FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or
Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ORGANIZATION
AND EMBEDDED SYSTEMS

SIXTH EDITION

Carl Hamacher
Queen's University

Zvonko Vranesic
University of Toronto

Safwat Zaky
University of Toronto

Naraig Manjikian
Queen's University

Mc Graw Hill
Connect
Learn
Succeed™

https://hemanthrajhemu.github.io

CONTENTS

BASIC STRUCTURE OF COMPUTERS

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- The different types of computers
- The basic structure of a computer and its operation
- Machine instructions and their execution
- Number and character representations
- Addition and subtraction of binary numbers
- Basic performance issues in computer systems
- A brief history of computer development

This book is about computer organization. It explains the function and design of the various units of digital computers that store and process information. It also deals with the input units of the computer which receive information from external sources and the output units which send computed results to external destinations. The input, storage, processing, and output operations are governed by a list of instructions that constitute a program.

Most of the material in the book is devoted to *computer hardware* and *computer architecture*. Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities. Computer architecture encompasses the specification of an instruction set and the functional behavior of the hardware units that implement the instructions.

Many aspects of programming and software components in computer systems are also discussed in the book. It is important to consider both hardware and software aspects of the design of the various computer components in order to gain a good understanding of computer systems.

1.1   COMPUTER TYPES

Since their introduction in the 1940s, digital computers have evolved into many different types that vary widely in size, cost, computational power, and intended use. Modern computers can be divided roughly into four general categories:

• *Embedded computers* are integrated into a larger device or system in order to automatically monitor and control a physical process or environment. They are used for a specific purpose rather than for general processing tasks. Typical applications include industrial and home automation, appliances, telecommunication products, and vehicles. Users may not even be aware of the role that computers play in such systems.

• *Personal computers* have achieved widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use. They support a variety of applications such as general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing. A number of classifications are used for personal computers. *Desktop computers* serve general needs and fit within a typical personal workspace. *Workstation computers* offer higher computational capacity and more powerful graphical display capabilities for engineering and scientific work. Finally, *Portable* and *Notebook computers* provide the basic features of a personal computer in a smaller lightweight package. They can operate on batteries to provide mobility.

• *Servers* and *Enterprise systems* are large computers that are meant to be shared by a potentially large number of users who access them from some form of personal computer over a public or private network. Such computers may host large databases and provide information processing for a government agency or a commercial organization.

• *Supercomputers* and *Grid computers* normally offer the highest performance. They are the most expensive and physically the largest category of computers. Supercomputers are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work. They have a high cost. Grid computers provide a more cost-effective alternative. They combine a large number of personal computers and

disk storage units in a physically distributed high-speed network, called a grid, which is managed as a coordinated computing resource. By evenly distributing the computational workload across the grid, it is possible to achieve high performance on large applications ranging from numerical computation to information searching.

There is an emerging trend in access to computing facilities, known as *cloud computing*. Personal computer users access widely distributed computing and storage server resources for individual, independent, computing needs. The Internet provides the necessary communication facility. Cloud hardware and software service providers operate as a utility, charging on a pay-as-you-use basis.

1.2   FUNCTIONAL UNITS

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. Later chapters will provide more details on individual units and their interconnections. We refer to the
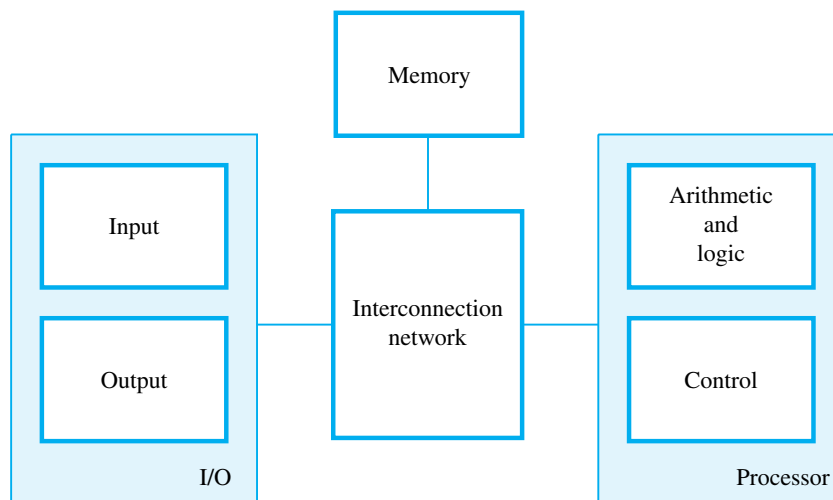
Figure 1.1   Basic functional units of a computer.

arithmetic and logic circuits, in conjunction with the main control circuits, as the *processor*. Input and output equipment is often collectively referred to as the *input-output* (I/O) unit.

We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. *Instructions*, or *machine instructions*, are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed

A *program* is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. *Data* are numbers and characters that are used as operands by the instructions. Data are also stored in the memory.

The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states. Each instruction, number, or character is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1, represented by the two stable states. Numbers are usually represented in positional binary notation, as discussed in Section 1.4. Alphanumeric characters are also expressed in terms of binary codes, as discussed in Section 1.5.

1.2.1   INPUT UNIT

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

1.2.2   MEMORY UNIT

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory
Primary memory, also called *main memory*, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The

memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called *words*. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the *word length* of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct *address* is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a *random-access memory* (RAM). The time required to access one word is called the *memory access time*. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a *cache*, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Now, suppose a number of instructions are executed repeatedly as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent *secondary storage* is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including *magnetic disks*, *optical disks* (DVD and CD), and *flash memory devices*.

1.2.3   ARITHMETIC AND LOGIC UNIT

Most computer operations are executed in the *arithmetic and logic unit* (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication,

division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elements called *registers*. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

1.2.4   OUTPUT UNIT

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a *printer*. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name *input/output* (I/O) unit in many cases.

1.2.5   CONTROL UNIT

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the *timing signals* that govern the transfers and determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

* The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
* Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
* Processed information leaves the computer through an output unit.
* All activities in the computer are directed by the control unit.

1.3   BASIC OPERATIONAL CONCEPTS

In Section 1.2, we stated that the activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.

A typical instruction might be

<div align="center">Load   R2, LOC</div>

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction

<div align="center">Add   R4, R2, R3</div>

adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

<div align="center">Store   R4, LOC</div>

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

Figure 1.2 shows how the memory and the processor can be connected. It also shows some components of the processor that have not been discussed yet. The interconnections between these components are not shown explicitly since we will only discuss their functional characteristics here. Chapter 5 describes the details of the interconnections as part of processor organization.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The *instruction register* (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The *program counter* (PC) is another specialized register. It
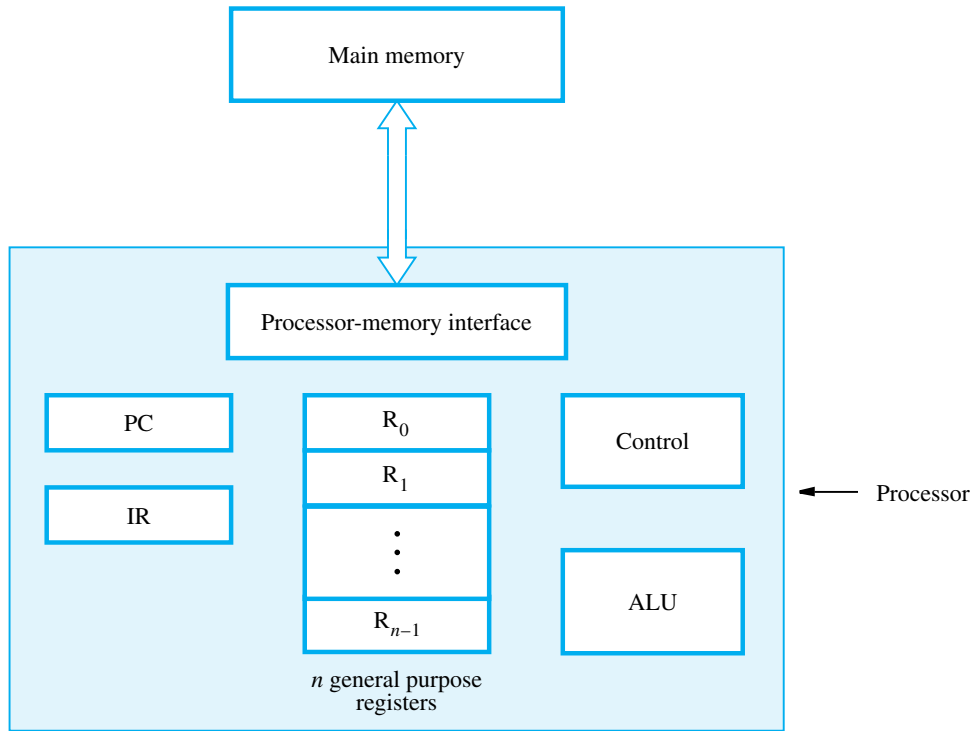
Figure 1.2   Connection between the processor and the main memory.

contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC *points* to the next instruction that is to be fetched from the memory. In addition to the IR and PC, Figure 1.2 shows *general-purpose registers* R_0 through R_{n-1}, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing. The roles of the general-purpose registers are explained in detail in Chapter 2.

The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register. If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Let us now consider some typical operating steps. A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage through the input unit. Execution of the program begins when the PC is set to point to the

first instruction of the program. The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be interpreted and executed.

Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register. If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an *interrupt* signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an *interrupt-service routine*. Because such diversions may alter the internal state of the processor, its state must be saved in the memory before servicing the interrupt request. Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

This section has provided an overview of the operation of a computer. Detailed discussion of these concepts is given in subsequent chapters, first from the point of view of the programmer in Chapters 2, 3, and 4, and then from the point of view of the hardware designer in later chapters.

1.4   NUMBER REPRESENTATION AND ARITHMETIC OPERATIONS

The most natural way to represent a number in a computer system is by a string of bits, called a binary number. We will first describe binary number representations for integers as well as arithmetic operations on them. Then we will provide a brief introduction to the representation of floating-point numbers.

1.4.1   INTEGERS

Consider an n-bit vector

$$B = b_{n-1} \ldots b_1 b_0$$

where $b_i = 0$ or 1 for $0 \leq i \leq n - 1$. This vector can represent an unsigned integer value $V(B)$ in the range 0 to $2^n - 1$, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We need to represent both positive and negative numbers. Three systems are used for representing such numbers:

- Sign-and-magnitude
- 1's-complement
- 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Figure 1.3 illustrates all three representations using 4-bit numbers. Positive values have identical representations in all systems, but negative values have different representations. In the *sign-and-magnitude* system, negative values are represented by changing the most

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Figure 1.3   Binary, signed-integer representations.

significant bit (b_3 in Figure 1.3) from 0 to 1 in the B vector of the corresponding positive value. For example, $+5$ is represented by 0101, and -5 is represented by 1101.

In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for -3 is obtained by complementing each bit in the vector 0011 to yield 1100. The same operation, bit complementing, is done to convert a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. For n-bit numbers, this operation is equivalent to subtracting the number from $2^n - 1$. In the case of the 4-bit numbers in Figure 1.3, we subtract from $2^4 - 1 = 15$, or 1111 in binary.

Finally, in the *2's-complement* system, forming the 2's-complement of an n-bit number is done by subtracting the number from 2^n. Hence, the 2's-complement of a number is obtained by adding 1 to the 1's-complement of that number.

Note that there are distinct representations for $+0$ and -0 in both the sign-and-magnitude and 1's-complement systems, but the 2's-complement system has only one representation for 0. For 4-bit numbers, as shown in Figure 1.3, the value -8 is representable in the 2's-complement system but not in the other systems. The sign-and-magnitude system seems the most natural, because we deal with sign-and-magnitude decimal values in manual computations. The 1's-complement system is easily related to this system, but the 2's-complement system may appear somewhat unnatural. However, we will show that the 2's-complement system leads to the most efficient way to carry out addition and subtraction operations. It is the one most often used in modern computers.

Addition of Unsigned Integers

Addition of 1-bit numbers is illustrated in Figure 1.4. The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in* to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

Figure 1.4   Addition of 1-bit numbers.

Addition and Subtraction of Signed Integers

We introduced three systems for representing positive and negative numbers, or, simply, *signed numbers*. These systems differ only in the way they represent negative values. Their relative merits from the standpoint of ease of performing arithmetic operations can be summarized as follows. The sign-and-magnitude system is the simplest representation, but it is also the most awkward for addition and subtraction operations. The 1's-complement method is somewhat better. The 2's-complement system is the most efficient method for performing addition and subtraction operations.

To understand 2's-complement arithmetic, consider addition modulo N (abbreviated as mod N). A helpful graphical device for the description of addition of unsigned integers mod N is a circle with the values 0 through $N-1$ marked along its perimeter, as shown in Figure 1.5a. Consider the case $N = 16$, shown in part (*b*) of the figure. The decimal values 0 through 15 are represented by their 4-bit binary values 0000 through 1111 around the outside of the circle. In terms of decimal values, the operation $(7 + 5)$ mod 16 yields the value 12. To perform this operation graphically, locate 7 (0111) on the outside of the circle and then move 5 units in the clockwise direction to arrive at the answer 12 (1100). Similarly, $(9 + 14)$ mod $16 = 7$; this is modeled on the circle by locating 9 (1001) and moving 14 units in the clockwise direction past the zero position to arrive at the answer 7 (0111). This graphical technique works for the computation of $(a + b)$ mod 16 for any unsigned integers a and b; that is, to perform addition, locate a and move b units in the clockwise direction to arrive at $(a + b)$ mod 16.

Now consider a different interpretation of the mod 16 circle. We will reinterpret the binary vectors outside the circle to represent the signed integers from -8 through $+7$ in the 2's-complement representation as shown inside the circle.

Let us apply the mod 16 addition technique to the example of adding $+7$ to -3. The 2's-complement representation for these numbers is 0111 and 1101, respectively. To add these numbers, locate 0111 on the circle in Figure 1.5b. Then move 1101 (13) steps in the clockwise direction to arrive at 0100, which yields the correct answer of $+4$. Note that the 2's-complement representation of -3 is interpreted as an unsigned value for the number of steps to move.

If we perform this addition by adding bit pairs from right to left, we obtain

$$
\begin{array}{ccccc}
  & 0 & 1 & 1 & 1 \\
+ & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 \\
\end{array}
$$

↑
Carry-out

If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer. In fact, this is always the case. Ignoring this carry-out is a natural result of using mod N arithmetic. As we move around the circle in Figure 1.5b, the value next to 1111 would normally be 10000. Instead, we go back to the value 0000.

The rules governing addition and subtraction of n-bit signed numbers using the 2's-complement representation system may be stated as follows:
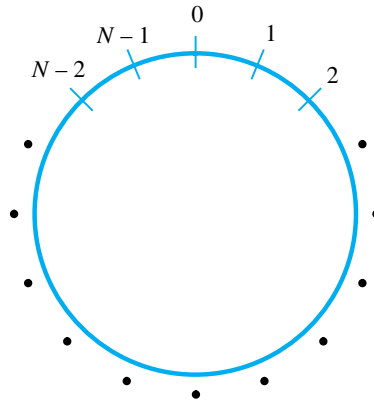
(a) Circle representation of integers mod N
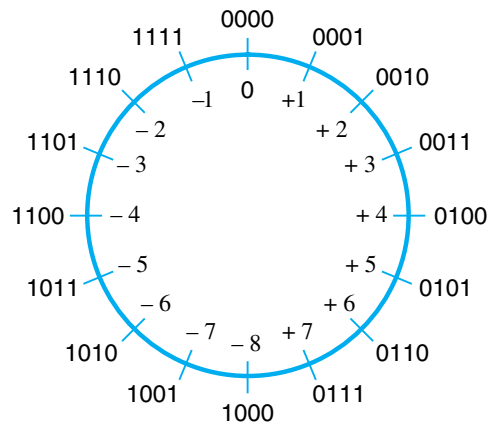
(b) Mod 16 system for 2's-complement numbers

Figure 1.5   Modular number systems and the 2's-complement system.

- To *add* two numbers, add their n-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

- To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2's-complement of Y, then add it to X using the *add* rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

Figure 1.6 shows some examples of addition and subtraction in the 2's-complement system. In all of these 4-bit examples, the answers fall within the representable range of -8 through $+7$. When answers do not fall within the representable range, we say that *arithmetic overflow* has occurred. A later subsection discusses such situations. The four addition operations (*a*) through (*d*) in Figure 1.6 follow the add rule, and the six subtraction operations (*e*) through (*j*) follow the subtract rule. The subtraction operation requires forming the 2's-complement of the subtrahend (the bottom value). This operation

Figure 1.6   2's-complement Add and Subtract operations.

is done in exactly the same manner for both positive and negative numbers. To form the 2's-complement of a number, form the bit complement of the number and add 1.

The simplicity of adding and subtracting signed numbers in 2's-complement representation is the reason why this number representation is used in modern computers. It might seem that the 1's-complement representation would be just as good as the 2's-complement system. However, although complementation is easy, the result obtained after an addition operation is not always correct. The carry-out, c_n, cannot be ignored. If $c_n = 0$, the result obtained is correct. If $c_n = 1$, then a 1 must be added to the result to make it correct. The need for this correction operation means that addition and subtraction cannot be implemented as conveniently in the 1's-complement system as in the 2's-complement system.

Sign Extension

We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2's-complement representation, the leftmost bit, which indicates the sign of the number, is a 1. A longer number with the same value is obtained by replicating the sign bit to the left as many times as needed. To see why this is correct, examine the mod 16 circle of Figure 1.5*b*. Compare it to larger circles for the mod 32 or mod 64 cases. The representations for the values -1, -2, etc., are exactly the same, with 1s added to the left. In summary, to represent a signed number in 2's-complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called *sign extension*.

Overflow in Integer Arithmetic

Using 2's-complement representation, n bits can represent values in the range -2^{n-1} to $+2^{n-1} - 1$. For example, the range of numbers that can be represented by 4 bits is -8 through $+7$, as shown in Figure 1.3. When the actual result of an arithmetic operation is outside the representable range, an *arithmetic overflow* has occurred.

When adding unsigned numbers, a carry-out of 1 from the most significant bit position indicates that an overflow has occurred. However, this is not always true when adding signed numbers. For example, using 2's-complement representation for 4-bit signed numbers, if we add $+7$ and $+4$, the sum vector is 1011, which is the representation for -5, an incorrect result. In this case, the carry-out bit from the MSB position is 0. If we add -4 and -6, we get $0110 = +6$, also an incorrect result. In this case, the carry-out bit is 1. Hence, the value of the carry-out bit from the sign-bit position is not an indicator of overflow. Clearly, overflow may occur only if both summands have the same sign. The addition of numbers with different signs cannot cause overflow because the result is always within the representable range.

These observations lead to the following way to detect overflow when adding two numbers in 2's-complement representation. Examine the signs of the two summands and the sign of the result. When both summands have the same sign, an overflow has occurred when the sign of the sum is not the same as the signs of the summands.

When subtracting two numbers, the testing method needed for detecting overflow has to be modified somewhat; but it is still quite straightforward. See Problem 1.10.

1.4.2 Floating-Point Numbers

Until now we have only considered integers, which have an implied binary point at the right end of the number, just after bit b_0. If we use a full word in a 32-bit word length computer to represent a signed integer in 2's-complement representation, the range of values that can be represented is -2^{31} to $+2^{31} - 1$. In decimal terms, this range is somewhat smaller than -10^{10} to $+10^{10}$.

The same 32-bit patterns can also be interpreted as fractions in the range -1 to $+1 - 2^{-31}$ if we assume that the implied binary point is just to the right of the sign bit; that is, between bit b_{31} and bit b_{30} at the left end of the 32-bit representation. In this case, the magnitude of the smallest fraction representable is approximately 10^{-10}.

Neither of these two *fixed-point* number representations has a range that is sufficient for many scientific and engineering calculations. For convenience, we would like to have a binary number representation that can easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In this case, the binary point is said to *float*, and the numbers are called *floating-point numbers*.

Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in the representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 3.7291×10^{-27}, -1.0341×10^2, -7.3000×10^{-14}, and so on. We say that these numbers have been given to 5 *significant digits* of precision. The *scale factors* 10^{23}, 10^{-27}, 10^2, and 10^{-14} indicate the actual position of the decimal point with respect to the significant digits. The same approach can be used to represent binary floating-point numbers in a computer, except that it is more appropriate to use 2 as the base of the scale factor. Because the base is fixed, it does not need to be given in the representation. The exponent may be positive or negative.

We conclude that a binary floating-point number can be represented by:

- a sign for the number
- some significant bits
- a signed scale factor exponent for an implied base of 2

An established international IEEE (Institute of Electrical and Electronics Engineers) standard for 32-bit floating-point number representation uses a sign bit, 23 significant bits, and 8 bits for a signed exponent of the scale factor, which has an implied base of 2. In decimal terms, the range of numbers represented is roughly $\pm 10^{-38}$ to $\pm 10^{38}$, which is adequate for most scientific and engineering calculations. The same IEEE standard also defines a 64-bit representation to accommodate more significant bits and more bits for the signed exponent, resulting in much higher precision and a much larger range of values.

Floating-point number representation and arithmetic operations on floating-point numbers are considered in detail in Chapter 9. Some of the commercial processors described in Appendices B to E include operations on floating-point numbers in their instruction sets and have processor registers dedicated to holding floating-point numbers.

1.5   Character Representation

The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange). Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes as shown in Table 1.1. It is convenient to use an 8-bit *byte* to represent and store a character. The code occupies the low-order seven bits. The high-order bit is usually set to 0. Note that the codes for the alphabetic and numeric characters are in increasing sequential order when interpreted as unsigned binary numbers. This facilitates sorting operations on alphabetic and numeric data.

The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system. This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

1.6   Performance

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language. We do not describe the details of compilers or operating systems in this book. However, Chapter 4 provides an overview of software, including a discussion of the role of compilers and operating systems. This book concentrates on the design of instruction sets, along with memory, processor, and I/O hardware, and the organization of both small and large computers. Section 1.2.2 describes how caches can improve memory performance. Some performance aspects of instruction sets are discussed in Chapter 2. In this section, we give an overview of how performance is affected by technology, as well as processor and system organization.

1.6.1   Technology

The technology of Very Large Scale Integration (VLSI) that is used to fabricate the electronic circuits for a processor on a single chip is a critical factor in the speed of execution of machine instructions. The speed of switching between the 0 and 1 states in logic circuits is largely determined by the size of the transistors that implement the circuits. Smaller transistors switch faster. Advances in fabrication technology over several decades have reduced transistor sizes dramatically. This has two advantages: instructions can be executed faster, and more transistors can be placed on a chip, leading to more logic functionality and more memory storage capacity.

Table 1.1   The 7-bit ASCII code.

Bit positions 3210	Bit positions 654							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgment
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACK	Acknowledgment	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

1.6.2   PARALLELISM

Performance can be increased by performing a number of operations in parallel. Parallelism can be implemented on many different levels.

Instruction-level Parallelism

The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. For example, the next instruction could be fetched from memory at the same time that an arithmetic operation is being performed on the register operands of the current instruction. This form of parallelism is called *pipelining*. It is discussed in detail in Chapter 6.

Multicore Processors

Multiple processing units can be fabricated on a single chip. In technical literature, the term *core* is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology *dual-core*, *quad-core*, and *octo-core* processors for chips that have two, four, and eight cores, respectively.

Multiprocessors

Computer systems may contain many processors, each possibly containing multiple cores. Such systems are called *multiprocessors*. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, and the term *shared-memory multiprocessor* is often used to make this clear. The high performance of these systems comes with much higher complexity and cost, arising from the use of multiple processors and memory units, along with more complex interconnection networks.

In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. The computers normally have access only to their own memory units. When the tasks they are executing need to share data, they do so by exchanging *messages* over a communication network. This property distinguishes them from shared-memory multiprocessors, leading to the name *message-passing multicomputers*.

Multiprocessors and multicomputers are described in Chapter 12.

1.7   HISTORICAL PERSPECTIVE

Electronic digital computers as we know them today have been developed since the 1940s. A long, slow evolution of mechanical calculating devices preceded the development of electronic computers. Here, we briefly sketch the history of computer development. A more extensive coverage can be found in Hayes [1].

In the 300 years before the mid-1900s, a series of increasingly complex mechanical devices, constructed from gear wheels, levers, and pulleys, were used to perform the basic operations of addition, subtraction, multiplication, and division. Holes on punched cards were mechanically sensed and used to control the automatic sequencing of a list of calculations, which essentially provided a programming capability. These devices enabled the computation of complete mathematical tables of logarithms and trigonometric functions as approximated by polynomials. Output results were punched on cards or printed on paper. Electromechanical relay devices, such as those used in early telephone switching systems, provided the means for performing logic functions in computers built in the late 1930s and early 1940s.

During World War II, the first electronic computer was designed and built at the University of Pennsylvania, using the vacuum tube technology developed for radios and military radar equipment. Vacuum tube circuits were used to perform logic operations and to store data. This technology initiated the modern era of electronic digital computers.

Development of the technologies used to fabricate processors, memories, and I/O units of computers has been divided into four generations: the first generation, 1945 to 1955; the second generation, 1955 to 1965; the third generation, 1965 to 1975; and the fourth generation, 1975 to the present.

1.7.1 THE FIRST GENERATION

The key concept of a stored program was introduced at the same time as the development of the first electronic digital computer. Programs and their data were located in the same memory, as they are today. This facilitates changing existing programs and data or preparing and loading new programs and data. Assembly language was used to prepare programs and was translated into machine language for execution.

Basic arithmetic operations were performed in a few milliseconds, using vacuum tube technology to implement logic functions. This provided a 100- to 1000-fold increase in speed relative to earlier mechanical and electromechanical technology. Mercury delay-line memory was used at first. I/O functions were performed by devices similar to typewriters. Magnetic core memories and magnetic tape storage devices were also developed.

1.7.2 THE SECOND GENERATION

The transistor was invented at AT&T Bell Laboratories in the late 1940s and quickly replaced the vacuum tube in implementing logic functions. This fundamental technology shift marked the start of the second generation. Magnetic core memories and magnetic drum storage devices were widely used in the second generation. Magnetic disk storage devices were developed in this generation. The earliest high-level languages, such as Fortran, were developed, making the preparation of application programs much easier. Compilers were developed to translate these high-level language programs into assembly language, which was then translated into executable machine-language form. IBM became a major computer manufacturer during this time.

1.7.3   THE THIRD GENERATION

Texas Instruments and Fairchild Semiconductor developed the ability to fabricate many transistors on a single silicon chip, called integrated-circuit technology. This enabled faster and less costly processors and memory elements to be built. Integrated-circuit memories began to replace magnetic core memories. This technological development marked the beginning of the third generation. Other developments included the introduction of microprogramming, parallelism, and pipelining. Operating system software allowed efficient sharing of a computer system by several user programs. Cache and virtual memories were developed. Cache memory makes the main memory appear faster than it really is, and virtual memory makes it appear larger. System 360 mainframe computers from IBM and the line of PDP minicomputers from Digital Equipment Corporation were dominant commercial products of the third generation.

1.7.4   THE FOURTH GENERATION

By the early 1970s, integrated-circuit fabrication techniques had evolved to the point where complete processors and large sections of the main memory of small computers could be implemented on single chips. This marked the start of the fourth generation. Tens of thousands of transistors could be placed on a single chip, and the name Very Large Scale Integration (VLSI) was coined to describe this technology. A complete processor fabricated on a single chip became known as a microprocessor. Companies such as Intel, National Semiconductor, Motorola, Texas Instruments, and Advanced Micro Devices have been the driving forces of this technology. Current VLSI technology enables the integration of multiple processors (cores) and cache memories on a single chip.

A particular form of VLSI technology, called Field Programmable Gate Arrays (FPGAs), has allowed system developers to design and implement processor, memory, and I/O circuits on a single chip to meet the requirements of specific applications, especially in embedded computer systems. Sophisticated computer-aided-design tools make it possible to develop FPGA-based products quickly. Companies such as Altera and Xilinx provide this technology, along with the required software development systems.

Embedded computer systems, portable notebook computers, and versatile mobile telephone handsets are now in widespread use. Desktop personal computers and workstations interconnected by wired or wireless local area networks and the Internet, with access to database servers and search engines, provide a variety of powerful computing platforms.

Organizational concepts such as parallelism and hierarchical memories have evolved to produce the high-performance computing systems of today as the fourth generation has matured. Supercomputers and Grid computers, at the upper end of high-performance computing, are used for weather forecasting, scientific and engineering computations, and simulations.

1.8   CONCLUDING REMARKS

This chapter has introduced basic concepts about the structure of computers and their operation. Machine instructions and programs have been described briefly. The addition and subtraction of binary numbers has been explained. Much of the terminology needed to deal with these subjects has been defined. Subsequent chapters provide detailed explanations of these terms and concepts, with an emphasis on architecture and hardware.

1.9   SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 1.1   **Problem:** List the steps needed to execute the machine instruction

Load   R2, LOC

in terms of transfers between the components shown in Figure 1.2 and some simple control commands. An overview of the steps needed is given in Section 1.3. Assume that the address of the memory location containing this instruction is initially in register PC.

Solution: The required steps are:

- Send the address of the instruction word from register PC to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register IR, where it is interpreted (decoded) by the control circuitry to determine the operation to be performed.
- Increment the contents of register PC to point to the next instruction in memory.
- Send the address value LOC from the instruction in register IR to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register R2.

Example 1.2   **Problem:** Quantify the effect on performance that results from the use of a cache in the case of a program that has a total of 500 instructions, including a 100-instruction loop that is executed 25 times. Determine the ratio of execution time without the cache to execution time with the cache. This ratio is called the *speedup*.

   Assume that main memory accesses require 10 units of time and cache accesses require 1 unit of time. We also make the following further assumptions so that we can simplify calculations in order to easily illustrate the advantage of using a cache:

- Program execution time is proportional to the total amount of time needed to fetch instructions from either the main memory or the cache, with operand data accesses being ignored.
- Initially, all instructions are stored in the main memory, and the cache is empty.
- The cache is large enough to contain all of the loop instructions.

Solution: Execution time without the cache is

$$T = 400 \times 10 + 100 \times 10 \times 25 = 29{,}000$$

Execution time with the cache is

$$T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7{,}400$$

Therefore, the speedup is

$$T/T_{cache} = 3.92$$

Problem: Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then perform addition and subtraction on each pair. Indicate whether or not overflow occurs for each case.   **Example 1.3**

(*a*) 7 and 13

(*b*) -12 and 9

Solution: The conversion and operations are:

(*a*) $7_{10} = 00111_2$ and $13_{10} = 01101_2$

   Adding these two positive numbers, we obtain 10100, which is a negative number. Therefore, overflow has occurred.

   To subtract them, we first form the 2's-complement of 01101, which is 10011. Then we perform addition with 00111 to obtain 11010, which is -6_{10}, the correct answer.

(*b*) $-12_{10} = 10100_2$ and $9_{10} = 01001_2$

   Adding these two numbers, we obtain $11101 = -3_{10}$, the correct answer.

   To subtract them, we first form the 2's-complement of 01001, which is 10111. Then we perform addition of the two negative numbers 10100 and 10111 to obtain 01011, which is a positive number. Therefore, overflow has occurred.

PROBLEMS

1.1   [E] Repeat Example 1.1 for the machine instruction

$$\text{Add} \quad \text{R4, R2, R3}$$

which is discussed in Section 1.3.

1.2   [E] Repeat Example 1.1 for the machine instruction

$$\text{Store} \quad \text{R4, LOC}$$

which is discussed in Section 1.3.

1.3   [M] (*a*) Give a short sequence of machine instructions for the task "Add the contents of memory location A to those of location B, and place the answer in location C". Instructions

$$\text{Load} \quad \text{R}i, \text{LOC}$$

and

$$\text{Store} \quad \text{R}i, \text{LOC}$$

are the only instructions available to transfer data between the memory and the general-purpose registers. Add instructions are described in Section 1.3. Do not change the contents of either location A or B.

(*b*) Suppose that Move and Add instructions are available with the formats

$$\text{Move} \quad \text{Location1, Location2}$$

and

$$\text{Add} \quad \text{Location1, Location2}$$

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (*a*)? If yes, give the sequence.

1.4   [M] (*a*) A program consisting of a total of 300 instructions contains a 50-instruction loop that is executed 15 times. The processor contains a cache, as described in Section 1.2.2. Fetching and executing an instruction that is in the main memory requires 20 time units. If the instruction is found in the cache, fetching and executing it requires only 2 time units. Ignoring operand data accesses, calculate the ratio of program execution time without the cache to execution time with the cache. This ratio is called the *speedup* due to the use of the cache. Assume that the cache is initially empty, that it is large enough to hold the loop, and that the program starts with all instructions in the main memory.

(*b*) Generalize part (*a*) by replacing the constants 300, 50, 15, 20, and 2 with the variables w, x, y, m, and c. Develop an expression for speedup.

(*c*) For the values $w = 300$, $x = 50$, $m = 20$, and $c = 2$ what value of y results in a speedup of 5?

(*d*) Consider the form of the expression for speedup developed in part (*b*). What is the upper limit on speedup as the number of loop iterations, *y*, becomes larger and larger?

1.5 **[M]** (*a*) A processor cache is discussed in Section 1.2.2. Suppose that execution time for a program is proportional to instruction fetch time. Assume that fetching an instruction from the cache takes 1 time unit, but fetching it from the main memory takes 10 time units. Also, assume that a requested instruction is found in the cache with probability 0.96. Finally, assume that if an instruction is not found in the cache it must first be fetched from the main memory into the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is called the *speedup* resulting from the presence of the cache.

(*b*) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat part (*a*) for a doubled cache size.

1.6 **[E]** Extend Figure 1.4 to incorporate both possibilities for a carry-in (0 or 1) to each of the four cases shown in the figure. Specify both the sum and carry-out bits for each of the eight new cases.

1.7 **[M]** Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case.

(*a*) 4 and 11
(*b*) 6 and 14
(*c*) −13 and 12
(*d*) −4 and 8
(*e*) −2 and −9
(*f*) −9 and −14

1.8 **[M]** Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

1.9 **[E]** A memory byte location contains the pattern 01010011. What decimal value does this pattern represent when interpreted as a binary number? What does it represent as an ASCII code?

1.10 **[E]** A way to detect overflow when adding two 2's-complement numbers is given at the end of Section 1.4.1. State how to detect overflow when subtracting two such numbers.

REFERENCES

1. J. P. Hayes, *Computer Architecture and Organization*, 3rd Ed., McGraw-Hill, New York, 1998.

This page intentionally left blank

2

INSTRUCTION SET ARCHITECTURE

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

This chapter considers the way programs are executed in a computer from the machine instruction set view-point. Chapter 1 introduced the general concept that both program instructions and data operands are stored in the memory. In this chapter, we discuss how instructions are composed and study the ways in which sequences of instructions are brought from the memory into the processor and executed to perform a given task. The addressing methods that are commonly used for accessing operands in memory locations and processor registers are also presented.

The emphasis here is on basic concepts. We use a generic style to describe machine instructions and operand addressing methods that are typical of those found in commercial processors. A sufficient number of instructions and addressing methods are introduced to enable us to present complete, realistic programs for simple tasks. These generic programs are specified at the assembly-language level, where machine instructions and operand addressing information are represented by symbolic names. A complete instruction set, including operand addressing methods, is often referred to as the *instruction set architecture* (ISA) of a processor. For the discussion of basic concepts in this chapter, it is not necessary to define a complete instruction set, and we will not attempt to do so. Instead, we will present enough examples to illustrate the capabilities of a typical instruction set.

The concepts introduced in this chapter and in Chapter 3, which deals with input/output techniques, are essential for understanding the functionality of computers. Our choice of the generic style of presentation makes the material easy to read and understand. Also, this style allows a general discussion that is not constrained by the characteristics of a particular processor.

Since it is interesting and important to see how the concepts discussed are implemented in a real computer, we supplement our presentation in Chapters 2 and 3 with four examples of popular commercial processors. These processors are presented in Appendices B to E. Appendix B deals with the Nios II processor from Altera Corporation. Appendix C presents the ColdFire processor from Freescale Semiconductor, Inc. Appendix D discusses the ARM processor from ARM Ltd. Appendix E presents the basic architecture of processors made by Intel Corporation. The generic programs in Chapters 2 and 3 are presented in terms of the specific instruction sets in each of the appendices.

The reader can choose only one processor and study the material in the corresponding appendix to get an appreciation for commercial ISA design. However, knowledge of the material in these appendices is not essential for understanding the material in the main body of the book.

The vast majority of programs are written in high-level languages such as C, C++, or Java. To execute a high-level language program on a processor, the program must be translated into the machine language for that processor, which is done by a compiler program. Assembly language is a readable symbolic representation of machine language. In this book we make extensive use of assembly language, because this is the best way to describe how computers work.

We will begin the discussion in this chapter by considering how instructions and data are stored in the memory and how they are accessed for processing.

2.1   MEMORY LOCATIONS AND ADDRESSES

We will first consider how the memory of a computer is organized. The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For

this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a *word* of information, and n is called the *word length*. The memory of a computer can be schematically represented as a collection of words, as shown in Figure 2.1.

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2. A unit of 8 bits is called a *byte*. Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section, after we have described instructions at the assembly-language level.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each location. It is customary to use numbers from 0 to $2^k - 1$, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2^k addressable locations. The 2^k addresses constitute the *address space* of the computer. For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 2^{20} (1,048,576). A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations, where 1G is 2^{30}. Other notational conventions
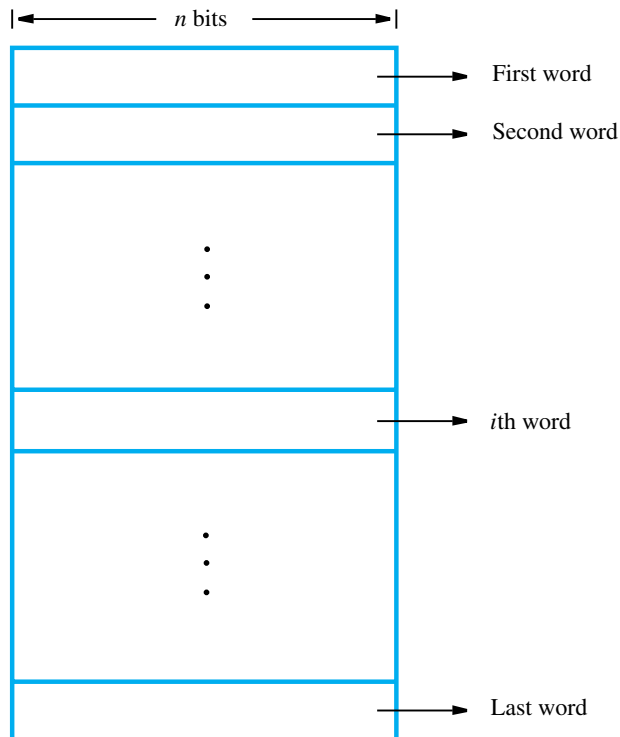
Figure 2.1   Memory words.
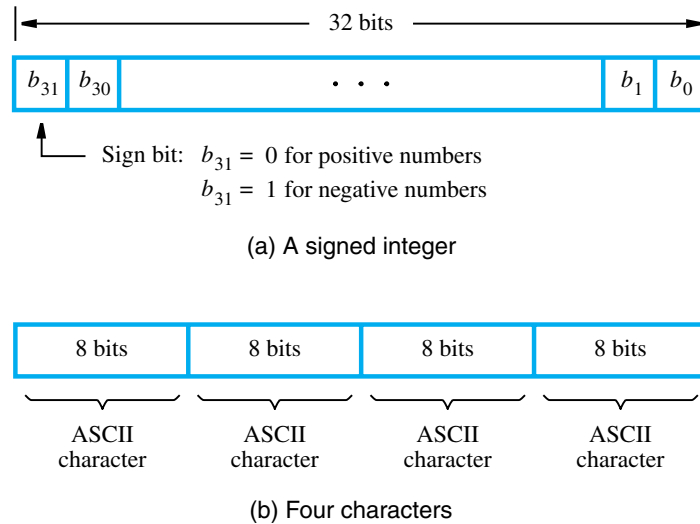
Figure 2.2   Examples of encoded information in a 32-bit word.

that are commonly used are K (kilo) for the number 2^{10} (1,024), and T (tera) for the number 2^{40}.

2.1.1 BYTE ADDRESSABILITY

We now have three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2, ... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, . . . , with each word consisting of four bytes.

2.1.2 BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.3. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, . . . , are taken as the addresses of successive words in the memory
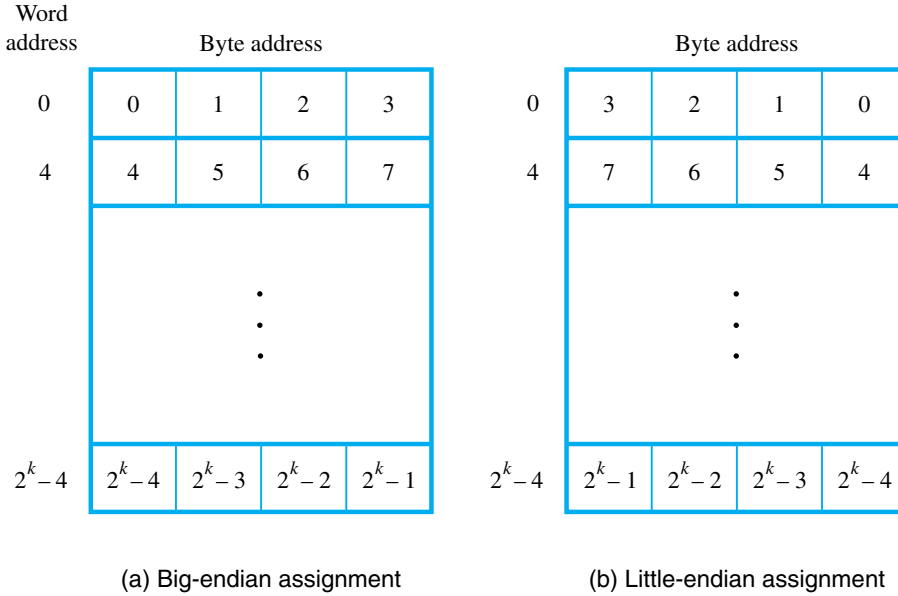
Figure 2.3 Byte and word addressing.

of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 2.2*a*. It is the most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is, $b_7, b_6, \ldots, b_0$, from left to right.

2.1.3 WORD ALIGNMENT

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, . . . , as shown in Figure 2.3. We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, . . . , and for a word length of 64 (2^3 bytes), aligned words begin at byte addresses 0, 8, 16, ...

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. But, the most common case is to use aligned addresses, which makes accessing of memory operands more efficient, as we will see in Chapter 8.

https://hemanthrajhemu.github.io

2.1.4 ACCESSING NUMBERS AND CHARACTERS

A number usually occupies one word, and can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

For programming convenience it is useful to have different ways of specifying addresses in program instructions. We will deal with this issue in Section 2.4.

2.2 MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Read* and *Write*.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

The details of the hardware implementation of these operations are treated in Chapters 5 and 6. In this chapter, we consider all operations from the viewpoint of the ISA, so we concentrate on the logical handling of instructions and operands.

2.3 INSTRUCTIONS AND INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation.

2.3.1   REGISTER TRANSFER NOTATION

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or OUTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression

$$R2 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R2.

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

In computer jargon, the words "transfer" and "move" are commonly used to mean "copy." Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

2.3.2   ASSEMBLY-LANGUAGE NOTATION

We need another type of notation to represent machine instructions and programs. For this, we use *assembly language*. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

$$\text{Load} \quad \text{R2, LOC}$$

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

$$\text{Add} \quad \text{R4, R2, R3}$$

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*, which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

2.3.3   RISC AND CISC INSTRUCTION SETS

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in "pipelined" fashion to overlap activity and reduce total execution time of a program, as we will discuss in Chapter 6. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

We will start our presentation by concentrating on RISC-style instruction sets because they are simpler and therefore easier to understand. Later we will deal with CISC-style instruction sets and explain the key differences between the two approaches.

2.3.4   INTRODUCTION TO RISC INSTRUCTION SETS

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A *load/store architecture* is used, in which
    - Memory operands are accessed only using Load and Store instructions.
    - All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time. If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

> Load   destination, source

or more specifically

> Load   processor_register, memory_location

The memory location can be specified in several ways. The term *addressing modes* is used to refer to the different ways in which this may be accomplished, as we will discuss in Section 2.4.

Let us now consider a typical arithmetic operation. The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program instructs the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. For simplicity, we will refer to the addresses of these locations as A, B, and C, respectively. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

$$C \leftarrow [A] + [B]$$

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

> Load   R2, A
> Load   R3, B
> Add   R4, R2, R3
> Store   R4, C

We say that Add is a *three-operand*, or a *three-address*, instruction of the form

> Add   destination, source1, source2

The Store instruction is of the form

> Store   source, destination

where the source is a processor register and the destination is a memory location. Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention.

Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

Add   R3, R2, R3

and the last instruction would become

Store   R3, C

2.3.5   INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

In the preceding subsection, we used the task C = A + B, implemented as $C \leftarrow [A] + [B]$, as an example. Figure 2.4 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses $i + 4$, $i + 8$, and $i + 12$. For simplicity, we assume that a desired
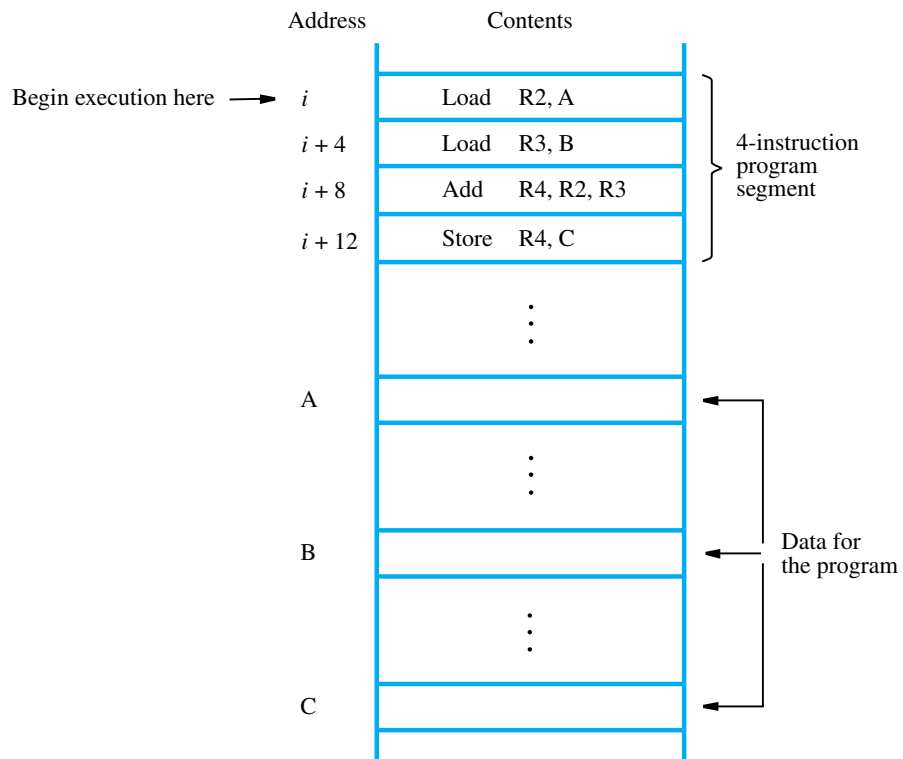
Figure 2.4   A program for $C \leftarrow [A] + [B]$.

memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved. We will resolve this issue later in Section 2.4.

Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location $i + 12$ is executed, the PC contains the value $i + 16$, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

2.3.6   BRANCHING

Consider the task of adding a list of *n* numbers. The program outlined in Figure 2.5 is a generalization of the program in Figure 2.4. The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2, . . . , NUM*n*, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Load and Add instructions, as in Figure 2.5, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure 2.6 shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch_if_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways, as will be described in Section 2.4. For now, we concentrate on how to create and control a program loop.

Assume that the number of entries in the list, *n*, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction

<div align="center">Subtract   R2, R2, #1</div>

i	Load	R2, NUM1
i + 4	Load	R3, NUM2
i + 8	Add	R2, R2, R3
i + 12	Load	R3, NUM3
i + 16	Add	R2, R2, R3
	⋮	
i + 8*n* − 12	Load	R3, NUM*n*
i + 8*n* − 8	Add	R2, R2, R3
i + 8*n* − 4	Store	R2, SUM
	⋮	
SUM		
NUM1		
NUM2		
	⋮	
NUM*n*		

Figure 2.5   A program for adding *n* numbers.

reduces the contents of R2 by 1 each time through the loop. (We will explain the significance of the number sign '#' in Section 2.4.1.) Execution of the loop is repeated as long as the contents of R2 are greater than zero.

We now introduce *branch* instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 2.6, the instruction

Branch_if_[R2]>0   LOOP
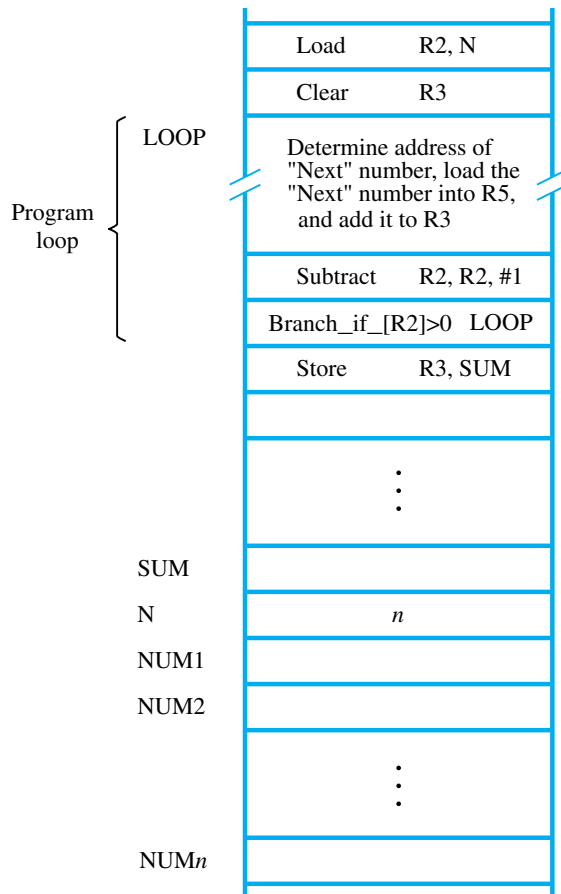
Figure 2.6 Using a loop to add *n* numbers.

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the *n*th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified

requirement. For example, the instruction that implements the action

<div align="center">Branch_if_[R4]>[R5]   LOOP</div>

may be written in generic assembly language as

<div align="center">Branch_greater_than   R4, R5, LOOP</div>

or using an actual mnemonic as

<div align="center">BGT   R4, R5, LOOP</div>

It compares the contents of registers R4 and R5, without changing the contents of either register. Then, it causes a branch to LOOP if the contents of R4 are greater than the contents of R5.

A different way of implementing branch instructions uses the concept of condition codes, which we will discuss in Section 2.10.2.

2.3.7   GENERATING MEMORY ADDRESSES

Let us return to Figure 2.6. The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, Ri, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same. We will discuss these in the next section.

2.4   ADDRESSING MODES

We have now seen some simple examples of assembly-language programs. In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways that reflect the nature of the information and how it is used. Programmers use *data structures* such as lists and arrays for organizing the data used in computations.

Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures. When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations. The

Table 2.1 RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute	LOC	EA = LOC
Register indirect	(Ri)	EA = [Ri]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]

EA = effective address
Value = a signed number
X = index value

different ways for specifying the locations of instruction operands are known as *addressing modes*. In this section we present the basic addressing modes found in RISC-style processors. A summary is provided in Table 2.1, which also includes the assembler syntax we will use for each mode. The assembler syntax defines the way in which instructions and the addressing modes of their operands are specified; it is discussed in Section 2.5.

2.4.1 IMPLEMENTATION OF VARIABLES AND CONSTANTS

Variables are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. This value can be changed as needed using appropriate instructions.

The program in Figure 2.5 uses only two addressing modes to access variables. We access an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

Register mode—The operand is the contents of a processor register; the name of the register is given in the instruction.

Absolute mode—The operand is in a memory location; the address of this location is given explicitly in the instruction.

Since in a RISC-style processor an instruction must fit in a single word, the number of bits that can be used to give an absolute address is limited, typically to 16 bits if the word length is 32 bits. To generate a 32-bit address, the 16-bit value is usually extended to 32 bits by replicating bit b_{15} into bit positions b_{31-16} (as in sign extension). This means that an absolute address can be specified in this manner for only a limited range of the full address space. We will deal with the issue of specifying full 32-bit addresses in Section 2.9. To keep our examples simple, we will assume for now that all addresses of memory locations involved in a program can be specified in 16 bits.

The instruction

<div align="center">Add   R4, R2, R3</div>

uses the Register mode for all three operands. Registers R2 and R3 hold the two source operands, while R4 is the destination.

The Absolute mode can represent global variables in a program. A declaration such as

<div align="center">Integer   NUM1, NUM2, SUM;</div>

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

The Absolute mode is used in the instruction

<div align="center">Load   R2, NUM1</div>

which loads the value in the memory location NUM1 into register R2.

Constants representing data or addresses are also found in almost every computer program. Such constants can be represented in assembly language using the Immediate addressing mode.

   Immediate mode—The operand is given explicitly in the instruction.

For example, the instruction

<div align="center">Add   R4, R6, 200$_{immediate}$</div>

adds the value 200 to the contents of register R6, and places the result into register R4. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the number sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

<div align="center">Add   R4, R6, #200</div>

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed. The effective address is then used to access the operand.

2.4.2 INDIRECTION AND POINTERS

The program in Figure 2.6 requires a capability for modifying the address of the memory operand during each pass through the loop. A good way to provide this capability is to use a processor register to hold the address of the operand. The contents of the register are then changed (incremented) during each pass to provide the address of the next number in the list that has to be accessed. The register acts as a *pointer* to the list, and we say that an item
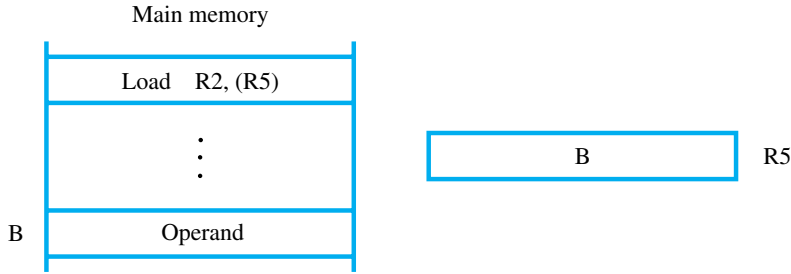
Figure 2.7 Register indirect addressing.

in the list is accessed *indirectly* by using the address in the register. The desired capability is provided by the indirect addressing mode.

   Indirect mode—The effective address of the operand is the contents of a register that is specified in the instruction.

We denote indirection by placing the name of the register given in the instruction in parentheses as illustrated in Figure 2.7 and Table 2.1.

   To execute the Load instruction in Figure 2.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

   Indirection and the use of pointers are important and powerful concepts in programming. They permit the same code to be used to operate on different data. For example, register R5 in Figure 2.7 serves as a pointer for the Load instruction to load an operand from the memory into register R2. At one time, R5 may point to location B in memory. Later, the program may change the contents of R5 to point to a different location, in which case the same Load instruction will load the value from that location into R2. Thus, a program segment that includes this Load instruction is conveniently reused with only a change in the pointer value.

   Let us now return to the program in Figure 2.6 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 2.8. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4. The initialization section of the program loads the counter value *n* from memory location N into R2. Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4. Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move   R4, #NUM1

	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

Figure 2.8   Use of indirect addressing in the program of Figure 2.6.

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as

$$\text{Add} \quad \text{R4, R0, \#NUM1}$$

It is often the case that Move is provided as a *pseudoinstruction* for the convenience of programmers, but it is actually implemented using the Add instruction.

The first three instructions in the loop in Figure 2.8 implement the unspecified instruction block starting at LOOP in Figure 2.6. The first time through the loop, the instruction

$$\text{Load} \quad \text{R5, (R4)}$$

fetches the operand at location NUM1 and loads it into R5. The first Add instruction adds this number to the sum in register R3. The second Add instruction adds 4 to the contents of the pointer R4, so that it will contain the address value NUM2 when the Load instruction is executed in the second pass through the loop.

As another example of pointers, consider the C-language statement

$$A = *B;$$

where B is a pointer variable and the '*' symbol is the operator for indirect accesses. This statement causes the contents of the memory location pointed to by B to be loaded into memory location A. The statement may be compiled into

Load	R2, B
Load	R3, (R2)
Store	R3, A

Indirect addressing through registers is used extensively. The program in Figure 2.8 shows the flexibility it provides.

2.4.3   INDEXING AND ARRAYS

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode—The effective address of the operand is generated by adding a constant value to the contents of a register.

For convenience, we will refer to the register used in this mode as the *index register*. Typically, this is just a general-purpose register. We indicate the Index mode symbolically as

$$X(Ri)$$

where X denotes a constant signed integer value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

The contents of the register are not changed in the process of generating the effective address.

In an assembly-language program, whenever a constant such as the value X is needed, it may be given either as an explicit number or as a symbolic name representing a numerical value. The way in which a symbolic name is associated with a specific numerical value will be discussed in Section 2.5. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is restricted to fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended (see Section 1.4) to the register length before being added to the contents of the register.

Figure 2.9 illustrates two ways of using the Index mode. In Figure 2.9*a*, the index register, R5, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use is illustrated in Figure 2.9*b*. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is held in a register.

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.10. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are *n* students in the class, and the value *n* is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

We should note that the list in Figure 2.10 represents a two-dimensional array having *n* rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible
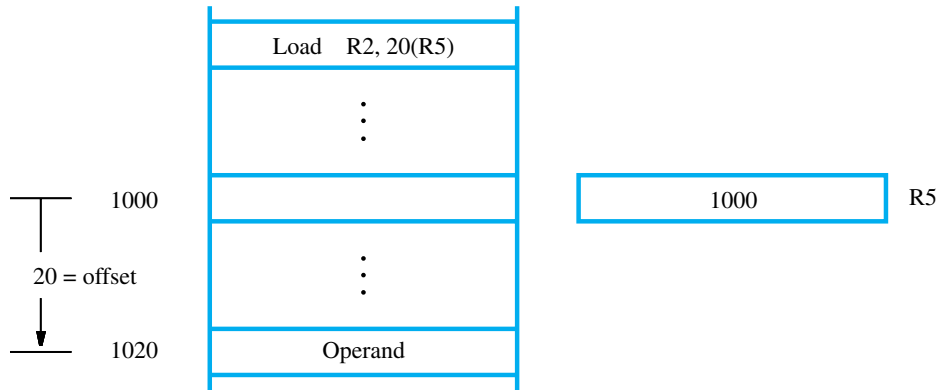
(a) Offset is given as a constant
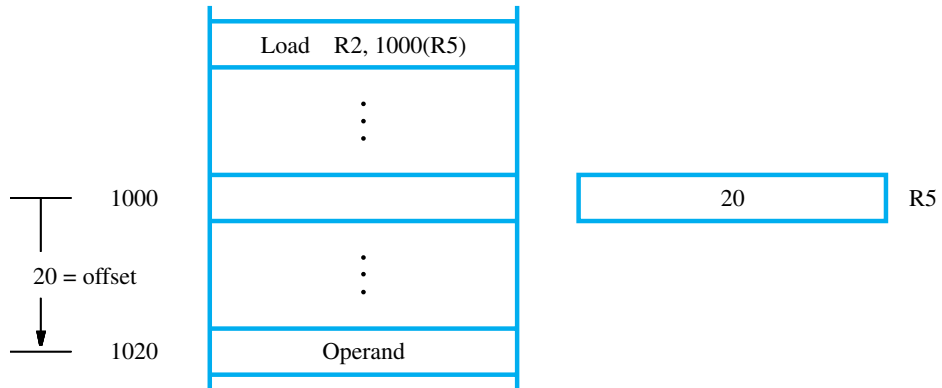
(b) Offset is in the index register

Figure 2.9   Indexed addressing.

program for this task is given in Figure 2.11. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.9*a* to access each of the three scores in a student's record. Register R2 is used as the index register. Before the loop is entered, R2 is set to point to the ID location of the first student record which is the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R3, R4, and R5, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R2), 8(R2), and 12(R2). The index register R2 is then incremented by 16 to point to the ID location of the second student. Register R6, initialized to contain the value *n*, is decremented by 1 at the end of each pass through the loop. When the contents of R6 reach 0, all student records have been accessed, and
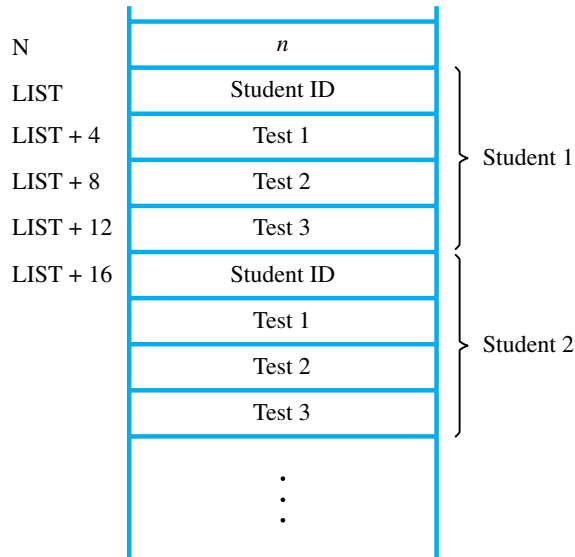
Figure 2.10 A list of students' marks.

Move	R2, #LIST	Get the address LIST.
Clear	R3	
Clear	R4	
Clear	R5	
Load	R6, N	Load the value *n*.
LOOP: Load	R7, 4(R2)	Add the mark for next student's
Add	R3, R3, R7	Test 1 to the partial sum.
Load	R7, 8(R2)	Add the mark for that student's
Add	R4, R4, R7	Test 2 to the partial sum.
Load	R7, 12(R2)	Add the mark for that student's
Add	R5, R5, R7	Test 3 to the partial sum.
Add	R2, R2, #16	Increment the pointer.
Subtract	R6, R6, #1	Decrement the counter.
Branch_if_[R6]>0	LOOP	Branch back if not finished.
Store	R3, SUM1	Store the total for Test 1.
Store	R4, SUM2	Store the total for Test 2.
Store	R5, SUM3	Store the total for Test 3.

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R2, are not changed when it is used in the Index addressing mode to access the scores. The contents of R2 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing that uses a register Ri and a constant offset X. Several variations of this basic form provide for efficient access to memory operands in practical programming situations (although they may not be included in some processors). For example, a second register Rj may be used to contain the offset X, in which case we can write the Index mode as

$$(Ri, Rj)$$

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(Ri, Rj)$$

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode.

Finally, we should note that in the basic Index mode

$$X(Ri)$$

if the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as

$$X(R0)$$

2.5   ASSEMBLY LANGUAGE

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Load, Store, Add, and

Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics*, such as LD, ST, ADD, and BR. A shorthand notation is also useful when identifying registers, such as R3 for register 3. Finally, symbols such as LOC may be defined as needed to represent particular memory locations. A complete set of such symbolic names and rules for their use constitutes a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*. The assembler program is one of a collection of utility programs that are a part of the system software of a computer. The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk. At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine-language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*. We will discuss how the assembler program works in Section 2.5.2 and in Chapter 4. First, we present a few aspects of assembly language itself.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower-case letters. In this section, we use capital letters to denote all names and labels in our examples to improve the readability of the text. For example, we write a Store instruction as

$$\text{ST} \quad \text{R2, SUM}$$

The mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer recognizes.

The OP-code mnemonic is followed by at least one blank space or tab character. Then the information that specifies the operands is given. In the Store instruction above, the source operand is in register R2. This information is followed by the specification of the destination operand, separated from the source operand by a comma. The destination operand is in the memory location that has its binary address represented by the name SUM.

Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used. For example, a numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode. The number sign usually denotes an immediate operand. Thus, the instruction

$$\text{ADD} \quad \text{R2, R3, \#5}$$

adds the number 5 to the contents of register R3 and puts the result into register R2. The number sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the Immediate addressing mode is indicated in the OP-code mnemonic.

For example, the previous Add instruction may be written as

<div align="center">ADDI   R2, R3, 5</div>

The suffix I in the mnemonic ADDI states that the second source operand is given in the Immediate addressing mode.

Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand. For example, if register R2 contains the address of a number in the memory, then this number can be loaded into register R3 using the instruction

<div align="center">LD   R3, (R2)</div>

2.5.1   ASSEMBLER DIRECTIVES

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as

<div align="center">TWENTY   EQU   20</div>

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program. Such statements, called *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program.

To illustrate the use of assembly language further, let us reconsider the program in Figure 2.8. In order to run this program on a computer, it is necessary to write its source code in the required assembly language, specifying all of the information needed to generate the corresponding object program. Suppose that each instruction and each data item occupies one word of memory. Also assume that the memory is byte-addressable and that the word length is 32 bits. Suppose also that the object program is to be loaded in the main memory as shown in Figure 2.12. The figure shows the memory addresses where the machine instructions and the required data items are to be found after the program is loaded for execution. If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

To provide this information, the source program may be written as shown in Figure 2.13. The program begins with the assembler directive, ORIGIN, which tells the assembler program where in the memory to place the instructions that follow. It specifies that the instructions

	100	Load	R2, N
	104	Clear	R3
	108	Move	R4, #NUM1
LOOP	112	Load	R5, (R4)
	116	Add	R3, R3, R5
	120	Add	R4, R4, #4
	124	Subtract	R2, R2, #1
	128	Branch_if_[R2]>0 LOOP	
	132	Store	R3, SUM
		$\vdots$	
SUM	200		
N	204	150	
NUM1	208		
NUM2	212		
		$\vdots$	
NUMn	804		

Figure 2.12   Memory arrangement for the program in Figure 2.8.

of the object program are to be loaded in the memory starting at address 100. It is followed by the source program instructions written with the appropriate mnemonics and syntax. Note that we use the statement

BGT   R2, R0, LOOP

to represent an instruction that performs the operation

Branch_if_[R2]>0   LOOP

The second ORIGIN directive tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 200. This is intended to be the location in which the final sum will be stored. A 4-byte space for the sum is reserved by means of the assembler directive RESERVE. The next word, at address 204, has to contain the value 150 which is the number of entries in the list.

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Figure 2.13   Assembly language representation for the program in Figure 2.12.

The DATAWORD directive is used to inform the assembler of this requirement. The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data. This directive does not cause any data to be loaded in these locations. Data may be loaded in the memory using an input procedure, as we will explain in Chapter 3. The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

We previously described how the EQU directive can be used to associate a specific value, which may be an address, with a particular name. A different way of associating addresses with names or labels is illustrated in Figure 2.13. Any statement that results in instructions or data being placed in a memory location may be given a memory address label. The assembler automatically assigns the address of that location to the label. For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1. Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200. Whenever SUM is encountered in the program, it will be replaced with this value. Using SUM as a label in

this manner is equivalent to using the assembler directive

$$\text{SUM} \quad \text{EQU} \quad 200$$

Similarly, the labels N and NUM1 are assigned the values 204 and 208, respectively, because they represent the addresses of the two word locations immediately following the word location with address 200.

Most assembly languages require statements in a source program to be written in the form

$$\text{Label:} \quad \text{Operation} \quad \text{Operand(s)} \quad \text{Comment}$$

These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters. The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded. Labels may also be associated with addresses of data items. In Figure 2.13 there are four labels: LOOP, SUM, N, and NUM1.

The Operation field contains an assembler directive or the OP-code mnemonic of the desired instruction. The Operand field contains addressing information for accessing the operands. The Comment field is ignored by the assembler program. It is used for documentation purposes to make the program easier to understand.

We have introduced only the very basic characteristics of assembly languages. These languages differ in detail and complexity from one computer to another.

2.5.2   ASSEMBLY AND EXECUTION OF PROGRAMS

A source program written in an assembly language must be assembled into a machine-language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the addresses given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands, and it reserves memory space as requested by RESERVE commands.

A key part of the assembly process is determining the values that replace the names. In some cases, where the value of a name is specified by an EQU directive, this is a straightforward task. In other cases, where a name is defined in the Label field of a given instruction, the value represented by the name is determined by the location of this instruction in the assembled object program. Hence, the assembler must keep track of addresses as it generates the machine code for successive instructions. For example, the names LOOP and SUM in the program of Figure 2.13 will be assigned the values 112 and 200, respectively.

In some cases, the assembler does not directly replace a name representing an address with the actual value of this address. For example, in a branch instruction, the name that specifies the location to which a branch is to be made (the branch target) is not replaced by the actual address. A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter

to the target instruction. The assembler computes this *branch offset*, which can be positive or negative, and puts it into the machine instruction. We will show how branch instructions may be implemented in Section 2.13.

The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a *loader* must already be in the memory. Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored. The assembler usually places this information in a header preceding the object code. Having loaded the object code, the loader starts execution of the object program by branching to the first instruction to be executed, which may be identified by an address label such as START. The assembler places that address in the header of the object code for the loader to use at execution time.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger* program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

In this section, we introduced some important issues in assembly and execution of programs. Chapter 4 provides a more detailed discussion of these issues.

2.5.3   NUMBER NOTATION

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

$$\text{ADDI} \quad \text{R2, R3, 93}$$

or as a binary number identified by an assembler-specific prefix symbol such as a percent sign, as in

$$\text{ADDI} \quad \text{R2, R3, \%01011101}$$

Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The first ten patterns 0000, 0001, . . . , 1001, referred to as *binary-coded decimal* (BCD), are represented by the digits 0, 1, . . . , 9. The remaining six 4-bit patterns, 1010, 1011, . . . , 1111, are represented by the letters A, B, . . . , F. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by the prefix 0x (as in the C language) or

by a dollar sign prefix. Thus, we would write

$$\text{ADDI} \quad \text{R2, R3, 0x5D}$$

2.6   STACKS

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, *last-in–first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the *stack pointer* (SP), is used to point to a particular stack structure called the *processor stack*, whose use will be explained shortly.

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 2.14 shows an example of a stack of word data items. The stack contains numerical values, with 43 at the bottom and -28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

$$\text{Subtract} \quad \text{SP, SP, \#4}$$
$$\text{Store} \quad \text{R}j, \text{(SP)}$$

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register Rj, the Store instruction will place this value on the stack. These two instructions copy the word from Rj onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as

$$\text{Load} \quad \text{R}j, \text{(SP)}$$
$$\text{Add} \quad \text{SP, SP, \#4}$$

These two instructions load (pop) the top value from the stack into register Rj and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.15 shows the effect of each of these operations on the stack in Figure 2.14.
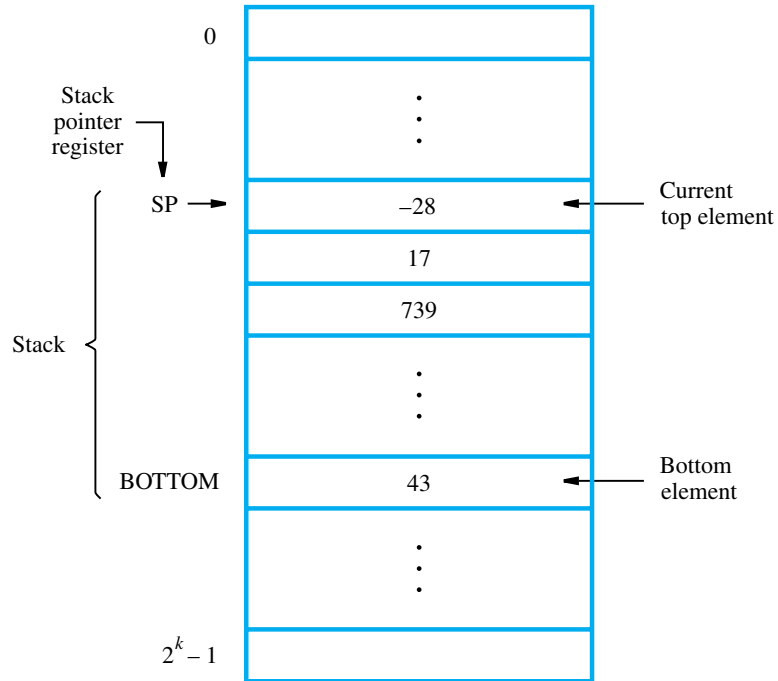
Figure 2.14   A stack of words in the memory.

2.7 SUBROUTINES

In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*. For example, a subroutine may evaluate a mathematical function, or it may sort a list of values into increasing or decreasing order.

It is possible to reproduce the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it, and it does so by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling
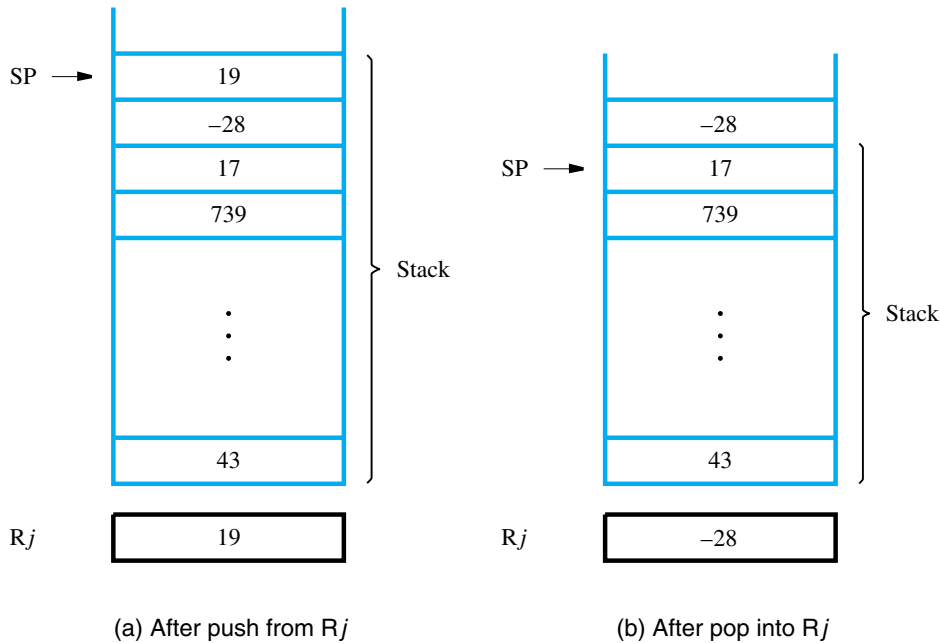
Figure 2.15   Effect of stack operations on the stack in Figure 2.14.

program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.
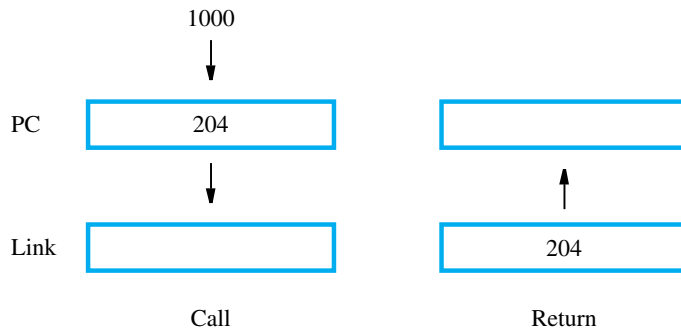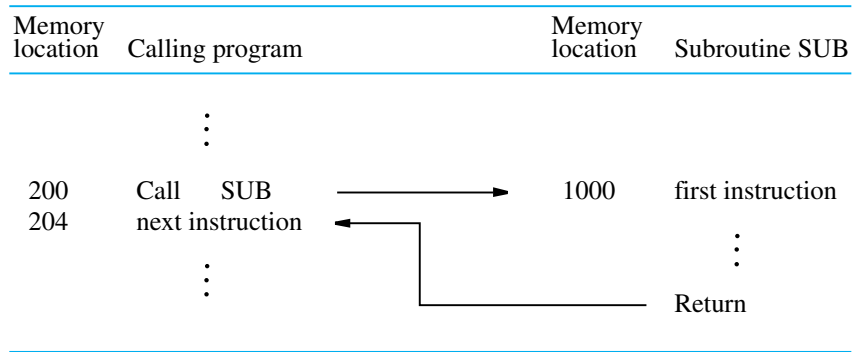
The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

Figure 2.16 illustrates how the PC and the link register are affected by the Call and Return instructions.

Memory location	Calling program		Memory location	Subroutine SUB
	⋮			
200	Call SUB	⟶	1000	first instruction
204	next instruction	⟵		⋮
	⋮			Return

Figure 2.16   Subroutine linkage using a link register.

2.7.1   SUBROUTINE NESTING AND THE PROCESSOR STACK

A common programming practice, called *subroutine nesting*, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register, overwriting its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it. The return address needed for this first return is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Correct sequencing of nested calls is achieved if a given subroutine SUB1 saves the return address currently in the link register on the stack, accessed through the stack pointer, SP, before it calls another subroutine SUB2. Then, prior to executing its own Return instruction, the subroutine SUB1 has to pop the saved return address from the stack and load it into the link register.

2.7.2 PARAMETER PASSING

When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as *parameter passing*. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack.

Passing parameters through processor registers is straightforward and efficient. Figure 2.17 shows how the program in Figure 2.8 for adding a list of numbers can be implemented as a subroutine, LISTADD, with the parameters passed through registers. The size of the list, n, contained in memory location N, and the address, NUM1, of the first number, are passed through registers R2 and R4. The sum computed by the subroutine is passed back to the calling program through register R3. The first four instructions in Figure 2.17 constitute the relevant part of the calling program. The first two instructions load n and NUM1 into

---

Calling program

Load	R2, N	Parameter 1 is list size.
Move	R4, #NUM1	Parameter 2 is list location.
Call	LISTADD	Call subroutine.
Store	R3, SUM	Save result.
⋮		

Subroutine

LISTADD:	Subtract	SP, SP, #4	Save the contents of
	Store	R5, (SP)	R5 on the stack.
	Clear	R3	Initialize sum to 0.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer by 4.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	
	Load	R5, (SP)	Restore the contents of R5.
	Add	SP, SP, #4	
	Return		Return to calling program.

Figure 2.17 Program of Figure 2.8 written as a subroutine; parameters passed through registers.

R2 and R4. The Call instruction branches to the subroutine starting at location LISTADD. This instruction also saves the return address (i.e., the address of the Store instruction in the calling program) in the link register. The subroutine computes the sum and places it in R3. After the Return instruction is executed by the subroutine, the sum in R3 is stored in memory location SUM by the calling program.

In addition to registers R2, R3, and R4, which are used for parameter passing, the subroutine also uses R5. Since R5 may be used in the calling program, its contents are saved by pushing them onto the processor stack upon entry to the subroutine and restored before returning to the calling program.

If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine. The processor stack provides a convenient and flexible mechanism for passing an arbitrary number of parameters. Figure 2.18 shows the program of Figure 2.8 rewritten as a subroutine, LISTADD, which uses the processor stack for parameter passing. The address of the first number in the list and the number of entries are pushed onto the processor stack pointed to by register SP. The subroutine is then called. The computed sum is placed on the stack before the return to the calling program.

Figure 2.19 shows the stack entries for this example. Assume that before the subroutine is called, the top of the stack is at level 1. The calling program pushes the address NUM1 and the value n onto the stack and calls subroutine LISTADD. The top of the stack is now at level 2. The subroutine uses four registers while it is being executed. Since these registers may contain valid data that belong to the calling program, their contents should be saved at the beginning of the subroutine by pushing onto the stack. The top of the stack is now at level 3. The subroutine accesses the parameters n and NUM1 from the stack using indexed addressing with offset values relative to the new top of the stack (level 3). Note that it does not change the stack pointer because valid data items are still at the top of the stack. The value n is loaded into R2 as the initial value of the count, and the address NUM1 is loaded into R4, which is used as a pointer to scan the list entries.

At the end of the computation, register R3 contains the sum. Before the subroutine returns to the calling program, the contents of R3 are inserted into the stack, replacing the parameter NUM1, which is no longer needed. Then the contents of the four registers used by the subroutine are restored from the stack. Also, the stack pointer is incremented to point to the top of the stack that existed when the subroutine was called, namely the parameter n at level 2. After the subroutine returns, the calling program stores the result in location SUM and lowers the top of the stack to its original level by incrementing the SP by 8.

Observe that for subroutine LISTADD in Figure 2.18, we did not use a pair of instructions

|	
Subtract	SP, SP, #4
Store	Rj, (SP)

to push the contents of each register on the stack. Since we have to save four registers, this would require eight instructions. We needed only five instructions by adjusting SP immediately to point to the top of stack that will be in effect once all four registers are saved. Then, we used the Index mode to store the contents of registers. We used the same optimization when restoring the registers before returning from the subroutine.

Assume top of stack is at level 1 in Figure 2.19.

	Move	R2, #NUM1	Push parameters onto stack.
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Load	R2, N	
	Subtract	SP, SP, #4	
	Store	R2, (SP)	
	Call	LISTADD	Call subroutine
			(top of stack is at level 2).
	Load	R2, 4(SP)	Get the result from the stack
	Store	R2, SUM	and save it in SUM.
	Add	SP, SP, #8	Restore top of stack
			(top of stack is at level 1).

$\vdots$

LISTADD:	Subtract	SP, SP, #16	Save registers
	Store	R2, 12(SP)	
	Store	R3, 8(SP)	
	Store	R4, 4(SP)	
	Store	R5, (SP)	(top of stack is at level 3).
	Load	R2, 16(SP)	Initialize counter to n.
	Load	R4, 20(SP)	Initialize pointer to the list.
	Clear	R3	Initialize sum to 0.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer by 4.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]>0	LOOP	
	Store	R3, 20(SP)	Put result in the stack.
	Load	R5, (SP)	Restore registers.
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Add	SP, SP, #16	(top of stack is at level 2).
	Return		Return to calling program.

Figure 2.18 Program of Figure 2.8 written as a subroutine; parameters passed on the stack.
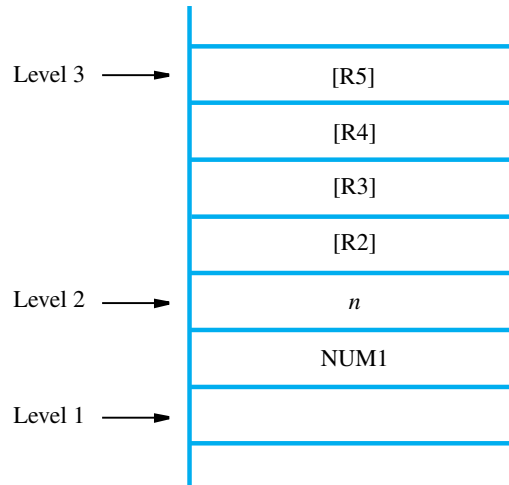
Figure 2.19   Stack contents for the program in Figure 2.18.

We should also note that some computers have special instructions for loading and storing multiple registers. For example, the four registers in Figure 2.18 may be saved on the stack by using the instruction

$$\text{StoreMultiple} \quad \text{R2}-\text{R5}, \, -(\text{SP})$$

The source registers are specified by the range R2−R5. The notation −(SP) specifies that the stack pointer must be adjusted accordingly. The minus sign in front indicates that SP must be decremented (by 4) before the contents of each register are placed on the stack.

Similarly, the instruction

$$\text{LoadMultiple} \quad \text{R2}-\text{R5}, \, (\text{SP})+$$

will load registers R2, R3, R4, and R5, in reverse order, with the values that were saved on the stack. The notation (SP)+ indicates that the stack pointer must be incremented (by 4) after each value has been loaded into the corresponding register. We will discuss the addressing modes denoted by −(SP) and (SP)+ in more detail in Section 2.9.1.

Parameter Passing by Value and by Reference

Note the nature of the two parameters, NUM1 and n, passed to the subroutines in Figures 2.17 and 2.18. The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called *passing by reference*. The second parameter is *passed by value*, that is, the actual number of entries, n, is passed to the subroutine.

2.7.3   THE STACK FRAME

Now, observe how space is used in the stack in the example in Figures 2.18 and 2.19. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program. Such space is called a *stack frame*. If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack.

Figure 2.20 shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the *frame pointer* (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the subroutine. When nested subroutines are used, the stack frame of the calling subroutine would also include the return address, as we will see in the example that follows.
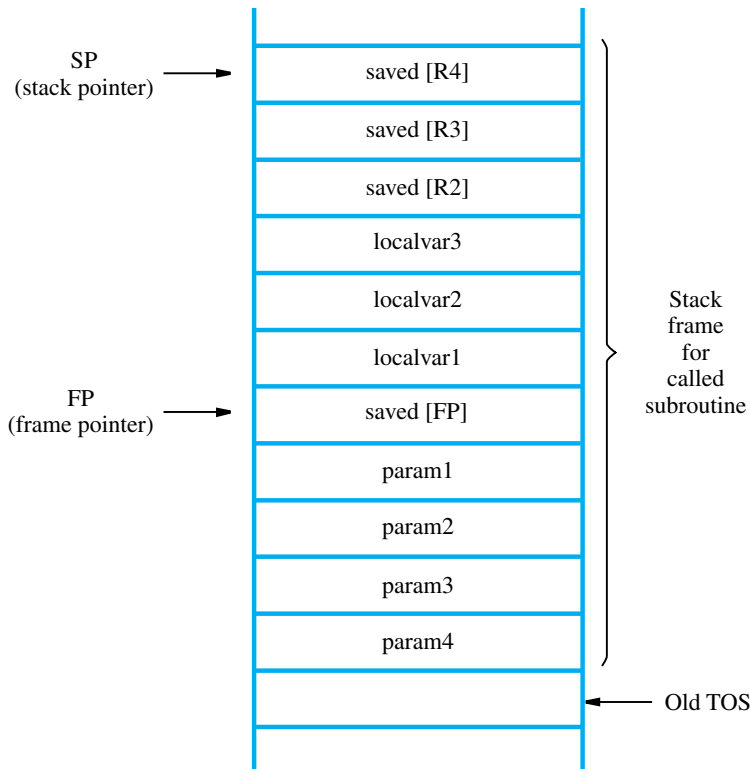
Figure 2.20   A subroutine stack frame example.

With the FP register pointing to the location just above the stored parameters, as shown in Figure 2.20, we can easily access the parameters and the local variables by using the Index addressing mode. The parameters can be accessed by using addresses 4(FP), 8(FP), ... The local variables can be accessed by using addresses −4(FP), −8(FP), ... The contents of FP remain fixed throughout the execution of the subroutine, unlike the stack pointer SP, which must always point to the current top element in the stack.

Now let us discuss how the pointers SP and FP are manipulated as the stack frame is allocated, used, and deallocated for a particular invocation of a subroutine. We begin by assuming that SP points to the old top-of-stack (TOS) element in Figure 2.20. Before the subroutine is called, the calling program pushes the four parameters onto the stack. Then the Call instruction is executed. At this time, SP points to the last parameter that was pushed on the stack. If the subroutine is to use the frame pointer, it should first save the contents of FP by pushing them on the stack, because FP is usually a general-purpose register and it may contain information of use to the calling program. Then, the contents of SP, which now points to the saved value of FP, are copied into FP.

Thus, the first three instructions executed in the subroutine are

Subtract	SP, SP, #4
Store	FP, (SP)
Move	FP, SP

The Move instruction copies the contents of SP into FP. After these instructions are executed, both SP and FP point to the saved FP contents. Space for the three local variables is now allocated on the stack by executing the instruction

Subtract   SP, SP, #12

Finally, the contents of processor registers R2, R3, and R4 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in Figure 2.20.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R4, R3, and R2 back into those registers, deallocates the local variables from the stack frame by executing the instruction

Add   SP, SP, #12

and pops the saved old value of FP back into FP. At this point, SP points to the last parameter that was placed on the stack. Next, the Return instruction is executed, transferring control back to the calling program.

The calling program is responsible for deallocating the parameters from the stack frame, some of which may be results passed back by the subroutine. After deallocation of the parameters, the stack pointer points to the old TOS, and we are back to where we started.

Stack Frames for Nested Subroutines

When nested subroutines are used, it is necessary to ensure that the return addresses are properly saved. When a calling program calls a subroutine, say SUB1, the return address is saved in the link register. Now, if SUB1 calls another subroutine, SUB2, it must save the

current contents of the link register before it makes the call to SUB2. The appropriate place for saving this return address is within the stack frame for SUB1. If SUB2 then calls SUB3, it must save the current contents of the link register within the stack frame associated with SUB2, and so on.

An example of a main program calling a first subroutine SUB1, which then calls a second subroutine SUB2, is shown in Figure 2.21. The stack frames corresponding to these two nested subroutines are shown in Figure 2.22. All parameters involved in this example are passed on the stack. The two figures only show the flow of control and data among the main program and the two subroutines. The actual computations are not shown.

The flow of execution is as follows. The main program pushes the two parameters *param2* and *param1* onto the stack, in that order, and then calls SUB1. This first subroutine is responsible for computing a single result and passing it back to the main program on the stack. During the course of its computations, SUB1 calls the second subroutine, SUB2, in order to perform some other subtask. SUB1 passes a single parameter *param3* to SUB2, and the result is passed back to it via the same location on the stack. After SUB2 executes its Return instruction, SUB1 loads this result into register R4. SUB1 then continues its computations and eventually passes the required answer back to the main program on the stack. When SUB1 executes its return to the main program, the main program stores this answer in memory location RESULT, restores the stack level, then continues with its computations at the next instruction at address 2040. Note how the return address to the calling program, 2028, is stored within the stack frame for SUB1 in Figure 2.22.

The comments in Figure 2.21 provide the details of how this flow of execution is managed. The first action performed by each subroutine is to save on the stack the contents of all registers used in the subroutine, including the frame pointer and link register (if needed). This is followed by initializing the frame pointer. SUB1 uses four registers, R2 to R5, and SUB2 uses two registers, R2 and R3. These registers, the frame pointer, and the link register in the case of SUB1, are restored just before the Return instructions are executed.

The Index addressing mode involving the frame pointer register FP is used to load parameters from the stack and place answers back on the stack. The byte offsets used in these operations are always $4, 8, \ldots$, as discussed for the general stack frame in Figure 2.20. Finally, note that each calling routine is responsible for removing its own parameters from the stack. This is done by the Add instructions, which lower the top of the stack.

2.8   ADDITIONAL INSTRUCTIONS

So far, we have introduced the following instructions: Load, Store, Move, Clear, Add, Subtract, Branch, Call, and Return. These instructions, along with the addressing modes in Table 2.1, have allowed us to write programs to illustrate machine instruction sequencing, including branching and subroutine linkage. In this section we introduce a few more instructions that are found in most instruction sets.

Memory location	Instructions		Comments
Main program			
	⋮		
2000	Load	R2, PARAM2	Place parameters on stack.
2004	Subtract	SP, SP, #4	
2008	Store	R2, (SP)	
2012	Load	R2, PARAM1	
2016	Subtract	SP, SP, #4	
2020	Store	R2, (SP)	
2024	Call	SUB1	Call the subroutine.
2028	Load	R2, (SP)	Store result.
2032	Store	R2, RESULT	
2036	Add	SP, SP, #8	Restore stack level.
2040	next instruction		
	⋮		
First subroutine			
2100 SUB1:	Subtract	SP, SP, #24	Save registers.
2104	Store	LINK_reg, 20(SP)	
2108	Store	FP, 16(SP)	
2112	Store	R2, 12(SP)	
2116	Store	R3, 8(SP)	
2120	Store	R4, 4(SP)	
2124	Store	R5, (SP)	
2128	Add	FP, SP, #16	Initialize the frame pointer.
2132	Load	R2, 8(FP)	Get first parameter.
2136	Load	R3, 12(FP)	Get second parameter.
	⋮		
	Load	R4, PARAM3	Place a parameter on stack.
	Subtract	SP, SP, #4	
	Store	R4, (SP)	
	Call	SUB2	
	Load	R4, (SP)	Get result from SUB2.
	Add	SP, SP, #4	
	⋮		
	Store	R5, 8(FP)	Place answer on stack.
	Load	R5, (SP)	Restore registers.
	Load	R4, 4(SP)	
	Load	R3, 8(SP)	
	Load	R2, 12(SP)	
	Load	FP, 16(SP)	
	Load	LINK_reg, 20(SP)	
	Add	SP, SP, #24	
	Return		Return to Main program.
...continued in part *b*.			

Figure 2.21   Nested subroutines (part *a*).

Memory location		Instructions		Comments

Second subroutine

3000	SUB2:	Subtract	SP, SP, #12	Save registers.
3004		Store	FP, 8(SP)	
		Store	R2, 4(SP)	
		Store	R3, (SP)	
		Add	FP, SP, #8	Initialize the frame pointer.
		Load	R2, 4(FP)	Get the parameter.
		⋮		
		Store	R3, 4(FP)	Place SUB2 result on stack.
		Load	R3, (SP)	Restore registers.
		Load	R2, 4(SP)	
		Load	FP, 8(SP)	
		Add	SP, SP, #12	
		Return		Return to Subroutine 1.

Figure 2.21   Nested subroutines (part *b*).

2.8.1 LOGIC INSTRUCTIONS

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described in Appendix A. It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

$$\text{And} \quad \text{R4, R2, R3}$$

computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4. An immediate form of this instruction may be

$$\text{And} \quad \text{R4, R2, \#Value}$$

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

Consider the following application for this logic instruction. Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. If it is, then a conditional branch to FOUNDZ is to be made. From Table 1.1 in Chapter 1, we find that the ASCII code for Z is 01011010, which is expressed in hexadecimal notation as 5A. The three-instruction sequence

And	R2, R2, #0xFF
Move	R3, #0x5A
Branch_if_[R2]=[R3]	FOUNDZ
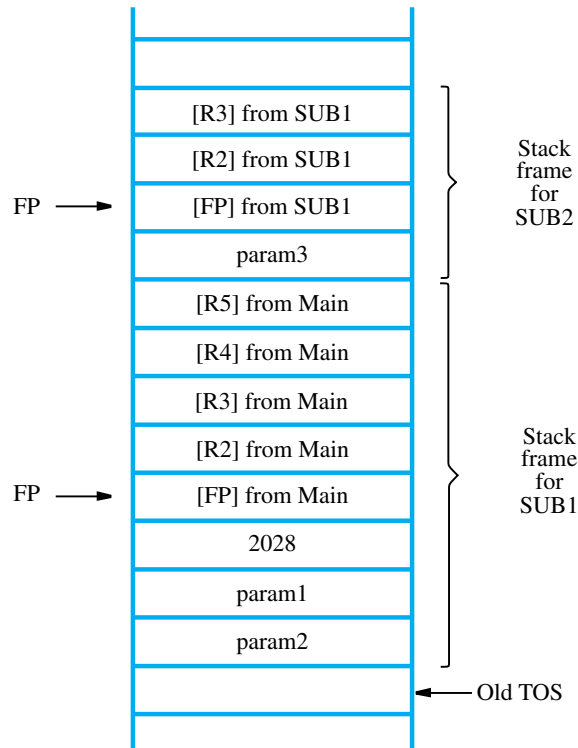
Figure 2.22   Stack frames for Figure 2.21.

implements the desired action. The And instruction clears all bits in the leftmost three character positions of R2 to zero, leaving the rightmost character unchanged. This is the result of using an immediate operand that has eight 1s at its right end, and 0s in the 24 bits to the left. The Move instruction loads the hex value 5A into R3. Since both R2 and R3 have 0s in the leftmost 24 bits, the Branch instruction compares the remaining character at the right end of R2 with the binary representation for the character Z, and causes a branch to FOUNDZ if there is a match.

2.8.2   SHIFT AND ROTATE INSTRUCTIONS

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a signed number, we use an arithmetic shift, which preserves the sign of the number.

Logical Shifts

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions

specified in a count operand contained in the instruction. The general form of a Logical-shift-left instruction is

$$\text{LShiftL} \quad \text{R}i, \text{R}j, \text{count}$$

which shifts the contents of register Rj left by a number of bit positions given by the count operand, and places the result in register Ri, without changing the contents of Rj. The count operand may be given as an immediate operand, or it may be contained in a processor register. To complete the description of the shift left operation, we need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. Vacated positions are filled with zeros. In computers that do not use condition code flags, the bits shifted out are simply dropped. In computers that use condition code flags, which will be discussed in Section 2.10.2, these bits are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word. Figure 2.23a shows an example of shifting the contents of register R3 left by two bit positions. The Logical-shift-right instruction, LShiftR, works in the same manner except that it shifts to the right. Figure 2.23b illustrates this operation.

Digit-Packing Example

Consider the following short task that illustrates the use of both shift operations and logic operations. Suppose that two decimal digits represented in ASCII code are located in the memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in *packed-BCD* format. Table 1.1 in Chapter 1 shows that the rightmost four bits of the ASCII code for a decimal digit correspond to the BCD code for the digit. Hence, the required task is to extract the low-order four bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED.

The instruction sequence shown in Figure 2.24 accomplishes the task using register R2 as a pointer to the ASCII characters in memory, and using registers R3 and R4 to develop the BCD digit codes. The program uses the LoadByte instruction, which loads a byte from the memory into the rightmost eight bit positions of a 32-bit processor register and clears the remaining higher-order bits to zero. The StoreByte instruction writes the rightmost byte in the source register into the specified destination location, but does not affect any other byte locations. The value 0xF in the And instruction is used to clear to zero all but the four rightmost bits in R4. Note that the immediate source operand is written as 0xF, which, interpreted as a 32-bit pattern, has 28 zeros in the most-significant bit positions.

Arithmetic Shifts

In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number. A study of the 2's-complement binary number representation in Figure 1.3 reveals that shifting a number one bit position to the left is equivalent to multiplying it by 2, and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost when shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers. This requirement when shifting right distinguishes arithmetic shifts from logical shifts in which the fill-in

(a) Logical shift left   LShiftL   R3, R3, #2

(b) Logical shift right   LShiftR   R3, R3, #2

(c) Arithmetic shift right   AShiftR   R3, R3, #2
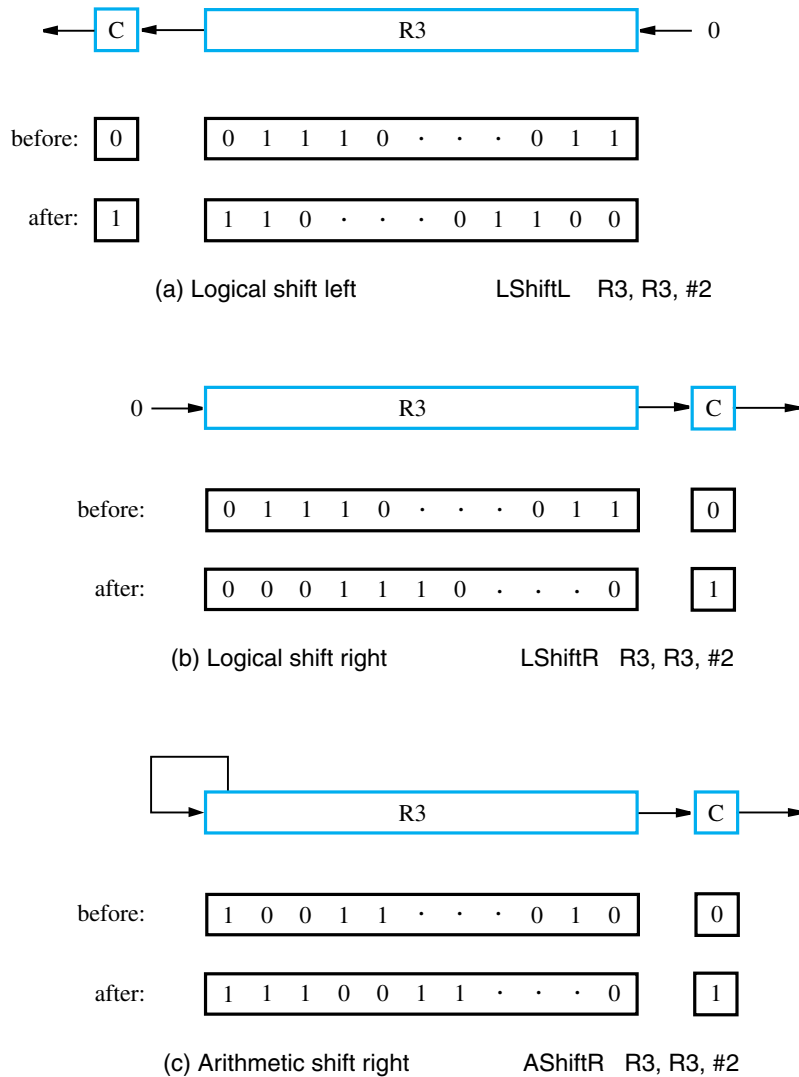
Figure 2.23   Logical and arithmetic shift instructions.

bit is always 0. Otherwise, the two types of shifts are the same. An example of an Arithmetic-shift-right instruction, AShiftR, is shown in Figure 2.23*c*. The Arithmetic-shift-left is exactly the same as the Logical-shift-left.

Rotate Operations

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. For situations where it is desirable to preserve all of the bits, rotate instructions may be used instead. These are instructions that

Move	R2, #LOC	R2 points to data.
LoadByte	R3, (R2)	Load first byte into R3.
LShiftL	R3, R3, #4	Shift left by 4 bit positions.
Add	R2, R2, #1	Increment the pointer.
LoadByte	R4, (R2)	Load second byte into R4.
And	R4, R4, #0xF	Clear high-order bits to zero.
Or	R3, R3, R4	Concatenate the BCD digits.
StoreByte	R3, PACKED	Store the result.

Figure 2.24   A routine that packs two BCD digits into a byte.

move the bits shifted out of one end of the operand into the other end. Two versions of both the Rotate-left and Rotate-right instructions are often provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. Figure 2.25 shows the left and right rotate operations with and without the C flag being included in the rotation. Note that when the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register. The OP codes RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations.

2.8.3   MULTIPLICATION AND DIVISION

Two signed integers can be multiplied or divided by machine instructions with the same format as we saw earlier for an Add instruction. The instruction

$$\text{Multiply} \quad Rk, Ri, Rj$$

performs the operation

$$Rk \leftarrow [Ri] \times [Rj]$$

The product of two n-bit numbers can be as large as $2n$ bits. Therefore, the answer will not necessarily fit into register Rk. A number of instruction sets have a Multiply instruction that computes the low-order n bits of the product and places it in register Rk, as indicated. This is sufficient if it is known that all products in some particular application task will fit into n bits. To accommodate the general $2n$-bit product case, some processors produce the product in two registers, usually adjacent registers Rk and $R(k + 1)$, with the high-order half being placed in register $R(k + 1)$.

An instruction set may also provide a signed integer Divide instruction

$$\text{Divide} \quad Rk, Ri, Rj$$

which performs the operation

$$Rk \leftarrow [Rj]/[Ri]$$

placing the quotient in Rk. The remainder may be placed in $R(k + 1)$, or it may be lost.

(a) Rotate left without carry   RotateL   R3, R3, #2

(b) Rotate left with carry   RotateLC   R3, R3, #2

(c) Rotate right without carry   RotateR   R3, R3, #2

(d) Rotate right with carry   RotateRC   R3, R3, #2
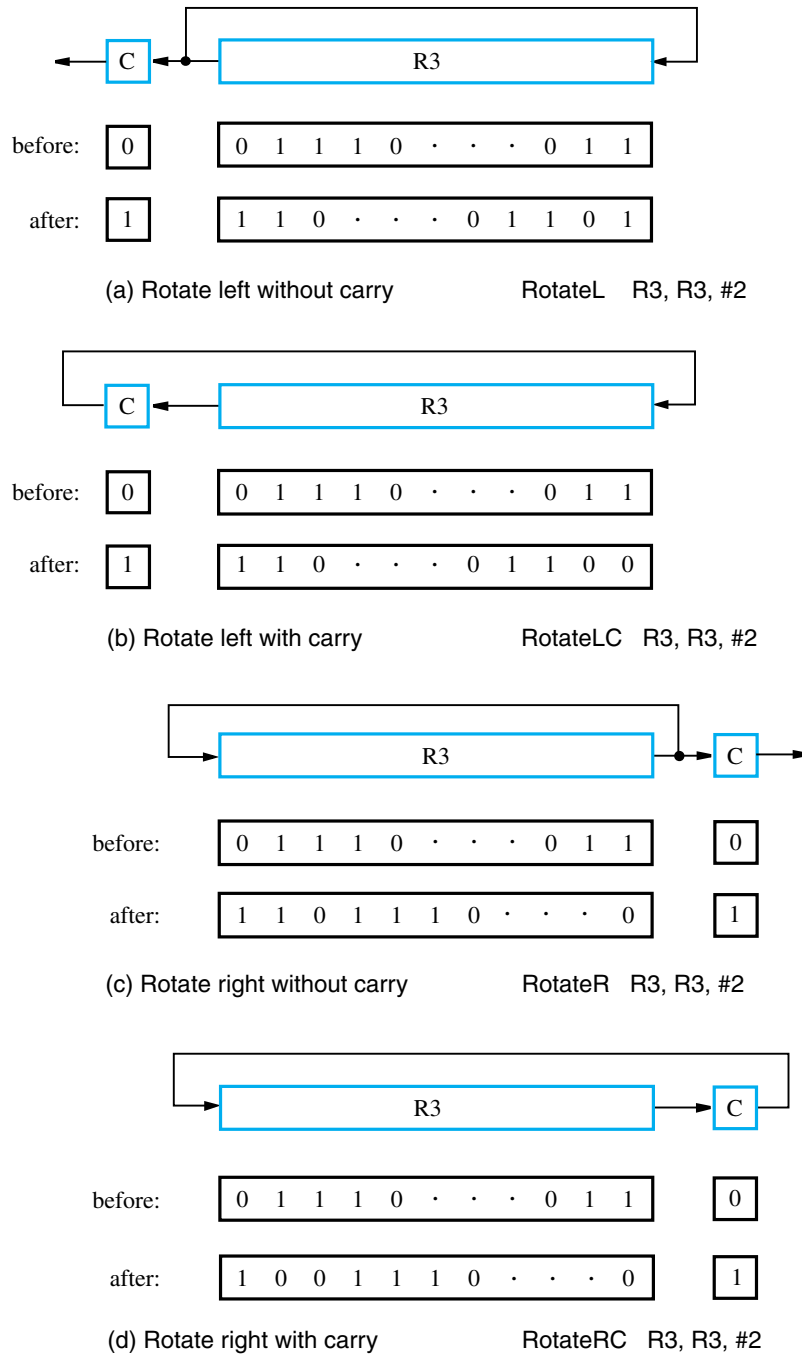
Figure 2.25   Rotate instructions.

Computers that do not have Multiply and Divide instructions can perform these and other arithmetic operations by using sequences of more basic instructions such as Add, Subtract, Shift, and Rotate. This will become more apparent when we describe the implementation of arithmetic operations in Chapter 9.

2.9 DEALING WITH 32-BIT IMMEDIATE VALUES

In the discussion of addressing modes, in Section 2.4.1, we raised the question of how a 32-bit value that represents a constant or a memory address can be loaded into a processor register. The Immediate and Absolute modes in a RISC-style processor restrict the operand size to 16 bits. Therefore, a 32-bit value cannot be given explicitly in a single instruction that must fit in a 32-bit word.

A possible solution is to use two instructions for this purpose. One approach found in RISC-style processors uses instructions that perform two different logical-OR operations. The instruction

$$\text{Or} \quad \text{Rdst, Rsrc, \#Value}$$

extends the 16-bit immediate operand by placing zeros into the high-order bit positions to form a 32-bit value, which is then ORed with the contents of register Rsrc. If Rsrc contains zero, then Rdst will just be loaded with the extended 32-bit value. Another instruction

$$\text{OrHigh} \quad \text{Rdst, Rsrc, \#Value}$$

forms a 32-bit value by taking the 16-bit immediate operand as the high-order bits and appending zeros as the low-order bits. This value is then ORed with the contents of Rsrc. Using these instructions, and assuming that R0 contains the value 0, we can load the 32-bit value 0x20004FF0 into register R2 as follows:

```
OrHigh      R2, R0, #0x2000
Or          R2, R2, #0x4FF0
```

To make it easier to write programs, a RISC-style instruction set may include pseudoinstructions that indicate an action that requires more than one machine instruction. Such pseudoinstructions are replaced with the corresponding machine-instruction sequence by the assembler program. For example, the pseudoinstruction

$$\text{MoveImmediateAddress} \quad \text{R2, LOC}$$

could be used to load a 32-bit address represented by the symbol LOC into register R2. In the assembled program, it would be replaced with two instructions using 16-bit values as shown above.

An alternative to using two instructions to load a 32-bit address into a register is to use more than one word per instruction. In that case, a two-word instruction could give the OP code and register specification in the first word, and include a 32-bit value in the second word. This is the approach found in CISC-style processors.

Finally, note that in the previous sections we always assumed that single Load and Store instructions can be used to access memory locations represented by symbolic names. This makes the example programs simpler and easier to read. The programs will run correctly if the required memory addresses can be specified in 16 bits. If longer addresses are involved, then the approach described above to construct 32-bit addresses must be used.

2.10   CISC INSTRUCTION SETS

In preceding sections, we introduced the RISC style of instruction sets. Now we will examine some important characteristics of *Complex Instruction Set Computers* (CISC).

One key difference is that CISC instruction sets are not constrained to the *load/store architecture*, in which arithmetic and logic operations can be performed only on operands that are in processor registers. Another key difference is that instructions do not necessarily have to fit into a single word. Some instructions may occupy a single word, but others may span multiple words.

Instructions in modern CISC processors typically do not use a three-address format. Most arithmetic and logic instructions use the *two-address* format

$$\text{Operation} \quad \text{destination, source}$$

An Add instruction of this type is

$$\text{Add} \quad \text{B, A}$$

which performs the operation $B \leftarrow [A] + [B]$ on memory operands. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that memory location B is both a source and a destination.

Consider again the task of adding two numbers

$$C = A + B$$

where all three operands may be in memory locations. Obviously, this cannot be done with a single two-address instruction. The task can be performed by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

$$\text{Move} \quad \text{C, B}$$

which performs the operation $C \leftarrow [B]$, leaving the contents of location B unchanged. The operation $C \leftarrow [A] + [B]$ can now be performed by the two-instruction sequence

$$\text{Move} \quad \text{C, B}$$
$$\text{Add} \quad \text{C, A}$$

Observe that by using this sequence of instructions the contents of neither A nor B locations are overwritten.

In some CISC processors one operand may be in the memory but the other must be in a register. In this case, the instruction sequence for the required task would be

$$
\begin{array}{ll}
\text{Move} & \text{R}i, \text{A} \\
\text{Add} & \text{R}i, \text{B} \\
\text{Move} & \text{C}, \text{R}i
\end{array}
$$

The general form of the Move instruction is

Move   destination, source

where both the source and destination may be either a memory location or a processor register. The Move instruction includes the functionality of the Load and Store instructions we used previously in the discussion of RISC-style processors. In the Load instruction, the source is a memory location and the destination is a processor register. In the Store instruction, the source is a register and the destination is a memory location. While Load and Store instructions are restricted to moving operands between memory and processor registers, the Move instruction has a wider scope. It can be used to move immediate operands and to transfer operands between two memory locations or between two registers.

2.10.1   ADDITIONAL ADDRESSING MODES

Most CISC processors have all of the five basic addressing modes—Immediate, Register, Absolute, Indirect, and Index. Three additional addressing modes are often found in CISC processors.

Autoincrement and Autodecrement Modes

There are two modes that are particularly convenient for accessing data items in successive locations in the memory and for implementation of stacks.

Autoincrement mode—The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory.

We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

(Ri)+

To access successive words in a byte-addressable memory with a 32-bit word length, the increment amount must be 4. Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands. Since the size of the operand is usually specified as part of the operation code of an instruction, it is sufficient to indicate the Autoincrement mode as (Ri)+.

As a companion for the Autoincrement mode, another useful mode accesses the memory locations in the reverse order:

Autodecrement mode—The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

$$-(\mathrm{R}i)$$

In this mode, operands are accessed in descending address order.

The reader may wonder why the address is decremented before it is used in the Autodecrement mode, and incremented after it is used in the Autoincrement mode. The main reason for this is to make it easy to use these modes together to implement a stack structure. Instead of needing two instructions

Subtract	SP, #4
Move	(SP), NEWITEM

to push a new item on the stack, we can use just one instruction

Move   −(SP), NEWITEM

Similarly, instead of needing two instructions

Move	ITEM, (SP)
Add	SP, #4

to pop an item from the stack, we can use just

Move   ITEM, (SP)+

Relative Mode

We have defined the Index mode by using general-purpose processor registers. Some computers have a version of this mode in which the program counter, PC, is used instead of a general-purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified *relative* to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode—The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

2.10.2   CONDITION CODES

Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero. The processor can maintain the information about these results for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called *condition code flags*. These flags are usually grouped together in a special processor register called the *condition code register* or *status register*. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The N and Z flags record whether the result of an arithmetic or logic operation is negative or zero. In some computers, they may also be affected by the value of the operand of a Move instruction. This makes it possible for a later conditional branch instruction to cause a branch based on the sign and value of the operand that was moved. Some computers also provide a special Test instruction that examines a value in a register or in the memory without modifying it, and sets or clears the N and Z flags accordingly.

The V flag indicates whether overflow has taken place. As explained in Section 1.4, overflow occurs when the result of an arithmetic operation is outside the range of values that can be represented by the number of bits available for the operands. The processor sets the V flag to allow the programmer to test whether overflow has occurred and branch to an appropriate routine that deals with the problem. Instructions such as Branch_if_overflow are usually provided for this purpose.

The C flag is set to 1 if a carry occurs from the most-significant bit position during an arithmetic operation. This flag makes it possible to perform arithmetic operations on operands that are longer than the word length of the processor. Such operations are used in multiple-precision arithmetic, which is discussed in Chapter 9.

Consider the Branch instruction in Figure 2.6. If condition codes are used, then the Subtract instruction would cause both N and Z flags to be cleared to 0 if the contents of register R2 are still greater than 0. The desired branching could be specified simply as

$$\text{Branch}{>}0 \quad \text{LOOP}$$

without indicating the register involved in the test. This instruction causes a branch if neither N nor Z is 1, that is, if the result produced by the Subtract instruction is neither negative nor equal to zero. Many conditional branch instructions are provided in the instruction set of a computer to enable a variety of conditions to be tested. The conditions are defined as logic expressions involving the condition code flags.

To illustrate the use of condition codes, consider again the program in Figure 2.8, which adds a list of numbers using RISC-style instructions. Using a CISC-style instruction set, this task can be implemented with fewer instructions, as shown in Figure 2.26. The

	Move	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Load address of the first number.
LOOP:	Add	R3, (R4)+	Add the next number to sum.
	Subtract	R2, #1	Decrement the counter.
	Branch>0	LOOP	Loop back if not finished.
	Move	SUM, R3	Store the final sum.

Figure 2.26   A CISC version of the program of Figure 2.8.

Add instruction uses the pointer register (R4) to access successive numbers in the list and add them to the sum in register R3. After accessing the source operand, the processor automatically increments the pointer, because the Autoincrement addressing mode is used to specify the source operand. The Subtract instruction sets the condition codes, which are then used by the Branch instruction.

2.11   RISC AND CISC STYLES

RISC and CISC are two different styles of instruction sets. We introduced RISC first because it is simpler and easier to understand. Having looked at some basic features of both styles, we should summarize their main characteristics.

RISC style is characterized by:

- Simple addressing modes
- All instructions fitting in a single word
- Fewer instructions in the instruction set, as a consequence of simple addressing modes
- Arithmetic and logic operations that can be performed only on operands in processor registers
- Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
- Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining which is presented in Chapter 6
- Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

CISC style is characterized by:

- More complex addressing modes
- More complex instructions, where an instruction may span multiple words

- Many instructions that implement complex tasks

- Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers

- Transfers from one memory location to another by using a single Move instruction

- Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

Before the 1970s, all computers were of CISC type. An important objective was to simplify the development of software by making the hardware capable of performing fairly complex tasks, that is, to move the complexity from the software level to the hardware level. This is conducive to making programs simpler and shorter, which was important when computer memory was smaller and more expensive to provide. Today, memory is inexpensive and most computers have large amounts of it.

RISC-style designs emerged as an attempt to achieve very high performance by making the hardware very simple, so that instructions can be executed very quickly in pipelined fashion as will be discussed in Chapter 6. This results in moving complexity from the hardware level to the software level. Sophisticated compilers were developed to optimize the code consisting of simple instructions. The size of the code became less important as memory capacities increased.

While the RISC and CISC styles seem to define two significantly different approaches, today's processors often exhibit what may seem to be a compromise between these approaches. For example, it is attractive to add some non-RISC instructions to a RISC processor in order to reduce the number of instructions executed, as long as the execution of these new instructions is fast. We will deal with the performance issues in detail in Chapter 6 where we discuss the concept of pipelining.

2.12 Example Programs

In this section we present two examples that further illustrate the use of machine instructions. The examples are representative of numeric and nonnumeric applications.

2.12.1 Vector Dot Product Program

The first example is a numerical application that is an extension of previous programs for adding numbers. In calculations that involve vectors and matrices, it is often necessary to compute the dot product of two vectors. Let A and B be two vectors of length n. Their dot product is defined as

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

Figures 2.27 and 2.28 show RISC- and CISC-style programs for computing the dot product and storing it in memory location DOTPROD. The first elements of each vector, $A(0)$ and

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

Figure 2.27   A RISC-style program for computing the dot product of two vectors.

	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Move	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Move	R6, (R2)+	Compute the product of
	Multiply	R6, (R3)+	next components.
	Add	R5, R6	Add to previous sum.
	Subtract	R4, #1	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	DOTPROD, R5	Store dot product in memory.

Figure 2.28   A CISC-style program for computing the dot product of two vectors.

$B(0)$, are stored at memory locations AVEC and BVEC, with the remaining elements in the following word locations.

The task of accumulating a sum of products occurs in many signal-processing applications. In this case, one of the vectors consists of the most recent n signal samples in a continuing time sequence of inputs to a signal-processing unit. The other vector is a set of n weights. The n signal samples are multiplied by the weights, and the sum of these products constitutes an output signal sample.

Some computer instruction sets combine the operations of the Multiply and Add instructions used in the programs in Figures 2.27 and 2.28 into a single MultiplyAccumulate instruction. This is done in the ARM processor presented in Appendix D.

2.12.2   STRING SEARCH PROGRAM

As an example of a non-numerical application, let us consider the problem of string search. Given two strings of ASCII-encoded characters, a long string T and a short string P, we want to determine if the pattern P is contained in the target T. Since P may be found in T in several places, we will simplify our task by being interested only in the first occurrence of P in T when T is searched from left to right. Let T and P consist of n and m characters, respectively, where $n > m$. The characters are stored in memory in consecutive byte locations. Assume that the required data are located as follows:

- T is the address of $T(0)$, which is the first character in string T.
- N is the address of a 32-bit word that contains the value n.
- P is the address of $P(0)$, which is the first character in string P.
- M is the address of a 32-bit word that contains the value m.
- RESULT is the address of a word in which the result of the search is to be stored. If the substring P is found in T, then the address of the corresponding location in T will be stored in RESULT; otherwise, the value -1 will be stored.

String search is an important and well-researched problem. Many algorithms have been developed. Since our main purpose is to illustrate the use of assembly-language instructions, we will use the simplest algorithm which is known as the *brute-force* algorithm. It is given in Figure 2.29.

In a RISC-style computer, the algorithm can be implemented as shown in Figure 2.30. The comments explain the use of various processor registers. Note that in the case of a failed search, the immediate value -1 will cause the contents of R8 to become equal to 0xFFFFFFFF, which represents -1 in 2's complement.

Figure 2.31 shows how the algorithm may be implemented in a CISC-style computer. Observe that the first instruction in LOOP2 loads a character from string T into register R8, which is followed by an instruction that compares this character with a character in string P. The reader may wonder why is it not possible to use a single instruction

$$\text{CompareByte}\quad (R6)+, (R7)+$$

to achieve the same effect. While CISC-style instruction sets allow operations that involve memory operands, they typically require that if one operand is in the memory, the other

```
for   i ← 0 to n − m do
      j ← 0
      while  j < m and P[j] = T[i + j] do
          j ← j + 1
      if  j = m return i
return  −1
```

Figure 2.29   A brute-force string search algorithm.

	Move	R2, #T	R2 points to string T.
	Move	R3, #P	R3 points to string P.
	Load	R4, N	Get the value n.
	Load	R5, M	Get the value m.
	Subtract	R4, R4, R5	Compute $n - m$.
	Add	R4, R2, R4	The address of $T(n - m)$.
	Add	R5, R3, R5	The address of $P(m)$.
LOOP1:	Move	R6, R2	Use R6 to scan through string T.
	Move	R7, R3	Use R7 to scan through string P.
LOOP2:	LoadByte	R8, (R6)	Compare a pair of
	LoadByte	R9, (R7)	characters in
	Branch_if_[R8]≠[R9]	NOMATCH	strings T and P.
	Add	R6, R6, #1	Point to next character in T.
	Add	R7, R7, #1	Point to next character in P.
	Branch_if_[R5]>[R7]	LOOP2	Loop again if not done.
	Store	R2, RESULT	Store the address of $T(i)$.
	Branch	DONE	
NOMATCH:	Add	R2, R2, #1	Point to next character in T.
	Branch_if_[R4]≥[R2]	LOOP1	Loop again if not done.
	Move	R8, # –1	Write –1 to indicate that
	Store	R8, RESULT	no match was found.
DONE:	next instruction		

Figure 2.30   A RISC-style program for string search.

operand must be in a processor register. A common exception is the Move instruction, which may involve two memory operands. This provides a simple way of moving data between different memory locations.

2.13   ENCODING OF MACHINE INSTRUCTIONS

In this chapter, we have introduced a variety of useful instructions and addressing modes. We have used a generic form of assembly language to emphasize basic concepts without relying on processor-specific acronyms or mnemonics. Assembly-language instructions symbolically express the actions that must be performed by the processor circuitry. To be executed in a processor, assembly-language instructions must be converted by the assembler program, as described in Section 2.5, into machine instructions that are encoded in a compact binary pattern.

Let us now examine how machine instructions may be formed. The Add instruction

<div align="center">

Add   Rdst, Rsrc1, Rsrc2

</div>

	Move	R2, #T	R2 points to string T.
	Move	R3, #P	R3 points to string P.
	Move	R4, N	Get the value n.
	Move	R5, M	Get the value m.
	Subtract	R4, R5	Compute $n - m$.
	Add	R4, R2	The address of $T(n - m)$.
	Add	R5, R3	The address of $P(m)$.
LOOP1:	Move	R6, R2	Use R6 to scan through string T.
	Move	R7, R3	Use R7 to scan through string P.
LOOP2:	MoveByte	R8, (R6)+	Compare a pair of
	CompareByte	R8, (R7)+	characters in
	Branch$\neq$0	NOMATCH	strings T and P.
	Compare	R5, R7	Check if at $P(m)$.
	Branch>0	LOOP2	Loop again if not done.
	Move	RESULT, R2	Store the address of $T(i)$.
	Branch	DONE	
NOMATCH:	Add	R2, #1	Point to next character in T.
	Compare	R4, R2	Check if at $T(n - m)$.
	Branch$\geq$0	LOOP1	Loop again if not done.
	Move	RESULT, #−1	No match was found.
DONE:	next instruction		

**Figure 2.31**   A CISC-style program for string search.

is representative of a class of three-operand instructions that use operands in processor registers. Registers Rdst, Rsrc1, and Rsrc2 hold the destination and two source operands. If a processor has 32 registers, then it is necessary to use five bits to specify each of the three registers in such instructions. If each instruction is implemented in a 32-bit word, the remaining 17 bits can be used to specify the OP code that indicates the operation to be performed. A possible format is shown in Figure 2.32*a*.

Now consider instructions in which one operand is given using the Immediate addressing mode, such as

$$\text{Add} \quad \text{Rdst, Rsrc, #Value}$$

Of the 32 bits available, ten bits are needed to specify the two registers. The remaining 22 bits must give the OP code and the value of the immediate operand. The most useful sizes of immediate operands are 32, 16, and 8 bits. Since 32 bits are not available, a good choice is to allocate 16 bits for the immediate operand. This leaves six bits for specifying the OP code. A possible format is presented in Figure 2.32*b*. This format can also be used for Load and Store instructions, where the Index addressing mode uses the 16-bit field to specify the offset that is added to the contents of the index register.
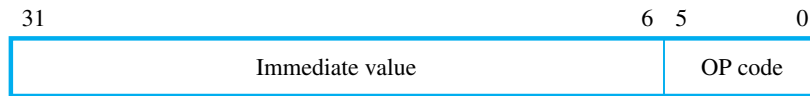
The format in Figure 2.32*b* can also be used to encode the Branch instructions. Consider the program in Figure 2.12. The Branch-greater-than instruction at memory address 128

31	27 26	22 21	17 16	0
Rsrc1	Rsrc2	Rdst	OP code	
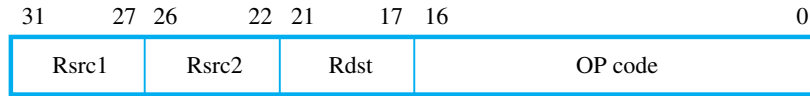
(a) Register-operand format

31	27 26	22 21	6 5	0
Rsrc	Rdst	Immediate operand	OP code	

(b) Immediate-operand format

31	6 5	0
Immediate value	OP code	

(c) Call format

Figure 2.32   Possible instruction formats.

could be written in a specific assembly language as

$$\text{BGT} \quad \text{R2, R0, LOOP}$$

if the contents of register R0 are zero. The registers R2 and R0 can be specified in the two register fields in Figure 2.32b. The six-bit OP code has to identify the BGT operation. The 16-bit immediate field can be used to provide the information needed to determine the branch target address, which is the location of the instruction with the label LOOP. The target address generally comprises 32 bits. Since there is no space for 32 bits, the BGT instruction makes use of the immediate field to give an offset from the location of this instruction in the program to the required branch target. At the time the BGT instruction is being executed, the program counter, PC, has been incremented to point to the next instruction, which is the Store instruction at address 132. Therefore, the branch offset is $132 - 112 = 20$. Since the processor computes the target address by adding the current contents of the PC and the branch offset, the required offset in this example is negative, namely -20.

Finally, we should consider the Call instruction, which is used to call a subroutine. It only needs to specify the OP code and an immediate value that is used to determine the address of the first instruction in the subroutine. If six bits are used for the OP code, then the remaining 26 bits can be used to denote the immediate value. This gives the format shown in Figure 2.32c.

In this section, we introduced the basic concept of encoding the machine instructions. Different commercial processors have instruction sets that vary in the details of implementation. Appendices B to E present the instruction sets of four processors that we have chosen as examples.

https://hemanthrajhemu.github.io

2.14   Concluding Remarks

This chapter introduced the representation and execution of instructions and programs at the assembly and machine level as seen by the programmer. The discussion emphasized the basic principles of addressing techniques and instruction sequencing. The programming examples illustrated the basic types of operations implemented by the instruction set of any modern computer. Commonly used addressing modes were introduced. The subroutine concept and the instructions needed to implement it were discussed. In the discussion in this chapter, we provided the contrast between two different approaches to the design of machine instruction sets—the RISC and CISC approaches.

2.15   Solved Problems

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Problem: Assume that there is a string of ASCII-encoded characters stored in memory starting at address STRING. The string ends with the Carriage Return (CR) character. Write a RISC-style program to determine the length of the string and store it in location LENGTH.   **Example 2.1**

Solution: Figure 2.33 presents a possible program. The characters in the string are compared to CR (ASCII code 0x0D), and a counter is incremented until the end of the string is reached.

	Move	R2, #STRING	R2 points to the start of the string.
	Clear	R3	R3 is a counter that is cleared to 0.
	Move	R4, #0x0D	ASCII code for Carriage Return.
LOOP:	LoadByte	R5, (R2)	Get the next character.
	Branch_if_[R5]=[R4]	DONE	Finished if character is CR.
	Add	R2, R2, #1	Increment the string pointer.
	Add	R3, R3, #1	Increment the counter.
	Branch	LOOP	Not finished, loop back.
DONE:	Store	R3, LENGTH	Store the count in location LENGTH.

Figure 2.33   Program for Example 2.1.

LIST	EQU	1000	Starting address of the list.
	ORIGIN	400	
	Move	R2, #LIST	R2 points to the start of the list.
	Load	R3, 4(R2)	R3 is a counter, initialize it with n.
	Add	R4, R2, #8	R4 points to the first number.
	Load	R5, (R4)	R5 holds the smallest number found so far.
LOOP:	Subtract	R3, R3, #1	Decrement the counter.
	Branch_if_[R3]=0	DONE	Finished if R3 is equal to 0.
	Add	R4, R4, #4	Increment the list pointer.
	Load	R6, (R4)	Get the next number.
	Branch_if_[R5]≤[R6]	LOOP	Check if smaller number found.
	Move	R5, R6	Update the smallest number found.
	Branch	LOOP	
DONE:	Store	R5, (R2)	Store the smallest number into SMALL.
	ORIGIN	1000	
SMALL:	RESERVE	4	Space for the smallest number found.
N:	DATAWORD	7	Number of entries in the list.
ENTRIES:	DATAWORD 4,5,3,6,1,8,2		Entries in the list.
	END		

Figure 2.34   Program for Example 2.2.

Example 2.2   **Problem:** We want to find the smallest number in a list of 32-bit positive integers. The word at address 1000 is to hold the value of the smallest number after it has been found. The next word contains the number of entries, n, in the list. The following n words contain the numbers in the list. The program is to start at address 400. Write a RISC-style program to find the smallest number and include the assembler directives needed to organize the program and data as specified. While the program has to be able to handle lists of different lengths, include in your code a small list of sample data comprising seven integers.

Solution: The program in Figure 2.34 accomplishes the required task. Comments in the program explain how this task is performed.

Example 2.3   **Problem:** Write a RISC-style program that converts an n-digit decimal integer into a binary number. The decimal number is given as n ASCII-encoded characters, as would be the case if the number is entered by typing it on a keyboard. Memory location N contains n, the ASCII string starts at DECIMAL, and the converted number is stored at BINARY.

Solution: Consider a four-digit decimal number, $D = d_3 d_2 d_1 d_0$. The value of this number is $((d_3 \times 10 + d_2) \times 10 + d_1) \times 10 + d_0$. This representation of the number is the basis for the conversion technique used in the program in Figure 2.35. Note that each ASCII-encoded

	Load	R2, N	Initialize counter R2 with n.
	Move	R3, #DECIMAL	R3 points to the ASCII digits.
	Clear	R4	R4 will hold the binary number.
LOOP:	LoadByte	R5, (R3)	Get the next ASCII digit.
	And	R5, R5, #0x0F	Form the BCD digit.
	Add	R4, R4, R5	Add to the intermediate result.
	Add	R3, R3, #1	Increment the digit pointer.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_[R2]=0	DONE	
	Multiply	R4, R4, #10	Multiply by 10.
	Branch	LOOP	Loop back if not done.
DONE:	Store	R4, BINARY	Store result in location BINARY.

Figure 2.35   Program for Example 2.3.

character is converted into a Binary Coded Decimal (BCD) digit before it is used in the computation. It is assumed that the converted value can be represented in no more than 32 bits.

Problem: Consider an array of numbers A(i,j), where $i = 0$ through $n - 1$ is the row index, and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of a computer one row after another, with elements of each row occupying m successive word locations. Assume that the memory is byte-addressable and that the word length is 32 bits. Write a RISC-style subroutine for adding column x to column y, element by element, leaving the sum elements in column y. The indices x and y are passed to the subroutine in registers R2 and R3. The parameters n and m are passed to the subroutine in registers R4 and R5, and the address of element A(0,0) is passed in register R6.   **Example 2.4**

Solution: A possible program is given in Figure 2.36. We have assumed that the values x, y, n, and m are stored in memory locations X, Y, N, and M. Also, the elements of the array are stored in successive words that begin at location ARRAY, which is the address of the element A(0,0). Comments in the program indicate the purpose of individual instructions.

Problem: We want to sort a list of characters stored in memory. The list consists of n bytes, not necessarily distinct, and each byte contains the ASCII code for a character from the set of letters A through Z. In the ASCII code, presented in Chapter 1, the letters A, B, . . . , Z, are represented by 7-bit patterns that have increasing values when interpreted as binary numbers. When an ASCII character is stored in a byte location, it is customary to set the most-significant bit position to 0. Using this code, we can sort a list of characters alphabetically by sorting their codes in increasing numerical order, considering them as positive numbers.   **Example 2.5**

	Load	R2, X	Load the value x.
	Load	R3, Y	Load the value y.
	Load	R4, N	Load the value n.
	Load	R5, M	Load the value m.
	Move	R6, #ARRAY	Load the address of A(0,0).
	Call	SUB	
	next instruction		
	⋮		
SUB:	Subtract	SP, SP, #4	
	Store	R7, (SP)	Save register R7.
	LShiftL	R5, R5, #2	Determine the distance in bytes between successive elements in a column.
	Subtract	R3, R3, R2	Form $y - x$.
	LShiftL	R3, R3, #2	Form $4(y - x)$.
	LShiftL	R2, R2, #2	Form $4x$.
	Add	R6, R6, R2	R6 points to A(0,x).
	Add	R7, R6, R3	R7 points to A(0,y).
LOOP:	Load	R2, (R6)	Get the next number in column x.
	Load	R3, (R7)	Get the next number in column y.
	Add	R2, R2, R3	Add the numbers and
	Store	R2, (R7)	store the sum.
	Add	R6, R6, R5	Increment pointer to column x.
	Add	R7, R7, R5	Increment pointer to column y.
	Subtract	R4, R4, #1	Decrement the row counter.
	Branch_if_[R4]>0	LOOP	Loop back if not done.
	Load	R7, (SP)	Restore R7.
	Add	SP, SP, #4	
	Return		Return to the calling program.

Figure 2.36   Program for Example 2.4.

Let the list be stored in memory locations LIST through LIST $+ n - 1$, and let n be a 32-bit value stored at address N. The sorting is to be done in place, that is, the sorted list is to occupy the same memory locations as the original list.

We can sort the list using a straight-selection sort algorithm. First, the largest number is found and placed at the end of the list in location LIST $+ n - 1$. Then the largest number in the remaining sublist of $n - 1$ numbers is placed at the end of the sublist in location LIST $+ n - 2$. The procedure is repeated until the list is sorted. A C-language program for this sorting algorithm is shown in Figure 2.37, where the list is treated as a one-dimensional array LIST(0) through LIST($n - 1$). For each sublist LIST(j) through LIST(0), the number in LIST(j) is compared with each of the other numbers in the sublist. Whenever a larger number is found in the sublist, it is interchanged with the number in LIST(j).

---

for   (j = n−1; j > 0; j = j − 1)
        { for (k = j−1; k > = 0; k = k − 1)
            { if   (LIST[k] > LIST[j])
                    { TEMP = LIST[k];
                        LIST[k] = LIST[j];
                        LIST[j] = TEMP;
                    }
            }

        }

Figure 2.37   C-language program for sorting.

	Move	R2, #LIST	Load LIST into base register R2.
	Move	R3, N	Initialize outer loop index
	Subtract	R3, #1	register R3 to $j = n − 1$.
OUTER:	Move	R4, R3	Initialize inner loop index
	Subtract	R4, #1	register R4 to $k = j − 1$.
	MoveByte	R5, (R2,R3)	Load LIST(j) into R5, which holds
			current maximum in sublist.
INNER:	CompareByte	(R2,R4), R5	If LIST(k) ≤ [R5],
	Branch≤0	NEXT	do not exchange.
	MoveByte	R6, (R2,R4)	Otherwise, exchange LIST(k)
	MoveByte	(R2,R4), R5	with LIST(j) and load
	MoveByte	(R2,R3), R6	new maximum into R5.
	MoveByte	R5, R6	Register R6 serves as TEMP.
NEXT:	Decrement	R4	Decrement index registers R4 and
	Branch≥0	INNER	R3, which also serve as
	Decrement	R3	loop counters, and branch
	Branch>0	OUTER	back if loops not finished.

Figure 2.38   A byte-sorting program.

Note that the C-language program traverses the list backwards. This order of traversal simplifies loop termination when a machine language program is written, because the loop is exited when an index is decremented to 0.

Write a CISC-style program that implements this sorting task.

Solution: A possible program is given in Figure 2.38.

PROBLEMS

2.1 [E] Given a binary pattern in some memory location, is it possible to tell whether this pattern represents a machine instruction or a number?

2.2 [E] Consider a computer that has a byte-addressable memory organized in 32-bit words according to the big-endian scheme. A program reads ASCII characters entered at a keyboard and stores them in successive byte locations, starting at location 1000. Show the contents of the two memory words at locations 1000 and 1004 after the word "Computer" has been entered.

2.3 [E] Repeat Problem 2.2 for the little-endian scheme.

2.4 [E] Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case?

(*a*) 12(R4)

(*b*) (R4,R5)

(*c*) 28(R4,R5)

(*d*) (R4)+

(*e*) −(R4)

2.5 [E] Write a RISC-style program that computes the expression SUM $= 580 + 68400 + 80000$.

2.6 [E] Write a CISC-style program for the task in Problem 2.5.

2.7 [E] Write a RISC-style program that computes the expression ANSWER $=$ A $\times$ B $+$ C $\times$ D.

2.8 [E] Write a CISC-style program for the task in Problem 2.7.

2.9 [M] Rewrite the addition loop in Figure 2.8 so that the numbers in the list are accessed in the reverse order; that is, the first number accessed is the last one in the list, and the last number accessed is at memory location NUM1. Try to achieve the most efficient way to determine loop termination. Would your loop execute faster than the loop in Figure 2.8?

2.10 [M] The list of student marks shown in Figure 2.10 is changed to contain j test scores for each student. Assume that there are n students. Write a RISC-style program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM $+$ 4, SUM $+$ 8, ... The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.11 for the 3-test case cannot be used. Use two nested loops. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that the memory area used to store the sums has been cleared to zero initially.

2.11 [M] Write a RISC-style program that finds the number of negative integers in a list of n 32-bit integers and stores the count in location NEGNUM. The value n is stored in memory location N, and the first integer in the list is stored in location NUMBERS. Include the necessary assembler directives and a sample list that contains six numbers, some of which are negative.

https://hemanthrajhemu.github.io

2.12 **[E]** Both of the following statement segments cause the value 300 to be stored in location 1000, but at different times.

ORIGIN	1000
DATAWORD	300

and

Move	R2, #1000
Move	R3, #300
Store	R3, (R2)

Explain the difference.

2.13 **[E]** Write an assembly-language program in the style of Figure 2.13 for the program in Figure 2.11. Assume the data layout of Figure 2.10.

2.14 **[E]** Write a CISC-style program for the task in Example 2.1. At most one operand of an instruction can be in the memory.

2.15 **[E]** Write a CISC-style program for the task in Example 2.2. At most one operand of an instruction can be in the memory.

2.16 **[M]** Write a CISC-style program for the task in Example 2.3. At most one operand of an instruction can be in the memory.

2.17 **[M]** Write a CISC-style program for the task in Example 2.4. At most one operand of an instruction can be in the memory.

2.18 **[M]** Write a RISC-style program for the task in Example 2.5.

2.19 **[E]** Register R5 is used in a program to point to the top of a stack containing 32-bit numbers. Write a sequence of instructions using the Index, Autoincrement, and Autodecrement addressing modes to perform each of the following tasks:

(*a*) Pop the top two items off the stack, add them, then push the result onto the stack.

(*b*) Copy the fifth item from the top into register R3.

(*c*) Remove the top ten items from the stack.

For each case, assume that the stack contains ten or more elements.

2.20 **[M]** Show the processor stack contents and the contents of the stack pointer, SP, immediately after each of the following instructions in the program in Figure 2.18 is executed. Assume that [SP] = 1000 at Level 1, before execution of the calling program begins.

(*a*) The second Store instruction in the subroutine

(*b*) The last Load instruction in the subroutine

(*c*) The last Store instruction in the calling program

2.21 **[M]** Consider the following possibilities for saving the return address of a subroutine:

(*a*) In a processor register

(*b*) In a memory location associated with the call, so that a different location is used when the subroutine is called from different places

(*c*) On a stack

Which of these possibilities supports subroutine nesting and which supports subroutine recursion (that is, a subroutine that calls itself)?

2.22 **[M]** In addition to the processor stack, it may be convenient to use another stack in some programs. The second stack is usually allocated a fixed amount of space in the memory. In this case, it is important to avoid pushing an item onto the stack when the stack has reached its maximum size. Also, it is important to avoid attempting to pop an item off an empty stack, which could result from a programming error. Write two short RISC-style routines, called SAFEPUSH and SAFEPOP, for pushing onto and popping off this stack structure, while guarding against these two possible errors. Assume that the element to be pushed/popped is located in register R2, and that register R5 serves as the stack pointer for this user stack. The stack is full if its topmost element is stored in location TOP, and it is empty if the last element popped was stored in location BOTTOM. The routines should branch to FULLERROR and EMPTYERROR, respectively, if errors occur. All elements are of word size, and the stack grows toward lower-numbered address locations.

2.23 **[M]** Repeat Problem 2.22 for CISC-style routines that can use Autoincrement and Autodecrement addressing modes.

2.24 **[D]** Another useful data structure that is similar to the stack is called a *queue*. Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

A FIFO queue of bytes is to be implemented in the memory, occupying a fixed region of k bytes. The necessary pointers are an IN pointer and an OUT pointer. The IN pointer keeps track of the location where the next byte is to be appended to the back of the queue, and the OUT pointer keeps track of the location containing the next byte to be removed from the front of the queue.

(*a*) As data items are added to the queue, they are added at successively higher addresses until the end of the memory region is reached. What happens next, when a new item is to be added to the queue?

(*b*) Choose a suitable definition for the IN and OUT pointers, indicating what they point to in the data structure. Use a simple diagram to illustrate your answer.

(*c*) Show that if the state of the queue is described only by the two pointers, the situations when the queue is completely full and completely empty are indistinguishable.

(*d*) What condition would you add to solve the problem in part (*c*)?

(*e*) Propose a procedure for manipulating the two pointers IN and OUT to append and remove items from the queue.

2.25 **[M]** Consider the queue structure described in Problem 2.24. Write APPEND and RE-MOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

2.26 **[M]** The dot-product computation is discussed in Section 2.12.1. This type of computation can be used in the following signal-processing task. An input signal time sequence IN(0), IN(1), IN(2), IN(3), . . . , is processed by a 3-element weight vector (WT(0), WT(1), WT(2)) = (1/8, 1/4, 1/2) to produce an output signal time sequence OUT(0), OUT(1), OUT(2), OUT(3), . . . , as follows:

$$OUT(0) = WT(0) \times IN(0) + WT(1) \times IN(1) + WT(2) \times IN(2)$$
$$OUT(1) = WT(0) \times IN(1) + WT(1) \times IN(2) + WT(2) \times IN(3)$$
$$OUT(2) = WT(0) \times IN(2) + WT(1) \times IN(3) + WT(2) \times IN(4)$$
$$OUT(3) = WT(0) \times IN(3) + WT(1) \times IN(4) + WT(2) \times IN(5)$$
$$\vdots$$

All signal and weight values are 32-bit signed numbers. The weights, inputs, and outputs, are stored in the memory starting at locations WT, IN, and OUT, respectively. Write a RISC-style program to calculate and store the output values for the first *n* outputs, where *n* is stored at location N.

Hint: Arithmetic right shifts can be used to do the multiplications.

2.27 **[M]** Write a subroutine MEMCPY for copying a sequence of bytes from one area in the main memory to another area. The subroutine should accept three input parameters in registers representing the *from* address, the *to* address, and the *length* of the sequence to be copied. The two areas may overlap. In all but one case, the subroutine should copy the bytes in the order of increasing addresses. However, in the case where the *to* address falls within the sequence of bytes to be copied, i.e., when the *to* address is between *from* and *from+length−1*, the subroutine must copy the bytes in the order of decreasing addresses by starting at the end of the sequence of bytes to be copied in order to avoid overwriting bytes that have not yet been copied.

2.28 **[M]** Write a subroutine MEMCMP for performing a byte-by-byte comparison of two sequences of bytes in the main memory. The subroutine should accept three input parameters in registers representing the *first* address, the *second* address, and the *length* of the sequences to be compared. It should use a register to return the count of the number of comparisons that do not match.

2.29 **[M]** Write a subroutine called EXCLAIM that accepts a single parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes representing an arbitrary collection of sentences, with the NUL control character (value 0) at the end of the string. The subroutine should scan the string beginning at address STRNG and replace every occurrence of a period ('.') with an exclamation mark ('!').

2.30 **[M]** Write a subroutine called ALLCAPS that accepts a parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes, with the NUL control character (value 0) at the end of the string. The subroutine should scan the string beginning at address STRNG and replace every occurrence of a lower-case letter ('a'—'z') with the corresponding upper-case letter ('A'—'Z').

2.31 **[M]** Write a subroutine called WORDS that accepts a parameter in a register representing the starting address STRNG in the main memory for a string of ASCII characters in successive bytes, with the NUL control character (value 0) at the end of the string. The string represents English text with the space character between words. The subroutine has to determine the number of words in the string (excluding the punctuation characters). It must return the result to the calling program in a register.

2.32 **[D]** Write a subroutine called INSERT that places a number in the correct ordered position within a list of positive numbers that are stored in increasing order of value. Three input parameters should be passed to the subroutine in processor registers, representing the starting address of the ordered list of numbers, the length of the list, and the new value to be inserted into the list. The subroutine should locate the appropriate position for the new value in the list, then shift all of the larger numbers up by one position to create space for storing the new value in the list.

2.33 **[D]** Write a subroutine called INSERTSORT that repeatedly uses the INSERT subroutine in Problem 2.32 to take an unordered list of numbers and create a new list with the same numbers in increasing order. The subroutine should accept three input parameters in registers representing the starting address OLDLIST for the unordered sequence of numbers, the length of the list, and the starting address NEWLIST for the ordered sequence of numbers.