1

What Is Assembly Language?

One of the first hurdles to learning assembly language programming is understanding just what assembly language is. Unlike other programming languages, there is no one standard format that all assemblers use. Different assemblers use different syntax for writing program statements. Many beginning assembly language programmers get caught up in trying to figure out the myriad of different possibilities in assembly language programming.

The first step in learning assembly language programming is defining just what type of assembly language programming you want to (or need to) use in your environment. Once you define your flavor of assembly language, it is easy to get started learning and using assembly language in both standalone and high-level language programs.

This chapter begins the journey by showing where assembly language comes from, and defining why assembly language programming is used. To understand assembly language programming, you must first understand the basics of its underlying purpose — programming in processor instruction code. Next, the chapter shows how high-level languages are converted to raw instruction code by compilers and linkers. After having that information, it will be easier for you to understand how assembly language programs and high-level language programs differ, and how they can both be used to complement one another.

Processor Instructions

At the lowest layer of operation, all computer processors (microcomputers, minicomputers, and mainframe computers) manipulate data based on binary codes defined internally in the processor chip by the manufacturer. These codes define what functions the processor should perform, utilizing the data provided by the programmer. These preset codes are referred to as *instruction codes*. Different types of processors contain different types of instruction codes. Processor chips are often categorized by the quantity and type of instruction codes they support.

While the different types of processors can contain different types of instruction codes, they all handle instruction code programs similarly. This section describes how processors handle instructions and what the instruction codes look like for a sample processor chip.

Instruction code handling

As a computer processor chip runs, it reads instruction codes that are stored in memory. Each instruction code set can contain one or more bytes of information that instruct the processor to perform a specific task. As each instruction code is read from memory, any data required for the instruction code is also stored and read in memory. The memory bytes that contain the instruction codes are no different than the bytes that contain the data used by the processor.

To differentiate between data and instruction codes, special *pointers* are used to help the processor keep track of where in memory the data and instruction codes are stored. This is shown in Figure 1-1.

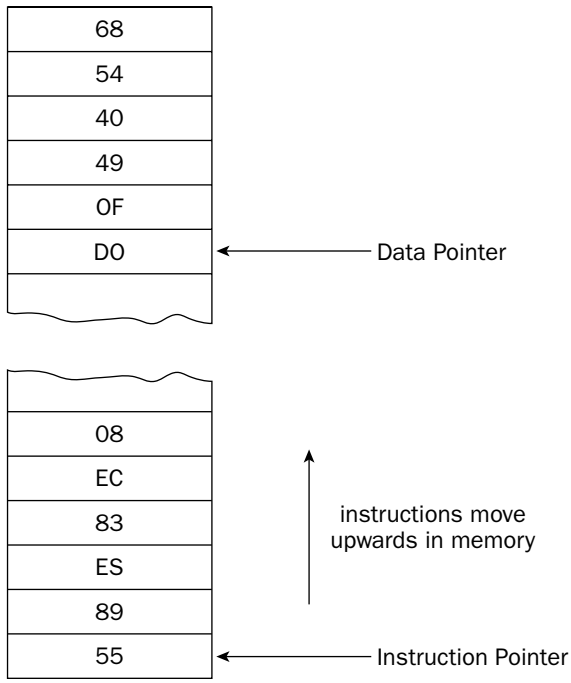

Figure 1-1

The *instruction pointer* is used to help the processor keep track of which instruction codes have already been processed and what code is next in line to be processed. Of course, there are special instruction codes that can change the location of the instruction pointer, such as jumping to a specific location in the program.

Similarly, a *data pointer* is used to help the processor keep track of where the data area in memory starts. This area is called the *stack*. As new data elements are placed in the stack, the pointer moves "down" in memory. As data is read from the stack, the pointer moves "up" in memory.

Each instruction code can contain one or more bytes of information for the processor to handle. For example, the instruction code bytes (in hexadecimal format)

```
C7 45 FC 01 00 00 00
```

tell an Intel IA-32 series processor to load the decimal value 1 into a memory offset location defined by a processor register. The instruction code contains several pieces of information (defined later in the "Opcode" section) that clearly define what function is to be performed by the processor. After the processor completes processing one instruction code set, it reads the next one in memory (as pointed to by the instruction pointer). The instructions must be placed in memory in the proper format and order for the processor to properly step through the program code.

Every instruction must contain at least 1 byte called the *operation code* (or *opcode* for short). The opcode defines what function the processor should perform. Each processor family has its own predefined opcodes that define all of the functions available. The next section shows how the opcodes used in the Intel IA-32 family of microprocessors are structured. These are the types of processor opcodes that are used in all of the examples in this book.

Instruction code format

The Intel IA-32 family of microprocessors includes all of the current types of microprocessors used in modern IBM-platform microcomputers (see Chapter 2, "The IA-32 Platform"), including the popular Pentium line of microprocessors. A specific format for instruction codes is used in the IA-32 family of microprocessors, and understanding the format of these instructions will help you in your assembly language programming. The IA-32 instruction code format consists of four main parts:

- Optional instruction prefix
- Operational code (opcode)
- Optional modifier
- Optional data element

Figure 1-2 shows the layout of the IA-32 instruction code format.

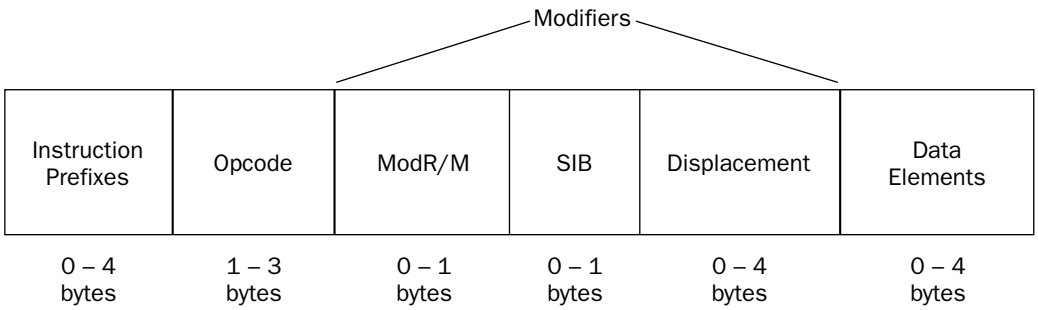

Figure 1-2

Each of the parts is used to completely define a specific instruction for the processor to perform. The following sections describe each of the four parts of the instruction code and how they define the instruction performed by the processor.

> *The Intel Pentium processor family is not the only set of processor chips to utilize the IA-32 instruction code format. The AMD corporation also produces a line of chips that are fully compatible with the Intel IA-32 instruction code format.*

Opcode

As shown in Figure 1-2, the only required part of the IA-32 instruction code format is the opcode. Each instruction code must include an opcode that defines the basic function or task to be performed by the processor.

The opcode is between 1 and 3 bytes in length, and uniquely defines the function that is performed. For example, the 2-byte opcode `0F A2` defines the IA-32 CPUID instruction. When the processor executes this instruction code, it returns specific information about the microprocessor in different registers. The programmer can then use additional instruction codes to extract the information from the processor registers to determine the type and model of microprocessor on which the program is running.

> *Registers are components within the processor chip that are used to temporarily store data while being handled by the processor. They are covered in more detail in Chapter 2, "The IA-32 Platform."*

Instruction prefix

The instruction prefix can contain between one and four 1-byte prefixes that modify the opcode behavior. These prefixes are categorized into four different groups, based on the prefix function. Only one prefix from each group can be used at one time to modify the opcode (thus the maximum of four prefix bytes). The four prefix groups are as follows:

- Lock and repeat prefixes
- Segment override and branch hint prefixes
- Operand size override prefix
- Address size override prefix

The lock prefix indicates that any shared memory areas will be used exclusively by the instruction. This is important for multiprocessor and hyperthreaded systems. The repeat prefixes are used to indicate a repeating function (usually used when handling strings).

The segment override prefixes define instructions that can override the defined segment register value (described in more detail in Chapter 2). The branch hint prefixes attempt to give the processor a clue as to the most likely path the program will take in a conditional jump statement (this is used with predictive branching hardware).

The operand size override prefix informs the processor that the program will switch between 16-bit and 32-bit operand sizes within the instruction code. This enables the program to warn the processor when it uses larger-sized operands, helping to speed up the assignment of data to registers.

The address size override prefix informs the processor that the program will switch between 16-bit and 32-bit memory addresses. Either size can be declared as the default size for the program, and this prefix informs the processor that the program is switching to the other.

Modifiers

Some opcodes require additional modifiers to define what registers or memory locations are involved in the function. The modifiers are contained in three separate values:

- addressing-form specifier (ModR/M) byte
- Scale-Index-Base (SIB) byte
- One, two, or four address displacement bytes

The ModR/M byte

The ModR/M byte consists of three fields of information, as shown in Figure 1-3.

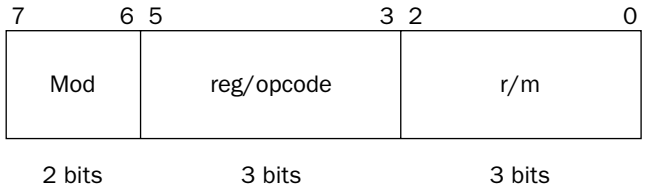

Figure 1-3

The mod field is used with the r/m field to define the register or addressing mode used in the instruction. There are 24 possible addressing modes, along with eight possible general-purpose registers that can be used in the instruction, making 32 possible values.

The reg/opcode field is used to enable three more bits to further define the opcode function (such as opcode subfunctions), or it can be used to define a register value.

The r/m field is used to define another register to use as the operand of the function, or it can be combined with the mod field to define the addressing mode for the instruction.

The SIB byte

The SIB byte also consists of three fields of information, as shown in Figure 1-4.

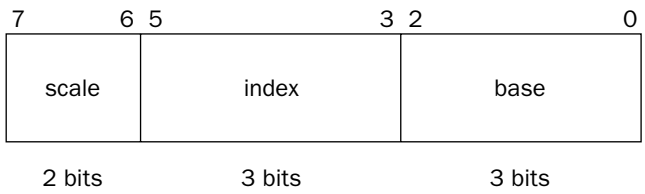

Figure 1-4

The scale field specifies the scale factor for the operation. The index field specifies the register that is used as the index register for memory access. The base field specifies the register that is used as the base register for memory access.

The combination of the ModR/M and SIB bytes creates a table that can define many possible combinations of registers and memory modes for accessing data. The Intel specification sheets for the Pentium processor define all of the possible combinations that are used with the ModR/M and SIB bytes.

The address displacement byte

The address displacement byte is used to indicate an offset to the memory location defined in the ModR/M and SIB bytes. This can be used as an index to a base memory location to either store or access data within memory.

Data element

The final part of the instruction code is the data element that is used by the function. While some instruction codes read data from memory locations or processor registers, some include data within the instruction code itself. Often this value is used to represent a static numeric value, such as a number to be added, or a memory location. This value can contain 1, 2, or 4 bytes of information, depending on the data size.

For example, the following sample instruction code shown earlier:

```
C7 45 FC 01 00 00 00
```

defines the opcode C7, which is the instruction to move a value to a memory location. The memory location is defined by the 45 FC modifier (which defines –4 bytes (the FC value) from the memory location pointed to by the value in the EBP register (the 45 value). The final 4 bytes define the integer value that is placed in that memory location (in this case, the value 1).

> *As you can see from this example, the value 1 was written as the 4-byte hexadecimal value 01 00 00 00. The order of the bytes in the data stream depends on the type of processor used. The IA-32 platform processors use "little-endian" notation, whereby the lower-value bytes appear first in order (when reading left to right). Other processors use "big-endian" order, whereby the higher-value bytes appear first in order. This concept is extremely important when specifying data and memory location values in your assembly language programs.*

High-Level Languages

If it looks like programming in pure processor instruction code is difficult, it is. Even the simplest of programs require the programmer to specify a lot of opcodes and data bytes. Trying to manage a huge program full of just instruction codes would be a daunting task. To help save the sanity of programmers, high-level languages (HLLs) were created.

HLLs enable programmers to create functions using simpler terms, rather than raw processor instruction codes. Special reserved keywords are used to define variables (memory locations for data), create loops (jump over instruction codes), and handle input and output from the program. However, the processor does not have any knowledge about how to handle the HLL code. The code must be converted by some mechanism to simple instruction code format for the processor to handle. This section defines the

different types of HLLs and then shows how the HLL code is converted to the instruction code for the processor to execute.

Types of high-level languages

While programmers can choose from many different HLLs available, they all can be classified into two different categories, based on how they are run on the computer:

- Compiled languages
- Interpreted languages

While it is possible for different implementations of the same programming language to be either compiled or interpreted, these categories are used to show how a particular HLL implementation defines how the programs are run on the processor. The following sections describe the methods used to run programs and show how they affect how the processor operates with them.

Compiled languages

Most production applications are created using compiled HLLs. The programmer creates a program using common statements for the language which carry out the logic of the application. The text program statements are then converted into a set of instruction codes that can be run on the processor. Usually, what is commonly called *compiling* a program is actually a two-step process:

- Compiling the HLL statements into raw instruction codes
- Linking the raw instruction codes to produce an executable program

Figure 1-5 demonstrates this process.

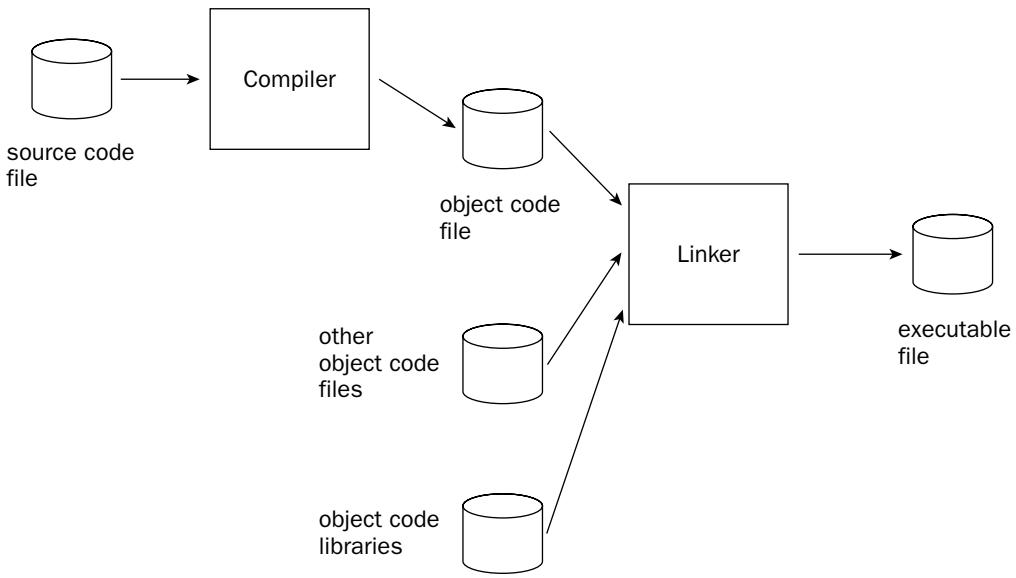

Figure 1-5

The compiling step converts the text programming language statements into the instruction codes required to carry out the application function. Each of the HLL lines of code are matched up with one or more instruction codes pertaining to the specific processor on which the application will run. For example, the simple HLL code

```
int main()
{
    int i = 1;
    exit(0);
}
```

is compiled into the following IA-32 instruction codes:

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

This step produces an intermediate file, called an *object code file*. The object code file contains the instruction codes that represent the core of the application functions, as shown above. The object code file itself cannot be run by the operating system. Often the host operating system requires special file formats for executable files (program files that can be run on the system), and the HLL program may require program functions from other object files. Another step is required to add these components.

After the code is compiled into an object file, a *linker* is used to link the application object code file with any additional object files required by the application and to create the final executable output file. The output of the linker is an executable file that can only be run on the operating system for which the program is written. Unfortunately, each operating system uses a different format for executable files, so an application compiled on a Microsoft Windows workstation will not work as is on a Linux workstation, and vice versa.

> *Object files that contain commonly used functions can be combined into a single file, called a library file. The library file can then be linked into multiple applications either at compile time (called static libraries), or at the time the application is run on the system (called dynamic libraries).*

Interpreted languages

As opposed to compiled programs, which run by themselves on a processor, an interpreted language program is read and run by a separate program. The separate program is a host for the application program, reading and interpreting the program as it is processed. It is the job of the host program to convert the interpreted program code into the proper instruction codes for the processor as the program is running.

Obviously, the downside to using interpreted languages is speed. Instead of the program being compiled directly to instruction codes that are run on the processor, an intermediary program reads each line of program code and processes the required functions. The amount of time the host program takes to read the code and execute it adds additional delays to the execution of the application.

With the resulting reduction in speed when using interpreted languages, you may be wondering why anyone still uses them. One answer is convenience. With compiled programs, every time a change is made to the program, the program must be recompiled and relinked with the proper code libraries. With interpreted programs, changes can be quickly made to the source code file and the program rerun to check for errors. In addition, with interpreted languages, the interpreter application automatically determines what functions need to be included with the core code to support functions.

> *Today's programming language environment muddies the waters between compiled and interpreted languages. No one specific language can be classified in either category. Instead, individual implementations of different HLLs are categorized. For example, while many BASIC programming implementations require interpreters to interpret the BASIC code into an executable program, there are many BASIC implementations that enable the programmer to compile the BASIC programs into executable instruction code.*

Hybrid languages

Hybrid languages are a recent trend in programming that combine the features of a compiled program with the versatility and ease of an interpreted program. A perfect example is the popular Java programming language.

The Java programming language is compiled into what is called *byte code*. The byte code is similar to the instruction code you would see on a processor, but is itself not compatible with any current processor family (although there have been plans to create a processor that can run Java byte code as instruction sets).

Instead, the Java byte code must be interpreted by a Java Virtual Machine (JVM), running separately on the host computer. The Java byte code is portable, in that it can be run by any JVM on any type of host computer. The advantage is that different platforms can have their own specific JVMs, which are used to interpret the same Java byte code without it having to be recompiled from the original source code.

High-level language features

If you are a professional programmer, most likely you do most (if not all) of your coding using a high-level language. You may or may not have had the luxury of choosing which HLL you use for your projects, but either way, there is no doubt that it makes your job easier. This section describes two of the most useful features of HLLs, portability and standardization, which help set HLLs apart from assembly language programming.

Portability

As described earlier in the "Processor Instructions" section, instruction code programming is highly dependent on the processor used in the computer. Each of the different families of processors utilize different instruction code formats, as well as different methods for storing data (big endian vs. little endian). Instruction codes written for an IA-32 platform will not work on a MIPS processor platform.

Imagine writing a 10,000-line instruction code program for your new application, which runs on a Sun Sparc workstation, and then being asked to port it to a Linux workstation running on a Pentium computer. Because the microprocessor used for the Sun Sparc workstation does not use the same instruction codes as the Pentium, all of your code would need to be redone for the new instruction codes — ouch.

HLLs have the capability to be ported to other operating systems and other processor platforms by simply recompiling the program on the new platform. When the program is recompiled, it is automatically rewritten using the instruction codes for the destination processor.

> *However, in practice, nontrivial programs use operating system APIs that make it difficult to simply recompile the source code for another platform. For example, a program directly using the MS Windows API will not compile under Linux.*

Standardization

Another useful feature of HLLs is the abundance of standards available for the languages. Both the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) have created standard specifications for many different HLLs.

This means that you are guaranteed to obtain the same results from source code compiled with a standard compiler on one type of operating system and processor as you would compiling on a different type of operating system and processor. Each compiler is created to interpret the standard language constructs into instruction code for the destination processor to produce the same functionality across the processor platforms.

Assembly Language

While creating large applications using an HLL is often simpler than using raw instruction codes, it doesn't necessarily mean that the resulting program will be efficient. Unfortunately, in order to increase portability and comply with standards, many compilers code to the "least common denominator." This means that compilers creating instruction codes for advanced processor chips may not utilize special instruction codes unique to those processors to help create faster applications.

One feature that many of the new processors on the market offer is advanced mathematics handling instruction codes. These instruction codes help speed up complex mathematical expression processing by using larger-than-normal byte sizes to represent numbers (either 64 or 128 bits). Unfortunately, many compilers don't take advantage of these advanced instruction codes. Fortunately, there is a simple solution for the programmer. In environments where execution speed is critical, assembly language programming can come to the rescue. Of course, the first step to improving execution speed is to ensure that the best algorithm is used in the first place. Optimizing a poor algorithm does not compensate for using a fast algorithm in the first place.

Assembly language enables programmers to directly create instruction code programs without having to worry about the many different instruction code set combinations on the processor. Instead, an assembly language program uses *mnemonics* to represent instruction codes. The mnemonics enables the programmer to use English-style words to represent individual instruction codes. The assembly language mnemonics are easily converted to the raw instruction codes by an assembler.

This section describes the assembly language mnemonic system, and how it is used to create raw instruction code programs that can be run on the processor.

An assembly language program consists of three components that are used to define the program operations:

- Opcode mnemonics
- Data sections
- Directives

The following sections describe each of these components and show how they are used within the assembly language program to create the resulting instruction code program.

Opcode mnemonics

The core of an assembly language program is the instruction codes used to create the program. To help facilitate writing the instruction codes, assemblers equate mnemonic words with instruction code functions, such as moving or adding data elements. For example, the instruction code sample

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

can be written in assembly language as follows:

```
push %ebp
mov  %esp, %ebp
sub  $0x8, %esp
movl $0x1, -4(%ebp)
sub  $0xc, %esp
push $0x0
call 8048348
```

Instead of having to know what each byte of instruction code represents, the assembly language programmer can use easier-to-remember mnemonic codes, such as `push`, `mov`, `sub`, and `call`, to represent the instruction codes.

Different assemblers use different mnemonics to represent instruction codes. While trends have emerged to standardize assembler mnemonics, there is still quite a vast variety of mnemonic codes, not only between processor families but even between assemblers used for the same processor instruction code sets.

> *Each processor manufacturer publishes developer manuals detailing all of the instruction codes implemented by a specific chip set. The Intel IA-32 developer manuals are freely available at the Intel Web site (www.intel.com). These developer manuals take over 1,000 pages just to enumerate and describe all of the instruction codes for the Pentium family of processors.*

Defining data

Besides the instruction codes, most programs also require data elements to be used to hold variable and constant data values that are used throughout the program. HLLs use variables to define sections of memory to hold data. For example, it is not uncommon to see the following in an HLL program:

```
long testvalue = 150;
char message[22] = {"This is a test message"};
float pi = 3.14159;
```

Each of these statements is interpreted by the HLL compiler to reserve memory locations of a specific number of bytes to store values that may or may not change during the course of the program. Each time the program references the variable name (such as `testvalue`), the compiler knows to access the specified location in memory to read or change the byte values.

Assembly language also enables the programmer to define data items that will be stored in memory. One of the advantages of programming in assembly language is that it provides you with greater control over where and how your data is stored in memory. The following sections describe two methods used to store and retrieve data in assembly language.

Using memory locations

Similar to the HLL method of defining data, assembly language enables you to declare a variable that points to a specific location in memory. Defining variables in assembly language consists of two parts:

1. A label that points to a memory location
2. A data type and default value for the memory bytes

The data type determines how many bytes are reserved for the variable. In an assembly language program, this would look like the following:

```
testvalue:
   .long 150
message:
    .ascii "This is a test message"
pi:
   .float 3.14159
```

As you can see from the data types, assembly language allows you to declare the type of data stored in the memory location, along with the default values placed in the memory location, similar to most HLL methods. Each data type occupies a specific number of bytes, starting at the memory location reserved for the label. This is shown in Figure 1-6.

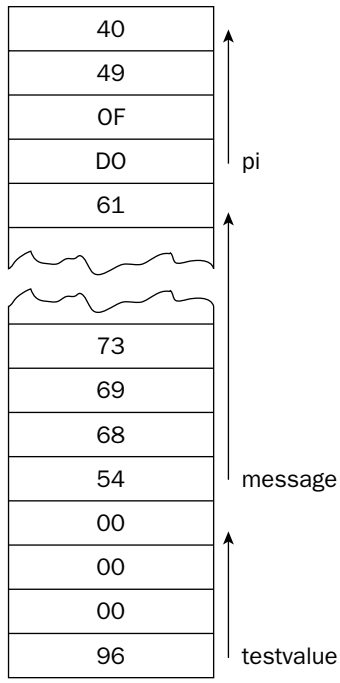

Figure 1-6

The first data element declared, `testvalue`, is placed in memory as a 4-byte hexadecimal value in little-endian order (96 00 00 00). The next data element, `message`, is placed immediately after the last byte of the `testvalue` data element. Because the `message` data element is a text value, it is placed in memory in the order the text characters appear in the string. Finally, the last data element, `pi`, is placed in memory immediately after the last byte of the `message` data element (the floating point is discussed in great detail in Chapter 7, "Using Numbers."

The memory locations are referenced within the assembly language program based on the label used to define the starting location. A sample assembly language program would look like the following:

```
movl testvalue, %ebx
addl $10, %ebx
movl %ebx, testvalue
```

The first instruction loads the EBX register with the 4-byte value located at the memory location pointed to by the `testvalue` label (which was defined with a value of 150). The next instruction adds 10 (in decimal) to the value stored in the EBX register and puts the result back in the EBX register. Finally, the register value is stored in the memory location referenced by the `testvalue` label. This new value can then be referenced again in the program using the `testvalue` label, and it will have the value of 160 (this process is explained in detail in Chapter 5, "Moving Data," and Chapter 8, "Basic Math Functions").

Using the stack

Another method used to store and retrieve data in assembly language is called the *stack*. The stack is a special memory area usually reserved for passing data elements between functions in the program. It can also be used for temporarily storing and retrieving data elements.

The stack is a region of memory reserved at the end of the memory range that the computer reserves for the application. A pointer (called the *stack pointer*) is used to point to the next memory location in the stack to put or take data. Much like a stack of papers, when a data element is placed in the stack, it becomes the first item that can be removed from the stack (assuming you can only take papers off of the top of the paper stack).

When calling functions in an assembly language program, you usually place any data elements that you want passed to the function on the top of the stack. When the function is called, it can retrieve the data elements from the stack.

> *The different methods of storing and retrieving data are discussed in greater detail in Chapter 5, "Moving Data."*

Directives

Instructions and data are not the only elements that make up an assembly language program. Assemblers reserve special keywords for instructing the assembler how to perform special functions as the mnemonics are converted to instruction codes.

You saw an example of directives in the previous section when the data elements were defined. The data types were declared using assembler directives used in the GNU assembler. The `.long`, `.ascii`, and `.float` directives are used to alert the assembler that a specific type of data is being declared. As shown in the example, directives are preceded by a period to set them apart from labels.

Directives are another area in which the different assemblers vary. Many different directives are used to help make the programmer's job of creating instruction codes easier. Some modern assemblers have lists of directives that can rival many HLL features, such as while loops, and if-then statements! The older, more traditional assemblers, however, keep the directives to a minimum, forcing the assembly language programmer to use the mnemonic codes to create the program logic.

One of the most important directives used in the assembly language program is the `.section` directive. This directive defines the section of memory in which the assembly language program is defining elements. All assembly language programs have at least three sections that must be declared:

- A data section
- A bss section
- A text section

The data section is used to declare the memory region where data elements are stored for the program. This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

The bss section is also a static memory section. It contains buffers for data to be declared later in the program. What makes this section special is that the buffer memory area is zero-filled.

The text section is the area in memory where the instruction code is stored. Again, this area is fixed, in that it contains only the instruction codes that are declared in the assembly language program.

These directives used in an assembly language program are demonstrated in Chapter 4, "A Sample Assembly Language Program."

Summary

While assembly language programming is often referred to as a single programming language category, in reality there are a wide variety of different types of assembly language assemblers. Each assembler uses slightly different formats to represent instruction codes, data, and special directives for assembling the final program. The first step to programming in assembly language is deciding which assembler you need to use, and what format it uses.

The purpose of using assembly language is to code as closely to raw processor code as possible. The code recognized by the processor is called instruction code. Each processor family has its own set of instruction codes that define the functions the processor can perform. Each processor family also uses specific formats for the instruction code. The Intel IA-32 family of processors uses a format that consists of four parts. An opcode is used to define which processor instruction should be used. An optional prefix may be used to modify the behavior of the instruction. An optional modifier may also be used to define what registers or memory locations are used in the instruction. Finally, an optional data element may be included, which defines specific data values used in the instruction.

Trying to create large-scale programs using raw instruction codes is not an easy task. Each instruction code must be programmed byte by byte in the proper order for the application to run. Instead of forcing programmers to learn all of the instruction codes, developers have created high-level languages, which enable programmers to create programs in a shorthand method, which is then converted into the proper instruction codes by a compiler. High-level languages use simple keywords and terms to define one or more instruction codes. This enables programmers to concentrate on the logic of the application program, rather than worry about the details of the underlying processor instruction codes.

The downside of using high-level languages is that the programmer is dependant on the compiler creator to convert programming logic to the instruction code run by the processor. There is no guarantee that the created instruction codes will be the most efficient method of programming the logic. For programmers who want maximum efficiency, or the capability to have greater control over how the program is handled by the processor, assembly language programming offers an alternative.

Assembly language programming enables the programmer to program with instruction codes, but by using simple mnemonic terms to refer to those instruction codes. This provides programmers with both the ease of a high-level language and the control offered by using instruction codes.

Unfortunately, assembly language assemblers are not standardized, and there are many different forms of assembly language. All assemblers contain three elements: opcode mnemonics, data elements, and

directives. The opcode mnemonics are used to code the programming logic, and data elements are used to define memory locations to hold both constant and variable data elements. Directives are one of the most controversial elements of assemblers. Directives help the programmer define specific functions, such as declaring data types, and define memory regions within the program. Some assemblers take directives to a higher level, providing directives that support many high-level language functions, such as advanced data management and logic programming.

The next chapter discusses the specific layout of the Intel IA-32 processor family. Before you can start programming for the Pentium family of processors, it is important to understand how the hardware is laid out. Knowing how the processor handles data will enable you to program more efficiently, increasing the speed of your applications.