ASSEMBLY LANGUAGE PROGRAMMING

Introduction

Objectives

- To introduce the assembly language and explain where it fits in the hierarchy of computer languages
- To discuss the advantages and disadvantages associated with programming in the assembly language
- To provide motivation to learn the assembly language
- To demonstrate performance advantages of the assembly language

1.1 A User's View of Computer Systems

A user's view of a computer system depends on the degree of abstraction provided by the underlying software. Figure 1.1 shows a hierarchy of levels at which users can interact with a computer system. Moving to the top of the hierarchy shields the user from the lower level details. At the highest level, the user interaction is limited to the interface provided by application software such as a spreadsheet, word processor, and so on. The user is expected to have only a rudimentary knowledge of how the system operates. Problem solving at this level, for example, involves composing a letter using the word processor software. At the next level, problem solving is done in one of the high-level languages such as C and Java. A user interacting with the system at this level should have detailed knowledge of software development. Typically, these users are application programmers. Level 4 users are knowledgeable about the application and the high-level language that they would use to write the application software. They may not, however, know internal details of the system unless they also happen to be involved in developing system software such as device drivers, assemblers, linkers, and so on. Both levels 4 and 5 are system-independent, i.e., independent of a particular processor used in the system. For example, an application program written in C can be executed on a system with an Intel processor or a PowerPC processor without modifying the source code. All we have to do is recompile the program with a C compiler native to the target system. By contrast, software development done at all levels below level 4 is system-dependent. Assembly language programming is referred to as low-level programming because each assembly language instruction performs a much lower-level task compared to an instruction in a high-level language. As a consequence, to perform the same task, assembly language code tends to be much larger than the equivalent high-level language code. Assembly language instructions are native to the processor used in the system. For example, a program written in the Pentium assembly language cannot be executed on the PowerPC processor. Programming in the assembly language also requires knowledge about system internal details such as the processor architecture, memory organization, and so on. Machine language is a close relative of the assembly language. Typically, there is a one-to-one correspondence between the assembly language and machine language instructions. The processor understands only the machine language, whose instructions consist of strings of 1's and 0's. We say more on these two languages in the next section. Even though the assembly language is considered a low-level language, programming in the assembly language will not expose you to all the nuts and bolts of the system. Our operating system hides several of the low-level details so that the assembly language programmer can breathe easily. For example, if we want to read input from the

keyboard, we can rely on the services provided by the operating system. Well, ultimately there has to be something to execute the machine language instructions. This is the system hardware, which consists of digital logic circuits and the associated support electronics.

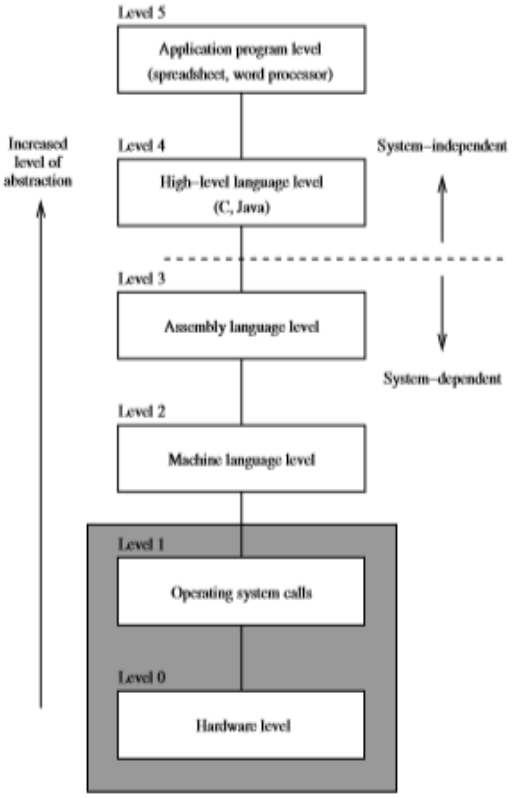


Figure 1.1 A user's view of a computer system.

1.2 What Is Assembly Language?

Assembly language is directly influenced by the instruction set and architecture of the processor. There are two basic types of processors: CISC (Complex Instruction Set Computers) and RISC (Reduced Instruction Set Computers). The Pentium is an example of a CISC processor.

Most current processors, however, follow the RISC design philosophy. As the name suggests, CISC processors use complex instructions. What is a complex instruction? For example, adding two integers is considered a simple instruction. But, an instruction that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction. RISC systems use only simple instructions. Furthermore, RISC systems assume that the required operands are in the processor's registers, not in the main memory. A CISC processor, on the other hand, does not impose such restrictions. So what? It turns out that characteristics like simple instructions and restrictions like register-based operands not only simplify the processor design but also result in a processor that provides improved application performance. The assembly language code must be processed by a program in order to generate the machine language code. Assembler is the program that translates assembly language code into the machine language. NASM (Netwide Assembler), MASM (Microsoft Assembler), and TASM (Borland Turbo Assembler) are some of the popular assemblers for the Pentium processors. We will use the SPIM simulator to run the MIPS assembly language programs. Here are some Pentium language examples:

```
inc result
 mov class_size,45
```

```
and mask1,128
add marks,10
```

The first instruction increments the variable result. This assembly language instruction is equivalent to

```
result++;
```

in C. The second instruction initializes class_size to 45. The equivalent statement in C is

```
class_size = 45;
```

The third instruction performs the bitwise and operation on mask1 and can be expressed in C as

```
mask1 = mask1 & 128;
```

The last instruction updates marks by adding 10. This is equivalent to

```
marks = marks + 10;
```

in C.

As you can see from these examples, most Pentium instructions use two addresses. In these instructions, one operand doubles as a source and destination (for example, marks and class_size). In contrast, the MIPS instructions use three addresses as shown below:

```
andi $t2,$t1,15
addu $t3,$t1,$t2
 move $t2,$t1
```

The operands in these instructions are in processor registers. The processor registers t1, t2, andt3 are identified by $. The andi instruction performs bitwise and of t1 contents with 15 and writes the result in t2. The second instruction adds contents of t1 and t2 and stores the result in t3. The last instruction copies the t1 value into t2. In contrast to our claim that MIPS uses three addresses, this instruction seems to use only two addresses. This is not really an instruction supported by MIPS processor—it is a synthesized assembly language instruction. When translated by the MIPS assembler, it is replaced by

```
addu $t2,$0,$t1
```

The second operand in this instruction is a special register that holds constant zero. Thus, copying the t1 value is treated as adding zero to it. These examples illustrate several points:

1. Assembly language instructions are cryptic.
2. Assembly language operations are expressed by using mnemonics (like and, inc, addu, and so on).
3. Assembly language instructions are low-level. For example, we cannot write the following in the Pentium assembly language:

```
add marks,value
```

This instruction is invalid because two variables, marks and value, cannot be used in a single instruction. In MIPS, for example, we cannot even write something like

```
addu class_size,45
```

as it expects all operands in the processor's internal registers.

We appreciate the readability of the assembly language instructions by looking at the equivalent machine language instructions. Here are some Pentium and MIPS machine language examples:

Pentium examples

Assembly language		Operation	Machine language (in hex)
nop		No operation	90
inc	result	Increment	FF060A00
mov	class_size, 45	Copy	C7060C002D00
and	mask, 128	Logical and	80260E0080
add	marks, 10	Integer addition	83060F000A

MIPS examples

Assembly language		Operation	Machine language (in hex)
nop		No operation	00000000
move	$t2,$t15	Copy	000A2021
andi	$t2,$t1,15	Logical and	312A000F
addu	$t3,$t1,$t2	Integer addition	012A5821

In the above tables, machine language instructions are written in the hexadecimal number system. If you are not familiar with this number system, consult Appendix A for a detailed discussion of various number systems. These examples visibly demonstrate one of the key differences between CISC and RISC processors: RISC processors use fixed-length machine language instructions whereas the machine language instructions of CISC processors vary in length. It is obvious from these examples that understanding the code of a program in the machine language is almost impossible. Since there is a one-to-one correspondence between the instructions of assembly language and machine language, it is fairly straightforward to translate instructions from the assembly language to the machine language. However, life was not so easy for some of the early programmers. When microprocessors were first introduced, some programming was in fact done in machine language!

1.3 Advantages of High-Level Languages

High-level languages are preferred to program applications, as they provide a convenient abstraction of the underlying system suitable for problem solving. Here are some advantages of programming in a high-level language:

1. Program development is faster. Many high-level languages provide structures (sequential, selection, iterative) that facilitate program development. Programs written in a high-level language are relatively small compared to the equivalent programs written in an assembly language. These programs are also easier to code and debug.

2. Programs are easier to maintain. Programming a new application can take from several weeks to several months and the life cycle of such an application software can be several years. Therefore, it is critical that software development be done with a view of software maintainability, which involves activities ranging from fixing bugs to generating the next version of the software. Programs written in a high-level language are easier to understand and, when good programming practices are followed, easier to maintain. Assembly language programs tend to be lengthy and take more time to code and debug. As a result, they are also difficult to maintain.

3. Programs are portable. High-level language programs contain very few processor-dependent details. As a result, they can be used with little or no modification on different computer systems. By contrast, assembly language programs are processor-specific.

1.4 Why Program in the Assembly Language?

There are two main reasons why programming is still done in the assembly language: (1) efficiency, and (2) accessibility to system hardware. Efficiency refers to how “good” a program is in achieving a given objective. Here we consider two objectives based on space (space efficiency) and time

(time efficiency). Space efficiency refers to the memory requirements of a program, i.e., the size of the executable code. Program A is said to be more space-efficient if it takes less memory space than program B to perform the same task. Very often, programs written in an assembly language tend to be more compact than those written in a high-level language. Time efficiency refers to the time taken to execute a program. Obviously a program that runs faster is said to be better from the time-efficiency point of view. If we craft assembly language programs carefully, they tend to run faster than their high-level language counterparts.

The superiority of the assembly language in generating compact code is becoming increasingly less important for several reasons. First, the savings in space pertain only to the program code and not to its data space. Thus, depending on the application, the savings in space obtained by converting an application program from some high-level language to the assembly language may not be substantial. Second, the cost of memory has been decreasing and memory capacity has been increasing. Thus, the size of a program is not a major hurdle anymore. Finally, compilers are becoming "smarter" in generating code that is both space and time-efficient. However, there are systems such as embedded controllers and handheld devices in which space efficiency is important. One of the main reasons for writing programs in the assembly language is to generate code that is time-efficient. The superiority of assembly language programs in producing efficient code is a direct manifestation of specificity. That is, assembly language programs contain only the code that is necessary to perform the given task. Even here, a "smart" compiler can optimize the code that can compete well with its equivalent written in the assembly language. Although the gap is narrowing with improvements in compiler technology, the assembly language still retains its advantage for now. The other main reason for writing assembly language programs is to have direct control over system hardware. High-level languages, on purpose, provide a restricted (abstract) view of the underlying hardware. Because of this, it is almost impossible to perform certain tasks that require access to the system hardware. For example, writing a device driver to a new scanner on the market almost certainly requires programming in the assembly language. Since the assembly language does not impose any restrictions, you can have direct control over the system hardware. If you are developing system software, you cannot avoid writing assembly language programs.

1.5 Typical Applications

We have identified three advantages to programming in an assembly language.

1. Time efficiency
2. Accessibility to hardware
3. Space efficiency

Time efficiency: Applications for which the execution speed is important fall under two categories:

1. Time convenience (to improve performance)
2. Time-critical (to satisfy functionality)

Applications in the first category benefit from time-efficient programs because it is convenient or desirable.

In time-critical applications, tasks have to be completed within a specified time period. These applications, also called real-time applications, include aircraft navigation systems, process control systems, robot control software, communications software, and target acquisition (e.g., missile tracking) software.

Accessibility to hardware: System software often requires direct control over the system hardware. Examples include operating systems, assemblers, compilers, linkers, loaders, device drivers, and network interfaces. Some applications also require hardware control. Video games are an obvious example.

Space efficiency: For most systems, compactness of application code is not a major concern. However, in portable and handheld devices, code compactness is an important factor. Space efficiency is also important in spacecraft control systems.

1.6 Why Learn the Assembly Language?

Programming in the assembly language is a tedious and error-prone process. The natural preference of a programmer is to program in some high-level language. We discussed a few good reasons why some applications cannot be programmed in a high-level language. Even these applications do not require the whole program to be written in the assembly language. In such instances, a small part of the program is written in the assembly language and the rest is written in some high-level language. Such programs are called hybrid or mixed-mode programs. Learning the assembly language has both practical and educational purposes. Even if you don't intend to program in an assembly language, studying it gives you a good understanding of computer systems. When you program in a high-level language, you are provided only a "black-box" view of the system. When programming in the assembly language, you need to understand the internal details of the system (for example, you need to know about processor internal registers). To understand the assembly language is to understand the computer system itself!. We use the popular Pentium processor to represent the CISC category. We study RISC processors by looking at the MIPS assembly language. Studying these two assembly languages gives you a solid foundation to understand the differences between the CISC and RISC design philosophies and how they impact execution speed of your programs. A final reason to learn the assembly language is the personal satisfaction that comes with learning something complex. Sure, learning the assembly language is more difficult than learning Java. But the assembly language gives you complete control over the system hardware. You feel powerful with the assembly language on your side, making the time spent learning assembly language worth your while. The insights provided by the assembly language will benefit you even if you program only in high-level languages.

1.7 Performance: C Versus Assembly Language

Now let's see how much better we can do by writing programs in assembly language. As an example, consider multiplying two 16-bit integers. Our strategy is to write the multiplication procedure in C (a representative high-level language) and in the Pentium assembly language and compare their execution times.

1.7.1 Multiplication Algorithm

The Pentium instruction set has two instructions for multiplication: one for unsigned integers and the other for signed integers. These instructions can be used to multiply two integers that can take up to 32 bits to represent them. Here we consider the algorithm that is based on the longhand multiplication. This algorithm, shown below, takes two n-bit unsigned integers and produces a 2n-bit product.

product := 0 for (i =1to n) if (least significant bit of the multiplier = 1) then product := product + multiplicand end if Left shift the multiplicand by one bit position Right shift the multiplier by one bit position end for

More details on this algorithm are given in Appendix A. The main program is shown in Program 1.1 on page 16. To avoid the influence of I/O (i.e., the printf and scanf statements), we time only the mult procedure. To do this, we use clock(), which is defined in the time.h header file. When clock() is invoked, it gives the current clock value in terms of number of clock ticks. The number of clock ticks per second is defined by CLOCKS_PER_SEC. Thus, to obtain the multiplication time in seconds, we have to divide the clock ticks by CLOCKS_PER_SEC. The C version of the mult procedure is given in Program 1.2 (page 17). This procedure multiplies two 16-bit integers. As you can see from this program listing, it directly follows the algorithm described before. The assembly

language version of the procedure is shown in Program 1.3. As you can see from this code, the assembly language statements are inserted into the procedure using the asm construct. For this reason, this method is called inline assembly. We give more details on this method in Chapter 17. At this time, you are not expected to make any sense out of this program.

Section 1.7 Performance: C Versus Assembly Language

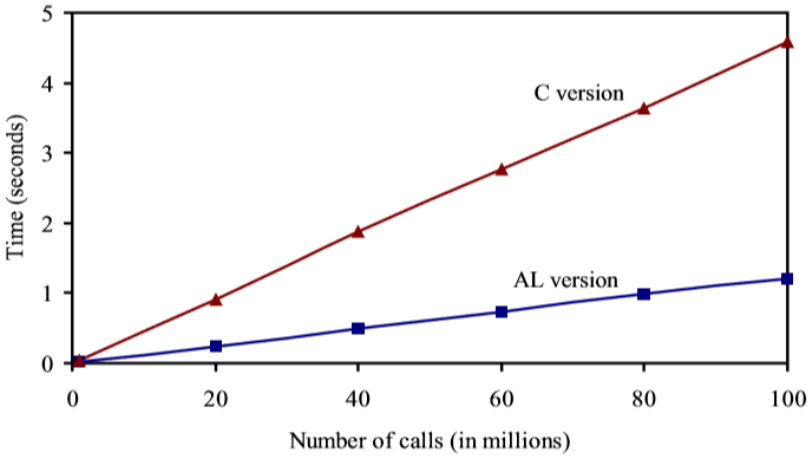


Figure 1.2 Multiplication time comparison on a 2.4-GHz Pentium 4 system: C version uses the multiplication procedure shown in Program 1.2; the assembly language (AL) version uses the assembly language procedure shown in Program 1.3.

Speedup Let us now look at the potential performance benefit we can get from using the assembly language. Toward this end, we present the multiplication times for the C and assembly language versions in Figure 1.2. These timings were obtained on a 2.4-GHz Pentium 4 system running Red Hat Linux 8.0. The y-axis gives the multiplication time in seconds. It is clear from this plot that the assembly language version runs substantially faster than the C version. This plot substantiates our claim that assembly language programs are time efficient. For example, to execute the procedure 100 million times, the C version takes about 3.5 seconds more than the assembly language version.

To quantify the performance difference, let us look at the speedup. We define speedup as

Speedup =

Execution time of the C version Execution time of the assembly language version

Speedup values greater than 1 indicate that performance of the assembly language version is better—the higher the speedup, the better the assembly language performance. For our application, we get a speedup of about 4. The reader should be cautioned that the improvement obtained by the assembly language programs depends on the application, compiler, and the type of processor, and so on. It is also important to write an efficient assembly language code in order to get better performance. If we write sloppy assembly language code, it may run slower than the compiler-generated code. This implies that critical analysis and efficient coding are very important to realize the potential performance gains from the assembly language. In practice, assembly language programming is limited to critical sections of a program. When we say critical, we mean either due to the nature of the application (e.g., real-time constraints) or due to performance reasons.

Basic Computer Organization

Objectives

• To provide a high-level view of computer organization
• To describe processor organization details
• To discuss memory organization and structure
• To introduce how input/output devices are interfaced
• To illustrate the importance of data alignment

Programming in a high-level language does not require a detailed knowledge of the system hardware. Assembly language programmers, however, should have some basic understanding of the underlying system architecture. A high-level view of computer systems, presented in Section 2.1, consists of three major components: a processor, a memory unit, and input/output devices.

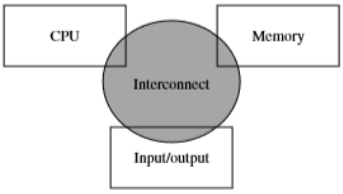


Figure 2.1 High-level view of a computer system.

2.1 Basic Components of a Computer System

A computer system has three main components: a central processing unit (CPU) or processor, a memory unit, and input/output (I/O) devices (see Figure 2.1). These three components are interconnected by a system bus. The term "bus" is used to represent a group of electrical signals or the wires that carry these signals. Figure 2.2 shows details of how they are interconnected and what actually constitutes the system bus. As shown in this figure, the three major components of the system bus are the address bus, data bus, and control bus. The width of the address bus determines the memory addressing capacity of the processor. The width of the data bus indicates the size of the data transferred between the processor and memory or I/O device. For example, the 8086 processor has a 20-bit address bus and a 16-bit data bus. The amount of physical memory that this processor can address is 220 bytes, or 1 MB, and each data transfer involves 16 bits. The Pentium, on the other hand, has 32 address lines and 64 data lines. Thus, the Pentium can address up to 232 bytes, or a 4-GB memory. Furthermore, each data transfer can move 64 bits. In comparison to the Pentium, Intel's 64-bit processor Itanium uses 64 address lines and 128 data lines. The control bus consists of a set of control signals. Typical control signals include memory read, memory write, I/O read, I/O write, interrupt, interrupt acknowledge, bus request, and bus grant. These control signals indicate the type of action taking place on the system bus. For example, when the processor is writing data into the memory, the memory write signal is asserted. Similarly, when the processor is reading from an I/O device, the I/O read signal is asserted. The system memory, also called main memory or primary memory, is used to store both program instructions and data. I/O devices such as the keyboard and display are used to provide user interface. I/O devices are also used to interface with secondary storage devices such as disks. The system bus is the communication medium for data transfers. Such data transfers are called the bus transactions. Some examples of bus transactions are memory read, memory write, I/O read, I/O write, and interrupt. Depending on the processor and the type of bus used, there may be other types of transactions. For example, Pentium supports a burst mode of data transfer in which up to four 64 bits of data can be transferred in a burst cycle.

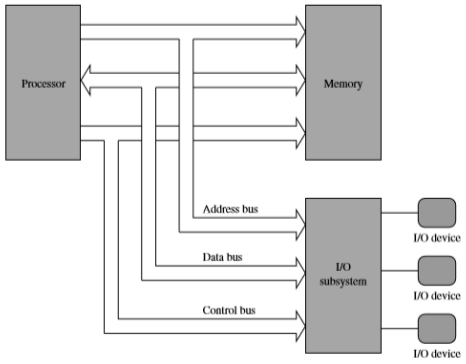


Figure 2.2 Simplified block diagram of a computer system.

Every bus transaction involves a master and a slave. The master is the initiator of the transaction and the slave is the target of the transaction. For example, when the processor wants to read data from the memory, it initiates a bus transaction, also called a bus cycle, in which the processor is the bus master and memory is the slave. The processor usually acts as the master of the system bus, while components like memory are usually slaves. Some components may act as slaves for some transactions and as masters for other transactions. When there is more than one master device, which is typically the case, the device requesting the use of the bus sends a bus request signal to the bus arbiter using the bus request control line. If the bus arbiter grants the request, it notifies the requesting device by sending a signal on the bus grant control line. The granted device, which acts as the master, can then use the bus for data transfer. The bus-request-grant procedure is called the bus protocol. Different buses use different bus protocols. In some protocols, permission to use the bus is granted for only one bus cycle; in others, permission is granted until the bus master relinquishes the bus.

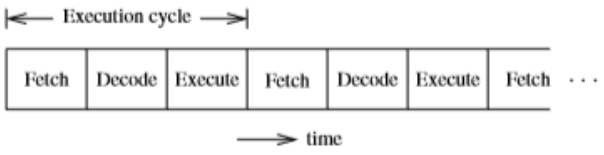


Figure 2.3 Execution cycle of a typical computer system.

2.2 The Processor

The processor acts as the controller of all actions or services provided by the system. It can be thought of as executing the following cycle forever:

1. Fetch an instruction from the memory;
2. Decode the instruction (i.e., identify the instruction);
3. Execute the instruction (i.e., perform the action specified by the instruction).

This process is often referred to as the fetch-decode-execute cycle, or simply the execution cycle. These instructions are translated by a compiler/assembler to an equivalent sequence of machine language instructions that the processor understands. The operating system, which provides instructions to the processor whenever a user program is not executing, loads the user program into the main memory. The operating system then indicates the location of the user program to the processor and instructs it to execute the program.

2.2.1 The Execution Cycle

The execution cycle of a processor is shown in Figure 2.3. Fetching an instruction from the main memory involves placing the appropriate address on the address bus and activating the memory read signal on the control bus to indicate to the memory unit that an instruction should be read from that location. The memory unit requires time to read the instruction at the addressed location. This time is called the access time. The memory then places the instruction on the data

bus. The processor, after instructing the memory unit to read, waits until the instruction is available on the data bus and then reads the instruction. Decoding involves identifying the instruction that has been fetched from the memory. To facilitate the decoding process, machine language instructions follow a particular instruction encoding scheme.

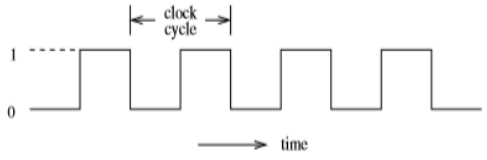


Figure 2.4 Clock signal of a computer system.

2.2.2 The System Clock

The system clock provides a timing signal to synchronize the operations of the system. A clock is a sequence of 1's and 0's, as shown in Figure 2.4. The clock frequency is measured in the number of cycles per second. This number is referred to as Hertz (Hz). We often use the abbreviations MHz and GHz to represent 106 and 109 cycles per second, respectively. The system clock defines the speed at which the system operates. All processor operations take multiple clock cycles. For example, transfer of data from a memory location to Pentium takes three clock cycles. Thus, the higher the clock rate, the faster the system can work. The clock period is defined as the length of time taken by one clock cycle.

$$\text{Clock period} = \frac{1}{\text{Clock frequency}}$$

For example, a clock frequency of 1 GHz yields a clock period of

$$\frac{1}{1 \times 10^9} = 1 \text{ ns}$$

If it takes three clock cycles to execute an instruction, it takes 3×1 ns = 3 ns. One way to increase the speed of a computer system is to use a higher clock frequency. For example, if we use a clock of 2 GHz, the instruction execution time reduces from 3 ns to 1.5 ns. Clock frequency increases with improvements in technology. The original IBM PC used a clock of 4.77 MHz. Current technology allows clock frequencies higher than 3 GHz.

2.3 Number of Addresses

One of the characteristics that shapes the architecture of a processor is the number of addresses used in its instructions. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

2.3.1 Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. Most current processors use three addresses. The MIPS processor we discuss in Chapter 12, for example, uses three addresses. Table 2.1 gives some sample instructions of a three-address machine. On these machines, the C statement

A = B +C * D – E + F + A

is converted to the following code:

```
mult T,C,D ; T = C * D
 add T,T,B ; T = B + C*D
```

sub T,T,E ; T = B + C*D – E
add T,T,F ; T = B + C*D - E + F

Table 2.1 Sample Three-Address Machine Instructions

Instruction	Semantics
`add dest,src1,src2`	Adds the two values at `src1` and `src2` and stores the result in `dest`
`sub dest,src1,src2`	Subtracts the second source operand at `src2` from the first at `src1` and stores the result in `dest`
`mult dest,src1,src2`	Multiplies the two values at `src1` and `src2` and stores the result in `dest`

Table 2.2 Sample Two-Address Machine Instructions

Instruction	Semantics
`load dest,src`	Copies the value at `src` to `dest`
`add dest,src`	Adds the two values at `src` and `dest` and stores the result in `dest`
`sub dest,src`	Subtracts the second source operand at `src` from the first at `dest` and stores the result in `dest`
`mult dest,src`	Multiplies the two values at `src` and `dest` and stores the result in `dest`

add A,A,T ; A = B + C*D - E + F + A

2.3.2 Two-Address Machines

In two-address machines, one address doubles as a source and destination. Usually, we use dest to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. We discuss the Pentium processor details in the next few chapters. Table 2.2 gives some sample instructions of a two-address machine. On these machines, the C statement

A=B+C*D-E+F+A

is converted to the following code:

load T,C ; T = C
mult T,D ; T = C*D
add T,B ; T = B + C*D
sub T,E ; T = B + C*D - E
add T,F ; T = B + C*D - E + F
add A,T ; A = B + C*D - E + F +A

2.3.3 One-Address Machines

In the early machines, when memory was expensive and slow, a special set of registers was used to provide one of the input operands as well as to receive the result of the operation. Because of this, these registers are called the accumulators. In most machines, there is just a single accumulator register. This kind of design, called the accumulator machines, makes sense if memory is expensive. In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no

need to store the result in memory: this reduces the need for larger memory and speeds up the computation by reducing the number of memory accesses.

2.3.4 Zero-Address Machines

In zero-address machines, the locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in–first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. We discuss the stack later in this book (see Section 5.1 on page 118). All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack.

The Load/Store Architecture

In this architecture, instructions operate on values stored in internal processor registers. Only load and store instructions move data between the registers and memory. Table 2.3 gives some sample instructions for the load/store machines. On these machines, the C statement

A=B+C*D-E+F+A

is converted to the following code:

```
load R1,B ; load B
load R2,C ; load C
load R3,D ; load D
load R4,E ; load E
load R5,F ; load F
load R6,A ; load A
mult R2,R2,R3 ; R2 = C*D
add R2,R2,R1 ; R2 = B + C*D
sub R2,R2,R4 ; R2 = B + C*D – E
add R2,R2,R5 ; R2 = B + C*D - E + F
add R2,R2,R6 ; R2 = B + C*D - E + F + A
store A,R2 ; store the result in A
```

Table 2.3 Sample Load/Store Machine Instructions

Instruction		Semantics
`load`	`Rd,addr`	Loads the `Rd` register with the value at address `addr`
`store`	`addr,Rs`	Stores the value in `Rs` register at address `addr`
`add`	`Rd,Rs1,Rs2`	Adds the two values in `Rs1` and `Rs2` registers and places the result in `Rd` register
`sub`	`Rd,Rs1,Rs2`	Subtracts the value in `Rs2` from that in `Rs1` and places the result in `Rd` register
`mult`	`Rd,Rs1,Rs2`	Multiplies the two values in `Rs1` and `Rs2` and places the result in `Rd` register

2.3.6 Processor Registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. The registers can be divided into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set. The number of addresses used in instructions partly influences the number of data registers and their use. For example, in three- and two-address machines, there is no need for the internal data registers. However, having a few internal registers improves performance by cutting down the number of memory accesses required to execute a program. RISC processors typically have a large number of registers.

Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.

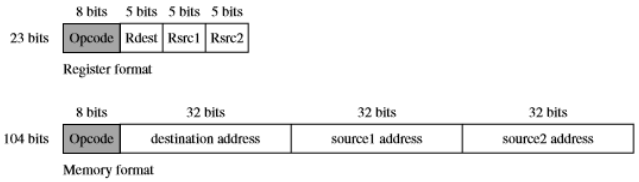


Figure 2.5 A comparison of the instruction size when the operands are in registers versus memory.

2.4 Flow of Control

Program execution, by default, proceeds sequentially. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC. When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in the MIPS processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern. However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as if-then-else and while statements to alter the execution behavior based on some run-time conditions.

2.4.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instructions in processors such as the Pentium are also called the jump instructions. Processors support two types of branches: unconditional and conditional. In both cases, the transfer control mechanism remains the same (see Figure 2.6).

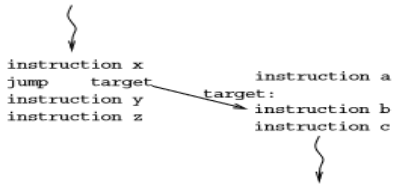


Figure 2.6 Control flow in branching.

Unconditional Branch

The simplest of the branch instructions is the unconditional branch, which transfers control to the specified target. Here is an example branch instruction:

branch target

Specification of the target address can be done in one of two ways: absolute address or PC relative address. In the former, the actual address of the target instruction is given. In the PC-relative method, the target address is specified relative to the PC contents. Most processors support absolute address for unconditional branches. Others support both formats. For example, MIPS processors support absolute address-based branch by

j target

and PC-relative unconditional branch by

b target

In fact, the last instruction is an assembly language instruction. The processor only supports the j instruction. If the absolute address is used, the processor transfers control by simply loading the specified target address into the PC register. If PC-relative addressing is used, the specified target address is added to the PC contents, and the result is placed in the PC. In either case, since the PC indicates the next instruction address, the processor will fetch the instruction at the intended target address. The main advantage of using the PC-relative address is that we can move the code from one block of memory to another without changing the target addresses. This type of code is called relocatable code. Relocatable code is not possible with absolute addresses.

Conditional Branch

In conditional branches, the jump is taken only if a specified condition is satisfied. For example, we may want to take a branch only if two values are equal. Such conditional branches are handled in one of two basic ways:

• Set-Then-Jump: In this design, testing for the condition and branching are separated. To achieve communication between these two instructions, a condition code register is used. The Pentium follows this design, which uses a flags register to record the result of the test condition. It uses a compare (cmp) instruction to test the condition. This instruction sets the various flag bits to indicate the relationship between the two compared values. Then we can use a conditional jump instruction to jump to the target location if the specified condition bit is set.

• Test-and-Jump: Most processors combine the testing and branching into a single instruction. We use the MIPS processor to illustrate the principle involved

in this strategy. The MIPS processor provides several branch instructions that test and branch. The one that we are interested in here is the branch on equal instruction shown below:

beq Rsrc1,Rsrc2,target

This conditional branch instruction tests the contents of the two registers Rsrc1 and Rsrc2 for equality and transfers control to target if equal.

Some processors maintain registers to record the condition of the arithmetic and logical operations. These are called condition code registers. These registers keep a record of the status of the last arithmetic/logical operation. For example, when we add two 32-bit integers, it is possible that the sum might require more than 32 bits. This is the overflow condition that the system should record. Normally, a bit in the condition code register is set to indicate this overflow condition. The MIPS processors, for example, do not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium uses condition registers, which are called the flags register. Some instruction sets provide branches based on comparisons to zero.

2.4.2 Procedure Calls

The use of procedures facilitates modular programming. Procedure calls are slightly different from the branches. Branches are one-way jumps: once the control has been transferred to the target location, computation proceeds from that location, as shown in Figure 2.6. In procedure calls, we have to return control to the calling program after executing the procedure. Control is returned to the instruction following the call instruction, as shown in Figure 2.7.

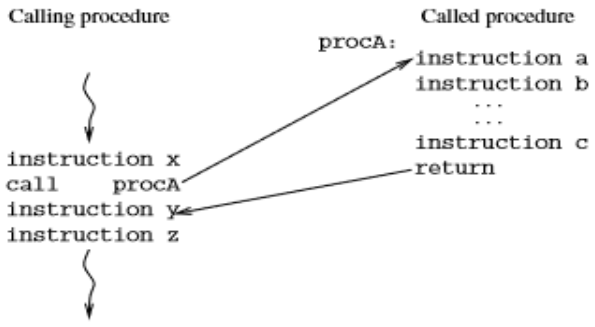


Figure 2.7 Control flow in procedure calls.

From Figures 2.6 and 2.7, you will notice that the branches and procedure calls are similar in their initial control transfer. For procedure calls, we need to return to the instruction following the procedure call. This return requires two pieces of information:

• End of Procedure: We have to indicate the end of the procedure so that the control can be returned. This is normally done by a special return instruction. For example, the Pentium uses ret and the MIPS uses the jr instruction to return from a procedure. We do the same in high-level languages as well. For example, in C, we use the return statement to indicate an end of procedure execution. High-level languages allow a default fall-through mechanism. That is, if we don't

explicitly specify the end of a procedure, control is returned at the end of the block. In the assembly language, we must specify the end of a procedure by using the return instruction.

• Return Address: How does the processor know where to return after completing a procedure? This piece of information is normally stored when the procedure is called. Thus, when a procedure is called, it not only modifies the PC as in the branch instruction, but also stores the return address. Where does it store the return address? Two main places are used: a special register or the stack. Both MIPS and Pentium processors store the address of the instruction following the call instruction.

The Pentium uses the stack to store the return address. Thus, each procedure call involves pushing the return address onto the stack before control is transferred to the procedure code. The return instruction retrieves this value from the stack to send control back to the instruction following the procedure call.

MIPS processors allow any general-purpose register to store the return address. The return statement specifies this register. The format of the return statement is

jr $ra

where ra is the register that contains the return address.

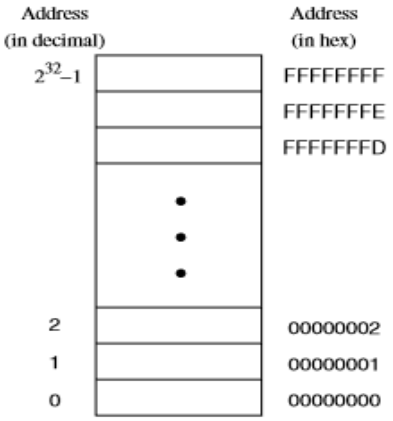


Figure 2.8 Logical view of the system memory.

Parameter Passing

The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method, parameters are placed in the processor's internal registers and the called procedure will read the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to read them off the stack.

The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in the MIPS processors. The Pentium,

due to the small number of registers, tends to use the stack for parameter passing.

2.5 Memory

The memory of a computer system consists of tiny electronic switches, with each switch set in one of two states: open or closed. It is, however, more convenient to think of these states as 0 and 1 rather than open and closed. A single such switch can be used to represent two (i.e., binary) numbers: a zero and a one. Thus, each switch can represent a binary digit or bit, as it is known. The memory unit consists of millions of such bits. In order to make memory more manageable, bits are organized into groups of eight bits called bytes. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory can be identified by its sequence number starting with 0, as shown in Figure 2.8. This is referred to as the memory address of the byte. Such memory is called byte addressable memory. The Pentium can address up to 4 GB (232 bytes) of main memory (see Figure 2.8). This magic number comes from the fact that the address bus of the Pentium has 32 address lines. This number is referred to as the memory address space (MAS). The memory address space of a system is determined by the address bus width of the processor used in the system. Typically, 32-bit processors support 32-bit addresses.

2.5.1 Two Basic Memory Operations

The memory unit supports two fundamental operations: read and write. The read operation reads a previously stored data and the write operation stores a value in memory. Both of these operations require an address in memory from which to read a value or to which to write a value. In addition, the write operation requires specification of the data to be written. The block diagram of the memory unit is shown in Figure 2.9. The address and data of the memory unit are connected to the address and data buses, respectively. The read and write signals come from the control bus. Two metrics are used to characterize memory. Access time refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the memory cycle time, which refers to the minimum time between successive memory operations.

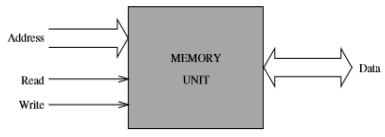


Figure 2.9 Block diagram of the system memory.

Memory transfer rates can be measured by the bandwidth metric. It specifies the number of bytes transferred per second. For example, a Pentium system with the PC133 memory can transfer 8 bytes at a frequency of 133 times per second. This gives us a bandwidth of 8 * 133 = 1064 MB/s. The read operation is non-destructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write

operation, on the other hand, is destructive, as writing a value into a location destroys the old contents of that memory location.

Steps in a typical read cycle

1. Place the address of the location to be read on the address bus,
2. Activate the memory read control signal on the control bus,
3. Wait for the memory to retrieve the data from the addressed memory location and place it on the data bus,
4. Read the data from the data bus,
5. Drop the memory read control signal to terminate the read cycle.

A simple Pentium read cycle takes three clock cycles. During the first clock cycle, steps 1 and 2 are performed. The Pentium waits until the end of the second clock and reads the data and drops the read control signal. If the memory is slower (and therefore cannot supply data within the specified time), the memory unit indicates its inability to the processor and the processor waits longer for the memory to supply data by inserting wait cycles. Note that each wait cycle introduces a waiting period equal to one system clock period and thus slows down the system operation.

Steps in a typical write cycle

1. Place the address of the location to be written on the address bus,
2. Place the data to be written on the data bus,
3. Activate the memory write control signal on the control bus,
4. Wait for the memory to store the data at the addressed location,
5. Drop the memory write signal to terminate the write cycle.

As with the read cycle, the Pentium requires three clock cycles to perform a simple write operation. During the first clock cycle, steps 1 and 3 are done. Step 2 is performed during the second clock cycle. Pentium gives memory time until the end of the second clock and drops the memory write signal. If the memory cannot write data at the maximum processor rate, wait cycles can be introduced to extend the write cycle.

2.5.2 Types of Memory

The memory unit can be implemented using a variety of memory chips—different speeds, different manufacturing technologies, and different sizes. The two basic types of memory are the read-only memory and read/write memory. A basic property of memory systems is that they are random access memories in that accessing any memory location (for reading or writing) takes the same time. Contrast this with data stored on a magnetic tape. Access time on the tape depends on the location of the data. Volatility is another important property of a memory unit. A volatile memory requires power to retain its contents. A nonvolatile memory can retain its values even in the absence of power.

Read-Only Memories

Read-only memory (ROM) allows only read operations to be performed. As the name suggests, we cannot write into this memory. The main advantage of ROM is that it is nonvolatile. Most ROM is factory-programmed and cannot be altered.

The term programming in this context refers to writing values into a ROM. This type of ROM is cheaper to manufacture in large quantities than other types of ROM. The program that controls the standard input and output functions (called BIOS), for instance, is kept in ROM. Current systems use the flash memory rather than a ROM.

Other types of ROM include programmable ROM(PROM) and erasable PROM(EPROM). PROM is useful in situations where the contents of ROM are not yet fixed. For instance, when the program is still in the development stage, it is convenient for the designer to be able to program the ROM locally rather than at the time of manufacture.

In PROM, a fuse is associated with each bit cell. If the fuse is on, the bit cell supplies a 1 when read. The fuse has to be burned to read a 0 from that bit cell. When PROM is manufactured, its contents are all set to 1. To program PROM, selective fuses are burned (to introduce 0's) by sending high current. This is the writing process and is not reversible (i.e., a burned fuse cannot be restored).

EPROM offers further flexibility during system prototyping. Contents of an EPROM can be erased by exposing it to ultraviolet light for a few minutes. Once erased, the EPROM can be reprogrammed.

Electrically erasable PROMs (EEPROMs) allow further flexibility. By exposing to ultraviolet light, we erase all the contents of an EPROM. EEPROMs, on the other hand, allow the user to selectively erase contents. Furthermore, erasing can be done in place; there is no need to place it in a special ultraviolet chamber. Flash memory is a special kind of EEPROM. One main difference between the EEPROM and flash memory lies in how the memory contents are erased. The EEPROM is byte-erasable whereas flash memory is block-erasable. Thus, writing in the flash memory involves erasing a block and rewriting it. Current systems use flash memory for BIOS so that changing BIOS versions is fairly straightforward (you just have to "flash" the new version). Flash memory is also becoming very popular as a removable media. The SmartMedia, CompactFlash, and Sony's Memory Stick are all examples of various forms of removable flash media.

Read/Write Memory

Read/write memory is commonly referred to as random access memory (RAM), even though ROM is also random access memory. This terminology is so entrenched in the literature that we follow it here with a cautionary note that RAM actually refers to RWM. Read/write memory can be divided into static and dynamic categories. Static random access memory (SRAM) retains the data, once written, without further manipulation so long as the source of power holds its value. SRAM is typically used for implementing the processor registers and cache memories. The bulk of main memory in a typical computer system, however, consists of dynamic random access memory (DRAM). DRAM is a complex memory device that uses a tiny capacitor to store a bit. A charged capacitor represents 1 bit. Since capacitors slowly lose their charge due to

leakage, they must be refreshed periodically to replace the charges representing 1 bit. A typical refresh period is about 64 ms. Reading from DRAM involves testing to see if the corresponding bit cells are charged. Unfortunately, this test destroys the charges on the bit cells. Thus, DRAM is a destructive read memory.
For proper operation, a read cycle is followed by a restore cycle. As a result, the DRAM cycle time, the actual time necessary between accesses, is typically about twice the read access time, which is the time necessary to retrieve a datum from the memory. Several types of DRAM chips are available. We briefly describe some of the most popular types next.

FPM DRAMs

Fast page-mode (FPM) DRAMs are an improvement over the previous generation DRAMs. FPM DRAMs exploit the fact that we access memory sequentially, most of the time.

OVERVIEW OF ASSEMBLY LANGUAGE

Assembly Language Statements

Assembly language programs are created out of three different classes of statements. Statements in the first class tell the processor what to do. These statements are called executable instructions. Each executable instruction consists of an operation code (opcode for short). Executable instructions cause the assembler to generate machine language instructions.

The second class of statements provides information to the assembler on various aspects of the assembly process. These instructions are called assembler directives or pseudo-ops. Assembler directives are nonexecutable and do not generate any machine language instructions. The last class of statements, called macros, are used as a shorthand notation for a group of statements. Macros permit the assembly language programmer to name a group of statements and refer to the group by the macro name. During the assembly process, each macro is replaced by the group of statements that it represents and assembled in place. This process is referred to as macro expansion. We use macros to provide the basic input and output capabilities to standalone assembly language programs.

Assembly language statements are entered one per line in the source file. All three classes of the assembly language statements use the same format:

[label] mnemonic [operands] [;comment]

The fields in the square brackets are optional in some statements. As a result of this format, it is common practice to align the fields to aid readability of assembly language programs. The assembler does not care about spaces between the fields. Now let us look at some sample assembly language statements.

```
repeat: inc result ; increment result by 1
```

The label repeat can be used to refer to this particular statement. The mnemonic inc indicates increment operation to be done on the data stored in memory at a location identified by result. The following assembler directive defines a constant CR. The ASCII carriage-return value is assigned to it by the EQU directive.

```
CR  EQU 0DH ;carriage-return character
```

Certain reserved words that have special meaning to the assembler are not allowed as labels. These include mnemonics such as inc and EQU.

It is a good programming practice to use blank lines and spaces to improve the readability of assembly language programs.

```
repeat: inc result ; increment result by 1
```

Data Allocation

In high-level languages, allocation of storage space for variables is done indirectly by specifying the data types of each variable used in the program. For example, in C, the following declarations allocate different amounts of storage space for each variable.

```
char response; /* allocates 1 byte */
 int value;  /* allocates 4 bytes */
 float total;  /* allocates 4 bytes */
double temp;  /* allocates 8 bytes */
```

These variable declarations not only specify the amount of storage required, but also indicate how the stored bit pattern should be interpreted. As an example, consider the following two statements in C:

```
unsigned value_1;
 int value_2;
```

Both variables use four bytes of storage. However, the bit pattern stored in them would be interpreted differently. For instance, the bit pattern (8FF08DB9H)

1000 1111 1111 0000 1000 1101 1011 1001

stored in the four bytes allocated for value_1 is interpreted as representing $+2.4149\times10^9$, while the same bit pattern stored in value_2 would be interpreted as -1.88006×10^9.

The general format of the storage allocation statement for initialized data is

[variable-name] define-directive initial-value [,initial-value], ⋯

The square brackets indicate optional items. The variable-name is used to identify the storage space allocated. The assembler associates an offset value for each variable name defined in the data segment. Note that no colon (:) follows the variable name (unlike a label identifying an executable statement). The define directive takes one of the five basic forms:

DB Define Byte ; allocates 1 byte
DW Define Word ; allocates 2 bytes
DD Define Doubleword ; allocates 4 bytes
DQ Define Quadword ; allocates 8 bytes
DT Define Ten Bytes ; allocates 10 bytes

We can also use numbers to initialize. For example,

```
sorted DB 79H
```

or

```
sorted DB 1111001B
```

is equivalent to

```
sorted DB 'y'
```

The following data definition statement allocates two bytes of contiguous storage and initializes it to 25159.

```
value DW 25159
```

Uninitialized Data

To reserve space for uninitialized data, we use RESB, RESW, and so on.

Each reserve directive takes a single operand that specifies the number of units of space (bytes, words, ...) to be reserved. There is a reserve directive for each define directive.
RESB Reserve a Byte
RESW Reserve a Word
RESD Reserve a Doubleword
RESQ Reserve a Quadword
REST Reserve Ten Bytes
Here are some examples:
response RESB 1
buffer RESW 100
total RESD 1

Multiple Definitions

Assembly language programs typically contain several data definition statements. For example, look at the following assembly language program fragment:

```
sort DB 'y' ; ASCII of y = 79H
value DW 25159 ; 25159D = 6247H
total DD 542803535 ; 542803535D = 205A864FH
```

Multiple data definitions can be abbreviated. For example, the following sequence of eight DB directives

```
message DB 'W'
DB 'E'
DB 'L'
DB 'C'
DB 'O'
DB 'M'
DB 'E'
DB '!'
```

can be abbreviated as

```
message DB 'W','E','L','C','O','M','E','!'
```

or even more compactly as

```
message DB 'WELCOME!'
```

Where Are the Operands?

Assembly language programs can be thought of as consisting of two logical parts: data and code. Most assembly language instructions require operands. There are several ways to specify the location of the operands. These are called the addressing modes.

An operand required by an instruction may be in any one of the following locations:

- in a register internal to the processor;
- in the instruction itself;
- in main memory (usually in the data segment);

• at an I/O port

Specification of an operand that is in a register is called register addressing mode, while immediate addressing mode refers to specifying an operand that is part of the instruction.

Register Addressing Mode

In this addressing mode, processor's internal registers contain the data to be manipulated by the instruction. For example, the instruction

mov EAX,EBX

requires two operands and both are in the processor registers. The syntax of the mov instruction is

mov destination,source

The mov instruction copies contents of source to destination. The contents of source are not destroyed. Thus,

mov EAX,EBX

Immediate Addressing Mode

In this addressing mode, data are specified as part of the instruction itself. As a result, even though the data are in memory, it is located in the code segment, not in the data segment. This addressing mode is typically used in instructions that require at least two data items to manipulate.

In the following example,

mov AL,75

the source operand 75 is specified in the immediate addressing mode and the destination operand is specified in the register-addressing mode. Such instructions are said to use mixed mode addressing.

Direct Addressing Mode

Operands specified in a memory-addressing mode require access to the main memory, usually to the data segment. As a result, they tend to be slower than either of the two previous addressing modes.

The examples that follow assume the following data definition statements in the program.

response DB 'Y' ; allocates a byte, initializes to Y

table1 TIMES 20 DD 0 ; allocates 80 bytes, initializes to 0

name1 DB 'Jim Ray' ; 7 bytes are initialized to Jim Ray

Indirect Addressing Mode

The direct addressing mode can be used in a straightforward way but is limited to accessing simple variables. For example, it is not useful in accessing the second element of table1 as in the following C statement:

table1[1] = 99

The indirect addressing mode remedies this deficiency. In this addressing mode, the offset or effective address of the data is in one of the general registers. For this

reason, this addressing mode is sometimes referred to as the register indirect addressing mode.

Data Transfer Instructions

The MOV Instruction

The mov instruction can take one of the following five forms:

mov register, register
mov register, immediate
mov memory, immediate
mov register, memory
mov memory, register

Here are some example mov statements:

mov [response],BH
mov EDX,[table1]
mov [name1+4],'K'

The XCHG Instruction

The xchg instruction exchanges 8-, 16-, or 32-bit source and destination operands. The syntax is similar to that of the mov instruction. Some examples are

xchg EAX,EDX
xchg [response],CL
xchg [total],DX

The XLAT Instruction

The xlat (translate) instruction can be used to perform character translation. The format of this instruction is shown below:

xlatb

To use this instruction, the EBX register must to be loaded with the starting address of the translation table and AL must contain an index value into the table. The xlat instruction adds contents of AL to EBX and reads the byte at the resulting address.

Overview of Assembly Language Instructions

This section briefly reviews some of the remaining assembly language instructions.

The INC and DEC Instructions

These instructions can be used to either increment or decrement the operands by one. The inc (INCrement) instruction adds one to its operand and the dec (DECrement) instruction subtracts one from its operand. Both instructions require a single operand. The operand can be either in a register or in memory. It does not make sense to use an immediate operand such as inc 55 or dec 109. The general format of these instructions is

inc destination
dec destination

where destination may be an 8-, 16- or 32-bit operand.

inc EBX ; increment 32-bit register
dec DL ; decrement 8-bit register

The ADD Instruction

The add instruction can be used to add two 8-, 16- or 32-bit operands. The syntax is

add destination, source

As with the mov instruction, add can also take the five basic forms depending on how the two operands are specified. The semantics of the add instruction are

destination = destination + source

The SUB and CMP Instructions

The sub (SUBtract) instruction can be used to subtract two 8-, 16- or 32-bit numbers. The syntax is

sub destination, source

The source operand is subtracted from the destination operand and the result is placed in the destination.

destination = destination – source

The cmp (CoMPare) instruction is used to compare two operands (equal, not equal, and so on). The cmp instruction performs the same operation as the sub instruction except that the result of subtraction is not saved. Thus, cmp does not disturb the source and destination operands.

Unconditional Jump

The unconditional jump instruction jmp, as its name implies, tells the processor that the next instruction to be executed is located at the label that is given as part of the instruction. This jump instruction has the form

jmp label

The following example

```
mov EAX,1
 inc_again:
 inc EAX
jmp inc_again
 mov EBX,EAX ...
```

results in an infinite loop incrementing EAX repeatedly The instruction

mov EBX,EAX

and all the instructions following it are never executed!

Conditional Jump

In conditional jump instructions, program execution is transferred to the target instruction only if the specified condition is satisfied. The general format is

j<cond> label

Logical Instructions

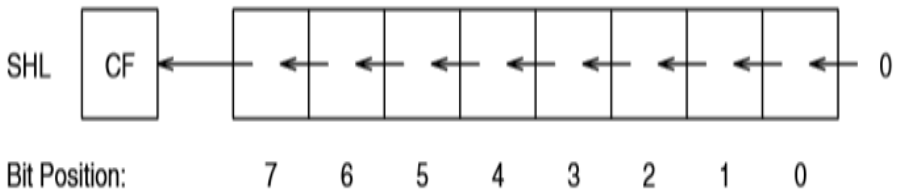
The Pentium instruction set provides several logical instructions including and, or, xor, and not. The syntax of these instructions is

```
and destination, source
or destination, source
 xor destination, source
 not destination
```

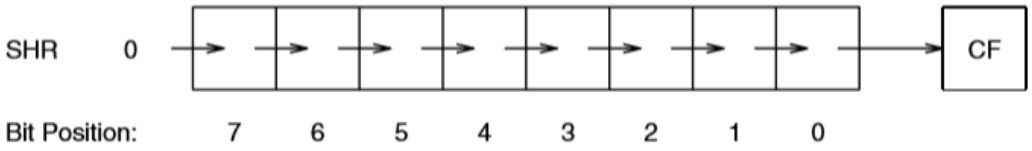
Shift Instructions

The Pentium instruction set includes several shift instructions. We discuss the following two instructions here: shl (SHift Left) and shr (SHift Right).

The shl instruction can be used to left-shift a destination operand. Each shift to the left by one bit position causes the leftmost bit to move to the carry flag (CF). The vacated rightmost bit is filled with a zero. The bit that was in CF is lost as a result of this operation.



The shr instruction works similarly but shifts bits to the right as shown below:



The general formats of these instructions are

shl destination, count shr destination, count

shl destination, CL shr destination, CL

Rotate Instructions

A drawback with the shift instructions is that the bits shifted out are lost. There may be situations where we want to keep these bits. The rotate family of instructions provides this facility. These instructions can be divided into two types: rotate without involving the carry flag, or through the carry flag.

Rotate Without Carry

There are two instructions in this group:

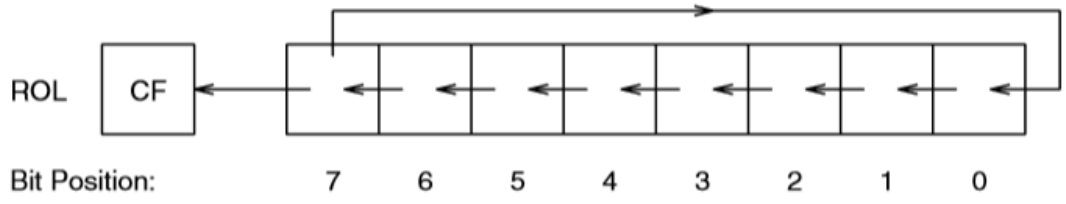
rol (ROtate Left) ror (ROtate Right)

The format of these instructions is similar to the shift instructions and is given below:

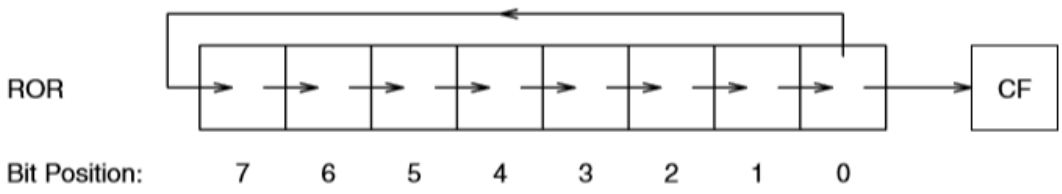
rol destination, count ror destination, count

rol destination, CL ror destination, CL

The rol instruction performs left rotation with the bits falling off on the left placed on the right side, as shown below:



The ror instruction performs right rotation as shown below:



Rotate Through

Carry The instructions

rcl (Rotate through Carry Left)

rcr (Rotate through Carry Right)

include the carry flag in the rotation process. That is, the bit that is rotated out at one end goes into the carry flag and the bit that was in the carry flag is moved into the vacated bit, as shown below: