

---

**CHAPTER**

**2**

**MACHINE INSTRUCTIONS  
AND PROGRAMS**

**CHAPTER OBJECTIVES**

In this chapter you will learn about:

- Machine instructions and program execution, including branching and subroutine call and return operations
- Number representation and addition/subtraction in the 2's-complement system
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Program-controlled Input/Output operations
- Operations on stack, queue, list, linked-list, and array data structures

This chapter considers the way programs are executed in a computer from the machine instruction set viewpoint. Chapter 1 introduced the general concept that both program instructions and data operands are stored in the memory. In this chapter, we 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 commonly used for accessing operands in memory locations and processor registers are 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. In assembly language, machine instructions and operand addressing information are represented by symbolic names. A complete instruction set is often referred to as the *instruction set architecture* (ISA) of a processor. In addition to specifying instructions, an ISA also specifies the addressing methods used for accessing data operands and the processor registers available for use by the instructions. 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 needed.

Chapter 3 presents ISAs for three commercial processors produced by the ARM, Motorola, and Intel companies. This chapter's generic programs are presented in Chapter 3 in each of those three instruction sets, providing the reader with examples from real machines.

The vast majority of programs are written in high-level languages such as C, C++, Java, or Fortran. The main purpose of using assembly language programming in this book is to describe how computers operate. To execute a high-level language program on a processor, the program must first be translated into the assembly language of that processor. The assembly language is a readable representation of the machine language for the processor. The relationship between high-level language and machine language features is a key consideration in computer design. We will discuss this issue a number of times.

All computers deal with numbers. They have instructions that perform basic arithmetic operations on data operands. Also, during the process of executing the machine instructions of a program, it is necessary to perform arithmetic operations to generate the numbers that represent addresses for accessing operand locations in the memory. To understand how these tasks are accomplished, the reader must know how numbers are represented in a computer and how they are manipulated in addition and subtraction operations. Therefore, in the first section of this chapter, we will introduce this topic. A detailed discussion of logic circuits that implement computer arithmetic is given in Chapter 6.

In addition to numeric data, computers deal with characters and character strings in order to process textual information. Here, in the first section, we also describe how characters are represented in the computer.

## 2.1 NUMBERS, ARITHMETIC OPERATIONS, AND CHARACTERS

Computers are built using logic circuits that operate on information represented by two-valued electrical signals (see Appendix A). We label the two values as 0 and 1; and we define the amount of information represented by such a signal as a *bit* of information, where bit stands for *binary digit*. The most natural way to represent a number in a computer system is by a string of bits, called a binary number. A text character can also be represented by a string of bits called a character code.

We will first describe binary number representations and arithmetic operations on these numbers, and then describe character representations.

### 2.1.1 NUMBER REPRESENTATION

Consider an  $n$ -bit vector

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

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

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

We obviously 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 2.1 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 significant bit ( $b_3$  in Figure 2.1) 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. Clearly, the same operation, bit complementing, is done in converting a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. The operation of forming the 1's-complement of a given number is equivalent to subtracting that number from  $2^n - 1$ , that is, from 1111 in the case of the 4-bit numbers in Figure 2.1. Finally, in the *2's-complement* system, forming the 2's-complement of a number is done by subtracting that number from  $2^n$ .

$B$	Values represented		
	$b_3b_2b_1b_0$	Sign and magnitude	1'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 2.1** Binary, signed-integer representations.

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, 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 seems unnatural. However, we will show in Section 2.1.3 that the 2's-complement system yields the most efficient way to carry out addition and subtraction operations. It is the one most often used in computers.

## 2.1.2 ADDITION OF POSITIVE NUMBERS

Consider adding two 1-bit numbers. The results are shown in Figure 2.2. Note that the sum of 1 and 1 requires the 2-bit vector 10 to represent 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.

$$\begin{array}{r}
 0 \\
 + 0 \\
 \hline
 0
 \end{array}
 \quad
 \begin{array}{r}
 1 \\
 + 0 \\
 \hline
 1
 \end{array}
 \quad
 \begin{array}{r}
 0 \\
 + 1 \\
 \hline
 1
 \end{array}
 \quad
 \begin{array}{r}
 1 \\
 + 1 \\
 \hline
 10
 \end{array}$$

$\uparrow$   
 Carry-out

**Figure 2.2** Addition of 1-bit numbers.

### 2.1.3 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

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.

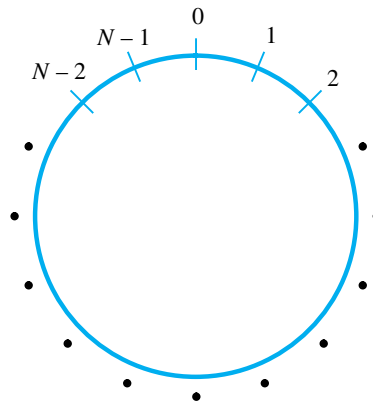
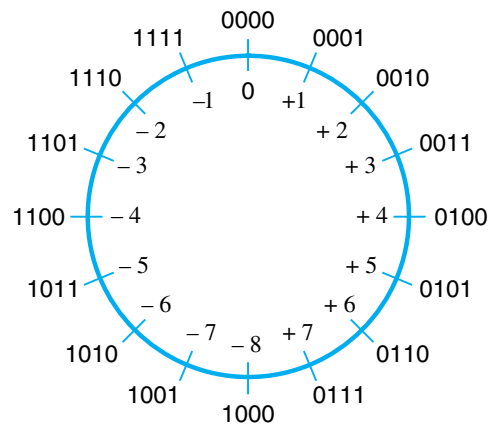
To understand 2's-complement arithmetic, consider addition modulo  $N$  (written as  $\text{mod } N$ ). A helpful graphical device for the description of addition  $\text{mod } N$  of positive integers is a circle with the  $N$  values, 0 through  $N - 1$ , marked along its perimeter, as shown in Figure 2.3a. Consider the case  $N = 16$ . The operation  $(7 + 4) \text{ mod } 16$  yields the value 11. To perform this operation graphically, locate 7 on the circle and then move 4 units in the clockwise direction to arrive at the answer 11. Similarly,  $(9 + 14) \text{ mod } 16 = 7$ ; this is modeled on the circle by locating 9 and moving 14 units in the clockwise direction to arrive at the answer 7. This graphical technique works for the computation of  $(a + b) \text{ mod } 16$  for any positive numbers  $a$  and  $b$ , that is, to perform addition, locate  $a$  and move  $b$  units in the clockwise direction to arrive at  $(a + b) \text{ mod } 16$ .

Now consider a different interpretation of the  $\text{mod } 16$  circle. Let the values 0 through 15 be represented by the 4-bit binary vectors 0000, 0001,  $\dots$ , 1111, according to the binary number system. Then reinterpret these binary vectors to represent the signed numbers from  $-8$  through  $+7$  in the 2's-complement method (see Figure 2.1), as shown in Figure 2.3b.

Let us apply the  $\text{mod } 16$  addition technique to the simple 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 2.3b. Then move 1101 (13) steps in the clockwise direction to arrive at 0100, which yields the correct answer of  $+4$ . If we perform this addition by adding bit pairs from right to left, we obtain

$$\begin{array}{r}
 0111 \\
 + 1101 \\
 \hline
 10100
 \end{array}$$

$\uparrow$   
 Carry-out

(a) Circle representation of integers mod  $N$ 

(b) Mod 16 system for 2's-complement numbers

**Figure 2.3** Modular number systems and the 2's-complement system.

Note that 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 2.3b, the value next to 1111 would normally be 10000. Instead, we go back to the value 0000.

We now state the rules governing the addition and subtraction of  $n$ -bit signed numbers using the 2's-complement representation system.

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

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

Figure 2.4 shows some examples of addition and subtraction. In all these 4-bit examples, the answers fall into the representable range of  $-8$  through  $+7$ . When answers do not fall within the representable range, we say that arithmetic overflow has occurred. The next section discusses such situations. The four addition operations (a) through (d) in Figure 2.4 follow rule 1, and the six subtraction operations (e) through (j) follow rule 2. The subtraction operation requires the subtrahend (the bottom value) to be

(a)	0010	(+2)		(b)	0100	(+4)
	+ 0011	(+3)			+ 1010	(-6)
	0101	(+5)			1110	(-2)
(c)	1011	(-5)		(d)	0111	(+7)
	+ 1110	(-2)			+ 1101	(-3)
	1001	(-7)			0100	(+4)
(e)	1101	(-3)	⇒		1101	
	- 1001	(-7)			+ 0111	
					0100	(+4)
(f)	0010	(+2)	⇒		0010	
	- 0100	(+4)			+ 1100	
					1110	(-2)
(g)	0110	(+6)	⇒		0110	
	- 0011	(+3)			+ 1101	
					0011	(+3)
(h)	1001	(-7)	⇒		1001	
	- 1011	(-5)			+ 0101	
					1110	(-2)
(i)	1001	(-7)	⇒		1001	
	- 0001	(+1)			+ 1111	
					1000	(-8)
(j)	0010	(+2)	⇒		0010	
	- 1101	(-3)			+ 0011	
					0101	(+5)

**Figure 2.4** 2's-complement add and subtract operations.

2's-complemented. This operation is done in exactly the same manner for both positive and negative numbers.

We often need to represent a number in the 2's-complement system by using a number of bits that is larger than some given size. For a positive number, this is achieved by adding 0s to the left. For a negative number, the leftmost bit, which is the sign bit, is a 1, and a longer number with the same value is obtained by replicating the sign bit to the left as many times as desired. To see why this is correct, examine the mod 16 circle of Figure 2.3*b*. Compare it to larger circles for the mod 32 or mod 64 cases. The representations for values  $-1$ ,  $-2$ , etc., would be 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*.

The simplicity of either adding or 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 cycle, which is conditional on the carry-out from the add operation, means that addition and subtraction cannot be implemented as conveniently in the 1's-complement system as in the 2's-complement system.

#### 2.1.4 OVERFLOW IN INTEGER ARITHMETIC

In the 2's-complement number representation system,  $n$  bits can represent values in the range  $-2^{n-1}$  to  $+2^{n-1} - 1$ . For example, using four bits, the range of numbers that can be represented is  $-8$  through  $+7$ , as shown in Figure 2.1. When the result of an arithmetic operation is outside the representable range, an *arithmetic overflow* has occurred.

When adding unsigned numbers, the carry-out,  $c_n$ , from the most significant bit position serves as the overflow indicator. However, this does not work for adding signed numbers. For example, when using 4-bit signed numbers, if we try to add the numbers  $+7$  and  $+4$ , the output sum vector,  $S$ , is 1011, which is the code for  $-5$ , an incorrect result. The carry-out signal from the MSB position is 0. Similarly, if we try to add  $-4$  and  $-6$ , we get  $S = 0110 = +6$ , another incorrect result, and in this case, the carry-out signal is 1. Thus, overflow may occur if both summands have the same sign. Clearly, the addition of numbers with different signs cannot cause overflow. This leads to the following conclusions:

1. Overflow can occur only when adding two numbers that have the same sign.
2. The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.

A simple way to detect overflow is to examine the signs of the two summands  $X$  and  $Y$  and the sign of the result. When both operands  $X$  and  $Y$  have the same sign, an overflow occurs when the sign of  $S$  is not the same as the signs of  $X$  and  $Y$ .

### 2.1.5 CHARACTERS

In addition to numbers, computers must be able to handle nonnumeric text information consisting of characters. Characters can be letters of the alphabet, decimal digits, punctuation marks, and so on. They are represented by codes that are usually eight bits long. One of the most widely used such codes is the American Standards Committee on Information Interchange (ASCII) code described in Appendix E.

## 2.2 MEMORY LOCATIONS AND ADDRESSES

Number and character operands, as well as instructions, are stored in the memory of a computer. We will now consider how the memory 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 2.5.

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 2's-complement number or four ASCII characters, each occupying 8 bits, as shown in Figure 2.6. 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 item location. It is customary to use numbers from 0 through  $2^k - 1$ , for some suitable value of  $k$ , as the addresses of successive locations in the memory. The  $2^k$  addresses constitute the *address space* of the computer, and the memory can have up to  $2^k$  addressable locations. 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 that are commonly used are K (kilo) for the number  $2^{10}$  (1,024), and T (tera) for the number  $2^{40}$ .

### 2.2.1 BYTE ADDRESSABILITY

We now have three basic information quantities to deal with: the 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

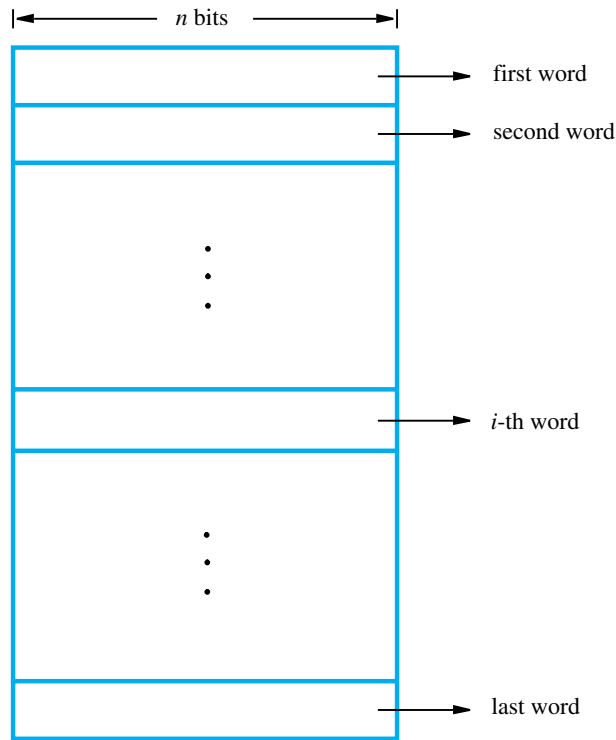
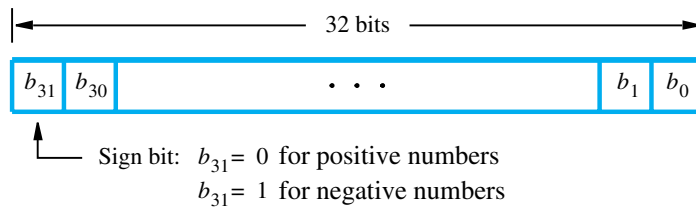
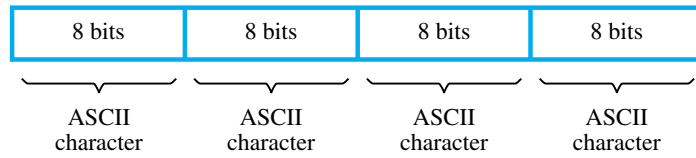


Figure 2.5 Memory words.



(a) A signed integer



(b) Four characters

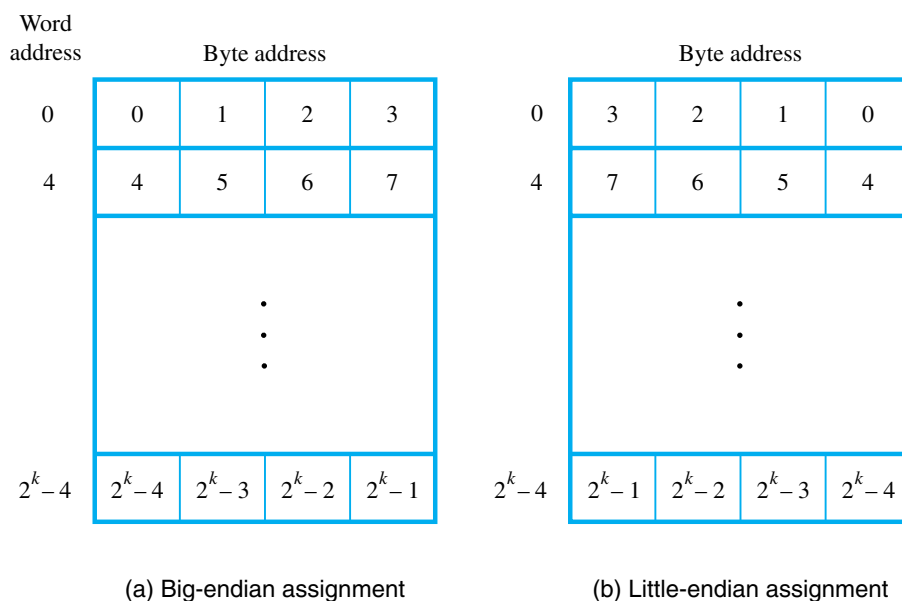
Figure 2.6 Examples of encoded information in a 32-bit word.

locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. 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.

## 2.2.2 BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.7. 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, as described in Section 2.1.1. 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 and are the addresses used when specifying memory read and write operations for words.

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 2.6a. It is the



**Figure 2.7** Byte and word addressing.

most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is,  $b_7, b_6, \dots, b_0$ , from left to right. There are computers, however, that use the reverse ordering.

### 2.2.3 WORD ALIGNMENT

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8,  $\dots$ , as shown in Figure 2.7. We say that the word locations have *aligned* addresses. In general, words are said to be aligned in memory 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,  $\dots$ , and for a word length of 64 ( $2^3$  bytes), aligned words begin at byte addresses 0, 8, 16,  $\dots$ .

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

### 2.2.4 ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS

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

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. There are two ways to indicate the length of the string. A special control character with the meaning “end of string” can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

## 2.3 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, *Load* (or *Read* or *Fetch*) and *Store* (or *Write*).

The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load 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 Store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

An information item of either one word or one byte can be transferred between the processor and the memory in a single operation. As described in Chapter 1, the processor contains a small number of registers, each capable of holding a word. These registers are either the source or the destination of a transfer to or from the memory. When a byte is transferred, it is usually located in the low-order (rightmost) byte position of the register.

The details of the hardware implementation of these operations are treated in Chapters 5 and 7. In this chapter, we are taking the ISA viewpoint, so we concentrate on the logical handling of instructions and operands. Specific hardware components, such as processor registers, are discussed only to the extent necessary to understand the execution of machine instructions and programs.

## 2.4 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 the first two types of instructions. To facilitate the discussion, we need some notation which we present first.

### 2.4.1 REGISTER TRANSFER NOTATION

We need to describe the transfer of information from one location in the 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 a location by a symbolic name standing for its hardware binary address. For example,

names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor register names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

$$R1 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

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.

## 2.4.2 ASSEMBLY LANGUAGE NOTATION

We need another type of notation to represent machine instructions and programs. For this, we use an *assembly language* format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

Move LOC,R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

Add R1,R2,R3

## 2.4.3 BASIC INSTRUCTION TYPES

The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program is a command to 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. We will use the variable names to refer to the corresponding memory location addresses. 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.

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands — A, B, and C. This *three-address* instruction can be represented symbolically as

Add A,B,C

Operands A and B are called the *source* operands, C is called the *destination* operand, and Add is the operation to be performed on the operands. A general instruction of this type has the format

Operation Source1,Source2,Destination

If  $k$  bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain  $3k$  bits for addressing purposes in addition to the bits needed to denote the Add operation. For a modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word for a reasonable word length. Thus, a format that allows multiple words to be used for a single instruction would be needed to represent an instruction of this type.

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that *two-address* instructions of the form

Operation Source,Destination

are available. An Add instruction of this type is

Add A,B

which performs the operation  $B \leftarrow [A] + [B]$ . 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 operand B is both a source and a destination.

A single two-address instruction cannot be used to solve our original problem, which is to add the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move B,C

which performs the operation  $C \leftarrow [B]$ , leaving the contents of location B unchanged. The word “Move” is a misnomer here; it should be “Copy.” However, this instruction name is deeply entrenched in computer nomenclature. The operation  $C \leftarrow [A] + [B]$

can now be performed by the two-instruction sequence

```
Move  B,C
Add   A,C
```

In all the instructions given above, the source operands are specified first, followed by the destination. This order is used in the assembly language expressions for machine instructions in many computers. But there are also many computers in which the order of the source and destination operands is reversed. We will see examples of both orderings in Chapter 3. It is unfortunate that no single convention has been adopted by all manufacturers. In fact, even for a particular computer, its assembly language may use a different order for different instructions. In this chapter, we will continue to give the source operands first.

We have defined three- and two-address instructions. But, even two-address instructions will not normally fit into one word for usual word lengths and address sizes. Another possibility is to have machine instructions that specify only one memory operand. When a second operand is needed, as in the case of an Add instruction, it is understood implicitly to be in a unique location. A processor register, usually called the *accumulator*, may be used for this purpose. Thus, the *one-address* instruction

```
Add  A
```

means the following: Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator. Let us also introduce the one-address instructions

```
Load  A
```

and

```
Store A
```

The Load instruction copies the contents of memory location A into the accumulator, and the Store instruction copies the contents of the accumulator into memory location A. Using only one-address instructions, the operation  $C \leftarrow [A] + [B]$  can be performed by executing the sequence of instructions

```
Load  A
Add   B
Store C
```

Note that the operand specified in the instruction may be a source or a destination, depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers — typically 8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the

processor. Because the number of registers is relatively small, only a few bits are needed to specify which register takes part in an operation. For example, for 32 registers, only 5 bits are needed. This is much less than the number of bits needed to give the address of a location in the memory. Because the use of registers allows faster processing and results in shorter instructions, registers are used to store data temporarily in the processor during processing.

Let  $R_i$  represent a general-purpose register. The instructions

Load  $A, R_i$

Store  $R_i, A$

and

Add  $A, R_i$

are generalizations of the Load, Store, and Add instructions for the single-accumulator case, in which register  $R_i$  performs the function of the accumulator. Even in these cases, when only one memory address is directly specified in an instruction, the instruction may not fit into one word.

When a processor has several general-purpose registers, many instructions involve only operands that are in the registers. In fact, in many modern processors, computations can be performed directly only on data held in processor registers. Instructions such as

Add  $R_i, R_j$

or

Add  $R_i, R_j, R_k$

are of this type. In both of these instructions, the source operands are the contents of registers  $R_i$  and  $R_j$ . In the first instruction,  $R_j$  also serves as the destination register, whereas in the second instruction, a third register,  $R_k$ , is used as the destination. Such instructions, where only register names are contained in the instruction, will normally fit into one word.

It is often necessary to transfer data between different locations. This is achieved with the instruction

Move Source, Destination

which places a copy of the contents of Source into Destination. When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. Thus,

Move  $A, R_i$

is the same as

Load  $A, R_i$

and

Move  $R_i, A$

is the same as

```
Store Ri,A
```

In this chapter, we will use Move instead of Load or Store.

In processors where arithmetic operations are allowed only on operands that are in processor registers, the  $C = A + B$  task can be performed by the instruction sequence

```
Move A,Ri
Move B,Rj
Add Ri,Rj
Move Rj,C
```

In processors where one operand may be in the memory but the other must be in a register, an instruction sequence for the required task would be

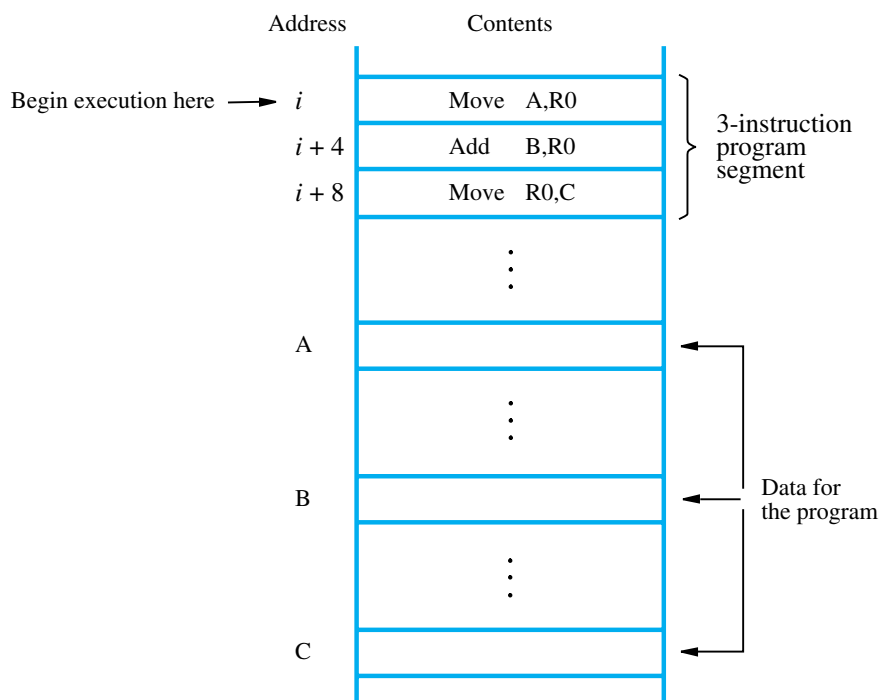
```
Move A,Ri
Add B,Ri
Move Ri,C
```

The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor. Hence, a substantial increase in speed is achieved when several operations are performed in succession on data in processor registers without the need to copy data to or from the memory. When machine language programs are generated by compilers from high-level languages, it is important to minimize the frequency with which data is moved back and forth between the memory and processor registers.

We have discussed three-, two-, and one-address instructions. It is also possible to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a *pushdown stack*. In this case, the instructions are called *zero-address* instructions. The concept of a pushdown stack is introduced in Section 2.8, and a computer that uses this approach is discussed in Chapter 11.

#### 2.4.4 INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

In the preceding discussion of instruction formats, we used the task  $C \leftarrow [A] + [B]$  for illustration. Figure 2.8 shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand per instruction and has a number of processor registers. We assume that the word length is 32 bits and the memory is byte addressable. The three instructions of the program are in successive word locations, starting at location  $i$ . Since each instruction is 4 bytes long, the second and third instructions start at addresses  $i + 4$  and  $i + 8$ . For simplicity, we also assume that a full memory address can be directly specified in a single-word instruction, although this is not usually possible for address space sizes and word lengths of current processors.



**Figure 2.8** A program for  $C \leftarrow [A] + [B]$ .

Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the instruction to be executed next. 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 Move instruction at location  $i + 8$  is executed, the PC contains the value  $i + 12$ , which is the address of the first instruction of the next program segment.

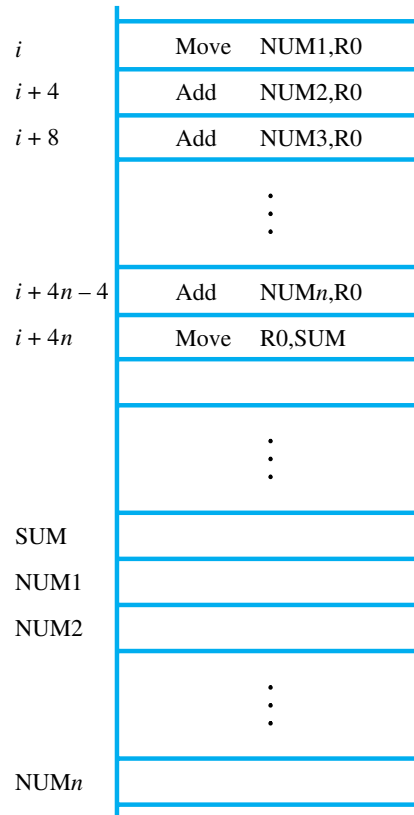
Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *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 often involves 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. In most processors, the

execute phase itself is divided into a small number of distinct phases corresponding to fetching operands, performing the operation, and storing the result.

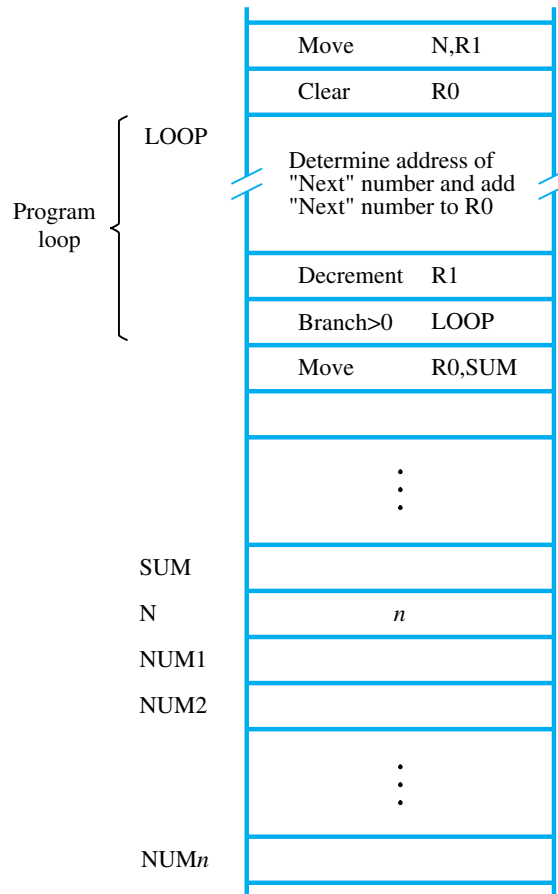
### 2.4.5 BRANCHING

Consider the task of adding a list of  $n$  numbers. The program outlined in Figure 2.9 is a generalization of the program in Figure 2.8. The addresses of the memory locations containing the  $n$  numbers are symbolically given as NUM1, NUM2, ..., NUM $n$ , and a separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Add instructions, it is possible to place a single Add instruction in a program loop, as shown in Figure 2.10. The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch>0. During each pass through this loop, the address of



**Figure 2.9** A straight-line program for adding  $n$  numbers.



**Figure 2.10** Using a loop to add  $n$  numbers.

the next list entry is determined, and that entry is fetched and added to R0. The address of an operand can be specified in various ways, as will be described in Section 2.5. 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 R1 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 R1 at the beginning of the program. Then, within the body of the loop, the instruction

Decrement R1

reduces the contents of R1 by 1 each time through the loop. (A similar type of operation is performed by an Increment instruction, which adds 1 to its operand.) Execution of the loop is repeated as long as the result of the decrement operation is greater than zero.

We now introduce *branch* instructions. This type of instruction loads a new value 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 2.10, the instruction

Branch>0 LOOP

(branch if greater than 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is 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 R0. At the end of the  $n$ th pass through the loop, the Decrement instruction produces a value of zero, and, hence, branching does not occur. Instead, the Move instruction is fetched and executed. It moves the final result from R0 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.

### 2.4.6 CONDITION CODES

The processor keeps track of information about the results of various operations 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 indicate whether the result of an arithmetic or logic operation is negative or zero. The N and Z flags may also be affected by instructions that transfer data, such as Move, Load, or Store. 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 and sets or clears the N and Z flags accordingly.

The V flag indicates whether overflow has taken place. As explained in Section 2.1.4, 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 corrects the problem. Instructions such as BranchIfOverflow are provided for this purpose. Also, as we will see in Chapter 4, a program interrupt may occur automatically as a result of the V bit being set, and the operating system will resolve what to do.

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 6.

The instruction Branch>0, discussed in Section 2.4.5, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. Many other conditional branch instructions are provided to enable a variety of conditions to be tested. The conditions are given as logic expressions involving the condition code flags.

In some computers, the condition code flags are affected automatically by instructions that perform arithmetic or logic operations. However, this is not always the case. A number of computers have two versions of an Add instruction, for example. One version, Add, does not affect the flags, but a second version, AddSetCC, does. This provides the programmer — and the compiler — with more flexibility when preparing programs for pipelined execution, as we will discuss in Chapter 8.

## 2.4.7 GENERATING MEMORY ADDRESSES

Let us return to Figure 2.10. The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in that block must refer to a different address during each pass. How are the addresses to be specified? The memory operand address cannot be given directly in a single Add 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,  $R_i$ , 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.

## 2.5 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. If we want to keep track of students' names, we can write them in a list. If we want to associate information with each name, for example to record telephone numbers or marks in various courses, we may organize this information in the form of a table. Programmers use organizations called *data structures* to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run. The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*. In this section we present the most important addressing modes found in modern processors. A summary is provided in Table 2.1.

**Table 2.1** Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	$R_i$	$EA = R_i$
Absolute (Direct)	LOC	$EA = LOC$
Indirect	$(R_i)$	$EA = [R_i]$
	(LOC)	$EA = [LOC]$
Index	$X(R_i)$	$EA = [R_i] + X$
Base with index	$(R_i, R_j)$	$EA = [R_i] + [R_j]$
Base with index and offset	$X(R_i, R_j)$	$EA = [R_i] + [R_j] + X$
Relative	$X(PC)$	$EA = [PC] + X$
Autoincrement	$(R_i)+$	$EA = [R_i];$ Increment $R_i$
Autodecrement	$-(R_i)$	Decrement $R_i;$ $EA = [R_i]$

EA = effective address  
Value = a signed number

### 2.5.1 IMPLEMENTATION OF VARIABLES AND CONSTANTS

Variables and constants are the simplest data types and 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. Thus, the value can be changed as needed using appropriate instructions.

The programs in Section 2.4 used only two addressing modes to access variables. We accessed 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 (address) 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. (In some assembly languages, this mode is called *Direct*.)

The instruction

```
Move LOC,R2
```

uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

```
Integer A, B;
```

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

Next, let us consider the representation of constants. Address and data constants can be represented in assembly language using the Immediate mode.

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

For example, the instruction

```
Move 200immediate, R0
```

places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp 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

```
Move #200,R0
```

Constant values are used frequently in high-level language programs. For example, the statement

```
A = B + 6
```

contains the constant 6. Assuming that A and B have been declared earlier as variables and may be accessed using the Absolute mode, this statement may be compiled as follows:

```

Move  B,R1
Add   #6,R1
Move  R1,A
    
```

Constants are also used in assembly language to increment a counter, test for some bit pattern, and so on.

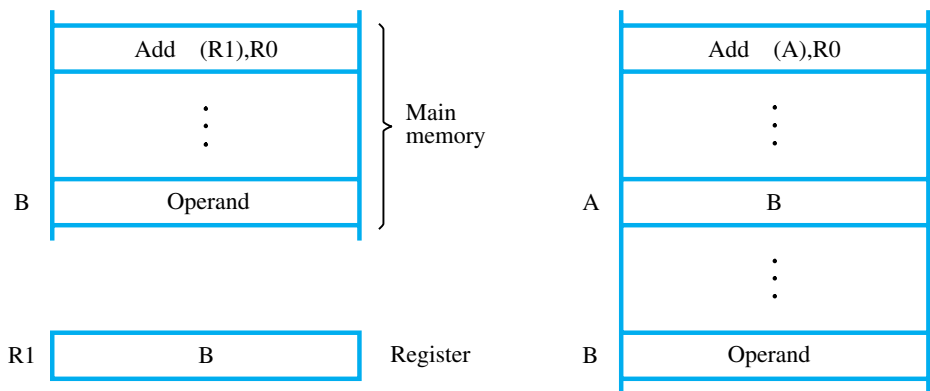
### 2.5.2 INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the *effective address (EA)* of the operand.

*Indirect mode* — The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

We denote indirection by placing the name of the register or the memory address given in the instruction in parentheses as illustrated in Figure 2.11 and Table 2.1.

To execute the Add instruction in Figure 2.11a, the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in Figure 2.11b. In this case, the processor first reads the contents of memory location A, then requests a



(a) Through a general-purpose register

(b) Through a memory location

Figure 2.11 Indirect addressing.

Address	Contents	
	Move	N,R1
	Move	#NUM1,R2
	Clear	R0
	Add	(R2),R0
	Add	#4,R2
	Decrement	R1
	Branch>0	LOOP
	Move	R0,SUM

} Initialization

→ LOOP

└──┬──┘

**Figure 2.12** Use of indirect addressing in the program of Figure 2.10.

second read operation using the value B as an address to obtain the operand.

The register or memory location that contains the address of an operand is called a *pointer*. Indirection and the use of pointers are important and powerful concepts in programming. Consider the analogy of a treasure hunt: In the instructions for the hunt you may be told to go to a house at a given address. Instead of finding the treasure there, you find a note that gives you another address where you will find the treasure. By changing the note, the location of the treasure can be changed, but the instructions for the hunt remain the same. Changing the note is equivalent to changing the contents of a pointer in a computer program. For example, by changing the contents of register R1 or location A in Figure 2.11, the same Add instruction fetches different operands to add to register R0.

Let us now return to the program in Figure 2.10 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 2.12. Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value  $n$  from memory location N into R1 and uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop in Figure 2.12 implement the unspecified instruction block starting at LOOP in Figure 2.10. The first time through the loop, the instruction

Add (R2),R0

fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Consider the C-language statement

A = \*B;

where B is a pointer variable. This statement may be compiled into

Move B,R1  
Move (R1),A

Using indirect addressing through memory, the same action can be achieved with

Move (B),A

Despite its apparent simplicity, indirect addressing through memory has proven to be of limited usefulness as an addressing mode, and it is seldom found in modern computers. We will see in Chapter 8 that an instruction that involves accessing the memory twice to get an operand is not well suited to pipelined execution.

Indirect addressing through registers is used extensively. The program in Figure 2.12 shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

### 2.5.3 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.

The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*. We indicate the Index mode symbolically as

$X(R_i)$

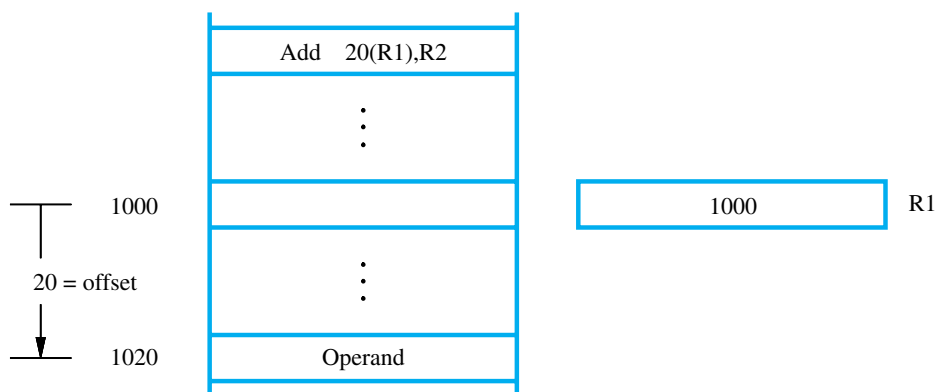
where  $X$  denotes the constant value contained in the instruction and  $R_i$  is the name of the register involved. The effective address of the operand is given by

$$EA = X + [R_i]$$

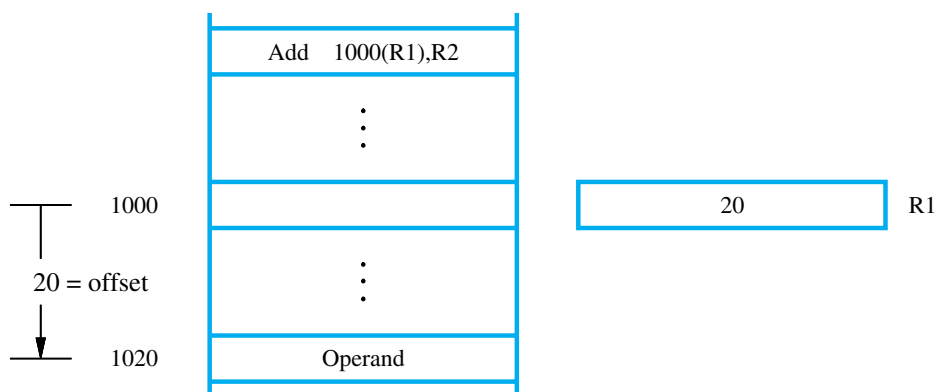
The contents of the index register are not changed in the process of generating the effective address.

In an assembly language program, the constant  $X$  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 2.6. When the instruction is translated into machine code, the constant  $X$  is given as a part of the instruction and is usually represented by fewer bits than the word length of the computer. Since  $X$  is a signed integer, it must be sign-extended (see Section 2.1.3) to the register length before being added to the contents of the register.

Figure 2.13 illustrates two ways of using the Index mode. In Figure 2.13a, the index register,  $R_1$ , 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 2.13b. 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 stored in a register.



(a) Offset is given as a constant

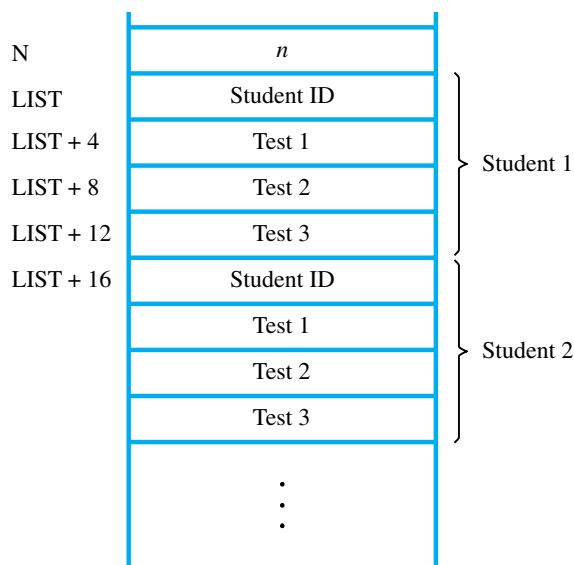


(b) Offset is in the index register

Figure 2.13 Indexed addressing.

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 2.14. 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 2.14 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.



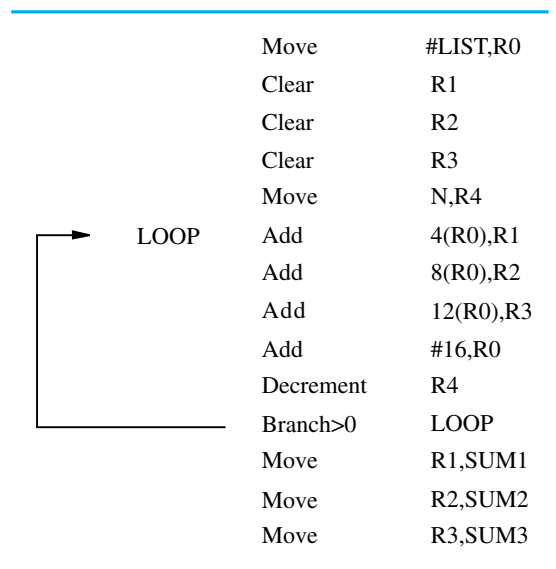
**Figure 2.14** A list of students' marks.

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 program for this task is given in Figure 2.15. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.13a to access each of the three scores in a student's record. Register R0 is used as the index register. Before the loop is entered, R0 is set to point to the ID location of the first student record; thus, it contains 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 R1, R2, and R3, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R0), 8(R0), and 12(R0). The index register R0 is then incremented by 16 to point to the ID location of the second student. Register R4, initialized to contain the value  $n$ , is decremented by 1 at the end of each pass through the loop. When the contents of R4 reach 0, all student records have been accessed, and 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 R1, R2, and R3, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R0, are not changed when it is used in the Index addressing mode to access the scores. The contents of R0 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.



**Figure 2.15** Indexed addressing used in accessing test scores in the list in Figure 2.14.

We have introduced the most basic form of indexed addressing. Several variations of this basic form provide for very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset  $X$ , in which case we can write the Index mode as

$$(R_i, R_j)$$

The effective address is the sum of the contents of registers  $R_i$  and  $R_j$ . 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.

As an example of where this flexibility may be useful, consider again the student record data structure shown in Figure 2.14. In the program in Figure 2.15, we used different index values in the three Add instructions at the beginning of the loop to access different test scores. Suppose each record contains a large number of items, many more than the three test scores of that example. In this case, we would need the ability to replace the three Add instructions with one instruction inside a second (nested) loop. Just as the successive starting locations of the records (the reference points) are maintained in the pointer register  $R_0$ , offsets to the individual items relative to the contents of  $R_0$  could be maintained in another register. The contents of that register would be incremented in successive passes through the inner loop. (See Problem 2.9.)

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

$$X(R_i, R_j)$$

In this case, the effective address is the sum of the constant  $X$  and the contents of registers  $R_i$  and  $R_j$ . 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  $(R_i, R_j)$  part of the addressing mode. In other words, this mode implements a three-dimensional array.

### 2.5.4 RELATIVE ADDRESSING

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if 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  $R_i$ .

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

Branch>0 LOOP

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Recall that during the execution of an instruction, the processor increments the PC to point to the next instruction. Most computers use this updated value in computing the effective address in the Relative mode. For example, suppose that the Relative mode is used to generate the branch target address LOOP in the Branch instruction of the program in Figure 2.12. Assume that the four instructions of the loop body, starting at LOOP, are located at memory locations 1000, 1004, 1008, and 1012. Hence, the updated contents of the PC at the time the branch target address is generated will be 1016. To branch to location LOOP (1000), the offset value needed is  $X = -16$ .

Assembly languages allow branch instructions to be written using labels to denote the branch target as shown in Figure 2.12. When the assembler program processes such an instruction, it computes the required offset value,  $-16$  in this case, and generates the corresponding machine instruction using the addressing mode  $-16(PC)$ .

### 2.5.5 ADDITIONAL MODES

So far we have discussed the five basic addressing modes — Immediate, Register, Absolute (Direct), Indirect, and Index — found in most computers. We have given a number of common versions of the Index mode, not all of which may be found in any one computer. Although these modes suffice for general computation, many computers

provide additional modes intended to aid certain programming tasks. The two modes described next are useful for accessing data items in successive locations in the memory.

*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 item in a list.

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

$$(Ri)+$$

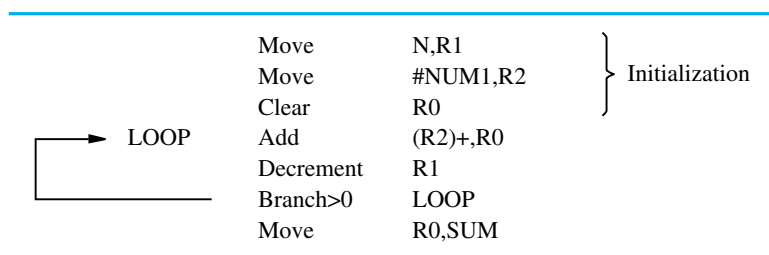
Implicitly, the increment amount is 1 when the mode is given in this form. But in a byte addressable memory, this mode would only be useful in accessing successive bytes of some list. To access successive words in a byte-addressable memory with a 32-bit word length, the increment must be 4. Computers 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  $(Ri)+$ .

If the Autoincrement mode is available, it can be used in the first Add instruction in Figure 2.12 and the second Add instruction can be eliminated. The modified program is shown in Figure 2.16.

As a companion for the Autoincrement mode, another useful mode accesses the items of a list 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

$$-(Ri)$$


**Figure 2.16** The Autoincrement addressing mode used in the program of Figure 2.12.

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 given in Section 2.8, where we show how these two modes can be used together to implement an important data structure called a stack.

The actions performed by the Autoincrement and Autodecrement addressing modes can obviously be achieved by using two instructions, one to access the operand and the other to increment or decrement the register that contains the operand address. Combining the two operations in one instruction reduces the number of instructions needed to perform the desired task. However, we will show in Chapter 8 that it is not always advantageous to combine two operations in a single instruction.

## 2.6 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 Move, Add, Increment, 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 MOV, ADD, INC, and BR. Similarly, we use the notation R3 to refer to register 3, and LOC to refer to a memory location. A complete set of such symbolic names and rules for their use constitute a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics in the 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. 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 2.6.2. First, we present a few aspects of the 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. We will use capital letters to denote all names and labels in our examples in order to improve the readability of the text. For example, we will write a Move instruction as

```
MOVE R0,SUM
```

The mnemonic `MOVE` represents the binary pattern, or *OP code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer understands.

The OP-code mnemonic is followed by at least one blank space character. Then the information that specifies the operands is given. In our example, the source operand is in register `R0`. This information is followed by the specification of the destination operand, separated from the source operand by a comma, with no intervening blanks. 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, the assembly language 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 sharp sign usually denotes an immediate operand. Thus, the instruction

```
ADD #5,R3
```

adds the number 5 to the contents of register `R3` and puts the result back into register `R3`. The sharp sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the intended addressing mode is indicated in the OP-code mnemonic. In this case, a given instruction has different OP-code mnemonics for different addressing modes. For example, the previous Add instruction may be written as

```
ADDI 5,R3
```

The suffix `I` in the mnemonic `ADDI` states that the 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 the number 5 is to be placed in a memory location whose address is held in register `R2`, the desired action can be specified as

```
MOVE #5,(R2)
```

or perhaps

```
MOVEI 5,(R2)
```

### 2.6.1 ASSEMBLER DIRECTIVES

In addition to providing a mechanism for representing instructions in a program, the 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 `SUM` is used to represent the value 200. This fact may be conveyed to the assembler program through a statement such as

```
SUM EQU 200
```

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 SUM should be replaced by the value 200 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 2.12. In order to run this program on a computer, it is necessary to write its source code in the required assembly language, specifying all the information needed to generate the corresponding object program. Suppose that each instruction and each data item occupies one word of memory. This is an oversimplification, but it helps keep the example straightforward. 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 2.17. The figure shows the memory addresses where

	100	Move	N,R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2),R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
			⋮
SUM	200		
N	204		100
NUM1	208		
NUM2	212		
			⋮
NUM <sub>n</sub>	604		

**Figure 2.17** Memory arrangement for the program in Figure 2.12.

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 2.18. The program begins with assembler directives. We have already discussed the `Equate` directive, `EQU`, which informs the assembler about the value of `SUM`. The second assembler directive, `ORIGIN`, tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 204. Since this location is to be loaded with the value 100 (which is the number of entries in the list), a `DATAWORD` directive is used to inform the assembler of this requirement. It states that the data value 100 is to be placed in the memory word at address 204.

Any statement that results in instructions or data being placed in a memory location may be given a memory address label. The label is assigned a value equal to the address

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
Assembler directives		MOVE	R0,SUM
		RETURN	
		END	START

**Figure 2.18** Assembly language representation for the program in Figure 2.17.

of that location. Because the DATAWORD statement is given the label N, the name N is assigned the value 204. Whenever N is encountered in the rest of the program, it will be replaced with this value. Using N as a label in this manner is equivalent to using the assembler directive

```
N EQU 204
```

The RESERVE directive declares that a memory block of 400 bytes is to be reserved for data, and that the name NUM1 is to be associated with address 208. 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 later in this chapter.

The second ORIGIN directive specifies that the instructions 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. 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. The END directive includes the label START, which is the address of the location at which execution of the program is to begin.

We have explained all statements in Figure 2.18 except RETURN. This is an assembler directive that identifies the point at which execution of the program should be terminated. It causes the assembler to insert an appropriate machine instruction that returns control to the operating system of the computer.

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

```
Label Operation Operand(s) Comment
```

These four *fields* are separated by an appropriate delimiter, typically one or more blank 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 2.18 there are five labels: SUM, N, NUM1, START, and LOOP.

The Operation field contains the OP-code mnemonic of the desired instruction or assembler directive. The Operand field contains addressing information for accessing one or more operands, depending on the type of instruction. 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.

## 2.6.2 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 address given in the `ORIGIN` assembler directives. It also inserts constants that may be given in `DATAWORD` commands and 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 `START` and `LOOP` will be assigned the values 100 and 112, 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 using the Relative addressing mode, as explained in Section 2.5. The assembler computes the *branch offset*, which is the distance to the target, and puts it into the machine instruction.

As the assembler scans through a source program, it keeps track of all names and the numerical values that correspond to them in a *symbol table*. Thus, when a name appears a second time, it is replaced with its value from the table. A problem arises when a name appears as an operand before it is given a value. For example, this happens if a forward branch is required. The assembler will not be able to determine the branch target, because the name referred to has not yet been recorded in the symbol table. A simple solution to this problem is to have the assembler scan through the source program twice. During the first pass, it creates a complete symbol table. At the end of this pass, all names will have been assigned numerical values. The assembler then goes through the source program a second time and substitutes values for all names from the symbol table. Such an assembler is called a *two-pass assembler*.

The assembler stores the object program on a magnetic disk. The object program must be loaded into the memory of the computer 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. Recall that the address of this instruction has been included in the assembly language program as the operand of the `END` assembler directive. The assembler includes this address in the header that precedes the object code on the disk.

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 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. We consider program debugging in more detail in Chapter 4.

### 2.6.3 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

```
ADD #93,R1
```

or as a binary number identified by a prefix symbol such as a percent sign, as in

```
ADD #%01011101,R1
```

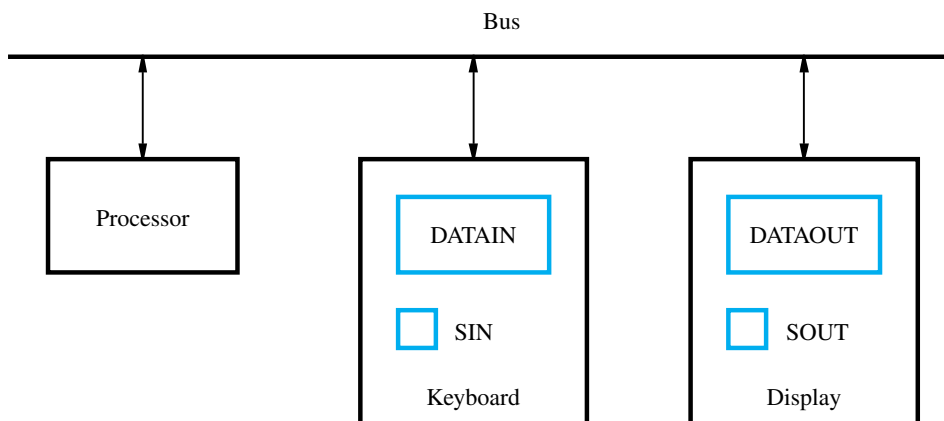
Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The hex notation is a direct extension of the BCD code given in Appendix E. The first ten patterns 0000, 0001, . . . , 1001, are represented by the digits 0, 1, . . . , 9, as in BCD. 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 a dollar sign prefix. Thus, we would write

```
ADD #$5D,R1
```

## 2.7 BASIC INPUT/OUTPUT OPERATIONS

Previous sections in this chapter described machine instructions and addressing modes. We have assumed that the data on which these instructions operate are already stored in the memory. We now examine the means by which data are transferred between the memory of a computer and the outside world. Input/Output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer. This subject is discussed in detail in Chapter 4. Here, we introduce a few basic ideas.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as *program-controlled I/O*. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can



**Figure 2.19** Bus connection for processor, keyboard, and display.

execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown in Figure 2.19. The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Consider the problem of moving a character code from the keyboard to the processor. Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register `DATAIN`, as shown in Figure 2.19. To inform the processor that a valid character is in `DATAIN`, a status control flag, `SIN`, is set to 1. A program monitors `SIN`, and when `SIN` is set to 1, the processor reads the contents of `DATAIN`. When the character is transferred to the processor, `SIN` is automatically cleared to 0. If a second character is entered at the keyboard, `SIN` is again set to 1 and the process repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, `DATAOUT`, and a status control flag, `SOUT`, are used for this transfer. When `SOUT` equals 1, the display is ready to receive a character. Under program control, the processor monitors `SOUT`, and when `SOUT` is set to 1, the processor transfers a character code to `DATAOUT`. The transfer of a character to `DATAOUT` clears `SOUT` to 0; when the display device is ready to receive a second character, `SOUT` is again set to 1. The buffer registers `DATAIN` and `DATAOUT` and the status flags `SIN`

and SOUT are part of circuitry commonly known as a *device interface*. The circuitry for each device is connected to the processor via a bus, as indicated in Figure 2.19.

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations:

```
READWAIT  Branch to READWAIT if SIN = 0
           Input from DATAIN to R1
```

The Branch operation is usually implemented by two machine instructions. The first instruction tests the status flag and the second performs the branch. Although the details vary from computer to computer, the main idea is that the processor monitors the status flag by executing a short *wait loop* and proceeds to transfer the input data when SIN is set to 1 as a result of a key being struck. The Input operation resets SIN to 0.

An analogous sequence of operations is used for transferring output to the display. An example is

```
WRITEWAIT Branch to WRITEWAIT if SOUT = 0
           Output from R1 to DATAOUT
```

Again, the Branch operation is normally implemented by two machine instructions. The wait loop is executed repeatedly until the status flag SOUT is set to 1 by the display when it is free to receive a character. The Output operation transfers a character from R1 to DATAOUT to be displayed, and it clears SOUT to 0.

We assume that the initial state of SIN is 0 and the initial state of SOUT is 1. This initialization is normally performed by the device control circuits when the devices are placed under computer control before program execution begins.

Until now, we have assumed that the addresses issued by the processor to access instructions and operands always refer to memory locations. Many computers use an arrangement called *memory-mapped I/O* in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT. Thus, no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the processor using instructions that we have already discussed, such as Move, Load, or Store. For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

```
MoveByte  DATAIN,R1
```

Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction

```
MoveByte  R1,DATAOUT
```

The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively. The MoveByte operation code signifies that the operand size is a byte, to distinguish it from the operation code

Move that has been used for word operands. We have established that the two data buffers in Figure 2.19 may be addressed as if they were two memory locations. It is possible to deal with the status flags SIN and SOUT in the same way, by assigning them distinct addresses. However, it is more common to include SIN and SOUT in *device status* registers, one for each of the two devices. Let us assume that bit  $b_3$  in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively. The read operation just described may now be implemented by the machine instruction sequence

```

READWAIT  Testbit    #3,INSTATUS
          Branch=0  READWAIT
          MoveByte  DATAIN,R1

```

The write operation may be implemented as

```

WRITEWAIT Testbit    #3,OUTSTATUS
          Branch=0  WRITEWAIT
          MoveByte  R1,DATAOUT

```

The Testbit instruction tests the state of one bit in the destination location, where the bit position to be tested is indicated by the first operand. If the bit tested is equal to 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is ready, that is, when the bit tested becomes equal to 1, the data are read from the input buffer or written into the output buffer.

The program shown in Figure 2.20 uses these two operations to read a line of characters typed at a keyboard and send them out to a display device. As the characters are read in, one by one, they are stored in a data area in the memory and then echoed

---

	Move	#LOC,R0	Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered in the keyboard buffer DATAIN.
	Branch=0	READ	
	MoveByte	DATAIN,(R0)	Transfer the character from DATAIN into the memory (this clears SIN to 0).
ECHO	TestBit	#3,OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(R0)+	Check if the character just read is CR (carriage return). If it is not CR, then branch back and read another character.
	Branch≠0	READ	Also, increment the pointer to store the next character.

---

**Figure 2.20** A program that reads a line of characters and displays it.

back out to the display. The program finishes when the carriage return character, CR, is read, stored, and sent to the display. The address of the first byte location of the memory data area where the line is to be stored is LOC. Register R0 is used to point to this area, and it is initially loaded with the address LOC by the first instruction in the program. R0 is incremented for each character read and displayed by the Autoincrement addressing mode used in the Compare instruction.

Program-controlled I/O requires continuous involvement of the processor in the I/O activities. Almost all of the execution time for the program in Figure 2.20 is accounted for in the two wait loops, while the processor waits for a character to be struck or for the display to become available. It is desirable to avoid wasting processor execution time in this situation. Other I/O techniques, based on the use of interrupts, may be used to improve the utilization of the processor. Such techniques will be discussed in Chapter 4.

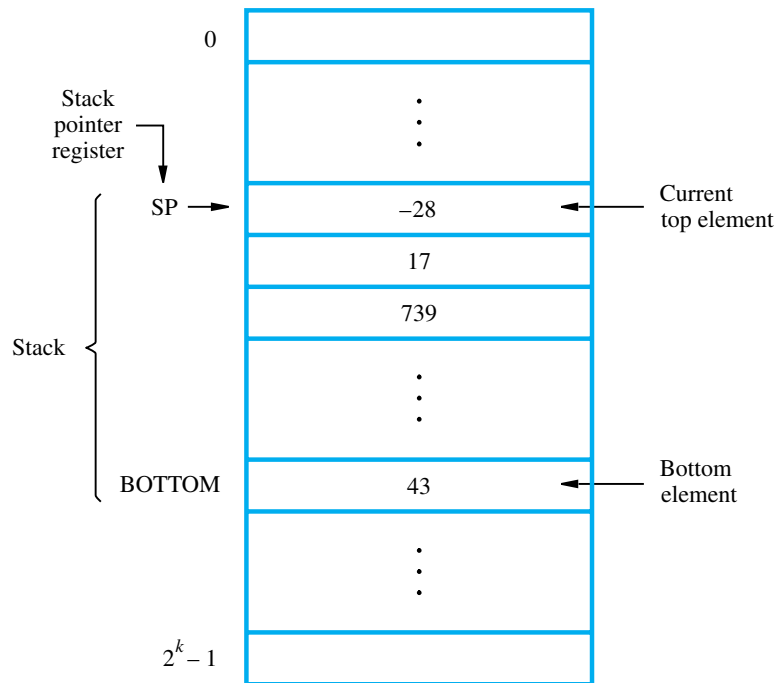
## 2.8 STACKS AND QUEUES

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. This section will describe stacks, as well as a closely related data structure called a queue.

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 or bytes, 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.

Data stored in the memory of a computer can be organized as 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 2.21 shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and  $-28$  at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the *stack pointer* (SP). It could be one of the general-purpose registers or a register dedicated to this function. If we assume a



**Figure 2.21** A stack of words in the memory.

byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract #4,SP
Move    NEWITEM,(SP)
```

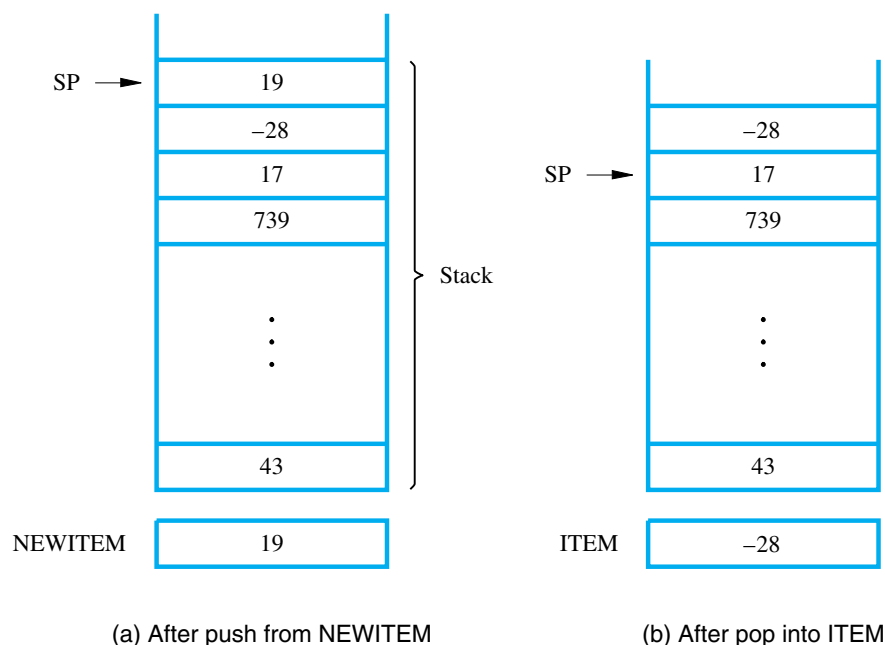
where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move. The pop operation can be implemented as

```
Move (SP),ITEM
Add #4,SP
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.22 shows the effect of each of these operations on the stack in Figure 2.21.

If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction

```
Move NEWITEM,—(SP)
```



**Figure 2.22** Effect of stack operations on the stack in Figure 2.21.

and the pop operation can be performed by

Move (SP)+,ITEM

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we must avoid pushing an item onto the stack when the stack has reached its maximum size. Also, we must avoid attempting to pop an item off an empty stack, which could result from a programming error. Suppose that a stack runs from location 2000 (BOTTOM) down no further than location 1500. The stack pointer is loaded initially with the address value 2004. Recall that SP is decremented by 4 before new data are stored on the stack. Hence, an initial value of 2004 means that the first item pushed onto the stack will be at location 2000. To prevent either pushing an item on a full stack or popping an item off an empty stack, the single-instruction push and pop operations can be replaced by the instruction sequences shown in Figure 2.23.

The Compare instruction

Compare src,dst

performs the operation

$[dst] - [src]$

and sets the condition code flags according to the result. It does not change the value of either operand.

---

SAFEPOP	Compare	#2000,SP	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Branch>0	EMPTYERROR	
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

---

(a) Routine for a safe pop operation

---

SAFEPOP	Compare	#1500,SP	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Branch≤0	FULLERROR	
	Move	NEWITEM,−(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

---

(b) Routine for a safe push operation

**Figure 2.23** Checking for empty and full errors in pop and push operations.

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.

Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a *circular buffer*. Let us assume that memory addresses from BEGINNING to END are assigned to the queue. The first entry in the queue is entered into location

BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues. As in the case of a stack, care must be taken to detect when the region assigned to the data structure is either completely full or completely empty (see Problems 2.18 and 2.19).

## 2.9 SUBROUTINES

In a given program, it is often necessary to perform a particular subtask many times on different data values. Such a subtask is usually called a *subroutine*. For example, a subroutine may evaluate the *sine* function or sort a list of values into increasing or decreasing order.

It is possible to include 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 the instructions that constitute the subroutine 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 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 program resumes execution is the location pointed to by the updated 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 instruction

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

- Branch to the address contained in the link register

Figure 2.24 illustrates this procedure.

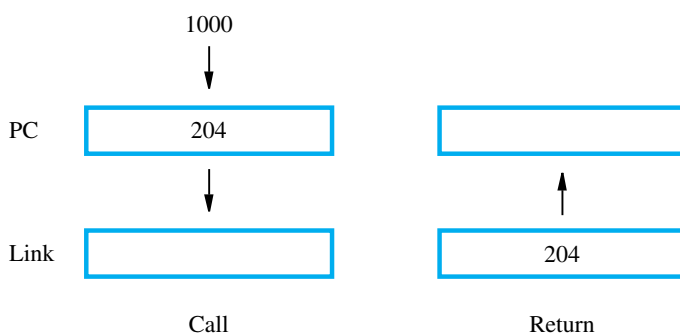
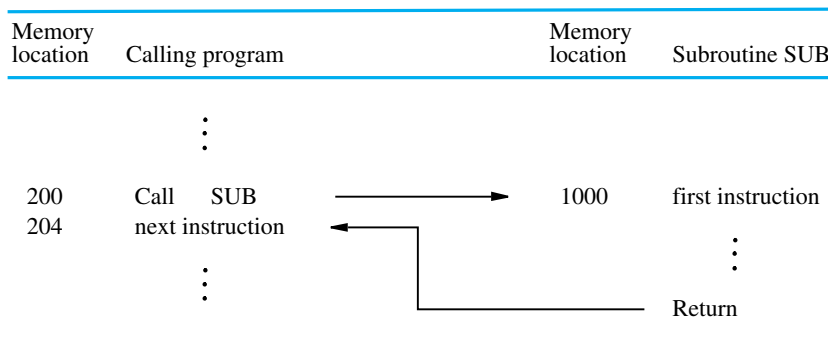


Figure 2.24 Subroutine linkage using a link register.

### 2.9.1 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, destroying 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 a stack. Many processors do this automatically as one of the operations performed by the Call instruction. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the *processor stack*. The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

## 2.9.2 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, in this case, 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 used for saving the return address.

Passing parameters through processor registers is straightforward and efficient. Figure 2.25 shows how the program in Figure 2.16 for adding a list of numbers can be implemented as a subroutine, 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  $R1$  and  $R2$ . The sum computed by the subroutine is passed back to the calling program through register  $R0$ . The first four instructions in Figure 2.25 constitute the relevant part of the calling program. The first two instructions load  $n$  and  $NUM1$  into  $R1$  and  $R2$ . The Call instruction branches to the subroutine starting at location  $LISTADD$ . This instruction also pushes the return address onto the processor stack. The subroutine computes the sum and places it in  $R0$ . After the return operation is performed by the subroutine, the sum is stored in memory location  $SUM$  by the calling program.

---

### Calling program

Move	$N,R1$	$R1$ serves as a counter.
Move	$\#NUM1,R2$	$R2$ points to the list.
Call	$LISTADD$	Call subroutine.
Move	$R0,SUM$	Save result.
:		

### Subroutine

$LISTADD$	Clear	$R0$	Initialize sum to 0.
$LOOP$	Add	$(R2)+,R0$	Add entry from list.
	Decrement	$R1$	
	Branch>0	$LOOP$	
	Return		Return to calling program.

---

**Figure 2.25** Program of Figure 2.16 written as a subroutine; parameters passed through registers.

If many parameters are involved, there may not be enough general-purpose registers available for passing them to the subroutine. Using a stack, on the other hand, is highly flexible; a stack can handle a large number of parameters. The following example illustrates this approach. Figure 2.26a shows the program of Figure 2.16 rewritten as a subroutine, LISTADD, which can be called by any other program to add a list of numbers. The parameters passed to this subroutine are the address of the first number in the list and the number of entries. The subroutine performs the addition and returns the computed sum. The parameters are pushed onto the processor stack pointed to by register SP. Assume that before the subroutine is called, the top of the stack is at level 1 in Figure 2.26b. The calling program pushes the address NUM1 and the value  $n$  onto the stack and calls subroutine LISTADD. The Call instruction also pushes the return address onto the stack. The top of the stack is now at level 2.

The subroutine uses three registers. Since these registers may contain valid data that belong to the calling program, their contents should be saved by pushing them onto the stack. We have used a single instruction, MoveMultiple, to store the contents of registers R0 through R2 on the stack. Many processors have such instructions. The top of the stack is now at level 3. The subroutine accesses the parameters  $n$  and NUM1 from the stack using indexed addressing. 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 R1 as the initial value of the count, and the address NUM1 is loaded into R2, which is used as a pointer to scan the list entries. At the end of the computation, register R0 contains the sum. Before the subroutine returns to the calling program, the contents of R0 are placed on the stack, replacing the parameter NUM1, which is no longer needed. Then the contents of the three registers used by the subroutine are restored from the stack. Now the top item on the stack is the return address 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.

### Parameter Passing by Value and by Reference

Note the nature of the two parameters, NUM1 and  $n$ , passed to the subroutines in Figures 2.25 and 2.26. 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.

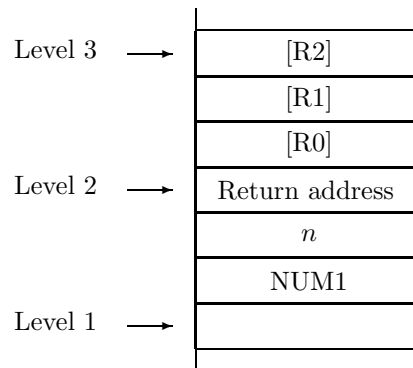
### 2.9.3 THE STACK FRAME

Now, observe how space is used in the stack in the example in Figure 2.26. 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, created at the time the subroutine is entered and freed up 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, they can also be allocated on the stack.

Assume top of stack is at level 1 below.

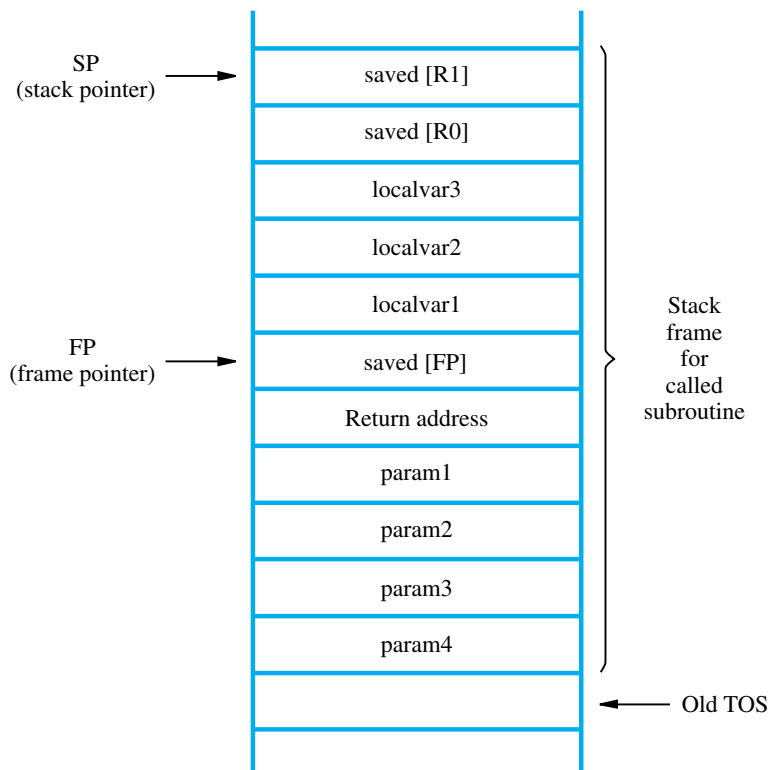
	Move	#NUM1,-(SP)	Push parameters onto stack.
	Move	N,-(SP)	
	Call	LISTADD	Call subroutine (top of stack at level 2).
	Move	4(SP),SUM	Save result.
	Add	#8,SP	Restore top of stack (top of stack at level 1).
	:		
LISTADD	MoveMultiple	R0-R2,-(SP)	Save registers (top of stack at level 3).
	Move	16(SP),R1	Initialize counter to <i>n</i> .
	Move	20(SP),R2	Initialize pointer to the list.
	Clear	R0	Initialize sum to 0.
LOOP	Add	(R2)+,R0	Add entry from list.
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,20(SP)	Put result on the stack.
	MoveMultiple	(SP)+,R0-R2	Restore registers.
	Return		Return to calling program.

(a) Calling program and subroutine



(b) Top of stack at various times

**Figure 2.26** Program of Figure 2.16 written as a subroutine; parameters passed on the stack.



**Figure 2.27** A subroutine stack frame example.

Figure 2.27 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. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with 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 *R0* and *R1* need to be saved because they will also be used within the subroutine.

With the *FP* register pointing to the location just above the stored return address, as shown in Figure 2.27, we can easily access the parameters and the local variables by using the Index addressing mode. The parameters can be accessed by using addresses  $8(\text{FP})$ ,  $12(\text{FP})$ ,  $\dots$ . The local variables can be accessed by using addresses  $-4(\text{FP})$ ,  $-8(\text{FP})$ ,  $\dots$ . 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 built, used, and dismantled for a particular invocation of the subroutine. We begin by

assuming that SP points to the old top-of-stack (TOS) element in Figure 2.27. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The Call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP.

Thus, the first two instructions executed in the subroutine are

```
Move  FP, -(SP)
Move  SP, 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 #12, SP
```

Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the figure.

The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction

```
Add #12, SP
```

and pops the saved old value of FP back into FP. At this point, SP points to the return address, so the Return instruction can be executed, transferring control back to the calling program.

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

### Stack Frames for Nested Subroutines

The stack is the proper data structure for holding return addresses when subroutines are nested. It should be clear that the complete stack frames for nested subroutines build up on the processor stack as they are called. In this regard, note that the saved contents of FP in the current frame at the top of the stack are the frame pointer contents for the stack frame of the subroutine that called the current subroutine.

An example of a main program calling a first subroutine SUB1, which then calls a second subroutine SUB2, is shown in Figure 2.28. The stack frames corresponding to these two nested subroutines are shown in Figure 2.29. All parameters involved in this example are passed on the stack. The figure only shows the flow of control and data among the three programs. 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 answer and passing it back to the main program on the stack. During the course of its computations, SUB1 calls the second subroutine,

Memory location		Instructions	Comments
<b>Main program</b>			
		:	
2000		Move PARAM2,-(SP)	Place parameters on stack.
2004		Move PARAM1,-(SP)	
2008		Call SUB1	
2012		Move (SP),RESULT	Store result.
2016		Add #8,SP	Restore stack level.
2020		next instruction	
		:	
<b>First subroutine</b>			
2100	SUB1	Move FP,-(SP)	Save frame pointer register.
2104		Move SP,FP	Load the frame pointer.
2108		MoveMultiple R0-R3,-(SP)	Save registers.
2112		Move 8(FP),R0	Get first parameter.
		Move 12(FP),R1	Get second parameter.
		:	
		Move PARAM3,-(SP)	Place a parameter on stack.
2160		Call SUB2	
2164		Move (SP)+,R2	Pop SUB2 result into R2.
		:	
		Move R3,8(FP)	Place answer on stack.
		MoveMultiple (SP)+,R0-R3	Restore registers.
		Move (SP)+,FP	Restore frame pointer register.
		Return	Return to Main program.
<b>Second subroutine</b>			
3000	SUB2	Move FP,-(SP)	Save frame pointer register.
		Move SP,FP	Load the frame pointer.
		MoveMultiple R0-R1,-(SP)	Save registers R0 and R1.
		Move 8(FP),R0	Get the parameter.
		:	
		Move R1,8(FP)	Place SUB2 result on stack.
		MoveMultiple (SP)+,R0-R1	Restore registers R0 and R1.
		Move (SP)+,FP	Restore frame pointer register.
		Return	Return to Subroutine 1.

**Figure 2.28** Nested subroutines.

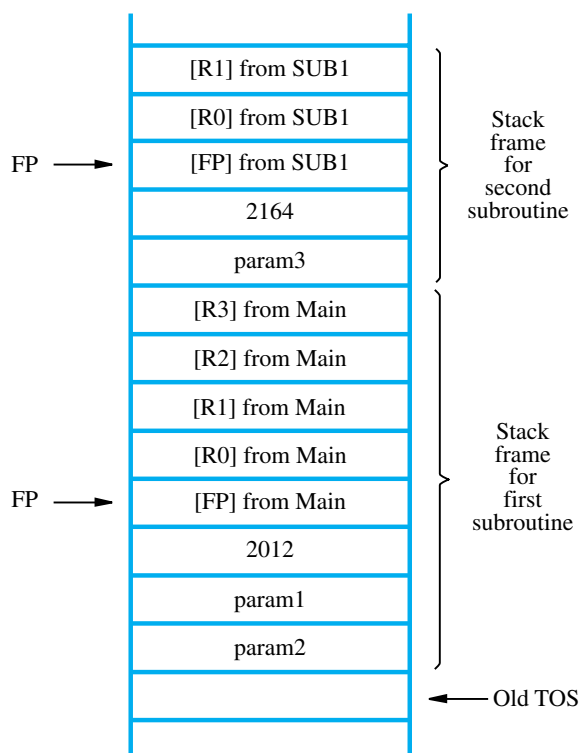


Figure 2.29 Stack frames for Figure 2.28.

SUB2, in order to perform some subtask. SUB1 passes a single parameter param3 to SUB2 and gets a result passed back to it. After SUB2 executes its Return instruction, this result is stored in register R2 by SUB1. 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 and continues with its computations at “next instruction.”

The comments in Figure 2.28 provide the details of how this flow of execution is managed. The first actions performed by each subroutine are to set the frame pointer, after saving its previous contents on the stack, and to save any other registers required. SUB1 uses four registers, R0 to R3, and SUB2 uses two registers, R0 and R1. These registers and the frame pointer are restored just before the returns 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 8, 12, . . . , as discussed for the general stack frame in Figure 2.27. Finally, note that the calling routines are responsible for removing parameters from the stack. This is done by the Add instruction in the main program, and by the Move instruction at location 2164 in SUB1.

## 2.10 ADDITIONAL INSTRUCTIONS

So far, we have introduced the following instructions: Move, Load, Store, Clear, Add, Subtract, Increment, Decrement, Branch, Testbit, Compare, Call, and Return. These 13 instructions, along with the addressing modes in Table 2.1, have allowed us to write routines to illustrate machine instruction sequencing, including branching and the subroutine structure. We also illustrated the basic memory-mapped I/O operations.

Even this small set of instructions has a number of redundancies. The Load and Store instructions can be replaced by Move, and the Increment and Decrement instructions can be replaced by Add and Subtract, respectively. Also, Clear can be replaced by a Move instruction containing an immediate operand of zero. Therefore, only 8 instructions would have been sufficient for our purposes. But, it is not unusual to have some redundancy in practical machine instruction sets. Certain simple operations can usually be accomplished in a number of different ways. Some alternatives may be more efficient than others. In this section we introduce a few more important instructions that are found in most instruction sets.

### 2.10.1 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

```
Not dst
```

complements all bits contained in the destination operand, changing 0s to 1s, and 1s to 0s. In Section 2.1.1, we saw that adding 1 to the 1's-complement of a signed positive number forms the negative version in 2's-complement representation. For example, in Figure 2.1, +3 (0011) is converted to -3 (1101) by adding 1 to the 1's-complement of 0011. If 3 is contained in register R0, the instructions

```
Not R0
Add #1,R0
```

achieve the conversion. Many computers have a single instruction

```
Negate R0
```

that accomplishes the same thing.

Now consider an application for the logic instruction And, which performs the bit-wise AND operation on the source and destination operands. Suppose that four ASCII characters are contained in the 32-bit register R0. In some task, we wish to determine if the leftmost character is Z. If it is, a conditional branch to YES is to be made. From Appendix E, we find that the ASCII code for Z is 01011010, which is expressed in

hexadecimal notation as 5A. The three-instruction sequence

```
And      #$FF000000,R0
Compare  #$5A000000,R0
Branch=0 YES
```

implements the desired action. The And instruction clears all bits in the rightmost three character positions of R0 to zero, leaving the leftmost character unchanged. This is the result of using an immediate source operand that has eight 1s at its left end, and 0s in the 24 bits to the right. The Compare instruction compares the remaining character at the left end of R0 with the binary representation for the character Z. The Branch instruction causes a branch to YES if there is a match.

The And instruction is often used in practical programming tasks where all bits of an operand except for some specified field are to be cleared to 0. In our example, the leftmost eight bits of R0 constitute the specified field.

### 2.10.2 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 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 left shift instruction is

```
LShiftL  count,dst
```

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 left shift 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, and the bits shifted out 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 2.30a shows an example of shifting the contents of register R0 left by two bit positions. The logical shift right instruction, LShiftR, works in the same manner except that it shifts to the right. Figure 2.30b 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

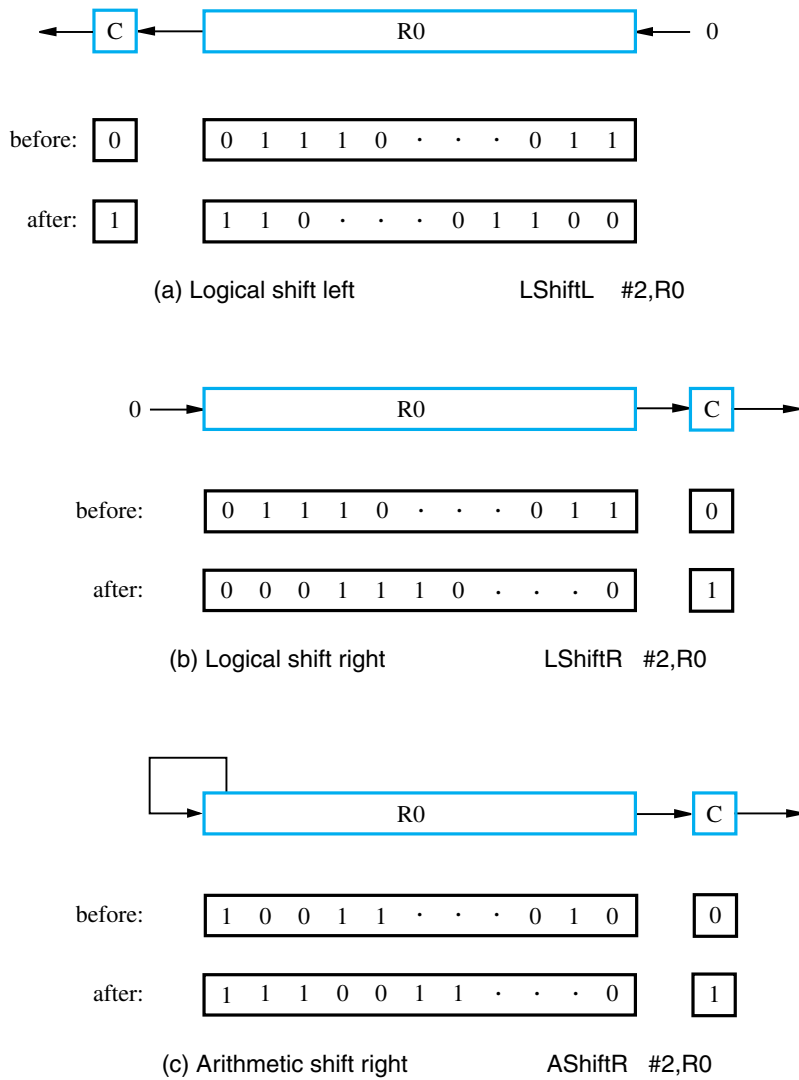


Figure 2.30 Logical and arithmetic shift instructions.

in 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. Tables E.1 and E.2 in Appendix E show 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 2.31 accomplishes the task using register R0 as a pointer to the ASCII characters in memory, and using registers R1 and R2 to

---

Move	#LOC,R0	R0 points to data.
MoveByte	(R0)+,R1	Load first byte into R1.
LShiftL	#4,R1	Shift left by 4 bit positions.
MoveByte	(R0),R2	Load second byte into R2.
And	#\$F,R2	Eliminate high-order bits.
Or	R1,R2	Concatenate the BCD digits.
MoveByte	R2,PACKED	Store the result.

---

**Figure 2.31** A routine that packs two BCD digits.

develop the BCD digit codes. When a MoveByte instruction transfers a byte between memory and a 32-bit processor register, we assume that the byte is located in the rightmost eight bit positions of the register. The And instruction is used to mask out all but the four rightmost bits in R2. Note that the immediate source operand is written as \$F, which, interpreted as a 32-bit pattern, has 28 zeros in the most-significant bit positions.

### Arithmetic Shifts

A study of the 2's-complement binary number representation in Figure 2.1 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 in 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. This requirement on right shifting distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are very similar. An example of an arithmetic right shift, AShiftR, is shown in Figure 2.30c. The arithmetic left shift is exactly the same as the logical left shift.

### 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. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. Figure 2.32 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 mnemonics RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations. The main use for Rotate instructions is in algorithms for performing arithmetic operations other than addition and subtraction, which we will encounter in Chapter 6.

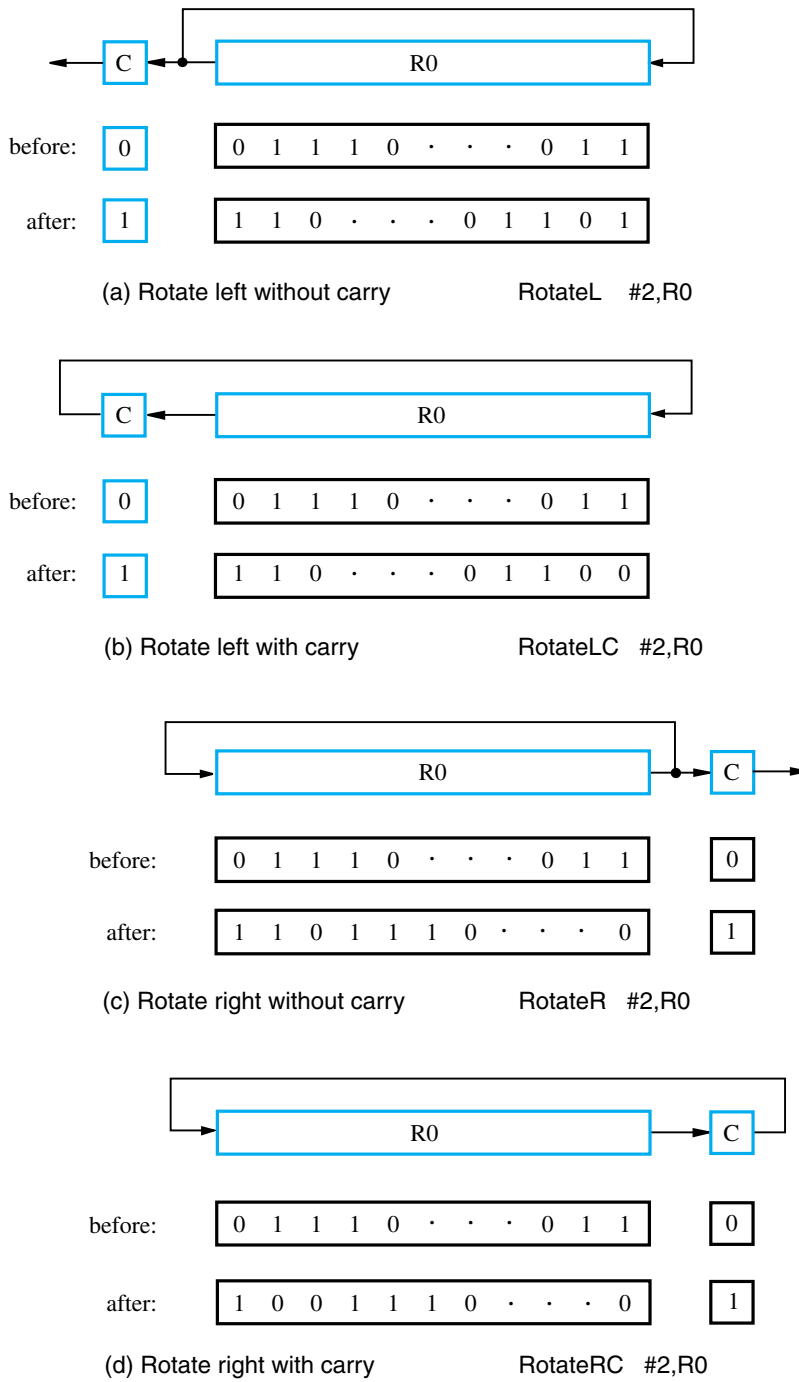


Figure 2.32 Rotate instructions.

### 2.10.3 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

Multiply  $R_i, R_j$

performs the operation

$$R_j \leftarrow [R_i] \times [R_j]$$

The product of two  $n$ -bit numbers can be as large as  $2n$  bits. Therefore, the answer will not necessarily fit into register  $R_j$ . A number of instruction sets have a Multiply instruction that computes the low-order  $n$  bits of the product and places it in register  $R_j$ , 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  $R_j$  and  $R(j + 1)$ , with the high-order half being placed in register  $R(j + 1)$ .

Although it is less common, some instruction sets provide a signed integer Divide instruction

Divide  $R_i, R_j$

which performs the operation

$$R_j \leftarrow [R_j]/[R_i]$$

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

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 6.

## 2.11 EXAMPLE PROGRAMS

In this section we present three examples that further illustrate the use of machine instructions. The examples are representative of numeric (vector processing) and non-numeric (sorting and linked-list manipulation) applications.

### 2.11.1 VECTOR DOT PRODUCT PROGRAM

The first example is a numerical application that is an extension of the loop program of Figure 2.16 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

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

---

	Move	#AVEC,R1	R1 points to vector A.
	Move	#BVEC,R2	R2 points to vector B.
	Move	N,R3	R3 serves as a counter.
	Clear	R0	R0 accumulates the dot product.
LOOP	Move	(R1)+,R4	Compute the product of
	Multiply	(R2)+,R4	next components.
	Add	R4,R0	Add to previous sum.
	Decrement	R3	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	R0,DOTPROD	Store dot product in memory.

---

**Figure 2.33** A program for computing the dot product of two vectors.

Figure 2.33 shows a program for computing the dot product and storing it in memory location DOTPROD. The first elements of each vector,  $A(0)$  and  $B(0)$ , 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 operation of the Multiply and Add instructions used in the program in Figure 2.33 into a single MultiplyAccumulate instruction. We will see an example of this in the ARM processor in Chapter 3.

### 2.11.2 BYTE-SORTING PROGRAM

Consider a program for sorting a list of bytes stored in memory into ascending alphabetic order. Assume that the list consists of  $n$  bytes, not necessarily distinct, and that each byte contains the ASCII code for a character from the set of letters A through Z. In the ASCII code, presented in Appendix E, 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.

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 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

---

```

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;
      }
    }
  }

```

---

(a) C-language program for sorting

---

	Move	#LIST,R0	Load LIST into base register R0.
	Move	N,R1	Initialize outer loop index
	Subtract	#1,R1	register R1 to $j = n - 1$ .
OUTER	Move	R1,R2	Initialize inner loop index
	Subtract	#1,R1	register R2 to $k = j - 1$ .
	MoveByte	(R0,R1),R3	Load LIST( $j$ ) into R3, which holds
			current maximum in sublist.
INNER	CompareByte	R3,(R0,R2)	If LIST( $k$ ) $\leq$ [R3],
	Branch $\leq$ 0	NEXT	do not exchange.
	MoveByte	(R0,R2),R4	Otherwise, exchange LIST( $k$ )
	MoveByte	R3,(R0,R2)	with LIST( $j$ ) and load
	MoveByte	R4,(R0,R1)	new maximum into R3.
	MoveByte	R4,R3	Register R4 serves as TEMP.
NEXT	Decrement	R2	Decrement index registers R2 and
	Branch $\geq$ 0	INNER	R1, which also serve as
	Decrement	R1	as loop counters, and branch
	Branch $>$ 0	OUTER	back if loops not finished.

---

(b) Assembly language program for sorting

**Figure 2.34** A byte-sorting program using straight-selection sort.

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 2.34a, 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  $LIST(j)$ .

The C-language program traverses the list backwards. This order of traversal simplifies loop termination in the machine language version of the program because the loop is exited when an index is decremented to 0.

An assembly language program that implements the sorting algorithm is given in Figure 2.34*b*. The comments in the program explain the use of various registers. The current maximum value is kept in register R3 while a sublist is being scanned. If a larger value is found, it is exchanged with the value in R3 and the new largest value is stored in LIST(*j*).

Control flow is handled differently in the two programs for purposes of efficiency in the assembly language program. Using the *if-then* control statement in the C-language program causes the three-line *then* clause to exchange LIST(*k*) and LIST(*j*) if LIST(*k*) > LIST(*j*). In the assembly language program, a branch is taken around the four-instruction exchange code if LIST(*k*) ≤ LIST(*j*).

If the machine instruction set allows a move operation from one memory location directly to another memory location, then the four-instruction exchange code in the inner loop in Figure 2.34*b* can be replaced by the three-instruction sequence

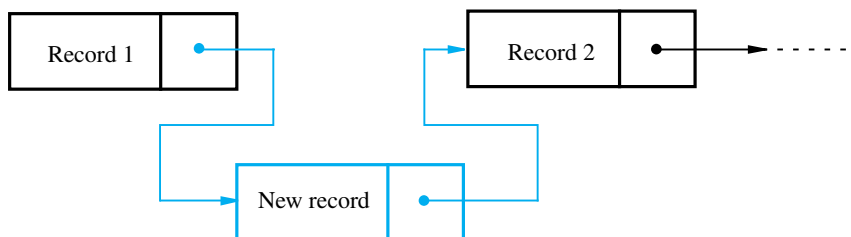
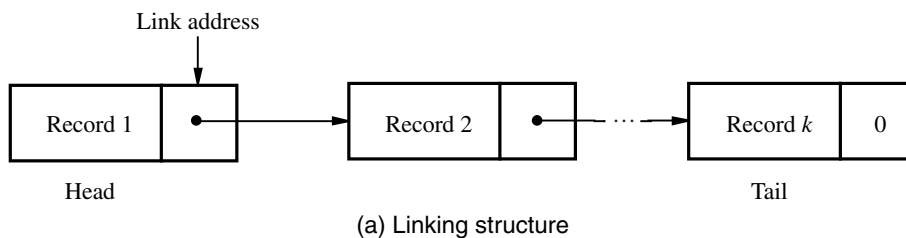
```
MoveByte (R0,R2),(R0,R1)
MoveByte R3,(R0,R2)
MoveByte (R0,R1),R3
```

As we will see in Chapter 3, the 68000 processor has this capability.

Finally, we note that the program in Figure 2.34*b* works correctly only if the list has at least two elements because the check for loop termination is done at the end of each loop. Hence, there is at least one pass through the loop, regardless of the value of *n*.

### 2.11.3 LINKED LISTS

Many nonnumeric application programs require that an ordered list of information items be represented and stored in memory in such a way that it is easy to add items to the list or to delete items from the list at any position while maintaining the desired order of items. This is a more general situation than found in the stack and queue data structures, discussed in Section 2.8, where items can only be added or deleted at the ends. Consider the following example. The course list of student test scores that we used in Section 2.5 to illustrate the Index addressing mode contains the unique student ID number in the first word of each four-word student record shown in Figure 2.14. Suppose we try to maintain this list of records in consecutive memory locations in some contiguous block of memory in increasing order of student ID numbers. This would facilitate printing and posting the list of test scores ordered by ID number. After the list is built, if a student withdraws from the course an empty record slot is created. It is then necessary to jump over the empty slot when going through the records to add up test scores or to print a listing. A more awkward situation arises after the initial construction of the list if another student registers in the course. To keep the list ordered, all records, starting from the one with the first ID number larger than the new ID would need to be moved to higher address locations to create a four-word space for the new record. Similarly,



(b) Inserting a new record between Record 1 and Record 2

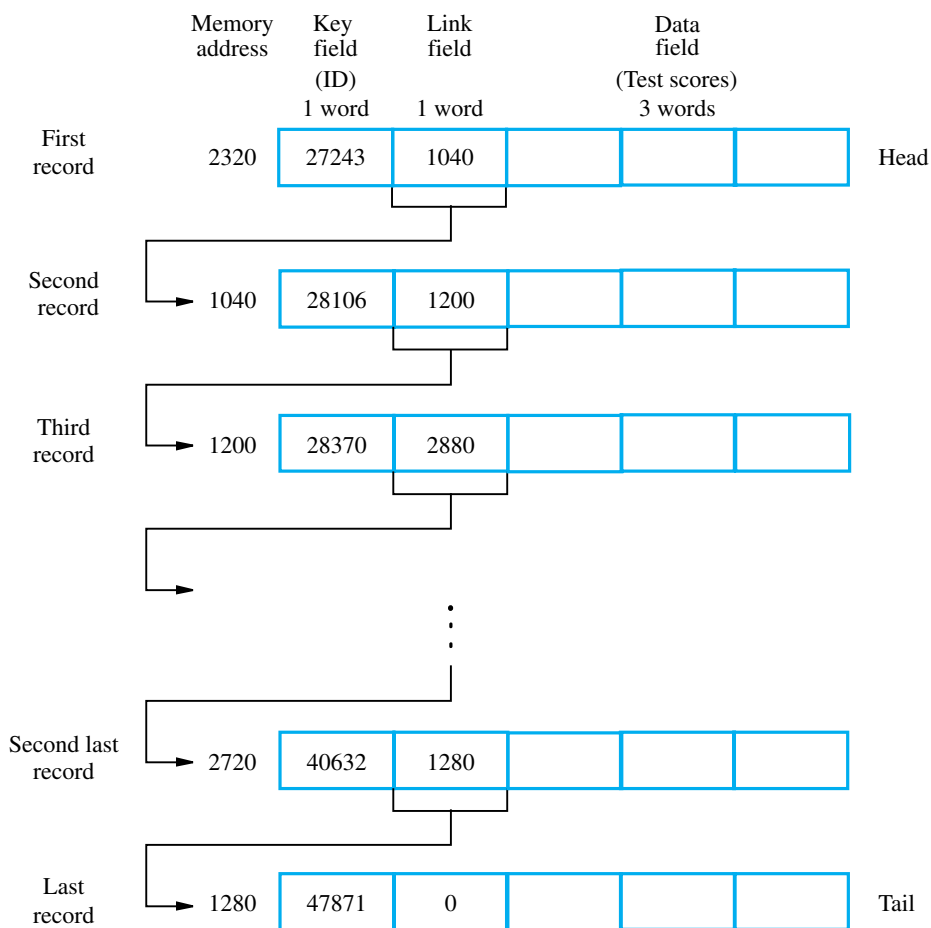
**Figure 2.35** Linked-list data structure.

to handle the previously mentioned withdrawal of a student, the resulting empty slot could be removed by moving all records after the empty slot to lower address locations, closing the gap.

A data structure called a *linked list* can be used to avoid both of these problems. Each record still occupies a consecutive four-word block in the memory, but successive records in the order do not necessarily occupy consecutive blocks in the memory address space. To enable connecting the blocks together to form the ordered list, each record contains an address value in a one-word *link* field that specifies the location of the next record in order. Hence the name linked list is used to describe this data structure. A schematic representation for a linked list is shown in Figure 2.35a. The first record in the list is called the *head*, and the last record is called the *tail*.

To insert a new record between record  $i$  and record  $i + 1$ , the link address in record  $i$  is copied into the link field in the new record and then the address of the new record is written into the link field of record  $i$ . This operation is shown schematically in Figure 2.35b. To delete record  $i$ , the address in its link field is copied into the link field of record  $i - 1$ .

Figure 2.36 shows an example of the student test score records linked together in memory, ordered by increasing ID numbers. Each record is now five words long. The first word, defined as the *key* field, contains the student ID number. The second word contains the link field, and the last three words are the data field that contains the three test scores. Assuming 32-bit words, a 2000-byte area of memory, starting at word address 1000, is allocated to contain the five-word records, 20 bytes per record, for up to 100 students. As students register for the course, they are assigned one of



**Figure 2.36** A list of student test scores organized as a linked list in memory.

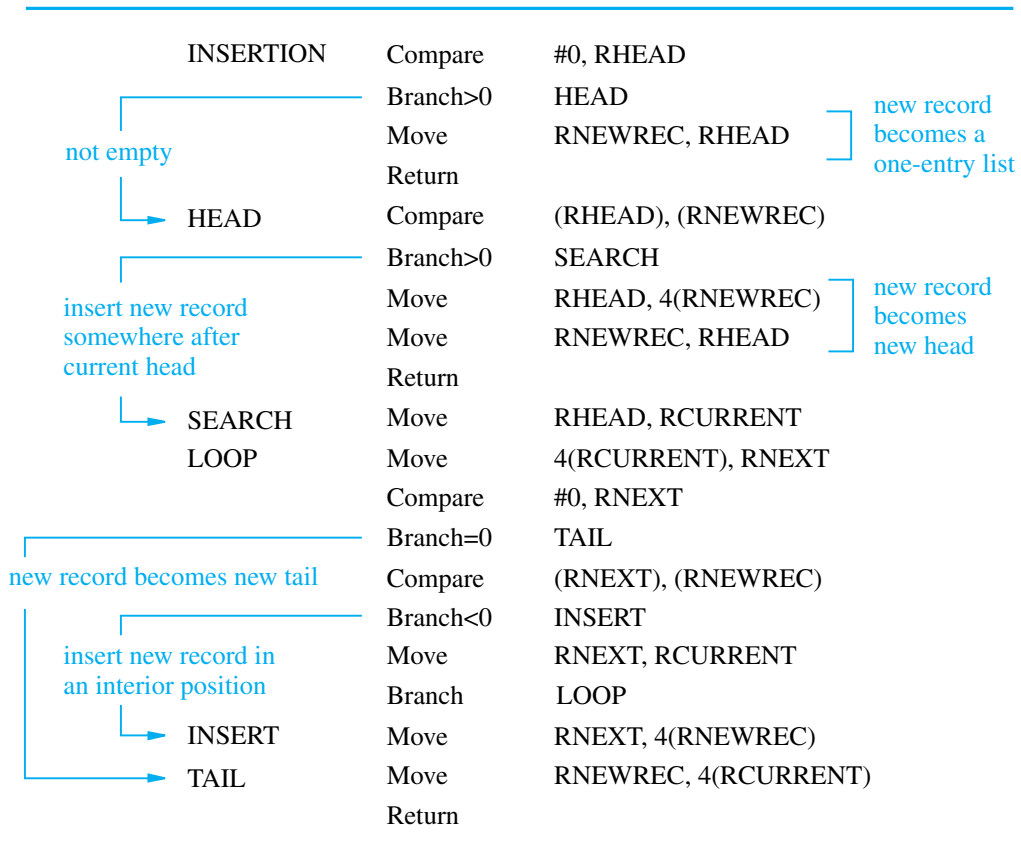
the available five-word record blocks in memory. It may be convenient to do this in block address order 1000, 1020, 1040, . . . , 2980, but that is not necessary. There is no particular relationship between student IDs and the order in which students register for the course. Therefore, the locations of the record blocks, ordered by student ID, will be scattered in some unpredictable way across the assigned memory area from block address 1000 to block address 2980.

The record with the current lowest ID number is at the head of the list, and the record with current highest ID number is in the tail position. A convenient way to access the list is to store the memory address of the head, in this case 2320, in a processor register called the *head pointer*. The address 1040 in the link field of the first record specifies the location of the second record. The link field of the second record contains the address 1200 of the third record, and so on. The link field of the last record is set to zero to denote that it is the tail entry in the list. If the list is empty, the head pointer contains zero.

**Insertion of a New Record**

Let us now give the steps needed to add a new record to the list shown in Figure 2.36. Suppose that the ID number of the new record is 28241, and the next available free record block is at address 2960. Trace forward from the head record until the first record with a larger ID is found. This is the record at memory location 1200, containing the ID 28370. Now insert the address link 1200 into the link field of the new record, and then insert the address of the new record, 2960, into the link field of the previous record at location 1040, overwriting the old value of 1200. Now, the new record has been inserted as the third record in the updated list, between the second and third records of the old list.

A subroutine for performing the insertion operation is shown in Figure 2.37. It is composed of three sections to handle the following three possible cases: the current list is empty, the new record becomes the new head of a nonempty list, or the new record is inserted in the list somewhere after the current head. The last case includes the possibility that the new record becomes the tail.



**Figure 2.37** A subroutine for inserting a new record into a linked list.

Consider now how the subroutine handles the three possible cases. A number of processor registers are used in the subroutine. Instead of the usual names R0, R1, R2, and so on, we use more descriptive names to aid understanding. RHEAD is the head pointer, and RNEWREC contains the address of the new record. The two registers RCURRENT and RNEXT contain the addresses of the current record and the next record as the list is scanned to find the correct position for inserting the new record. The link field of the new record is initially set to zero. If it becomes the new tail, no further changes to this field are necessary.

The first Compare/Branch pair of instructions checks whether or not the list is empty. If it is empty (RHEAD contains 0), the new record becomes a one-entry list by moving its address into RHEAD, followed by a Return instruction; otherwise, the second Compare/Branch pair checks whether or not the new record becomes the new head. If it does, the two Move instructions make the necessary changes to the link field of the new record and the contents of RHEAD, and a Return is executed. If the new record does not become the new head, the last half of the subroutine determines the position in the list where the new record is to be inserted. It is then inserted at the correct interior position by the last two Move instructions, or it is made the new tail by the last Move instruction. We have omitted saving and restoring registers in this subroutine to improve readability and understanding of the insertion operation.

### Deletion of a Record

The deletion of an existing record from a linked list is an easier operation than the insertion of a new record. We simply go forward through the list until we find the ID of the record that is to be deleted. The necessary link field adjustments are then made.

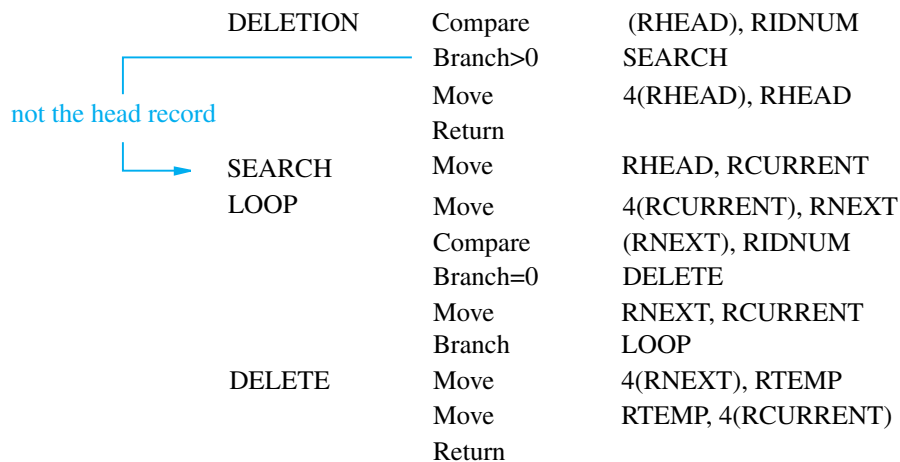
Figure 2.38 shows a subroutine that implements the deletion operation. We assume that a register RIDNUM contains the ID of the record to be deleted. Registers RHEAD, RCURRENT, and RNEXT, play the same roles as in the insertion subroutine. The first Compare/Branch pair of instructions checks whether or not the record to be deleted is the head. If it is, the record is deleted by moving the link field address of the head record into RHEAD. Note that if the head is the only record in the list, then its link field contains zero, signifying that it is also the tail. So moving this zero into RHEAD properly signifies the empty list condition. If the head record is not the record to be deleted, then a branch is made to SEARCH. Now, registers RCURRENT and RNEXT are used to search forward from the head until the desired record is found. When the desired record is found by the second Compare/Branch pair, a branch is made to DELETE. The record, pointed to by RNEXT, is removed by transferring its link field to the link field of the previous record, pointed to by RCURRENT. The last two Move instructions accomplish this transfer through the register RTEMP. If a memory-to-memory Move instruction is available, then the single instruction

```
Move 4(RNEXT),4(RCURRENT)
```

can replace these two Move instructions.

### Error Conditions

The insertion and deletion subroutines in Figures 2.37 and 2.38 do not take into account the possibility of two error conditions. The insertion subroutine makes the



**Figure 2.38** A subroutine for deleting a record from a linked list.

assumption that there is no record in the list with the new ID, and the deletion subroutine assumes that there is a record with the ID to be deleted. Modifying the subroutines to account for these error possibilities is considered in problems 2.23 and 2.24.

## 2.12 ENCODING OF MACHINE INSTRUCTIONS

We have introduced a variety of useful instructions and addressing modes. These instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. Actually, the form in which we have presented the instructions is indicative of the forms used in assembly languages, except that we tried to avoid using acronyms for the various operations, which are awkward to memorize and are likely to be specific to a particular commercial processor. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as *machine instructions*. The instructions that use symbolic names and acronyms are called *assembly language instructions*, which are converted into the machine instructions using the assembler program as explained in Section 2.6.

In the previous sections, we made a simplifying assumption that all instructions are one word in length. Since we usually refer to 32-bit words, our assumption implies that this length is adequate to represent the necessary information. Let us now consider the validity of this assumption.

We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers or 8-bit ASCII-encoded characters. The type of operation that is to be performed and the type of operands used may be specified using an encoded

binary pattern referred to as the *OP code* for the given instruction. Suppose that 8 bits are allocated for this purpose, giving 256 possibilities for specifying different instructions. This leaves 24 bits to specify the rest of the required information.

Let us examine some typical cases. The instruction

Add R1,R2

has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing mode is used for each operand.

The instruction

Move 24(R0),R5

requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24. Suppose that three bits are used to specify an addressing mode in Table 2.1. Then six bits have to be available for this purpose, denoting the chosen addressing modes of the source and destination operands. Hence, there are 10 bits left to give the index value. If these 10 bits suffice to express an adequate range of signed numbers for indexing purposes, then the instruction fits into our 32-bit word.

The shift instruction

LshiftR #2,R0

and the move instruction

Move #\$3A,R1

have to indicate the immediate values 2 and \$3A, respectively, in addition to the 18 bits used to specify the OP code, the addressing modes, and the register. This limits the size of the immediate operand to what is expressible in 14 bits.

Consider next the branch instruction

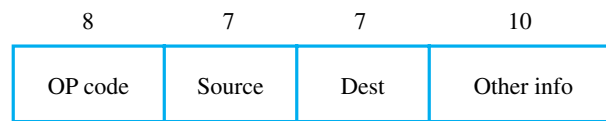
Branch>0 LOOP

Again, 8 bits are used for the OP code, leaving 24 bits to specify the branch offset. Since the offset is a 2's-complement number, the branch target address must be within  $2^{23}$  bytes of the location of the branch instruction. To branch to an instruction outside this range, a different addressing mode has to be used, such as Absolute or Register Indirect. Branch instructions that use these modes are usually called Jump instructions.

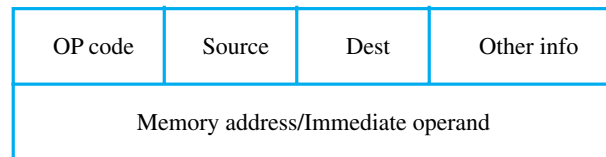
In all these examples, the instructions can be encoded in a 32-bit word. Figure 2.39a depicts a possible format. There is an 8-bit OP-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction

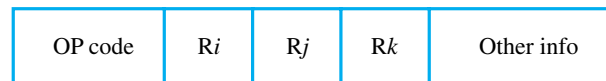
Move R2,LOC



(a) One-word instruction



(b) Two-word instruction



(c) Three-operand instruction

**Figure 2.39** Encoding instructions into 32-bit words.

requires 18 bits to denote the OP code, the addressing modes, and the register. This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient. If we want to be able to give a complete 32-bit address in the instruction, then the only solution is to include a second word as a part of this instruction, in which case the additional word can contain the required memory address. A suitable format is shown in Figure 2.39*b*. The first word may be the same as in part *a* of the figure. The second is a full memory address. This format can also accommodate instructions such as

And # $\$$ FF000000,R2

in which case the second word gives a full 32-bit immediate operand.

If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

Move LOC1,LOC2

then it becomes necessary to use two additional words for the 32-bit addresses of the operands.

This approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used. Using multiple words, we can implement quite complex instructions, closely resembling operations in high-level

programming languages. The term *complex instruction set computer* (CISC) has been used to refer to processors that use instruction sets of this type.

There exists a radically different alternative to this approach. If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction. But, it is still possible to define a highly functional instruction set, which makes extensive use of the processor registers. Thus, we can have

Add R1,R2

but not

Add LOC,R2

Instead of the latter instruction, we can use

Add (R3),R2

provided that we load the address LOC into register R3 before the instruction is executed. In this case, register R3 is being used as a pointer to the desired memory location.

This raises the issue of how to load a 32-bit address into a register that serves as a pointer to memory locations. One possibility is to direct the assembler to place the desired address in a word location in a data area close to the program. Then the Relative addressing mode can be used to load the address. This assumes that the index field contained in the Load instruction is large enough to reach the location containing the desired address. Another possibility is to use logical and shift instructions to construct the desired 32-bit address by giving it in parts that are small enough to be specifiable using the Immediate addressing mode. This issue is considered in more detail for the ARM processor in Chapter 3. All ARM instructions are encoded into a single 32-bit word.

The restriction that an instruction must occupy only one word has led to a style of computers that have become known as *reduced instruction set computers* (RISC). The RISC approach introduced other restrictions, such as that all manipulation of data must be done on operands that are already in processor registers. This restriction means that the above addition would need a two-instruction sequence

Move (R3),R1

Add R1,R2

If the Add instruction only has to specify the two registers, it will need just a portion of a 32-bit word. So, we may provide a more powerful instruction that uses three operands

Add R1,R2,R3

which performs the operation

$$R3 \leftarrow [R1] + [R2]$$

A possible format for such an instruction is shown in Figure 2.39c. Of course, the processor has to be able to deal with such three-operand instructions. In an instruction set where all arithmetic and logical operations use only register operands, the only memory references are made to load/store the operands into/from the processor registers.

RISC-type instruction sets typically have fewer and less complex instructions than CISC-type sets. We will discuss the relative merits of RISC and CISC approaches in Chapter 8, which deals with the details of processor design.

### 2.13 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. Several addressing modes were introduced, including the important concepts of pointers and indexed addressing. Basic I/O operations were discussed, showing how characters are transferred between the processor and keyboard and display devices. The subroutine concept and the instructions needed to implement it were also discussed. Subroutine linkage methods provided an example of the application of the stack data structure. The way in which machine instructions manipulate other data structures was also explained. Queues, arrays, and linked lists were considered. We described two different approaches to the design of machine instruction sets — the CISC and RISC approaches. The execution-time performance of these two design styles will be further developed in Chapter 8.

### PROBLEMS

- 2.1** Represent the decimal values 5,  $-2$ , 14,  $-10$ , 26,  $-19$ , 51, and  $-43$ , as signed, 7-bit numbers in the following binary formats:

- (a) Sign-and-magnitude
- (b) 1's-complement
- (c) 2's-complement

(See Appendix E for decimal-to-binary integer conversion.)

- 2.2** (a) Convert the following pairs of decimal numbers to 5-bit, signed, 2's-complement, binary numbers and add them. State whether or not overflow occurs in each case.

- (a) 5 and 10
- (b) 7 and 13
- (c)  $-14$  and 11
- (d)  $-5$  and 7
- (e)  $-3$  and  $-8$
- (f)  $-10$  and  $-13$

- (b) Repeat Part a 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.

- 2.3** Given a binary pattern in some memory location, is it possible to tell whether this pattern represents a machine instruction or a number?
- 2.4** A memory byte location contains the pattern 00101100. What does this pattern represent when interpreted as a binary number? What does it represent as an ASCII code?
- 2.5** 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 name “Johnson” has been entered.
- 2.6** Repeat Problem 2.5 for the little-endian scheme.
- 2.7** A program reads ASCII characters representing the digits of a decimal number as they are entered at a keyboard and stores the characters in successive memory bytes. Examine the ASCII code in Appendix E and indicate what operation is needed to convert each character into an equivalent binary number.
- 2.8** Write a program that can evaluate the expression

$$A \times B + C \times D$$

in a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions, and that all values fit in the accumulator.

- 2.9** The list of student marks shown in Figure 2.14 is changed to contain  $j$  test scores for each student. Assume that there are  $n$  students. Write an assembly language 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 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. 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  $j$  is stored in memory location  $J$ , placed ahead of location  $N$ .
- 2.10** (a) Rewrite the dot product program in Figure 2.33 for an instruction set in which the arithmetic and logic operations can only be applied to operands in processor registers. The two instructions Load and Store are used to transfer operands between registers and the memory.
- (b) Calculate the values of the constants  $k_1$  and  $k_2$  in the expression  $k_1 + k_2n$ , which represents the number of memory accesses required to execute your program for Part a, including instruction word fetches. Assume that each instruction occupies a single word.
- 2.11** Repeat Problem 2.10 for a computer with two-address instructions, which can perform operations such as

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

where A and B can be either memory locations or processor registers. Which computer

requires fewer memory accesses? (Chapter 8 on pipelining gives a different perspective on the answer to this question.)

- 2.12** “Having a large number of processor registers makes it possible to reduce the number of memory accesses needed to perform complex tasks.” Devise a simple computational task to show the validity of this statement for a processor that has four registers compared to another that has only two registers.
- 2.13** Registers R1 and R2 of a computer contain the decimal values 1200 and 4600. What is the effective address of the memory operand in each of the following instructions?

- (a) Load 20(R1),R5
- (b) Move #3000,R5
- (c) Store R5,30(R1,R2)
- (d) Add  $-(R2)$ ,R5
- (e) Subtract (R1)+,R5

- 2.14** Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an assembly language program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.
- 2.15** 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 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 R1 and R2. The parameters  $n$  and  $m$  are passed to the subroutine in registers R3 and R4, and the address of element  $A(0,0)$  is passed in register R0. Any of the addressing modes in Table 2.1 can be used. At most, one operand of an instruction can be in the memory.
- 2.16** Both of the following statements cause the value 300 to be stored in location 1000, but at different times.

```
ORIGIN    1000
DATAWORD 300
```

and

```
Move #300,1000
```

Explain the difference.

- 2.17** Register R5 is used in a program to point to the top of a stack. 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, and 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.
- 2.18** A FIFO queue of bytes is to be implemented in the memory, occupying a fixed region of  $k$  bytes. You need two pointers, 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 queue, and the OUT pointer keeps track of the location containing the next byte to be removed from 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.
- 2.19** Consider the queue structure described in Problem 2.18. Write APPEND and REMOVE 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.
- 2.20** 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)?
- 2.21** The subroutine call instruction of a computer saves the return address in a processor register called the link register, RL. What would you do to allow subroutine nesting? Would your scheme allow the subroutine to call itself?
- 2.22** Assume you want to organize subroutine calls on a computer as follows: When routine Main wishes to call subroutine SUB1, it calls an intermediate routine, CALLSUB, and passes to it the address of SUB1 as a parameter in register R1. CALLSUB saves the return address on a stack, making sure that the upper limit of the stack is not exceeded. Then it branches to SUB1. To return to the calling program, subroutine SUB1 calls another intermediate routine, RETRN. This routine checks that the stack is not empty and then uses the top element to return to the original calling program.
- Write routines CALLSUB and RETRN, assuming that the subroutine call instruction saves the return address in a link register, RL. The upper and lower limits of the

stack are recorded in memory locations UPPERLIMIT and LOWERLIMIT, respectively.

- 2.23** The linked-list insertion subroutine in Figure 2.37 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register ERROR if this occurs or return a zero if the insertion is successful.
- 2.24** The linked-list deletion subroutine in Figure 2.38 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful or leave RIDNUM unchanged if the record is not in the list.