brackets and we're done:

   ```
   a = x y * z - t + 2.0 /
   ```

6.6.2 Converting Postfix Notation to Assembly Language

Once you've translated an arithmetic expression into postfix notation, finishing the conversion to assembly language is easy. All you have to do is issue an `fld` instruction whenever you encounter an operand and issue an appropriate

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000117

arithmetic instruction when you encounter an operator. This section uses the completed examples from the previous section to demonstrate how little there is to this process.

```
x = y z - a * a b c * + 3.14159 / -
```

1. Convert y to fld(y).
2. Convert z to fld(z).
3. Convert - to fsub().
4. Convert a to fld(a).
5. Convert * to fmul().
6. Continuing in a left-to-right fashion, generate the following code for the expression:

```
        fld( y );
        fld( z );
        fsub();
        fld( a );
        fmul();
        fld( a );
        fld( b );
        fld( c );
        fmul();
        fadd();
        fldpi();        // Loads pi (3.14159)
        fdiv();
        fsub();

        fstp( x );      // Store result away into x.
```

Here's the translation for the second example in the previous section:

```
a = x y * z - t + 2.0 /
        fld( x );
        fld( y );
        fmul();
        fld( z );
        fsub();
        fld( t );
        fadd();
        fld( 2.0 );
        fdiv();

        fstp( a );      // Store result away into a.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000118

As you can see, the translation is fairly simple once you've converted the infix notation to postfix notation. Also note that, unlike integer expression conversion, you don't need any explicit temporaries. It turns out that the FPU stack provides the temporaries for you.[3] For these reasons, conversion of floating-point expressions into assembly language is actually easier than converting integer expressions.

6.7 HLA Standard Library Support for Floating-Point Arithmetic

Chapter 2 briefly mentioned the `stdin.getf` function. What it left out of that discussion is where `stdin.getf` returns the floating-point value is reads from the standard input. Now that you've seen the floating-point extensions to the 80x86, it's possible to finish the discussion of that standard library function. The `stdin.getf` function reads a string of characters from the standard input, converts those characters to an 80-bit floating-point number, and leaves the result sitting on the FPU stack (in ST0).

The HLA Standard Library also provides the `math.hhf` module that includes several mathematical functions that the FPU doesn't directly support as well as support for various functions (like sine and cosine) that the FPU partially supports. Some of the functions that the `math.hhf` module provides are `acos`, `acot`, `acsc`, `asec`, `asin`, `cot`, `csc`, `sec`, 2^x, 10^x, y^x, e^x, `log`, and `ln`. Please consult the HLA standard library documentation for more information about these functions and other mathematical functions the HLA standard library supports.

6.8 For More Information

The Intel/AMD processor manuals fully describe the operation of each of the integer and floating-point arithmetic instructions, including a detailed description of how these instructions affect the condition code bits and other flags in the EFLAGS and FPU status registers. To write the best possible assembly language code, you need to be intimately familiar with how the arithmetic instructions affect the execution environment, so spending time with the Intel/AMD manuals is a good idea.

The HLA Standard Library provides a large number of floating-point functions for which there are no individual machine instructions. The HLA Standard Library also provides functions like `math.sin` and `math.cos` that overcome limitations of the native machine instructions. See the HLA Standard Library reference manual for more details. Also, the HLA Standard Library is available in source code form, so you can look at the implementation of these mathematical functions for more examples of floating-point coding.

---

[3] This assumes, of course, that your calculations aren't so complex that you exceed the eight-element limitation of the FPU stack.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000119

Chapter 8 discusses multiprecision integer arithmetic. See that chapter for details on handling integer operands that are greater than 32 bits in size.

The 80x86 SSE instruction set found on later members of the CPU provides support for floating-point arithmetic using the SSE register set. Consult *http://webster.cs.ucr.edu/* or the Intel/AMD documentation for details concerning the SSE floating-point instruction set.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000120

7

LOW-LEVEL CONTROL STRUCTURES

This chapter discusses "pure" assembly language control statements. You'll need to master these low-level control structures before you can claim to be an assembly language programmer. By the time you finish this chapter, you should be able to stop using HLA's high-level control statements and synthesize them using low-level 80x86 machine instructions.

The last section of this chapter discusses *hybrid* control structures that combine the features of HLA's high-level control statements with the 80x86 control instructions. These combine the power and efficiency of the low-level control statements with the readability of high-level control statements. Advanced assembly programmers may want to use these hybrid statements to improve their programs' readability without sacrificing efficiency.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000121

7.1 Low-Level Control Structures

Until now, most of the control structures you've seen and have used in your programs are similar to the control structures found in high-level languages like Pascal, C++, and Ada. While these control structures make learning assembly language easy, they are not true assembly language statements. Instead, the HLA compiler translates these control structures into a sequence of "pure" machine instructions that achieve the same result as the high-level control structures. This text uses the high-level control structures to allow you to learn assembly language without having to learn everything all at once. Now, however, it's time to put aside these high-level control structures and learn how to write your programs in *real* assembly language, using low-level control structures.

7.2 Statement Labels

Assembly language low-level control structures make extensive use of *labels* within your source code. A low-level control structure usually transfers control between two points in your program. You typically specify the destination of such a transfer using a statement label. A statement label consists of a valid (unique) HLA identifier and a colon. For example:

```
aLabel:
```

Of course, as for procedure, variable, and constant identifiers, you should attempt to choose descriptive and meaningful names for your labels. The example identifier above, `aLabel`, is hardly descriptive or meaningful.

Statement labels have one important attribute that differentiates them from most other identifiers in HLA: You don't have to declare a label before you use it. This is important, because low-level control structures must often transfer control to some point later in the code; therefore the label may not be defined by the time you reference it.

You can do three things with labels: transfer control to a label via a jump (`goto`) instruction, call a label via the `call` instruction, and take the address of a label. There is very little else you can directly do with a label (of course, there is very little else you would want to do with a label, so this is hardly a restriction). The program in Listing 7-1 demonstrates two ways to take the address of a label in your program and print out the address (using the `lea` instruction and using the `&` address-of operator):

```
program labelDemo;
#include( "stdlib.hhf" );

begin labelDemo;

    lbl1:

        lea( ebx, lbl1 );
        mov( &lbl2, eax );
        stdout.put( "&lbl1=$", ebx, " &lbl2=", eax, nl );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000122

```
    lbl2:

end labelDemo;
```

Listing 7-1: Displaying the address of statement labels in a program

HLA also allows you to initialize double-word variables with the addresses of statement labels. However, there are some restrictions on labels that appear in the initialization portions of variable declarations. The most important restriction is that you must define the statement label at the same lexical level as the variable declaration. That is, if you reference a statement label in the initializer of a variable declaration appearing in the main program, the statement label must also be in the main program. Conversely, if you take the address of a statement label in a local variable declaration, that symbol must appear in the same procedure as the local variable. Listing 7-2 demonstrates the use of statement labels in variable initialization:

```
program labelArrays;
#include( "stdlib.hhf" );

static
    labels:dword[2] := [ &lbl1, &lbl2 ];

    procedure hasLabels;
    static
        stmtLbls: dword[2] := [ &label1, &label2 ];

    begin hasLabels;

        label1:

            stdout.put
            (
                "stmtLbls[0]= $", stmtLbls[0], nl,
                "stmtLbls[1]= $", stmtLbls[4], nl
            );

        label2:

    end hasLabels;

begin labelArrays;

    hasLabels();
    lbl1:

        stdout.put( "labels[0]= $", labels[0], " labels[1]=", labels[4], nl );

    lbl2:

end labelArrays;
```

Listing 7-2: Initializing dword variables with the address of statement labels

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000123

Once in a while, you'll need to refer to a label that is not within the current procedure. The need for this is sufficiently rare that this text will not describe all the details. See the HLA documentation for more details should you ever need to do this.

7.3 Unconditional Transfer of Control (jmp)

The jmp (jump) instruction unconditionally transfers control to another point in the program. There are three forms of this instruction: a direct jump and two indirect jumps. These instructions take the following forms:

```
jmp label;
jmp( reg32 );
jmp( mem32 );
```

The first instruction is a direct jump above. For direct jumps you normally specify the target address using a statement label. The label appears either on the same line as an executable machine instruction or by itself on a line preceding an executable machine instruction. The direct jump is completely equivalent to a goto statement in a high-level language.[1]

Here's an example:

```
        << statements >>
        jmp laterInPgm;
            .
            .
            .
laterInPgm:
        << statements >>
```

The second form of the jmp instruction given earlier—jmp(*reg32*);—is a *register indirect* jump instruction. This instruction transfers control to the instruction whose address appears in the specified 32-bit general-purpose register. To use this form of the jmp instruction, you must load a 32-bit register with the address of some machine instruction prior to the execution of the jmp. You could use this instruction to implement a *state machine* by loading a register with the address of some label at various points throughout your program and then use a single indirect jump at a common point to transfer control to one of those labels. The short sample program in Listing 7-3 demonstrates how you could use the jmp in this manner.

```
program regIndJmp;
#include( "stdlib.hhf" );

static
    i:int32;
```

[1] Unlike high-level languages, where your instructors usually forbid you to use goto statements, you will find that the use of the jmp instruction in assembly language is essential.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000124

```
begin regIndJmp;

    // Read an integer from the user and set ebx to
    // denote the success or failure of the input.

    try

        stdout.put( "Enter an integer value between 1 and 10: " );
        stdin.get( i );
        mov( i, eax );
        if( eax in 1..10 ) then

            mov( &GoodInput, ebx );

        else

            mov( &valRange, ebx );

        endif;

      exception( ex.ConversionError )

        mov( &convError, ebx );

      exception( ex.ValueOutOfRange )

        mov( &valRange, ebx );

    endtry;

    // Okay, transfer control to the appropriate
    // section of the program that deals with
    // the input.

    jmp( ebx );

    valRange:
        stdout.put( "You entered a value outside the range 1..10" nl );
        jmp Done;

    convError:
        stdout.put( "Your input contained illegal characters" nl );
        jmp Done;

    GoodInput:
        stdout.put( "You entered the value ", i, nl );

    Done:


end regIndJmp;
```

Listing 7-3: Using register-indirect jmp instructions

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000125

The third form of the jmp instruction given earlier is a memory-indirect jmp. This form of the jmp instruction fetches the double-word value from the memory location and jumps to that address. This is similar to the register-indirect jmp except the address appears in a memory location rather than in a register. Listing 7-4 demonstrates a rather trivial use of this form of the jmp instruction.

```
program memIndJmp;
#include( "stdlib.hhf" );

static
    LabelPtr:dword := &stmtLabel;

begin memIndJmp;

    stdout.put( "Before the JMP instruction" nl );
    jmp( LabelPtr );

        stdout.put( "This should not execute" nl );

    stmtLabel:

        stdout.put( "After the LabelPtr label in the program" nl );

end memIndJmp;
```

Listing 7-4: Using memory-indirect jmp instructions

WARNING *Unlike the HLA high-level control structures, the low-level jmp instructions can cause you a lot of trouble. In particular, if you do not initialize a register with the address of a valid instruction and you jump indirectly through that register, the results are undefined (though this will usually cause a general protection fault). Similarly, if you do not initialize a double-word variable with the address of a legal instruction, jumping indirectly through that memory location will probably crash your program.*

7.4 The Conditional Jump Instructions

Although the jmp instruction provides transfer of control, it is inconvenient to use when making decisions such as those you'll need to implement statements like if and while. The 80x86's conditional jump instructions handle this task.

The conditional jumps test one or more CPU flags to see if they match some particular pattern. If the flag settings match the condition, the conditional jump instruction transfers control to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the instruction following the conditional jump. Some conditional jump instructions simply test the setting of the sign, carry, overflow, and zero flags. For example, after the execution of a shl instruction, you could test the carry flag to determine if the shl shifted a 1 out of the H.O. bit of its operand. Likewise,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000126

you could test the zero flag after a test instruction to check if the result was 0. Most of the time, however, you will probably execute a conditional jump after a cmp instruction. The cmp instruction sets the flags so that you can test for less than, greater than, equality, and so on.

The conditional jmp instructions take the following form:

```
jcc label;
```

The *cc* in jcc indicates that you must substitute some character sequence that specifies the type of condition to test. These are the same characters the setcc instruction uses. For example, js stands for *jump* if the sign flag is set. A typical js instruction is:

```
js ValueIsNegative;
```

In this example, the js instruction transfers control to the ValueIsNegative label if the sign flag is currently set; control falls through to the next instruction following the js instruction if the sign flag is clear.

Unlike the unconditional jmp instruction, the conditional jump instructions do not provide an indirect form. They only allow a branch to a statement label in your program.

NOTE *Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions.*

Tables 7-1, 7-2, and 7-3 list all the aliases for a particular instruction. These tables also list the opposite branches. You'll soon see the purpose of the opposite branches.

Table 7-1: jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
jc	Jump if carry	Carry = 1	jb, jnae	jnc
jnc	Jump if no carry	Carry = 0	jnb, jae	jc
jz	Jump if zero	Zero = 1	je	jnz
jnz	Jump if not zero	Zero = 0	jne	jz
js	Jump if sign	Sign = 1		jns
jns	Jump if no sign	Sign = 0		js
jo	Jump if overflow	Overflow = 1		jno
jno	Jump if no overflow	Overflow = 0		jo
jp	Jump if parity	Parity = 1	jpe	jnp
jpe	Jump if parity even	Parity = 1	jp	jpo
jnp	Jump if no parity	Parity = 0	jpo	jp
jpo	Jump if parity odd	Parity = 0	jnp	jpe

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000127

Table 7-2: jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposite
ja	Jump if above (>)	Carry = 0, Zero = 0	jnbe	jna
jnbe	Jump if not below or equal (not <=)	Carry = 0, Zero = 0	ja	jbe
jae	Jump if above or equal (>=)	Carry = 0	jnc, jnb	jnae
jnb	Jump if not below (not <)	Carry = 0	jnc, jae	jb
jb	Jump if below (<)	Carry = 1	jc, jnae	jnb
jnae	Jump if not above or equal (not >=)	Carry = 1	jc, jb	jae
jbe	Jump if below or equal (<=)	Carry = 1 or Zero = 1	jna	jnbe
jna	Jump if not above (not >)	Carry = 1 or Zero = 1	jbe	ja
je	Jump if equal (=)	Zero = 1	jz	jne
jne	Jump if not equal (¦)	Zero = 0	jnz	je

Table 7-3: jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
jg	Jump if greater (>)	Sign = Overflow or Zero = 0	jnle	jng
jnle	Jump if not less than or equal (not <=)	Sign = Overflow or Zero = 0	jg	jle
jge	Jump if greater than or equal (>=)	Sign = Overflow	jnl	jge
jnl	Jump if not less than (not <)	Sign = Overflow	jge	jl
jl	Jump if less than (<)	Sign <> Overflow	jnge	jnl
jnge	Jump if not greater or equal (not >=)	Sign <> Overflow	jl	jge
jle	Jump if less than or equal (<=)	Sign <> Overflow or Zero = 1	jng	jnle
jng	Jump if not greater than (not >)	Sign <> Overflow or Zero = 1	jle	jg
je	Jump if equal (=)	Zero = 1	jz	jne
jne	Jump if not equal (¦)	Zero = 0	jnz	je

One brief comment about the Opposite column is in order. In many instances you will need to be able to generate the opposite of a specific branch instruction (examples appear later in this section). With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the jcc instruction is not an n, insert an n after the j. For example, je becomes jne and jl becomes jnl.
- If the second letter of the jcc instruction is an n, then remove that n from the instruction. For example, jng becomes jg and jne becomes je.

The two exceptions to this rule are jpe (jump if parity is even) and jpo (jump if parity is odd). These exceptions cause few problems because (1) you'll hardly ever need to test the parity flag, and (2) you can use the aliases jp and jnp as synonyms for jpe and jpo. The "N/No N" rule applies to jp and jnp.

Though you *know* that jge is the opposite of jl, get in the habit of using jnl rather than jge as the opposite jump instruction for jl. It's too easy in an

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000128

important situation to start thinking "greater is the opposite of less" and substitute jg instead. You can avoid this confusion by always using the "N/No N" rule.

The 80x86 conditional jump instructions give you the ability to split program flow into one of two paths depending on some condition. Suppose you want to increment the AX register if BX is equal to CX. You can accomplish this with the following code:

```
        cmp( bx, cx );
        jne SkipStmts;
        inc( ax );
SkipStmts:
```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the "opposite branch (N/No N)" rule given earlier to select the opposite branch.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the ENTER key (carriage return):

```
        mov( 0, edi );
RdLnLoop:
        stdin.getc();               // Read a character into the al register.
        mov( al, Input[ edi ] );    // Store away the character.
        inc( edi );                 // Move on to the next character.
        cmp( al, stdio.cr );        // See if the user pressed Enter.
        jne RdLnLoop;
```

Like the setcc instructions, the conditional jump instructions come in two basic categories: those that test specific processor flags (e.g., jz, jc, jno) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a cmp instruction. The cmp instruction sets the flags so that you can use a ja, jae, jb, jbe, je, or jne instruction to test for unsigned less than, less than or equal, equal, unequal, greater than, or greater than or equal. Simultaneously, the cmp instruction sets the flags so that you can also do a signed comparison using the jl, jle, je, jne, jg, and jge instructions.

The conditional jump instructions only test the 80x86 flags; they do not affect any of them.

7.5 "Medium-Level" Control Structures: jt and jf

HLA provides two special conditional jump instructions: jt (jump if true) and jf (jump if false). These instructions take the following syntax:

```
jt( boolean_expression ) target_label;
jf( boolean_expression ) target_label;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000129

The *boolean_expression* is the standard HLA boolean expression allowed by `if..endif` and other HLA high-level language statements. These instructions evaluate the boolean expression and jump to the specified label if the expression evaluates true (`jt`) or false (`jf`).

These are not real 80x86 instructions. HLA compiles them into a sequence of one or more 80x86 machine instructions that achieve the same result. In general, you should not use these two instructions in your main code; they offer few benefits over using an `if..endif` statement and they are no more readable than the pure assembly language sequences they compile into. HLA provides these "medium-level" instructions so that you may create your own high-level control structures using macros (see Chapter 9 and the HLA reference manual for more details).

7.6 Implementing Common Control Structures in Assembly Language

Because a primary goal of this chapter is to teach you how to use the low-level machine instructions to implement decisions, loops, and other control constructs, it would be wise to show you how to implement these high-level statements using pure assembly language. The following sections provide this information.

7.7 Introduction to Decisions

In its most basic form, a *decision* is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the `if..then..endif` statement in HLA:

```
if( expression ) then
    << statements >>
endif;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following C/C++ statement:

```
if( (( x < y ) && ( z > t )) || ( a != b ) )
    stmt1;
```

A "brute force" approach to converting this statement into assembly language might produce the following:

```
mov( x, eax );
cmp( eax, y );
setl( bl );        // Store x<y in bl.
mov( z, eax );
cmp( eax, t );
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000130

```
        setg( bh );        // Store z>t in bh.
        and( bh, bl );     // Put (x<y) && (z>t) into bl.
        mov( a, eax );
        cmp( eax, b );
        setne( bh );       // Store a != b into bh.
        or( bh, bl );      // Put (x<y) && (z>t) || (a!=b) into bl
        je SkipStmt1;      // Branch if result is false.

    << Code for Stmt1 goes here. >>

SkipStmt1:
```

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) C/C++ statements:

```
        bl = x < y;
        bh = z > t;
        bl = bl && bh;
        bh = a != b;
        bl = bl || bh;
        if( bl )
            << Stmt1 >>;
```

Now compare this with the following "improved" code:

```
        mov( a, eax );
        cmp( eax, b );
        jne DoStmt;
        mov( x, eax );
        cmp( eax, y );
        jnl SkipStmt;
        mov( z, eax );
        cmp( eax, t );
        jng SkipStmt;
DoStmt:
        << Place code for Stmt1 here. >>
SkipStmt:
```

Two things should be apparent from the code sequences above: First, a single conditional statement in C/C++ (or some other HLL) may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, you should exercise care when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: `if` statements, `switch/case` statements, and indirect jumps. The following sections describe these program structures, how to use them, and how to write them in assembly language.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000131

7.7.1 if..then..else Sequences

The most common conditional statements are the `if..then..endif` and `if..then..else..endif` statements. These two statements take the form shown in Figure 7-1.

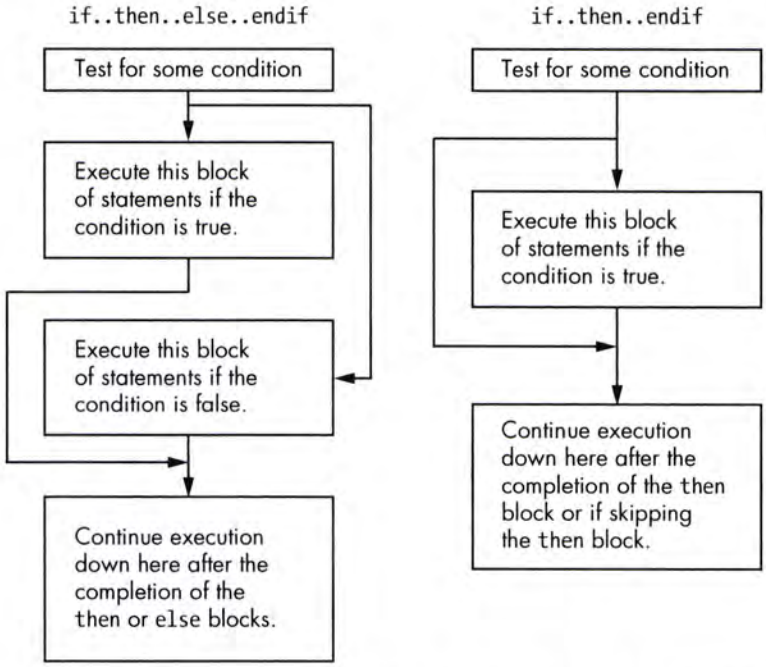


Figure 7-1: `if..then..else..endif` and `if..then..endif` statement flow

The `if..then..endif` statement is just a special case of the `if..then..else..endif` statement (with an empty `else` block). Therefore, we'll consider only the more general `if..then..else..endif` form. The basic implementation of an `if..then..else..endif` statement in 80x86 assembly language looks something like this:

```
    << Sequence of statements to test some condition >>
        jcc ElseCode;
    << Sequence of statements corresponding to the THEN block >>

        jmp EndOfIf;

ElseCode:
    << Sequence of statements corresponding to the ELSE block >>

EndOfIf:
```

Note that `jcc` represents some conditional jump instruction. For example, to convert the C/C++ statement

```
    if( a == b )
        c = d;
    else
        b = b + 1;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000132

to assembly language, you could use the following 80x86 code:

```
        mov( a, eax );
        cmp( eax, b );
        jne ElsePart;
        mov( d, c );
        jmp EndOfIf;

ElseBlk:
        inc( b );

EndOfIf:
```

For simple expressions like (a == b) generating the proper code for an if..then..else..endif statement is almost trivial. Should the expression become more complex, the code complexity increases as well. Consider the following C/C++ if statement presented earlier:

```
    if( (( x > y ) && ( z < t )) || ( a != b ) )
        c = d;
```

When processing complex if statements such as this one, you'll find the conversion task easier if you break the if statement into a sequence of three different if statements as follows:

```
    if( a != b ) c = d;
    else if( x > y)
        if( z < t )
            c = d;
```

This conversion comes from the following C/C++ equivalents:

```
if( expr1 && expr2 ) stmt;
```

is equivalent to

```
if( expr1 ) if( expr2 ) stmt;
```

and

```
if( expr1 || expr2 ) stmt;
```

is equivalent to

```
if( expr1 ) stmt;
else if( expr2 ) stmt;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000133

In assembly language, the former if statement becomes

```
// if( (( x > y ) && ( z < t )) || ( a != b ) )
//      c = d;

          mov( a, eax );
          cmp( eax, b );
          jne DoIF;
          mov( x, eax );
          cmp( eax, y );
          jng EndOfIF;
          mov( z, eax );
          cmp( eax, t );
          jnl EndOfIf;
DoIf:
          mov( d, eax );
          mov( eax, c );
EndOfIf:
```

As you can see, testing a condition can easily become more complex than the statements appearing in the else and then blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act on the results of that condition, it happens all the time. Therefore, you should be prepared to accept this.

Probably the biggest problem with complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. A big advantage high-level languages offer over assembly language is that expressions are much easier to read and comprehend. The high-level version is (more) self-documenting, whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of if..then..else..endif statements. An elegant implementation of the example above is as follows:

```
// if ((x > y) && (z < t)) or (a != b)  c = d;
// Implemented as:
// if (a != b) then goto DoIf;

          mov( a, eax );
          cmp( eax, b );
          jne DoIf;

// if not (x > t) then goto EndOfIf;

          mov( x, eax );
          cmp( eax, y );
          jng EndOfIf;

// if not (z < t) then goto EndOfIf;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000134

```
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

// then block:

DoIf:
        mov( d, eax );
        mov( eax, c );

// End of if statement

EndOfIf:
```

Admittedly, this appears to be going overboard for such a simple example. The following would probably suffice:

```
// if ( (( x > y ) && ( z < t )) || ( a != b ) )  c = d;
// Test the boolean expression:

        mov( a, eax );
        cmp( eax, b );
        jne DoIf;
        mov( x, eax );
        cmp( eax, y );
        jng EndOfIf;
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

// then block:

DoIf:
        mov( d, eax );
        mov( eax, c );

// End of if statement

EndOfIf:
```

However, as your `if` statements become complex, the density (and quality) of your comments become more and more important.

7.7.2 Translating HLA if Statements into Pure Assembly Language

Translating HLA `if` statements into pure assembly language is very easy. The boolean expressions that the HLA `if` statement supports were specifically chosen to expand into a few simple machine instructions. The following paragraphs discuss the conversion of each supported boolean expression into pure machine code.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000135

if(*flag_specification*) then *stmts* endif;

This form is, perhaps, the easiest HLA if statement to convert. To execute the code immediately following the then keyword if a particular flag is set (or clear), all you need do is skip over the code if the flag is clear (set). This requires only a single conditional jump instruction for implementation, as the following examples demonstrate:

```
// if( @c ) then inc( eax );  endif;

        jnc SkipTheInc;

            inc( eax );

        SkipTheInc:

// if( @ns ) then neg( eax ); endif;

        js SkipTheNeg;

            neg( eax );

        SkipTheNeg:
```

if(*register*) then *stmts* endif;

This form uses the test instruction to check the specified register for 0. If the register contains 0 (false), then the program jumps around the statements after the then clause with a jz instruction. Converting this statement to assembly language requires a test instruction and a jz instruction, as the following examples demonstrate:

```
// if( eax ) then mov( false, eax );  endif;

        test( eax, eax );
        jz DontSetFalse;

            mov( false, eax );

        DontSetFalse:

// if( al ) then mov( bl, cl );  endif;

        test( al, al );
        jz noMove;

            mov( bl, cl );

        noMove:
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000136

if(!*register*) then *stmts* endif;

This form of the `if` statement uses the `test` instruction to check the specified register to see if it is 0. If the register is not 0 (true), then the program jumps around the statements after the `then` clause with a `jnz` instruction. Converting this statement to assembly language requires a `test` instruction and a `jnz` instruction in a manner identical to the previous examples.

if(*boolean_variable*) then *stmts* endif;

This form of the `if` statement compares the boolean variable against 0 (false) and branches around the statements if the variable contains false. HLA implements this statement by using the `cmp` instruction to compare the boolean variable to 0, and then it uses a `jz` (`je`) instruction to jump around the statements if the variable is false. The following example demonstrates the conversion:

```
// if( bool ) then mov( 0, al );  endif;

        cmp( bool, false );
        je SkipZeroAL;

            mov( 0, al );

        SkipZeroAL:
```

if(!*boolean_variable*) then *stmts* endif;

This form of the `if` statement compares the boolean variable against 0 (false) and branches around the statements if the variable contains true (the opposite condition of the previous example). HLA implements this statement by using the `cmp` instruction to compare the boolean variable to 0 and then it uses a `jnz` (`jne`) instruction to jump around the statements if the variable contains true. The following example demonstrates the conversion:

```
// if( !bool ) then mov( 0, al );  endif;

        cmp( bool, false );
        jne SkipZeroAL;

            mov( 0, al );

        SkipZeroAL:
```

if(*mem_reg relop mem_reg_const*) then *stmts* endif;

HLA translates this form of the `if` statement into a `cmp` instruction and a conditional jump that skips over the statements on the opposite condition specified by the `relop` operator. Table 7-4 lists the correspondence between operators and conditional jump instructions.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000137

Table 7-4: if Statement Conditional Jump Instructions

Relational operation	Conditional jump instruction if both operands are unsigned	Conditional jump instruction if either operand is signed
= or ==	jne	jne
<> or !=	je	je
<	jnb	jnl
<=	jnbe	jnle
>	jna	jng
>=	jnae	jnge

Here are a few examples of if statements translated into pure assembly language that use expressions involving relational operators:

```
// if( al == ch ) then inc( cl ); endif;

        cmp( al, ch );
        jne SkipIncCL;

            inc( cl );

        SkipIncCL:

// if( ch >= 'a' ) then and( $5f, ch ); endif;

        cmp( ch, 'a' );
        jnae NotLowerCase

            and( $5f, ch );

        NotLowerCase:

// if( (type int32 eax ) < -5 ) then mov( -5, eax );  endif;

        cmp( eax, -5 );
        jnl DontClipEAX;

            mov( -5, eax );

        DontClipEAX:

// if( si <> di ) then inc( si );  endif;

        cmp( si, di );
        je DontIncSI;

            inc( si );

        DontIncSI:
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000138

if(*reg/mem* in *LowConst..HiConst*) then *stmts* endif;

HLA translates this `if` statement into a pair of `cmp` instructions and a pair of conditional jump instructions. It compares the register or memory location against the lower-valued constant and jumps if less than (signed) or below (unsigned) past the statements after the `then` clause. If the register or memory location's value is greater than or equal to *`LowConst`*, the code falls through to the second `cmp` and conditional jump pair that compares the register or memory location against the higher constant. If the value is greater than (above) this constant, a conditional jump instruction skips the statements in the `then` clause.

Here's an example:

```
// if( eax in 1000..125_000 ) then sub( 1000, eax );  endif;

        cmp( eax, 1000 );
        jb DontSub1000;
        cmp( eax, 125_000 );
        ja DontSub1000;

            sub( 1000, eax );

        DontSub1000:

// if( i32 in -5..5 ) then add( 5, i32 ); endif;

        cmp( i32, -5 );
        jl NoAdd5;
        cmp( i32, 5 );
        jg NoAdd5;

            add(5, i32 );

        NoAdd5:
```

if(*reg/mem* not in *LowConst..HiConst*) then *stmts* endif;

This form of the HLA `if` statement tests a register or memory location to see if its value is outside a specified range. The implementation is very similar to the previous code except you branch to the `then` clause if the value is less than the *`LowConst`* value or greater than the *`HiConst`* value, and you branch over the code in the `then` clause if the value is within the range specified by the two constants. The following examples demonstrate how to do this conversion:

```
// if( eax not in 1000..125_000 ) then add( 1000, eax );  endif;

        cmp( eax, 1000 );
        jb Add1000;
        cmp( eax, 125_000 );
        jbe SkipAdd1000;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000139

```
            Add1000:
            add( 1000, eax );

        SkipAdd1000:

// if( i32 not in -5..5 ) then mov( 0, i32 );  endif;

        cmp( i32, -5 );
        jl Zeroi32;
        cmp( i32, 5 );
        jle SkipZero;

            Zeroi32:
            mov( 0, i32 );

        SkipZero:
```

7.7.3 Implementing Complex if Statements Using Complete Boolean Evaluation

Many boolean expressions involve conjunction (and) or disjunction (or) operations. This section describes how to convert boolean expressions into assembly language. There are two different ways to convert complex boolean expressions involving conjunction and disjunction into assembly language: using complete boolean evaluation or using short-circuit boolean evaluation. This section discusses complete boolean evaluation. The next section discusses short-circuit boolean evaluation.

Conversion via complete boolean evaluation is almost identical to converting arithmetic expressions into assembly language. Indeed, the previous chapter on arithmetic covers this conversion process. About the only thing worth noting about that process is that you do not need to store the result in some variable; once the evaluation of the expression is complete, you check to see if you have a false (0) or true (1, or nonzero) result to take whatever action the boolean expression dictates. As you can see in the examples in the preceding sections, you can often use the fact that the last logical instruction (and/or) sets the zero flag if the result is false and clears the zero flag if the result is true. This lets you avoid explicitly testing for the result. Consider the following if statement and its conversion to assembly language using complete boolean evaluation:

```
//      if( (( x < y ) && ( z > t )) || ( a != b ) )
//          << Stmt1 >>;

        mov( x, eax );
        cmp( eax, y );
        setl( bl );     // Store x<y in bl.
        mov( z, eax );
        cmp( eax, t );
        setg( bh );     // Store z>t in bh.
        and( bh, bl );  // Put (x<y) && (z>t) into bl.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000140

```
        mov( a, eax );
        cmp( eax, b );
        setne( bh );     // Store a != b into bh.
        or( bh, bl );    // Put (x<y) && (z>t) || (a != b) into bl.
        je SkipStmt1;    // Branch if result is false.

    << Code for Stmt1 goes here. >>

SkipStmt1:
```

This code computes a boolean result in the BL register and then, at the end of the computation, tests this value to see if it contains true or false. If the result is false, this sequence skips over the code associated with `Stmt1`. The important thing to note in this example is that the program will execute each and every instruction that computes this boolean result (up to the `je` instruction).

7.7.4 Short-Circuit Boolean Evaluation

If you are willing to expend a little more effort, you can usually convert a boolean expression to a much shorter and faster sequence of assembly language instructions using *short-circuit boolean evaluation.* Short-circuit boolean evaluation attempts to determine whether an expression is true or false by executing only some of the instructions that would compute the complete expression. For this reason, plus the fact that short-circuit boolean evaluation doesn't require the use of any temporary registers, HLA uses short-circuit evaluation when translating complex boolean expressions into assembly language.

Consider the expression `a && b`. Once we determine that `a` is false, there is no need to evaluate `b` because there is no way the expression can be true. If and `b` represent subexpressions rather than simple variables, the savings possible with short-circuit boolean evaluation are apparent. As a concrete example, consider the subexpression `((x<y) && (z>t))` from the previous section. Once you determine that `x` is not less than `y`, there is no need to check to see if `z` is greater than `t` because the expression will be false regardless of `z` and `t`'s values. The following code fragment shows how you can implement short-circuit boolean evaluation for this expression:

```
// if( (x<y) && (z>t) ) then ...

        mov( x, eax );
        cmp( eax, y );
        jnl TestFails;
        mov( z, eax );
        cmp( eax, t );
        jng TestFails;

            << Code for THEN clause of IF statement >>

        TestFails:
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000141

Notice how the code skips any further testing once it determines that x is not less than y. Of course, if x is less than y, then the program has to test z to see if it is greater than t; if not, the program skips over the then clause. Only if the program satisfies both conditions does the code fall through to the then clause.

For the logical or operation the technique is similar. If the first subexpression evaluates to true, then there is no need to test the second operand. Whatever the second operand's value is at that point, the full expression still evaluates to true. The following example demonstrates the use of short-circuit evaluation with disjunction (or):

```
// if( ch < 'A' || ch > 'Z' )
//     then stdout.put( "Not an uppercase char" );
// endif;

        cmp( ch, 'A' );
        jb ItsNotUC
        cmp( ch, 'Z' );
        jna ItWasUC;

            ItsNotUC:
            stdout.put( "Not an uppercase char" );

        ItWasUC:
```

Because the conjunction and disjunction operators are commutative, you can evaluate the left or right operand first if it is more convenient to do so.[2] As one last example in this section, consider the full boolean expression from the previous section:

```
// if( (( x < y ) && ( z > t )) || ( a != b ) ) << Stmt1 >>;

        mov( a, eax );
        cmp( eax, b );
        jne DoStmt1;
        mov( x, eax );
        cmp( eax, y );
        jnl SkipStmt1;
        mov( z, eax );
        cmp( eax, t );
jng SkipStmt1;

            DoStmt1:
            << Code for Stmt1 goes here. >>

        SkipStmt1:
```

[2] However, be aware of the fact that some expressions depend on the leftmost subexpression evaluating one way in order for the rightmost subexpression to be valid; for example, a common test in C/C++ is `if( x != NULL && x->y )...`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1021, Page 000142