Unit 1---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

1.1---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

Digital Logic Circuits

IN THIS CHAPTER

1-1 Digital Computers

The digital computer is a digital system that performs various computational tasks. The word *digital* implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, . . . , 9, for example, provide 10 discrete values. The first electronic digital computers, developed in the late 1940s, were used primarily for numerical computations. In this case the discrete elements are the digits. From this application the term *digital computer* has emerged. In practice, digital computers function more reliably if only two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e., true-or-false, yes-or-no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be *binary*.

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a *bit*. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet. By judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations.

In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1. The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 75$$

The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French.

A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. A sequence of

program

instructions for the computer is called a *program*. The data that are manipulated by the program constitute the *data base*.

A computer system is composed of its hardware and the system software available for its use. The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the *operating system*. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an application program, but the compiler that translates the high-level language program to machine language is a system program. The customer who buys a computer system would need, in addition to the hardware, any available software needed for effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware.

computer hardware

The hardware of the computer is usually divided into three major parts, as shown in Fig. 1-1. The central processing unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called a random-access memory (RAM) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and
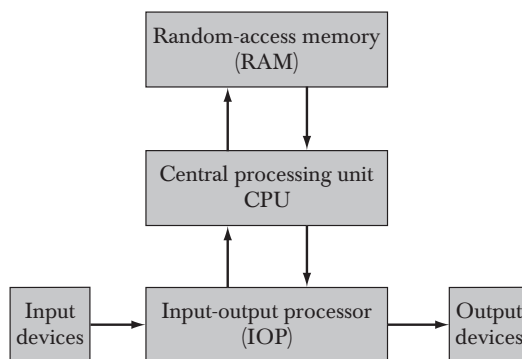
Figure 1-1   Block diagram of a digital computer.

output processor (IOP) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

This book provides the basic knowledge necessary to understand the hardware operations of a computer system. The subject is sometimes considered from three different points of view, depending on the interest of the investigator. When dealing with computer hardware it is customary to distinguish between what is referred to as computer organization, computer design, and computer architecture.

computer organization

Computer organization is concerned with the way the hardware components operate and the way they are connected together to form the computer system. The various components are assumed to be in place and the task is to investigate the organizational structure to verify that the computer parts operate as intended.

computer design

Computer design is concerned with the hardware design of the computer. Once the computer specifications are formulated, it is the task of the designer to develop hardware for the system. Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as *computer implementation*.

computer architecture

Computer architecture is concerned with the structure and behavior of the computer as seen by the user. It includes the information, formats, the instruction set, and techniques for addressing memory. The architectural design of a computer system is concerned with the specifications of the various functional modules, such as processors and memories, and structuring them together into a computer system.

Two basic types of computer architectures are von Neumann architecture and Harvard architecture. von Neumann architecture describes a general framework, or structure, that a computer's hardware, programming, and data should follow. Although other structures for computing have been devised and implemented, the vast majority of computers in use today operate according to the von

Neumann architecture. Von Neumann envisioned the structure of a computer system as being composed of the following components:

1. the central arithmetic unit, which today is called the arithmetic-logic unit (ALU). This unit performs the computer's computational and logical functions;

2. memory; more specifically, the computer's main, or fast, memory, such as random access memory (RAM);

3. a control unit that directs other components of the computer to perform certain actions, such as directing the fetching of data or instructions from memory to be processed by the ALU; and

4. man-machine interfaces; i.e., input and output devices, such as a keyboard for input and display monitor for output, as shown in Fig. 1.1.

Of course, computer technology has developed extensively since von Neumann's time. For instance, due to integrated circuitry and miniaturization, the ALU and control unit have been integrated onto the same microprocessor "chip", becoming an integrated part of the computer's central processing unit (CPU). The most noteworthy concept contained in von Neumann's first report was most likely that of the stored-program principle. This principle holds that data, as well as the instructions used to manipulate that data, should be stored together in the same memory area of the computer and instructions are carried out sequentially, one instruction at a time. The sequential execution of programming imposes a sort of 'speed limit' on program execution, since only one instruction at a time can be handled by the computer's processor. It means that the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same signal pathways and memory.

The Harvard architecture uses physically separate storage and signal pathways for their instructions and data. The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape (24-bits wide) and data in relay latches (23-digits wide). In a computer with Harvard architecture, the CPU can read both an instruction and data from memory at the same time, leading to double the memory bandwidth.

An example of computer architecture based on the von Neumann architecture is the desktop personal computer. Microcontroller (single-chip microcomputer)-based computer systems and DSP (Digital Signal Processor)-based computer systems are examples for Harvard architecture.

The book deals with all three subjects associated with computer hardware. In Chapters 1 through 4 we present the various digital components used in the organization and design of computer systems. Chapters 5 through 7 cover the steps that a designer must go through to design and program an elementary digital computer. Chapters 8 and 9 deal with the architecture of the central processing unit. In Chapters 11 and 12 we present the organization and architecture of the input–output processor and the memory unit.

1-2 Logic Gates

Binary information is represented in digital computers by physical quantities called *signals*. Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and 0.5 volts to represent binary input and output corresponding to 1 and 0, respectively.

gates

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called *gates*. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input–output relationship of the binary variables for each gate can be represented in tabular form by a *truth table*. The basic logic gates are AND and inclusive OR with multiple inputs and NOT with a single input. Each gate with more than one input is sensitive to either logic 0 or logic 1 input at any one of its inputs, generating the output according to its function. For example, a multi-input AND gate is sensitive to logic 0 on any one of its inputs, irrespective of any values at other inputs.

The names, graphic symbols, algebraic functions, and truth tables of eight logic gates are listed in Fig. 1-2, with applicable sensitivity input values. Each gate has one or two binary input variables designated by A and B and one binary output variable designated by x. The AND gate produces the AND logic function: that is, the output is 1 if input A and input B are both equal to 1; otherwise, the output is 0. These conditions are also specified in the truth table for the AND gate. The table shows that output x is 1 only when both input A and input B are 1. The algebraic operation symbol of the AND function is the same as the multiplication symbol of ordinary arithmetic. We can either use a dot between the variables or concatenate the variables without an operation symbol between them. AND gates may have more than two inputs, and by definition, the output is 1 if and only if all inputs are 1.

OR

The OR gate produces the inclusive-OR function; that is, the output is 1 if input A or input B or both inputs are 1; otherwise, the output is 0. The algebraic symbol of the OR function is $+$, similar to arithmetic addition. OR gates may have more than two inputs, and by definition, the output is 1 if any input is 1.

inverter

The inverter circuit inverts the logic sense of a binary signal. It produces the NOT, or complement, function. The algebraic symbol used for the logic complement is either a prime or a bar over the variable symbol. In this book we use a prime for the logic complement of a binary variable, while a bar over the letter is reserved for designating a complement microoperation as defined in Chap. 4.

The small circle in the output of the graphic symbol of an inverter designates a logic complement. A triangle symbol by itself designates a buffer circuit. A
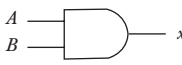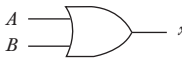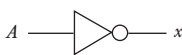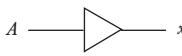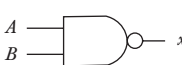
Name	Graphic symbol	Algebraic function	Truth table	Input sensitivity
AND	A, B → x	$x = A \cdot B$ or $x = AB$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 0$ $0\ 1\ \vert\ 0$ $1\ 0\ \vert\ 0$ $1\ 1\ \vert\ 1$	0
OR	A, B → x	$x = A + B$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 0$ $0\ 1\ \vert\ 1$ $1\ 0\ \vert\ 1$ $1\ 1\ \vert\ 1$	1
Inverter	A → x	$x = A'$	$A\ \vert\ x$ $0\ \vert\ 1$ $1\ \vert\ 0$	Not Applicable
Buffer	A → x	$x = A$	$A\ \vert\ x$ $0\ \vert\ 0$ $1\ \vert\ 1$	Not Applicable
NAND	A, B → x	$x = (AB)'$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 1$ $0\ 1\ \vert\ 1$ $1\ 0\ \vert\ 1$ $1\ 1\ \vert\ 0$	0
NOR	A, B → x	$x = (A + B)'$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 1$ $0\ 1\ \vert\ 0$ $1\ 0\ \vert\ 0$ $1\ 1\ \vert\ 0$	1
Exclusive-OR (XOR)	A, B → x	$x = A \oplus B$ or $x = A'B + AB'$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 0$ $0\ 1\ \vert\ 1$ $1\ 0\ \vert\ 1$ $1\ 1\ \vert\ 0$	Not Applicable
Exclusive-NOR or equivalence	A, B → x	$x = (A \oplus B)'$ or $x = AB + A'B'$	$A\ B\ \vert\ x$ $0\ 0\ \vert\ 1$ $0\ 1\ \vert\ 0$ $1\ 0\ \vert\ 0$ $1\ 1\ \vert\ 1$	Not Applicable

Figure 1-2 Digital logic gates with applicable input sensitivity values.

1.2---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

1-2 Logic Gates

Binary information is represented in digital computers by physical quantities called *signals*. Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and 0.5 volts to represent binary input and output corresponding to 1 and 0, respectively.

gates

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called *gates*. Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input–output relationship of the binary variables for each gate can be represented in tabular form by a *truth table*. The basic logic gates are AND and inclusive OR with multiple inputs and NOT with a single input. Each gate with more than one input is sensitive to either logic 0 or logic 1 input at any one of its inputs, generating the output according to its function. For example, a multi-input AND gate is sensitive to logic 0 on any one of its inputs, irrespective of any values at other inputs.

The names, graphic symbols, algebraic functions, and truth tables of eight logic gates are listed in Fig. 1-2, with applicable sensitivity input values. Each gate has one or two binary input variables designated by A and B and one binary output variable designated by x. The AND gate produces the AND logic function: that is, the output is 1 if input A and input B are both equal to 1; otherwise, the output is 0. These conditions are also specified in the truth table for the AND gate. The table shows that output x is 1 only when both input A and input B are 1. The algebraic operation symbol of the AND function is the same as the multiplication symbol of ordinary arithmetic. We can either use a dot between the variables or concatenate the variables without an operation symbol between them. AND gates may have more than two inputs, and by definition, the output is 1 if and only if all inputs are 1.

OR

The OR gate produces the inclusive-OR function; that is, the output is 1 if input A or input B or both inputs are 1; otherwise, the output is 0. The algebraic symbol of the OR function is $+$, similar to arithmetic addition. OR gates may have more than two inputs, and by definition, the output is 1 if any input is 1.

inverter

The inverter circuit inverts the logic sense of a binary signal. It produces the NOT, or complement, function. The algebraic symbol used for the logic complement is either a prime or a bar over the variable symbol. In this book we use a prime for the logic complement of a binary variable, while a bar over the letter is reserved for designating a complement microoperation as defined in Chap. 4.

The small circle in the output of the graphic symbol of an inverter designates a logic complement. A triangle symbol by itself designates a buffer circuit. A

Name	Graphic symbol	Algebraic function	Truth table	Input sensitivity
AND	A B — x	$x = A \cdot B$ or $x = AB$	$A\ B \mid x$ $0\ 0 \mid 0$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$	0
OR	A B — x	$x = A + B$	$A\ B \mid x$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$	1
Inverter	A — x	$x = A'$	$A \mid x$ $0 \mid 1$ $1 \mid 0$	Not Applicable
Buffer	A — x	$x = A$	$A \mid x$ $0 \mid 0$ $1 \mid 1$	Not Applicable
NAND	A B — x	$x = (AB)'$	$A\ B \mid x$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$	0
NOR	A B — x	$x = (A + B)'$	$A\ B \mid x$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$	1
Exclusive-OR (XOR)	A B — x	$x = A \oplus B$ or $x = A'B + AB'$	$A\ B \mid x$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$	Not Applicable
Exclusive-NOR or equivalence	A B — x	$x = (A \oplus B)'$ or $x = AB + A'B'$	$A\ B \mid x$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$	Not Applicable

Figure 1-2 Digital logic gates with applicable input sensitivity values.

buffer does not produce any particular logic function since the binary value of the output is the same as the binary value of the input. This circuit is used merely for power amplification. For example, a buffer that uses 3 volts for binary 1 will produce an output of 3 volts when its input is 3 volts. However, the amount of electrical power needed at the input of the buffer is much less than the power produced at the output of the buffer. The main purpose of the buffer is to drive other gates that require a large amount of power.

NAND   The NAND function is the complement of the AND function, as indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle. The designation NAND is derived from the abbreviation of NOT-AND. The NOR gate is the complement of the OR gate and uses an OR

NOR   graphic symbol followed by a small circle. Both NAND and NOR gates may have more than two inputs, and the output is always the complement of the AND or OR function, respectively.

exclusive-OR   The exclusive-OR gate has a graphic symbol similar to the OR gate except for the additional curved line on the input side. The output of this gate is 1 if any input is 1 but excludes the combination when both inputs are 1. The exclusive-OR function has its own algebraic symbol or can be expressed in terms of AND, OR, and complement operations as shown in Fig. 1-2. The exclusive-NOR is the complement of the exclusive-OR, as indicated by the small circle in the graphic symbol. The output of this gate is 1 only if both inputs are equal to 1 or both inputs are equal to 0. A more fitting name for the exclusive-OR operation would be an odd function; that is, its output is 1 if an odd number of inputs are 1. Thus in a three-input exclusive-OR (odd) function, the output is 1 if only one input is 1 or if all three inputs are 1. The exclusive-OR and exclusive-NOR gates are commonly available with two inputs, and only seldom are they found with three or more inputs.

1-3   Boolean Algebra

A Boolean algebra is an algebra (set, operations, elements) consisting of a set B with ≥ 2 elements, together with three operations–the AND operation · (Boolean product), the OR operation + (Boolean sum), and the NOT operation $'$ (complement)–defined on the set, such that for any element a, b, c, … of set B, $a \cdot b$, $a + b$, and a' are in B. Consider the four-element Boolean algebra $B_4 = (\{0, x, y, 1\}; \cdot, +, '; 0, 1)$. The AND, OR, and NOT operations are described by the following tables:

·	0	x	y	1
0	0	0	0	0
x	0	x	0	x
y	0	0	y	y
1	0	x	y	1

+	0	x	y	1
0	0	x	y	1
x	x	x	1	1
y	y	1	y	1
1	1	1	1	1

$'$	
0	1
x	y
y	x
1	0

Consider the two-element Boolean algebra $B_2 = (\{0, 1\}; \cdot, +, '; 0, 1)$. The three operations. (AND), + (OR), '(NOT) are defined as follows:

$\cdot$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	1

$'$	
0	1
1	0

The two-element Boolean algebra B_2 among all other B_i, where $i > 2$, defined as switching algebra, is the most useful. Switching algebra consists of two elements represented by 1 and 0 as the largest number and the smallest number respectively.

*Boolean algebra is a switching algebra that deals with binary variables and logic operations. The variables are designated by letters such as A, B, x, and y. The three basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

Boolean function

$$F = x + y'z$$

The function F is equal to 1 if x is 1 or if both y' and z are equal to 1; F is equal to 0 otherwise. But saying that $y' = 1$ is equivalent to saying that $y = 0$ since y' is the complement of y. Therefore, we may say that F is equal to 1 if $x = 1$ or if $yz = 01$. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2^n combinations of the n binary variables. As shown in Fig. l-3(a), there are eight possible distinct combinations for assigning bits to the three variables x, y, and z. The function F is equal to 1 for those combinations where $x = 1$ or $yz = 01$; it is equal to 0 for all other combinations.

truth table

A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates. The logic diagram for F is shown in Fig. l-3(b). There is an inverter for input y to generate its

logic diagram

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table

(b) Logic diagram

Figure 1-3 Truth table and logic diagram for $F = x + y'z$.

*Two Element

complement y'. There is an AND gate for the term $y'z$, and an OR gate is used to combine the two terms. In a logic diagram, the variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit.

The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in algebraic form a truth table relationship between binary variables.
2. Express in algebraic form the input–output relationship of logic diagrams.
3. Find simpler circuits for the same function.

A Boolean function specified by a truth table can be expressed algebraically in many different ways. Two ways of forming Boolean expressions are canonical and non-canonical forms. Canonical forms express all binary variables in every product (AND) or sum (OR) term of the Boolean function. To determine the canonical sum-of-products form for a Boolean function $F(A, B, C) = A'B + C' + ABC$, which is in non-canonical form, the following steps are used:

$$F = A'B + C' + ABC$$

$$= A'B(C + C') + (A + A')(B + B')C' + ABC,$$

where $x + x' = 1$ is a basic identity of Boolean algebra

$$= A'BC + A'BC' + ABC' + AB'C' + A'BC' + A'B'C' + ABC$$

$$= A'BC + A'BC' + ABC' + AB'C' + A'B'C' + ABC$$

By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

Table 1-1 lists the most basic identities of Boolean algebra. All the identities in the table can be proven by means of truth tables. The first eight identities show the basic relationship between a single variable and itself, or in

TABLE 1-1 Basic Identities of Boolean Algebra

(1) $x + 0 = x$	(2) $x \cdot 0 = 0$
(3) $x + 1 = 1$	(4) $x \cdot 1 = x$
(5) $x + x = x$	(6) $x \cdot x = x$
(7) $x + x' = 1$	(8) $x \cdot x' = 0$
(9) $x + y = y + x$	(10) $xy = yx$
(11) $x + (y + z) = (x + y) + z$	(12) $x(yz) = (xy)z$
(13) $x(y + z) = xy + xz$	(14) $x + yz = (x + y)(x + z)$
(15) $(x + y)' = x'y'$	(16) $(xy)' = x' + y'$
(17) $(x')' = x$	

buffer does not produce any particular logic function since the binary value of the output is the same as the binary value of the input. This circuit is used merely for power amplification. For example, a buffer that uses 3 volts for binary 1 will produce an output of 3 volts when its input is 3 volts. However, the amount of electrical power needed at the input of the buffer is much less than the power produced at the output of the buffer. The main purpose of the buffer is to drive other gates that require a large amount of power.

NAND   The NAND function is the complement of the AND function, as indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle. The designation NAND is derived from the abbreviation of NOT-AND. The NOR gate is the complement of the OR gate and uses an OR

NOR   graphic symbol followed by a small circle. Both NAND and NOR gates may have more than two inputs, and the output is always the complement of the AND or OR function, respectively.

exclusive-OR   The exclusive-OR gate has a graphic symbol similar to the OR gate except for the additional curved line on the input side. The output of this gate is 1 if any input is 1 but excludes the combination when both inputs are 1. The exclusive-OR function has its own algebraic symbol or can be expressed in terms of AND, OR, and complement operations as shown in Fig. 1-2. The exclusive-NOR is the complement of the exclusive-OR, as indicated by the small circle in the graphic symbol. The output of this gate is 1 only if both inputs are equal to 1 or both inputs are equal to 0. A more fitting name for the exclusive-OR operation would be an odd function; that is, its output is 1 if an odd number of inputs are 1. Thus in a three-input exclusive-OR (odd) function, the output is 1 if only one input is 1 or if all three inputs are 1. The exclusive-OR and exclusive-NOR gates are commonly available with two inputs, and only seldom are they found with three or more inputs.

1-3   Boolean Algebra

A Boolean algebra is an algebra (set, operations, elements) consisting of a set B with ≥ 2 elements, together with three operations—the AND operation $\cdot$ (Boolean product), the OR operation $+$ (Boolean sum), and the NOT operation $'$ (complement)—defined on the set, such that for any element a, b, c, … of set B, $a \cdot b$, $a + b$, and a' are in B. Consider the four-element Boolean algebra $B_4 = (\{0, x, y, 1\}; \cdot, +, '; 0, 1)$. The AND, OR, and NOT operations are described by the following tables:

$\cdot$	0	x	y	1
0	0	0	0	0
x	0	x	0	x
y	0	0	y	y
1	0	x	y	1

$+$	0	x	y	1
0	0	x	y	0
x	x	x	1	1
y	y	1	y	1
1	1	1	1	1

$'$	
0	1
x	y
y	x
1	0

Consider the two-element Boolean algebra $B_2 = (\{0, 1\}; \cdot, +, '; 0, 1)$. The three operations. (AND), + (OR), '(NOT) are defined as follows:

$\cdot$	0	1
0	0	0
1	0	1

+	0	1
0	0	1
1	1	1

$'$	
0	1
1	0

The two-element Boolean algebra B_2 among all other B_i, where $i > 2$, defined as switching algebra, is the most useful. Switching algebra consists of two elements represented by 1 and 0 as the largest number and the smallest number respectively.

*Boolean algebra is a switching algebra that deals with binary variables and logic operations. The variables are designated by letters such as A, B, x, and y. The three basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal sign. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

Boolean function

$$F = x + y'z$$

The function F is equal to 1 if x is 1 or if both y' and z are equal to 1; F is equal to 0 otherwise. But saying that $y' = 1$ is equivalent to saying that $y = 0$ since y' is the complement of y. Therefore, we may say that F is equal to 1 if $x = 1$ or if $yz = 01$. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2^n combinations of the n binary variables. As shown in Fig. l-3(a), there are eight possible distinct combinations for assigning bits to the three variables x, y, and z. The function F is equal to 1 for those combinations where $x = 1$ or $yz = 01$; it is equal to 0 for all other combinations.

truth table

A Boolean function can be transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates. The logic diagram for F is shown in Fig. l-3(b). There is an inverter for input y to generate its

logic diagram

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth table

(b) Logic diagram

Figure 1-3   Truth table and logic diagram for $F = x + y'z$.

*Two Element

complement y'. There is an AND gate for the term $y'z$, and an OR gate is used to combine the two terms. In a logic diagram, the variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit.

The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in algebraic form a truth table relationship between binary variables.
2. Express in algebraic form the input–output relationship of logic diagrams.
3. Find simpler circuits for the same function.

A Boolean function specified by a truth table can be expressed algebraically in many different ways. ==Two ways of forming Boolean expressions are canonical and non-canonical forms.== Canonical forms express all binary variables in every product (AND) or sum (OR) term of the Boolean function. To determine the canonical sum-of-products form for a Boolean function $F(A, B, C) = A'B + C' + ABC$, which is in non-canonical form, the following steps are used:

$$F = A'B + C' + ABC$$

$$= A'B(C + C') + (A + A')(B + B')C' + ABC,$$

where $x + x' = 1$ is a basic identity of Boolean algebra

$$= A'BC + A'BC' + ABC' + AB'C' + A'BC' + A'B'C' + ABC$$

$$= A'BC + A'BC' + ABC' + AB'C' + A'B'C' + ABC$$

By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

Table 1-1 lists the most basic identities of Boolean algebra. All the identities in the table can be proven by means of truth tables. The first eight identities show the basic relationship between a single variable and itself, or in

TABLE 1-1 Basic Identities of Boolean Algebra

(1) $x + 0 = x$	(2) $x \cdot 0 = 0$
(3) $x + 1 = 1$	(4) $x \cdot 1 = x$
(5) $x + x = x$	(6) $x \cdot x = x$
(7) $x + x' = 1$	(8) $x \cdot x' = 0$
(9) $x + y = y + x$	(10) $xy = yx$
(11) $x + (y + z) = (x + y) + z$	(12) $x(yz) = (xy)z$
(13) $x(y + z) = xy + xz$	(14) $x + yz = (x + y)(x + z)$
(15) $(x + y)' = x'y'$	(16) $(xy)' = x' + y'$
(17) $(x')' = x$	

conjunction with the binary constants 1 and 0. The next five identities (9 through 13) are similar to ordinary algebra. Identity 14 does not apply in ordinary algebra but is very useful in manipulating Boolean expressions. Identities 15 and 16 are called DeMorgan's theorems and are discussed below. The last identity states that if a variable is complemented twice, one obtains the original value of the variable.

The identities listed in the table apply to single variables or to Boolean functions expressed in terms of binary variables. For example, consider the following Boolean algebra expression:

$$AB' + C'D + AB' + C'D$$

By letting $x = AB' + C'D$ the expression can be written as $x + x$. From identity 5 in Table 1-1 we find that $x + x = x$. Thus the expression can be reduced to only two terms:

$$AB' + C'D + A'B + C'D = AB' + CD$$

DeMorgan's theorem   DeMorgan's theorem is very important in dealing with NOR and NAND gates. It states that a NOR gate that performs the $(x + y)'$ function is equivalent to the function $x'y'$. Similarly, a NAND function can be expressed by either $(xy)'$ or $(x' + y')$. For this reason the NOR and NAND gates have two distinct graphic symbols, as shown in Figs. 1-4 and 1-5. Instead of representing a NOR gate with an OR graphic symbol followed by a circle, we can represent it by an AND graphic symbol preceded by circles in all inputs. The invert-AND symbol for the NOR gate follows from DeMorgan's theorem and from the convention that small circles denote complementation. Similarly, the NAND gate has two distinct symbols, as shown in Fig. 1-5. NAND and NOR gates can be used to implement any Boolean function, including basic logic gates such as AND, OR, and NOT. Hence, NAND and NOR gates are called as Universal gates.

Figure 1-4   Two graphic symbols for NOR gate.

(a) OR-invert   (b) invert-AND
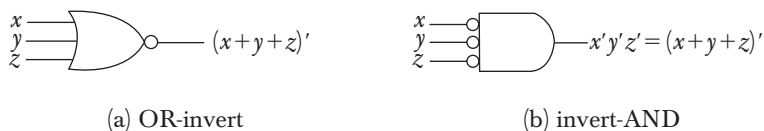
Figure 1-5   Two graphic symbols for NAND gate.

(a) AND-invert   (b) invert-OR
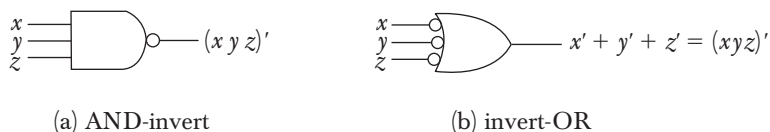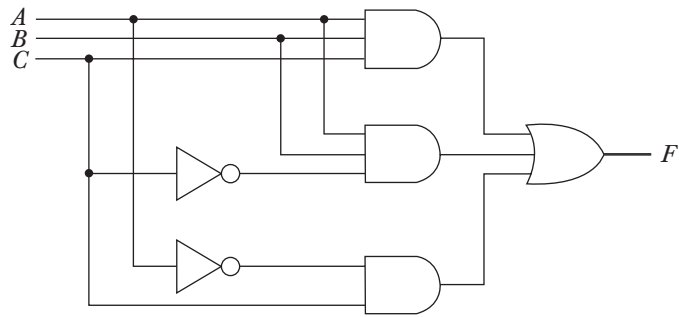
(a) $F = ABC + ABC' + A'C$
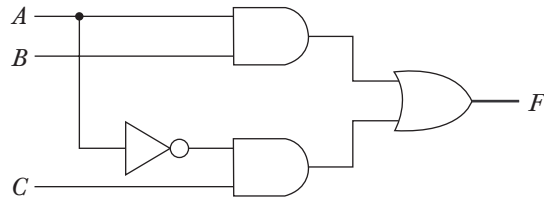
(b) $F = AB + A'C$
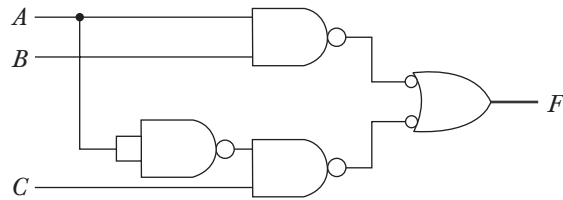
(c) $F = AB + AC'$ using NAND gates

Figure 1-6   Three logic diagrams for the same Boolean function.

To see how Boolean algebra manipulation is used to simplify digital circuits, consider the logic diagram of Fig. l-6(a). The output of the circuit can be expressed algebraically as follows:

$$F = ABC + ABC' + A'C$$

Each term corresponds to one AND gate, and the OR gate forms the logical sum of the three terms. Two inverters are needed to complement A' and C'. The expression can be simplified using Boolean algebra.

$$F = ABC + ABC' + A'C = AB(C + C') + A'C = AB + A'C$$

Note that $(C + C)' = 1$ by identity 7 and $AB \cdot 1 = AB$ by identity 4 in Table 1-1.

The logic diagram of the simplified expression is drawn in Fig. l-6(b) and Fig. l-6(c). It requires only four gates rather than the six gates used in the circuit of Fig. l-6(a). The two circuits are equivalent and produce the same truth table relationship between inputs *A*, *B*, *C* and output *F*.

Complement of a Function

The complement of a function *F* when expressed in a truth table is obtained by interchanging l's and O's in the values of *F* in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(x_1 + x_2 + x_3 + \cdots + x_n)' = x_1' x_2' x_3' \cdots x_n'$$

$$(x_1 x_2 x_3 \cdots x_n)' = x_1' + x_2' + x_3' + \cdots + x_n'$$

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D$$

$$F' = (A' + B')(C + D)(B + D')$$

The complement expression is obtained by interchanging AND and OR operations and complementing each individual variable. Note that the complement of *C'* is *C*.

1-4   Map Simplification

The complexity of the logic diagram that implements a Boolean function is related directly to the complexity of the algebraic expression from which the function is implemented. The truth table representation of a function is unique, but the function can appear in many different forms when expressed algebraically. The expression may be simplified using the basic relations of Boolean algebra. However, this procedure is sometimes difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. Two methods of simplifying Boolean algebraic expressions are the map method and the tabular method. The map method is used for functions upto six variables. To manipulate functions of a large number of variables, the tabular method also known as the Quine-McCluskey method, is used. If a function to be minimized is not in a canonical form, it must first be converted into canonical form before applying Quine-McCluskey tabular

procedure. Another tabular method, known as the iterative consensus method, begins the simplification process even if the function is not in a canonical form. The map method provides a simple, straightforward procedure for simplifying Boolean expressions. This method may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as the Karnaugh map or K-map.

minterm Each combination of the variables in a truth table is called a minterm. For example, the truth table of Fig. 1-3 contains eight minterms. When expressed in a truth table a function of n variables will have 2^n minterms, equivalent to the 2^n binary numbers obtained from n bits. A Boolean function is equal to 1 for some minterms and to 0 for others. The information contained in a truth table may be expressed in compact form by listing the decimal equivalent of those minterms that produce a 1 for the function. For example, the truth table of Fig. 1-3 can be expressed as follows:

$$F(x, y, z) = \Sigma\ (1, 4, 5, 6, 7)$$

The letters in parentheses list the binary variables in the order that they appear in the truth table. The symbol Σ stands for the sum of the minterms that follow in parentheses. The minterms that produce 1 for the function are listed in their decimal equivalent. The minterms missing from the list are the ones that produce 0 for the function.

The map is a diagram made up of squares, with each square representing one minterm. The squares corresponding to minterms that produce 1 for the function are marked by a 1 and the others are marked by a 0 or are left empty. By recognizing various patterns and combining squares marked by l's in the map, it is possible to derive alternative algebraic expressions for the function, from which the most convenient may be selected.

The maps for functions of two, three, and four variables are shown in Fig. 1-7. The number of squares in a map of n variables is 2^n. The 2^n minterms are listed by an equivalent decimal number for easy reference. The minterm numbers are assigned in an orderly arrangement such that adjacent squares represent minterms that differ by only one variable. The variable names are listed across both sides of the diagonal line in the corner of the map. The 0's and l's marked along each row and each column designate the value of the variables. Each variable under brackets contains half of the squares in the map where that variable appears unprimed. The variable appears with a prime (complemented) in the remaining half of the squares.

The minterm represented by a square is determined from the binary assignments of the variables along the left and top edges in the map. For example, minterm 5 in the three-variable map is 101 in binary, which may be obtained from the 1 in the second row concatenated with the 01 of the second column. This minterm represents a value for the binary variables A, B, and C, with A and C

(a) Two-variable map　　(b) Three-variable map

(c) Four-variable map

Figure 1-7 Maps for two-, three-, and four-variable functions.

being unprimed and B being primed (i.e., $AB'C$). On the other hand, minterm 5 in the four-variable map represents a minterm for four variables. The binary number contains the four bits 0101, and the corresponding term it represents is $A'BC'D$.

adjacent squares　　　Minterms of adjacent squares in the map are identical except for one variable, which appears complemented in one square and uncomplemented in the adjacent square. According to this definition of adjacency, the squares at the extreme ends of the same horizontal row are also to be considered adjacent. The same applies to the top and bottom squares of a column. As a result, the four corner squares of a map must also be considered to be adjacent.

A Boolean function represented by a truth table is plotted into the map by inserting l's in those squares where the function is 1. The squares containing l's are combined in groups of adjacent squares. These groups must contain a number of squares that is an integral power of 2. Groups of combined adjacent squares may share one or more squares with one or more groups. Each group of squares represents an algebraic term, and the OR of those terms gives the simplified algebraic expression for the function. The following examples show the use of the map for simplifying Boolean functions.

In the first example we will simplify the Boolean function

$$F(A, B, C) = \Sigma (3, 4, 6, 7)$$

The logic diagram of the simplified expression is drawn in Fig. l-6(b) and Fig. l-6(c). It requires only four gates rather than the six gates used in the circuit of Fig. l-6(a). The two circuits are equivalent and produce the same truth table relationship between inputs A, B, C and output F.

Complement of a Function

The complement of a function F when expressed in a truth table is obtained by interchanging l's and O's in the values of F in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(x_1 + x_2 + x_3 + \cdots + x_n)' = x_1' x_2' x_3' \cdots x_n'$$
$$(x_1 x_2 x_3 \cdots x_n)' = x_1' + x_2' + x_3' + \cdots + x_n'$$

From the general DeMorgan's theorem we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D$$
$$F' = (A' + B')(C + D)(B + D')$$

The complement expression is obtained by interchanging AND and OR operations and complementing each individual variable. Note that the complement of C' is C.

1-4 Map Simplification

The complexity of the logic diagram that implements a Boolean function is related directly to the complexity of the algebraic expression from which the function is implemented. The truth table representation of a function is unique, but the function can appear in many different forms when expressed algebraically. The expression may be simplified using the basic relations of Boolean algebra. However, this procedure is sometimes difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. Two methods of simplifying Boolean algebraic expressions are the map method and the tabular method. The map method is used for functions upto six variables. To manipulate functions of a large number of variables, the tabular method also known as the Quine-McCluskey method, is used. If a function to be minimized is not in a canonical form, it must first be converted into canonical form before applying Quine-McCluskey tabular

procedure. Another tabular method, known as the iterative consensus method, begins the simplification process even if the function is not in a canonical form. The map method provides a simple, straightforward procedure for simplifying Boolean expressions. This method may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as the Karnaugh map or K-map.

minterm   Each combination of the variables in a truth table is called a minterm. For example, the truth table of Fig. 1-3 contains eight minterms. When expressed in a truth table a function of n variables will have 2^n minterms, equivalent to the 2^n binary numbers obtained from n bits. A Boolean function is equal to 1 for some minterms and to 0 for others. The information contained in a truth table may be expressed in compact form by listing the decimal equivalent of those minterms that produce a 1 for the function. For example, the truth table of Fig. 1-3 can be expressed as follows:

$$F(x, y, z) = \Sigma \ (1, 4, 5, 6, 7)$$

The letters in parentheses list the binary variables in the order that they appear in the truth table. The symbol Σ stands for the sum of the minterms that follow in parentheses. The minterms that produce 1 for the function are listed in their decimal equivalent. The minterms missing from the list are the ones that produce 0 for the function.

The map is a diagram made up of squares, with each square representing one minterm. The squares corresponding to minterms that produce 1 for the function are marked by a 1 and the others are marked by a 0 or are left empty. By recognizing various patterns and combining squares marked by l's in the map, it is possible to derive alternative algebraic expressions for the function, from which the most convenient may be selected.

The maps for functions of two, three, and four variables are shown in Fig. 1-7. The number of squares in a map of n variables is 2^n. The 2^n minterms are listed by an equivalent decimal number for easy reference. The minterm numbers are assigned in an orderly arrangement such that adjacent squares represent minterms that differ by only one variable. The variable names are listed across both sides of the diagonal line in the corner of the map. The 0's and l's marked along each row and each column designate the value of the variables. Each variable under brackets contains half of the squares in the map where that variable appears unprimed. The variable appears with a prime (complemented) in the remaining half of the squares.

The minterm represented by a square is determined from the binary assignments of the variables along the left and top edges in the map. For example, minterm 5 in the three-variable map is 101 in binary, which may be obtained from the 1 in the second row concatenated with the 01 of the second column. This minterm represents a value for the binary variables A, B, and C, with A and C

(a) Two-variable map

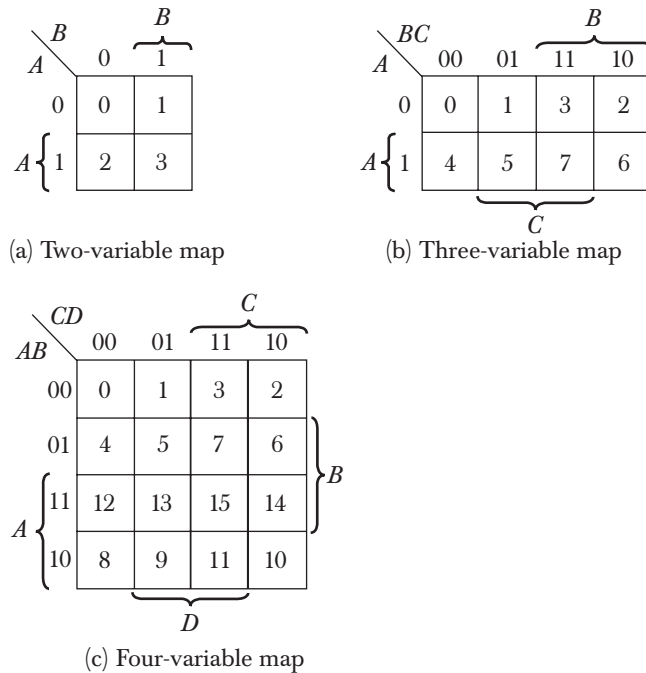(b) Three-variable map

(c) Four-variable map

Figure 1-7   Maps for two-, three-, and four-variable functions.

being unprimed and B being primed (i.e., $AB'C$). On the other hand, minterm 5 in the four-variable map represents a minterm for four variables. The binary number contains the four bits 0101, and the corresponding term it represents is $A'BC'D$.

adjacent squares   Minterms of adjacent squares in the map are identical except for one variable, which appears complemented in one square and uncomplemented in the adjacent square. According to this definition of adjacency, the squares at the extreme ends of the same horizontal row are also to be considered adjacent. The same applies to the top and bottom squares of a column. As a result, the four corner squares of a map must also be considered to be adjacent.

A Boolean function represented by a truth table is plotted into the map by inserting l's in those squares where the function is 1. The squares containing l's are combined in groups of adjacent squares. These groups must contain a number of squares that is an integral power of 2. Groups of combined adjacent squares may share one or more squares with one or more groups. Each group of squares represents an algebraic term, and the OR of those terms gives the simplified algebraic expression for the function. The following examples show the use of the map for simplifying Boolean functions.

In the first example we will simplify the Boolean function

$$F(A, B, C) = \Sigma (3, 4, 6, 7)$$

Figure 1-8   Map for $F(A, B, C) = \Sigma (3, 4, 6, 7)$.

The three-variable map for this function is shown in Fig. 1-8. There are four squares marked with l's, one for each minterm that produces 1 for the function. These squares belong to minterms 3, 4, 6, and 7 and are recognized from Fig. l-7(b). Two adjacent squares are combined in the third column. This column belongs to both B and C and produces the term BC. The remaining two squares with l's in the two corners of the second row are adjacent and belong to row A and the two columns of C', so they produce the term AC'. The simplified algebraic expression for the function is the OR of the two terms:

$$F = BC + AC'$$

The second example simplifies the following Boolean function:

$$F(A, B, C) = \Sigma (0, 2, 4, 5, 6)$$

The five minterms are marked with l's in the corresponding squares of the three-variable map shown in Fig. 1-9. The four squares in the first and fourth columns are adjacent and represent the term C'. The remaining square marked with a 1 belongs to minterm 5 and can be combined with the square of minterm 4 to produce the term AB'. The simplified function is

$$F = C' + AB'$$
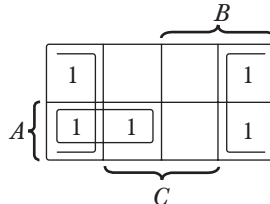
Figure 1-9   Map for $F(A, B, C) = \Sigma (3, 4, 6, 7)$.
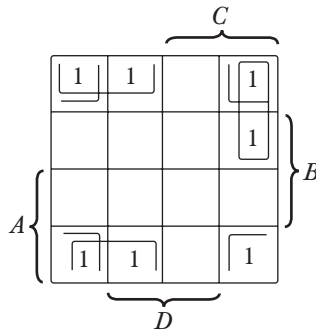
Figure 1-10   Map for $F(A, B, C, D) = \sum (0, 1, 2, 6, 8, 9, 10)$.

The third example needs a four-variable map.

$$F(A,\ B,\ C,\ D) = \sum (0,\ 1,\ 2,\ 6,\ 8,\ 9,\ 10)$$

The area in the map covered by this four-variable function consists of the squares marked with 1's in Fig. 1-10. The function contains 1's in the four corners that, when taken as a group, give the term $B'D'$. This is possible because these four squares are adjacent when the map is considered with top and bottom or left and right edges touching. The two 1's on the left of the top row are combined with the two 1's on the left of the bottom row to give the term $B'C'$. The remaining 1 in the square of minterm 6 is combined with minterm 2 to give the term $A'CD'$. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

Product-of-Sums Simplification

The Boolean expressions derived from the maps in the preceding examples were expressed in sum-of-products form. The product terms are AND terms and the sum denotes the ORing of these terms. It is sometimes convenient to obtain the algebraic expression for the function in a product-of-sums form. The sums are OR terms and the product denotes the ANDing of these terms. With a minor modification, a product-of-sums form can be obtained from a map.

The procedure for obtaining a product-of-sums expression follows from the basic properties of Boolean algebra. The 1's in the map represent the minterms that produce 1 for the function. The squares not marked by 1 represent the minterms that produce 0 for the function. If we mark the empty squares with 0's and combine them into groups of adjacent squares, we obtain the complement of the function, F'. Taking the complement of F' produces an expression for F in product-of-sums form. The best way to show this is by example.
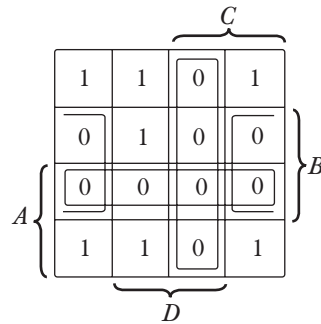
Figure 1-11   Map for $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$.

We wish to simplify the following Boolean function in both sum-of-products form and product-of-sums form:

$$F(A, B, C, D) = \Sigma\,(0, 1, 2, 5, 8, 9, 10)$$

The l's marked in the map of Fig. 1-11 represent the minterms that produce a 1 for the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with l's gives the simplified function in sum-of-products form:

$$F = B'D' + B'C' + A'C'D$$

If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

$$F' = AB + CD + BD'$$

Taking the complement of F', we obtain the simplified function in product-of-sums form:

$$F = (A' + B')(C' + D')(B' + D)$$

The logic diagrams of the two simplified expressions are shown in Fig. 1-12. The sum-of-products expression is implemented in Fig. l-12(a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in Fig. l-12(b) in product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case it is assumed that the input variables are directly available in their complement, so inverters are not included. The pattern established in Fig. 1-12 is the general form by which any Boolean function is implemented when expressed in one of the standard

(a) Sum of products:
$F = B'D' + B'C' + A'C'D$

(b) Product of sums:
$F = (A' + B') (C' + D') (B' + D)$
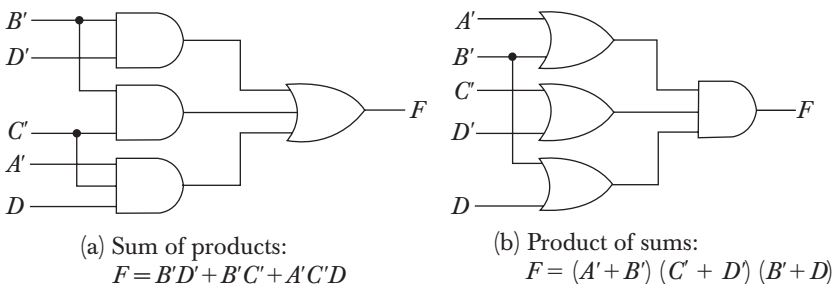
Figure 1-12 Logic diagrams with AND and OR gates.

forms. AND gates are connected to a single OR gate when in sum-of-products form. OR gates are connected to a single AND gate when in product-of-sums form.

NAND implementation

A sum-of-products expression can be implemented with NAND gates as shown in Fig. l-13(a). Note that the second NAND gate is drawn with the graphic symbol of Fig. l-5(b). There are three lines in the diagram with small circles at both ends. Two circles in the same line designate double complementation, and since $(x')' = x$, the two circles can be removed and the resulting diagram is equivalent to the one shown in Fig. l-12(a). Similarly, a product-of-sums expression can be imple-

NOR implementation

mented with NOR gates as shown in Fig. 1-13(b). The second NOR gate is drawn with the graphic symbol of Fig. l-4(b). Again the two circles on both sides of each line may be removed, and the diagram so obtained is equivalent to the one shown in Fig. l-12(b).

Don't-Care Conditions

The l's and 0's in the map represent the minterms that make the function equal to 1 or 0. There are occasions when it does not matter if the function produces 0 or 1 for a given minterm. Since the function may be either 0 or 1, we say that we don't care what the function output is to be for this minterm. ==Minterms that may

don't-care conditions

produce either 0 or 1 for the function are said to be don't-care conditions and are marked with an $\times$ in the map. These don't-care conditions can be used to provide further simplification of the algebraic expression.==

Figure 1-13 Logic diagrams with NAND or NOR gates.

(a) With NAND gates

(b) With NOR gates
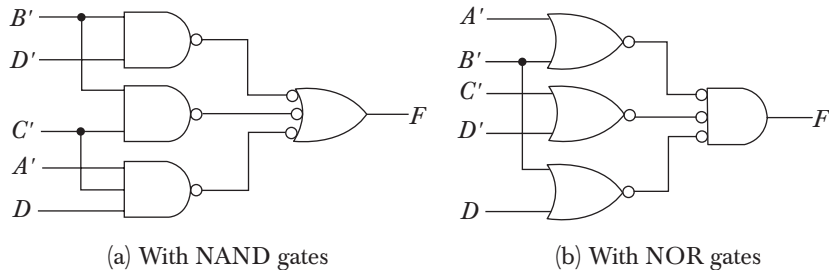
When choosing adjacent squares for the function in the map, the $\times$'s may be assumed to be either 0 or 1, whichever gives the simplest expression. In addition, an $\times$ need not be used at all if it does not contribute to the simplification of the function. In each case, the choice depends only on the simplification that can be achieved. As an example, consider the following Boolean function together with the don't-care minterms:

$$F(A, B, C) = \Sigma\, (0, 2, 6)$$
$$d(A, B, C) = \Sigma\, (1, 3, 5)$$

The minterms listed with F produce a 1 for the function. The don't-care minterms listed with d may produce either a 0 or a 1 for the function. The remaining minterms, 4 and 7, produce a 0 for the function. The map is shown in Fig. 1-14. The minterms of F are marked with 1's, those of d are marked with $\times$'s, and the remaining squares are marked with 0's. The 1's and $\times$'s are combined in any convenient manner so as to enclose the maximum number of adjacent squares. It is not necessary to include all or any of the $\times$'s, but all the 1's must be included. By including the don't care minterms 1 and 3 with the 1's in the first row we obtain the term A'. The remaining 1 for minterm 6 is combined with minterm 2 to obtain the term BC'. The simplified expression is

$$F = A' + BC'$$

Note that don't-care minterm 5 was not included because it does not contribute to the simplification of the expression. Note also that if don't-care minterms 1 and 3 were not included with the 1's, the simplified expression for F would have been

$$F = A'C' + BC'$$

This would require two AND gates and an OR gate, as compared to the expression obtained previously, which requires only one AND and one OR gate.

The function is determined completely once the $\times$'s are assigned to the 1's or 0's in the map. Thus the expression

$$F = A' + BC'$$

represents the Boolean function

$$F(A, B, C) = \Sigma\, (0, 1, 2, 3, 6)$$

It consists of the original minterms 0, 2, and 6 and the don't-care minterms 1 and 3. Minterm 5 is not included in the function. Since minterms 1, 3, and 5 were specified as being don't-care conditions, we have chosen minterms 1 and 3 to produce a 1 and minterm 5 to produce a 0. This was chosen because this assignment produces the simplest Boolean expression.

Figure 1-14 Example of map with don't-care conditions.

1-5 Combinational Circuits

Digital logic circuits are basically categorized into two types:

1. Combinational circuits in which there are no feedback paths from outputs to inputs and there is no memory.

2. Sequential circuits in which feedback paths exist from outputs to inputs, and they have memory.

block diagram

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. At any given time, the binary values of the outputs are a function of the binary combination of the inputs. A block diagram of a combinational circuit is shown in Fig. 1-15. The n binary input variables come from an external source, the m binary output variables go to an external destination, and in between there is an interconnection of logic gates. A combinational circuit transforms binary information from the given input data to the required output data. Combinational circuits are employed in digital computers for generating binary control decisions and for providing digital components required for data processing.

A combinational circuit can be described by a truth table showing the binary relationship between the n input variables and the m output variables. The truth table lists the corresponding output binary values for each of the 2^n input combinations. A combinational circuit can also be specified with m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

analysis

The analysis of a combinational circuit starts with a given logic circuit diagram and culminates with a set of Boolean functions or a truth table. If the digital

Figure 1-15 Block diagram of a combinational circuit.

circuit is accompanied by a verbal explanation of its function, the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, it is necessary to interpret the operation of the circuit from the derived Boolean functions or the truth table. The success of such investigation is enhanced if one has experience and familiarity with digital circuits. The ability to correlate a truth table or a set of Boolean functions with an information-processing task is an art that one acquires with experience.

design   The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

1. The problem is stated.
2. The input and output variables are assigned letter symbols.
3. The truth table that defines the relationship between inputs and outputs is derived.
4. The simplified Boolean functions for each output are obtained.
5. The logic diagram is drawn.

To demonstrate the design of combinational circuits, we present two examples of simple arithmetic circuits. These circuits serve as basic building blocks for the construction of more complicated arithmetic circuits.

Half-Adder

The most basic digital arithmetic circuit is the addition of two binary digits. A combinational circuit that performs the arithmetic addition of two bits is called a half-adder. One that performs the addition of three bits (two significant bits and a previous carry) is called a full-adder. The name of the former stems from the fact that two half-adders are needed to implement a full-adder.

The input variables of a half-adder are called the augend and addend bits. The output variables the sum and carry. It is necessary to specify two output variables because the sum of $1 + 1$ is binary 10, which has two digits. We assign symbols x and y to the two input variables, and S (for sum) and C (for carry) to the two output variables. The truth table for the half-adder is shown in Fig. l-16(a). The C output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs can be obtained directly from the truth table:

$$S = x'y + xy' = x \oplus y$$
$$C = xy$$

The logic diagram is shown in Fig. l-16(b). It consists of an exclusive-OR gate and an AND gate. A half-adder logic module of an exclusive-OR gate and an AND gate can be used to implement universal logic gates NAND and NOR.

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table   (b) Logic diagram

Figure 1-16   Half-adder.

Figure 1-16(c) shows the use of half-adder modules to construct NAND and NOR gates.

Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S (for sum) and C (for carry). The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is shown in Table 1-2. The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The maps of Fig. 1-17 are used to find algebraic expressions for the two output variables. The 1's in the squares for the maps of S and C are determined
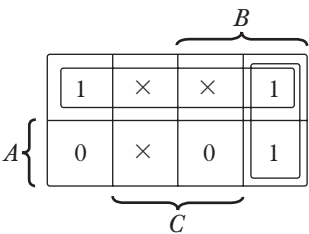
Figure 1-14   Example of map with don't-care conditions.

1-5   Combinational Circuits

Digital logic circuits are basically categorized into two types:

1. Combinational circuits in which there are no feedback paths from outputs to inputs and there is no memory.

2. Sequential circuits in which feedback paths exist from outputs to inputs, and they have memory.
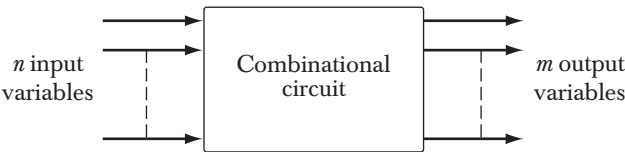
block diagram

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. At any given time, the binary values of the outputs are a function of the binary combination of the inputs. A block diagram of a combinational circuit is shown in Fig. 1-15. The n binary input variables come from an external source, the m binary output variables go to an external destination, and in between there is an interconnection of logic gates. A combinational circuit transforms binary information from the given input data to the required output data. Combinational circuits are employed in digital computers for generating binary control decisions and for providing digital components required for data processing.

A combinational circuit can be described by a truth table showing the binary relationship between the n input variables and the m output variables. The truth table lists the corresponding output binary values for each of the 2^n input combinations. A combinational circuit can also be specified with m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

analysis

The analysis of a combinational circuit starts with a given logic circuit diagram and culminates with a set of Boolean functions or a truth table. If the digital

Figure 1-15   Block diagram of a combinational circuit.

circuit is accompanied by a verbal explanation of its function, the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, it is necessary to interpret the operation of the circuit from the derived Boolean functions or the truth table. The success of such investigation is enhanced if one has experience and familiarity with digital circuits. The ability to correlate a truth table or a set of Boolean functions with an information-processing task is an art that one acquires with experience.

design   The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

1. The problem is stated.
2. The input and output variables are assigned letter symbols.
3. The truth table that defines the relationship between inputs and outputs is derived.
4. The simplified Boolean functions for each output are obtained.
5. The logic diagram is drawn.

To demonstrate the design of combinational circuits, we present two examples of simple arithmetic circuits. These circuits serve as basic building blocks for the construction of more complicated arithmetic circuits.

Half-Adder

The most basic digital arithmetic circuit is the addition of two binary digits. A combinational circuit that performs the arithmetic addition of two bits is called a half-adder. One that performs the addition of three bits (two significant bits and a previous carry) is called a full-adder. The name of the former stems from the fact that two half-adders are needed to implement a full-adder.

The input variables of a half-adder are called the augend and addend bits. The output variables the sum and carry. It is necessary to specify two output variables because the sum of $1 + 1$ is binary 10, which has two digits. We assign symbols x and y to the two input variables, and S (for sum) and C (for carry) to the two output variables. The truth table for the half-adder is shown in Fig. l-16(a). The C output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs can be obtained directly from the truth table:

$$S = x'y + xy' = x \oplus y$$
$$C = xy$$

The logic diagram is shown in Fig. l-16(b). It consists of an exclusive-OR gate and an AND gate. A half-adder logic module of an exclusive-OR gate and an AND gate can be used to implement universal logic gates NAND and NOR.
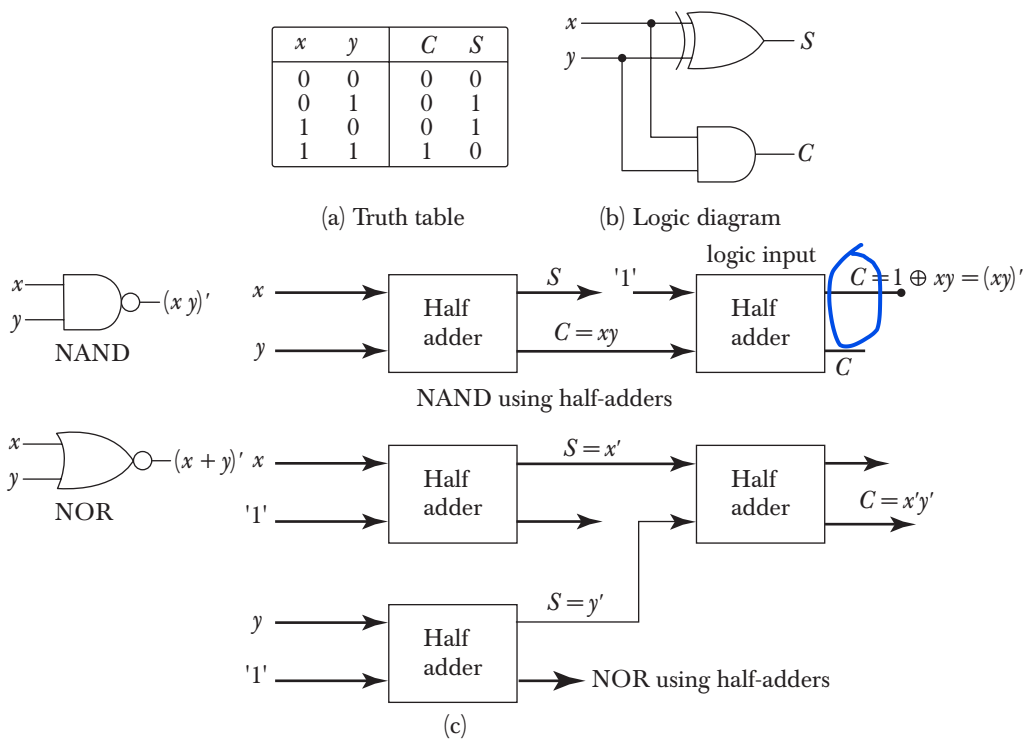
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table   (b) Logic diagram

Figure 1-16   Half-adder.

Figure 1-16(c) shows the use of half-adder modules to construct NAND and NOR gates.

Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S (for sum) and C (for carry). The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is shown in Table 1-2. The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The maps of Fig. 1-17 are used to find algebraic expressions for the two output variables. The l's in the squares for the maps of S and C are determined
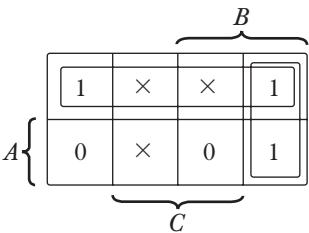
TABLE 1-2 Truth Table for Full-Adder

Inputs			Outputs	
x	y	z	C	S
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

directly from the minterms in the truth table. The squares with 1's for the S output do not combine in groups of adjacent squares. But since the output is 1 when an odd number of inputs are 1, S is an odd function and represents the exclusive-OR relation of the variables (see the discussion at the end of Sec. 1-2). The squares with 1's for the C output may be combined in a variety of ways. One possible expression for C is

$$C = xy + (x'y + xy')z$$

Realizing that $x'y + xy' = x \oplus y$ and including the expression for output S, we obtain the two Boolean expressions for the full-adder:

$$S = x \oplus y \oplus z$$
$$C = xy + (x \oplus y)z$$

The logic diagram of the full-adder is drawn in Fig. 1-18. Note that the full adder circuit consists of two half-adders and an OR gate. When used in subsequent chapters, the full-adder (FA) will be designated by a block diagram as shown in Fig. 1-18(b).

Figure 1-17   Maps for full-adder.

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$= x \oplus y \oplus z$$

$$C = xy + xz + yz$$
$$= xy + (x'y + xy')z$$

(a) Logic diagram   (b) Block diagram

Figure 1-18   Full-adder circuit.

1-6   Flip-Flops

The digital circuits considered thus far have been combinational, where the outputs at any given time are entirely dependent on the inputs that are present at that time. Although every digital system is likely to have a combinational circuit, most systems encountered in practice also include storage elements, which require that the system be described in terms of sequential circuits. The most common type of sequential circuit is the synchronous type. Synchronous sequential circuits employ signals that affect the storage elements only at discrete instants of time. Synchronization is achieved by a timing device called a clock pulse generator that produces a periodic train of *clock pulses*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of the synchronization pulse. Clocked synchronous sequential circuits are the type most frequently encountered in practice. They seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which may be considered separately.

clocked sequential circuit   The storage elements employed in clocked sequential circuits are called flip-flops. A flip-flop is a binary cell capable of storing one bit of information. It has two outputs, one for the normal value and one for the complement value of the bit stored in it. A flip-flop maintains a binary state until directed by a clock pulse to switch states. The difference among various types of flip-flops is in the number of inputs they possess and in the manner in which the inputs affect the binary state. The most common types of flip-flops are presented below.

SR Flip-Flop

The graphic symbol of the *SR* flip-flop is shown in Fig. 1-19(a). It has three inputs, labeled *S* (for set), *R* (for reset), and *C* (for clock). It has an output *Q* and sometimes the flip-flop has a complemented output, which is indicated with a small circle at the other output terminal. There is an arrowhead-shaped symbol in front of the letter *C* to designate a *dynamic input*. The dynamic indicator symbol denotes the fact that the flip-flop responds to a positive transition (from 0 to 1) of the input clock signal.

The operation of the *SR* flip-flop is as follows. If there is no signal at the clock input *C*, the output of the circuit cannot change irrespective of the values at inputs

S	R	Q(t+1)	
0	0	Q (t)	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

(a) Graphic symbol   (b) Characteristic table

Figure 1-19   *SR* flip-flop.

S and *R*. Only when the clock signal changes from 0 to 1 can the output be affected according to the values in inputs *S* and *R*. If $S = 1$ and $R = 0$ when *C* changes from 0 to 1, output *Q* is set to 1. If $S = 0$ and $R = 1$ when *C* changes from 0 to 1, output *Q* is cleared to 0. If both *S* and *R* are 0 during the clock transition, the output does not change. When both *S* and *R* are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit.

The characteristic table shown in Fig. l-19(b) summarizes the operation of the *SR* flip-flop in tabular form. The *S* and *R* columns give the binary values of the two inputs. $Q(t)$ is the binary state of the *Q* output at a given time (referred to as *present state*). $Q(t + 1)$ is the binary state of the *Q* output after the occurrence of a clock transition (referred to as next state). If $S = R = 0$, a clock transition produces no change of state [i.e., $Q(t + 1) = Q(t)$]. If $S = 0$ and $R = 1$, the flip-flop goes to the 0 (clear) state. If $S = 1$ and $R = 0$, the flip-flop goes to the 1 (set) state. The *SR* flip-flop should not be pulsed when $S = R = 1$ since it produces an indeterminate next state. This indeterminate condition makes the *SR* flip-flop difficult to manage and therefore it is seldom used in practice.

D Flip-Flop

The *D* (data) flip-flop is a slight modification of the *SR* flip-flop. An *SR* flip-flop is converted to a *D* flip-flop by inserting an inverter between *S* and *R* and assigning the symbol *D* to the single input. The *D* input is sampled during the occurrence of a clock transition from 0 to 1. If $D = 1$, the output of the flip-flop goes to the 1 state, but if $D = 0$, the output of the flip-flop goes to the 0 state.

The graphic symbol and characteristic table of the *D* flip-flop are shown in Fig. 1-20. From the characteristic table we note that the next state $Q(t + 1)$ is

Figure 1-20   *D* flip-flop

D	Q(t+1)	
0	0	Clear to 0
1	1	Set to 1

(a) Graphic symbol   (b) Characteristic table

determined from the D input. The relationship can be expressed by a characteristic equation:

$$Q(t + 1) = D$$

This means that the Q output of the flip-flop receives its value from the D input every time that the clock signal goes through a transition from 0 to 1.

Note that no input condition exists that will leave the state of the D flip-flop unchanged. Although a D flip-flop has the advantage of having only one input (excluding C), it has the disadvantage that its characteristic table does not have a "no change" condition $Q(t + 1) = Q(t)$. The "no change" condition can be accomplished either by disabling the clock signal or by feeding the output back into the input, so that clock pulses keep the state of the flip-flop unchanged.

JK Flip-Flop

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate condition of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state.

The graphic symbol and characteristic table of the JK flip-flop are shown in Fig. 1-21. The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input. Instead of the indeterminate condition, the JK flip-flop has a complement condition $Q(t + 1) = Q'(t)$ when both J and K are equal to 1.

T Flip-Flop

Another type of flip-flop found in textbooks is the T (toggle) flip-flop. This flip-flop, shown in Fig. 1-22, is obtained from a JK type when inputs J and K are connected to provide a single input designated by T. The T flip-flop therefore has only two conditions. When $T = 0$ ($J = K = 0$) a clock transition does not change the state of

Figure 1-21 JK flip-flop

J	K	$Q(t + 1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	$Q'(t)$	Complement

(a) Graphic symbol   (b) Characteristic table

(a) Logic diagram   (b) Block diagram
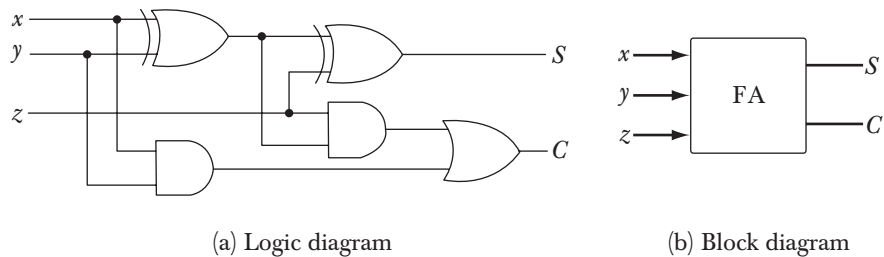
Figure 1-18   Full-adder circuit.

1-6   Flip-Flops

The digital circuits considered thus far have been combinational, where the outputs at any given time are entirely dependent on the inputs that are present at that time. Although every digital system is likely to have a combinational circuit, most systems encountered in practice also include storage elements, which require that the system be described in terms of sequential circuits. The most common type of sequential circuit is the synchronous type. Synchronous sequential circuits employ signals that affect the storage elements only at discrete instants of time. Synchronization is achieved by a timing device called a clock pulse generator that produces a periodic train of *clock pulses*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of the synchronization pulse. Clocked synchronous sequential circuits are the type most frequently encountered in practice. They seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which may be considered separately.

clocked sequential circuit

The storage elements employed in clocked sequential circuits are called flip-flops. A flip-flop is a binary cell capable of storing one bit of information. It has two outputs, one for the normal value and one for the complement value of the bit stored in it. A flip-flop maintains a binary state until directed by a clock pulse to switch states. The difference among various types of flip-flops is in the number of inputs they possess and in the manner in which the inputs affect the binary state. The most common types of flip-flops are presented below.

SR Flip-Flop

The graphic symbol of the *SR* flip-flop is shown in Fig. 1-19(a). It has three inputs, labeled S (for set), R (for reset), and C (for clock). It has an output Q and sometimes the flip-flop has a complemented output, which is indicated with a small circle at the other output terminal. There is an arrowhead-shaped symbol in front of the letter C to designate a *dynamic input*. The dynamic indicator symbol denotes the fact that the flip-flop responds to a positive transition (from 0 to 1) of the input clock signal.

The operation of the *SR* flip-flop is as follows. If there is no signal at the clock input C, the output of the circuit cannot change irrespective of the values at inputs
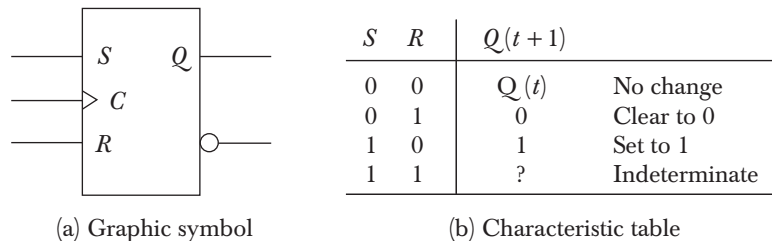
S	R	Q(t + 1)	
0	0	Q(t)	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

(a) Graphic symbol   (b) Characteristic table

Figure 1-19   *SR* flip-flop.

S and *R*. Only when the clock signal changes from 0 to 1 can the output be affected according to the values in inputs *S* and *R*. If $S = 1$ and $R = 0$ when *C* changes from 0 to 1, output *Q* is set to 1. If $S = 0$ and $R = 1$ when *C* changes from 0 to 1, output *Q* is cleared to 0. If both *S* and *R* are 0 during the clock transition, the output does not change. When both *S* and *R* are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit.
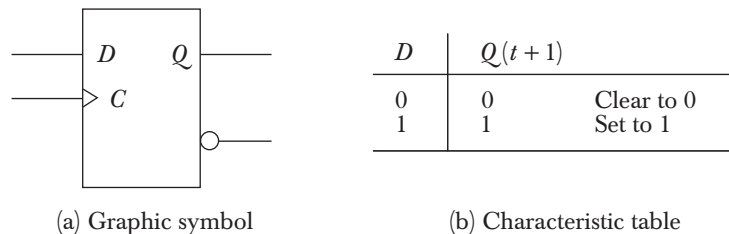
The characteristic table shown in Fig. l-19(b) summarizes the operation of the *SR* flip-flop in tabular form. The *S* and *R* columns give the binary values of the two inputs. $Q(t)$ is the binary state of the *Q* output at a given time (referred to as *present state*). $Q(t + 1)$ is the binary state of the *Q* output after the occurrence of a clock transition (referred to as next state). If $S = R = 0$, a clock transition produces no change of state [i.e., $Q(t + 1) = Q(t)$]. If $S = 0$ and $R = 1$, the flip-flop goes to the 0 (clear) state. If $S = 1$ and $R = 0$, the flip-flop goes to the 1 (set) state. The *SR* flip-flop should not be pulsed when $S = R = 1$ since it produces an indeterminate next state. This indeterminate condition makes the *SR* flip-flop difficult to manage and therefore it is seldom used in practice.

D Flip-Flop

The *D* (data) flip-flop is a slight modification of the *SR* flip-flop. An *SR* flip-flop is converted to a *D* flip-flop by inserting an inverter between *S* and *R* and assigning the symbol *D* to the single input. The *D* input is sampled during the occurrence of a clock transition from 0 to 1. If $D = 1$, the output of the flip-flop goes to the 1 state, but if $D = 0$, the output of the flip-flop goes to the 0 state.

The graphic symbol and characteristic table of the *D* flip-flop are shown in Fig. 1-20. From the characteristic table we note that the next state $Q(t + 1)$ is

Figure 1-20   *D* flip-flop

D	Q(t + 1)	
0	0	Clear to 0
1	1	Set to 1

(a) Graphic symbol   (b) Characteristic table

determined from the D input. The relationship can be expressed by a characteristic equation:

$$Q(t + 1) = D$$

This means that the Q output of the flip-flop receives its value from the D input every time that the clock signal goes through a transition from 0 to 1.

Note that no input condition exists that will leave the state of the D flip-flop unchanged. Although a D flip-flop has the advantage of having only one input (excluding C), it has the disadvantage that its characteristic table does not have a "no change" condition $Q(t + 1) = Q(t)$. The "no change" condition can be accomplished either by disabling the clock signal or by feeding the output back into the input, so that clock pulses keep the state of the flip-flop unchanged.
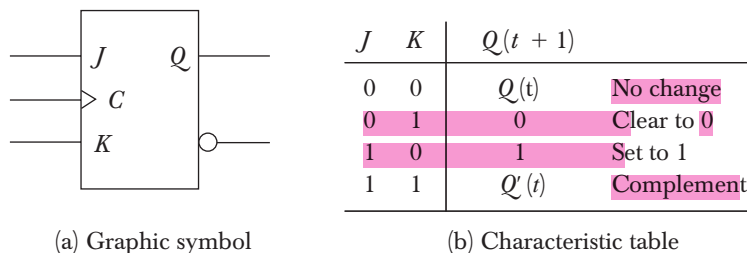
JK Flip-Flop

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate condition of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state.

The graphic symbol and characteristic table of the JK flip-flop are shown in Fig. 1-21. The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input. Instead of the indeterminate condition, the JK flip-flop has a complement condition $Q(t + 1) = Q'(t)$ when both J and K are equal to 1.

T Flip-Flop

Another type of flip-flop found in textbooks is the T (toggle) flip-flop. This flip-flop, shown in Fig. 1-22, is obtained from a JK type when inputs J and K are connected to provide a single input designated by T. The T flip-flop therefore has only two conditions. When $T = 0$ ($J = K = 0$) a clock transition does not change the state of
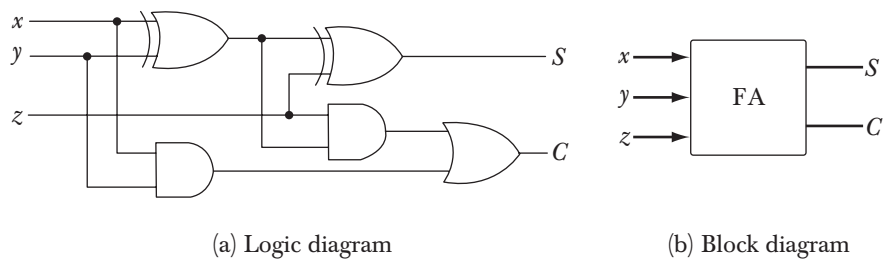
Figure 1-21 JK flip-flop

J	K	$Q(t + 1)$	
0	0	$Q(t)$	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	$Q'(t)$	Complement

(a) Graphic symbol   (b) Characteristic table

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

(a) Graphic symbol   (b) Characteristic table

Figure 1-22 T flip-flop

the flip-flop. When $T = 1$ ($J = K = 1$) a clock transition complements the state of the flip-flop. These conditions can be expressed by a characteristic equation:

$$Q(t + 1) = Q(t) \oplus T$$

Edge-Triggered Flip-Flops

The most common type of flip-flop used to synchronize the state change during a clock pulse transition is the edge-triggered flip-flop. In this type of flip-flop, output transitions occur at a specific level of the clock pulse. When the pulse input level exceeds this threshold level, the inputs are locked out so that the flip-flop is unresponsive to further changes in inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the rising edge of the clock signal (positive-edge transition), and others cause a transition on the falling edge (negative-edge transition).

clock pulses

    Figure l-23(a) shows the clock pulse signal in a positive-edge-triggered D flip-flop. The value in the D input is transferred to the Q output when the clock makes a positive transition. The output cannot change when the clock is in the 1 level, in the 0 level, or in a transition from the 1 level to the 0 level. The effective positive clock transition includes a minimum time called the setup time in which the D input must remain at a constant value before the transition, and a definite time called the *hold time* in which the D input must not change after the positive transition. The effective positive transition is usually a very small fraction of the total period of the clock pulse.

    Figure l-23(b) shows the corresponding graphic symbol and timing diagram for a negative-edge-triggered D flip-flop. The graphic symbol includes a negation small circle in front of the dynamic indicator at the C input. This denotes a negative-edge-triggered behavior. In this case the flip-flop responds to a transition from the 1 level to the 0 level of the clock signal.

master-slave flip-flop

    Another type of flip-flop used in some systems is the master-slave flip-flop. This type of circuit consists of two flip-flops. The first is the master, which responds to the positive level of the clock, and the second is the slave, which responds to the negative level of the clock. The result is that the output changes

(a) Positive-edge-triggered *D* flip-flop

(b) Negative-edge-triggered *D* flip-flop

Figure 1-23   Edge-triggered flip-flop.

during the l-to-0 transition of the clock signal. The trend is away from the use of master-slave flip-flops and toward edge-triggered flip-flops.

Flip-flops available in integrated circuit packages will sometimes provide special input terminals for setting or clearing the flip-flop asynchronously. These inputs are usually called "preset" and "clear." They affect the flip-flop on a negative level of the input signal without the need of a clock pulse. These inputs are useful for bringing the flip-flops to an initial state prior to its clocked operation.

Excitation Tables

The characteristic tables of flip-flops specify the next state when the inputs and the present state are known. During the design of sequential circuits we usually know the required transition from present state to next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason we need a table that lists the required input combinations for a given change of state. Such a table is called a flip-flop excitation table.

Table 1-3 lists the excitation tables for the four types of flip-flops. Each table consists of two columns, $Q(t)$ and $Q(t + 1)$, and a column for each input to show how the required transition is achieved. There are four possible transitions from present state $Q(t)$ to next state $Q(t + 1)$. The required input conditions for each of these transitions are derived from the information available in the characteristic tables. The symbol $\times$ in the tables represents a don't-care condition; that is, it does not matter whether the input to the flip-flop is 0 or 1.

TABLE 1-3 Excitation Table for Four Flip-Flops

$Q(t)$	$Q(t + 1)$	S	R	$Q(t)$	$Q(t + 1)$	D
	SR flip-flop				*D* flip-flop	
0	0	0	×	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	0
1	1	×	0	1	1	1

$Q(t)$	$Q(t + 1)$	J	K	$Q(t)$	$Q(t + 1)$	T
	JK flip-flop				*T* flip-flop	
0	0	0	×	0	0	0
0	1	1	×	0	1	1
1	0	×	1	1	0	1
1	1	×	0	1	1	0

The reason for the don't-care conditions in the excitation tables is that there are two ways of achieving the required transition. For example, in a *JK* flip-flop, a transition from present state of 0 to a next state of 0 can be achieved by having inputs *J* and *K* equal to 0 (to obtain no change) or by letting $J = 0$ and $K = 1$ to clear the flip-flop (although it is already cleared). In both cases *J* must be 0, but *K* is 0 in the first case and 1 in the second. Since the required transition will occur in either case, we mark the *K* input with a don't-care × and let the designer choose either 0 or 1 for the *K* input, whichever is more convenient.

1-7   Sequential Circuits

A sequential circuit is an interconnection of flip-flops and gates. The gates by themselves constitute a combinational circuit, but when included with the flip-flops, the overall circuit is classified as a sequential circuit. The block diagram of a clocked sequential circuit is shown in Fig. 1-24. It consists of a combinational

Figure 1-24   Block diagram of a clocked synchronous sequential circuit.

circuit and a number of clocked flip-flops. In general, any number or type of flip-flops may be included. As shown in the diagram, the combinational circuit block receives binary signals from external inputs and from the outputs of flip-flops. The outputs of the combinational circuit go to external outputs and to inputs of flip-flops. The gates in the combinational circuit determine the binary value to be stored in the flip-flops after each clock transition. The outputs of flip-flops, in turn, are applied to the combinational circuit inputs and determine the circuit's behavior. This process demonstrates that the external outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops. Moreover, the next state of flip-flops is also a function of their present state and external inputs. Thus a sequential circuit is specified by a time sequence of external inputs, external outputs, and internal flip-flop binary states.

Flip-Flop Input Equations

An example of a sequential circuit is shown in Fig. 1-25. It has one input variable x, one output variable y, and two clocked D flip-flops. The AND gates, OR gates, and inverter form the combinational logic part of the circuit. The interconnections

Figure 1-25   Example of a sequential circuit

input equation

among the gates in the combinational circuit can be specified by a set of Boolean expressions. The part of the combinational circuit that generates the inputs to flip-flops are described by a set of Boolean expressions called flip-flop input equations. We adopt the convention of using the flip-flop input symbol to denote the input equation variable name and a subscript to designate the symbol chosen for the output of the flip-flop. Thus, in Fig. 1-25, we have two input equations, designated D_A and D_B. The first letter in each symbol denotes the D input of a D flip-flop. The subscript letter is the symbol name of the flip-flop. The input equations are Boolean functions for flip-flop input variables and can be derived by inspection of the circuit. Since the output of the OR gate is connected to the D input of flip-flop A, we write the first input equation as

$$D_A = Ax + Bx$$

where A and B are the outputs of the two flip-flops and x is the external input. The second input equation is derived from the single AND gate whose output is connected to the D input of flip-flop B:

$$D_B = A'x$$

The sequential circuit also has an external output, which is a function of the input variable and the state of the flip-flops. This output can be specified algebraically by the expression

$$y = Ax' + Bx'$$

From this example we note that a flip-flop input equation is a Boolean expression for a combinational circuit. The subscripted variable is a binary variable name for the output of a combinational circuit. This output is always connected to a flip-flop input.

State Table

present state

next state

The behavior of a sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. Both the outputs and the next state are a function of the inputs and the present state. A sequential circuit is specified by a state table that relates outputs and next states as a function of inputs and present states. In clocked sequential circuits, the transition from present state to next state is activated by the presence of a clock signal.

The state table for the circuit of Fig. 1-25 is shown in Table 1-4. The table consists of four sections, labeled *present state, input, next state,* and *output*. The present-state section shows the states of flip-flops A and B at any given time t. The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock period later at time $t + 1$. The output section gives the value of y for each present state and input condition.

(a) Positive-edge-triggered D flip-flop

(b) Negative-edge-triggered D flip-flop

Figure 1-23 Edge-triggered flip-flop.

during the l-to-0 transition of the clock signal. The trend is away from the use of master-slave flip-flops and toward edge-triggered flip-flops.

Flip-flops available in integrated circuit packages will sometimes provide special input terminals for setting or clearing the flip-flop asynchronously. These inputs are usually called "preset" and "clear." They affect the flip-flop on a negative level of the input signal without the need of a clock pulse. These inputs are useful for bringing the flip-flops to an initial state prior to its clocked operation.

Excitation Tables

The characteristic tables of flip-flops specify the next state when the inputs and the present state are known. During the design of sequential circuits we usually know the required transition from present state to next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason we need a table that lists the required input combinations for a given change of state. Such a table is called a flip-flop excitation table.

Table 1-3 lists the excitation tables for the four types of flip-flops. Each table consists of two columns, $Q(t)$ and $Q(t + 1)$, and a column for each input to show how the required transition is achieved. There are four possible transitions from present state $Q(t)$ to next state $Q(t + 1)$. The required input conditions for each of these transitions are derived from the information available in the characteristic tables. The symbol $\times$ in the tables represents a don't-care condition; that is, it does not matter whether the input to the flip-flop is 0 or 1.

TABLE 1-3 Excitation Table for Four Flip-Flops

$Q(t)$	$Q(t+1)$	S	R		$Q(t)$	$Q(t+1)$	D
	SR flip-flop					*D* flip-flop	
0	0	0	$\times$		0	0	0
0	1	1	0		0	1	1
1	0	0	1		1	0	0
1	1	$\times$	0		1	1	1

$Q(t)$	$Q(t+1)$	J	K		$Q(t)$	$Q(t+1)$	T
	JK flip-flop					*T* flip-flop	
0	0	0	$\times$		0	0	0
0	1	1	$\times$		0	1	1
1	0	$\times$	1		1	0	1
1	1	$\times$	0		1	1	0

The reason for the don't-care conditions in the excitation tables is that there are two ways of achieving the required transition. For example, in a *JK* flip-flop, a transition from present state of 0 to a next state of 0 can be achieved by having inputs *J* and *K* equal to 0 (to obtain no change) or by letting $J = 0$ and $K = 1$ to clear the flip-flop (although it is already cleared). In both cases *J* must be 0, but *K* is 0 in the first case and 1 in the second. Since the required transition will occur in either case, we mark the *K* input with a don't-care $\times$ and let the designer choose either 0 or 1 for the *K* input, whichever is more convenient.

1-7 Sequential Circuits

A sequential circuit is an interconnection of flip-flops and gates. The gates by themselves constitute a combinational circuit, but when included with the flip-flops, the overall circuit is classified as a sequential circuit. The block diagram of a clocked sequential circuit is shown in Fig. 1-24. It consists of a combinational

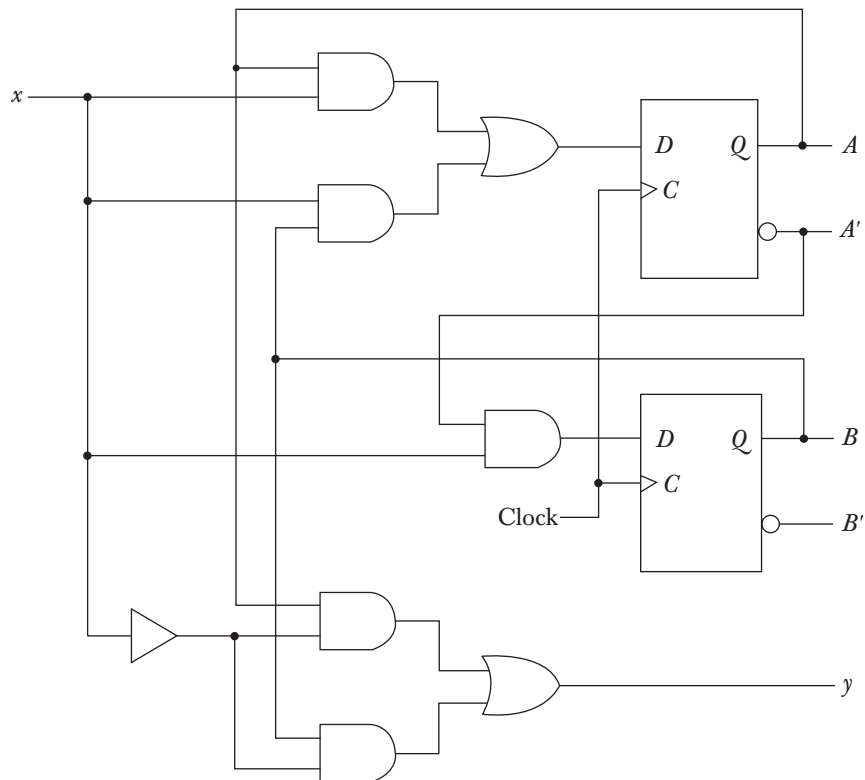**Figure 1-24** Block diagram of a clocked synchronous sequential circuit.

circuit and a number of clocked flip-flops. In general, any number or type of flip-flops may be included. As shown in the diagram, the combinational circuit block receives binary signals from external inputs and from the outputs of flip-flops. The outputs of the combinational circuit go to external outputs and to inputs of flip-flops. The gates in the combinational circuit determine the binary value to be stored in the flip-flops after each clock transition. The outputs of flip-flops, in turn, are applied to the combinational circuit inputs and determine the circuit's behavior. This process demonstrates that the external outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops. Moreover, the next state of flip-flops is also a function of their present state and external inputs. Thus a sequential circuit is specified by a time sequence of external inputs, external outputs, and internal flip-flop binary states.

Flip-Flop Input Equations

An example of a sequential circuit is shown in Fig. 1-25. It has one input variable x, one output variable y, and two clocked D flip-flops. The AND gates, OR gates, and inverter form the combinational logic part of the circuit. The interconnections

Figure 1-25   Example of a sequential circuit

among the gates in the combinational circuit can be specified by a set of Boolean expressions. The part of the combinational circuit that generates the inputs to flip-flops are described by a set of Boolean expressions called flip-flop input equations. We adopt the convention of using the flip-flop input symbol to denote the input equation variable name and a subscript to designate the symbol chosen for the output of the flip-flop. Thus, in Fig. 1-25, we have two input equations, designated D_A and D_B. The first letter in each symbol denotes the D input of a D flip-flop. The subscript letter is the symbol name of the flip-flop. The input equations are Boolean functions for flip-flop input variables and can be derived by inspection of the circuit. Since the output of the OR gate is connected to the D input of flip-flop A, we write the first input equation as

input equation

$$D_A = Ax + Bx$$

where A and B are the outputs of the two flip-flops and x is the external input. The second input equation is derived from the single AND gate whose output is connected to the D input of flip-flop B:

$$D_B = A'x$$

The sequential circuit also has an external output, which is a function of the input variable and the state of the flip-flops. This output can be specified algebraically by the expression

$$y = Ax' + Bx'$$

From this example we note that a flip-flop input equation is a Boolean expression for a combinational circuit. The subscripted variable is a binary variable name for the output of a combinational circuit. This output is always connected to a flip-flop input.
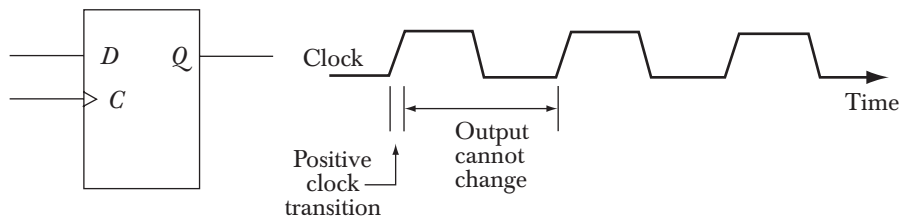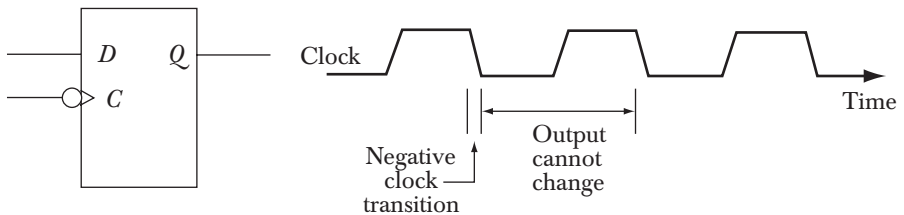
State Table

The behavior of a sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. Both the outputs and the next state are a function of the inputs and the present state. A sequential circuit is specified by a state table that relates outputs and next states as a function of inputs and present states. In clocked sequential circuits, the transition from present state to next state is activated by the presence of a clock signal.

present state

The state table for the circuit of Fig. 1-25 is shown in Table 1-4. The table consists of four sections, labeled *present state, input, next state,* and *output*. The present-state section shows the states of flip-flops A and B at any given time t. The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock period later at time $t + 1$. The output section gives the value of y for each present state and input condition.

next state

Unit 2---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

Family	Speed (n sec)	Power Dissipation (m watts)	Faxout	Polar Supply (V)	High Logic Level (V)	Low Logic Level (V)
TTL	10	10	10	$+5$	$+3$	0.2
ECL	2	40	high	-5.2	-0.9	$+1.75$
CMOS	25	low	high	3–15	V_{cc}	0

Figure 2-1   Comparison of the basic logic families.

The metal-oxide semiconductor (MOS) is a unipolar transistor that depends on the flow of only one type of carrier, which may be electrons (n-channel) or holes (p-channel). This is in contrast to the bipolar transistor used in TTL and ECL gates, where both carriers exist during normal operation. A p-channel MOS is referred to as PMOS and an n-channel as NMOS. NMOS is the one that is commonly used in circuits with only one type of MOS transistor. The complementary MOS (CMOS) technology uses PMOS and NMOS transistors connected in a complementary fashion in all circuits. The most important advantages of CMOS over bipolar are the high packing density of circuits, a simpler processing technique during fabrication, and a more economical operation because of low power consumption. Figure 2-1 is a comparison of the basic logic families.

Because of their many advantages, integrated circuits are used exclusively to provide various digital components needed in the design of computer systems. To understand the organization and design of digital computers it is very important to be familiar with the various components encountered in integrated circuits. For this reason, the most basic components are introduced in this chapter with an explanation of their logical properties. These components provide a catalog of elementary digital functional units commonly used as basic building blocks in the design of digital computers.

2-2   Decoders

Discrete quantities of information are represented in digital computers with binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. If the n-bit coded information has unused bit combinations, the decoder may have less than 2^n outputs.

The decoders presented in this section are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) binary combinations of the n input variables. A decoder has n inputs and m outputs and is also referred to as an $n \times m$ decoder.
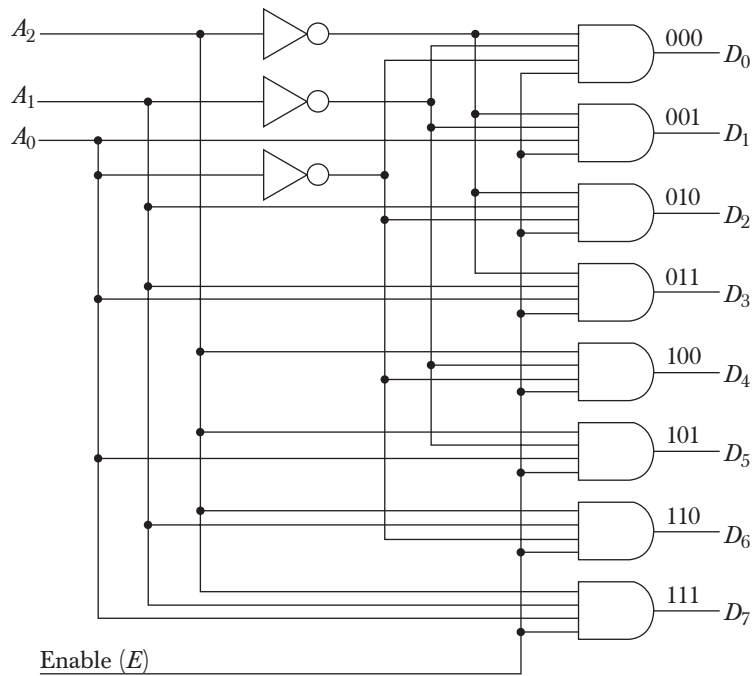
Figure 2-2   3-to-8-line decoder.

The logic diagram of a 3-to-8-line decoder is shown in Fig. 2-2. The three data inputs, A_0, A_1, and A_2 are decoded into eight outputs, each output representing one of the combinations of the three binary input variables. The three inverters provide the complement of the inputs, and each of the eight AND gates generates one of the binary combination. A particular application of this decoder is a binary-to-octal conversion. The input variables represent a binary number and the outputs represent the eight digits of the octal number system. However, a 3-to-8-line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each combination of the binary code.

Enable input   Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of Fig. 2-2 has one enable input, E. The decoder is enabled when E is equal to 1 and disabled when E is equal to 0.

The operation of the decoder can be clarified using the truth table listed in Table 2-1. When the enable input E is equal to 0, all the outputs are equal to 0 regardless of the values of the other three data inputs. The three ×'s in the table designate don't-care conditions. When the enable input is equal to 1, the decoder operates in a normal fashion. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output variable whose value is equal to 1 represents the octal number equivalent of the binary number that is available in the input data lines.

TABLE 2-1 Truth Table for 3-to-8-Line Decoder

Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

NAND Gate Decoder

Some decoders are constructed with NAND instead of AND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder outputs in their complement form. A 2-to-4-line decoder with an enable input constructed with NAND gates is shown in-Fig. 2-3. The circuit operates with complemented outputs and a complemented enable input E. The decoder is enabled when E is equal to 0. As indicated by the truth table, only one output is equal to 0 at any given time; the other three outputs are equal to 1. The output whose value is equal to 0 represents the equivalent binary number in inputs A_1 and A_0. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are selected and all outputs are equal to 1. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal level. Some decoders have two or

Figure 2-3 2-to-4-line decoder with NAND gates.

E	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	x	x	1	1	1	1

(a) Logic diagram

(b) Truth table
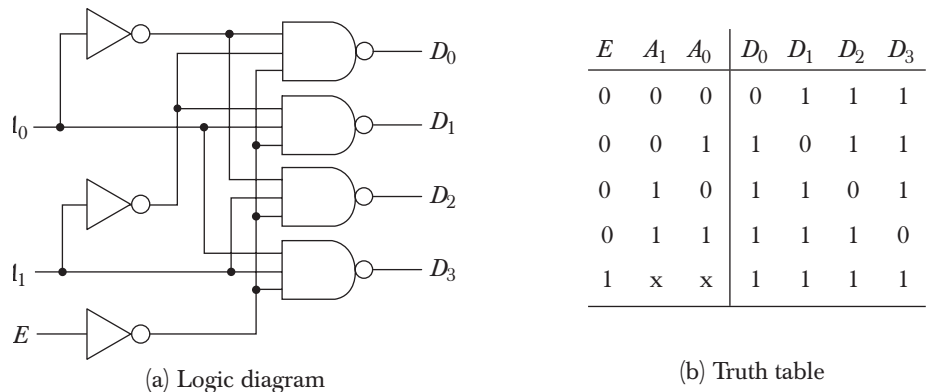
more enable inputs that must satisfy a given logic condition in order to enable the circuit.

Decoder Expansion

There are occasions when a certain-size decoder is needed but only smaller sizes are available. When this occurs it is possible to combine two or more decoders with enable inputs to form a larger decoder. Thus if a 6-to-64-line decoder is needed, it is possible to construct it with four 4-to-16-line decoders.

Figure 2-4 shows how decoders with enable inputs can be connected to form a larger decoder. Two 2-to-4-line decoders are combined to achieve a 3-to-8-line decoder. The two least significant bits of the input are connected to both decoders. The most significant bit is connected to the enable input of one decoder and through an inverter to the enable input of the other decoder. It is assumed that each decoder is enabled when its E input is equal to 1. When E is equal to 0, the decoder is disabled and all its outputs are in the 0 level. When $A_2 = 0$, the upper decoder is enabled and the lower is disabled. The lower decoder outputs become inactive with all outputs at 0. The outputs of the upper decoder generate outputs D_0 through D_3, depending on the values of A_1 and A_0 (while $A_2 = 0$). When $A_2 = 1$, the lower decoder is enabled and the upper is disabled. The lower decoder output generates the binary equivalent D_4 through D_7 since these binary numbers have a 1 in the A_2 position.

The example demonstrates the usefulness of the enable input in decoders or any other combinational logic component. Enable inputs are a convenient feature for interconnecting two or more circuits for the purpose of expanding the digital component into a similar function but with more inputs and outputs.

Figure 2-4   A 3×8 decoder constructed with two 2×4 decoders.

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or less) input lines and n outputs lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder, whose truth table is given in Table 2-2. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise, the circuit has no meaning.

TABLE 2-2 Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output $A_0 = 1$ if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for the other two outputs. These conditions can be expressed by the following Boollean functions:

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

2-3 Multiplexers

multiplexer

A multiplexer is a combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line. The selection of a particular input data line for the output is determined by a set of selection inputs. A 2^n-to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.

A 4-to-l-line multiplexer is shown in Fig. 2-5. Each of the four data inputs I_0 through I_3 is applied to one input of an AND gate. The two selection inputs S_1 and

Figure 2-5 4-to-1-line multiplexer.

S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate to provide the single output. To demonstrate the circuit operation, consider the case when $S_1 S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1. The third input of the gate is connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output.

The 4-to-l line multiplexer of Fig. 2-5 has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 2^6 binary combinations. This is an excessively long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in Table 2-3. The table demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs S_1 and S_0. When the selection inputs

TABLE 2-3 Function Table for 4-to-1-Line Multiplexer

Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

are equal to 00, output Y is equal to input I_0. When the selection inputs are equal to 01, input I_1 has a path to output Y, and similarly for the other two combinations.

data selector

The multiplexer is also called a *data selector,* since it selects one of many data inputs and steers the binary information to the output.

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed they decode the input selection lines. In general, a 2^n-to-l-line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one from each data input. The size of the multiplexer is specified by the number 2^n of its data inputs and the single output. It is then implied that it also contains n input selection lines. The multiplexer is often abbreviated as MUX.

As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for expanding two or more multiplexers to a multiplexer with a larger number of inputs.

In some cases two or more multiplexers are enclosed within a single integrated circuit package. The selection and the enable inputs in multiple-unit construction are usually common to all multiplexers. As an illustration, the block diagram of a quadruple 2-to-l-line multiplexer is shown in Fig. 2-6. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or B_0. Similarly, output Y_1 may have the value of A_1 or B_1 and so on. One input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal

Figure 2-6   Quadruple 2-to-1 line multiplexers.

E	S	Y
0	x	All 0's
1	0	A
1	1	B

(b) Function table

(a) Block diagram

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or less) input lines and n outputs lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder, whose truth table is given in Table 2-2. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise, the circuit has no meaning.

TABLE 2-2 Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output $A_0 = 1$ if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for the other two outputs. These conditions can be expressed by the following Boollean functions:

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

2-3   Multiplexers

multiplexer

A multiplexer is a combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line. The selection of a particular input data line for the output is determined by a set of selection inputs. A 2^n-to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.

A 4-to-l-line multiplexer is shown in Fig. 2-5. Each of the four data inputs I_0 through I_3 is applied to one input of an AND gate. The two selection inputs S_1 and
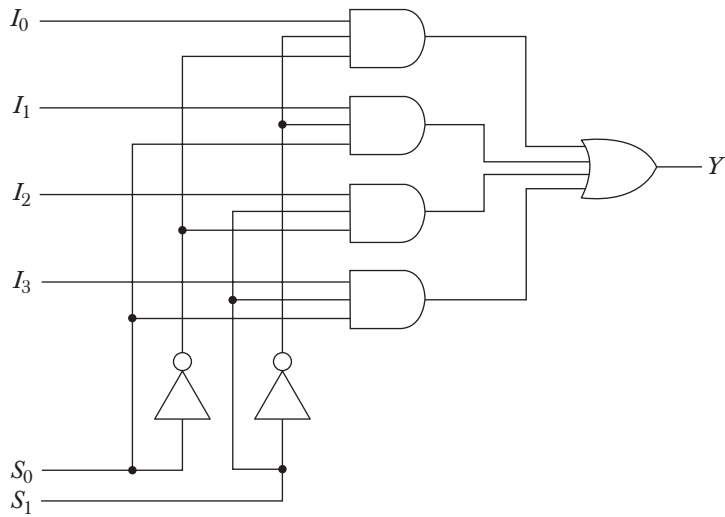
Figure 2-5 4-to-1-line multiplexer.

S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate to provide the single output. To demonstrate the circuit operation, consider the case when $S_1 S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1. The third input of the gate is connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output.

The 4-to-l line multiplexer of Fig. 2-5 has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 2^6 binary combinations. This is an excessively long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in Table 2-3. The table demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs S_1 and S_0. When the selection inputs

TABLE 2-3 Function Table for 4-to-1-Line Multiplexer

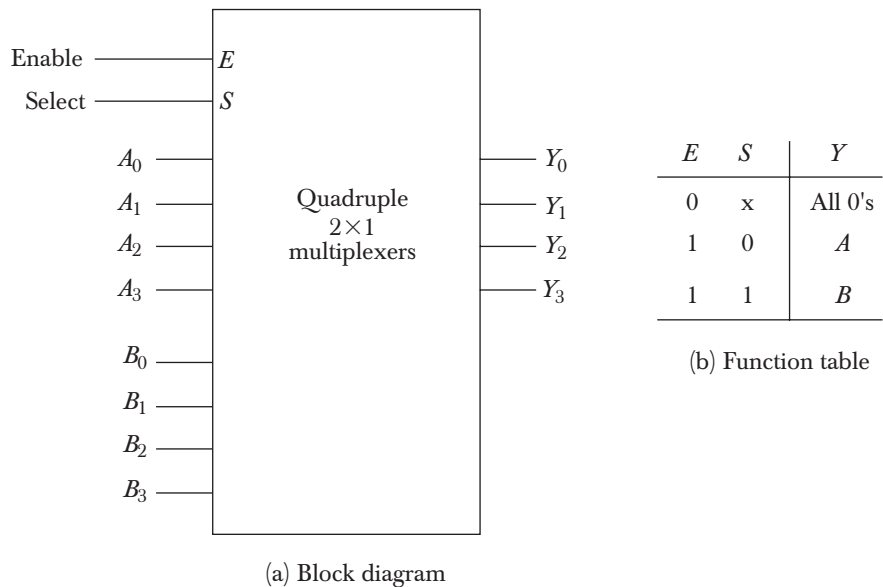| Select		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

are equal to 00, output Y is equal to input I_0. When the selection inputs are equal to 01, input I_1 has a path to output Y, and similarly for the other two combinations. The multiplexer is also called a *data selector,* since it selects one of many data inputs and steers the binary information to the output.

data selector

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed they decode the input selection lines. In general, a 2^n-to-l-line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one from each data input. The size of the multiplexer is specified by the number 2^n of its data inputs and the single output. It is then implied that it also contains n input selection lines. The multiplexer is often abbreviated as MUX.

As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for expanding two or more multiplexers to a multiplexer with a larger number of inputs.

In some cases two or more multiplexers are enclosed within a single integrated circuit package. The selection and the enable inputs in multiple-unit construction are usually common to all multiplexers. As an illustration, the block diagram of a quadruple 2-to-l-line multiplexer is shown in Fig. 2-6. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or B_0. Similarly, output Y_1 may have the value of A_1 or B_1 and so on. One input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal

Figure 2-6   Quadruple 2-to-1 line multiplexers.

E	S	Y
0	x	All 0's
1	0	A
1	1	B

(b) Function table

(a) Block diagram

operation. Although the circuit contains four multiplexers, we can also think of it as a circuit that selects one of two 4-bit data lines. As shown in the function table, the unit is enabled when $E = 1$. Then, if $S = 0$, the four A inputs have a path to the four outputs. On the other hand, if $S = 1$, the four B inputs are applied to the outputs. The outputs have all 0's when $E = 0$, regardless of the values of S.

Typical applications of multiplexers are data routing, parallel-to-serial conversion, and logic function generation. An n-variable logic function can be generated using n-select inputs of a multiplexer. Digital Multiplexers are thus considered universal logic modules.

2-4   Registers

A register is a group of flip-flops with each flip-flop capable of storing one bit of information. An n-bit register has a group of n flip-flops and is capable of storing any binary information of n bits. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops and gates that effect their transition. The flip-flops hold the binary information and the gates control when and how new information is transferred into the register.

Various types of registers are available commercially. The simplest register is one that consists only of flip-flops, with no external gates. Figure 2-7 shows such a register constructed with four D flip-flops. The common clock input triggers all flip-flops on the rising edge of each pulse, and the binary data available at the four inputs are transferred into the 4-bit register. The four outputs can be sampled at any time to obtain the binary information stored in the register. The *clear* input goes to a special terminal in each flip-flop. When this input goes to 0, all flip-flops are reset asynchronously. The clear input is useful for clearing the register to all 0's prior to its clocked operation. The clear input must be maintained at logic 1 during normal clocked operation. Note that the clock signal enables the D input but that the clear input is independent of the clock.

register load   The transfer of new information into a register is referred to as *loading* the register. If all the bits of the register are loaded simultaneously with a common clock pulse transition, we say that the loading is done in parallel. A clock transition applied to the C inputs of the register of Fig. 2-7 will load all four inputs I_0 through I_3 in parallel. In this configuration, the clock must be inhibited from the circuit if the content of the register must be left unchanged.

Register with Parallel Load

Most digital systems have a master clock generator that supplies a continuous train of clock pulses. The clock pulses are applied to all flip-flops and registers in the system. The master clock acts like a pump that supplies a constant beat to all parts of the system. A separate control signal must be used to decide which specific clock pulse will have an effect on a particular register.
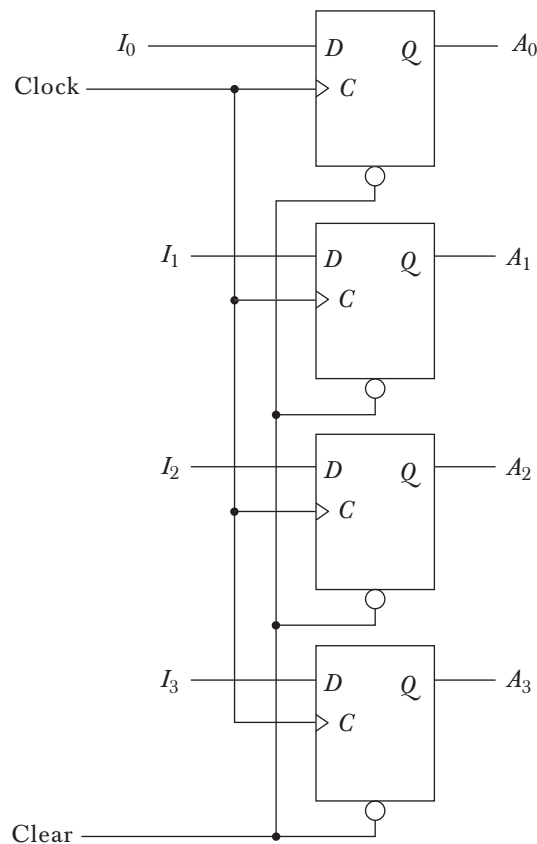
Figure 2-7   4-bit register.

A 4-bit register with a load control input that is directed through gates and into the D inputs is shown in Fig. 2-8. The C inputs receive clock pulses at all times. The buffer gate in the clock input reduces the power requirement from the clock generator. Less power is required when the clock is connected to only one input gate instead of the power consumption that four inputs would have required if the buffer were not used.

load input   The load input in the register determines the action to be taken with each clock pulse. When the load input is 1, the data in the four inputs are transferred into the register with the next positive transition of a clock pulse. When the load input is 0, the data inputs are inhibited and the D inputs of the flip-flops are connected to their outputs. The feedback connection from output to input is necessary because the D flip-flop does not have a "no change" condition. With each clock pulse, the D input determines the next state of the output. To leave the output unchanged, it is necessary to make the D input equal to the present value of the output.

Note that the clock pulses are applied to the C inputs at all times. The load input determines whether the next pulse will accept new information or leave the
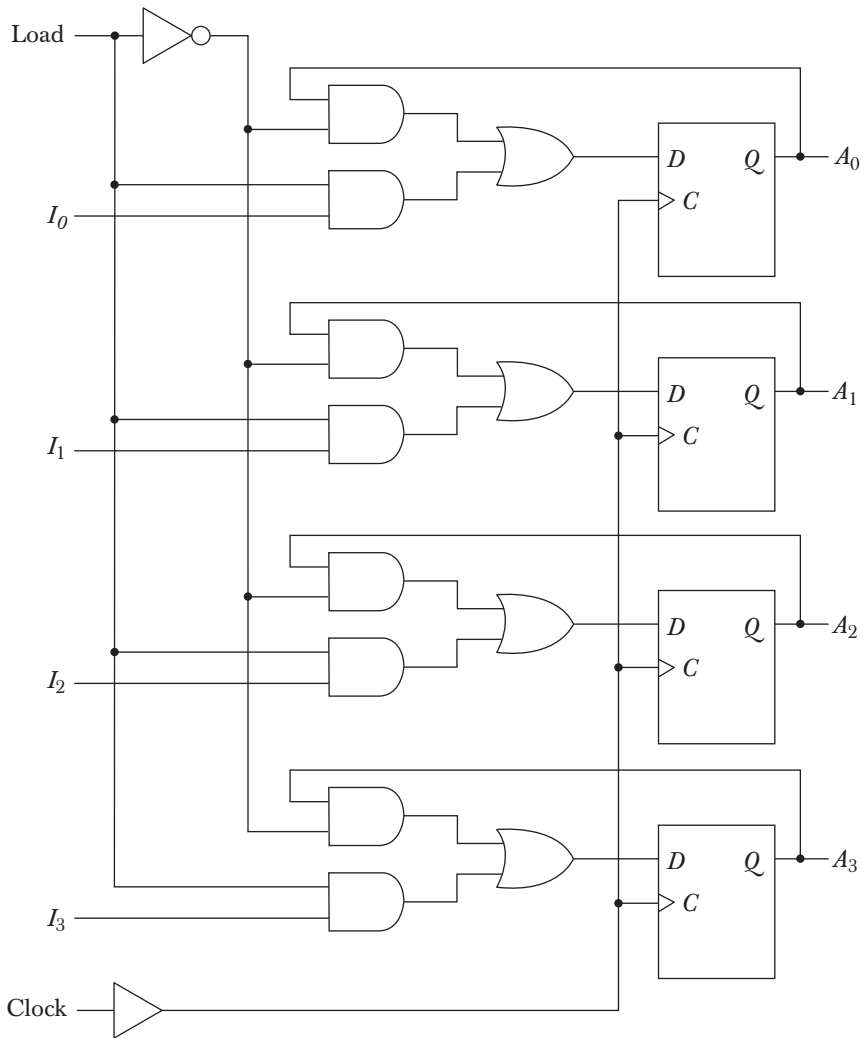
Figure 2-8 4-bit register with parallel load.

information in the register intact. The transfer of information from the inputs into the register is done simultaneously with all four bits during a single pulse transition.

2-5 Shift Registers

A register capable of shifting its binary information in one or both directions is called a shift register. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of

2.7---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

TABLE 2-5 Function Table for the Register of Fig. 2-12

Clock	Clear	Load	Increment	Operation
↑	0	0	0	No change
↑	0	0	1	Increment count by 1
↑	0	1	×	Load inputs I_0 through I_3
↑	1	×	×	Clear outputs to 0

increment

operations. The *increment* operation adds one to the content of a register. By enabling the count input during one clock period, the content of the register can be incremented by one.

2-7   Memory Unit

word

byte

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. The memory stores binary information in groups of bits called *words*. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of l's and 0's and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary-coded information. A group of eight bits is called a *byte*. Most computer memories use words whose number of bits is a multiple of 8. Thus a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of memories in commercial computers is usually stated as the total number of bytes that can be stored.

The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word. Special input lines called address lines select one particular word. Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3, up to $2^k - 1$ where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines. A decoder inside the memory accepts this address and opens the paths needed to select the bits of the specified word. Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits. It is customary to refer to the number of words (or bytes) in a memory with one of the letters K (kilo), M (mega), or G (giga). K is equal to 2^{10}, M is equal to 2^{20}, and G is equal to 2^{30}. Thus, $64K = 2^{16}$, $2M = 2^{21}$, and $4G = 2^{32}$.

Two major types of memories are used in computer systems: random-access memory (RAM) and read-only memory (ROM). These semiconductor memories are classified into Random Access Memories (RAMs) and Sequential Access Memories (SAMs) based on access time. Memories constructed with shift registers, Charge Coupled Devices (CCDs), or bubble memories are examples of SAMs. RAMs are categorized into ROMs, Read Mostly Memories (RMMs), and Read Write Memories (RWMs). ROMs are of two types: Masked Programmed

ROMs and user Programmed PROMs. Two types of RMMs are Erasable and Programmable (EPROM), and Electrically Erasable (EEPROM). RWMs are Static RAM (SRAM) and Dynamic RAM (DRAM). Static RAMs have memory cells as common Flip-Flops. Dynamic RAMs have memory cells that must be refreshed, read and written periodically to avoid loss of memory cells.

Random-Access Memory

RAM

In random-access memory (RAM) the memory cells can be accessed for information transfer from any desired random location. That is, the process of locating a word in memory is the same and requires an equal amount of time no matter where the cells are located physically in memory: thus the name "random access."

Communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer. A block diagram of a RAM unit is shown in Fig. 2-13. The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory. The k address lines provide a binary number of k bits that specify a particular word chosen among the 2^k available inside the memory. The two control inputs specify the direction of transfer desired.

write and read operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Apply the binary address of the desired word into the address lines.
2. Apply the data bits that must be stored in memory into the data input lines.
3. Activate the *write* input.

Figure 2-13   Block diagram of random access memory (RAM).
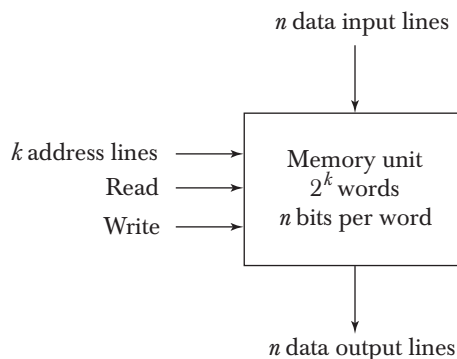
The memory unit will then take the bits presently available in the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Apply the binary address of the desired word into the address lines.
2. Activate the *read* input.

The memory unit will then take the bits from the word that has been selected by the address and apply them into the output data lines. The content of the selected word does not change after reading.

Read-Only Memory

As the name implies, a read-only memory (ROM) is a memory unit that performs the read operation only; it does not have a write capability. This implies that the binary information stored in a ROM is made permanent during the hardware production of the unit and cannot be altered by writing different words into it. Whereas a RAM is a general-purpose device whose contents can be altered during the computational process, a ROM is restricted to reading words that are permanently stored within the unit. The binary information to be stored, specified by the designer, is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

An $m \times n$ ROM is an array of binary cells organized into m words of n bits each. As shown in the block diagram of Fig. 2-14, a ROM has k address input lines to select one of $2^k = m$ words of memory, and n output lines, one for each bit of the word. An integrated circuit ROM may also have one or more enable inputs for expanding a number of packages into a ROM with larger capacity.

k address input lines

$m \times n$ ROM
$(m = 2^k)$
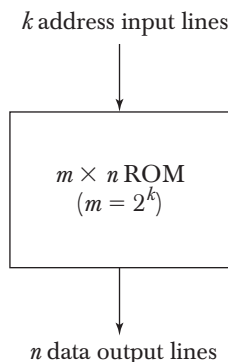
n data output lines

Figure 2-14 Block diagram of read only memory (ROM).

The ROM does not need a read-control line since at any given time, the output lines automatically provide the n bits of the word selected by the address value. Because the outputs are a function of only the present inputs (the address lines), a ROM is classified as a combinational circuit. In fact, a ROM is constructed internally with decoders and a set of OR gates. There is no need for providing storage capabilities as in a RAM, since the values of the bits in the ROM are permanently fixed.

ROMs find a wide range of applications in the design of digital systems. Basically, a ROM generates an input–output relation specified by a truth table. As such, it can implement any combinational circuit with k inputs and n outputs. When employed in a computer system as a memory unit, the ROM is used for storing fixed programs that are not to be altered and for tables of constants that are not subject to change. ROM is also employed in the design of control units for digital computers. As such, they are used to store coded information that represents the sequence of internal control variables needed for enabling the various operations in the computer. A control unit that utilizes a ROM to store binary control information is called a microprogrammed control unit. This subject is discussed in more detail in Chapter 7.

Types of ROMs

The required paths in a ROM may be programmed in three different ways. The first, *mask programming,* is done by the semiconductor company during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table that he or she wishes the ROM to satisfy. The truth table may be submitted in a special form provided by the manufacturer or in a specified format on a computer output medium. The manufacturer makes the corresponding mask for the paths to produce the l's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM. For this reason, mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

PROM

For small quantities it is more economical to use a second type of ROM called a *programmable read-only memory* or PROM. When ordered, PROM units contain all the fuses intact, giving all l's in the bits of the stored words. The fuses in the PROM are blown by application of current pulses through the output terminals for each address. A blown fuse defines a binary 0 state, and an intact fuse gives a binary 1 state. This allows users to program PROMs in their own laboratories to achieve the desired relationship between input addresses and stored words. Special instruments called *PROM programmers* are available commercially to facilitate this procedure. In any case, all procedures for programming ROMs are hardware procedures even though the word "programming" is used.

The hardware procedure for programming ROMs or PROMs is irreversible, and once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed. A third type of ROM available is called *erasable PROM* or EPROM.

The EPROM can be restructured to the initial value even though its fuses have been blown previously. When the EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as fuses. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of words. Certain PROMs can be erased with electrical signals instead of ultraviolet light. These PROMs are called *electrically erasable PROM* or *EEPROM*
EEPROM. Flash memory is a form of EEPROM in which a block of bytes can be erased in a very short duration. Example applications of EEPROM devices are:

1. storing current time and date in a machine.
2. storing port statusses.

Examples of flash memory device applications are:

1. storing messages in a mobile phone.
2. storing photographs in a digital camera.

PROBLEMS

2-1. TTL SSI come mostly in 14-pin 1C packages. Two pins are reserved for power supply and the other pins are used for input and output terminals. How many circuits are included in one such package if it contains the following type of circuits? (a) Inverters; (b) two-input exclusive-OR gates; (c) three-input OR gates; (d) four-input AND gates; (e) five-input NOR gates; (f) eight-input NAND gates; (g) clocked *JK* flip-flops with asynchronous clear.

2-2. MSI chips perform elementary digital functions such as decoders, multiplexers, registers, and counters. The following are TTL-type integrated circuits that provide such functions. Find their description in a data book and compare them with the corresponding component presented in this chapter.
   a. IC type 74155 dual 2-to-4-line decoders.
   b. IC type 74157 quadruple 2-to-l-line multiplexers.
   c. IC type 74194 4-bit bidirectional shift register with parallel load.
   d. IC type 74163 4-bit binary counter with parallel load and synchronous clear.

2-3. Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and one 2-to-4-line decoder. Use block diagrams similar to Fig. 2-4.

2-4. Draw the logic diagram of a 2-to-4-line decoder with only NOR gates. Include an enable input.

2-5. Modify the decoder of Fig. 2-3 so that the circuit is enabled when $E = 1$ and disabled when $E = 0$. List the modified truth table.

2-6. Draw the logic diagram of an eight-input, three-output encoder whose truth table is given in Table 2-2. What is the output when all the inputs are equal to 0? What is the output when only input D_0 is equal to 0? Establish a procedure that will distinguish between these two cases.

2-7.   Construct a 16-to-l-line multiplexer with two 8-to-l-line multiplexers and one 2-to-l-line multiplexer. Use block diagrams for the three multiplexers.

2-8.   Draw the block diagram of a dual 4-to-l-line multiplexers and explain its operation by means of a function table.

2-9.   Include a two-input AND gate with the register of Fig. 2-7 and connect the gate output to the clock inputs of all the flip-flops. One input of the AND gate receives the clock pulses from the clock pulse generator. The other input of the AND gate provides a parallel load control. Explain the operation of the modified register.

2-10. What is the purpose of the buffer gate in the clock input of the register of Fig. 2-8?

2-11. Include a synchronous clear capability to the register with parallel load of Fig. 2-8.

2-12. The content of a 4-bit register is initially 1101. The register is shifted six times to the right with the serial input being 101101. What is the content of the register after each shift?

2-13. What is the difference between serial and parallel transfer? Using a shift register with parallel load, explain how to convert serial input data to parallel output and parallel input data to serial output.

2-14. A ring counter is a shift register as in Fig. 2-9 with the serial output connected to the serial input. Starting from an initial state of 1000, list the sequence of states of the four flip-flops after each shift.

2-15. The 4-bit bidirectional shift register with parallel load shown in Fig. 2-10 is enclosed within one IC package.
   a. Draw a block diagram of the IC showing all inputs and outputs. Include two pins for power supply.
   b. Draw a block diagram using two ICs to produce an 8-bit bidirectional shift register with parallel load.

2-16. How many flip-flops will be complemented in a 10-bit binary counter to reach the next count after (a) 1001100111; (b) 0011111111?

2-17. Show the connections between four 4-bit binary counters with parallel load (Fig. 2-12) to produce a 16-bit binary counter with parallel load. Use a block diagram for each 4-bit counter.

2-18. Show how the binary counter with parallel load of Fig. 2-12 can be made to operate as a divide-by-N counter (i.e., a counter that counts from 0000 to N-and back to 0000). Specifically show the circuit for a divide-by-10 counter using the counter of Fig. 2-12 and an external AND gate.

2-19. The following memory units are specified by the number of words times the number of bits per word. How many address lines and input-output data lines are needed in each case? (a) 2K × 16; (b) 64K × 8; (c) 16M × 32; (d) 4G × 64.

2-20. Specify the number of bytes that can be stored in the memories listed in Prob. 2–19.

2-21. How many 128 × 8 memory chips are needed to provide a memory capacity of 4096 × 16?

2-22. Given a 32 × 8 ROM chip with an enable input, show the external connections necessary to construct a 128 × 8 ROM with four chips and a decoder.

2-23. A ROM chip of 4096 × 8 bits has two enable inputs and operates from a 5-volt power supply. How many pins are needed for the integrated circuit package? Draw a block diagram and label all input and output terminals in the ROM.

REFERENCES

1. Hill, F. J., and G. R. Peterson, *Introduction to Switching Theory and Logical Design,* 3rd ed. New York: John Wiley, 1981.

2. Mano, M. M., *Digital Design,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1991.

3. Roth, C. H., *Fundamentals of Logic Design,* 3rd ed. St. Paul, MN: West Publishing, 1985.

4. Sandige, R. S., *Modern Digital Design*. New York: McGraw-Hill, 1990.

5. Shiva, S. G., *Introduction to Logic Design*. Glenview, II: Scott, Foresman, 1988.

6. Wakerly, J. F., *Digital Design Principles and Practices*. Englewood Cliffs, NJ: Prentice Hall, 1990.

7. Ward, S. A., and R. H. Halstead, Jr., *Computation Structures*. Cambridge, MA: MIT Press, 1990.

$R1$ and the address is in AR. The write operation can be stated symbolically as follows:

$$\text{Write:}\quad M[AR] \leftarrow R1$$

This causes a transfer of information from $R1$ into the memory word M selected by the address in AR.

4-4   Arithmetic Microoperations

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operation on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on nonnumeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec. 4-2. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperations. In the next two sections we present the logic and shift microoperations.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

$$R3 \leftarrow R1 + R2$$

add
microoperation

specifies an *add* microoperation. It states that the contents of register $R1$ are added to the contents of register $R2$ and the sum transferred to register $R3$. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 4-3. Subtraction is most often

*subtract
microoperation*

implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow R1 + \overline{R2} + 1$$

$\overline{R2}$ is the symbol for the 1's complement of $R2$. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of $R1$ to the 2's complement of $R2$ is equivalent to $R1 - R2$.

TABLE 4-3 Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations of multiply and divide are not listed in Table 4-3. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift (see Chapter 10).

Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder (see Fig. 1-17). The digital circuit that
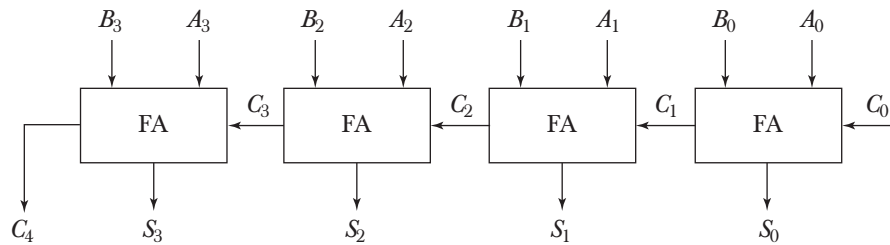
Figure 4-6   4-bit binary adder.

binary adder

generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4-6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4. The S outputs of the full-adders generate the required sum bits.

full-adder

An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The n data bits for the A inputs come from one register (such as $R1$), and the n data bits for the B inputs come from another register (such as $R2$). The sum can be transferred to a third register or to one of the source registers ($R1$ or $R2$), replacing its previous content.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements as discussed in Sec. 3-2. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

adder-subtractor

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 4-7. The mode input M controls the operation. When $M = 0$ the circuit is an adder and when $M = 1$ the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the
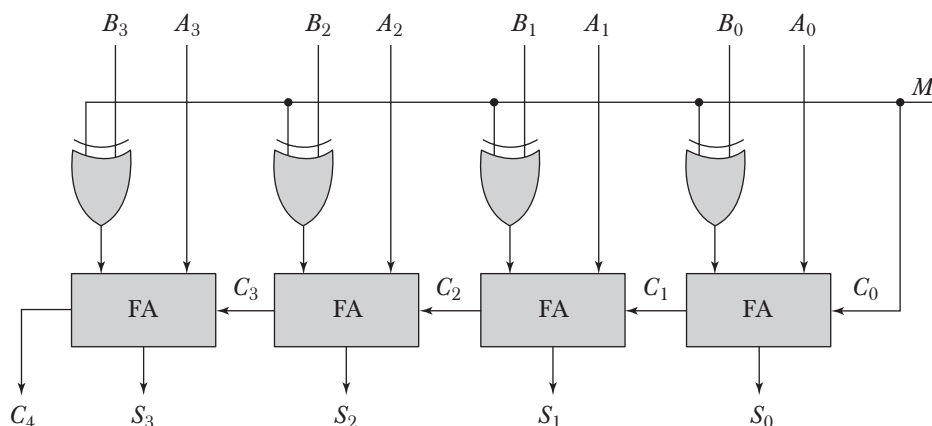
Figure 4-7 4-bit adder-subtractor.

2's complement of B. For unsigned numbers, this gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$ provided that there is no overflow.

Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter (see Fig. 2-10). Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders (see Fig. 1-16) connected in cascade.

incrementer

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 4-8. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

The circuit of Fig. 4-8 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.
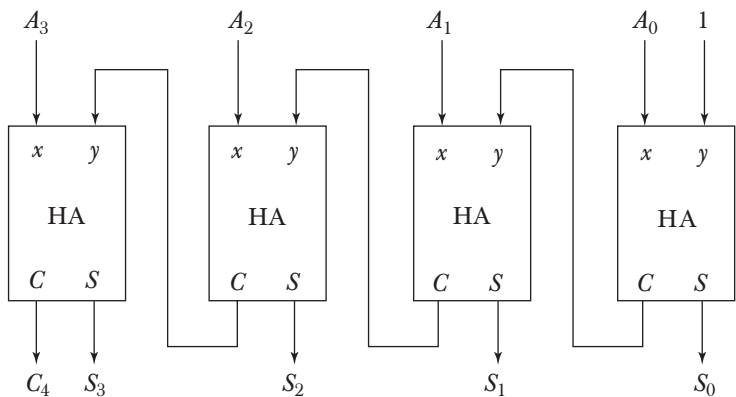
Figure 4-8 4-bit binary incrementer.

Arithmetic Circuit

arithmetic circuit

The arithmetic microoperations listed in Table 4-3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig. 4-9. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B are connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers

input carry

are controlled by two selection inputs, S_1 and S_0. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries a connected from one stage to the next.

The output of the binary adder is calculated from the following arithmetic sum:

$$D = A + Y + C_{in}$$

where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol $+$ in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 4-4.
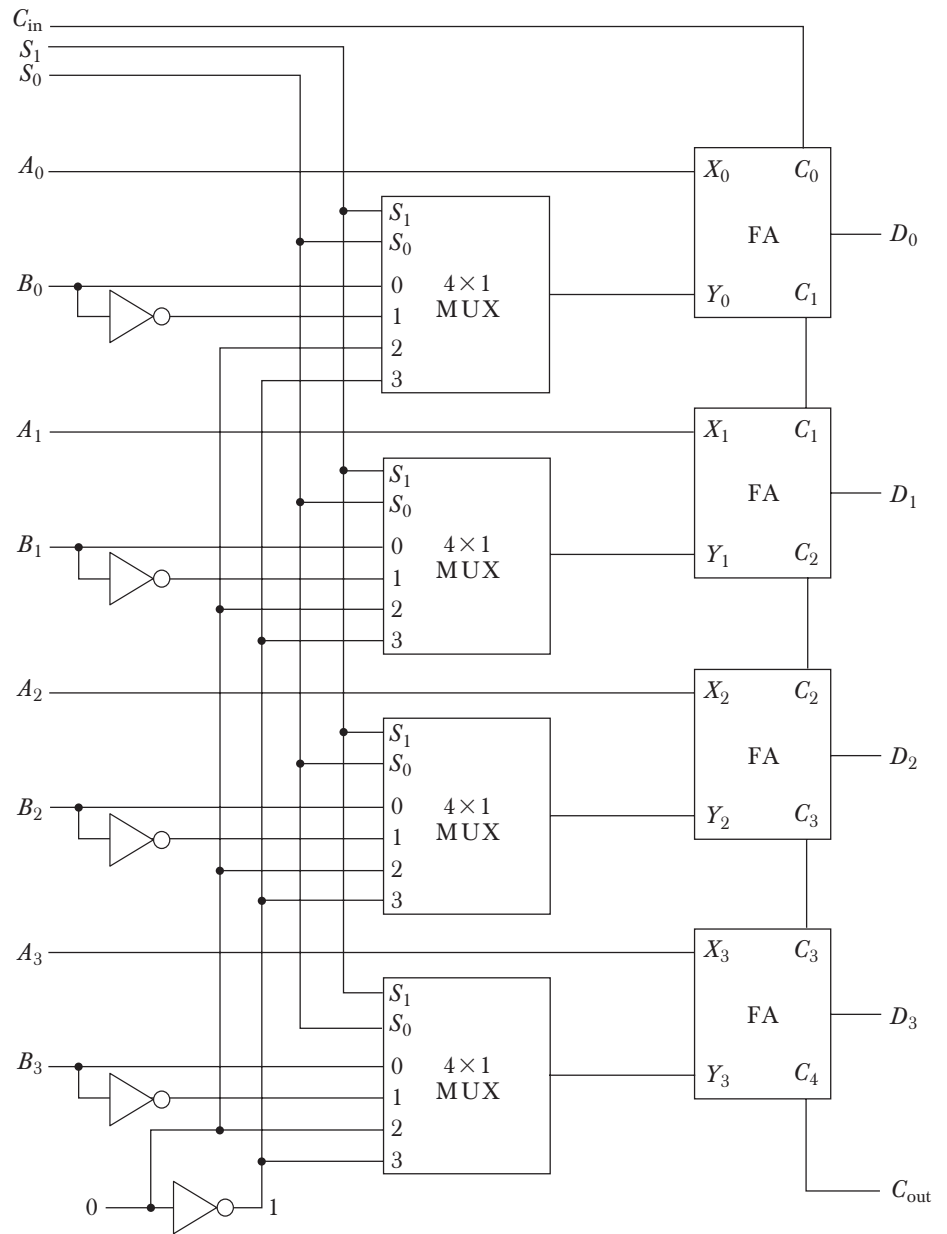
Figure 4-9 4-bit arithmetic circuit.

Data Representation

IN THIS CHAPTER

3-1   Data Types

The term data refers to factual information used for analysis or reasoning. Data itself has no meaning, but becomes information when it is assigned a meaning or interpreted. Information is a collection of facts or data that is communicated. Binary information in digital computers is stored in memory or processor registers. Registers contain either data or control information. Control information is a bit or a group of bits used to specify the sequence of command signals needed for manipulation of the data in other registers. Data are numbers and other binary-coded information that are operated on to achieve required computational results. In this chapter we present the most common types of data found in digital computers and show how the various data types are represented in binary-coded form in computer registers.

The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form. This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only l's and 0's. The binary number system is the most natural system to use in a digital computer. But sometimes it is convenient to employ different number systems, especially the decimal number system, since it is used by people to perform arithmetic computations.

Number Systems

radix

A number system of *base,* or *radix,* r is a system that uses distinct symbols for r digits. Numbers are represented by a string of digit symbols. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits. For example, the decimal number system in everyday use employs the radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The string of digits 724.5 is interpreted to represent the quantity

decimal

$$7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

that is, 7 hundreds, plus 2 tens, plus 4 units, plus 5 tenths. Every decimal number can be similarly interpreted to find the quantity it represents.

binary

The *binary* number system uses the radix 2. The two digit symbols used are 0 and 1. The string of digits 101101 is interpreted to represent the quantity

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$$

To distinguish between different radix numbers, the digits will be enclosed in parentheses and the radix of the number inserted as a subscript. For example, to show the equality between decimal and binary forty-five we will write $(101101)_2 = (45)_{10}$.

octal
hexadecimal

Besides the decimal and binary number systems, the *octal* (radix 8) and *hexadecimal* (radix 16) are important in digital computer work. The eight symbols of the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. The 16 symbols of the hexadecimal system are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The last six symbols are, unfortunately, identical to the letters of the alphabet and can cause confusion at times. However, this is the convention that has been adopted. When used to represent hexadecimal digits, the symbols A, B, C, D, E, F correspond to the decimal numbers 10, 11, 12, 13, 14, 15, respectively.

A number in radix r can be converted to the familiar decimal system by forming the sum of the weighted digits. For example, octal 736.4 is converted to decimal as follows:

$$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$$
$$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$$

The equivalent decimal number of hexadecimal F3 is obtained from the following calculation:

$$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3 = (243)_{10}$$

conversion

Conversion from decimal to its equivalent representation in the radix r system is carried out by separating the number into its *integer* and *fraction* parts and

converting each part separately. The conversion of a decimal integer into a base r representation is done by successive divisions by r and accumulation of the remainders. The conversion of a decimal fraction to radix r representation is accomplished by successive multiplications by r and accumulation of the integer digits so obtained. Figure 3-1 demonstrates these procedures.

The conversion of decimal 41.6875 into binary is done by first separating the number into its integer part 41 and fraction part .6875. The integer part is converted by dividing 41 by $r = 2$ to give an integer quotient of 20 and a remainder of 1. The quotient is again divided by 2 to give a new quotient and remainder. This process is repeated until the integer quotient becomes 0. The coefficients of the binary number are obtained from the remainders with the first remainder giving the low-order bit of the converted binary number.

The fraction part is converted by multiplying it by $r = 2$ to give an integer and a fraction. The new fraction (*without* the integer) is multiplied again by 2 to give a new integer and a new fraction. This process is repeated until the fraction part becomes zero or until the number of digits obtained gives the required accuracy. The coefficients of the binary fraction are obtained from the integer digits with the first integer computed being the digit to be placed next to the binary point. Finally, the two parts are combined to give the total required conversion.

Octal and Hexadecimal Numbers

The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each. The corresponding octal digit is then assigned to each group of bits and the string of digits so obtained gives the octal equivalent of the binary number. Consider, for example, a 16-bit register. Physically, one may think of the

Figure 3-1   Conversion of decimal 41.6875 into binary.

Integer = 41   Fraction = 0.6875

$$
\begin{array}{c|c}
41 & \\
20 & 1 \\
10 & 0 \\
5 & 0 \\
2 & 1 \\
1 & 0 \\
0 & 1 \\
\end{array}
\qquad
\begin{array}{r}
0.6875 \\
\underline{2} \\
1.3750 \\
\underline{\times\ 2} \\
0.7500 \\
\underline{\times\ 2} \\
1.5000 \\
\underline{\times\ 2} \\
1.0000 \\
\end{array}
$$

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$$(41.6875)_{10} = (101001.1011)_2$$
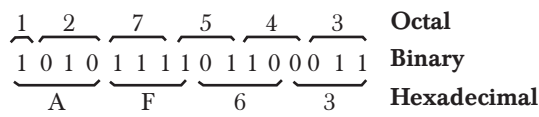
Figure 3-2   Binary, octal, and hexadecimal conversion.

register as composed of 16 binary storage cells, with each cell capable c hold-ing either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown in Fig. 3-2. Since a binary number consists of a string of l's and 0's, the 16-bit register can be used to store any binary number from 0 to $2^{16} - 1$. For the particular example shown, the binary number stored in the register is the equivalent of decimal 44899. Starting from the low-order bit, we partition the register into groups of three bits each (the sixteenth bit remains in a group by itself). Each group of three bits is assigned its octal equivalent and placed on top of the register. The string of octal digits so obtained represents the octal equivalent of the binary number.

Conversion from binary to hexadecimal is similar except that the bits are divided into groups of four. The corresponding hexadecimal digit for each group of four bits is written as shown below the register of Fig. 3-2. The string of hexadecimal digits so obtained represents the hexadecimal equivalent of the binary number. The corresponding octal digit for each group of three bits is easily remembered after studying the first eight entries listed in Table 3-1. The correspondence between a hexadecimal digit and its equivalent 4-bit code can be found in the first 16 entries of Table 3-2.
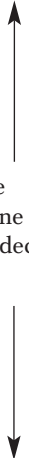
TABLE 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

Table 3-1 lists a few octal numbers and their representation in registers in binary-coded form. The binary code is obtained by the procedure explained above. Each octal digit is assigned a 3-bit code as specified by the entries of the first eight digits in the table. Similarly, Table 3-2 lists a few hexadecimal numbers and their representation in registers in binary-coded form. Here the binary code is obtained by assigning to each hexadecimal digit the 4-bit code listed in the first 16 entries of the table.

Comparing the binary-coded octal and hexadecimal numbers with their binary number equivalent we find that the bit combination in all three representations is exactly the same. For example, decimal 99, when converted to binary, becomes 1100011. The binary-coded octal equivalent of decimal 99 is 001 100 011 and the binary-coded hexadecimal of decimal 99 is 0110 0011. If we neglect the leading zeros in these three binary representations, we find that their bit combination is identical. This should be so because of the straightforward conversion that exists between binary numbers and octal or hexadecimal. The point of all this is that a string of l's and 0's stored in a register could represent a binary number, but this same string of bits may be interpreted as holding an octal number in binary-coded form (if we divide the bits in groups of three) or as holding a hexadecimal number in binary-coded form (if we divide the bits in groups of four).

TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

The registers in a digital computer contain many bits. Specifying the content of registers by their binary values will require a long string of binary digits. It is more convenient to specify content of registers by their octal or hexadecimal equivalent. The number of digits is reduced by one-third in the octal designation and by one-fourth in the hexadecimal designation. For example, the binary number 1111 1111 1111 has 12 digits. It can be expressed in octals as 7777 (four digits) or in hexadecimal as FFF (three digits). Computer manuals invariably choose either the octal or the hexadecimal designation for specifying contents of registers.

Decimal Representation

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to solve this conflict is to convert all input decimal numbers into binary numbers, let the computer perform all arithmetic operations in binary and then convert the binary results back to decimal for the human user to understand. However, it is also possible for the computer to perform arithmetic operations directly with decimal numbers provided they are placed in registers in a coded form. Decimal numbers enter the computer usually as binary-coded alphanumeric characters. These codes, introduced later, may contain from six to eight bits for each decimal digit. When decimal numbers are used for internal arithmetic computations, they are converted to a binary code with four bits per digit.

binary code

A binary code is a group of n bits that assume up to 2^n distinct combinations of l's and 0's with each combination representing one element of the set that is being coded. For example, a set of four elements can be coded by a 2-bit code with each element assigned one of the following bit combinations; 00, 01, 10, or 11. A set of eight elements requires a 3-bit code, a set of 16 elements requires a 4-bit code, and so on. A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six combinations will remain unassigned. Numerous different codes can be obtained by arranging four bits in 10 distinct combinations. The bit assignment most commonly used for the decimal digits is the straight binary assignment listed in the first 10 entries of Table 3-3. This particular code is called *binary-coded decimal* and is commonly

BCD

referred to by its abbreviation BCD. Other decimal codes are sometimes used and a few of them are given in Sec. 3-5.

It is very important to understand the difference between the *conversion* of decimal numbers into binary and the *binary coding* of decimal numbers. For example, when *converted* to a binary number, the decimal number 99 is represented by the string of bits 1100011, but when represented in BCD, it becomes 1001 1001. The *only* difference between a decimal number represented by the familiar digit symbols 0, 1, 2, . . . , 9 and the BCD symbols 0001, 0010, . . . , 1001 is in the symbols used to represent the digits–the

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

Decimal number	Binary-coded decimal (BCD) number	
0	0000	
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

number itself is exactly the same. A few decimal numbers and their representation in BCD are listed in Table 3-3.

Alphanumeric Representation

character

Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters. An *alphanumeric character set* is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, +, and =. Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included). In the first case, the binary code will require six bits and in the second case, seven bits. The stan-

ASCII

dard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters. The binary code for the uppercase letters, the decimal digits, and a few special characters is listed in Table 3-4. Note that the decimal digits in ASCII can be converted to BCD by removing the three high-order bits, 011. A complete list of ASCII characters is provided in Table 11-1.

Binary codes play an important part in digital computer operations. The codes must be in binary because registers can only hold binary information. One must realize that binary codes merely change the symbols, not the meaning of the discrete elements they represent. The operations specified for digital computers must take into consideration the meaning of the

TABLE 3-4 American Standard Code for information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

bits stored in registers so that operations are performed on operands of the same type. In inspecting the bits of a computer register at random, one is likely to find that it represents some type of coded information rather than a binary number.

Binary codes can be formulated for any set of discrete elements such as the musical notes and chess pieces and their positions on the chessboard. Binary codes are also used to formulate instructions that specify control information for the computer. This chapter is concerned with *data* representation. Instruction codes are discussed in Chap. 5.

3-2   Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the $(r - 1)$'s complement.

When the value of the base r is substituted in the name, the two types are referred to as the 2's and l's complement for binary numbers and the 10's and 9's complement for decimal numbers.

$(r - 1)$'s Complement

9's complement

Given a number N in base r having n digits, the $(r - 1)$'s complement of N is defined as $(r^n - 1) - N$. For decimal numbers $r = 10$ and $r - 1 = 9$, so the 9's complement of N is $(10^n - 1) - N$. Now, 10^n represents a number that consists of a single 1 followed by n 0's. $10^n - 1$ is a number represented by n 9's. For example, with $n = 4$ we have $10^4 = 10000$ and $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of 546700 is $999999 - 546700 = 453299$ and the 9's complement of 12389 is $99999 - 12389 = 87610$.

1's complement

For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n l's. For example, with $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the l's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the l's complement of a binary number is formed by changing l's into 0's and 0's into l's. For example, the l's complement of 1011001 is 0100110 and the l's complement of 0001111 is 1110000.

The $(r - 1)$'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

$(r$'s) Complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Comparing with the $(r - 1)$'s complement, we note that the r's complement is obtained by adding 1 to the $(r - 1)$'s complement since

10's complement

$r^n - N = [(r^n - 1) - N] + 1$. Thus the 10's complement of the decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's complement value. The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the l's complement value.

Since 10^n is a number represented by a 1 followed by n 0's, then $10^n - N$, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9. The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's

2's complement

complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing l's by 0's and 0's by l's in all other higher, significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing l's by 0's and 0's by l's in the other four most significant bits.

In the definitions above it was assumed that the numbers do not have a radix point. If the original number N contains a radix point, it should be removed temporarily to form the r's or $(r-1)$'s complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The r's complement of N is $r^n - N$. The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

subtraction   The subtraction of two n-digit unsigned numbers $M - N(N \neq 0)$ in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result $M - N$.

end carry   3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Consider, for example, the subtraction $72532 - 13250 = 59282$. The 10's complement of 13250 is 86750. Therefore:

$$
\begin{array}{rl}
M = & 72532 \\
\text{10's complement of } N = & +86750 \\
\hline
\text{Sum} = & 159282 \\
\text{Discard end carry } 10^5 = & -100000 \\
\hline
\text{Answer} = & 59282
\end{array}
$$

Now consider an example with $M < N$. The subtraction $13250 - 72532$ produces negative 59282. Using the procedure with complements, we have

$$
\begin{array}{rl}
M = & 13250 \\
\text{10's complement of } N = & +27468 \\
\hline
\text{Sum} = & 40718
\end{array}
$$

TABLE 3-4 American Standard Code for information Interchange (ASCII)

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

bits stored in registers so that operations are performed on operands of the same type. In inspecting the bits of a computer register at random, one is likely to find that it represents some type of coded information rather than a binary number.

Binary codes can be formulated for any set of discrete elements such as the musical notes and chess pieces and their positions on the chessboard. Binary codes are also used to formulate instructions that specify control information for the computer. This chapter is concerned with *data* representation. Instruction codes are discussed in Chap. 5.

3-2   Complements

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the $(r - 1)$'s complement.

When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

$(r - 1)$'s Complement

9's complement

Given a number N in base r having n digits, the $(r - 1)$'s complement of N is defined as $(r^n - 1) - N$. For decimal numbers $r = 10$ and $r - 1 = 9$, so the 9's complement of N is $(10^n - 1) - N$. Now, 10^n represents a number that consists of a single 1 followed by n 0's. $10^n - 1$ is a number represented by n 9's. For example, with $n = 4$ we have $10^4 = 10000$ and $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of 546700 is $999999 - 546700 = 453299$ and the 9's complement of 12389 is $99999 - 12389 = 87610$.

1's complement

For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, with $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The $(r - 1)$'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

$(r$'s$)$ Complement

10's complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and 0 for $N = 0$. Comparing with the $(r - 1)$'s complement, we note that the r's complement is obtained by adding 1 to the $(r - 1)$'s complement since $r^n - N = [(r^n - 1) - N] + 1$. Thus the 10's complement of the decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's complement value. The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's complement value.

2's complement

Since 10^n is a number represented by a 1 followed by n 0's, then $10^n - N$, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9. The 10's complement of 246700 is 753300 and is obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9. Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher, significant bits. The 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits.

In the definitions above it was assumed that the numbers do not have a radix point. If the original number N contains a radix point, it should be removed temporarily to form the r's or $(r-1)$'s complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The r's complement of N is $r^n - N$. The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

Subtraction of Unsigned Numbers

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

subtraction   The subtraction of two n-digit unsigned numbers $M - N (N \neq 0)$ in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result $M - N$.

end carry   3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Consider, for example, the subtraction $72532 - 13250 = 59282$. The 10's complement of 13250 is 86750. Therefore:

$$
\begin{array}{rl}
M = & 72532 \\
\text{10's complement of } N = & +86750 \\
\hline
\text{Sum} = & 159282 \\
\text{Discard end carry } 10^5 = & -100000 \\
\hline
\text{Answer} = & 59282
\end{array}
$$

Now consider an example with $M < N$. The subtraction $13250 - 72532$ produces negative 59282. Using the procedure with complements, we have

$$
\begin{array}{rl}
M = & 13250 \\
\text{10's complement of } N = & +27468 \\
\hline
\text{Sum} = & 40718
\end{array}
$$

There is no end carry

Answer is negative $59282 = $ 10's complement of 40718

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers $X = 1010100$ and $Y = 1000011$, we perform the subtraction $X - Y$ and $Y - X$ using 2's complements:

$$
\begin{array}{rr}
X = & 1010100 \\
\text{2's complement of } Y = & +0111101 \\
\hline
\text{Sum} = & 10010001 \\
\text{Discard end carry } 2^7 = & -10000000 \\
\hline
\text{Answer: } X - Y = & 0010001 \\
\end{array}
$$

$$
\begin{array}{rr}
Y = & +000011 \\
\text{2's complement of } X = & +0101100 \\
\hline
\text{Sum} = & 1101111 \\
\end{array}
$$

There is no end carry

Answer is negative $0010001 = $ 2's complement of 1101111

3-3   Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with l's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the left-most position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

binary point   In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is

always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

signed numbers   When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

   1. Signed-magnitude representation
   2. Signed-1's complement representation
   3. Signed 2's complement representation

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. $+14$ is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110. Note that each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit. Although there is only one way to represent $+14$, there are three different ways to represent -14 with eight bits.

In signed-magnitude representation	1 0001110
In signed-1's complement representation	1 1110001
In signed-2's complement representation	1 1110010

The signed-magnitude representation of -14 is obtained from $+14$ by complementing only the sign bit. The signed-1's complement representation of -14 is obtained by complementing all the bits of $+14$, including the sign bit. The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

   The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The 1's complement imposes difficulties because

it has two representations of 0 (+0 and −0). It is seldom used for arithmetic operations except in some older computers. The l's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, $(+25) + (−37) = −(37 − 25) = −12$ and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magnitude representation is described in Sec. 10-2.) By contrast, the rule for adding *2's complement* numbers in the signed-2's complement system does not require a comparison *addition* or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

+ 6	00000110		− 6	11111010
+13	00001101		+13	00001101
+19	00010011		+ 7	00000111
+ 6	00000110		− 6	11111010
−13	11110011		−13	11110011
− 7	11111001		−19	11101101

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to −7.

There is no end carry

Answer is negative $59282 = 10$'s complement of 40718

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers $X = 1010100$ and $Y = 1000011$, we perform the subtraction $X - Y$ and $Y - X$ using 2's complements:

$$
\begin{aligned}
X &= 1010100 \\
\text{2's complement of } Y &= +0111101 \\
\text{Sum} &= 10010001 \\
\text{Discard end carry } 2^7 &= -10000000 \\
\text{Answer: } X - Y &= 0010001 \\
\\
Y &= 000011 \\
\text{2's complement of } X &= +0101100 \\
\text{Sum} &= 1101111
\end{aligned}
$$

There is no end carry

Answer is negative $0010001 = 2$'s complement of 1101111

3-3   Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with l's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

binary point   In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is

always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

signed numbers When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. Signed-magnitude representation
2. Signed-1's complement representation
3. Signed 2's complement representation

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the l's or 2's complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14:00001110. Note that each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit. Although there is only one way to represent +14, there are three different ways to represent −14 with eight bits.

In signed-magnitude representation	1 0001110
In signed-1's complement representation	1 1110001
In signed-2's complement representation	1 1110010

The signed-magnitude representation of −14 is obtained from +14 by complementing only the sign bit. The signed-1's complement representation of −14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The l's complement imposes difficulties because

it has two representations of 0 ($+0$ and -0). It is seldom used for arithmetic operations except in some older computers. The l's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, $(+25) + (-37) = -(37 - 25) = -12$ and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magnitude representation is described in Sec. 10-2.) By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

2's complement addition

$+\ 6$	00000110	-6	11111010
$+13$	00001101	$+13$	00001101
$+19$	00010011	$+7$	00000111
$+\ 6$	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of $+7$. We therefore recognize the original negative number to be equal to -7.

Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows: Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed .This is demonstrated by the following relationship:

$$(\pm A) - (+B) \qquad = (\pm A) + (-B)$$
$$(\pm A) - (-B) \qquad = (\pm A) + (+B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of $(-6) - (-13) = +7$. In binary with eight bits this is written as $11111010 - 11110011$. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give $(+13)$. In binary this is $11111010 + 00001101 = 100000111$. Removing the end carry, we obtain the correct answer 00000111 $(+7)$.

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains $n + 1$ bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the

sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, $+70$ and $+80$, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary $+127$ to binary -128. Since the sum of the two numbers is $+150$, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

	carries: 0 1		carries: 1 0
$+70$	0 1000110	-70	1 0111010
$+80$	0 1010000	-80	1 0110000
$+150$	1 0010110	-150	0 1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

overflow detection   An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

Decimal Fixed-Point Representation

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

$$0100 \ 0011 \ 1000 \ 0101$$

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. Also, the circuits required to perform decimal

arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits. It is customary to designate a plus with four 0's and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10's complement form. Consider the addition $(+375) + (-240) = +135$ done in the signed-10's complement system.

$$
\begin{array}{rl}
0\ 375 & (0000\ 0011\ 0111\ 0101)_{\text{BCD}} \\
+\ 9\ 760 & (1001\ 0111\ 0110\ 0000)_{\text{BCD}} \\
\hline
0\ 135 & (0000\ 0001\ 0011\ 0101)_{\text{BCD}}
\end{array}
$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain $+135$. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders (see Fig. 10-18).

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

3-4   Floating-Point Representation

mantissa
exponent

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number $+6132.789$ is represented in floating-point with a fraction and an exponent as follows:

Fraction	Exponent
$+0.6132789$	$+04$

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$.

   Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

fraction

   A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number $+1001.11$ is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number $+4$. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

normalization

A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's. The number can be normalized by shifting it

three positions to the left and discarding the leading 0's to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in the mantissa and exponent.

Two main standard forms of floating-point numbers are from the following organizations that decide standards: ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronic Engineers). The ANSI 32-bit floating-point numbers in byte format with examples are given below:

Byte Format:

Byte 1	Byte 2	Byte 3	Byte 4
SEEEE	·IMMMMMMM	MMMMMMMM	MMMMMMMM

Exponent ⎵   Binary Point ↑   Mantissa
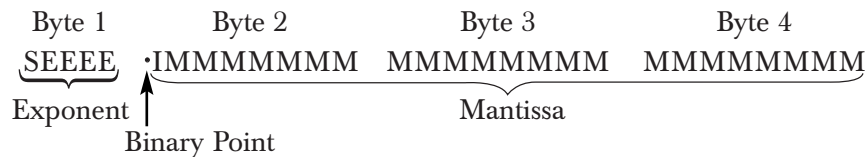
S = Sign of Mantissa, E = Exponent Bits in 2's complement, M = Mantissa Bits

Examples:

$$13 = 1101 = 0.1101 \times 2^4$$
$$= 00000100 \ 11010000 \ 00000000 \ 00000000$$
$$-17 = -10001 = -0.10001 \times 2^5$$
$$= 10000101 \ 10001000 \ 00000000 \ 00000000$$
$$-0.125 = -0.001 = -.1 \times 2^{-2}$$
$$= 11111110 \ 10000000 \ 00000000 \ 00000000$$

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers. Arithmetic operations with floating-point numbers are discussed in Sec. 10-5.

3-5   Other Binary Codes

In previous sections we introduced the most common types of binary-coded data found in digital computers. Other binary codes for decimal numbers and

5.1---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

Basic Computer Organization and Design

IN THIS CHAPTER

5-1   Instruction Codes

In this chapter we introduce a basic computer and show how its operation can be specified with register transfer statements. The organization of the computer is defined by its internal registers, the riming and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. Although the basic computer presented in this chapter is very small compared to commercial computers, it has the advantage of being simple enough so we can demonstrate the design process without too many complications.

The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers. The general-purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations,

operands, and the sequence by which processing has to occur. The data-processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.

instruction code

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its

operation code

operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations. As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

At this point we must recognize the relationship between a computer operation and a microoperation. An operation is part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation code, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation. For this reason, an operation code is sometimes called a macrooperation because it specifies a set of micro-operations.

The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their address. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers. There are many variations for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats

are conceived by computer designers who specify the architecture of the computer. In this chapter we choose a particular instruction code to explain the basic organization and design of digital computers.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

opcode

Figure 5-1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 5-1 Stored program organization.
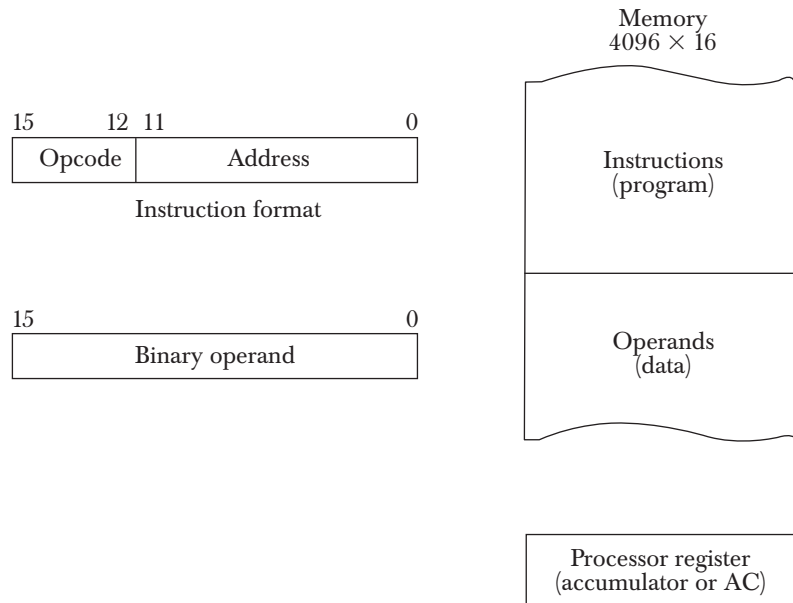
accumulator (AC)

Computers that have a single-processor register usually assign to it the name accumulator and label it *AC*. The operation is performed with the memory operand and the content of *AC*.

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear *AC*, complement *AC*, and increment *AC* operate on data stored in the *AC* register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

Indirect Address

immediate instruction

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

As an illustration of this configuration, consider the instruction code format shown in Fig. 5-2(a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by *I*. The mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Fig. 5-2(b). It is placed in address 22 in memory. The *I* bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of *AC*. The instruction in address 35 shown in Fig. 5-2(c) has a mode bit $I = 1$. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of *AC*. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself. We define the *effective address* to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction. Thus the effective address in the instruction of Fig. 5-2(b) is 457 and in the instruction of Fig 5-2(c) is 1350.

effective address

The direct and indirect addressing modes are used in the computer presented in this chapter. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of

15	14	12	11	0
I	Opcode		Address	

(a) Instruction format

Memory

| 22 | 0 | ADD | 457 |

| 457 | Operand |

(+)

AC

(b) Direct address

Memory

| 35 | 1 | ADD | 300 |

| 300 | 1350 |

| 1350 | Operand |

(+)

AC

(c) Indirect address

Figure 5-2   Demonstration of direct and indirect address.

data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

5-2   Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction code after it is read

from memory. The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in Fig. 5-3. The registers are also listed in Table 5-1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation, part of the instruction and a bit to specify a direct or indirect address. The data register (*DR*) holds the operand read from memory. The accumulator (*AC*) register is a general-purpose processing register. The instruction read from memory is placed in the instruction register (*IR*). The temporary register (*TR*) is used for holding temporary data during the processing.

TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

program counter (PC)

The memory address register (*AR*) has 12 bits since this is the width of a memory address. The program counter (*PC*) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The *PC* goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to *PC* to become the address of the next instruction. To read an instruction, the content of *PC* is taken as the address for memory and a memory read cycle is initiated. *PC* is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (*INPR*) receives an 8-bit character from an input device. The output register (*OUTR*) holds an 8-bit character for an output device.

```
     11                    0
     ┌──────────────────────┐
     │          PC          │
     └──────────────────────┘


     11                    0
     ┌──────────────────────┐              ┌──────────────────────────┐
     │          AR          │              │                          │
     └──────────────────────┘              │         Memory           │
                                           │       4096 words         │
     15                    0               │     16 bits per word     │
     ┌──────────────────────┐              │                          │
     │          IR          │              └──────────────────────────┘
     └──────────────────────┘


     15                    0               15                        0
     ┌──────────────────────┐              ┌──────────────────────────┐
     │          TR          │              │           DR             │
     └──────────────────────┘              └──────────────────────────┘


     7        0  7         0               15                        0
     ┌──────────┐ ┌──────────┐             ┌──────────────────────────┐
     │   OUTR   │ │   INPR   │             │           AC             │
     └──────────┘ └──────────┘             └──────────────────────────┘
```

Figure 5-3   Basic computer registers and memory.

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit (to be presented in Sec. 5-4). Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We have shown in Sec. 4-3 how to construct a bus system using multiplexers or three-state buffer gates. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 5-4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The *load* (LD) particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

```
 15  14      12  11                    0
 ┌───┬──────────┬──────────────────────┐
 │ I │  Opcode  │       Address        │
 └───┴──────────┴──────────────────────┘
```

(a) Instruction format

(b) Direct address   (c) Indirect address

Figure 5-2 Demonstration of direct and indirect address.

data. The pointer could be placed in a processor register instead of memory as done in commercial computers.

5-2 Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction code after it is read

from memory. The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in Fig. 5-3. The registers are also listed in Table 5-1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation, part of the instruction and a bit to specify a direct or indirect address. The data register (DR) holds the operand read from memory. The accumulator (AC) register is a general-purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing.

TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

program counter (PC)

The memory address register (AR) has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register ($INPR$) receives an 8-bit character from an input device. The output register ($OUTR$) holds an 8-bit character for an output device.
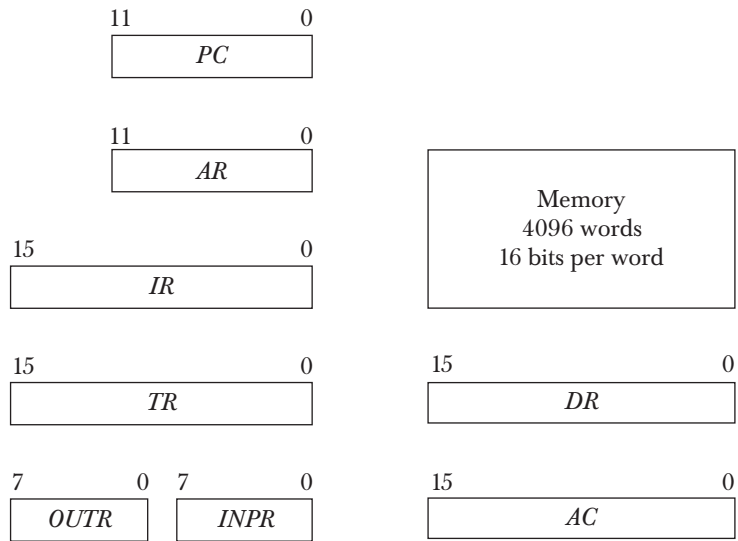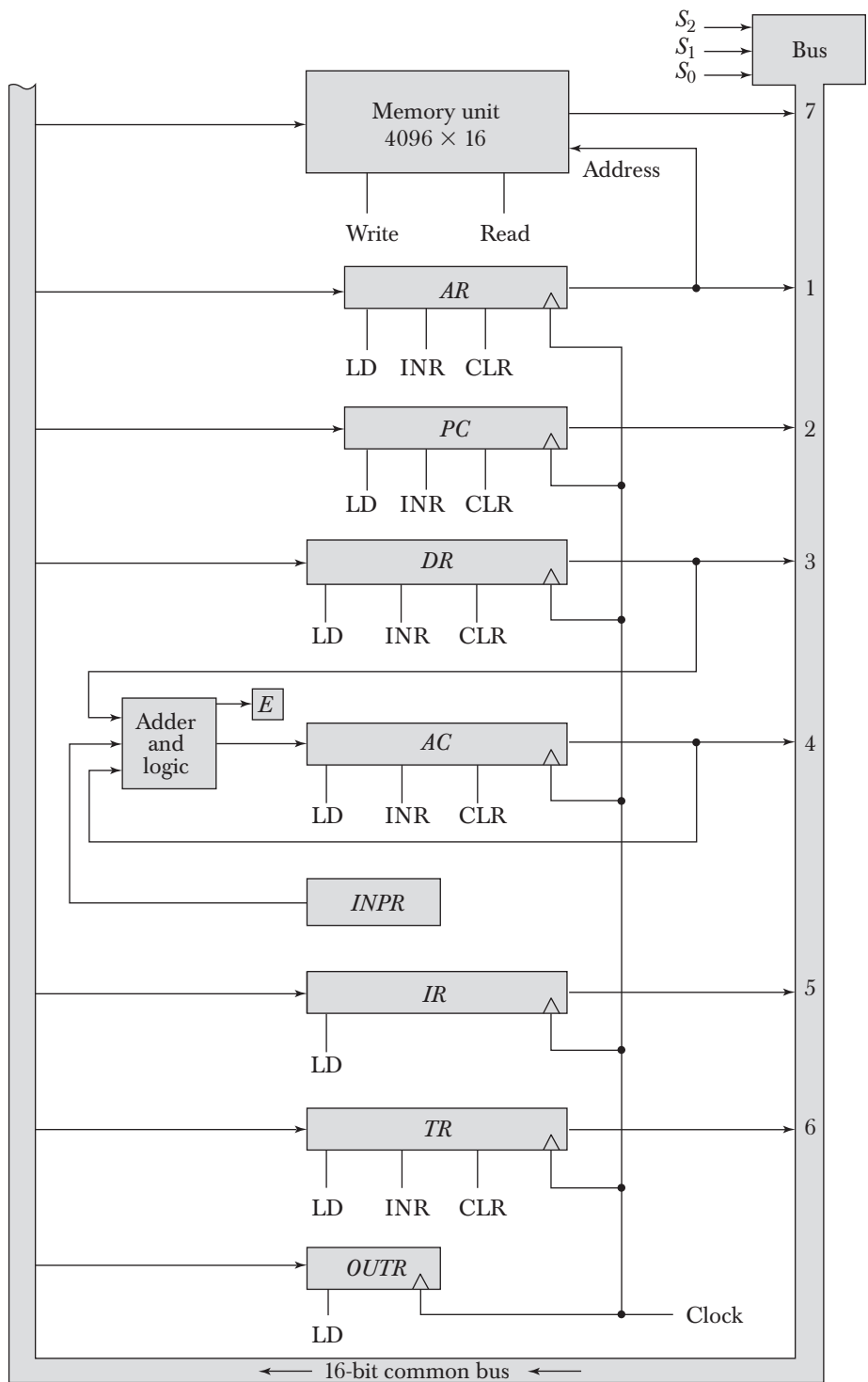
11   0

PC

11   0

AR

Memory 4096 words 16 bits per word

15   0

IR

15   0   15   0

TR		DR

7   0 7   0   15   0

OUTR	INPR		AC

Figure 5-3 Basic computer registers and memory.

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit (to be presented in Sec. 5-4). Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. We have shown in Sec. 4-3 how to construct a bus system using multiplexers or three-state buffer gates. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 5-4.

      The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_0. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The

load (LD)

particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

132

Four registers, *DR*, *AC*, *IR*, and *TR*, have 16 bits each. Two registers, *AR* and *PC*, have 12 bits each since they hold a memory address. When the contents of *AR* or *PC* are applied to the 16-bit common bus, the four most significant bits are set to 0's. When *AR* or *PC* receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register *INPR* and the output register *OUTR* have 8 bits each and communicate with the eight least significant bits in the bus. *INPR* is connected to provide information to the bus but *OUTR* can only receive information from the bus. This is because *INPR* receives a character from an input device which is then transferred to *AC*. *OUTR* receives a character from *AC* and delivers it to an output device. There is no transfer from *OUTR* to any of the other registers.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). This type of register is equivalent to a binary counter with parallel load and synchronous clear similar to the one shown in Fig. 2-11. The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input. This type of register is shown in Fig. 2-7.

memory address   The input data and output data of the memory are connected to the common bus, but the memory address is connected to *AR*. Therefore, *AR* must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except *AC*.

The 16 inputs of *AC* come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of *AC*. They are used to implement register microoperations such as complement *AC* and shift *AC*. Another set of 16-bit inputs come from the data regisler *DR*. The inputs from *DR* and *AC* are used for arithmetic and logic microoperations, such as add *DR* to *AC* or AND *DR* to *AC*. The result of an addition is transferred to *AC* and the end carry-out of the addition is transferred to flip-flop *E* (extended *AC* bit). A third set of 8-bit inputs come from the input register *INPR*. The operation of *INPR* and *OUTR* is explained in Sec. 5-7.

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into *AC*. For example, the two microoperations

$$DR \leftarrow AC \quad \text{and} \quad AC \leftarrow DR$$

can be executed at the same time. This can be done by placing the content of *AC* on the bus (with $S_2 S_1 S_0 = 100$), enabling the LD (load) input of *DR*,

transferring the content of *DR* through the adder and logic circuit into *AC*, and enabling the LD (load) input of *AC*, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.
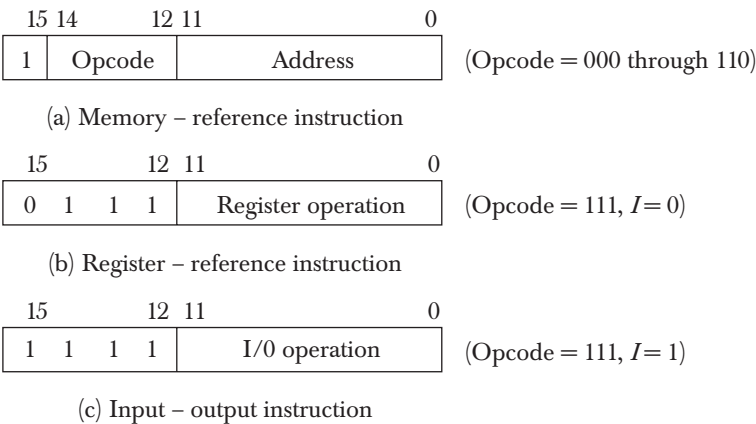
5-3   Computer Instructions

Instruction format

The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode *I*. *I* is equal to 0 for direct address and to 1 for indirect address (see Fig. 5-2). The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the *AC* register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input–output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the left-most bit of the instruction. The remaining 12 bits are used to specify the type of input–output operation or test performed.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode *I*. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If

Figure 5-5   Basic computer instruction formats.

(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input–output type. Note that the bit in position 15 of the instruction code is designated by the symbol I but is not used as a mode bit when the operation code is equal to 111.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register-reference and input–output instructions use the remaining 12 bits as part of the operation code, the total number of instructions can exceed eight. In fact, the total number of instructions chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 5-2. The symbol designation is a three-letter word and represents an abbreviation intended for

TABLE 5-2 Basic Computer Instructions

Symbol	Hexadecimal code		Description
	$I = 0$	$I = 1$	
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

hexadecimal code   programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When $I = 0$, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When $I = 1$, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binar) equivalent of the remaining 12 bits. The input–output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Instruction Set Completeness

Before investigating the operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision-making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the

transferring the content of *DR* through the adder and logic circuit into *AC*, and enabling the LD (load) input of *AC*, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.
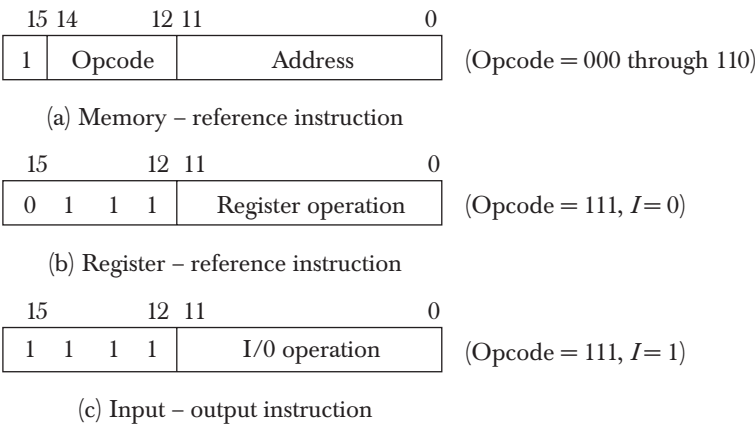
5-3   Computer Instructions

Instruction format

The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode *I*. *I* is equal to 0 for direct address and to 1 for indirect address (see Fig. 5-2). The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the *AC* register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input–output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input–output operation or test performed.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode *I*. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If

Figure 5-5   Basic computer instruction formats.

(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input–output type. Note that the bit in position 15 of the instruction code is designated by the symbol I but is not used as a mode bit when the operation code is equal to 111.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register-reference and input–output instructions use the remaining 12 bits as part of the operation code, the total number of instructions can exceed eight. In fact, the total number of instructions chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 5-2. The symbol designation is a three-letter word and represents an abbreviation intended for

TABLE 5-2 Basic Computer Instructions

Symbol	Hexadecimal code		Description
	$I = 0$	$I = 1$	
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

hexadecimal code   programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol *I*. When $I = 0$, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When $I = 1$, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to *E* since the last bit is 1.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits. The input–output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Instruction Set Completeness

Before investigating the operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision-making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the

user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

The instructions listed in Table 5-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC(CMA), and clear AC(CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables (listed in Table 4-6). Moving information from memory to AC is accomplished with the load AC(LDA) instruction. Storing information from AC into memory is done with the store AC(STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. The programs are presented in Chap. 6 together with other programming examples for the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way we can demonstrate the basic principles of computer organization and design without going into excessive complex details. In Chap. 8 we present a complete list of computer instructions that are included in most commercial computers.

The function of each instruction listed in Table 5-2 and the microoperations needed for their execution are presented in Secs. 5-5 through 5-7 We delay this discussion because we must first consider the control unit and understand its internal organization.

5-4 Timing and Control

clock pulses

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a

control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

hardwired control   There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section. A microprogrammed control unit for a similar computer is presented in Chap. 7.

microprogrammed control

control unit   The block diagram of the control unit is shown in Fig. 5-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Fig. 5-4. The instruction register is shown again in Fig. 5-6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3×8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the *timing signals*   counter are decoded into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

The sequence counter SC can be incremented or cleared synchronously (see the counter of Fig. 2-11). Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4×16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_0. As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3 T_4: \quad SC \leftarrow 0$$

The timing diagram of Fig. 5-7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock

Instruction register (*IR*)

Figure 5-6 Control unit of basic computer.

clears *SC* to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. *SC* is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.) If *SC* is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

The instructions listed in Table 5-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC(CMA), and clear AC(CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables (listed in Table 4-6). Moving information from memory to AC is accomplished with the load AC(LDA) instruction. Storing information from AC into memory is done with the store AC(STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. The programs are presented in Chap. 6 together with other programming examples for the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way we can demonstrate the basic principles of computer organization and design without going into excessive complex details. In Chap. 8 we present a complete list of computer instructions that are included in most commercial computers.

The function of each instruction listed in Table 5-2 and the microoperations needed for their execution are presented in Secs. 5-5 through 5-7 We delay this discussion because we must first consider the control unit and understand its internal organization.

5-4   Timing and Control

clock pulses

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a

hardwired control

control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various compo-

microprogrammed control

nents if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section. A microprogrammed control unit for a similar computer is presented in Chap. 7.

control unit

The block diagram of the control unit is shown in Fig. 5-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Fig. 5-4. The instruction register is shown again in Fig. 5-6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3×8 decoder. The eight outputs of the decoder are designated by the symbols D_0 through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the

timing signals

counter are decoded into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

The sequence counter SC can be incremented or cleared synchronously (see the counter of Fig. 2-11). Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4×16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_0. As an example, consider the case where SC is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3 T_4: \quad SC \leftarrow 0$$

The timing diagram of Fig. 5-7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock

Instruction register (*IR*)

Figure 5-6   Control unit of basic computer.

clears *SC* to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. *SC* is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.) If *SC* is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

Figure 5-7   Example of control timing signals.

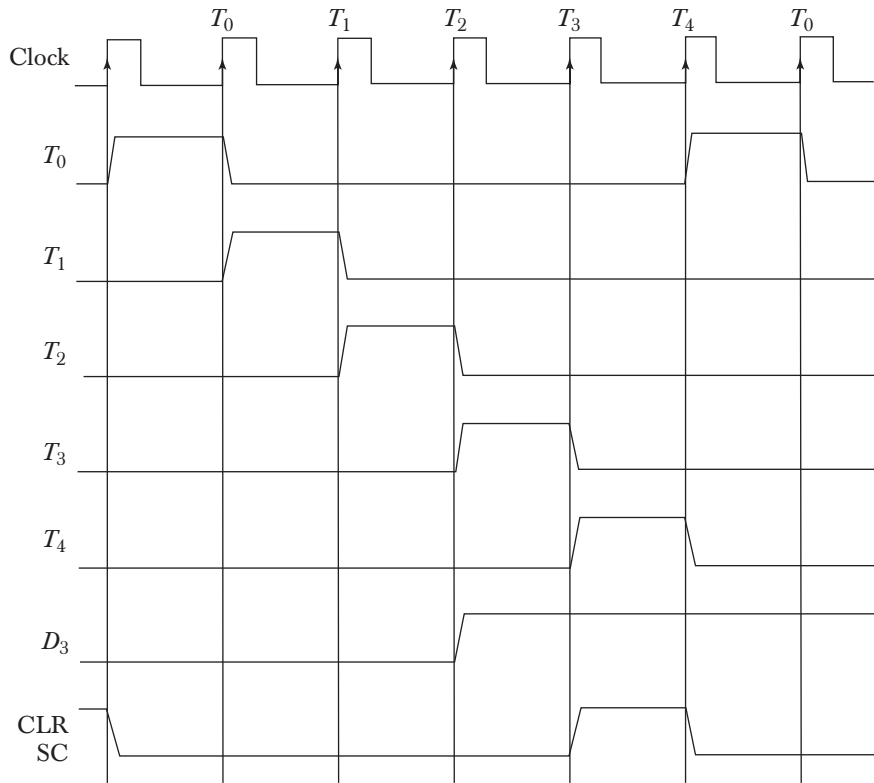The last three waveforms in Fig. 5-7 show how *SC* is cleared when $D_3 T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function $D_3 T_4$ becomes active. This signal is applied to the CLR input of *SC*. On the next positive clock transition (the one marked T_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if *SC* were incremented instead of cleared.

A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the

processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0: \quad AR \leftarrow PC$$

specifies a transfer of the content of PC into AR if timing signal T_0 is active. T_0 is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

5-5 Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

T_0: $AR \leftarrow PC$

T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$

T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0-11)$, $I \leftarrow IR(15)$

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing

Figure 5-8   Register transfers for the fetch phase.
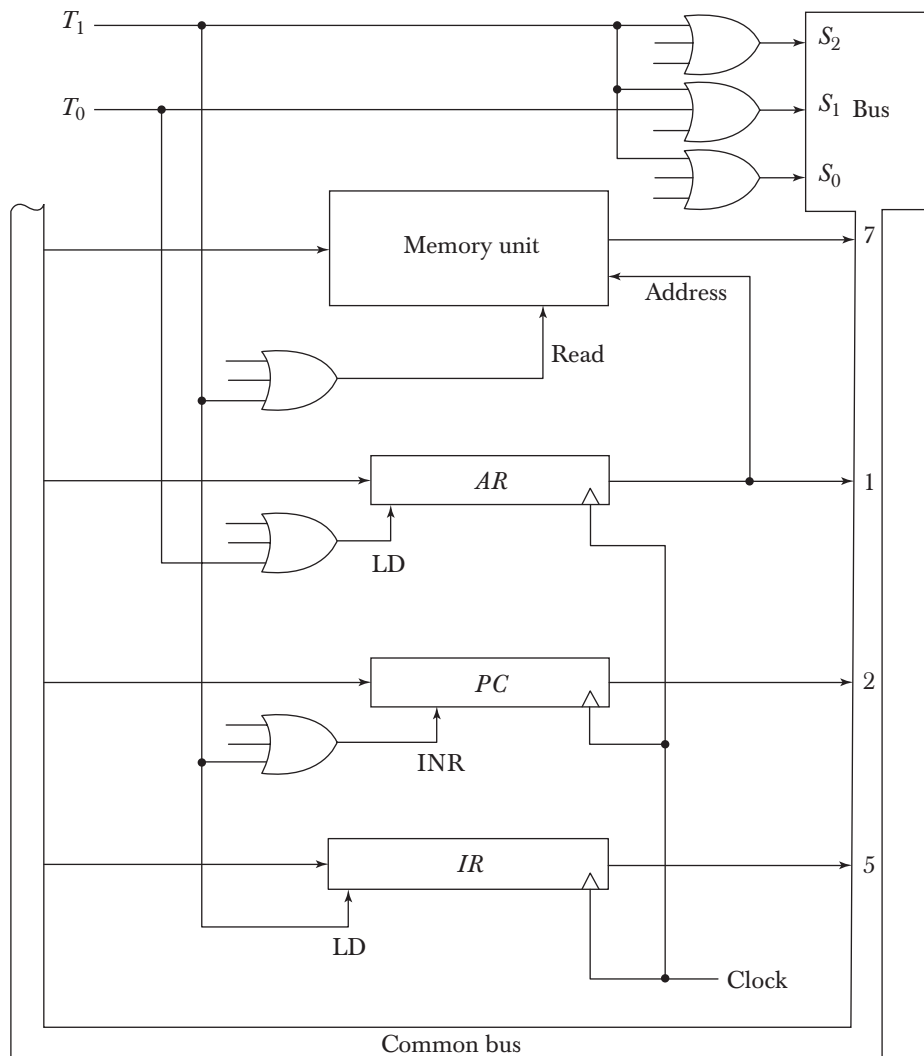
signal T_1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At rime T_2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T_0, T_1, and T_2.

Figure 5-8 shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

The next clock transition initiates the transfer from PC to AR since $T_0 = 1$. In order to implement the second statement

$$T_1: \qquad IR \leftarrow M[AR], \qquad PC \leftarrow PC + 1$$

it is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since $T_1 = 1$.

Figure 5-8 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

Determine the Type of Instruction

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. 5-9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. 5-5.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. 5-5 we determine that if $D_7 = 1$, the instruction must be a
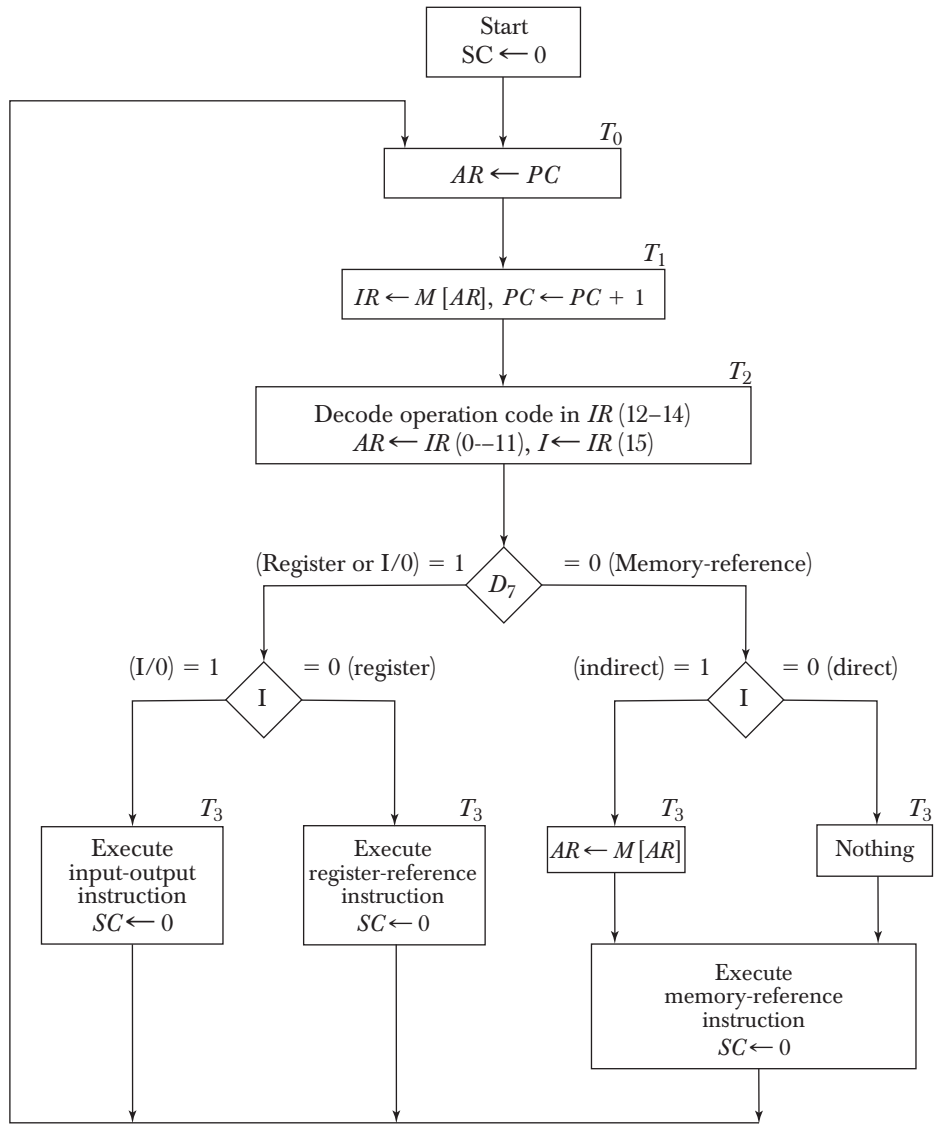
Figure 5-9   Flowchart for instruction cycle (initial configuration).

register-reference or input–output type. If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If $D_7 = 0$ and $I = 1$, we have a memory-reference instruction with an indirect address. It is then necessary to read the

indirect address effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement

$$AR \leftarrow M[AR]$$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

$$D_7' I T_3: \quad AR \leftarrow M[AR]$$
$$D_7' I T_3: \quad \text{Nothing}$$
$$D_7 I' T_3: \quad \text{Execute a register-reference instruction}$$
$$D_7 I T_3: \quad \text{Execute an input–output instruction}$$

When a memory-reference instruction with $I = 0$ is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D_7' T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4. A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with $T_0 = 1$.

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement $SC \leftarrow SC + 1$, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement $SC \leftarrow 0$.

The register transfers needed for the execution of the register-reference instructions are presented in this section. The memory-reference instructions are explained in the next section. The input-output instructions are included in Sec. 5-7.

Register-Reference Instructions

Register-reference instructions are recognized by the control when $D_7 = 1$ and $I = 0$. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR (0–11). They were also transferred to AR during time T_2.

The control functions and microoperations for the register-reference instructions are listed in Table 5-3. These instructions are executed with the

clock transition associated with timing variable T_3. Each control function needs the Boolean relation $D_7 I' T_3$, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in $IR(0-11)$. By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rB_i. For example, the instruction CLA has the hexadecimal code 7800 (see Table 5-2), which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D_7. Bit 11 in IR is 1 and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is $D_7 I' T_3 B_{11} = rB_{11}$. The execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1). The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC $(15) = 0$; it is negative when $AC(15) = 1$. The content of AC is zero $(AC = 0)$ if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start–stop flip-flop must be set manually.

TABLE 5-3 Execution of Register-Reference Instructions

$D_7 I' T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0-11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \bar{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC^* \rightarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

5-6   Memory-Reference Instructions

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 5-2, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

effective address

Table 5-4 lists the seven memory-reference instructions. The decoded output D_i for $i = 0, 1, 2, 3, 4, 5$, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when $I = 0$, or during timing signal T_3 when $I = 1$. The execution of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is presented at the end of this section.

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{\text{out}}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$,
		If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of

the operation is transferred to AC. The microoperations that execute this instruction are:

$$D_0 T_4: \quad DR \leftarrow M[AR]$$

$$D_0 T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

The control function for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into DR. The clock transition associated with the next timing signal T_5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T_0 to start a new instruction cycle.

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are

$$D_1 T_4: \quad DR \leftarrow M[AR]$$

$$D_1 T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

$$D_2 T_4: \quad DR \leftarrow M[AR]$$

$$D_2 T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into AC. The adder and logic circuit receive information

5-6 Memory-Reference Instructions

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 5-2, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

effective address

Table 5-4 lists the seven memory-reference instructions. The decoded output D_i for $i = 0, 1, 2, 3, 4, 5,$ and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when $I = 0$, or during timing signal T_3 when $I = 1$. The execution of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is presented at the end of this section.

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \land M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{\text{out}}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$
		If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of

the operation is transferred to *AC*. The microoperations that execute this instruction are:

$$D_0T_4: \qquad DR \leftarrow M[AR]$$

$$D_0T_5: \qquad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

The control function for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into *DR*. The clock transition associated with the next timing signal T_5 transfers to *AC* the result of the AND logic operation between the contents of *DR* and *AC*. The same clock transition clears *SC* to 0, transferring control to timing signal T_0 to start a new instruction cycle.

ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of *AC*. The sum is transferred into *AC* and the output carry C_{out} is transferred to the *E* (extended accumulator) flip-flop. The microoperations needed to execute this instruction are

$$D_1T_4: \qquad DR \leftarrow M[AR]$$

$$D_1T_5: \qquad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory-reference instruction.

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to *AC*. The microoperations needed to execute this instruction are

$$D_2T_4: \qquad DR \leftarrow M[AR]$$

$$D_2T_5: \qquad AC \leftarrow DR, \quad SC \leftarrow 0$$

Looking back at the bus system shown in Fig. 5-4 we note that there is no direct path from the bus into *AC*. The adder and logic circuit receive information

from DR which can be transferred into AC. Therefore, it is necessary to read the memory word into DR first and then transfer the content of DR into AC. The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of AC we can maintain one clock cycle per microoperation.

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

$$D_3T_4: \qquad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

$$D_4T_4: \qquad PC \leftarrow AR, \quad SC \leftarrow 0$$

The effective address from AR is transferred through the common bus to PC. Resetting SC to 0 transfers control to T_0. The next instruction, is then fetched and executed from the memory address given by the new value in PC.

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified in Table 5-4 with the following register transfer:

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 5-10. The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the *return address*). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

return address

$$M[135] \leftarrow 21, \qquad PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.

subroutine call

The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor register or in a portion of memory called a stack. This is discussed in more detail in Sec. 8-7.

Figure 5-10 Example of BSA instruction execution.

(a) Memory, PC, and AR at time T_4 (b) Memory and PC after execution
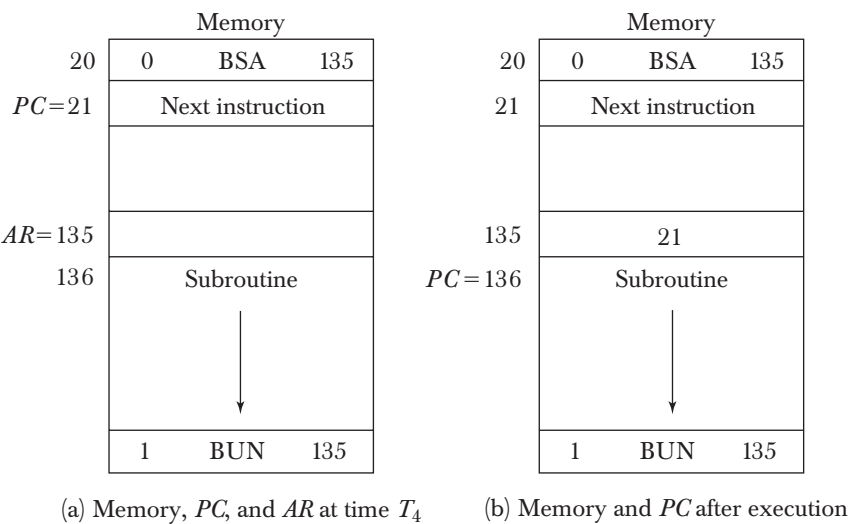
It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

ISZ: Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

Control Flowchart

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. 5-11. The control functions are indicated on top of each box. The microoperations that are performed during time T_4, T_5, or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

Note that we need only seven timing signals to execute the longest instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.
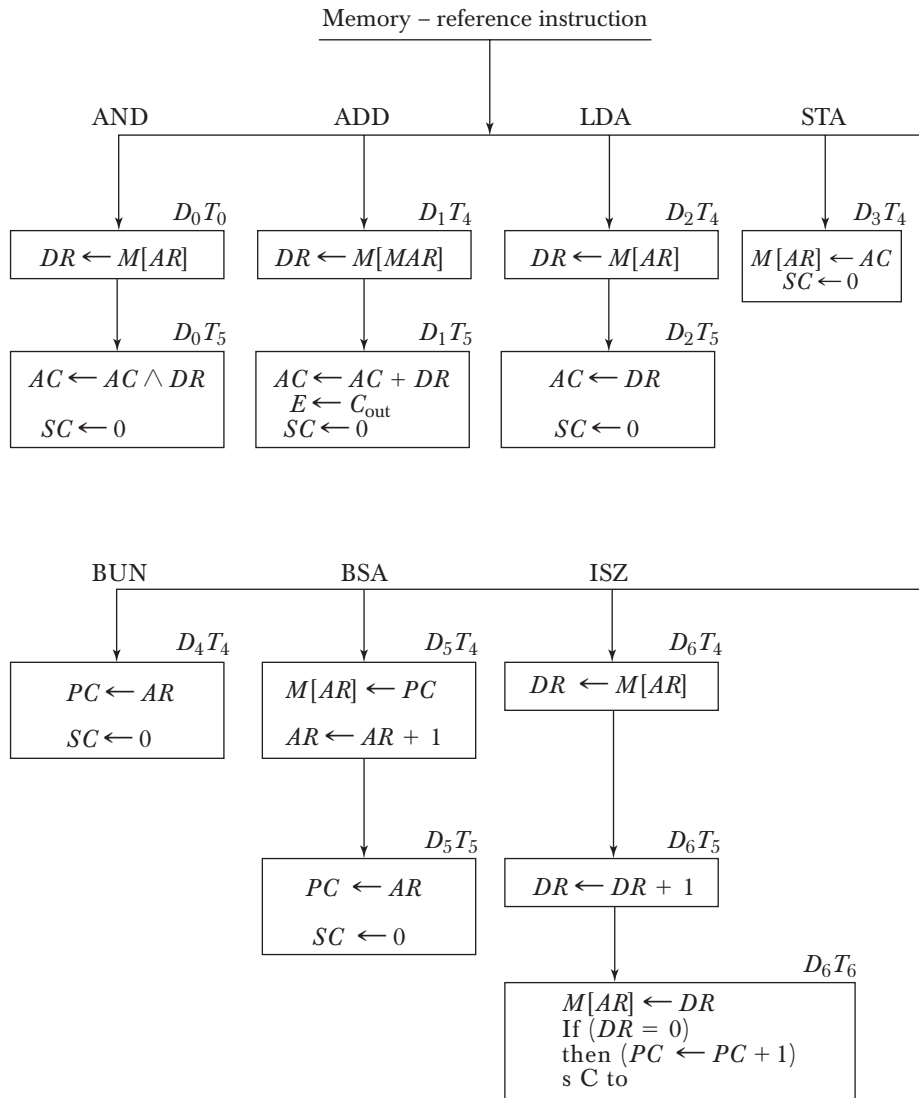
Figure 5-11   Flowchart for memory-reference instructions.

5-7   Input–Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types

of input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer. Input–output organization is discussed further in Chap. 11.
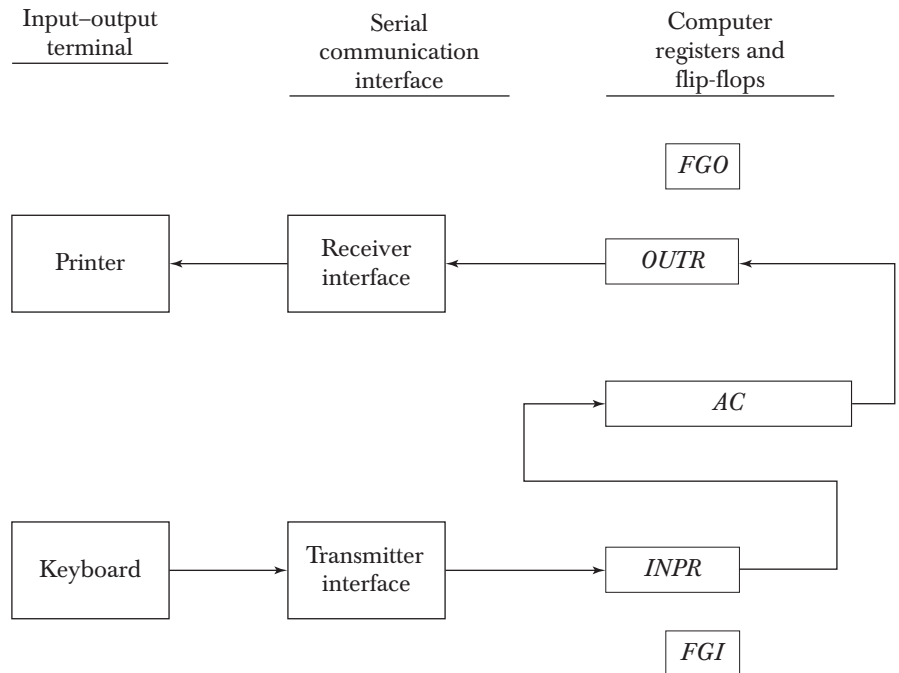
Input–Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register *INPR*. The serial information for the printer is stored in the output register *OUTR*. These two registers communicate with a communication interface serially and with the *AC* in parallel. The input–output configuration is shown in Fig. 5-12. The transmitter interface receives serial information from the keyboard and transmits it to *INPR*. The receiver interface receives information from *OUTR* and sends it to the printer serially. The operation of the serial communication interface is explained in Sec. 11-3.

input register   The input register *INPR* consists of eight bits and holds an alphanumeric input information. The 1-bit input flag *FGI* is a control flip-flop. The flag bit is

Figure 5-12   Input–output configuration.

set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag *FGI* is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into *INPR* and the input flag *FGI* is set to 1. As long as the flag is set, the information in *INPR* cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from *INPR* is transferred in parallel into *AC* and *FGI* is cleared to 0. Once the flag is cleared, new information can be shifted into *INPR* by striking another key.

output register   The output register *OUTR* works similarly but the direction of information flow is reversed. Initially, the output flag *FGO* is set to 1. The computer checks the flag bit; if it is 1, the information from *AC* is transferred in parallel to *OUTR* and *FGO* is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets *FGO* to 1. The computer does not load a new character into *OUTR* when *FGO* is 0 because this condition indicates that the output device is in the process of printing the character.

Input–Output Instructions

Input and output instructions are needed for transferring information to and from *AC* register, for checking the flag bits, and for controlling the interrupt facility. Input–output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input–output instructions are listed in Table 5-5. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in $IR(6–11)$. By assigning the symbol B_i to bit i of IR, all control functions can

TABLE 5-5 Input–Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(\text{i}) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear *SC*
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

5.7---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING
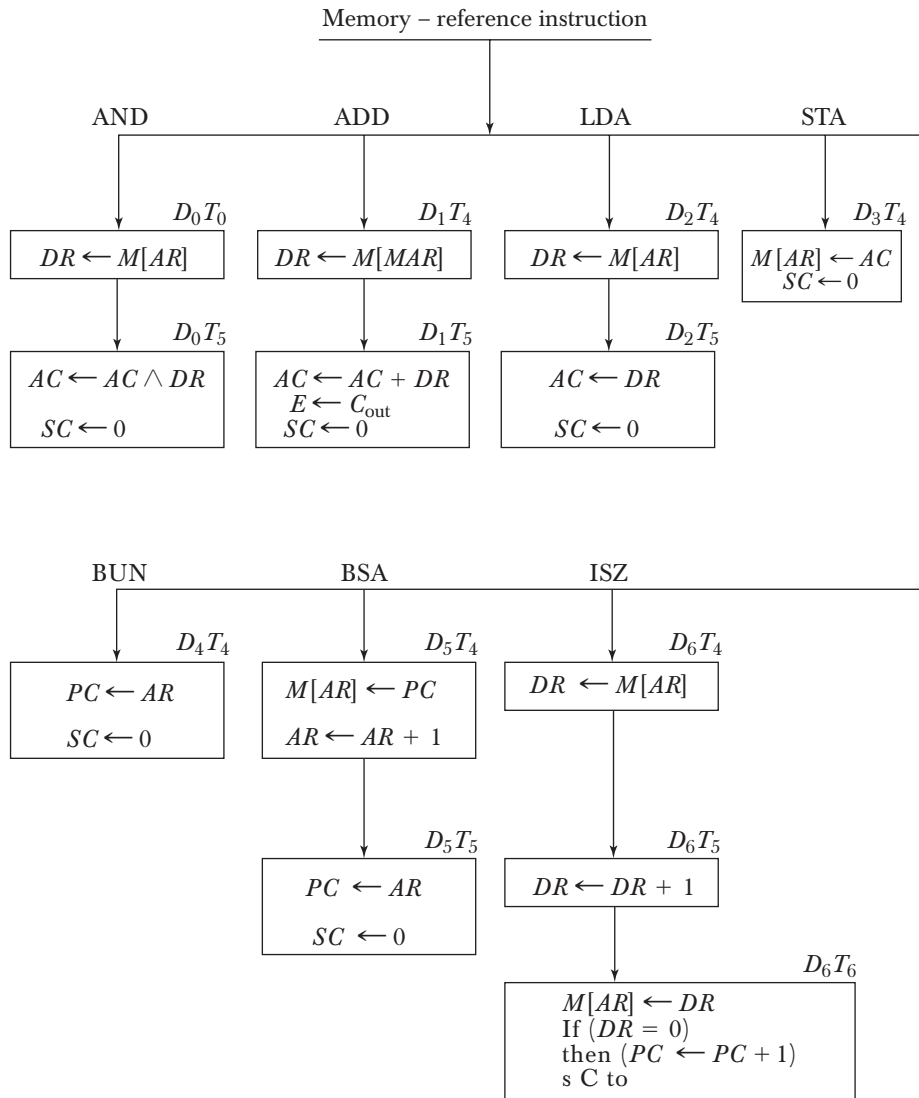
Figure 5-11 Flowchart for memory-reference instructions.

5-7 Input–Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types

of input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer. Input–output organization is discussed further in Chap. 11.
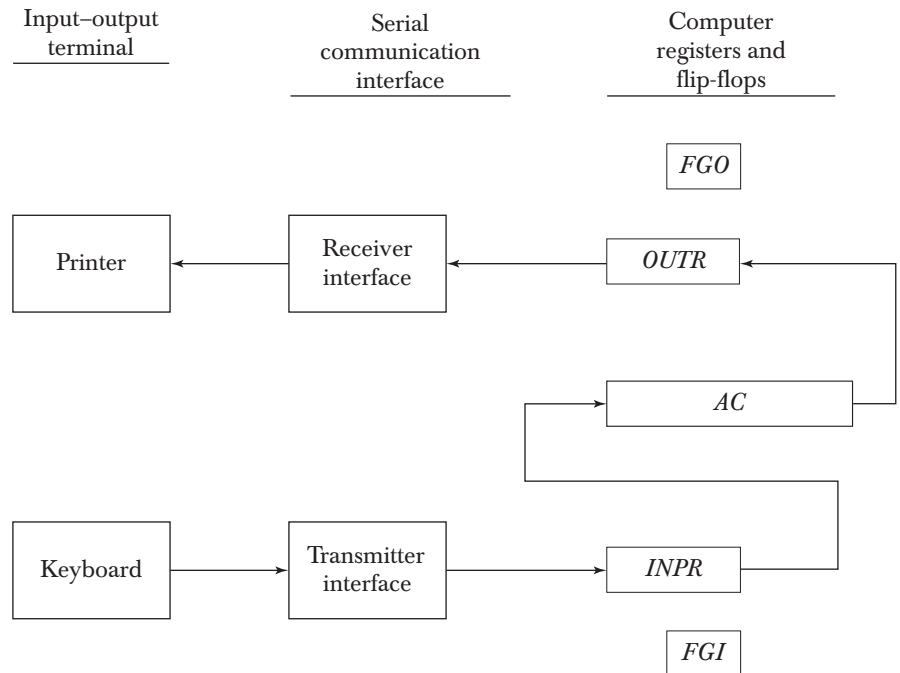
Input–Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register *INPR*. The serial information for the printer is stored in the output register *OUTR*. These two registers communicate with a communication interface serially and with the *AC* in parallel. The input–output configuration is shown in Fig. 5-12. The transmitter interface receives serial information from the keyboard and transmits it to *INPR*. The receiver interface receives information from *OUTR* and sends it to the printer serially. The operation of the serial communication interface is explained in Sec. 11-3.

input register   The input register *INPR* consists of eight bits and holds an alphanumeric input information. The 1-bit input flag *FGI* is a control flip-flop. The flag bit is

Figure 5-12   Input–output configuration.

set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag *FGI* is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into *INPR* and the input flag *FGI* is set to 1. As long as the flag is set, the information in *INPR* cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from *INPR* is transferred in parallel into *AC* and *FGI* is cleared to 0. Once the flag is cleared, new information can be shifted into *INPR* by striking another key.

output register   The output register *OUTR* works similarly but the direction of information flow is reversed. Initially, the output flag *FGO* is set to 1. The computer checks the flag bit; if it is 1, the information from *AC* is transferred in parallel to *OUTR* and *FGO* is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets *FGO* to 1. The computer does not load a new character into *OUTR* when *FGO* is 0 because this condition indicates that the output device is in the process of printing the character.

Input–Output Instructions

Input and output instructions are needed for transferring information to and from *AC* register, for checking the flag bits, and for controlling the interrupt facility. Input–output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input–output instructions are listed in Table 5-5. These instructions are executed with the clock transition associated with timing signal T_3. Each control function needs a Boolean relation $D_7 I T_3$, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in $IR(6-11)$. By assigning the symbol B_i to bit i of IR, all control functions can

TABLE 5-5 Input–Output Instructions

$D_7 I T_3 = p$ (common to all input–output instructions)
$IR(\text{i}) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear *SC*
INP	pB_{11}:	$AC(0-7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0-7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

be denoted by pB_i for $i = 6$ though 11. The sequence counter SC is cleared to 0 when $p = D_7IT_3 = 1$.

The INP instruction transfers the input information from $INPR$ into the eight low-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register $OUTR$ and clears the output flag to 0. The next two instructions in Table 5-5 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. (Examples of input and output programs are given in Sec. 6-8.) The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input–output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs. Assume that the input–output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take eare of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.
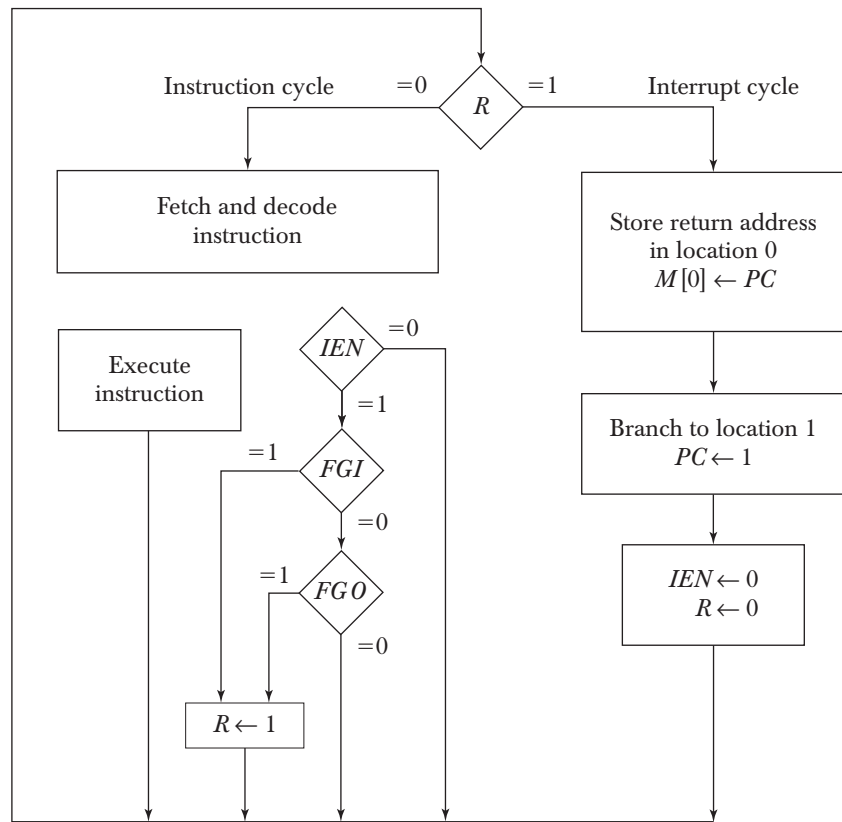
Figure 5-13   Flowchart for interrupt cycle.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-13. An interrupt flip-flop R is included in the computer. When $R = 0$, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If *IEN* is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while *IEN* = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

interrupt cycle   The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in *PC* is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor

register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into *PC* and clears *IEN* and *R* so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Fig. 5-14. Suppose that an interrupt occurs and *R* is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in *PC*. The programmer has previously placed an input–output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 5-14(a).

When control reaches timing signal T_0 and finds that $R = 1$, it proceeds with the interrupt cycle. The content of *PC* (256) is stored in memory location 0, *PC* is set to 1, and *R* is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of *PC*. The branch instruction at address 1 causes the program to transfer to the input–output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set *IEN* to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 5-14(b).

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction

Figure 5-14   Demonstration of the interrupt cycle.

(a) Before interrupt   (b) After interrupt cycle
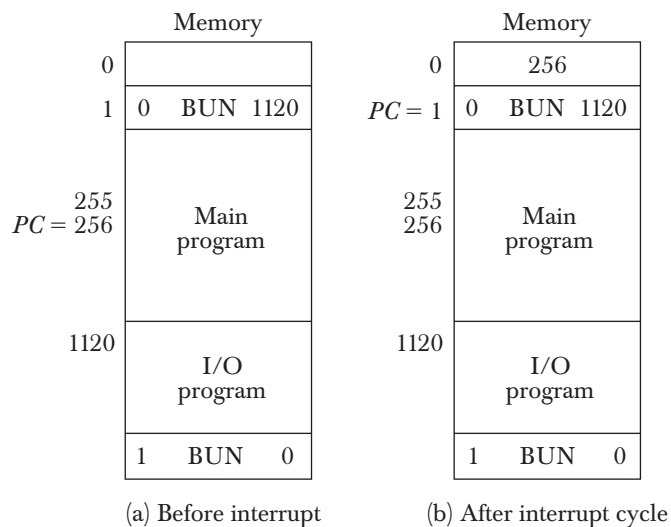
is read from memory during the fetch phase, control goes to the indirect phase (because $I = 1$) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

Interrupt Cycle

We are now ready to list the register transfer statements for the interrupt cycle. The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if $IEN = 1$ and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$$T_0' T_1' T_2'(IEN)(FGI + FGO): \quad R \leftarrow 1$$

modified fetch phase

The symbol $+$ between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and $T_0' T_1' T_2'$.

　　We now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Fig. 5-9) we will AND the three timing signals with R' so that the fetch and decode phases will be recognized from the three control functions $R' T_0$, $R' T_1$, and $R' T_2$. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if $R = 0$. Otherwise, if $R = 1$, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of microoperations:

$$RT_0: \quad AR \leftarrow 0, \quad TR \leftarrow PC$$
$$RT_1: \quad M[AR] \leftarrow TR, \quad PC \leftarrow 0$$
$$RT_2: \quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0$$

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_0 by clearing SC to 0. The beginning of the next instruction cycle has the condition $R' T_0$ and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

5-8 Complete Computer Description

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in Fig. 5-15. The interrupt flip-flop R may be set at any time during the indirect or execute phases. Control returns to timing signal T_0 after SC is cleared to 0. If $R = 1$, the computer goes through an interrupt cycle. If $R = 0$, the computer goes through an instruction cycle. If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the flowchart of Fig. 5-11. If the instruction is one of the register-reference instructions, it is executed with one of the microoperations listed in Table 5-3. If it is an input–output instruction, it is executed with one of the microoperations listed in Table 5-5.

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table. The entries in the table are taken from Figs. 5-11 and 5-16, and Tables 5-3 and 5-5.

The control functions and microoperations for the entire computer are summarized in Table 5-6. The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer. The control functions and conditional control statements listed in the table formulate the Boolean functions for the gates in the control unit. The list of microoperations specifies the type of control inputs needed for the registers and memory. A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

5-9 Design of Basic Computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each
2. Nine registers: *AR*, *PC*, *DR*, *AC*, *IR*, *TR*, *OUTR*, *INPR*, and *SC*
3. Seven flip-flops: *I, S, E, R, IEN, FGI,* and *FGO*
4. Two decoders: a 3×8 operation decoder and a 4×16 timing decoder
5. A 16-bit common bus
6. Control logic gates
7. Adder and logic circuit connected to the input of *AC*

The functional block diagram of the hypothetic BASIC computer is as shown below:
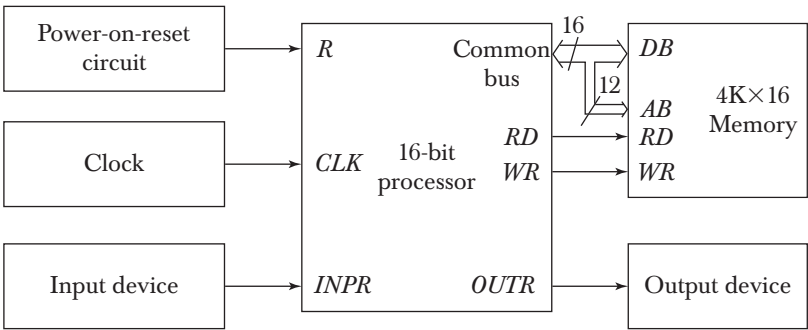
Figure 5-15 Flow chart for hypothetic basic computer.

The memory unit is a standard component that can be obtained readily from a commercial source. The registers are of the type shown in Fig. 2-11 and are similar to integrated circuit type 74163. The flip-flops can be either of the D or JK type, as described in Sec. 1-6. The two decoders are standard components similar to the ones presented in Sec. 2-2. The common bus system can be constructed with sixteen 8×1 multiplexers in a configuration similar to the one shown in Fig. 4-3. We are now going to show how to design the control logic gates. The next section deals with the design of the adder and logic circuit associated with AC.

Control Logic Gates

control unit

The block diagram of the control logic gates is shown in Fig. 5-6. The inputs to this circuit come from the two decoders, the I flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic are: AC bits 0 through 15 to check if $AC = 0$ and to detect the sign bit in $AC(15)$; DR bits 0 through 15 to check if $DR = 0$; and the values of the seven flip-flops.

The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers
2. Signals to control the read and write inputs of memory
3. Signals to set, clear, or complement the flip-flops
4. Signals for S_2, S_1, and S_0 to select a register for the bus
5. Signals to control the AC adder and logic circuit

The specifications for the various control signals can be obtained directly from the list of register transfer statements in Table 5-6.

Control of Registers and Memory

The registers of the computer connected to a common bus system are shown in Fig. 5-4. The control inputs of the registers are LD (load), INR (increment),

characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only two categories used from this classification are SIMD array processors discussed in Sec. 9-7, and MIMD multiprocessors presented in Chap. 13.

Parallel processing computers are required to meet the demands of large scale computations in many scientific, engineering, military, medical, artificial intelligence, and basic research areas. The following are some representative applications of parallel processing computers: Numerical weather forecasting, computational aerodynamics, finite-element analysis, remote-sensing applications, genetic engineering, computer-asseted tomography, and weapon research and defence.

In this chapter we consider parallel processing under the following main topics:

1. Pipeline processing
2. Vector processing
3. Array processors

Pipeline processing is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

9-2   Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the suboperation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all

segment activity. In this way the information flows through the pipeline one step at a time.

an example   The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The suboperations performed in each segment of the pipeline are as follows:

$$R1 \leftarrow A_i, \quad R2 \leftarrow B_i \qquad \text{Input } A_i \text{ and B}_i$$
$$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i \qquad \text{Multiply and input } C_i$$
$$R5 \leftarrow R3 + R4 \qquad \text{Add } C_i \text{ to product}$$

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1. The first clock pulse transfers A_1 and B_1 into

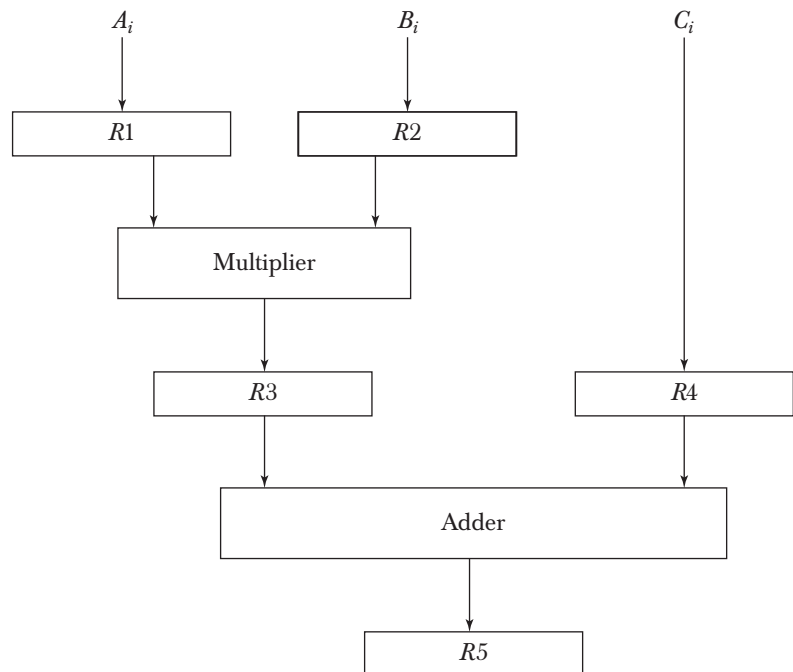**Figure 9-2** Example or pipeline processing.

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	–	–	–
2	A_2	B_2	$A_1 * B_1$	C_1	–
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	–	–	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	–	–	–	–	$A_7 * B_7 + C_7$

$R1$ and $R2$. The second clock pulse transfers the product of $R1$ and $R2$ into $R3$ and C_1 into $R4$. The same clock pulse transfers A_2 and B_2 into $R1$ and $R2$. The third clock pulse operates on all three segments simultaneously. It places A_3 and B_3 into $R1$ and $R2$, transfers the product of $R1$ and $R2$ into $R3$, transfers C_2 into $R4$, and places the sum of $R3$ and $R4$ into $R5$. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

General Considerations

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a four-segment pipeline is illustrated in Fig. 9-3. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a *task* as the total operation performed going through all the segments in the pipeline.

task

space-time diagram

The behavior of a pipeline can be illustrated with a *space-time* diagram. This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4. The horizontal axis displays the time in clock cycles and the vertical axis gives
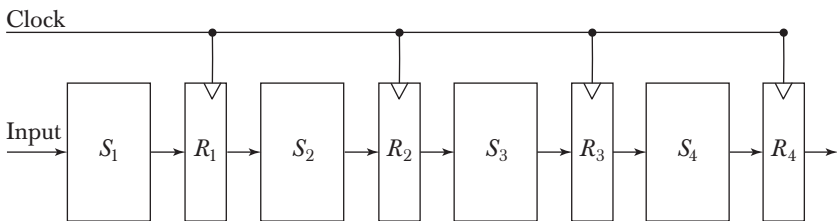
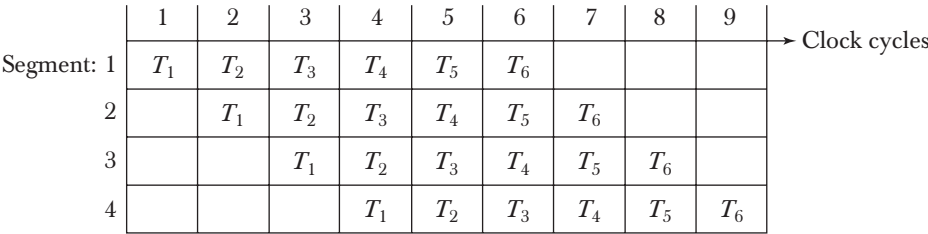Clock

Figure 9-3   Four-segment pipeline.

the segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_1, while segment 1 is busy with task T_2. Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.

Now consider the case where a k-segment pipeline with a clock cycle time t_p is used to execute n tasks. The first task T_1 requires a time equal to kt_p to complete its operation since there are k segments in the pipe. The remaining $n - 1$ tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to $(n - 1)t_p$. Therefore, to complete n tasks using a k-segment pipeline requires $k + (n - 1)$ clock cycles. For example, the diagram of Fig. 9-4 shows four segments and six tasks. The time required to complete all the operations is $4 + (6 - 1) = 9$ clock cycles, as indicated in the diagram.

Next consider a nonpipeline unit that performs the same operation and takes a time equal to t_n to complete each task. The total time required for n *speedup* tasks is nt_n. The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4   Space-time diagram for pipeline.

	1	2	3	4	5	6	7	8	9	
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				→ Clock cycles
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

As the number of tasks increases, n becomes much larger than $k - 1$, and $k + n - 1$ approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$. Including this assumption, the speedup reduces to

$$S = \frac{Kt_p}{t_p} = K$$

This shows that the theoretical maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a suboperation in each segment be equal to $t_p = 20$ ns. Assume that the pipeline has $k = 4$ segments and executes $n = 100$ tasks in sequence. The pipeline system will take $(k + n - 1)\, t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a nonpipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes 60/20 = 3.

To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct k identical units that will be operating in parallel. The implication is that a k-segment pipeline processor can be expected to equal the performance of k copies of an equivalent nonpipeline circuit under equal operating conditions. This is illustrated in Fig. 9-5, where four identical circuits are connected in parallel. Each P circuit performs the same task of an equivalent pipeline circuit. Instead of operating with the input data in sequence as in a pipeline, the parallel circuits accept four input data items simultaneously and perform four tasks at the same rime. As far as the speed of operation is concerned, this is equivalent to a four segment pipeline. Note that the four-unit circuit of Fig. 9-5 constitutes a single-instruction multiple-data (SIMD) organization since the same instruction is used to operate on multiple data in parallel.

There are various reasons why the pipeline cannot operate at its maximum theoretical rate. Different segments may take different times to complete their suboperation. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other
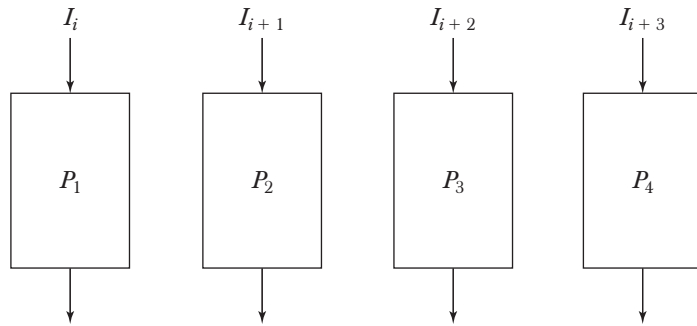
Figure 9-5   Multiple functional units in parallel.

segments to waste time while waiting for the next clock. Moreover, it is not always correct to assume that a nonpipe circuit has the same time delay as that of an equivalent pipeline circuit. Many of the intermediate registers will not be needed in a single-unit circuit, which can usually be constructed entirely as a combinational circuit. Nevertheless, the pipeline technique provids a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An *arithmetic pipeline* divides an arithmetic operation into sub-operations for execution in the pipeline segments. An *instruction pipeline* oper-ates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle. The two types of pipelines are explained in the following sections.

9-3   Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier as described in Fig. 10-10, with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into suboperations as demonstrated in Sec. 10-5. We will now show an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^{a}$$
$$Y = B \times 2^{b}$$

A and *B* are two fractions that represent the mantissas and *a* and *b* are the exponents. The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 9-6. The registers labeled *R* are placed between the segments to store intermediate results. The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

This follows the procedure outlined in the flowchart of Fig. 10-15 but with some variations that are used to reduce the execution time of the suboperations. The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Fig. 9-6 refers to binary numbers. Consider the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$
$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $3 - 2 = 1$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of *Y* to the right to obtain

$$X = 0.9504 \times 10^3$$
$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum
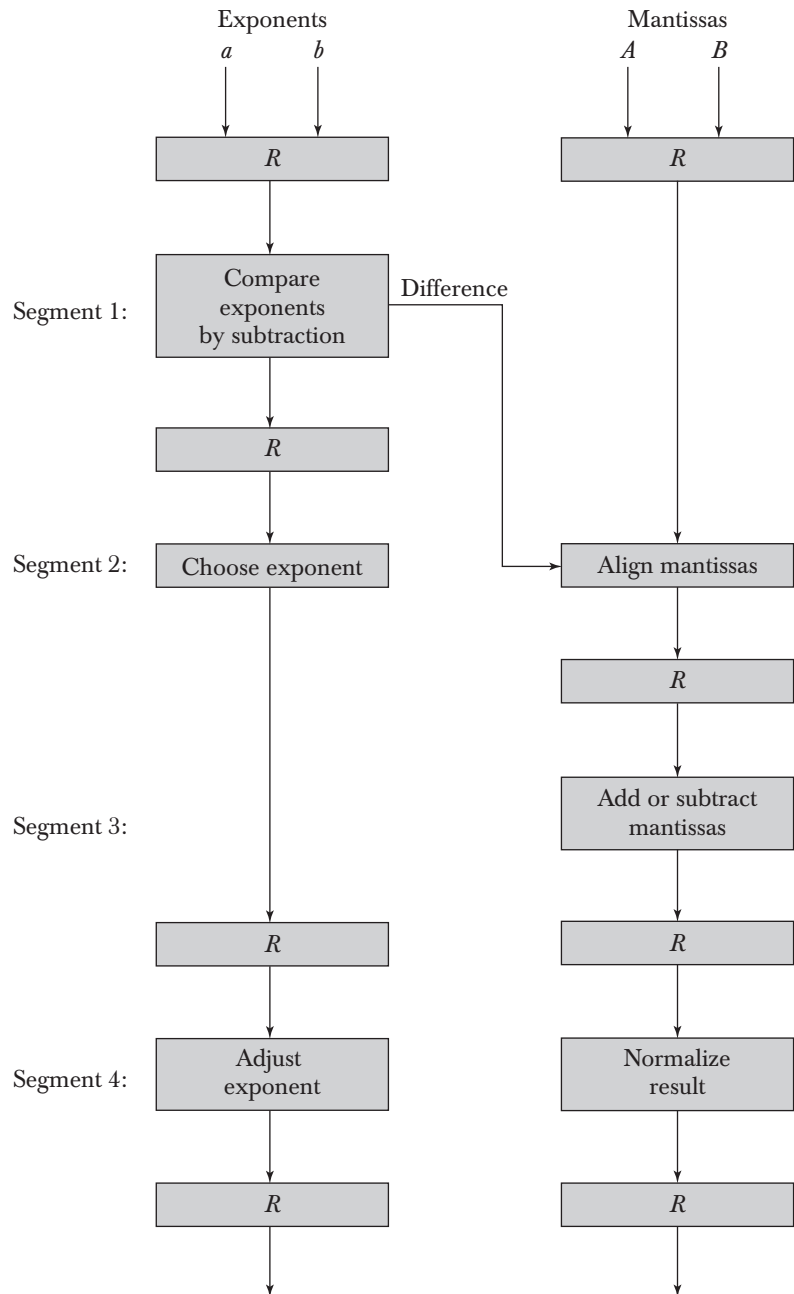
$$Z = 1.0324 \times 10^3$$

Figure 9-6 Pipeline for floating-point addition and subtraction.

8.1---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

Central Processing Unit

IN THIS CHAPTER

8-1   Introduction

CPU

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Fig. 8-1. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The CPU performs a variety of functions dictated by the type of instructions that art Incorporated in the computer. Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions. This includes the instruction formats, addressing modes, the instruction set, and the general organization of the CPU registers leading to two computer architectures as reduced instruction set computer (RISC) and complex instruction set computer (CISC). Based on memory usage for programs and data, two architectures, namely nonembedded and embedded are evolved. Nonembedded computer architectures are basically stored program computer (SPC) architectures in which programs and data reside in the same memory system. Embedded architectures are basically Harvard computer architectures in which programs and data
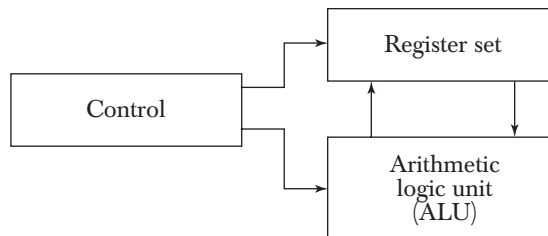
Figure 8-1   Major components of CPU.

reside in different memory systems leading to doubling the memory bandwidth. Example of nonembedded computers are all desktop systems such as personal computers. Example of embedded computers are microcontroller-based systems and digital signal processor-based (DSP) systems.

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions. The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

Design examples of simple CPUs are carried out in Chaps. 5 and 7. This chapter describes the organization and architecture of the CPU with an emphasis on the user's view of the computer. We briefly describe how the registers communicate with the ALU through buses and explain the operation of the memory stack. We then present the type of instruction formats available, the addressing modes used to retrieve data from memory, and typical instructions commonly incorporated in computers. The last section presents the concept of reduced instruction set computer (RISC).

8-2   General Register Organization

In the programming examples of Chap. 6, we have shown that memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

bus system

A bus organization for seven CPU registers is shown in Fig. 8-2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

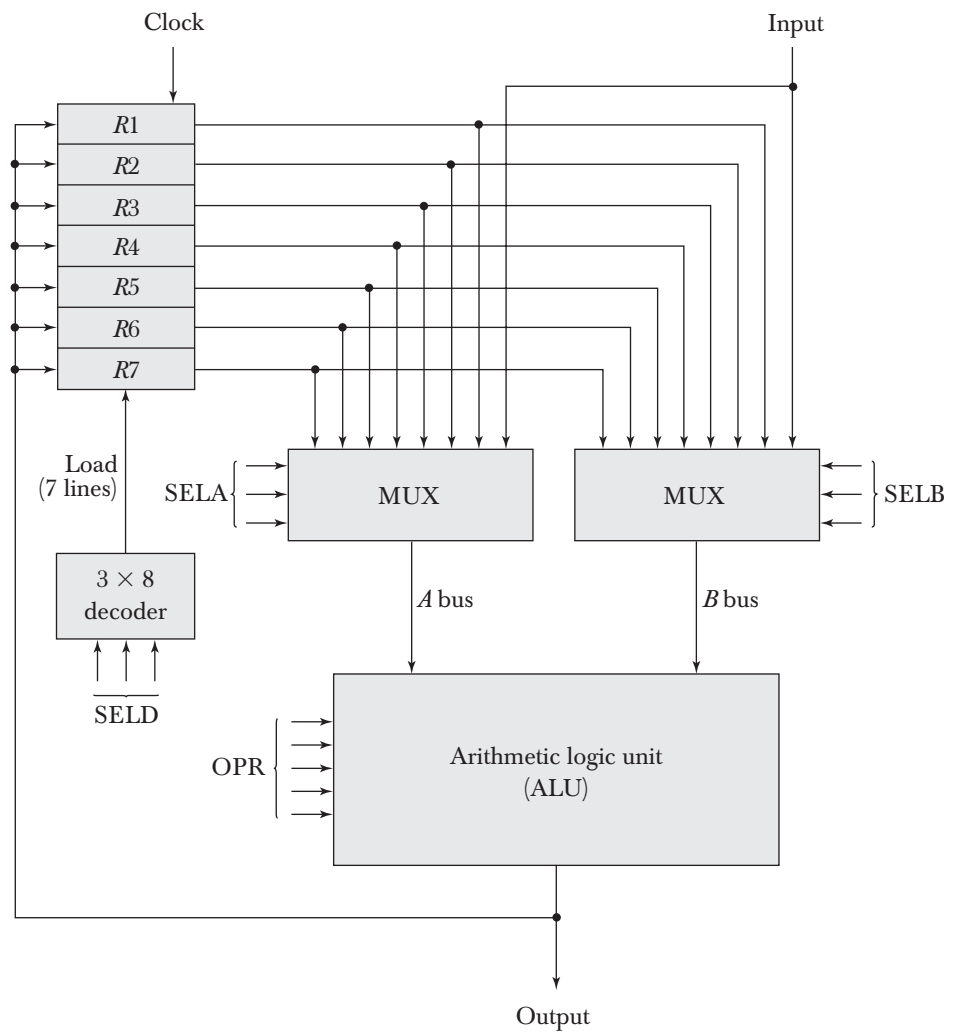the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of $R2$ into bus A.
2. MUX B selector (SELB): to place the content of $R3$ into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.
4. Decoder destination selector (SELD): to transfer the content of the output bus into $R1$.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary information from the output bus is transferred into $R1$. To achieve a fast response time, the ALU is constructed with high-speed circuits. The buses are implemented with multiplexers or three-state gates, as shown in Sec. 4-3.

Control Word

control word

There are 14 binary selection inputs in the unit, and their combined value specifies a *control word*. The 14-bit control word is defined in Fig. 8-2(b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

(a) Block diagram

3	3	3	5
SELA	SELB	SELD	OPR

(b) Control word

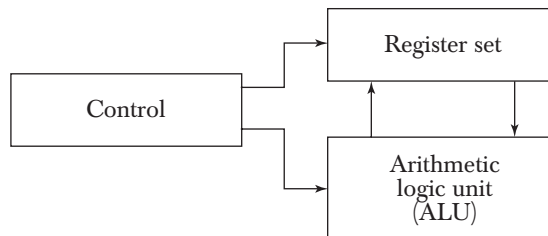**Figure 8-2**   Register set with common ALU.

Figure 8-1   Major components of CPU.

reside in different memory systems leading to doubling the memory bandwidth. Example of nonembedded computers are all desktop systems such as personal computers. Example of embedded computers are microcontroller-based systems and digital signal processor-based (DSP) systems.

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set. From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions. The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

Design examples of simple CPUs are carried out in Chaps. 5 and 7. This chapter describes the organization and architecture of the CPU with an emphasis on the user's view of the computer. We briefly describe how the registers communicate with the ALU through buses and explain the operation of the memory stack. We then present the type of instruction formats available, the addressing modes used to retrieve data from memory, and typical instructions commonly incorporated in computers. The last section presents the concept of reduced instruction set computer (RISC).

8-2   General Register Organization

In the programming examples of Chap. 6, we have shown that memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

bus system

A bus organization for seven CPU registers is shown in Fig. 8-2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

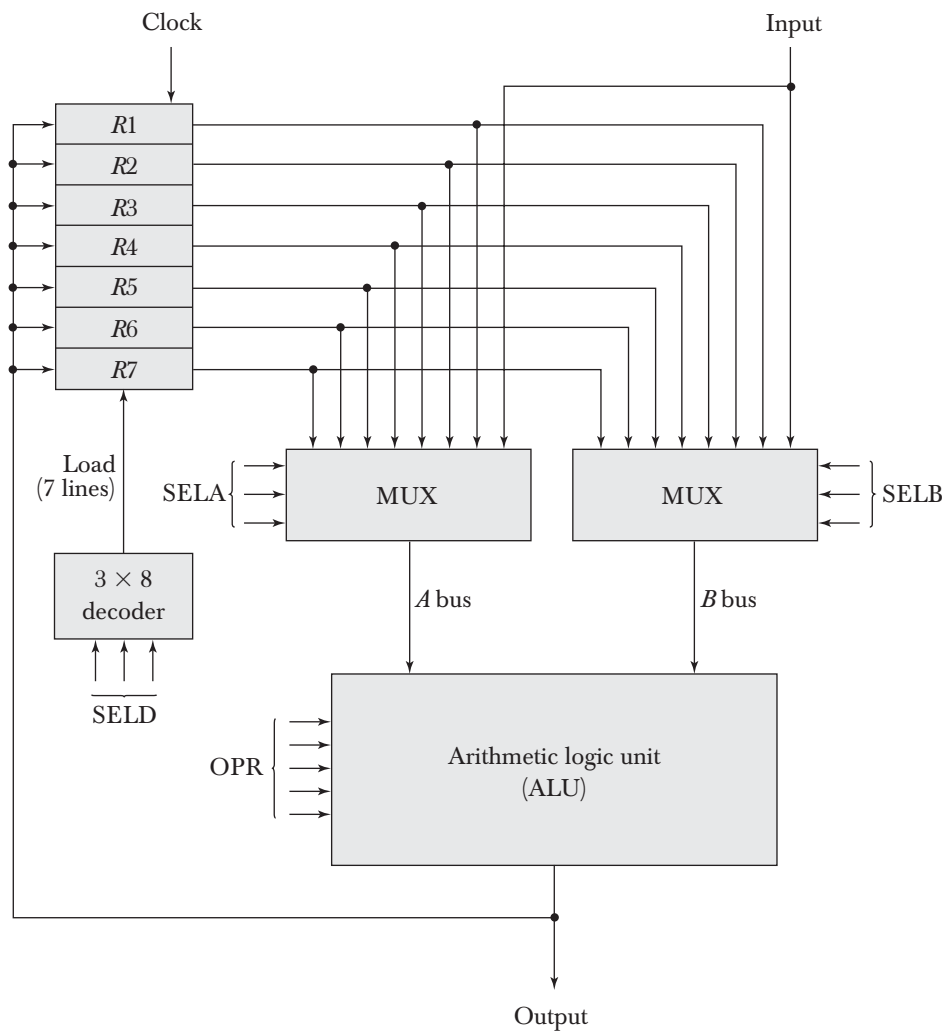the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of $R2$ into bus A.
2. MUX B selector (SELB): to place the content of $R3$ into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.
4. Decoder destination selector (SELD): to transfer the content of the output bus into $R1$.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary information from the output bus is transferred into $R1$. To achieve a fast response time, the ALU is constructed with high-speed circuits. The buses are implemented with multiplexers or three-state gates, as shown in Sec. 4-3.

Control Word

control word

There are 14 binary selection inputs in the unit, and their combined value specifies a *control word*. The 14-bit control word is defined in Fig. 8-2(b). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

(a) Block diagram

3	3	3	5
SELA	SELB	SELD	OPR

(b) Control word

Figure 8-2   Register set with common ALU.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

The encoding of the register selections is specified in Table 8-1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

ALU   The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of the ALU to provide a preshift capability, or at the output of the ALU to provide postshifting capability. In some cases, the shift operations are included with the ALU. An arithmetic logic and shift unit was designed in Sec. 4-7. The function table for this ALU is listed in Table 4-8. The encoding of the ALU operations for the CPU is taken from Sec. 4-7 and is specified in Table 8-2. The OPR field has five bits and each operation is designated with a symbolic name.

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add $A + B$	ADD
00101	Subtract $A - B$	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Examples of Microoperations

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

specifies $R2$ for the A input of the ALU, $R3$ for the B input of the ALU, $R1$ for the destination register, and an ALU operation to subtract $A - B$. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables 8-1 and 8-2. The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

The control word for this microoperation and a few others are listed in Table 8-3.

The increment and transfer microoperations do not use the B input of the ALU. For these cases, the B field is marked with a dash. We assign 000 to any unused field when formulating the binary control word, although any other binary number may be used. To place the content of a register into the output terminals we place the content of the register into the A input of the ALU, but none of the registers are selected to accept the data. The ALU operation TSFA places the data from the register, through the ALU, into the output terminals. The direct transfer from input to output is accomplished with a control word of all 0's (making the B field 000).

TABLE 8-3 Examples of Microoperations for the CPU

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	–	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	–	R7	TSFA	001 000 111 00000
Output $\leftarrow R2$	R2	–	None	TSFA	010 000 000 00000
Output $\leftarrow$ Input	Input	–	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	–	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as microprogrammed control. A microprogrammed control unit is shown in Fig. 7-8. The binary control word for the CPU will come from the outputs of the control memory marked "micro-ops."

8-3   Stack Organization

LIFO

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The

stack pointer

register that holds the address for the stack is called a stack pointer (*SP*) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called *push* (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called *pop* (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack'. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack. The stack pointer register *SP* contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: *A*, *B*, and *C*, in that order. Item *C* is on top of the stack so that the content of *SP* is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of *SP*. Item *B* is now on top of the stack
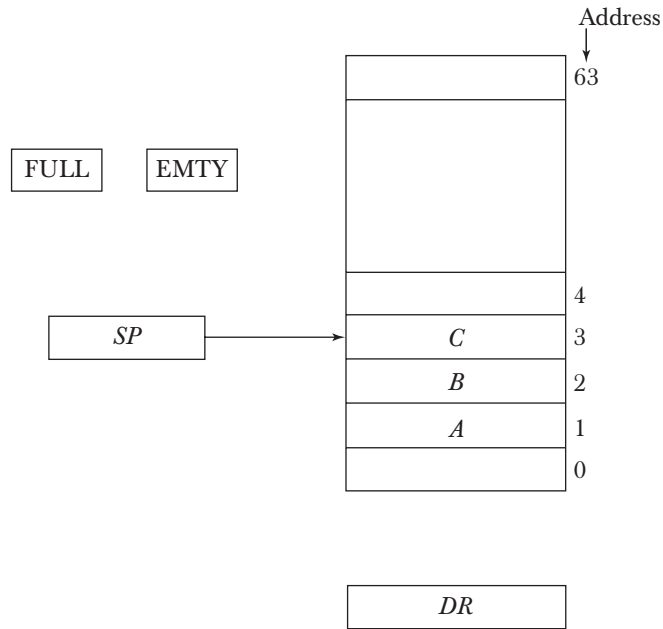
Figure 8-3   Block diagram of a 64-word stack.

since *SP* holds address 2. To insert a new item, the stack is pushed by incrementing *SP* and writing a word in the next-higher location in the stack. Note that item *C* has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since $111111 + 1 = 1000000$ in binary, but *SP* can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMTY* is set to 1 when the stack is empty of items. *DR* is the data register that holds the binary data to be written into or read out of the stack.

Initially, *SP* is cleared to 0, *EMTY* is set to 1, and *FULL* is cleared to 0, so that *SP* points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if *FULL* = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

push

$$SP \leftarrow SP + 1 \qquad \text{Increment stack pointer}$$
$$M[SP] \leftarrow DR \qquad \text{Write item on top of the stack}$$

$$\text{If } (SP = 0) \text{ then } (FULL \leftarrow 1) \qquad \text{Check if stack is full}$$

$$EMTY \leftarrow 0 \qquad\qquad\qquad\qquad \text{Mark the stack not empty}$$

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from *DR* into the top of the stack. Note that *SP* holds the address of the top of the stack and that $M[SP]$ denotes the memory word specified by the address presently available in *SP*. The first item stored in the stack is at address 1. The last item is stored at address 0. If *SP* reaches 0, the stack is full of items, so *FULL* is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing *SP*, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so *EMTY* is cleared to 0.

pop

A new item is deleted from the stack if the stack is not empty (if $EMTY = 0$). The pop operation consists of the following sequence of micro-operations:

$$DR \leftarrow M[SP] \qquad\qquad\qquad \text{Read item from the top of stack}$$

$$SP \leftarrow SP - 1 \qquad\qquad\qquad \text{Decrement stack pointer}$$

$$\text{If } (SP = 0) \text{ then } (EMTY \leftarrow 1) \quad \text{Check if stack is empty}$$

$$FULL \leftarrow 0 \qquad\qquad\qquad\qquad \text{Mark the stack not full}$$

The top item is read from the stack into *DR*. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so *EMTY* is set to l. This condition is reached if the item read was in location 1. Once this item is read out, *SP* is decremented and reaches the value 0, which is the initial value of *SP*. Note that if a pop operation reads the item from location 0 and then *SP* is decremented, *SP* changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when $FULL = 1$ or popped when $EMTY = 1$.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 8-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter *PC* points at the address of the next instruction in the program. The address register *AR* points at an array of

A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as microprogrammed control. A microprogrammed control unit is shown in Fig. 7-8. The binary control word for the CPU will come from the outputs of the control memory marked "micro-ops."

8-3   Stack Organization

LIFO

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (*SP*) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

stack pointer

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called *push* (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called *pop* (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack'. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack. The stack pointer register *SP* contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: *A*, *B*, and *C*, in that order. Item *C* is on top of the stack so that the content of *SP* is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of *SP*. Item *B* is now on top of the stack
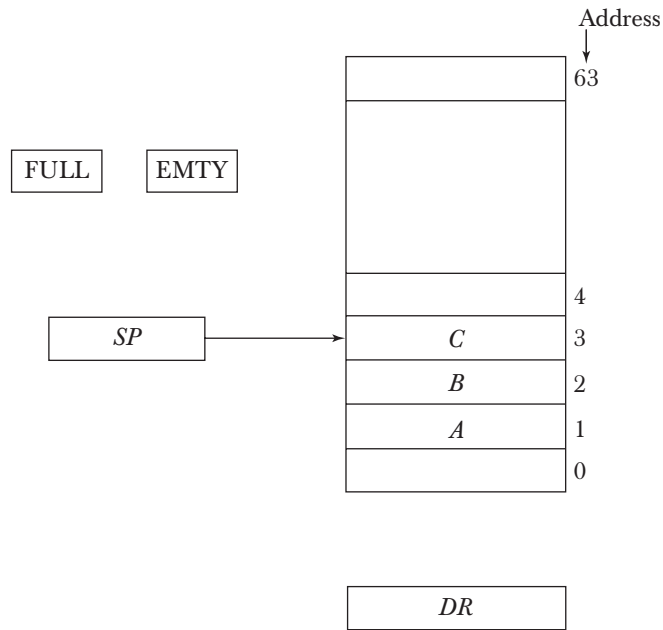
Address

Figure 8-3   Block diagram of a 64-word stack.

since *SP* holds address 2. To insert a new item, the stack is pushed by incrementing *SP* and writing a word in the next-higher location in the stack. Note that item *C* has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since $111111 + 1 = 1000000$ in binary, but *SP* can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMTY* is set to 1 when the stack is empty of items. *DR* is the data register that holds the binary data to be written into or read out of the stack.

Initially, *SP* is cleared to 0, *EMTY* is set to 1, and *FULL* is cleared to 0, so that *SP* points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if *FULL* = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

push

$$SP \leftarrow SP + 1 \qquad \text{Increment stack pointer}$$
$$M[SP] \leftarrow DR \qquad \text{Write item on top of the stack}$$

| If $(SP = 0)$ then $(FULL \leftarrow 1)$ | Check if stack is full |
| $EMTY \leftarrow 0$ | Mark the stack not empty |

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from *DR* into the top of the stack. Note that *SP* holds the address of the top of the stack and that $M[SP]$ denotes the memory word specified by the address presently available in *SP*. The first item stored in the stack is at address 1. The last item is stored at address 0. If *SP* reaches 0, the stack is full of items, so *FULL* is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing *SP*, the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so *EMTY* is cleared to 0.

pop

A new item is deleted from the stack if the stack is not empty (if $EMTY = 0$). The pop operation consists of the following sequence of micro-operations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

The top item is read from the stack into *DR*. The stack pointer is then decremented. If its value reaches zero, the stack is empty, so *EMTY* is set to l. This condition is reached if the item read was in location 1. Once this item is read out, *SP* is decremented and reaches the value 0, which is the initial value of *SP*. Note that if a pop operation reads the item from location 0 and then *SP* is decremented, *SP* changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when $FULL = 1$ or popped when $EMTY = 1$.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 8-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter *PC* points at the address of the next instruction in the program. The address register *AR* points at an array of
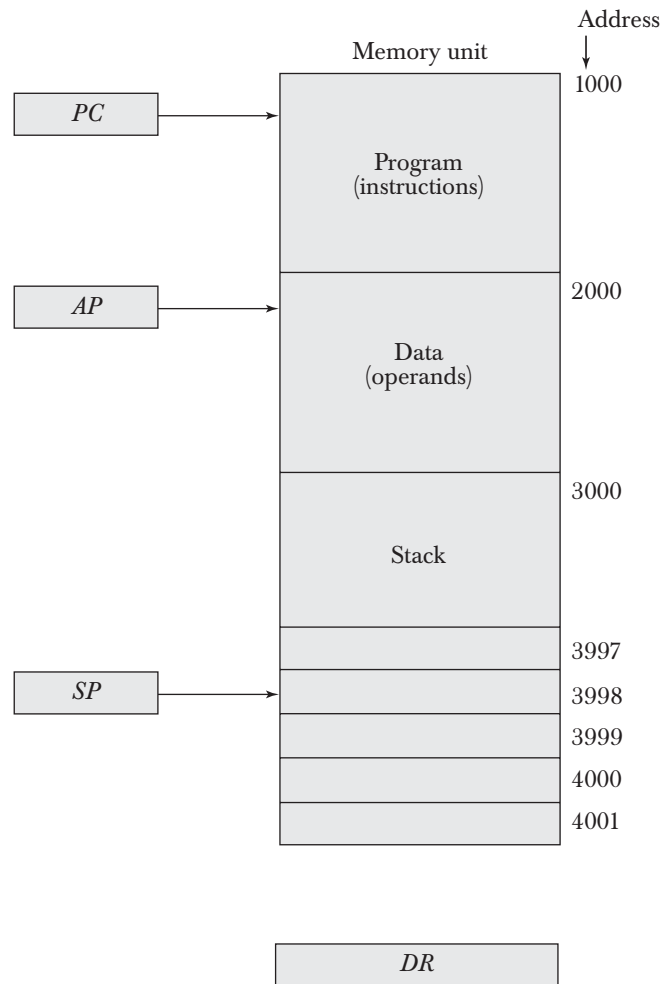
Figure 8-4   Computer memory with program, data, and stack segments.

data. The stack pointer *SP* points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. *PC* is used during the fetch phase to read an instruction. *AR* is used during the execute phase to read an operand. *SP* is used to push or pop items into or from the stack.

As shown in Fig. 8-4, the initial value of *SP* is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

We assume that the items in the stack communicate with a data register *DR*. A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from *DR* into the top of the stack. A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

The top item is read from the stack into *DR*. The stack pointer is then incremented to point at the next item in the stack.

stack limits   Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, *SP* is compared with the upper-limit register and after a pop operation, *SP* is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through *SP*, and (2) updating *SP*. Which of the two microoperations is done first and whether *SP* is updated by incrementing or decrementing depends on the organization of the stack. In Fig. 8-4 the stack grows by *decreasing* the memory address. The stack may be constructed to grow by *increasing* the memory address as in Fig. 8-3. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that *SP* points at the next *empty* location above the top of the stack. In this case the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer. The common arithmetic expressions

to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory. A program for a RISC-type CPU consists of LOAD and STORE instructions that have one memory and one register address, and computational-type instructions that have three addresses with all three specifying processor registers. The following is a program to evaluate $X = (A + B) * (C + D)$.

```
LOAD     R1, A          R1 ← M[A]
LOAD     R2, B          R2 ← M[B]
LOAD     R3, C          R3 ← M[C]
LOAD     R4, D          R4 ← M[D]
ADD      R1, R1, R2     R1 ← R1 + R2
ADD      R3, R3, R2     R3 ← R3 + R4
MUL      R1, R1, R3     R1 ← R1 * R3
STORE    X, R1          M[X] ← R1
```

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

8-5 Addressing Modes

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced assembly language programmer flexibility for writing programs that are more efficient with respect to the number of instructions and execution time.

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the

computer. The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

program counter (PC)

There is one register in the computer called the program counter or *PC* that keeps track of the instructions in the program stored in memory. *PC* holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

mode field

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions. Zero-address

Figure 8-6   Instruction format with mode field.

Opcode	Mode	Address

instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A *k*-bit field can specify any one of 2^k registers.

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address of the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

Autoincrement or Autodecrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction and the effective address used by the control when executing the *effective address* instruction. The *effective address* is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is

the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in Chap. 5. They are summarized here for reference.

Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address. The indirect address mode is also explained in Sec. 5-1 in conjunction with Fig. 5-2.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.

Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is $826 + 24 = 850$. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.

The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when, the index-mode instruction is used. In computers with many processor registers, any one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 8-7. The two-word instruction at address 200 and 201 is a "load to *AC*" instruction with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. *PC* has the value 200 for fetching this instruction. The content of processor register *R*1 is 400, and the content of an index register *XR* is 100. *AC* receives the operand after the instruction is executed. The figure lists a few pertinent addresses and shows the memory content at each of these addresses.

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

$PC = 200$

$R1 = 400$

$XR = 100$

AC

Figure 8-7 Numerical example for addressing modes.

The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC. In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is $500 + 202 = 702$ and the operand is 325. (Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is $XR + 500 = 100 + 500 = 600$ and the operand is 900. In the register mode the operand is in $R1$ and 400 is loaded into AC. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of $R1$ and the operand loaded into AC is 700. The autoincrement mode is the same as the register indirect mode except that $R1$ is incremented to 401 after the execution of the instruction. The autodecrement mode decrements $R1$ to 399 prior to the execution of the instruction. The operand loaded into AC is now 450. Table 8-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

TABLE 8-4 Tabular List of Numerical Example

Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	–	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

8-6 Data Transfer and Manipulation

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field is different in different computers, even for the same operation. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same *set of basic* instruction. Nevertheless, there is a set of basic operations that most, if not *operations* all, computers include in their instruction repertoire. The basic set of operations available in a typical computer is the subject covered in this and the next section.

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions are those that perform arithmetic, logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer. The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table 8-5 gives a list of eight data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

The *load* instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The *store* instruction designates a transfer from a processor register into memory. The *move* instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The *exchange* instruction swaps information between two registers or a register and a memory word. The *input* and *output* instructions transfer data among processor registers and input or output terminals. The *push* and *pop* instructions transfer data between processor registers and a memory stack.

It must be realized that the instructions listed in Table 8-5, as well as in subsequent tables in this section, are often associated with a variety of addressing modes. Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes. For example, the mnemonic for *load immediate* becomes LDI. Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the *load to accumulator* instruction when used with eight

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

TABLE 8-6 Eight Addressing Modes for the Load Instruction

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1) +	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

different addressing modes. Table 8-6 shows the recommended assembly language convention and the actual transfer accomplished in each case. *ADR* stands for an address, *NBR* is a number or operand, *X* is an index register, *R*1 is a processor register, and *AC* is the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the program counter *PC*. The # character precedes the operand in an immediate-mode instruction. An indexed mode instruction is recognized by a register that is placed in parentheses after the symbolic address. The register mode is symbolized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses. The autoincrement mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The autodecrement mode would use a minus instead. To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the particular computer.

Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

A list of data manipulation instructions will look very much like the list of micro-operations given in Chap. 4. It must be realized, however, that each instruction when executed in the computer must go through the fetch phase to read its binary

8.8 (only characteristics, i.e., pg. 282 – 284) ---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOI

the program while external interrupts are asynchronous. If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

software interrupt

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

8-8 Reduced Instruction Set Computer (RISC)

An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set chosen for a particular computer determines the way that machine language programs are constructed. Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them. As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes. The trend into computer hardware complexity was influenced by various factors, such as upgrading existing models to provide more customer applications, adding instructions that facilitate the translation from high-level language into machine language programs, and striving to develop machines that move functions from software implementation into hardware implementation. A computer with a large number of

CISC

instructions is classified as a *complex instruction set computer,* abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often.

RISC

This type of computer is classified as a *reduced instruction set computer* or RISC.

In this section we introduce the major characteristics of CISC and RISC architectures and then present the instruction set and instruction format of a RISC processor.

CISC Characteristics

The design of an instruction set for a computer must take into consideration not only machine language constructs, but also the requirements imposed on the use of high-level programming languages. The translation from high-level to machine language programs is done by means of a compiler program. One reason for the trend to provide a complex instruction set is the desire to simplify the compilation and improve the overall computer performance. The task of a compiler is to generate a sequence of machine instructions for each high-level language statement. The task is simplified if there are machine instructions that implement the statements directly. The essential goal of a CISC architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

Another characteristic of CISC architecture is the incorporation of variable-length instruction formats. Instructions that require register operands may be only two bytes in length, but instructions that need two memory addresses may need five bytes to include the entire instruction code. If the computer has 32-bit words (four bytes), the first instruction occupies half a word, while the second instruction needs one word in addition to one byte in the next word. Packing variable instruction formats in a fixed-length memory word requires special decoding circuits that count bytes within words and frame the instructions according to their byte length.

The instructions in a typical CISC processor provide direct manipulation of operands residing in memory. For example, an ADD instruction may specify one operand in memory through index addressing and a second operand in memory through a direct addressing. Another memory location may be included in the instruction to store the sum. This requires three memory references during execution of the instruction. Although CISC processors have instructions that use only processor registers, the availability of other modes of operations tend to simplify high-level language compilation. However, as more instructions and addressing modes are incorporated into a computer, the more hardware logic is needed to implement and support them, and this may cause the computations to slow down. In summary, the major characteristics of CISC architecture are:

1. A large number of instructions–typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently

3. A large variety of addressing modes–typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed control

The small set of instructions of a typical RISC processor consists mostly of register-to-register operations, with only simple load and store operations for memory access. Thus each operand is brought into a processor register with a load instruction. All computations are done among the data stored in processor registers. Results are transferred to memory by means of store instructions. This architectural feature simplifies the instruction set and encourages the optimization of register manipulation. The use of only a few addressing modes results from the fact that almost all instructions have simple register addressing. Other addressing modes may be included, such as immediate operands and relative mode.

By using a relatively simple instruction format, the instruction length can be fixed and aligned on word boundaries. An important aspect of RISC instruction format is that it is easy to decode. Thus the operation code and register fields of the instruction code can be accessed simultaneously by the control. By simplifying the instructions and their format, it is possible to simplify the control logic. For faster operations, a hardwired control is preferable over a microprogrammed control. An example of hardwired control is presented in Chap. 5 in conjunction with the control unit of the basic computer. Examples of microprogrammed control are presented in Chap. 7.

A characteristic of RISC processors is their ability to execute one instruction per clock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipelining. A load or store instruction may require two clock cycles because access to

memory takes more time than register operations. Efficient pipelining, as well as a few other characteristics, are sometimes attributed to RISC, although they may exist in non-RISC architectures as well. Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language programs

A large number of registers is useful for storing intermediate results and for optimizing operand references. The advantage of register storage as opposed to memory storage is that registers can transfer information to other registers much faster than the transfer of information to and from memory. Thus register-to-memory operations can be minimized by keeping the most frequent accessed operands in registers. Studies that show improved performance for RISC architecture do not differentiate between the effects of the reduced instruction set and the effects of a large register file. For this reason a large number of registers in the processing unit are sometimes associated with RISC processors. The use of overlapped register windows when transferring program control after a procedure call is explained below. Instruction pipeline in RISC is presented in Sec. 9-5 after we explain the concept of pipelining.

pipelining

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time-consuming operations. Some computers provide multiple-register banks, and each procedure is allocated its own bank of registers. This eliminates the need for saving and restoring register values. Some computers use the memory stack to store the parameters that are needed by the procedure, but this requires a memory access every time the stack is accessed.

A characteristic of some RISC processors is their use of *overlapped register windows* to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a

new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

The concept of overlapped register windows is illustrated in Fig. 8-9. The system has a total of 74 registers. Registers $R0$ through $R9$ are global registers that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results between adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

As an example, suppose that procedure A calls procedure B. Registers $R26$ through $R31$ are common to both procedures, and therefore procedure A stores the parameters for procedure B in these registers. Procedure B uses local registers $R32$ through $R41$ for local variable storage. If procedure B calls procedure C, it will pass the parameters through registers $R42$ through $R47$. When procedure B is ready to return at the end of its computation, the program stores results of the computation in registers $R26$ through $R31$ and transfers back to the register window of procedure A. Note that registers $R10$ through $R15$ are common to procedures A and D because the four windows have a circular organization with A being adjacent to D.
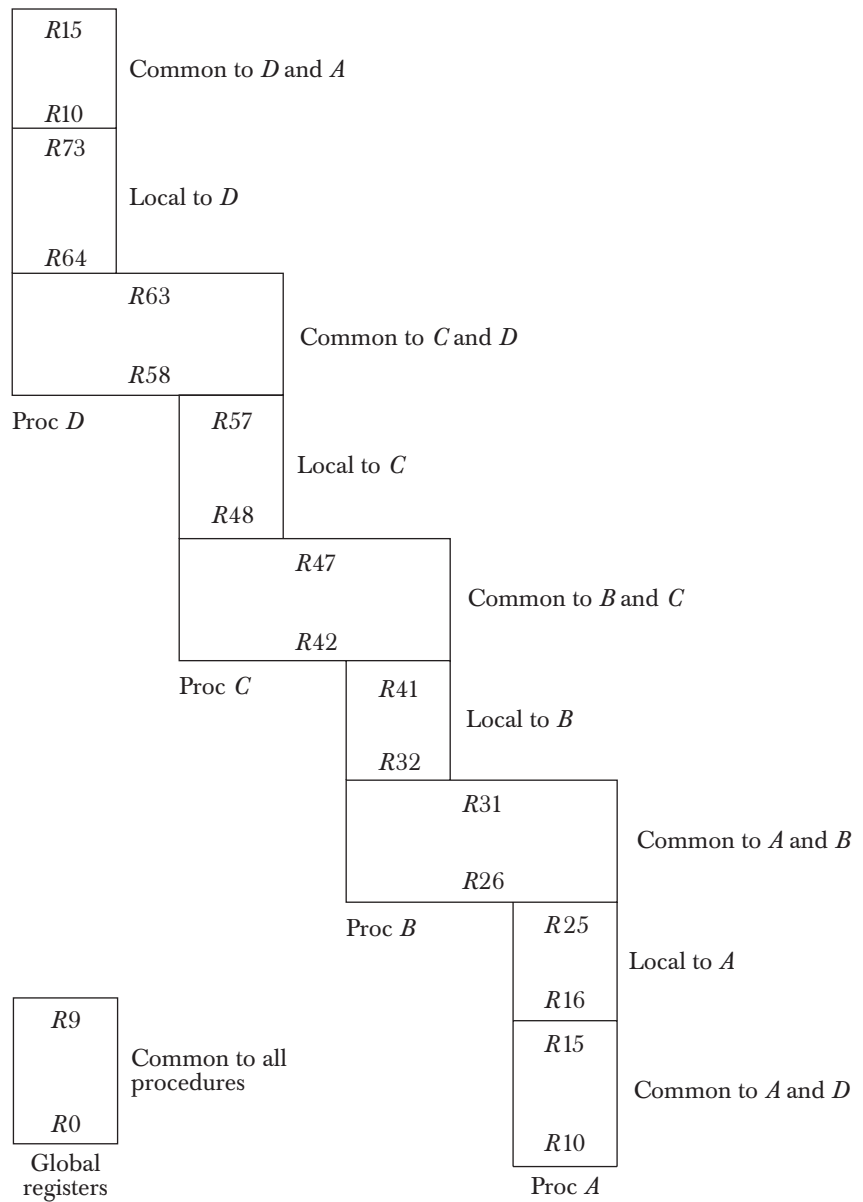
As mentioned previously, the 10 global registers $R0$ through $R9$ are available to all procedures. Each procedure in Fig. 8-9 has available a total of 32 registers while it is active. This includes 10 global registers, 10 local registers, six low overlapping registers, and six high overlapping registers. Other fixed-size register window schemes are possible, and each may differ in the size of the register window and the size of the total register file. In general, the organization of register windows will have the following relationships:

number of global registers $= G$

number of local registers in each window $= L$

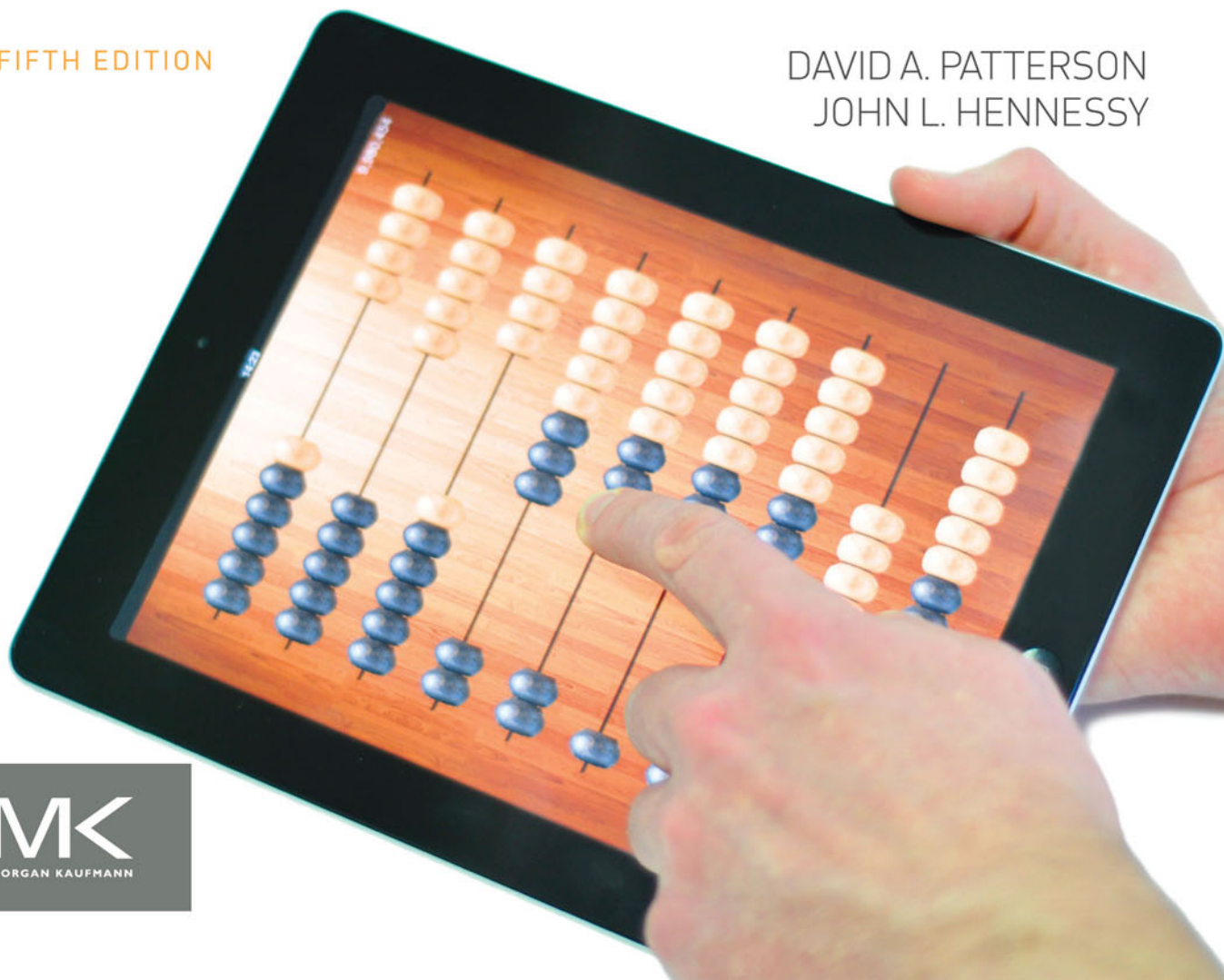number of registers common to two windows $= C$

number of windows $= W$

Figure 8-9   Overlapped register windows.

7.1 (page 632-Introduction of Multicore Processor)---:) https://github.com/AvinashShrivastav/MATHEMATICS

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

FIFTH EDITION

DAVID A. PATTERSON
JOHN L. HENNESSY

In Praise of *Computer Organization and Design: The Hardware/ Software Interface,* Fifth Edition

"Textbook selection is often a frustrating act of compromise—pedagogy, content coverage, quality of exposition, level of rigor, cost. *Computer Organization and Design* is the rare book that hits all the right notes across the board, without compromise. It is not only the premier computer organization textbook, it is a shining example of what all computer science textbooks could and should be."

—Michael Goldweber, *Xavier University*

"I have been using *Computer Organization and Design* for years, from the very first edition. The new Fifth Edition is yet another outstanding improvement on an already classic text. The evolution from desktop computing to mobile computing to Big Data brings new coverage of embedded processors such as the ARM, new material on how software and hardware interact to increase performance, and cloud computing. All this without sacrificing the fundamentals."

—Ed Harcourt, *St. Lawrence University*

"To Millennials: *Computer Organization and Design* is *the* computer architecture book you should keep on your (virtual) bookshelf. The book is both old and new, because it develops venerable principles—Moore's Law, abstraction, common case fast, redundancy, memory hierarchies, parallelism, and pipelining—but illustrates them with contemporary designs, e.g., ARM Cortex A8 and Intel Core i7."

—Mark D. Hill, *University of Wisconsin-Madison*

"The new edition of *Computer Organization and Design* keeps pace with advances in emerging embedded and many-core (GPU) systems, where tablets and smartphones will are quickly becoming our new desktops. This text acknowledges these changes, but continues to provide a rich foundation of the fundamentals in computer organization and design which will be needed for the designers of hardware and software that power this new class of devices and systems."

—Dave Kaeli, *Northeastern University*

"The Fifth Edition of *Computer Organization and Design* provides more than an introduction to computer architecture. It prepares the reader for the changes necessary to meet the ever-increasing performance needs of mobile systems and big data processing at a time that difficulties in semiconductor scaling are making all systems power constrained. In this new era for computing, hardware and software must be co-designed and system-level architecture is as critical as component-level optimizations."

—Christos Kozyrakis, *Stanford University*

"Patterson and Hennessy brilliantly address the issues in ever-changing computer hardware architectures, emphasizing on interactions among hardware and software components at various abstraction levels. By interspersing I/O and parallelism concepts with a variety of mechanisms in hardware and software throughout the book, the new edition achieves an excellent holistic presentation of computer architecture for the PostPC era. This book is an essential guide to hardware and software professionals facing energy efficiency and parallelization challenges in Tablet PC to cloud computing."
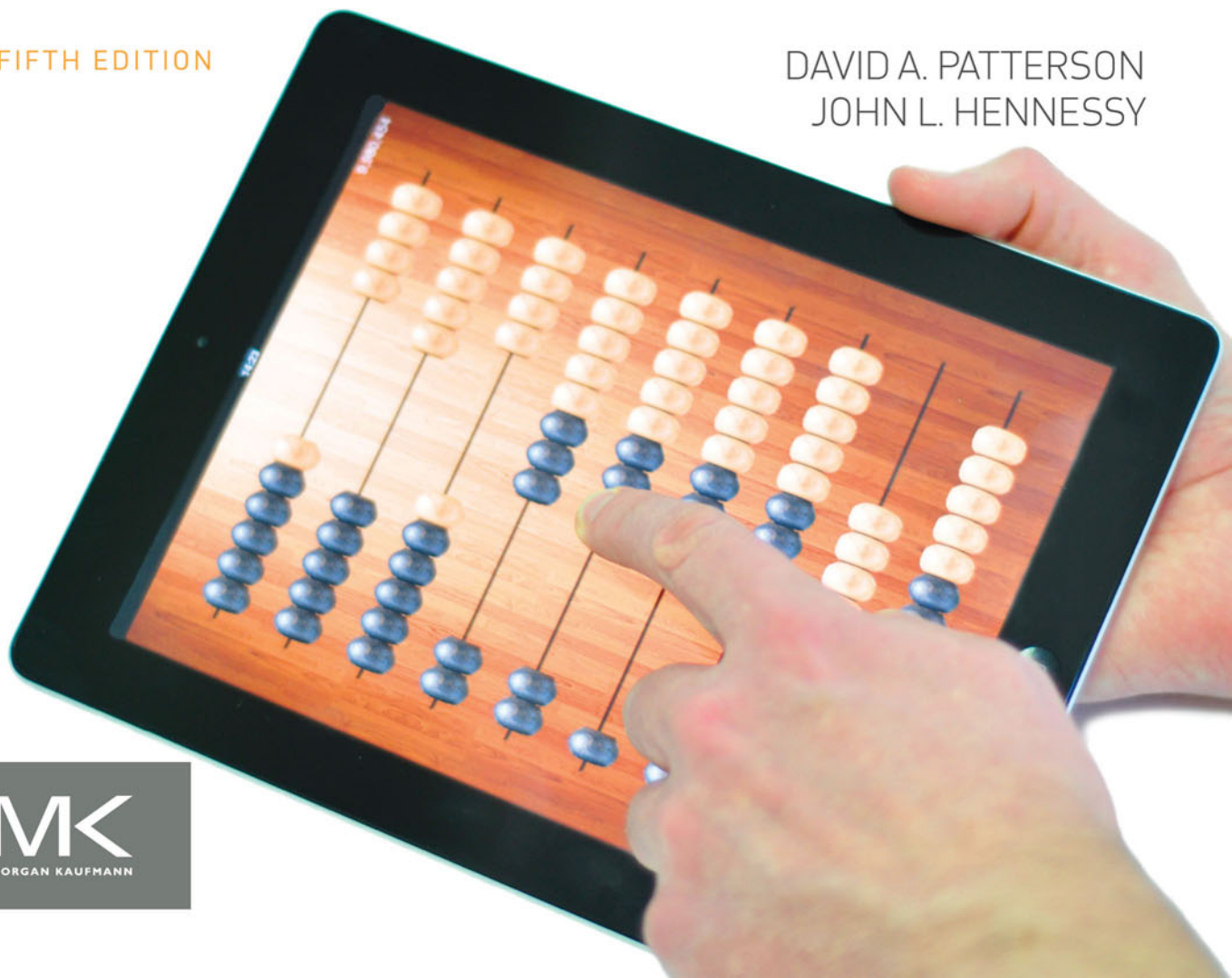
—Jae C. Oh, *Syracuse University*

7.7 (page 654-656) Characteristics of GPU Vs. CPU---:) https://github.com/AvinashShrivastav/MATHEMATIO

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

FIFTH EDITION

DAVID A. PATTERSON
JOHN L. HENNESSY

MK

MORGAN KAUFMANN

In Praise of *Computer Organization and Design: The Hardware/Software Interface,* **Fifth Edition**

"Textbook selection is often a frustrating act of compromise—pedagogy, content coverage, quality of exposition, level of rigor, cost. *Computer Organization and Design* is the rare book that hits all the right notes across the board, without compromise. It is not only the premier computer organization textbook, it is a shining example of what all computer science textbooks could and should be."

—Michael Goldweber, *Xavier University*

"I have been using *Computer Organization and Design* for years, from the very first edition. The new Fifth Edition is yet another outstanding improvement on an already classic text. The evolution from desktop computing to mobile computing to Big Data brings new coverage of embedded processors such as the ARM, new material on how software and hardware interact to increase performance, and cloud computing. All this without sacrificing the fundamentals."

—Ed Harcourt, *St. Lawrence University*

"To Millennials: *Computer Organization and Design* is *the* computer architecture book you should keep on your (virtual) bookshelf. The book is both old and new, because it develops venerable principles—Moore's Law, abstraction, common case fast, redundancy, memory hierarchies, parallelism, and pipelining—but illustrates them with contemporary designs, e.g., ARM Cortex A8 and Intel Core i7."

—Mark D. Hill, *University of Wisconsin-Madison*

"The new edition of *Computer Organization and Design* keeps pace with advances in emerging embedded and many-core (GPU) systems, where tablets and smartphones will are quickly becoming our new desktops. This text acknowledges these changes, but continues to provide a rich foundation of the fundamentals in computer organization and design which will be needed for the designers of hardware and software that power this new class of devices and systems."

—Dave Kaeli, *Northeastern University*

"The Fifth Edition of *Computer Organization and Design* provides more than an introduction to computer architecture. It prepares the reader for the changes necessary to meet the ever-increasing performance needs of mobile systems and big data processing at a time that difficulties in semiconductor scaling are making all systems power constrained. In this new era for computing, hardware and software must be co-designed and system-level architecture is as critical as component-level optimizations."

—Christos Kozyrakis, *Stanford University*

"Patterson and Hennessy brilliantly address the issues in ever-changing computer hardware architectures, emphasizing on interactions among hardware and software components at various abstraction levels. By interspersing I/O and parallelism concepts with a variety of mechanisms in hardware and software throughout the book, the new edition achieves an excellent holistic presentation of computer architecture for the PostPC era. This book is an essential guide to hardware and software professionals facing energy efficiency and parallelization challenges in Tablet PC to cloud computing."

—Jae C. Oh, *Syracuse University*

record separator (RS) and file separator (FS). The communication control characters are useful during the transmission of text between remote terminals. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message when transmitted through a communication medium.

byte

    ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font. When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded character.

11-2   Input–Output Interface

Input–output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

3. Data codes and formats in peripherals differ from the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.
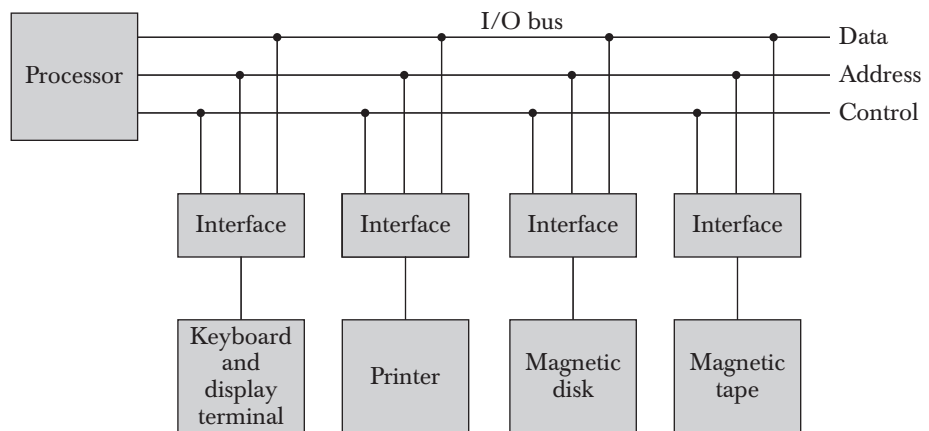
interface

    To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called *interface* units because they interface between the processor bus and the peripheral device. The word "Interface" is a general term for the point of contact between two parts of a system. In digital computer system the interface is refered as a complementary set of signal connection points between two parts of a system. Therefore, "to interface" means to attach two or more components or systems,

via their respective interface points for data exchanges between them. Two main types of interface are CPU interface that corresponds to the system bus and input–output interface that depends on the nature of input–output device. To attach an input–output device to CPU and input–output interface, circuit is placed between the device and the system bus. This circuit is meant for matching the signal formats and timing characteristics of the CPU interface to those of the input–output device interface. The main function of input–output interface circuit are data conversion, synchronization and device selection. Data conversion refers to conversion between digital and analog signals, and conversion between serial and parallel data formats. Synchronation refers to matching of operating speeds of CPU and other peripherals. Device selection refers to the selection of I/O device by CPU in a queue manner. In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 11-1. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

Figure 11-1   Connection of I/O bus to input–output devices.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

I/O command

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

control command

A *control command* is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

status

A *status command* is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

output data

A *data output command* causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

input data

The *data input command* is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data,

address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

   1. Use two separate buses, one for memory and the other for I/O.
   2. Use one common bus for both memory and I/O but have separate control lines for each.
   3. Use one common bus for memory and I/O with common control lines.

IOP

   In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel. In Sec. 11-7 we discuss the function of the IOP in more detail.

Isolated versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read.or write lines. The *I/O read* and *I/O write* control lines are enabled during an I/O transfer. The *memory read* and *memory write* control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the *isolated I/O method* for assigning addresses in a common bus.

isolated I/O

   In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

   The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same

memory-mapped

address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as *memory-mapped I/O.* The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduces the memory address range available.

In a memory-mapped I/O organization there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.

Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input–output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O all instructions that refer to memory are also available for I/O.

I/O port

Example of I/O Interface

An example of an I/O interface unit is shown in block diagram form in Fig. 11-2. It consists of two data registers called *ports,* a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The I/O data to and from the device can be transferred into either port *A* or port *B.* The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports *A* and *B* registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular interface register with which the CPU communicates.

11.4---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

directs the movement of data through the registers. Whenever the F_i bit of the control register is set ($F_i = 1$) and the F_{i+1} bit is reset ($F'_{i+1} = 1$), a clock is generated causing register $R(I+1)$ to accept the data from register RI. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. This ripple-through operation stops when the data reach a register RI with the next flip-flop F_{i+1} being set to 1, or at the last register $R4$. An overall master clear is used to initialize all control register flip-flops to 0.

Data are inserted into the buffer provided that the *input ready* signal is enabled. This occurs when the first control flip-flop F_1 is reset, indicating that register $R1$ is empty. Data are loaded from the input lines by enabling the clock in $R1$ through the *insert* control line. The same clock sets F_1, which disables the *input ready* control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that $R2$ is empty. The data in $R1$ are transferred into $R2$ and F_1 is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word. If the FIFO is full, F_1 remains set and the *input ready* line stays in the 0 state. Note that the two control lines *input ready* and *insert* constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The *output ready* control line is enabled when the last control flip-flop F_4 is set, indicating that there are valid data in the output register $R4$. The output data from $R4$ are accepted by a destination unit, which then enables the *delete* control signal. This resets F_4, causing *output ready* to disable, indicating that the data on the output are no longer valid. Only after the *delete* signal goes back to 0 can the data from $R3$ move into $R4$. If the FIFO is empty, there will be no data in $R3$ and F_4 will remain in the reset state. Note that the two control lines *output ready* and *delete* constitute a source-initiated pair of handshake lines.

11-4 Modes of Transfer

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

programmed I/O

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

interrupt

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

DMA

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O-memory transfers are infrequent compared to processor access to memory. DMA transfer is discussed in more detail in Sec. 11-6.

IOP

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in Sec. 11-7.

Example of Programmed I/O

In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 11-10. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig. 11-5.

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. 11-11. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. A program that

Figure 11-10   Data transfer from I/O device to CPU.
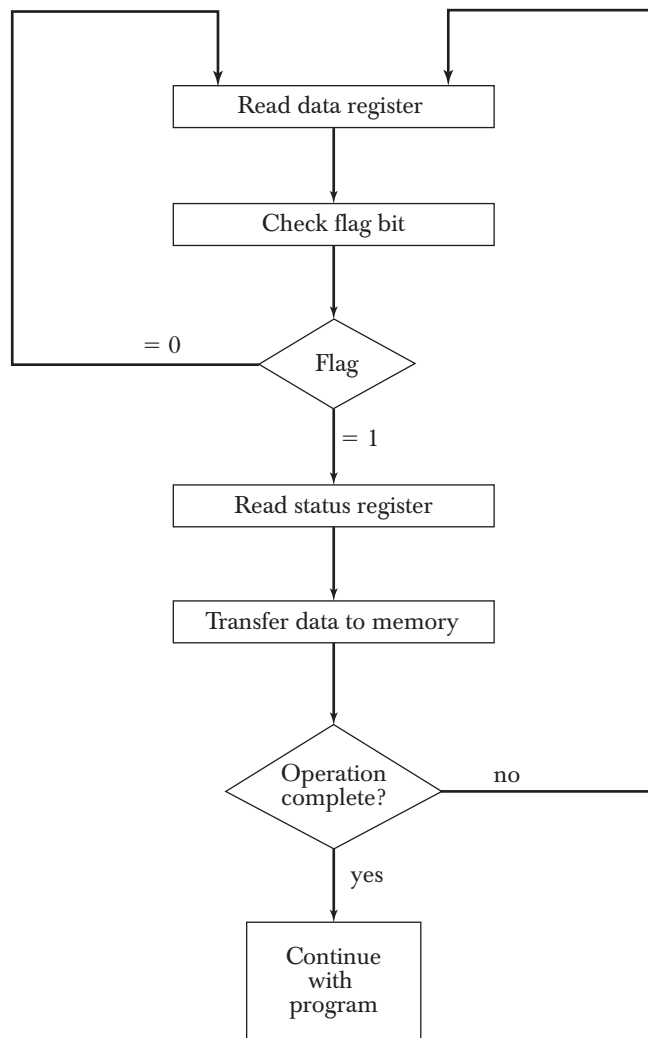

F = Flag bit
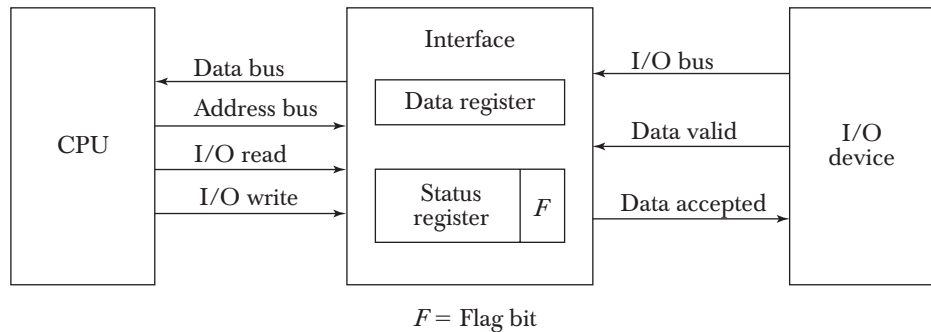
Figure 11-11   Flowchart for CPU program to input data.

stores input characters in a memory buffer using the instructions defined in Chap. 6 is listed in Table 6-21.

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1 μs. Assume that the input device transfers

its data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 μs. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

Interrupt-Initiated I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called *vectored interrupt* and the other, *nonvectored interrupt*. In a nonvectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the *interrupt vector*. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored. A system with vectored interrupt is demonstrated in Sec. 11-5.

vectored interrupt

Software Considerations

The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers. In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.

I/O routines

Software control of input–output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

11-5 Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer. As discussed in Sec. 8-7, some processors also push the current PSW (program status word) onto the stack and load a new PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of I/O interrupts.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

priority interrupt   A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

polling   Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine for all interrupts consists

of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer. The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy-chaining method.

Daisy-Chaining Priority

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Fig. 11-12. The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts arc pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative-logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own *vector address (VAD)*   interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with $PI = 1$ and $PO = 0$
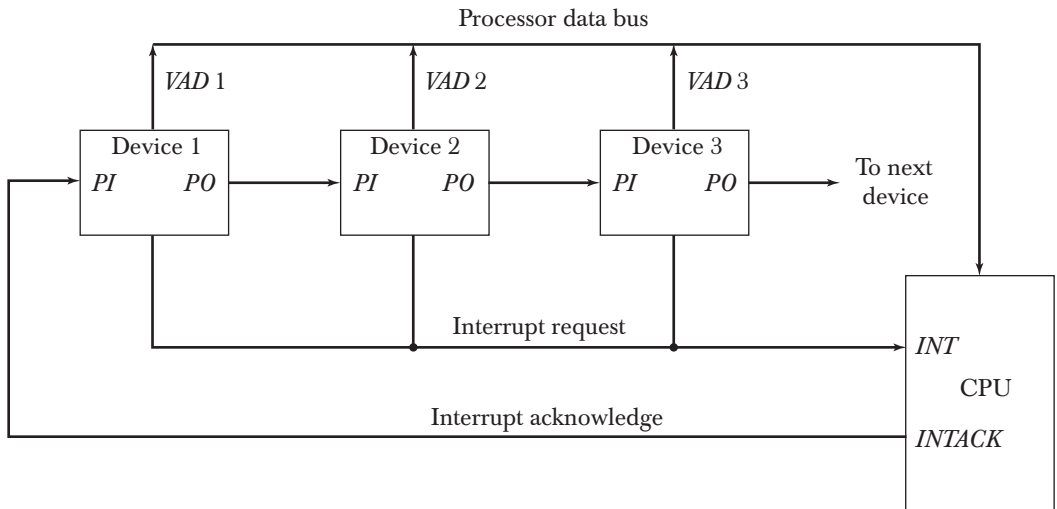
Processor data bus

Figure 11-12 Daisy-chain priority interrupt.

is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 11-13 shows the internal logic that must be included within each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If $PI = 0$, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If $PI = 1$ and $RF = 0$, then $PO = 1$ and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when $PI = 1$ and $RF = 1$. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to

the programmer can use any bit configuration for the mask register. The interrupt status bit must be cleared so it can be set again when a higher-priority interrupt occurs. The contents of processor registers are saved because they may be needed by the program that has been interrupted after control returns to it. The interrupt enable *IEN* is then set to allow other (higher-priority) interrupts and the computer proceeds to service the interrupt request.

The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear interrupt enable bit *IEN*.
2. Restore contents of processor registers.
3. Clear the bit in the interrupt register belonging to the source that has been serviced.
4. Set lower-level priority bits in the mask register.
5. Restore return address into *PC* and set *IEN*.

The bit in the interrupt register belonging to the source of the interrupt must be cleared so that it will be available again for the source to interrupt. The lower-priority bits in the mask register (including the bit of the source being interrupted) are set so they can enable the interrupt. The return to the interrupted program is accomplished by restoring the return address to *PC*. Note that the hardware must be designed so that no interrupts occur while executing steps 2 through 5; otherwise, the return address may be lost and the information in the mask and processor registers may be ambiguous if an interrupt is acknowledged while executing the operations in these steps. For this reason *IEN* is initially cleared and then set after the return address is transferred into *PC*.

The initial and final operations listed above are referred to as *overhead* operations or *housekeeping* chores. They are not part of the service program proper but are essential for processing interrupts. All overhead operations can be implemented by software. This is done by inserting the proper instructions at the beginning and at the end of each service routine. Some of the overhead operations can be done automatically by the hardware. The contents of processor registers can be pushed into a stack by the hardware before branching to the service routine. Other initial and final operations can be assigned to the hardware. In this way, it is possible to reduce the time between receipt of an interrupt and the execution of the instructions that service the interrupt source.

11-6 Direct Memory Access (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly

bus request

bus grant

burst transfer

cycle stealing

would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 11-16 shows two control signals in the CPU that facilitate the DMA transfer. The *bus request* (*BR*) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance (see Sec. 4-3). The CPU activates the *bus grant* (*BG*) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA *burst transfer,* a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called *cycle stealing* allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines

Figure 11-16   CPU bus signals for DMA transfer.
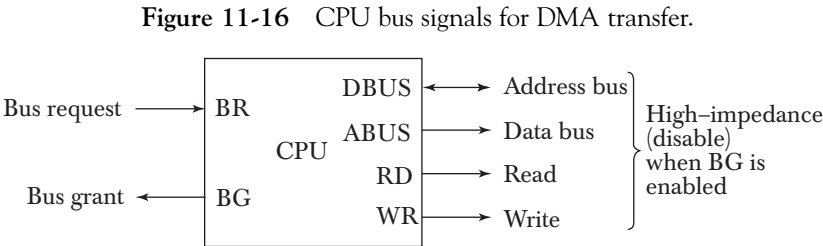
are used for direct communication with the memory The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 11-17 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When $BG = 1$, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

Figure 11-17   Block diagram of DMA controller.
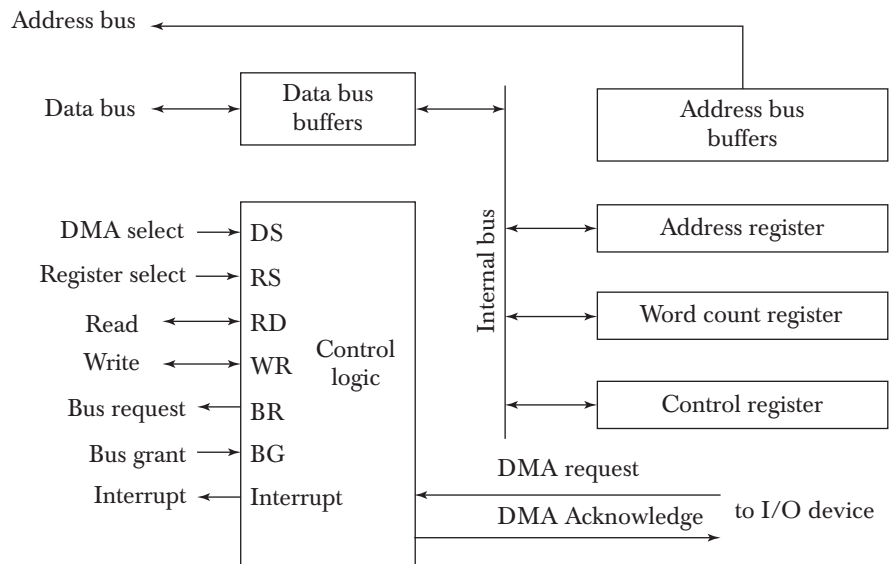
The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Fig. 11-18. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the *DS* and *RS* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device. Note that the *RD* and *WR* lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the *BG* line. When *BG* = 0, the *RD* and *WR* are input lines allowing the CPU to communicate with the internal DMA registers. When *BG* = 1, the *RD* and *WR* are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

12.1 (up to pg-446)---:) https://github.com/AvinashShrivastav/MATHEMATICS-FOR-COMPUTING

<div style="border:1px solid #000; padding:1em;">

CHAPTER TWELVE

Memory Organization

</div>

IN THIS CHAPTER

12-1   Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the *main memory*. Devices that provide backup storage are called *auxiliary memory*. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other
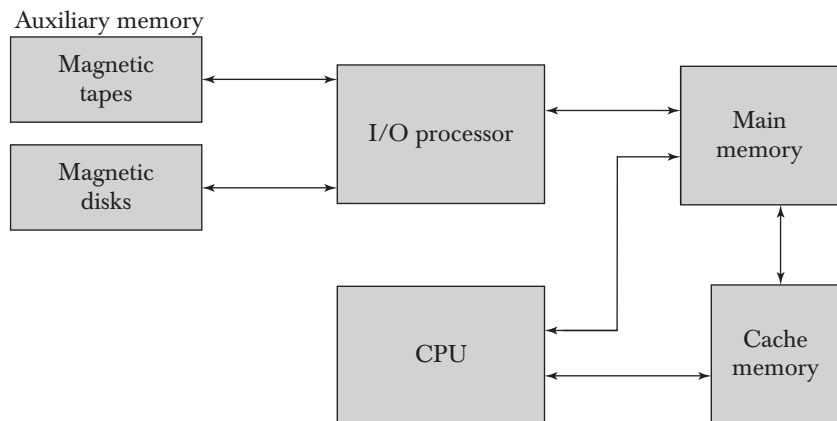
auxiliary memory

informatior is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. Figure 12-1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

cache memory   A special very-high-speed memory called a *cache* is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations. By

Figure 12-1   Memory hierarchy in a computer system.

making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100 ns, while main memory access time may be 700 ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

multiprogramming   Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called *multiprogramming,* refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

With multiprogramming the need arises for running partial programs, for varying the amount of main memory in use by a given program, and for moving programs around the memory hierarchy. Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be

executed, it is transferred to main memory to be executed by the CPU. Thus one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the *memory management system*. The hardware for a memory management system is presented in Sec. 12-7.

12-2 Main Memory

Random-access memory (RAM)

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, *static* and *dynamic*. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles. One of the major applications of the static RAM is in implementing the cache memories. The dynamic RAMs are used for implementing the main memory. Most of the desktop personnel computer systems are dynamic RAMs with improved performance characteristics such as multibank DRAM, extended dataout DRAM, synchronous DRAM, and Direct RAM bus DRAM.

read-only memory (ROM)

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

bootstrap loader

Among other things, the ROM portion of main memory is needed for storing an initial program called a *bootstrap loader*. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is

computer startup

turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024×8 memory constructed with 128×8 RAM chips and 512×8 ROM chips.

RAM and ROM Chips

bidirectional bus

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation, or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

Figure 12-2   Typical RAM chip.
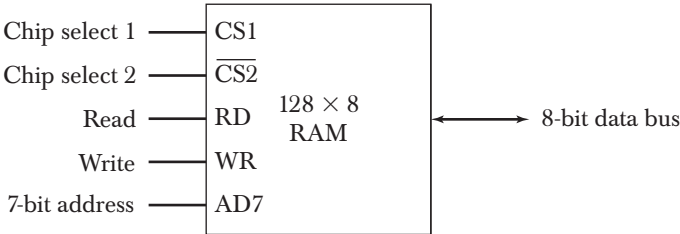
(a) Block diagram

CS1	$\overline{\text{CS2}}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

The block diagram of a RAM chip is shown in Fig. 12-2. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations of read or write.

The function table listed in Fig. 12-2(b) specifies the operation of the RAM chip. The unit is in operation only when $CS1 = 1$ and $\overline{CS2} = 0$. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When $CS1 = 1$ and $\overline{CS2} = 0$, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 12-3. For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be $CS1 = 1$ and $\overline{CS2} = 0$ for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a *memory address map,* is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM