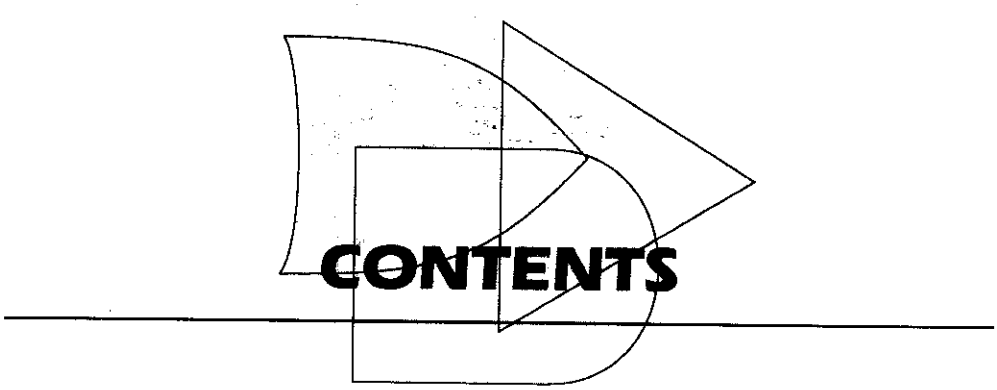
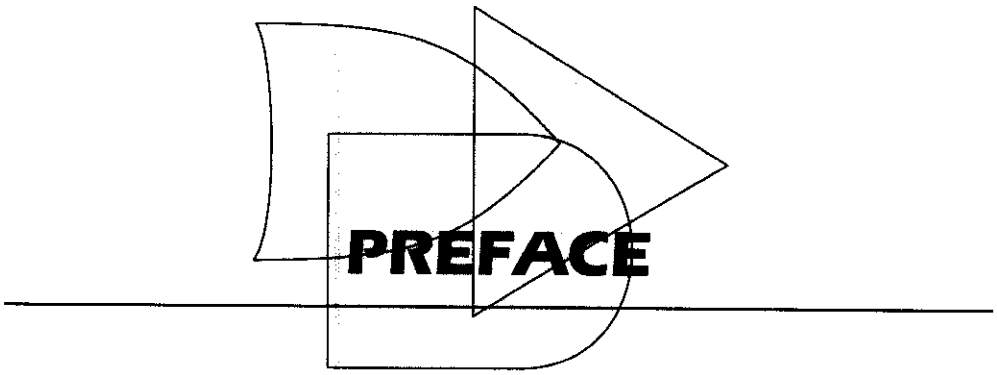


CONTENTS



PREFACE

Digital Design is concerned with the design of digital electronic circuits. The subject is also known by other names such as logic design, digital logic, switching circuits, and digital systems. Digital circuits are employed in the design of systems such as digital computers, control systems, data communications, and many other applications that require electronic digital hardware. This book presents the basic tools for the design of digital circuits and provides methods and procedures suitable for a variety of digital design applications.

Many features of the second edition remain the same as those of the first edition. The material is still organized in the same manner. The first five chapters cover combinational circuits. The next three chapters deal with synchronous clocked sequential circuits. Asynchronous sequential circuits are introduced next. The last three chapters deal with various aspects of commercially available integrated circuits.

The second edition, however, offers several improvements over the first edition. Many sections have been rewritten to clarify the presentation. Chapters 1 through 7 and Chapter 10 have been revised by adding new up-to-date material and deleting obsolete subjects. New problems have been formulated for the first seven chapters. These replace the problem set from the first edition. Three new experiments have been added in Chapter 11. Chapter 12, a new chapter, presents the IEEE standard graphic symbols for logic elements.

The following is a brief description of the subjects that are covered in each chapter with an emphasis on the revisions that were made in the second edition.

Chapter 1 presents the various binary systems suitable for representing information in digital systems. The binary number system is explained and binary codes are illustrated. A new section has been added on signed binary numbers.

Chapter 2 introduces the basic postulates of Boolean algebra and shows the correlation between Boolean expressions and their corresponding logic diagrams. All possible logic operations for two variables are investigated and from that, the most useful logic gates used in the design of digital systems are determined. The characteristics of integrated circuit gates are mentioned in this chapter but a more detailed analysis of the electronic circuits of the gates is done in Chapter 11.

Chapter 3 covers the map and tabulation methods for simplifying Boolean expressions. The map method is also used to simplify digital circuits constructed with AND-OR, NAND, or NOR gates. All other possible two-level gate circuits are considered and their method of implementation is summarized in tabular form for easy reference.

Chapter 4 outlines the formal procedures for the analysis and design of combinational circuits. Some basic components used in the design of digital systems, such as adders and code converters, are introduced as design examples. The sections on multilevel NAND and NOR implementation have been revised to show a simpler procedure for converting AND-OR diagrams to NAND or NOR diagrams.

Chapter 5 presents various medium scale integration (MSI) circuits and programmable logic device (PLD) components. Frequently used digital logic functions such as parallel adders and subtractors, decoders, encoders, and multiplexers, are explained, and their use in the design of combinational circuits is illustrated with examples. In addition to the programmable read only memory (PROM) and programmable logic array (PLA) the book now shows the internal construction of the programmable array logic (PAL). These three PLD components are extensively used in the design and implementation of complex digital circuits.

Chapter 6 outlines the formal procedures for the analysis and design of clocked synchronous sequential circuits. The gate structure of several types of flip-flops is presented together with a discussion on the difference between pulse level and pulse transition triggering. Specific examples are used to show the derivation of the state table and state diagram when analyzing a sequential circuit. A number of design examples are presented with added emphasis on sequential circuits that use D-type flip-flops.

Chapter 7 presents various sequential digital components such as registers, shift registers, and counters. These digital components are the basic building blocks from which more complex digital systems are constructed. The sections on the random access memory (RAM) have been completely revised and a new section deals with the Hamming error correcting code.

Chapter 8 presents the algorithmic state machine (ASM) method of digital design. The ASM chart is a special flow chart suitable for describing both sequential and parallel operations with digital hardware. A number of design examples demonstrate the use of the ASM chart in the design of state machines.

Chapter 9 presents formal procedures for the analysis and design of asynchronous sequential circuits. Methods are outlined to show how an asynchronous sequential cir-

cuit can be implemented as a combinational circuit with feedback. An alternate implementation is also described that uses SR latches as the storage elements in an asynchronous sequential circuit.

Chapter 10 presents the most common integrated circuit digital logic families. The electronic circuits of the common gate in each family is analyzed using electrical circuit theory. A basic knowledge of electronic circuits is necessary to fully understand the material in this chapter. Two new sections are included in the second edition. One section shows how to evaluate the numerical values of four electrical characteristics of a gate. The other section introduces the CMOS transmission gate and gives a few examples of its usefulness in the construction of digital circuits.

Chapter 11 outlines 18 experiments that can be performed in the laboratory with hardware that is readily and inexpensively available commercially. These experiments use standard integrated circuits of the TTL type. The operation of the integrated circuits is explained by referring to diagrams in previous chapters where similar components are originally introduced. Each experiment is presented informally rather than in a step-by-step fashion so that the student is expected to produce the details of the circuit diagram and formulate a procedure for checking the operation of the circuit in the laboratory.

Chapter 12 presents the standard graphic symbols for logic functions recommended by ANSI/IEEE standard 91-1984. These graphic symbols have been developed for SSI and MSI components so that the user can recognize each function from the unique graphic symbol assigned to it. The best time to learn the standard symbols is while learning about digital systems. Chapter 12 shows the standard graphic symbols of all the integrated circuits used in the laboratory experiments of Chapter 11.

The various digital componets that are represented throughout the book are similar to commercial MSI circuits. However, the text does not mention specific integrated circuits except in Chapters 11 and 12. The practical application of digital design will be enhanced by doing the suggested experiments in Chapter 11 while studying the theory presented in the text.

Each chapter in the book has a list of references and a set of problems. Answers to most of the problems appear in the Appendix to aid the student and to help the independent reader. A *solutions manual* is available for the instructor from the publisher.

M. Morris Mano

Binary Systems

1-1 DIGITAL COMPUTERS AND DIGITAL SYSTEMS

Digital computers have made possible many scientific, industrial, and commercial advances that would have been unattainable otherwise. Our space program would have been impossible without real-time, continuous computer monitoring, and many business enterprises function efficiently only with the aid of automatic data processing. Computers are used in scientific calculations, commercial and business data processing, air traffic control, space guidance, the educational field, and many other areas. The most striking property of a digital computer is its generality. It can follow a sequence of instructions, called a *program,* that operates on given data. The user can specify and change programs and/or data according to the specific need. As a result of this flexibility, general-purpose digital computers can perform a wide variety of information-processing tasks.

The general-purpose digital computer is the best-known example of a digital system. Other examples include telephone switching exchanges, digital voltmeters, digital counters, electronic calculators, and digital displays. Characteristic of a digital system is its manipulation of *discrete elements* of information. Such discrete elements may be electric impulses, the decimal digits, the letters of an alphabet, arithmetic operations, punctuation marks, or any other set of meaningful symbols. The juxtaposition of discrete elements of information represents a quantity of information. For example, the letters *d, o,* and *g* form the word *dog*. The digits 237 form a number. Thus, a sequence of discrete elements forms a language, that is, a discipline that conveys information. Early digital computers were used mostly for numerical computations. In this case, the

discrete elements used are the digits. From this application, the term *digital computer* has emerged. A more appropriate name for a digital computer would be a "discrete information-processing system."

Discrete elements of information are represented in a digital system by physical quantities called *signals*. Electrical signals such as voltages and currents are the most common. The signals in all present-day electronic digital systems have only two discrete values and are said to be *binary*. The digital-system designer is restricted to the use of binary signals because of the lower reliability of many-valued electronic circuits. In other words, a circuit with ten states, using one discrete voltage value for each state, can be designed, but it would possess a very low reliability of operation. In contrast, a transistor circuit that is either on or off has two possible signal values and can be constructed to be extremely reliable. Because of this physical restriction of components, and because human logic tends to be binary, digital systems that are constrained to take discrete values are further constrained to take binary values.

Discrete quantities of information arise either from the nature of the process or may be quantized from a continuous process. For example, a payroll schedule is an inherently discrete process that contains employee names, social security numbers, weekly salaries, income taxes, etc. An employee's paycheck is processed using discrete data values such as letters of the alphabet (names), digits (salary), and special symbols such as $. On the other hand, a research scientist may observe a continuous process but record only specific quantities in tabular form. The scientist is thus quantizing his continuous data. Each number in his table is a discrete element of information.

Many physical systems can be described mathematically by differential equations whose solutions as a function of time give the complete mathematical behavior of the process. An *analog computer* performs a direct *simulation* of a physical system. Each section of the computer is the analog of some particular portion of the process under study. The variables in the analog computer are represented by continuous signals, usually electric voltages that vary with time. The signal variables are considered analogous to those of the process and behave in the same manner. Thus, measurements of the analog voltage can be substituted for variables of the process. The term *analog signal* is sometimes substituted for *continuous signal* because "analog computer" has come to mean a computer that manipulates continuous variables.

To simulate a physical process in a digital computer, the quantities must be quantized. When the variables of the process are presented by real-time continuous signals, the latter are quantized by an analog-to-digital conversion device. A physical system whose behavior is described by mathematical equations is simulated in a digital computer by means of numerical methods. When the problem to be processed is inherently discrete, as in commercial applications, the digital computer manipulates the variables in their natural form.

A block diagram of the digital computer is shown in Fig. 1-1. The memory unit stores programs as well as input, output, and intermediate data. The processor unit performs arithmetic and other data-processing tasks as specified by a program. The control unit supervises the flow of information between the various units. The control unit retrieves the instructions, one by one, from the program that is stored in memory. For

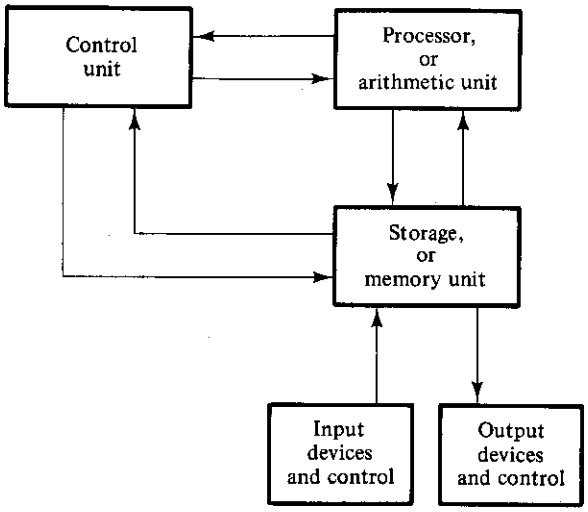


FIGURE 1-1
Block diagram of a digital computer

each instruction, the control unit informs the processor to execute the operation specified by the instruction. Both program and data are stored in memory. The control unit supervises the program instructions, and the processor manipulates the data as specified by the program.

The program and data prepared by the user are transferred into the memory unit by means of an input device such as a keyboard. An output device, such as a printer, receives the result of the computations and the printed results are presented to the user. The input and output devices are special digital systems driven by electromechanical parts and controlled by electronic digital circuits.

An electronic calculator is a digital system similar to a digital computer, with the input device being a keyboard and the output device a numerical display. Instructions are entered in the calculator by means of the function keys, such as plus and minus. Data are entered through the numeric keys. Results are displayed directly in numeric form. Some calculators come close to resembling a digital computer by having printing capabilities and programmable facilities. A digital computer, however, is a more powerful device than a calculator. A digital computer can accommodate many other input and output devices; it can perform not only arithmetic computations, but logical operations as well and can be programmed to make decisions based on internal and external conditions.

A digital computer is an interconnection of digital modules. To understand the operation of each digital module, it is necessary to have a basic knowledge of digital systems and their general behavior. The first four chapters of the book introduce the basic tools of digital design such as binary numbers and codes, Boolean algebra, and the basic building blocks from which electronic digital circuits are constructed. Chapters 5 and 7 present the basic components found in the processor unit of a digital computer.

The operational characteristics of the memory unit are explained at the end of Chapter 7. The design of the control unit is discussed in Chapter 8 using the basic principles of sequential circuits from Chapter 6.

It has already been mentioned that a digital computer manipulates discrete elements of information and that these elements are represented in the binary form. Operands used for calculations may be expressed in the binary number system. Other discrete elements, including the decimal digits, are represented in binary codes. Data processing is carried out by means of binary logic elements using binary signals. Quantities are stored in binary storage elements. The purpose of this chapter is to introduce the various binary concepts as a frame of reference for further detailed study in the succeeding chapters.

1-2 BINARY NUMBERS

A decimal number such as 7392 represents a quantity equal to 7 thousands plus 3 hundreds, plus 9 tens, plus 2 units. The thousands, hundreds, etc. are powers of 10 implied by the position of the coefficients. To be more exact, 7392 should be written as

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

However, the convention is to write only the coefficients and from their position deduce the necessary powers of 10. In general, a number with a decimal point is represented by a series of coefficients as follows:

$$a_5 a_4 a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3}$$

The a_j coefficients are one of the ten digits (0, 1, 2, . . . , 9), and the subscript value j gives the place value and, hence, the power of 10 by which the coefficient must be multiplied.

$$10^5 a_5 + 10^4 a_4 + 10^3 a_3 + 10^2 a_2 + 10^1 a_1 + 10^0 a_0 + 10^{-1} a_{-1} + 10^{-2} a_{-2} + 10^{-3} a_{-3}$$

The decimal number system is said to be of *base*, or *radix*, 10 because it uses ten digits and the coefficients are multiplied by powers of 10. The *binary* system is a different number system. The coefficients of the binary numbers system have two possible values: 0 and 1. Each coefficient a_j is multiplied by 2^j. For example, the decimal equivalent of the binary number 11010.11 is 26.75, as shown from the multiplication of the coefficients by powers of 2:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 26.75$$

In general, a number expressed in base-r system has coefficients multiplied by powers of r:

$$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_2 \cdot r^2 + a_1 \cdot r + a_0$$
$$+ a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + \cdots + a_{-m} \cdot r^{-m}$$

The coefficients a_j range in value from 0 to $r - 1$. To distinguish between numbers of different bases, we enclose the coefficients in parentheses and write a subscript equal to the base used (except sometimes for decimal numbers, where the content makes it obvious that it is decimal). An example of a base-5 number is

$$(4021.2)_5 = 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} = (511.4)_{10}$$

Note that coefficient values for base 5 can be only 0, 1, 2, 3, and 4.

It is customary to borrow the needed r digits for the coefficients from the decimal system when the base of the number is less than 10. The letters of the alphabet are used to supplement the ten decimal digits when the base of the number is greater than 10. For example, in the *hexadecimal* (base 16) number system, the first ten digits are borrowed from the decimal system. The letters A, B, C, D, E, and F are used for digits 10, 11, 12, 13, 14, and 15, respectively. An example of a hexadecimal number is

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16 + 15 = (46687)_{10}$$

The first 16 numbers in the decimal, binary, octal, and hexadecimal systems are listed in Table 1-1.

TABLE 1-1
Numbers with Different Bases

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Arithmetic operations with numbers in base r follow the same rules as for decimal numbers. When other than the familiar base 10 is used, one must be careful to use only the r allowable digits. Examples of addition, subtraction, and multiplication of two binary numbers are as follows:

augend:	101101	minuend:	101101	multiplicand:	1011
addend:	+100111	subtrahend:	−100111	multiplier:	× 101
sum:	1010100	difference:	000110		1011
					0000
					1011
				product:	110111

The sum of two binary numbers is calculated by the same rules as in decimal, except that the digits of the sum in any significant position can be only 0 or 1. Any carry obtained in a given significant position is used by the pair of digits one significant position higher. The subtraction is slightly more complicated. The rules are still the same as in decimal, except that the borrow in a given significant position adds 2 to a minuend digit. (A borrow in the decimal system adds 10 to a minuend digit.) Multiplication is very simple. The multiplier digits are always 1 or 0. Therefore, the partial products are equal either to the multiplicand or to 0.

1-3 NUMBER BASE CONVERSIONS

A binary number can be converted to decimal by forming the sum of the powers of 2 of those coefficients whose value is 1. For example

$$(1010.011)_2 = 2^3 + 2^1 + 2^{-2} + 2^{-3} = (10.375)_{10}$$

The binary number has four 1's and the decimal equivalent is found from the sum of four powers of 2. Similarly, a number expressed in base r can be converted to its decimal equivalent by multiplying each coefficient with the corresponding power of r and adding. The following is an example of octal-to-decimal conversion:

$$(630.4)_8 = 6 \times 8^2 + 3 \times 8 + 4 \times 8^{-1} = (408.5)_{10}$$

The conversion from decimal to binary or to any other base-r system is more convenient if the number is separated into an *integer part* and a *fraction part* and the conversion of each part done separately. The conversion of an *integer* from decimal to binary is best explained by example.

Example 1-1 Convert decimal 41 to binary. First, 41 is divided by 2 to give an integer quotient of 20 and a remainder of $\frac{1}{2}$. The quotient is again divided by 2 to give a new quotient and remainder. This process is continued until the integer quotient becomes 0. The *coefficients* of the desired binary number are obtained from the *remainders* as follows:

	Integer quotient		Remainder	Coefficient
$\frac{41}{2} =$	20	+	$\frac{1}{2}$	$a_0 = 1$
$\frac{20}{2} =$	10	+	0	$a_1 = 0$
$\frac{10}{2} =$	5	+	0	$a_2 = 0$
$\frac{5}{2} =$	2	+	$\frac{1}{2}$	$a_3 = 1$
$\frac{2}{2} =$	1	+	0	$a_4 = 0$
$\frac{1}{2} =$	0	+	$\frac{1}{2}$	$a_5 = 1$

answer: $(41)_{10} = (a_5a_4a_3a_2a_1a_0)_2 = (101001)_2$

The arithmetic process can be manipulated more conveniently as follows:

Integer	Remainder
41	
20	1
10	0
5	0
2	1
1	0
0	1

101001 = answer ■

The conversion from decimal integers to any base-r system is similar to the example, except that division is done by r instead of 2.

Example 1-2

Convert decimal 153 to octal. The required base r is 8. First, 153 is divided by 8 to give an integer quotient of 19 and a remainder of 1. Then 19 is divided by 8 to give an integer quotient of 2 and a remainder of 3. Finally, 2 is divided by 8 to give a quotient of 0 and a remainder of 2. This process can be conveniently manipulated as follows:

153	
19	1
2	3
0	2

$= (231)_8$ ■

The conversion of a decimal *fraction* to binary is accomplished by a method similar to that used for integers. However, multiplication is used instead of division, and integers are accumulated instead of remainders. Again, the method is best explained by example.

Example 1-3

Convert $(0.6875)_{10}$ to binary. First, 0.6875 is multiplied by 2 to give an integer and a fraction. The new fraction is multiplied by 2 to give a new integer and a new fraction. This process is continued until the fraction becomes 0 or until the number of digits have sufficient accuracy. The coefficients of the binary number are obtained from the integers as follows:

	Integer		Fraction	Coefficient
$0.6875 \times 2 =$	1	+	0.3750	$a_{-1} = 1$
$0.3750 \times 2 =$	0	+	0.7500	$a_{-2} = 0$
$0.7500 \times 2 =$	1	+	0.5000	$a_{-3} = 1$
$0.5000 \times 2 =$	1	+	0.0000	$a_{-4} = 1$

Answer: $(0.6875)_{10} = (0.a_{-1}a_{-2}a_{-3}a_{-4})_2 = (0.1011)_2$ ■

To convert a decimal fraction to a number expressed in base r, a similar procedure is used. Multiplication is by r instead of 2, and the coefficients found from the integers may range in value from 0 to $r - 1$ instead of 0 and 1.

Example 1-4

Convert $(0.513)_{10}$ to octal.

$$0.513 \times 8 = 4.104$$
$$0.104 \times 8 = 0.832$$
$$0.832 \times 8 = 6.656$$
$$0.656 \times 8 = 5.248$$
$$0.248 \times 8 = 1.984$$
$$0.984 \times 8 = 7.872$$

The answer, to seven significant figures, is obtained from the integer part of the products:

$$(0.513)_{10} = (0.406517\ldots)_8$$ ■

The conversion of decimal numbers with both integer and fraction parts is done by converting the integer and fraction separately and then combining the two answers. Using the results of Examples 1-1 and 1-3, we obtain

$$(41.6875)_{10} = (101001.1011)_2$$

From Examples 1-2 and 1-4, we have

$$(153.513)_{10} = (231.406517)_8$$

1-4 OCTAL AND HEXADECIMAL NUMBERS

The conversion from and to binary, octal, and hexadecimal 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 digits each, starting from the binary point and proceeding to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

$$\underset{2}{\underbrace{10}}\ \underset{6}{\underbrace{110}}\ \underset{1}{\underbrace{001}}\ \underset{5}{\underbrace{101}}\ \underset{3}{\underbrace{011}}\ .\ \underset{7}{\underbrace{111}}\ \underset{4}{\underbrace{100}}\ \underset{0}{\underbrace{000}}\ \underset{6}{\underbrace{110}}\ )_2 = (26153.7460)_8$$

Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of four digits:

$$\underset{2}{\underbrace{10}}\ \underset{C}{\underbrace{1100}}\ \underset{6}{\underbrace{0110}}\ \underset{B}{\underbrace{1011}}\ .\ \underset{F}{\underbrace{1111}}\ \underset{2}{\underbrace{0010}}\ )_2 = (2C6B.F2)_{16}$$

The corresponding hexadecimal (or octal) digit for each group of binary digits is easily remembered after studying the values listed in Table 1-1.

Conversion from octal or hexadecimal to binary is done by a procedure reverse to the above. Each octal digit is converted to its three-digit binary equivalent. Similarly, each hexadecimal digit is converted to its four-digit binary equivalent. This is illustrated in the following examples:

$$(673.124)_8 = (\ \underset{6}{\underbrace{110}}\ \underset{7}{\underbrace{111}}\ \underset{3}{\underbrace{011}}\ .\ \underset{1}{\underbrace{001}}\ \underset{2}{\underbrace{010}}\ \underset{4}{\underbrace{100}}\ )_2$$

$$(306.D)_{16} = (\ \underset{3}{\underbrace{0011}}\ \underset{0}{\underbrace{0000}}\ \underset{6}{\underbrace{0110}}\ .\ \underset{D}{\underbrace{1101}}\ )_2$$

Binary numbers are difficult to work with because they require three or four times as many digits as their decimal equivalent. For example, the binary number 111111111111 is equivalent to decimal 4095. However, digital computers use binary numbers and it is sometimes necessary for the human operator or user to communicate directly with the machine by means of binary numbers. One scheme that retains the binary system in the computer but reduces the number of digits the human must consider

utilizes the relationship between the binary number system and the octal or hexadecimal system. By this method, the human thinks in terms of octal or hexadecimal numbers and performs the required conversion by inspection when direct communication with the machine is necessary. Thus the binary number 111111111111 has 12 digits and is expressed in octal as 7777 (four digits) or in hexadecimal as FFF (three digits). During communication between people (about binary numbers in the computer), the octal or hexadecimal representation is more desirable because it can be expressed more compactly with a third or a quarter of the number of digits required for the equivalent binary number. When the human communicates with the machine (through console switches or indicator lights or by means of programs written in *machine language*), the conversion from octal or hexadecimal to binary and vice versa is done by inspection by the human user.

1-5 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 radix complement and the diminished radix complement. The first is referred to as the r's complement and the second as 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 complement and 1's complement for binary numbers, and the 10's complement and 9's complement for decimal numbers.

Diminished Radix 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, if $n = 4$, we have $10^4 = 10{,}000$ and $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9. Some numerical examples follow.

The 9's complement of 546700 is $999999 - 546700 = 453299$.

The 9's complement of 012398 is $999999 - 012398 = 987601$.

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, if $n = 4$, 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, when subtracting binary digits from 1, we can have either $1 - 0 = 1$ or $1 - 1 = 0$, which 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 to 0's and 0's to 1's. The following are some numerical examples.

The 1's complement of 1011000 is 0100111.

The 1's complement of 0101101 is 1010010.

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

Radix 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 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.

Since 10^n is a number represented by a 1 followed by n 0's, $10^n - N$, which is the 10's complement of N, can be formed also by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and subtracting all higher significant digits from 9.

The 10's complement of 012398 is 987602.

The 10's complement of 246700 is 753300.

The 10's complement of the first number is obtained by subtracting 8 from 10 in the least significant position and subtracting all other digits from 9. The 10's complement of the second number is obtained by leaving the two least significant 0's 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 replacing 1's with 0's and 0's with 1's in all other higher significant digits.

The 2's complement of 1101100 is 0010100.

The 2's complement of 0110111 is 1001001.

The 2's complement of the first number is obtained by leaving the two least significant 0's and the first 1 unchanged, and then replacing 1's with 0's and 0's with 1's in the other four most-significant digits. The 2's complement of the second number is obtained by leaving the least significant 1 unchanged and complementing all other digits.

In the previous definitions, it was assumed that the numbers do not have a radix point. If the original number N contains a radix point, the point should be removed

temporarily in order 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 with Complements

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 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.

The subtraction of two n-digit unsigned numbers $M - N$ 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; what is left is the result $M - N$.
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.

The following examples illustrate the procedure.

Example 1-5 Using 10's complement, subtract 72532 − 3250.

M =	72532
10's complement of N =	+ 96750
Sum =	169282
Discard end carry 10^5 =	−100000
Answer =	69282 ■

Note that M has 5 digits and N has only 4 digits. Both numbers must have the same number of digits; so we can write N as 03250. Taking the 10's complement of N produces a 9 in the most significant position. The occurrence of the end carry signifies that $M \geq N$ and the result is positive.

Example 1-6 Using 10's complement, subtract 3250 − 72532.

$$\begin{aligned} M &= \quad 03250 \\ \text{10's complement of } N &= +\ 27468 \\ \text{Sum} &= \quad 30718 \end{aligned}$$

There is no end carry.

Answer: −(10's complement of 30718) = −69282 ■

Note that since 3250 < 72532, the result is negative. Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for this case. When subtracting with complements, the negative answer is recognized from the absence of the end carry and the complemented result. When working with paper and pencil, we can change the answer to a signed negative number in order to put it in a familiar form.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined before.

Example 1-7 Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction (a) $X - Y$ and (b) $Y - X$ using 2's complements.

(a)

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

(b)

$$\begin{aligned} Y &= \quad 1000011 \\ \text{2's complement of } X &= +\ 0101100 \\ \text{Sum} &= \quad 1101111 \end{aligned}$$

There is no end carry.

Answer: $Y - X = -$(2's complement of 1101111) = −0010001 ■

Subtraction of unsigned numbers can be done also by means of the $(r - 1)$'s complement. Remember that the $(r - 1)$'s complement is one less than the r's complement. Because of this, the result of adding the minuend to the complement of the subtrahend produces a sum that is 1 less than the correct difference when an end carry occurs. Removing the end carry and adding 1 to the sum is referred to as an *end-around carry*.

Example 1-8 Repeat Example 1-7 using 1's complement.

(a) $X - Y = 1010100 - 1000011$

X =	1010100
1's complement of Y =	+ 0111100
Sum =	10010000
End-around carry	+ 1
Answer: $X - Y$ =	0010001

(b) $Y - X = 1000011 - 1010100$

Y =	1000011
1's complement of X =	+ 0101011
Sum =	1101110

There is no end carry.

Answer: $Y - X = -$(1's complement of 1101110) $= -0010001$

Note that the negative result is obtained by taking the 1's complement of the sum since this is the type of complement used. The procedure with end-around carry is also applicable for subtracting unsigned decimal numbers with 9's complement.

1-6 SIGNED BINARY NUMBERS

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 binary digits, commonly referred to as *bits*. 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 0 for positive and 1 for negative.

It is important to realize that both signed and unsigned binary numbers consist of a string of bits when represented in a computer. The user determines whether the number is signed or unsigned. If the binary number is signed, then the leftmost bit represents the sign and the rest of the bits represent the number. If the binary number is assumed to be unsigned, then the leftmost bit is the most significant bit of the number. For example, the string of bits 01001 can be considered as 9 (unsigned binary) or a +9 (signed binary) because the leftmost bit is 0. The string of bits 11001 represent the binary equivalent of 25 when considered as an unsigned number or as − 9 when considered as a signed number because of the 1 in the leftmost position, which designates neg-

ative, and the other four bits, which represent binary 9. Usually, there is no confusion in identifying the bits if the type of representation for the number is known in advance.

The representation of the signed numbers in the last example is referred to as the *signed-magnitude* convention. In this notation, the number consists of a magnitude and a symbol (+ or −) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic. When arithmetic operations are implemented in a computer, it is more convenient to use a different system for representing negative numbers, referred to as the *signed-complement* system. In this system, a negative number is indicated by its complement. Whereas the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement. Since positive numbers always start with 0 (plus) in the leftmost position, the complement will always start with a 1, indicating a negative number. The signed-complement system can use either the 1's or the 2's complement, but the 2's complement is the most common.

As an example, consider the number 9 represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position followed by the binary equivalent of 9 to give 00001001. Note that all eight bits must have a value and, therefore, 0's are inserted following the sign bit up to the first 1. Although there is only one way to represent +9, there are three different ways to represent − 9 with eight bits:

In signed-magnitude representation:	10001001
In signed-1's-complement representation:	11110110
In signed-2's-complement representation:	11110111

In signed-magnitude, −9 is obtained from +9 by changing the sign bit in the leftmost position from 0 to 1. In signed-1's complement, −9 is obtained by complementing all the bits of +9, including the sign bit. The signed-2's-complement representation of −9 is obtained by taking the 2's complement of the positive number, including the 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 some difficulties and is seldom used for arithmetic operations except in some older computers. The 1'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, as will be shown in the next chapter. The following discussion of signed binary arithmetic deals exclusively with the signed-2's-complement representation of negative numbers. The same procedures can be applied to the signed-1's-complement system by including the end-around carry as done with unsigned 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 same procedure applies to binary numbers in signed-magnitude representation. In contrast, the rule for adding numbers in the signed-complement system does not require a comparison or subtraction, but only addition. The procedure is very simple and can be stated as follows for binary numbers.

> The addition of two signed binary numbers with negative numbers represented in signed-2's-complement form is obtained from the addition of the two numbers, including their sign bits. A carry out of the sign-bit position is discarded.

Numerical examples for addition follow. Note that negative numbers must be initially in 2's complement and that the sum obtained after the addition if negative 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 addition with the sign bit included. Any carry out of the sign-bit position is discarded, and negative results are automatically in 2's-complement form.

In order to obtain a correct answer, we must ensure that the result has a sufficient number of bits to accommodate the sum. If we start with two n-bit numbers and the sum occupies $n + 1$ bits, we say that an overflow occurs. When one performs the addition with paper and pencil, an overflow is not a problem since we are not limited by the width of the page. We just add another 0 to a positive number and another 1 to a negative number in the most-significant position to extend them to $n + 1$ bits and then perform the addition. Overflow is a problem in computers because the number of bits that hold a number is finite, and a result that exceeds the finite value by 1 cannot be accommodated.

The complement form of representing negative numbers is unfamiliar to those 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 occurs because 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:

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

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-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.

1-7 BINARY CODES

Electronic digital systems use signals that have two distinct values and circuit elements that have two stable states. There is a direct analogy among binary signals, binary circuit elements, and binary digits. A binary number of n digits, for example, may be represented by n binary circuit elements, each having an output signal equivalent to a 0 or a 1. Digital systems represent and manipulate not only binary numbers, but also many other discrete elements of information. Any discrete element of information distinct among a group of quantities can be represented by a binary code. Binary codes play an important role in digital computers. The codes must be in binary because computers can only hold 1's and 0's. It must be realized that binary codes merely change the symbols, not the meaning of the elements of information that they represent. If we inspect the bits of a computer at random, we will find that most of the time they represent some type of coded information rather than binary numbers.

A *bit*, by definition, is a binary digit. When used in conjunction with a binary code, it is better to think of it as denoting a binary quantity equal to 0 or 1. To represent a

group of 2^n distinct elements in a binary code requires a minimum of n bits. This is because it is possible to arrange n bits in 2^n distinct ways. For example, a group of four distinct quantities can be represented by a two-bit code, with each quantity assigned one of the following bit combinations: 00, 01, 10, 11. A group of eight elements requires a three-bit code, with each element assigned to one and only one of the following: 000, 001, 010, 011, 100, 101, 110, 111. The examples show that the distinct bit combinations of an n-bit code can be found by counting in binary from 0 to $(2^n - 1)$. Some bit combinations are unassigned when the number of elements of the group to be coded is not a multiple of the power of 2. The ten decimal digits 0, 1, 2, . . . , 9 are an example of such a group. A binary code that distinguishes among ten elements must contain at least four bits; three bits can distinguish a maximum of eight elements. Four bits can form 16 distinct combinations, but since only ten digits are coded, the remaining six combinations are unassigned and not used.

Although the *minimum* number of bits required to code 2^n distinct quantities is n, there is no *maximum* number of bits that may be used for a binary code. For example, the ten decimal digits can be coded with ten bits, and each decimal digit assigned a bit combination of nine 0's and a 1. In this particular binary code, the digit 6 is assigned the bit combination 0001000000.

Decimal Codes

Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be obtained by arranging four or more bits in ten distinct possible combinations. A few possibilities are shown in Table 1-2.

TABLE 1-2
Binary codes for the decimal digits

Decimal digit	(BCD) 8421	Excess-3	84-2-1	2421	(Biquinary) 5043210
0	0000	0011	0000	0000	0100001
1	0001	0100	0111	0001	0100010
2	0010	0101	0110	0010	0100100
3	0011	0110	0101	0011	0101000
4	0100	0111	0100	0100	0110000
5	0101	1000	1011	1011	1000001
6	0110	1001	1010	1100	1000010
7	0111	1010	1001	1101	1000100
8	1000	1011	1000	1110	1001000
9	1001	1100	1111	1111	1010000

The BCD (binary-code decimal) is a straight assignment of the binary equivalent. It is possible to assign weights to the binary bits according to their positions. The weights in the BCD code are 8, 4, 2, 1. The bit assignment 0110, for example, can be interpreted by the weights to represent the decimal digit 6 because $0 \times 8 + 1 \times 4 +$

$1 \times 2 + 0 \times 1 = 6$. It is also possible to assign negative weights to a decimal code, as shown by the 8, 4, −2, −1 code. In this case, the bit combination 0110 is interpreted as the decimal digit 2, as obtained from $0 \times 8 + 1 \times 4 + 1 \times (-2) + 0 \times (-1) = 2$. Two other weighted codes shown in the table are the 2421 and the 5043210. A decimal code that has been used in some old computers is the excess-3 code. This is an unweighted code; its code assignment is obtained from the corresponding value of BCD after the addition of 3.

Numbers are represented in digital computers either in binary or in decimal through a binary code. When specifying data, the user likes to give the data in decimal form. The input decimal numbers are stored internally in the computer by means of a decimal code. Each decimal digit requires at least four binary storage elements. The decimal numbers are converted to binary when arithmetic operations are done internally with numbers represented in binary. It is also possible to perform the arithmetic operations directly in decimal with all numbers left in a coded form throughout. For example, the decimal number 395, when converted to binary, is equal to 110001011 and consists of nine binary digits. The same number, when represented internally in the BCD code, occupies four bits for each decimal digit, for a total of 12 bits: 001110010101. The first four bits represent a 3, the next four a 9, and the last four a 5.

It is very important to understand the difference between *conversion* of a decimal number to binary and the binary *coding* of a decimal number. In each case, the final result is a series of bits. The bits obtained from conversion are binary digits. Bits obtained from coding are combinations of 1's and 0's arranged according to the rules of the code used. Therefore, it is extremely important to realize that a series of 1's and 0's in a digital system may sometimes represent a binary number and at other times represent some other discrete quantity of information as specified by a given binary code. The BCD code, for example, has been chosen to be both a code and a direct binary conversion, as long as the decimal numbers are integers from 0 to 9. For numbers greater than 9, the conversion and the coding are completely different. This concept is so important that it is worth repeating with another example. The binary conversion of decimal 13 is 1101; the coding of decimal 13 with BCD is 00010011.

From the five binary codes listed in Table 1-2, the BCD seems the most natural to use and is indeed the one most commonly encountered. The other four-bit codes listed have one characteristic in common that is not found in BCD. The excess-3, the 2, 4, 2, 1, and the 8, 4, −2, −1 are self-complementing codes, that is, the 9's complement of the decimal number is easily obtained by changing 1's to 0's and 0's to 1's. For example, the decimal 395 is represented in the 2, 4, 2, 1 code by 001111111011. Its 9's complement 604 is represented by 110000000100, which is easily obtained from the replacement of 1's by 0's and 0's by 1's. This property is useful when arithmetic operations are internally done with decimal numbers (in a binary code) and subtraction is calculated by means of 9's complement.

The biquinary code shown in Table 1-2 is an example of a seven-bit code with error-detection properties. Each decimal digit consists of five 0's and two 1's placed in the corresponding weighted columns. The error-detection property of this code may be understood if one realizes that digital systems represent binary 1 by one distinct signal

and binary 0 by a second distinct signal. During transmission of signals from one location to another, an error may occur. One or more bits may change value. A circuit in the receiving side can detect the presence of more (or less) than two 1's and if the received combination of bits does not agree with the allowable combination, an error is detected.

Error-Detection Code

Binary information can be transmitted from one location to another by electric wires or other communication medium. Any external noise introduced into the physical communication medium may change some of the bits from 0 to 1 or vice versa. The purpose of an error-detection code is to detect such bit-reversal errors. One of the most common ways to achieve error detection is by means of a *parity bit*. A parity bit is an extra bit included with a message to make the total number of 1's transmitted either odd or even. A message of four bits and a parity bit P are shown in Table 1-3. If an odd parity is adopted, the P bit is chosen such that the total number of 1's is odd in the five bits that constitute the message and P. If an even parity is adopted, the P bit is chosen so that the total number of 1's in the five bits is even. In a particular situation, one or the other parity is adopted, with even parity being more common.

The parity bit is helpful in detecting errors during the transmission of information from one location to another. This is done in the following manner. An even parity bit is generated in the sending end for each message transmission. The message, together with the parity bit, is transmitted to its destination. The parity of the received data is

TABLE 1-3
Parity bit

Odd parity		Even parity	
Message	P	Message	P
0000	1	0000	0
0001	0	0001	1
0010	0	0010	1
0011	1	0011	0
0100	0	0100	1
0101	1	0101	0
0110	1	0110	0
0111	0	0111	1
1000	0	1000	1
1001	1	1001	0
1010	1	1010	0
1011	0	1011	1
1100	1	1100	0
1101	0	1101	1
1110	0	1110	1
1111	1	1111	0

checked in the receiving end. If the parity of the received information is not even, it means that at least one bit has changed value during the transmission. This method detects one, three, or any odd combination of errors in each message that is transmitted. An even combination of errors is undetected. Additional error-detection schemes may be needed to take care of an even combination of errors.

What is done after an error is detected depends on the particular application. One possibility is to request retransmission of the message on the assumption that the error was random and will not occur again. Thus, if the receiver detects a parity error, it sends back a negative acknowledge message. If no error is detected, the receiver sends back an acknowledge message. The sending end will respond to a previous error by transmitting the message again until the correct parity is received. If, after a number of attempts, the transmission is still in error, a message can be sent to the human operator to check for malfunctions in the transmission path.

Gray Code

Digital systems can be designed to process data in discrete form only. Many physical systems supply continuous output data. These data must be converted into digital form before they are applied to a digital system. Continuous or analog information is converted into digital form by means of an analog-to-digital converter. It is sometimes convenient to use the Gray code shown in Table 1-4 to represent the digital data when it is converted from analog data. The advantage of the Gray code over binary numbers is that only one bit in the code group changes when going from one number to the next. For example, in going from 7 to 8, the Gray code changes from 0100 to 1100. Only the

TABLE 1-4
Four-bit Gray code

Gray code	Decimal equivalent
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9
1111	10
1110	11
1010	12
1011	13
1001	14
1000	15

first bit from the left changes from 0 to 1; the other three bits remain the same. When comparing this with binary numbers, the change from 7 to 8 will be from 0111 to 1000, which causes all four bits to change values.

The Gray code is used in applications where the normal sequence of binary numbers may produce an error or ambiguity during the transition from one number to the next. If binary numbers are used, a change from 0111 to 1000 may produce an intermediate erroneous number 1001 if the rightmost bit takes more time to change than the other three bits. The Gray code eliminates this problem since only one bit changes in value during any transition between two numbers.

A typical application of the Gray code occurs when analog data are represented by continuous change of a shaft position. The shaft is partitioned into segments, and each segment is assigned a number. If adjacent segments are made to correspond with the Gray-code sequence, ambiguity is eliminated when detection is sensed in the line that separates any two segments.

ASCII Character Code

Many applications of digital computers require the handling of data not only of numbers, but also of letters. For instance, an insurance company with thousands of policy holders will use a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters such as $. 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 a set contains between 36 and 64 elements if only capital letters are included, or between 64 and 128 elements if both uppercase and lowercase letters are included. In the first case, we need a binary code of six bits, and in the second we need a binary code of seven bits.

The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters, as shown in Table 1-5. The seven bits of the code are designated by b_1 through b_7, with b_7 being the most-significant bit. The letter *A*, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions. The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters such as %, *, and $.

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed in the table with their full functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication-control characters. Format effectors are characters that control the layout of printing. They include the familiar typewriter controls such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to

TABLE 1-5
American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	¦
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	−	o	DEL

Control characters

NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

separate the data into divisions such as paragraphs and pages. They include characters such as 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 telephone wires.

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 character.

Other Alphanumeric Codes

Another alphanumeric code used in IBM equipment is the EBCDIC (Extended Binary-Coded Decimal Interchange Code). It uses eight bits for each character. EBCDIC has the same character symbols as ASCII, but the bit assignment for characters is different. As the name implies, the binary code for the letters and numerals is an extension of the binary-coded decimal (BCD) code. This means that the last four bits of the code range from 0000 though 1001 as in BCD.

When characters are used internally in a computer for data processing (not for transmission purposes), it is sometimes convenient to use a 6-bit code to represent 64 characters. A 6-bit code can specify 64 characters consisting of the 26 capital letters, the 10 numerals, and up to 28 special characters. This set of characters is usually sufficient for data-processing purposes. Using fewer bits to code characters has the advantage of reducing the space needed to store large quantities of alphanumeric data.

A code developed in the early stages of teletype transmission is the 5-bit Baudot code. Although five bits can specify only 32 characters, the Baudot code represents 58 characters by using two modes of operation. In the mode called *letters*, the five bits encode the 26 letters of the alphabet. In the mode called *figures*, the five bits encode the numerals and other characters. There are two special characters that are recognized by both modes and used to shift from one mode to the other. The *letter-shift* character places the reception station in the letters mode, after which all subsequent character codes are interpreted as letters. The *figure-shift* character places the system in the figures mode. The shift operation is analogous to the shifting operation on a typewriter with a shift lock key.

When alphanumeric information is transferred to the computer using punched cards, the alphanumeric characters are coded with 12 bits. Programs and data in the past were prepared on punched cards using the Hollerith code. A punched card consists of 80 columns and 12 rows. Each column represents an alphanumeric character of 12 bits with holes punched in the appropriate rows. A hole is sensed as a 1 and the absence of a hole is sensed as a 0. The 12 rows are marked, starting from the top, as 12, 11, 0, 1,

2, . . . , 9. The first three are called the zone punch and the last nine are called the numeric punch. Decimal digits are represented by a single hole in a numeric punch. The letters of the alphabet are represented by two holes in a column, one in the zone punch and the other the numeric punch. Special characters are represented by one, two, or three holes in a column. The 12-bit card code is ineffecient in its use of bits. Consequently, computers that receive input from a card reader convert the input 12-bit card code into an internal six-bit code to conserve bits of storage.

1-8 BINARY STORAGE AND REGISTERS

The discrete elements of information in a digital computer must have a physical existence in some information-storage medium. Furthermore, when discrete elements of information are represented in binary form, the information-storage medium must contain binary storage elements for storing individual bits. A *binary cell* is a device that possesses two stable states and is capable of storing one bit of information. The input to the cell receives excitation signals that set it to one of the two states. The output of the cell is a physical quantity that distinguishes between the two states. The information stored in a cell is a 1 when it is in one stable state and a 0 when in the other stable state. Examples of binary cells are electronic flip-flop circuits, ferrite cores used in memories, and positions punched with a hole or not punched in a card.

Registers

A *register* is a group of binary cells. Since a cell stores one bit of information, it follows that a register with n cells can store any discrete quantity of information that contains n bits. The *state* of a register is an n-tuple number of 1's and 0's, with each bit designating the state of one cell in the register. The *content* of a register is a function of the interpretation given to the information stored in it. Consider, for example, the following 16-cell register:

1	1	0	0	0	0	1	1	1	1	0	0	1	0	0	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Physically, one may think of the register as composed of 16 binary cells, with each cell storing either a 1 or a 0. Suppose that the bit configuration stored in the register is as shown. The state of the register is the 16-tuple number 1100001111001001. Clearly, a register with n cells can be in one of 2^n possible states. Now, if one assumes that the content of the register represents a binary integer, then obviously the register can store any binary number from 0 to $2^{16} - 1$. For the particular example shown, the content of the register is the binary equivalent of the decimal number 50121. If it is assumed that the register stores alphanumeric characters of an eight-bit code, the content of the reg-

ister is any two meaningful characters. For the ASCII code with an even parity placed in the eighth most-significant bit position the previous example represents the two characters C (left eight bits) and I (right eight bits). On the other hand, if one interprets the content of the register to be four decimal digits represented by a four-bit code, the content of the register is a four-digit decimal number. In the excess-3 code, the previous example is the decimal number 9096. The content of the register is meaningless in BCD since the bit combination 1100 is not assigned to any decimal digit. From this example, it is clear that a register can store one or more discrete elements of information and that the same bit configuration may be interpreted differently for different types of elements of information. It is important that the user store meaningful information in registers and that the computer be programmed to process this information according to the *type* of information stored.

Register Transfer

A digital computer is characterized by its registers. The memory unit (Fig. 1-1) is merely a collection of thousands of registers for storing digital information. The processor unit is composed of various registers that store operands upon which operations are performed. The control unit uses registers to keep track of various computer sequences, and every input or output device must have at least one register to store the information transferred to or from the device. An *interregister transfer* operation, a basic operation in digital systems, consists of a transfer of the information stored in one register into another. Figure 1-2 illustrates the transfer of information among registers and demonstrates pictorially the transfer of binary information from a keyboard into a register in the memory unit. The input unit is assumed to have a keyboard, a control circuit, and an input register. Each time a key is struck, the control enters into the input register an equivalent eight-bit alphanumeric character code. We shall assume that the code used is the ASCII code with an odd-parity eighth bit. The information from the input register is transferred into the eight least significant cells of a processor register. After every transfer, the input register is cleared to enable the control to insert a new eight-bit code when the keyboard is struck again. Each eight-bit character transferred to the processor register is preceded by a shift of the previous character to the next eight cells on its left. When a transfer of four characters is completed, the processor register is full, and its contents are transferred into a memory register. The content stored in the memory register shown in Fig. 1-2 came from the transfer of the characters JOHN after the four appropriate keys were struck.

To process discrete quantities of information in binary form, a computer must be provided with (1) devices that hold the data to be processed and (2) circuit elements that manipulate individual bits of information. The device most commonly used for holding data is a register. Manipulation of binary variables is done by means of digital logic circuits. Figure 1-3 illustrates the process of adding two 10-bit binary numbers. The memory unit, which normally consists of thousands of registers, is shown in the

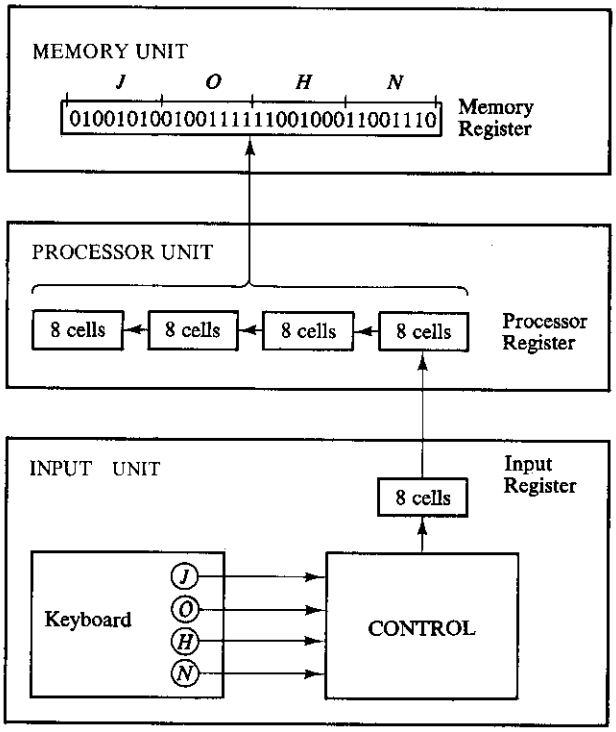


FIGURE 1-2
Transfer of information with registers

diagram with only three of its registers. The part of the processor unit shown consists of three registers, R1, R2, and R3, together with digital logic circuits that manipulate the bits of R1 and R2 and transfer into R3 a binary number equal to their arithmetic sum. Memory registers store information and are incapable of processing the two operands. However, the information stored in memory can be transferred to processor registers. Results obtained in processor registers can be transferred back into a memory register for storage until needed again. The diagram shows the contents of two operands transferred from two memory registers into R1 and R2. The digital logic circuits produce the sum, which is transferred to register R3. The contents of R3 can now be transferred back to one of the memory registers.

The last two examples demonstrated the information-flow capabilities of a digital system in a very simple manner. The registers of the system are the basic elements for storing and holding the binary information. The digital logic circuits process the information. Digital logic circuits and their manipulative capabilities are introduced in the next section. Registers and memory are presented in Chapter 7.

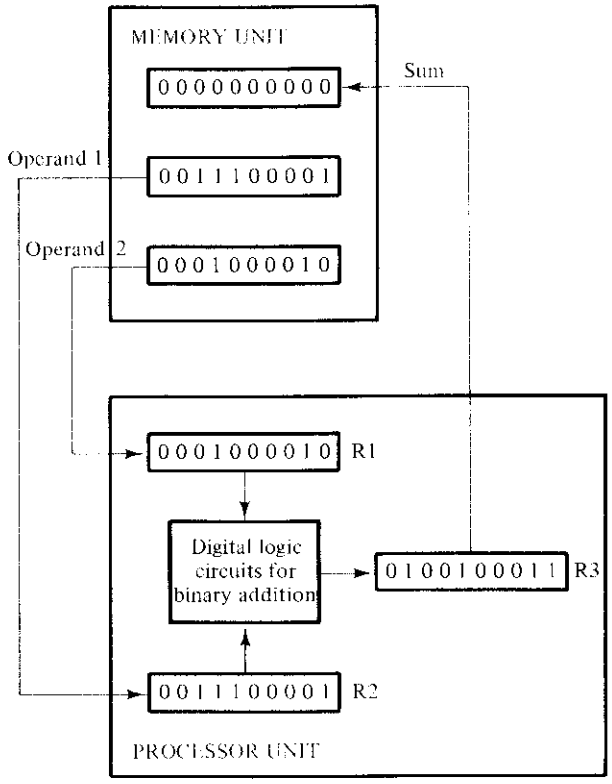


FIGURE 1-3
Example of binary information processing

1-9 BINARY LOGIC

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables take may be called by different names (e.g., *true* and *false*, *yes* and *no*, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values of 1 and 0. Binary logic is used to describe, in a mathematical way, the manipulation and processing of binary information. It is particularly suited for the analysis and design of digital systems. For example, the digital logic circuits of Fig. 1-3 that perform the binary arithmetic are circuits whose behavior is most conveniently expressed by means of binary variables and logical operations. The binary logic to be introduced in this section is equivalent to an algebra called Boolean algebra. The formal presentation of a two-valued Boolean algebra is covered in more detail in Chapter 2. The purpose of this section is to introduce Boolean algebra in a heuristic manner and relate it to digital logic circuits and binary signals.

Definition of Binary Logic

Binary logic consists of binary variables and logical operations. The variables are designated by letters of the alphabet such as *A, B, C, x, y, z,* etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read "x AND y is equal to z." The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ *and* $y = 1$; otherwise $z = 0$. (Remember that x, y, and z are binary variables and can be equal either to 1 or 0, and nothing else.)
2. OR: This operation is represented by a plus sign. For example, $x + y = z$ is read "x OR y is equal to z," meaning that $z = 1$ if $x = 1$ *or* if $y = 1$ *or* if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.
3. NOT: This operation is represented by a prime (sometimes by a bar). For example, $x' = z$ (or $\overline{x} = z$) is read "not x is equal to z," meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$; but if $x = 0$, then $z = 1$.

Binary logic resembles binary arithmetic, and the operations AND and OR have some similarities to multiplication and addition, respectively. In fact, the symbols used for AND and OR are the same as those used for multiplication and addition. However, binary logic should not be confused with binary arithmetic. One should realize that an arithmetic variable designates a number that may consist of many digits. A logic variable is always either a 1 or a 0. For example, in binary arithmetic, we have $1 + 1 = 10$ (read: "one plus one is equal to 2"), whereas in binary logic, we have $1 + 1 = 1$ (read: "one OR one is equal to one").

For each combination of the values of x and y, there is a value of z specified by the definition of the logical operation. These definitions may be listed in a compact form using *truth tables*. A truth table is a table of all possible combinations of the variables showing the relation between the values that the variables may take and the result of the operation. For example, the truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs. The result of the operation for each combination is then listed in a separate row. The truth tables for AND, OR, and NOT are listed in Table 1-6. These tables clearly demonstrate the definition of the operations.

Switching Circuits and Binary Signals

The use of binary variables and the application of binary logic are demonstrated by the simple switching circuits of Fig. 1-4. Let the manual switches A and B represent two binary variables with values equal to 0 when the switch is open and 1 when the switch is closed. Similarly, let the lamp L represent a third binary variable equal to 1 when the light is on and 0 when off. For the switches in series, the light turns on if A *and* B are

TABLE 1-6
Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

closed. For the switches in parallel, the light turns on if A *or* B is closed. It is obvious that the two circuits can be expressed by means of binary logic with the AND and OR operations, respectively:

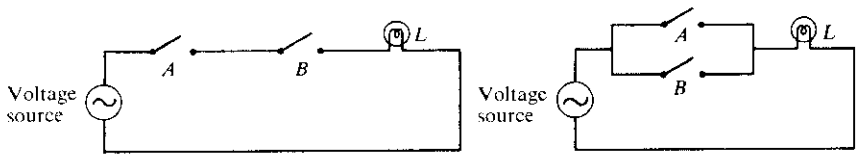
$$L = A \cdot B \qquad \text{for the circuit of Fig. 1-4(a)}$$

$$L = A + B \qquad \text{for the circuit of Fig. 1-4(b)}$$

Electronic digital circuits are sometimes called *switching circuits* because they behave like a switch, with the active element such as a transistor either conducting (switch closed) or not conducting (switch open). Instead of changing the switch manually, an electronic switching circuit uses binary signals to control the conduction or nonconduction state of the active element. Electrical signals such as voltages or currents exist throughout a digital system in either one of two recognizable values (except during transition). Voltage-operated circuits, for example, respond to two separate voltage levels, which represent a binary variable equal to logic-1 or logic-0. For example, a particular digital system may define logic-1 as a signal with a nominal value of 3 volts and logic-0 as a signal with a nominal value of 0 volt. As shown in Fig. 1-5, each voltage level has an acceptable deviation from the nominal. The intermediate region between the allowed regions is crossed only during state transitions. The input terminals of digital circuits accept binary signals within the allowable tolerances and respond at the output terminal with binary signals that fall within the specified tolerances.

Logic Gates

Electronic digital circuits are also called *logic circuits* because, with the proper input, they establish logical manipulation paths. Any desired information for computing or



(a) Switches in series – logic AND (b) Switches in parallel – logic OR

FIGURE 1-4
Switching circuits that demonstrate binary logic

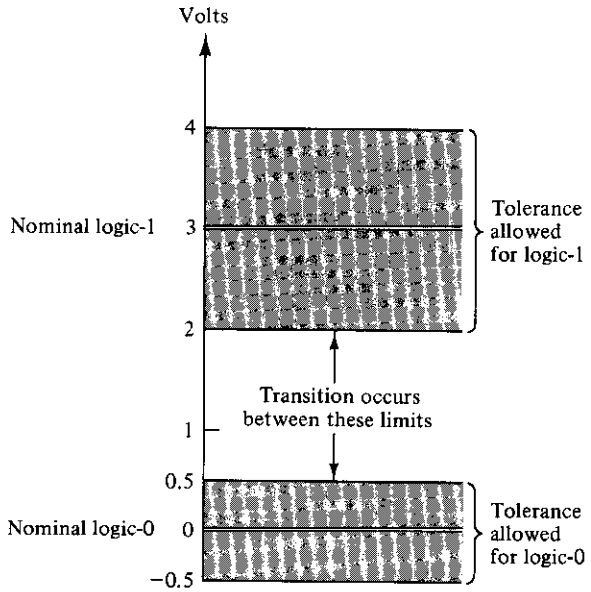
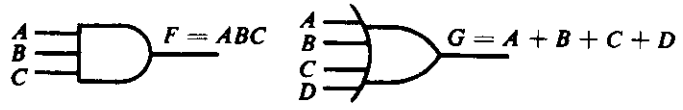


FIGURE 1-5
Example of binary signals

control can be operated upon by passing binary signals through various combinations of logic circuits, each signal representing a variable and carrying one bit of information. Logic circuits that perform the logical operations of AND, OR, and NOT are shown with their symbols in Fig. 1-6. These circuits, called *gates,* are blocks of hardware that produce a logic-1 or logic-0 output signal if input logic requirements are satisfied. Note that four different names have been used for the same type of circuits: digital circuits, switching circuits, logic circuits, and gates. All four names are widely used, but we shall refer to the circuits as AND, OR, and NOT gates. The NOT gate is sometimes called an *inverter circuit* since it inverts a binary signal.

The input signals x and y in the two-input gates of Fig. 1-6 may exist in one of four possible states: 00, 10, 11, or 01. These input signals are shown in Fig. 1-7, together with the output signals for the AND and OR gates. The timing diagrams in Fig. 1-7 il-

(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter



(d) Three-input AND gate (e) Four-input OR gate

FIGURE 1-6
Symbols for digital logic circuits

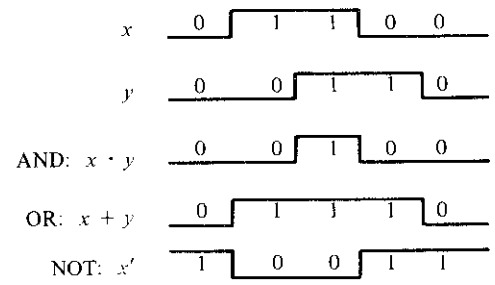


FIGURE 1-7
Input–output signals for gates (a), (b), and (c) of Fig. 1-6

lustrate the response of each circuit to each of the four possible input binary combinations. The reason for the name "inverter" for the NOT gate is apparent from a comparison of the signal x (input of inverter) and that of x' (output of inverter).

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with four inputs are shown in Fig. 1-6. The three-input AND gate responds with a logic-1 output if all three input signals are logic-1. The output produces a logic-0 signal if any input is logic-0. The four-input OR gate responds with a logic-1 when any input is a logic-1. Its output becomes logic-0 if all input signals are logic-0.

The mathematical system of binary logic is better known as Boolean, or switching, algebra. This algebra is conveniently used to describe the operation of complex networks of digital circuits. Designers of digital systems use Boolean algebra to transform circuit diagrams to algebraic expressions and vice versa. Chapters 2 and 3 are devoted to the study of Boolean algebra, its properties, and manipulative capabilities. Chapter 4 shows how Boolean algebra may be used to express mathematically the interconnections among networks of gates.

REFERENCES

1. CAVANAGH, J. J., *Digital Computer Arithmetic*, New York: McGraw-Hill, 1984.
2. HWANG, K., *Computer Arithmetic*, New York: John Wiley, 1979.
3. SCHMID, H., *Decimal Computation*, New York: John Wiley, 1974.
4. KNUTH, D. E., *The Art of Computer Programming: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1969.
5. FLORES, I., *The Logic of Computer Arithmetic*. Englewood Cliffs, NJ: Prentice-Hall, 1963.
6. RICHARD, R. K., *Arithmetic Operations in Digital Computers*. New York: Van Nostrand, 1955.
7. MANO, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
8. CHU, Y., *Computer Organization and Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

PROBLEMS

1-1 List the first 16 numbers in base 12. Use the letters A and B to represent the last two digits.

1-2 What is the largest binary number that can be obtained with 16 bits? What is its decimal equivalent?

1-3 Convert the following binary numbers to decimal: 101110; 1110101.11; and 110110100.

1-4 Convert the following numbers with the indicated bases to decimal: $(12121)_3$; $(4310)_5$; $(50)_7$; and $(198)_{12}$.

1-5 Convert the following decimal numbers to binary: 1231; 673.23; 10^4; and 1998.

1-6 Convert the following decimal numbers to the indicated bases:
(a) 7562.45 to octal.
(b) 1938.257 to hexadecimal.
(c) 175.175 to binary.

1-7 Convert the hexadecimal number F3A7C2 to binary and octal.

1-8 Convert the following numbers from the given base to the other three bases indicated.
(a) Decimal 225 to binary, octal, and hexadecimal.
(b) Binary 11010111 to decimal, octal, and hexadecimal.
(c) Octal 623 to decimal, binary, and hexadecimal.
(d) Hexadecimal 2AC5 to decimal, octal, and binary.

1-9 Add and multiply the following numbers without converting to decimal.
(a) $(367)_8$ and $(715)_8$.
(b) $(15F)_{16}$ and $(A7)_{16}$.
(c) $(110110)_2$ and $(110101)_2$.

1-10 Perform the following division in binary: 11111111/101.

1-11 Determine the value of base x if $(211)_x = (152)_8$.

1-12 Noting that $3^2 = 9$, formulate a simple procedure for converting base-3 numbers directly to base-9. Use the procedure to convert $(2110201102220112)_3$ to base 9.

1-13 Find the 9's complement of the following 8-digit decimal numbers: 12349876; 00980100; 90009951; and 00000000.

1-14 Find the 10's complement of the following 6-digit decimal numbers: 123900; 090657; 100000; and 000000.

1-15 Find the 1's and 2's complements of the following 8-digit binary numbers: 10101110; 10000001; 10000000; 00000001; and 00000000.

1-16 Perform subtraction with the following unsigned decimal numbers by taking the 10's complement of the subtrahend.
(a) 5250 − 1321
(b) 1753 − 8640
(c) 20 − 100
(d) 1200 − 250

1-17 Perform the subtraction with the following unsigned binary numbers by taking the 2's complement of the subtrahend.

(a) 11010 − 10000
(b) 11010 − 1101
(c) 100 − 110000
(d) 1010100 − 1010100

1-18 Perform the arithmetic operations (+42) + (−13) and (−42) − (−13) in binary using the signed-2's-complement representation for negative numbers.

1-19 The binary numbers listed have a sign in the leftmost position and, if negative, are in 2's-complement form. Perform the arithmetic operations indicated and verify the answers.
(a) 101011 + 111000
(b) 001110 + 110010
(c) 111001 − 001010
(d) 101011 − 100110

1-20 Represent the following decimal numbers in BCD: 13597; 93286; and 99880.

1-21 Determine the binary code for each of the ten decimal digits using a weighted code with weights 7, 4, 2, and 1.

1-22 The $(r - 1)$'s complement of base-6 numbers is called the 5's complement.
(a) Determine a procedure for obtaining the 5's complement of base-6 numbers.
(b) Obtain the 5's complement of $(543210)_6$.
(c) Design a 3-bit code to represent each of the six digits of the base-6 number system. Make the binary code self-complementing so that the 5's complement is obtained by changing 1's to 0's and 0's to 1's in all the bits of the coded number.

1-23 Represent decimal number 8620 in (a) BCD, (b) excess-3 code, (c)2421 code, and (d) as a binary number.

1-24 Represent decimal 3864 in the 2421 code of Table 1-2. Show that the code is self-complementing by taking the 9's complement of 3864.

1-25 Assign a binary code in some orderly manner to the 52 playing cards. Use the minimum number of bits.

1-26 List the ten BCD digits with an even parity in the leftmost position. (Total of five bits per digit.) Repeat with an odd-parity bit.

1-27 Write your full name in ASCII using an eight-bit code with the leftmost bit always 0. Include a space between names and a period after a middle initial.

1-28 Decode the following ASCII code: 1001010 1101111 1101000 1101110 0100000 1000100 1101111 1100101.

1-29 Show the bit configuration that represents the decimal number 295 (a) in binary, (b) in BCD, and (c) in ASCII.

1-30 How many printing characters are there in ASCII? How many of them are not letters or numerals?

1-31 The state of a 12-bit register is 010110010111. What is its content if it represents:
(a) three decimal digits in BCD;
(b) three decimal digits in the excess-3 code;
(c) three decimal digits in the 2421 code?

1-32 Show the contents of all registers in Fig. 1-3 if the two binary numbers added have the decimal equivalent of 257 and 514.

1-33 Show the signals (by means of diagram similar to Fig. 1-7) of the outputs F and G in the two gates of Figs. 1-6(d) and (e). Use all 16 possible combinations of the input signals A, B, C, and D.

1-34 Express the switching circuit shown in the figure in binary logic notation.

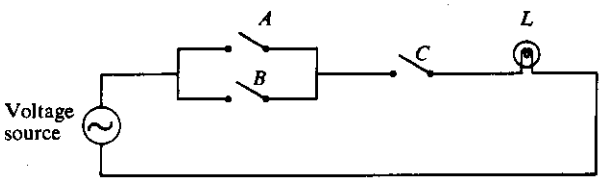


FIGURE P1-34

Boolean Algebra and Logic Gates

2-1 BASIC DEFINITIONS

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects having a common property. If S is a set, and x and y are certain objects, then $x \in S$ denotes that x is a member of the set S, and $y \notin S$ denotes that y is not an element of S. A set with a denumerable number of elements is specified by braces: $A = \{1, 2, 3, 4\}$, i.e., the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair of elements from S a unique element from S. As an example, consider the relation $a * b = c$. We say that $*$ is a binary operator if it specifies a rule for finding c from the pair (a, b) and also if $a, b, c \in S$. However, $*$ is not a binary operator if $a, b \in S$, whereas the rule finds $c \notin S$.

The postulates of a mathematical system form the basic assumptions from which it is possible to deduce the rules, theorems, and properties of the system. The most common postulates used to formulate various algebraic structures are:

1. *Closure.* A set S is closed with respect to a binary operator if, for every pair of elements of S, the binary opertor specifies a rule for obtaining a unique element of S. For example, the set of natural numbers $N = \{1, 2, 3, 4, \ldots\}$ is closed with respect to the binary operator plus $(+)$ by the rules of arithmetic addition, since for any $a, b \in N$ we obtain a unique $c \in N$ by the operation $a + b = c$. The set of natural numbers is not closed with respect to the binary operator minus $(-)$ by the rules of arithmetic subtraction because $2 - 3 = -1$ and $2, 3 \in N$, while $(-1) \notin N$.

2. *Associative law.* A binary operator $*$ on a set S is said to be associative whenever

$$(x * y) * z = x * (y * z) \qquad \text{for all } x, y, z, \in S$$

3. *Commutative law.* A binary operator $*$ on a set S is said to be commutative whenever

$$x * y = y * x \qquad \text{for all } x, y \in S$$

4. *Identity element.* A set S is said to have an identity element with respect to a binary operation $*$ on S if there exists an element $e \in S$ with the property

$$e * x = x * e = x \qquad \text{for every } x \in S$$

Example: The element 0 is an identity element with respect to operation $+$ on the set of integers $I = \{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \}$ since

$$x + 0 = 0 + x = x \qquad \text{for any } x \in I$$

The set of natural numbers N has no identity element since 0 is excluded from the set.

5. *Inverse.* A set S having the identity element e with respect to a binary operator $*$ is said to have an inverse whenever, for every $x \in S$, there exists an element $y \in S$ such that

$$x * y = e$$

Example: In the set of integers I with $e = 0$, the inverse of an element a is $(-a)$ since $a + (-a) = 0$.

6. *Distributive law.* If $*$ and $\cdot$ are two binary operators on a set S, $*$ is said to be distributive over $\cdot$ whenever

$$x * (y \cdot z) = (x * y) \cdot (x * z)$$

An example of an algebraic structure is a *field.* A field is a set of elements, together with two binary operators, each having properties 1 to 5 and both operators combined to give property 6. The set of real numbers together with the binary operators $+$ and $\cdot$ form the field of real numbers. The field of real numbers is the basis for arithmetic and ordinary algebra. The operators and postulates have the following meanings:

The binary operator $+$ defines addition.
The additive identity is 0.
The additive inverse defines subtraction.
The binary operator $\cdot$ defines multiplication.
The multiplicative identity is 1.
The multiplicative inverse of $a = 1/a$ defines division, i.e., $a \cdot 1/a = 1$.
The only distributive law applicable is that of $\cdot$ over $+$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

2-2 AXIOMATIC DEFINITION OF BOOLEAN ALGEBRA

In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system now called *Boolean algebra*. In 1938 C. E. Shannon introduced a two-valued Boolean algebra called *switching algebra*, in which he demonstrated that the properties of bistable electrical switching circuits can be represented by this algebra. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

Boolean algebra is an algebraic structure defined on a set of elements B together with two binary operators + and $\cdot$ provided the following (Huntington) postulates are satisfied:

1. (**a**) Closure with respect to the operator +.
   (**b**) Closure with respect to the operator $\cdot$.
2. (**a**) An identity element with respect to +, designated by 0: $x + 0 = 0 + x = x$.
   (**b**) An identity element with respect to $\cdot$, designated by 1: $x \cdot 1 = 1 \cdot x = x$.
3. (**a**) Commutative with respect to +: $x + y = y + x$.
   (**b**) Commutative with respect to $\cdot$: $x \cdot y = y \cdot x$.
4. (**a**) $\cdot$ is distributive over +: $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.
   (**b**) + is distributive over $\cdot$: $x + (y \cdot z) = (x + y) \cdot (x + z)$.
5. For every element $x \in B$, there exists an element $x' \in B$ (called the complement of x) such that (a) $x + x' = 1$ and (b) $x \cdot x' = 0$.
6. There exists at least two elements $x, y \in B$ such that $x \neq y$.

Comparing Boolean algebra with arithmetic and ordinary algebra (the field of real numbers), we note the following differences:

1. Huntington postulates do not include the associative law. However, this law holds for Boolean algebra and can be derived (for both operators) from the other postulates.
2. The distributive law of + over $\cdot$, i.e., $x+(y \cdot z) = (x + y) \cdot (x + z)$, is valid for Boolean algebra, but not for ordinary algebra.
3. Boolean algebra does not have additive or multiplicative inverses; therefore, there are no subtraction or division operations.
4. Postulate 5 defines an operator called *complement* that is not available in ordinary algebra.
5. Ordinary algebra deals with the real numbers, which consitute an infinite set of elements. Boolean algebra deals with the as yet undefined set of elements B, but in the two-valued Boolean algebra defined below (and of interest in our subsequent use of this algebra), B is defined as a set with only two elements, 0 and 1.

Boolean algebra resembles ordinary algebra in some respects. The choice of symbols + and $\cdot$ is intentional to facilitate Boolean algebraic manipulations by persons already familiar with ordinary algebra. Although one can use some knowledge from

ordinary algebra to deal with Boolean algebra, the beginner must be careful not to substitute the rules of ordinary algebra where they are not applicable.

It is important to distinguish between the elements of the set of an algebraic structure and the variables of an algebraic system. For example, the elements of the field of real numbers are numbers, whereas variables such as a, b, c, etc., used in ordinary algebra, are symbols that stand for real numbers. Similarly in Boolean algebra, one defines the elements of the set B, and variables such as x, y, z are merely symbols that represent the elements. At this point, it is important to realize that in order to have a Boolean algebra, one must show:

1. the elements of the set B,
2. the rules of operation for the two binary operators, and
3. that the set of elements B, together with the two operators, satisfies the six Huntington postulates.

One can formulate many Boolean algebras, depending on the choice of elements of B and the rules of operation. In our subsequent work, we deal only with a two-valued Boolean algebra, i.e., one with only two elements. Two-valued Boolean algebra has applications in set theory (the algebra of classes) and in propositional logic. Our interest here is with the application of Boolean algebra to gate-type circuits.

Two-Valued Boolean Algebra

A two-valued Boolean algebra is defined on a set of two elements, $B = \{0, 1\}$, with rules for the two binary operators + and · as shown in the following operator tables (the rule for the complement operator is for verification of postulate 5):

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

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

x	x'
0	1
1	0

These rules are exactly the same as the AND, OR, and NOT operations, respectively, defined in Table 1-6. We must now show that the Huntington postulates are valid for the set $B = \{0, 1\}$ and the two binary operators defined before.

1. *Closure* is obvious from the tables since the result of each operation is either 1 or 0 and $1, 0 \in B$.
2. From the tables we see that
   (a) $0 + 0 = 0 \qquad 0 + 1 = 1 + 0 = 1$
   (b) $1 \cdot 1 = 1 \qquad 1 \cdot 0 = 0 \cdot 1 = 0$
   which establishes the two *identity elements* 0 for + and 1 for · as defined by postulate 2.

3. The *commutative* laws are obvious from the symmetry of the binary operator tables.
4. **(a)** The *distributive* law $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ can be shown to hold true from the operator tables by forming a truth table of all possible values of x, y, and z. For each combination, we derive $x \cdot (y + z)$ and show that the value is the same as $(x \cdot y) + (x \cdot z)$.

x	y	z	$y + z$	$x \cdot (y + z)$	$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

   (b) The *distributive* law of + over · can be shown to hold true by means of a truth table similar to the one above.
5. From the complement table it is easily shown that
   (a) $x + x' = 1$, since $0 + 0' = 0 + 1 = 1$ and $1 + 1' = 1 + 0 = 1$.
   (b) $x \cdot x' = 0$, since $0 \cdot 0' = 0 \cdot 1 = 0$ and $1 \cdot 1' = 1 \cdot 0 = 0$, which verifies postulate 5.
6. Postulate 6 is satisfied because the two-valued Boolean algebra has two distinct elements, 1 and 0, with $1 \neq 0$.

We have just established a two-valued Boolean algebra having a set of two elements, 1 and 0, two binary operators with operation rules equivalent to the AND and OR operations, and a complement operator equivalent to the NOT operator. Thus, Boolean algebra has been defined in a formal mathematical manner and has been shown to be equivalent to the binary logic presented heuristically in Section 1-9. The heuristic presentation is helpful in understanding the application of Boolean algebra to gate-type circuits. The formal presentation is necessary for developing the theorems and properties of the algebraic system. The two-valued Boolean algebra defined in this section is also called "switching algebra" by engineers. To emphasize the similarities between two-valued Boolean algebra and other binary systems, this algebra was called "binary logic" in Section 1-9. From here on, we shall drop the adjective "two-valued" from Boolean algebra in subsequent discussions.

2-3 BASIC THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA

Duality

The Huntington postulates have been listed in pairs and designated by part (a) and part (b). One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the *duality principle*. It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, the identity elements and the elements of the set B are the same: 1 and 0. The duality principle has many applications. If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems

Table 2-1 lists six theorems of Boolean algebra and four of its postulates. The notation is simplified by omitting the · whenever this does not lead to confusion. The theorems and postulates listed are the most basic relationships in Boolean algebra. The reader is advised to become familiar with them as soon as possible. The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. The proofs of the theorems with one variable are presented below. At the right is listed the number of the postulate that justifies each step of the proof.

TABLE 2-1
Postulates and Theorems of Boolean Algebra

Postulate 2	(a) $x + 0 = x$	(b) $x \cdot 1 = x$
Postulate 5	(a) $x + x' = 1$	(b) $x \cdot x' = 0$
Theorem 1	(a) $x + x = x$	(b) $x \cdot x = x$
Theorem 2	(a) $x + 1 = 1$	(b) $x \cdot 0 = 0$
Theorem 3, involution	$(x')' = x$	
Postulate 3, commutative	(a) $x + y = y + x$	(b) $xy = yx$
Theorem 4, associative	(a) $x + (y + z) = (x + y) + z$	(b) $x(yz) = (xy)z$
Postulate 4, distributive	(a) $x(y + z) = xy + xz$	(b) $x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a) $(x + y)' = x'y'$	(b) $(xy)' = x' + y'$
Theorem 6, absorption	(a) $x + xy = x$	(b) $x(x + y) = x$

THEOREM 1(a): $x + x = x$.

$x + x = (x + x) \cdot 1$	by postulate:	2(b)
$= (x + x)(x + x')$		5(a)
$= x + xx'$		4(b)
$= x + 0$		5(b)
$= x$		2(a)

THEOREM 1(b): $x \cdot x = x$.

$x \cdot x = xx + 0$	by postulate:	2(a)
$= xx + xx'$		5(b)
$= x(x + x')$		4(a)
$= x \cdot 1$		5(a)
$= x$		2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of part (a). Any dual theorem can be similarly derived from the proof of its corresponding pair.

THEOREM 2(a): $x + 1 = 1$.

$x + 1 = 1 \cdot (x + 1)$	by postulate:	2(b)
$= (x + x')(x + 1)$		5(a)
$= x + x' \cdot 1$		4(b)
$= x + x'$		2(b)
$= 1$		5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x \cdot x' = 0$, which defines the complement of x. The complement of x' is x and is also $(x')'$. Therefore, since the complement is unique, we have that $(x')' = x$.

The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven. Take, for example, the absorption theorem.

THEOREM 6(a): $x + xy = x$.

$$
\begin{aligned}
x + xy &= x \cdot 1 + xy && \text{by postulate: } 2(b)\\
&= x(1 + y) && 4(a)\\
&= x(y + 1) && 3(a)\\
&= x \cdot 1 && 2(a)\\
&= x && 2(b)
\end{aligned}
$$

THEOREM 6(b): $x(x + y) = x$ by duality.

The theorems of Boolean algebra can be shown to hold true by means of truth tables. In truth tables, both sides of the relation are checked to yield identical results for all possible combinations of variables involved. The following truth table verifies the first absorption theorem.

			=
x	y	xy	$x + xy$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

The algebraic proofs of the associative law and DeMorgan's theorem are long and will not be shown here. However, their validity is easily shown with truth tables. For example, the truth table for the first DeMorgan's theorem $(x + y)' = x'y'$ is shown below.

x	y	$x + y$	$(x + y)'$	x'	y'	$x'y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Operator Precedence

The operator precedence for evaluating Boolean expressions is (1) parentheses, (2) NOT, (3) AND, and (4) OR. In other words, the expression inside the parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, then follows the AND, and finally the OR. As an example, consider

the truth table for DeMorgan's theorem. The left side of the expression is $(x + y)'$. Therefore, the expression inside the parentheses is evaluated first and the result then complemented. The right side of the expression is $x'y'$. Therefore, the complement of x and the complement of y are both evaluated first and the result is then ANDed. Note that in ordinary arithmetic, the same precedence holds (except for the complement) when multiplication and addition are replaced by AND and OR, respectively.

Venn Diagram

A helpful illustration that may be used to visualize the relationships among the variables of a Boolean expression is the *Venn diagram*. This diagram consists of a rectangle such as shown in Fig. 2-1, inside of which are drawn overlapping circles, one for each variable. Each circle is labeled by a variable. We designate all points inside a circle as belonging to the named variable and all points outside a circle as not belonging to the variable. Take, for example, the circle labeled x. If we are inside the circle, we say that $x = 1$; when outside, we say $x = 0$. Now, with two overlapping circles, there are four distinct areas inside the rectangle: the area not belonging to either x or y $(x'y')$, the area inside circle y but outside x $(x'y)$, the area inside circle x but outside y (xy'), and the area inside both circles (xy).

Venn diagrams may be used to illustrate the postulates of Boolean algebra or to show the validity of theorems. Figure 2-2, for example, illustrates that the area belonging to xy is inside the circle x and therefore $x + xy = x$. Figure 2-3 illustrates the distributive law $x(y + z) = xy + xz$. In this diagram, we have three overlapping circles, one for each of the variables x, y, and z. It is possible to distinguish eight distinct areas in a three-variable Venn diagram. For this particular example, the distributive law is demonstrated by noting that the area intersecting the circle x with the area enclosing y or z is the same area belonging to xy or xz.

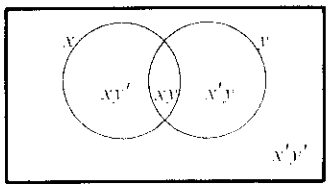


FIGURE 2-1
Venn diagram for two variables

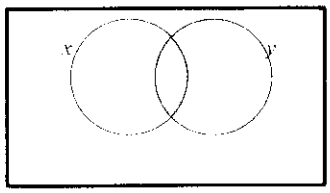
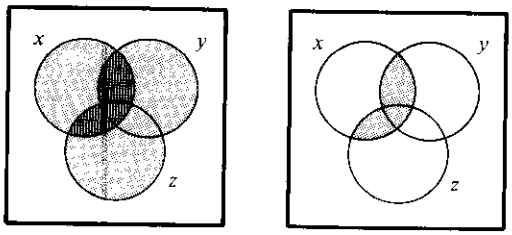


FIGURE 2-2
Venn diagram illustration $x = xy + x$



$x(y + z)$ $xy + xz$

FIGURE 2-3
Venn diagram illustration of the distributive law

2-4 BOOLEAN FUNCTIONS

A binary variable can take the value of 0 or 1. A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, and unary operator NOT, parentheses, and an equal sign. For a given value of the variables, the function can be either 0 or 1. Consider, for example, the Boolean function

$$F_1 = xyz'$$

The function F_1 is equal to 1 if $x = 1$ *and* $y = 1$ *and* $z' = 1$; otherwise $F_1 = 0$. The above is an example of a Boolean function represented as an algebraic expression. A Boolean function may also be represented in a truth table. To represent a function in a truth table, we need a list of the 2^n combinations of 1's and 0's of the n binary variables, and a column showing the combinations for which the function is equal to 1 or 0. As shown in Table 2-2, there are eight possible distinct combinations for assigning bits to three variables. The column labeled F_1 contains either a 0 or a 1 for each of these combinations. The table shows that the function F_1 is equal to 1 only when $x = 1$, $y = 1$, and $z = 0$. It is equal to 0 otherwise. (Note that the statement $z' = 1$ is equivalent to saying that $z = 0$.) Consider now the function

TABLE 2-2
Truth Tables for $F_1 = xyz'$, $F_2 = x + y'z$, $F_3 = x'y'z + x'yz + xy'$, and $F_4 = xy' + x'z$

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

$$F_2 = x + y'z$$

$F_2 = 1$ if $x = 1$ or if $y = 0$, while $z = 1$. In Table 2-2, $x = 1$ in the last four rows and $yz = 01$ in rows 001 and 101. The latter combination applies also for $x = 1$. Therefore, there are five combinations that make $F_2 = 1$. As a third example, consider the function

$$F_3 = x'y'z + x'yz + xy'$$

This is shown in Table 2-2 with four 1's and four 0's. F_4 is the same as F_3 and is considered below.

Any Boolean function can be represented in a truth table. The number of rows in the table is 2^n, where n is the number of binary variables in the function. The 1's and 0's combinations for each row is easily obtained from the binary numbers by counting from 0 to $2^n - 1$. For each row of the table, there is a value for the function equal to either 1 or 0. The question now arises, is it possible to find two algebraic expressions that specify the same function? The answer to this question is yes. As a matter of fact, the manipulation of Boolean algebra is applied mostly to the problem of finding simpler expressions for the same function. Consider, for example, the function:

$$F_4 = xy' + x'z$$

From Table 2-2, we find that F_4 is the same as F_3, since both have identical 1's and 0's for each combination of values of the three binary variables. In general, two functions of n binary variables are said to be equal if they have the same value for all possible 2^n combinations of the n variables.

A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR, and NOT gates. The implementation of the four functions introduced in the previous discussion is shown in Fig. 2-4. The logic diagram includes an inverter circuit for every variable present in its complement form. (The inverter is unnecessary if the complement of the variable is available.) There is an AND gate for each term in the expression, and an OR gate is used to combine two or more terms. From the diagrams, it is obvious that the implementation of F_4 requires fewer gates and fewer inputs than F_3. Since F_4 and F_3 are equal Boolean functions, it is more economical to implement the F_4 form than the F_3 form. To find simpler circuits, one must know how to manipulate Boolean functions to obtain equal and simpler expressions. What constitutes the best form of a Boolean function depends on the particular application. In this section, consideration is given to the criterion of equipment minimization.

Algebraic Manipulation

A *literal* is a primed or unprimed variable. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate, and each term is implemented with a gate. The minimization of the number of literals and the number of terms results in a circuit with less equipment. It is not always possible to minimize both simultaneously; usually, further criteria must be available. At the moment, we

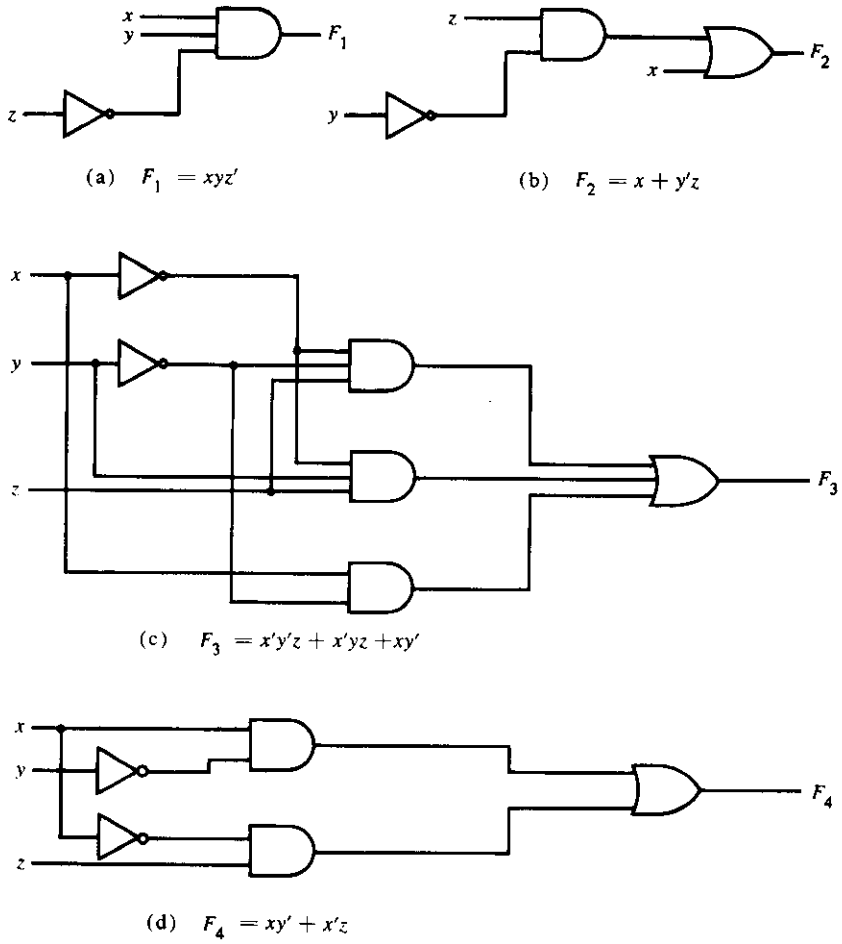


FIGURE 2-4
Implementation of Boolean functions with gates

shall narrow the minimization criterion to literal minimization. We shall discuss other criteria in Chapter 5. The number of literals in a Boolean function can be minimized by algebraic manipulations. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only method available is a cut-and-try procedure employing the postulates, the basic theorems, and any other manipulation method that becomes familiar with use. The following examples illustrate this procedure.

Example 2-1

Simplify the following Boolean functions to a minimum number of literals.

1. $x + x'y = (x + x')(x + y) = 1 \cdot (x + y) = x + y$
2. $x(x' + y) = xx' + xy = 0 + xy = xy$

3. $x'y'z + x'yz + xy' = x'z(y' + y) + xy' = x'z + xy'$
4. $xy + x'z + yz = xy + x'z + yz(x + x')$
$= xy + x'z + xyz + x'yz$
$= xy(1 + z) + x'z(1 + y)$
$= xy + x'z$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$ by duality from function 4. ■

Functions 1 and 2 are the duals of each other and use dual expressions in corresponding steps. Function 3 shows the equality of the functions F_3 and F_4 discussed previously. The fourth illustrates the fact that an increase in the number of literals sometimes leads to a final simpler expression. Function 5 is not minimized directly but can be derived from the dual of the steps used to derive function 4.

Complement of a Function

The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F. The complement of a function may be derived algebraically through DeMorgan's theorem. This pair of theorems is listed in Table 2-1 for two variables. DeMorgan's theorems can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived below. The postulates and theorems are those listed in Table 2-1.

$$\begin{aligned}
(A + B + C)' &= (A + X)' && \text{let } B + C = X \\
&= A'X' && \text{by theorem 5(a) (DeMorgan)} \\
&= A' \cdot (B + C)' && \text{substitute } B + C = X \\
&= A' \cdot (B'C') && \text{by theorem 5(a) (DeMorgan)} \\
&= A'B'C' && \text{by theorem 4(b) (associative)}
\end{aligned}$$

DeMorgan's theorems for any number of variables resemble in form the two variable case and can be derived by successive substitutions similar to the method used in the above derivation. These theorems can be generalized as follows:

$$(A + B + C + D + \cdots + F)' = A'B'C'D' \cdots F'$$
$$(ABCD \cdots F)' = A' + B' + C' + D' + \cdots + F'$$

The generalized form ofDeMorgan's theorem states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

Example 2-2 Find the complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorem as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$
$$F_2' = [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')' \cdot (yz)'$$
$$= x' + (y + z)(y' + z')$$ ■

A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal. This method follows from the generalized DeMorgan's theorem. Remember that the dual of a function is obtained from the interchange of AND and OR operators and 1's and 0's.

Example 2-3

Find the complement of the functions F_1 and F_2 of Example 2-2 by taking their duals and complementing each literal.

1. $F_1 = x'yz' + x'y'z$.
   The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
   Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
2. $F_2 = x(y'z' + yz)$.
   The dual of F_2 is $x + (y' + z')(y + z)$.
   Complement each literal: $x' + (y + z)(y' + z') = F_2'$. ■

2-5 CANONICAL AND STANDARD FORMS

Minterms and Maxterms

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now consider two binary variabes x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy', and xy. Each of these four AND terms represents one of the distinct areas in the Venn diagram of Fig. 2-1 and is called a *minterm*, or a *standard product*. In a similar manner, n variables can be combined to form 2^n minterms. The 2^n different minterms may be determined by a method similar to the one shown in Table 2-3 for three variables. The binary numbers from 0 to $2^n - 1$ are listed under the n variables. Each minterm is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1. A symbol for each minterm is also shown in the table and is of the form m_j, where j denotes the decimal equivalent of the binary number of the minterm designated.

In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called *maxterms*, or *standard sums*. The eight maxterms for three variables, together with their symbolic designation, are listed in Table 2-3. Any 2^n maxterms for n variables may be determined similarly. Each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1. Note that each maxterm is the complement of its corresponding minterm, and vice versa.

TABLE 2-3
Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function, and then taking the OR of all those terms. For example, the function f_1 in Table 2-4 is determined by expressing the combinations 001, 100, and 111 as $x'y'z$, $xy'z'$, and xyz, respectively. Since each one of these minterms results in $f_1 = 1$, we should have

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$

Similarly, it may be easily verified that

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

These examples demonstrate an important property of Boolean algebra: Any Boolean function can be expressed as a sum of minterms (by "sum" is meant the ORing of terms).

TABLE 2-4
Functions of Three Variables

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Now consider the complement of a Boolean function. It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms. The complement of f_1 is read as

$$f_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

If we take the complement of f_1', we obtain the function f_1:

$$\begin{aligned} f_1 &= (x + y + z)(x + y' + z)(x + y' + z')(x' + y + z')(x' + y' + z) \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \end{aligned}$$

Similarly, it is possible to read the expression for f_2 from the table:

$$\begin{aligned} f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z) \\ &= M_0M_1M_2M_4 \end{aligned}$$

These examples demonstrate a second important property of Boolean algebra: Any Boolean function can be expressed as a product of maxterms (by "product" is meant the ANDing of terms). The procedure for obtaining the product of maxterms directly from the truth table is as follows. Form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms. Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form*.

Sum of Minterms

It was previously stated that for n binary variables, one can obtain 2^n distinct minterms, and that any Boolean function can be expressed as a sum of minterms. The minterms whose sum defines the Boolean function are those that give the 1's of the function in a truth table. Since the function can be either 1 or 0 for each minterm, and since there are 2^n minterms, one can calculate the possible functions that can be formed with n variables to be 2^{2^n}. It is sometimes convenient to express the Boolean function in its sum of minterms form. If not in this form, it can be made so by first expanding the expression into a sum of AND terms. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables. The following examples clarifies this procedure.

Example 2-4 Express the Boolean function $F = A + B'C$ in a sum of minterms. The function has three variables, A, B, and C. The first term A is missing two variables; therefore:

$$A = A(B + B') = AB + AB'$$

This is still missing one variable:

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

The second term $B'C$ is missing one variable:

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

Combining all terms, we have

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

But $AB'C$ appears twice, and according to theorem 1 ($x + x = x$), it is possible to remove one of them. Rearranging the minterms in ascending order, we finally obtain

$$F = A'B'C + AB'C' + AB'C + ABC' + ABC$$
$$= m_1 + m_4 + m_5 + m_6 + m_7$$ ■

It is sometimes convenient to express the Boolean function, when in its sum of minterms, in the following short notation:

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

The summation symbol Σ stands for the ORing of terms; the numbers following it are the minterms of the function. The letters in parentheses following F form a list of the variables in the order taken when the minterm is converted to an AND term.

An alternate procedure for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table. Consider the Boolean function given in Example 2-4:

$$F = A + B'C$$

The truth table shown in Table 2-5 can be derived directly from the algebraic expression by listing the eight binary combinations under variables A, B, and C and inserting

TABLE 2-5
Truth Table for $F = A + B'C$

A	B	C	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

1's under F for those combinations where $A = 1$, and $BC = 01$. From the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Product of Maxterms

Each of the 2^{2^n} functions of n binary variables can be also expressed as a product of maxterms. To express the Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law, $x + yz = (x + y)(x + z)$. Then any missing variable x in each OR term is ORed with xx'. This procedure is clarified by the following example.

Example 2-5 Express the Boolean function $F = xy + x'z$ in a product of maxterm form. First, convert the function into OR terms using the distributive law:

$$\begin{aligned} F &= xy + x'z = (xy + x')(xy + z) \\ &= (x + x')(y + x')(x + z)(y + z) \\ &= (x' + y)(x + z)(y + z) \end{aligned}$$

The function has three variables: x, y, and z. Each OR term is missing one variable; therefore:

$$\begin{aligned} x' + y &= x' + y + zz' = (x' + y + z)(x' + y + z') \\ x + z &= x + z + yy' = (x + y + z)(x + y' + z) \\ y + z &= y + z + xx' = (x + y + z)(x' + y + z) \end{aligned}$$

Combining all the terms and removing those that appear more than once, we finally obtain:

$$\begin{aligned} F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\ &= M_0M_2M_4M_5 \end{aligned}$$

■

A convenient way to express this function is as follows:

$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

The product symbol, Π, denotes the ANDing of maxterms; the numbers are the maxterms of the function.

Conversion between Canonical Forms

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function. This is because the original function is expressed by those minterms that make the function equal to 1, whereas its complement is a 1 for those minterms that the function is a 0. As an example, consider the function

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

This has a complement that can be expressed as

$$F'(A, B, C) = \Sigma(0, 2, 3) = m_0 + m_2 + m_3$$

Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:

$$F = (m_0 + m_2 + m_3)' = m_0' \cdot m_2' \cdot m_3' = M_0 M_2 M_3 = \Pi(0, 2, 3)$$

The last conversion follows from the definition of minterms and maxterms as shown in Table 2-3. From the table, it is clear that the following relation holds true:

$$m_j' = M_j$$

That is, the maxterm with subscript j is a complement of the minterm with the same subscript j, and vice versa.

The last example demonstrates the conversion between a function expressed in sum of minterms and its equivalent in product of maxterms. A similar argument will show that the conversion between the product of maxterms and the sum of minterms is similar. We now state a general conversion procedure. To convert from one canonical form to another, interchange the symbols Σ and Π and list those numbers missing from the original form. In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2^n, where n is the number of binary variables in the function.

A Boolean function can be converted from an algebraic expression to a product of maxterms by using a truth table and the canonical conversion procedure. Consider, for example, the Boolean expression

$$F = xy + x'z$$

First, we derive the truth table of the function, as shown in Table 2-6. The 1's under F in the table are determined from the combination of the variable where $xy = 11$ and

TABLE 2-6
Truth Table for $F = xy + x'z$

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

$xz = 01$. The minterms of the function are read from the truth table to be 1, 3, 6, and 7. The function expressed in sum of minterms is

$$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

Since there are a total of eight minterms or maxterms in a function of three variable, we determine the missing terms to be 0, 2, 4, and 5. The function expressed in product of maxterm is

$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

This is the same answer obtained in Example 2-5.

Standard Forms

The two canonical forms of Boolean algebra are basic forms that one obtains from reading a function from the truth table. These forms are very seldom the ones with the least number of literals, because each minterm or maxterm must contain, by definition, *all* the variables either complemented or uncomplemented.

Another way to express Boolean functions is in *standard* form. In this configuration, the terms that form the function may contain one, two, or any number of literals. There are two types of standard forms: the sum of products and product of sums.

The *sum of products* is a Boolean expression containing AND terms, called *product terms*, of one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed in sum of products is

$$F_1 = y' + xy + x'yz'$$

The expression has three product terms of one, two, and three literals each, respectively. Their sum is in effect an OR operation.

A *product of sums* is a Boolean expression containing OR terms, called *sum terms*. Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed in product of sums is

$$F_2 = x(y' + z)(x' + y + z' + w)$$

This expression has three sum terms of one, two, and four literals each, respectively. The product is an AND operation. The use of the words *product* and *sum* stems from the smiilarity of the AND operation to the arithmetic product (multiplication) and the similarity of the OR operation to the arithmetic sum (addition).

A Boolean function may be expressed in a nonstandard form. For example, the function

$$F_3 = (AB + CD)(A'B' + C'D')$$

is neither in sum of products nor in product of sums. It can be changed to a standard form by using the distributive law to remove the parentheses:

$$F_3 = A'B'CD + ABC'D'$$

2-6 OTHER LOGIC OPERATIONS

When the binary operators AND and OR are placed between two variables, x and y, they form two Boolean functions, $x \cdot y$ and $x + y$, respectively. It was stated previously that there are 2^{2^n} functions for n binary variables. For two variables, $n = 2$, and the number of possible Boolean functions is 16. Therefore, the AND and OR functions are only two of a total of 16 possible functions formed with two binary variables. It would be instructive to find the other 14 functions and investigate their properties.

The truth tables for the 16 functions formed with two binary variables, x and y, are listed in Table 2-7. In this table, each of the 16 columns, F_0 to F_{15}, represents a truth table of one possible function for the two given variables, x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F. Some of the functions are shown with an operator symbol. For example, F_1 represents the truth table for AND and F_7 represents the truth table for OR. The operator symbols for these functions are $\cdot$ and $+$, respectively.

TABLE 2-7
Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Operator symbol			.	/		/		$\oplus$	+	$\downarrow$	$\odot$	′	$\subset$	′	$\supset$	$\uparrow$	

The 16 functions listed in truth table form can be expressed algebraically by means of Boolean expressions. This is shown in the first column of Table 2-8. The Boolean expressions listed are simplified to their minimum number of literals.

Although each function can be expressed in terms of the Boolean operators AND, OR, and NOT, there is no reason one cannot assign special operator symbols for expressing the other functions. Such operator symbols are listed in the second column of Table 2-8. However, all the new symbols shown, except for the exclusive-OR symbol, $\oplus$, are not in common use by digital designers.

Each of the functions in Table 2-8 is listed with an accompanying name and a comment that explains the function in some way. The 16 functions listed can be subdivided into three categories:

1. Two functions that produce a constant 0 or 1.
2. Four functions with unary operations: complement and transfer.
3. Ten functions with binary operators that define eight different operations: AND, OR, NAND, NOR, exclusive-OR, equivalence, inhibition, and implication.

TABLE 2-8
Boolean Expressions for the 16 Functions of Two Variables

Boolean functions	Operator symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$x \odot y$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Any function can be equal to a constant, but a binary function can be equal to only 1 or 0. The complement function produces the complement of each of the binary variables. A function that is equal to an input variable has been given the name *transfer*, because the variable x or y is transferred through the gate that forms the function without changing its value. Of the eight binary operators, two (inhibition and implication) are used by logicians but are seldom used in computer logic. The AND and OR operators have been mentioned in conjunction with Boolean algebra. The other four functions are extensively used in the design of digital systems.

The NOR function is the complement of the OR function and its name is an abbreviation of *not-OR*. Similarly, NAND is the complement of AND and is an abbreviation of *not-AND*. The exclusive-OR, abbreviated XOR or EOR, is similar to OR but excludes the combination of *both* x and y being equal to 1. The equivalence is a function that is 1 when the two binary variables are equal, i.e., when both are 0 or both are 1. The exclusive-OR and equivalence functions are the complements of each other. This can be easily verified by inspecting Table 2-7. The truth table for the exclusive-OR is F_6 and for the equivalence is F_9, and these two functions are the complements of each other. For this reason, the equivalence function is often called exclusive-NOR, i.e., exclusive-OR-NOT.

Boolean algebra, as defined in Section 2-2, has two binary operators, which we have called AND and OR, and a unary operator, NOT (complement). From the definitions,

we have deduced a number of properties of these operators and now have defined other binary operators in terms of them. There is nothing unique about this procedure. We could have just as well started with the operator NOR ($\downarrow$), for example, and later defined AND, OR, and NOT in terms of it. There are, nevertheless, good reasons for introducing Boolean algebra in the way it has been introduced. The concepts of "and," "or," and "not" are familiar and are used by people to express everyday logical ideas. Moreover, the Huntington postulates reflect the dual nature of the algebra, emphasizing the symmetry of + and · with respect to each other.

2-7 DIGITAL LOGIC GATES

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these types of gates. The possibility of constructing gates for the other logic operations is of practial interest. Factors to be weighed when considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table 2-8, two are equal to a constant and four others are repeated twice. There are only ten functions left to be considered as candidates for logic gates. Two, inhibition and implication, are not commutative or associative and thus are impractical to use as standard logic gates. The other eight: complement, transfer, AND, OR, NAND, NOR, exclusive-OR, and equivalence, are used as standard gates in digital design.

The graphic symbols and truth tables of the eight gates are shown in Fig. 2-5. Each gate has one or two binary input variables designated by x and y and one binary output variable designated by F. The AND, OR, and inverter circuits were defined in Fig. 1-6. The inverter circuit inverts the logic sense of a binary variable. It produces the NOT, or complement, function. The small circle in the output of the graphic symbol of an inverter designates the logic complement. The triangle symbol by itself designates a buffer circuit. A buffer produces the *transfer* function but does not produce any particular logic operation, since the binary value of the output is equal to the binary value of the input. This circuit is used merely for power amplification of the signal and is equivalent to two inverters connected in cascade.

The NAND function is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle. The NOR function is the complement of the OR function and uses an OR graphic symbol followed by a small circle. The NAND and NOR gates are extensively used as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because Boolean functions can be easily implemented with them.

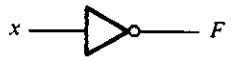
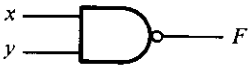
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = xy$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$
OR		$F = x + y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$
Inverter		$F = x'$	$x \mid F$ $0 \mid 1$ $1 \mid 0$
Buffer		$F = x$	$x \mid F$ $0 \mid 0$ $1 \mid 1$
NAND		$F = (xy)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
NOR		$F = (x + y)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= x \odot y$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$

FIGURE 2-5
Digital logic gates

The exclusive-OR gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

Extension to Multiple Inputs

The gates shown in Fig. 2-5, except for the inverter and buffer, can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have

$$x + y = y + x \qquad \text{commutative}$$

and

$$(x + y) + z = x + (y + z) = x + y + z \qquad \text{associative}$$

which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.

The NAND and NOR functions are commutative and their gates can be extended to have more than two inputs, provided the definition of the operation is slightly modified. The difficulty is that the NAND and NOR operators are not associative, i.e., $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$, as shown in Fig. 2-6 and below:

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$

$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

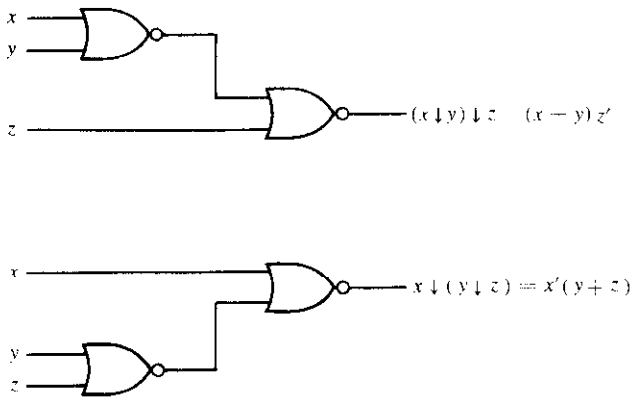


FIGURE 2-6
Demonstrating the nonassociativity of the NOR operator: $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$

To overcome this difficulty, we define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate. Thus, by definition, we have

$$x \downarrow y \downarrow z = (x + y + z)'$$
$$x \uparrow y \uparrow z = (xyz)'$$

The graphic symbols for the 3-input gates are shown in Fig. 2-7. In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper sequence of the gates. To demonstrate this, consider the circuit of Fig. 2-7(c). The Boolean function for the circuit must be written as

$$F = [(ABC)'(DE)']' = ABC + DE$$

The second expression is obtained from DeMorgan's theorem. It also shows that an expression in sum of products can be implemented with NAND gates. Further discussion of NAND and NOR gates can be found in Sections 3-6, 4-7, and 4-8.

The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs. However, multiple-input exclusive-OR gates are uncommon from the hardware standpoint. In fact, even a 2-input function is usually constructed with other types of gates. Moreoever, the definition of the function must be modified when extended to more than two variables. The exclusive-OR is an *odd* function, i.e., it is equal to 1 if the input variables have an odd number of 1's. The construction of a 3-input exclusive-OR function is shown in Fig. 2-8. It is normally implemented by cascading 2-input gates, as shown in (a). Graphically, it can be represented with a single 3-input gate, as shown in (b). The truth table in (c) clearly indicates that the output F is equal to 1 if only one input is equal to 1 or if all three inputs are equal to 1, i.e., when the total number of 1's in the input variables is *odd*. Further discussion of exclusive-OR can be found in Section 4-9.

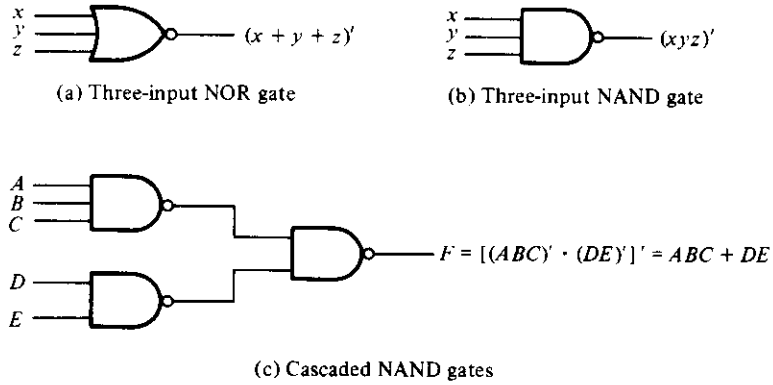
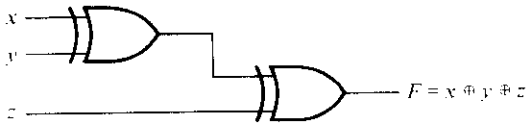
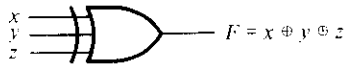


FIGURE 2-7
Multiple-input and cascaded NOR and NAND gates



(a) Using 2-input gates



(b) 3-input gate

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

(c) Truth table

FIGURE 2-8
3-input exclusive-OR gate

2-8 INTEGRATED CIRCUITS

Digital circuits are constructed with integrated circuits. An integrated circuit (abbreviated IC) is a small silicon semiconductor crystal, called a *chip*, containing the electronic components for the digital gates. The various gates are interconnected inside the chip to form the required circuit. The chip is mounted in a ceramic or plastic container, and connections are welded to external pins to form the integrated circuit. The number of pins may range from 14 in a small IC package to 64 or more in a larger package. The size of the IC package is very small. For example, four AND gates are enclosed inside a 14-pin IC package with dimensions of 20 × 8 × 3 millimeters. An entire microprocessor is enclosed within a 64-pin IC package with dimensions of 50 × 15 × 4 millimeters. Each IC has a numeric designation printed on the surface of the package for identification. Vendors publish data books that contain descriptions and all other information about the ICs that they manufacture.

Levels of Integration

Digital ICs are often categorized according to their circuit complexity as measured by the number of logic gates in a single package. The differentiation between those chips that have a few internal gates and those having hundreds or thousands of gates is made by a customary reference to a package as being either a small-, medium-, large-, or very large-scale integration device.

Small-scale integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.

Medium-sale integration (MSI) devices have a complexity of approximately 10 to 100 gates in a single package. They usually perform specific elementary digital operations such as decoders, adders, or multiplexers. MSI digital components are introduced in Chapters 5 and 7.

Large-scale integration (LSI) devices contain between 100 and a few thousand gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices. Some LSI components are presented in Chapters 5 and 7.

Very large-scale integration (VLSI) devices contain thousands of gates within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capabilities to create structures that previously were uneconomical.

Digital Logic Families

Digital integrated circuits are classified not only by their complexity or logical operation, but also by the specific circuit technology to which they belong. The circuit technology is referred to as a digital logic family. Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or an inverter gate. The electronic components used in the construction of the basic circuit are usually used as the name of the technology. Many different logic families of digital integrated circuits have been introduced commercially. The following are the most popular:

TTL	transistor-transistor logic
ECL	emiter-coupled logic
MOS	metal-oxide semiconductor
CMOS	complementary metal-oxide semiconductor

TTL is a widespread logic family that has been in operation for some time and is considered as standard. ECL has an advantage in systems requiring high-speed operation. MOS is suitable for circuits that need high component density, and CMOS is preferable in systems requiring low power consumption.

The analysis of the basic electronic digital gate circuit in each logic family is presented in Chapter 10. The reader familiar with basic electronics can refer to Chapter 10 at this time to become acquainted with these electronic circuits. Here we restrict the discussion to the general properties of the various IC gates available commercially.

The transistor-transistor logic family evolved from a previous technology that used diodes and transistors for the basic NAND gate. This technology was called DTL for diode-transistor logic. Later the diodes were replaced by transistors to improve the circuit operation and the name of the logic family was changed to TTL.

Emitter-coupled logic (ECL) circuits provide the highest speed among the integrated digital logic families. ECL is used in systems such as supercomputers and signal processors, where high speed is essential. The transistors in ECL gates operate in a nonsaturated state, a condition that allows the achievement of propagation delays of 1 to 2 nanoseconds.

The metal-oxide semiconductor (MOS) is a unipolar transistor that depends upon 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. Complementary MOS (CMOS) technology uses one PMOS and one NMOS transistor connected in a complementary fashion in all circuits. The most important advantages of MOS over bipolar transistors are the high packing density of circuits, a simpler procesing technique during fabrication, and a more economical operation because of the low power consumption.

The characteristics of digital logic families are usually compared by analyzing the circuit of the basic gate in each family. The most important parameters that are evaluated and compared are discussed in Section 10-2. They are listed here for reference.

Fan-out specifies the number of standard loads that the output of a typical gate can drive without impairing its normal operation. A standard load is usually defined as the amount of current needed by an input of another similar gate of the same family.

Power dissipation is the power consumed by the gate that must be available from the power supply.

Propagation delay is the average transition delay time for the signal to propagate from input to output. The operating speed is inversely proportional to the propagation delay.

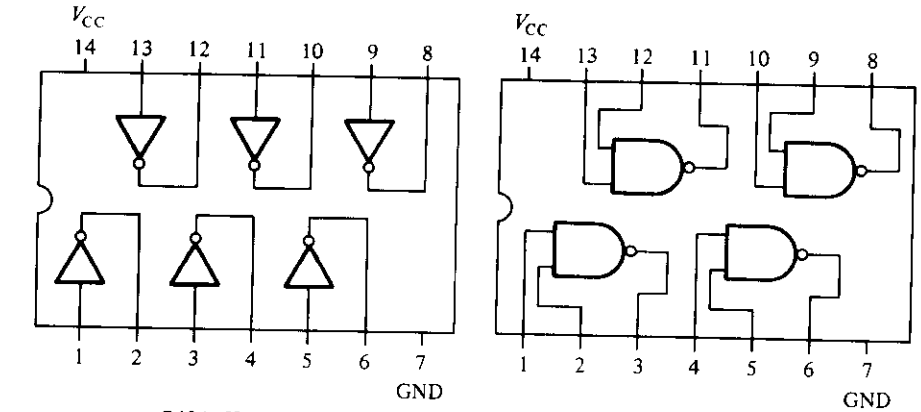
Noise margin is the minimum external noise voltage that causes an undesirable change in the circuit output.

Integrated-Circuit Gates

Some typical SSI circuits are shown in Fig. 2-9. Each IC is enclosed within a 14- or 16-pin package. A notch placed on the left side of the package is used to reference the pin numbers. The pins are numbered along the two sides starting from the notch and continuing counterclockwise. The inputs and outputs of the gates are connected to the package pins, as indicated in each diagram.

TTL IC's are usually distinguished by their numerical designation as the 5400 and 7400 series. The former has a wide operating temperature range, suitable for military use, and the latter has a narrower temperature range, suitable for commercial use. The numeric designation of 7400 series means that IC packages are numbered as 7400, 7401, 7402, etc. Fig. 2-9(a) shows two TTL SSI circuits. The 7404 provides six (hex) inverters in a package. The 7400 provides four (quadruple) 2-input NAND gates. The terminals marked V_{CC} and GRD (ground) are the power-supply pins that require a voltage of 5 volts for proper operation. The two logic levels for TTL are 0 and 3.5 volts.

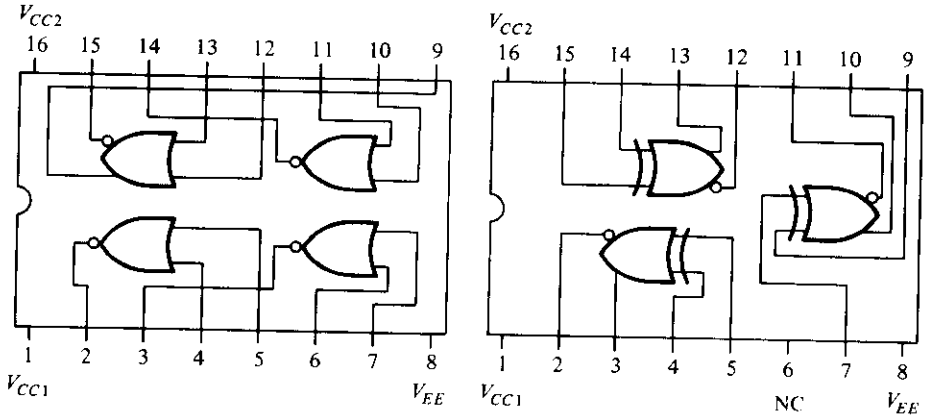
The TTL logic family actually consists of several subfamilies or series. Table 2-9 lists the name of each series and the prefix designation that identifies the IC as being part of that series. As mentioned before, ICs that are part of the standard TTL have an



7404—Hex inverters

7400—Quadruple 2-input NAND gates

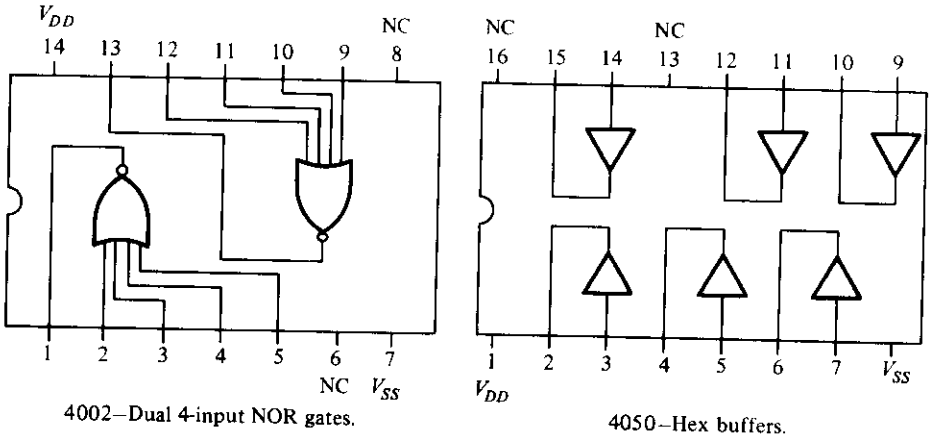
(a) TTL gates.



10102—Quadruple 2-input NOR gates

10107—Triple exclusive-OR/NOR gates

(b) ECL gates.



4002—Dual 4-input NOR gates.

4050—Hex buffers.

(c) CMOS gates.

FIGURE 2-9
Some typical integrated-circuit gates

TABLE 2-9
Various Series of the TTL Logic Family

TTL Series	Prefix	Example
Standard TTL	74	7486
High-speed TTL	74H	74H86
Low-power TTL	74L	74L86
Schottky TTL	74S	74S86
Low-power Schottky TTL	74LS	74LS86
Advanced Schottky TTL	74AS	74AS86
Advanced Low-power Schottky TTL	74ALS	74ALS86

identification number that starts with 74. Likewise, ICs that are part of the high-speed TTL series have an identification number that starts with 74H, ICs in the Schottky TTL series start with 74S, and similarly for the other series. The different characteristics of the various TTL series are listed in Table 10-2 in Chapter 10. The differences between the various TTL series are in their electrical characteristics, such as power dissipation, propagation delay, and switching speed. They do not differ in the pin assignment or logic operation performed by the internal circuits. For example, all the ICs listed in Table 2-9 with an 86 number, no matter what the prefix, contain four exclusive-OR gates with the same pin assignment in each package.

The most common ECL ICs are designated as the 10000 series. Figure 2-9(b) shows two ECL circuits. The 10102 provides four 2-input NOR gates. Note that an ECL gate may have two outputs, one for the NOR function and another for the OR function. The 10107 IC provides three exclusive-OR gates. Here again there are two outputs from each gate; the other output provides the exclusive-NOR function. ECL gates have three terminals for power supply. V_{CC1} and V_{CC2} are usually connected to ground, and V_{EE} to a -5.2-volt supply. The two logic levels for ECL are -0.8 and -1.8 volts.

CMOS circuits of the 4000 series are shown in Fig. 2-9(c). Only two 4-input NOR gates can be accommodated in the 4002 because of pin limitation. The 4050 IC provides six buffer gates. Both ICs have unused terminals marked NC (no connection). The terminal marked V_{DD} requires a power-supply voltage from 3 to 15 volts, whereas V_{SS} is usually connected to ground. The two logic levels are 0 and V_{DD} volts.

The original 4000 series of CMOS circuits was designed independently from the TTL series. Since TTL became a standard in the industry, vendors started to supply other CMOS circuits that are pin compatible with similar TTL ICs. For example, the 74C04 is a CMOS circuit that is pin compatible with TTL 7404. This means that it has six inverters connected to the pins of the package, as shown in Fig. 2-9(a). The CMOS series available commercially are listed in Table 2-10. The 74HC series operates at higher speeds than the 74C series. The 74HCT series is both electrically and pin compatible with TTL devices. This means that 74HCT ICs can be connected directly to TTL ICs without the need of interfacing circuits.

TABLE 2-10
Various Series of the CMOS Logic Family

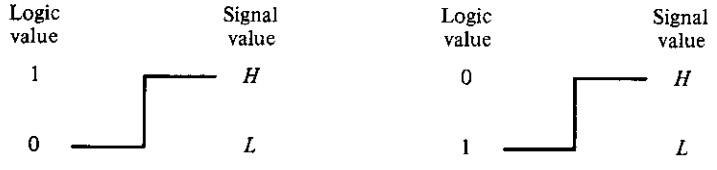
CMOS series	Prefix	Example
Original CMOS	40	4009
Pin compatible with TTL	74C	74C04
High-speed and pin compatible with TTL	74HC	74HCO4
High-speed and electrically compatible with TTL	74HCT	74HCT04

Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic-1 and the other logic-0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. 2-10. The higher signal level is designated by H and the lower signal level by L. Choosing the high-level H to represent logic-1 defines a positive logic system. Choosing the low-level L to represent logic-1 defines a negative logic system. The terms positive and negative are somewhat misleading since both signals may be positive or both may be negative. It is not the actual signal values that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

Integrated-circuit data sheets define digital gates not in terms of logic values, but rather in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity. Consider, for example, the TTL gate shown in Fig. 2-11(b). The truth table for this gate as given in a data book is listed in Fig. 2-11(a). This specifies the physical behavior of the gate when H is 3.5 volts and L is 0 volt. The truth table of Fig. 2-11(c) assumes positive logic assignment with $H = 1$ and $L = 0$. This truth table is the same as the one for the AND operation. The graphic symbol for a positive logic AND gate is shown in Fig. 2-11(d).

Now consider the negative logic assignment for the same physical gate with $L = 1$ and $H = 0$. The result is the truth table of Fig. 2-11(e). This table represents the OR operation even though the entries are reversed. The graphic symbol for the negative logic OR gate is shown in Fig. 2-11(f). The small triangles in the inputs and output

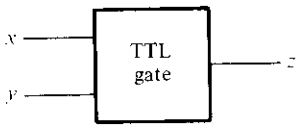


(a) Positive logic (b) Negative logic

FIGURE 2-10
Signal assignment and logic polarity

x	y	z
L	L	L
L	H	L
H	L	L
H	H	H

(a) Truth table with H and L



(b) Gate block diagram

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

(c) Truth table for positive logic

(d) Positive logic AND gate

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

(e) Truth table for negative logic

(f) Negative logic OR gate

FIGURE 2-11
Demonstration of positive and negative logic

designate a *polarity indicator*. The presence of this polarity indicator along a terminal signifies that negative logic is assumed for the signal. Thus, the same physical gate can operate either as a positive logic AND gate or as a negative logic OR gate.

The conversion from positive logic to negative logic, and vice versa, is essentially an operation that changes 1's to 0's and 0's to 1's in both the inputs and the output of a gate. Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function. The result of this conversion is that all AND operations are converted to OR operations (or graphic symbols) and vice versa. In addition, one must not forget to include the polarity-indicator triangle in the graphic symbols when negative logic is assumed. In this book, we will not use negative logic gates and assume that all gates operate with a positive logic assignment.

REFERENCES

1. BOOLE, G., *An Investigation of the Laws of Thought*. New York: Dover, 1954.
2. SHANNON, C. E., "A Symbolic Analysis of Relay and Switching Circuits." *Trans. AIEE,* **57** (1938), 713–723.
3. HUNTINGTON, E. V., "Sets of Independent Postulates for the Algebra of Logic." *Trans. Am. Math. Soc.*, **5** (1904). 288–309.
4. BIRKHOFF, G., and T. C. Bartee, *Modern Applied Algebra*. New York: McGraw-Hill, 1970.
5. HOHN, F. E., *Applied Boolean Algebra,* 2nd Ed. New York: Macmillan, 1966.
6. WHITESITT, J. E., *Boolean Algebra and Its Application*. Reading, MA: Addison-Wesley, 1961.
7. FRIEDMAN, A. D., and P. R. MENON, *Theory and Design of Switching Circuits*. Rockville, MD: Computer Science Press, 1975.
8. *The TTL Data Book*. Dallas: Texas Instruments, 1988.
9. TOCCI, R. J., *Digital Systems Principles and Applications,* 4th Ed. Englewood Cliffs, NJ: Prentice-Hall, 1988.

PROBLEMS

2-1 Demonstrate by means of truth tables the validity of the following identities:
(a) DeMorgan's theorem for three variables: $(xyz)' = x' + y' + z'$.
(b) The second distributive law: $x + yz = (x + y)(x + z)$.
(c) The consensus theorem: $xy + x'z + yz = xy + x'z$. (This is done algebraically in Example 2-1, part 4.)

2-2 Simplify the following Boolean expressions to a minimum number of literals.
(a) $x'y' + xy + x'y$
(b) $(x + y)(x + y')$
(c) $x'y + xy' + xy + x'y'$
(d) $x' + xy + xz' + xy'z'$
(e) $xy' + y'z' + x'z'$ [use the consensus theorem, Problem 2-1(c)].

2-3 Simplify the following Boolean expressions to a minimum number of literals:
(a) $ABC + A'B + ABC'$
(b) $x'yz + xz$
(c) $(x + y)'(x' + y')$
(d) $xy + x(wz + wz')$
(e) $(BC' + A'D)(AB' + CD')$

2-4 Reduce the following Boolean expressions to the indicated number of literals:
(a) $A'C' + ABC + AC'$ to three literals
(b) $(x'y' + z)' + z + xy + wz$ to three literals
(c) $A'B(D' + C'D) + B(A + A'CD)$ to one literal
(d) $(A' + C)(A' + C')(A + B + C'D)$ to four literals

2-5 Find the complement of $F = x + yz$; then show that $F \cdot F' = 0$ and $F + F' = 1$.

2-6 Find the complement of the following expressions:
(a) $xy' + x'y$
(b) $(AB' + C)D' + E$
(c) $AB(C'D + CD') + A'B'(C' + D)(C + D')$
(d) $(x + y' + z)(x' + z')(x + y)$

2-7 Using DeMorgan's theorem, convert the following Boolean expressions to equivalent expressions that have only OR and complement operations. Show that the functions can be implemented with logic diagrams that have only OR gates and inverters.
(a) $F = x'y' + x'z + y'z$
(b) $F = (y + z')(x + y)(y' + z)$

2-8 Using DeMorgan's theorem, convert the two Boolean expressions listed in Problem 2-7 to equivalent expressions that have only AND and complement operations. Show that the functions can be implemented with only AND gates and inverters.

2-9 Obtain the truth table of the following functions and express each function in sum of minterms and product of maxterms:
(a) $(xy + z)(y + xz)$
(b) $(A' + B)(B' + C)$
(c) $y'z + wxy' + wxz' + w'x'z$

2-10 For the Boolean function F given in the truth table, find the following:
(a) List the minterms of the function.
(b) List the minterms of F'.
(c) Express F in sum of minterms in algebraic form.
(d) Simplify the function to an expression with a minimum number of literals.

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

2-11 Given the following Boolean function:

$$F = xy'z + x'y'z + w'xy + wx'y + wxy$$

(a) Obtain the truth table of the function.
(b) Draw the logic diagram using the original Boolean expression.
(c) Simplify the function to a minimum number of literals using Boolean algebra.
(d) Obtain the truth table of the function from the simplified expression and show that it is the same as the one in part (a).

(e) Draw the logic diagram from the simplified expression and compare the total number of gates with the diagram of part (b).

2-12 Express the following functions in sum of minterms and product of maxterms:
(a) $F(A, B, C, D) = B'D + A'D + BD$
(b) $F(x, y, z) = (xy + z)(xz + y)$

2-13 Express the complement of the following functions in sum of minterms:
(a) $F(A, B, C, D) = \Sigma(0, 2, 6, 11, 13, 14)$
(b) $F(x, y, z) = \Pi(0, 3, 6, 7)$

2-14 Convert the following to the other canonical form:
(a) $F(x, y, z) = \Sigma(1, 3, 7)$
(b) $F(A, B, C, D) = \Pi(0, 1, 2, 3, 4, 6, 12)$

2-15 The sum of all the minterms of a Boolean function of n variables is equal to 1.
(a) Prove the above statement for $n = 3$.
(b) Suggest a procedure for a general proof.

2-16 Convert the following expressions into sum of products and product of sums:
(a) $(AB + C)(B + C'D)$
(b) $x' + x(x + y')(y + z')$

2-17 Draw the logic diagram corresponding to the following Boolean expressions without simplifying them:
(a) $BC' + AB + ACD$
(b) $(A + B)(C + D)(A' + B + D)$
(c) $(AB + A'B')(CD' + C'D)$

2-18 Show that the dual of the exclusive-OR is equal to its complement.

2-19 By substituting the Boolean expression equivalent of the binary operations as defined in Table 2-8, show the following:
(a) The inhibition operation is neither commutative nor associative.
(b) The exclusive-OR operation is commutative and associative.

2-20 Verify the truth table for the three-variable exclusive-OR function listed in Fig. 2-8(c). Do that by listing all eight combinations of x, y, and z; then evaluate $A = x \oplus y$; and then evaluate $F = A \oplus z = x \oplus y \oplus z$.

2-21 TTL SSI come mostly in 14-pin packages. Two pins are reserved for power and the other 12 pins are used for input and output terminals. Determine the number of gates that can be enclosed in one package if it contains the folowing type of gates:
(a) Two-input exclusive-OR gates
(b) Three-input AND gates
(c) Four-input NAND gates
(d) Five-input NOR gates
(e) Eight-input NAND gates

2-22 Show that a positive logic NAND gate is a negative logic NOR gate and vice versa.

2-23 An integrated-circuit logic family has NAND gates with fan-out of 5 and buffer gates with fan-out of 10. Show how the output signal of a single NAND gate can be applied to 50 other NAND-gate inputs without overloading the output gate. Use buffers to satisfy the fan-out requirements.

Simplification of Boolean Functions

3-1 THE MAP METHOD

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, expressed algebraically, it can appear in many different forms. Boolean functions may be simplified by algebraic means as discussed in Section 2-4. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method provides a simple straightforward procedure for minimizing Boolean functions. This method may be regarded either as a pictorial form of a truth table or as an extension of the Venn diagram. The map method, first proposed by Veitch and modified by Karnaugh, is also known as the "Veitch diagram" or the "Karnaugh map."

The map is a diagram made up of squares. Each square represents one minterm. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function. In fact, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which he can select the simplest one. We shall assume that the simplest algebraic expression is any one in a sum of products or product of sums that has a minimum number of literals. (This expression is not necessarily unique.)

3-2 TWO- AND THREE-VARIABLE MAPS

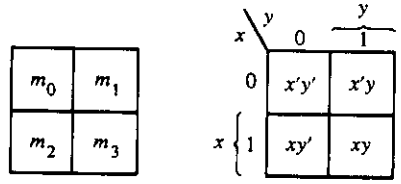
A two-variable map is shown in Fig. 3-1(a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm. The map is redrawn in (b) to show the relationship between the squares and the two variables. The 0's and 1's marked for each row and each column designate the values of variables x and y, respectively. Notice that x appears primed in row 0 and unprimed in row 1. Similarly, y appears primed in column 0 and unprimed in column 1.

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function xy is shown in Fig. 3-2(a). Since xy is equal to m_3, a 1 is placed inside the square that belongs to m_3. Similarly, the function $x + y$ is represented in the map of Fig. 3-2(b) by three squares marked with 1's. These squares are found from the minterms of the function:

$$x + y = x'y + xy' + xy = m_1 + m_2 + m_3$$

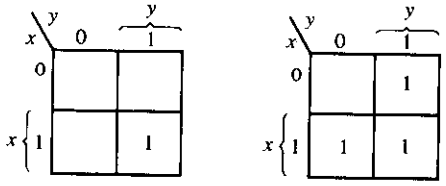
The three squares could have also been determined from the intersection of variable x in the second row and variable y in the second column, which encloses the area belonging to x or y.

A three-variable map is shown in Fig. 3-3. There are eight minterms for three binary variables. Therefore, a map consists of eight squares. Note that the minterms are not arranged in a binary sequence, but in a sequence similar to the Gray code listed in Table 1-4. The characteristic of this sequence is that only one bit changes from 1 to 0



(a) (b)

FIGURE 3-1
Two-variable map



(a) xy (b) $x + y$

FIGURE 3-2
Representation of functions in the map

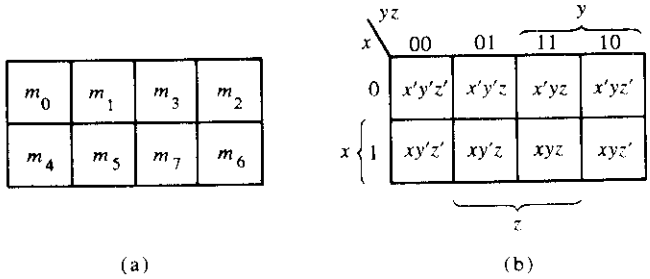


FIGURE 3-3
Three-variable map

or from 0 to 1 in the listing sequence. The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables. For example, the square assigned to m_5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Another way of looking at square $m_5 = xy'z$ is to consider it to be in the row marked x and the column belonging to $y'z$ (column 01). Note that there are four squares where each variable is equal to 1 and four where each is equal to 0. The variable appears unprimed in those four squares where it is equal to 1 and primed in those squares where it is equal to 0. For convenience, we write the variable with its letter symbol under the four squares where it is unprimed.

To understand the usefulness of the map for simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares. Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other. For example, m_5 and m_7 lie in two adjacent squares. Variable y is primed in m_5 and unprimed in m_7, whereas the other two variables are the same in both squares. From the postulates of Boolean algebra, it follows that the sum of two minterms in adjacent squares can be simplified to a single AND term consisting of only two literals. To clarify this, consider the sum of two adjacent squares such as m_5 and m_7:

$$m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

Here the two squares differ by the variable y, which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares that are ORed together will cause a removal of the different variable. The following example explains the procedure for minimizing a Boolean function with a map.

Example 3-1

Simplify the Boolean function

$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

First, a 1 is marked in each minterm that represents the function. This is shown in Fig. 3-4, where the squares for minterms 010, 011, 100, and 101 are marked with 1's. The next step is to find possible adjacent squares. These are indicated in the map by two rectangles, each enclosing two 1's. The upper right rectangle represents the area en-

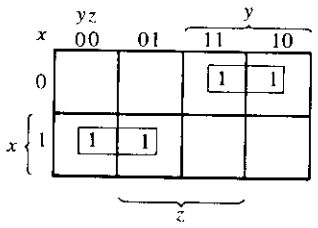


FIGURE 3-4
Map for Example 3-1; $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

closed by $x'y$. This is determined by observing that the two-square area is in row 0, corresponding to x', and the last two columns, corresponding to y. Similarly, the lower left rectangle represents the product term xy'. (The second row represents x and the two left columns represent y'.) The logical sum of these two product terms gives the simplified expression

$$F = x'y + xy'$$ ■

There are cases where two squares in the map are considered to be adjacent even though they do not touch each other. In Fig. 3-3, m_0 is adjacent to m_2 and m_4 is adjacent to m_6 because the minterms differ by one variable. This can be readily verified algebraically.

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$
$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

Consequently, we must modify the definition of adjacent squares to include this and other similar cases. This is done by considering the map as being drawn on a surface where the right and left edges touch each other to form adjacent squares.

Example 3-2

Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

The map for this function is shown in Fig. 3-5. There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined in the third column to give a two-literal term yz. The remaining two squares with 1's are also adjacent by the new definition and are shown in the diagram with their values enclosed in half rectangles. These two squares when combined, give the two-literal term xz'. The simplified function becomes

$$F = yz + xz'$$ ■

Consider now any combination of four adjacent squares in the three-variable map. Any such combination represents the logical sum of four minterms and results in an ex-

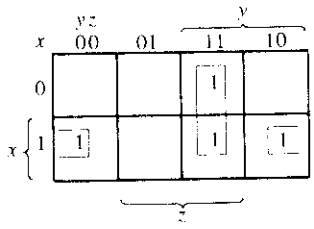


FIGURE 3-5
Map for Example 3-2; $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

pression of only one literal. As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to a single literal term z'.

$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\ &= x'z'(y' + y) + xz'(y' + y) \\ &= x'z' + xz' = z'(x' + x) = z' \end{aligned}$$

The number of adjacent squares that may be combined must always represent a number that is a power of two such as 1, 2, 4, and 8. As a larger number of adjacent squares are combined, we obtain a product term with fewer literals.

One square represents one minterm, giving a term of three literals.

Two adjacent squares represent a term of two literals.

Four adjacent squares represent a term of one literal.

Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

Example 3-3 Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

The map for F is shown in Fig. 3-6. First, we combine the four adjacent squares in the first and last columns to give the single literal term z'. The remaining single square representing minterm 5 is combined with an adjacent square that has already been used once. This is not only permissible, but rather desirable since the two adjacent squares give the two-literal term xy' and the single square represents the three-literal minterm $xy'z$. The simplified function is

$$F = z' + xy'$$ ■

If a function is not expressed in sum of minterms, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms. It is necessary to make sure that the algebraic ex-

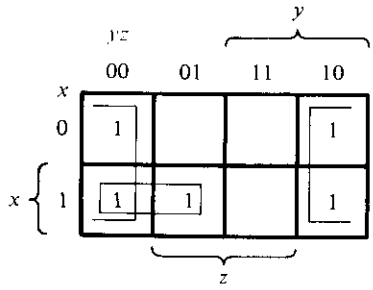


FIGURE 3-6
Map for Example 3-3; $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

pression is in sum of products form. Each product term can be plotted in the map in one, two, or more squares. The minterms of the function are then read directly from the map.

Example 3-4

Given the following Boolean function:

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

(a) Express it in sum of minterms.
(b) Find the minimal sum of products expression.

Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term $A'C$ are found in Fig. 3-7 from the coincidence of A' (first row) and C (two middle columns) to give squares 001 and 011. Note that when marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term $A'B$, which has 1's in squares 011 and 010, but square 011 is common with the first term $A'C$, so only one 1 is marked in it. Continuing in this fashion, we determine that the term $AB'C$ belongs in square 101, corresponding to minterm 5, and the term BC has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Fig. 3-7. The minterms are read

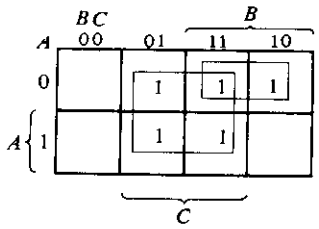


FIGURE 3-7
Map for Example 3-4; $A'C + A'B + AB'C + BC = C + A'B$

directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum of minterms form:

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

The sum of products expression as originally given has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$ ■

3-3 FOUR-VARIABLE MAP

The map for Boolean functions of four binary variables is shown in Fig. 3-8. In (a) are listed the 16 minterms and the squares assigned to each. In (b) the map is redrawn to show the relationship with the four variables. The rows and columns are numbered in a reflected-code sequence, with only one digit changing value between two adjacent rows or columns. The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m_{13}.

The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example, m_0 and m_2 form adjacent squares, as do m_3 and m_{11}. The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

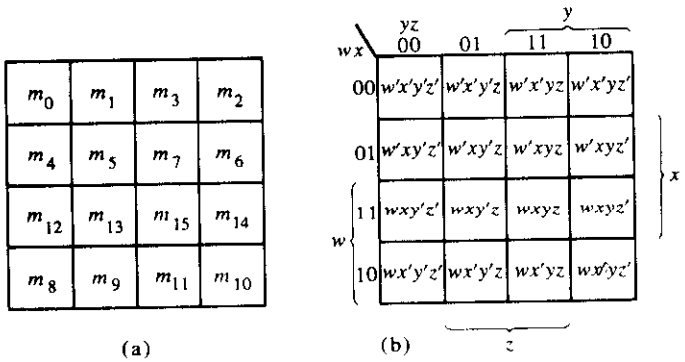


FIGURE 3-8
Four-variable map

One square represents one minterm, giving a term of four literals.
Two adjacent squares represent a term of three literals.
Four adjacent squares represent a term of two literals.
Eight adjacent squares represent a term of one literal.
Sixteen adjacent squares represent the function equal to 1.

No other combination of squares can simplify the function. The following two examples show the procedure used to simplify four-variable Boolean functions.

Example 3-5

Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3-9. Eight adjacent squares marked with 1's can be combined to form the one literal term y'. The remaining three 1's on the right cannot be combined to give a simplified term. They must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$. Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term of four literals), we combine it with squares already used to form an area of four adjacent squares. These squares comprise the two middle rows and the two end columns, giving the term xz'. The simplified function is

$$F = y' + w'z' + xz'$$

■

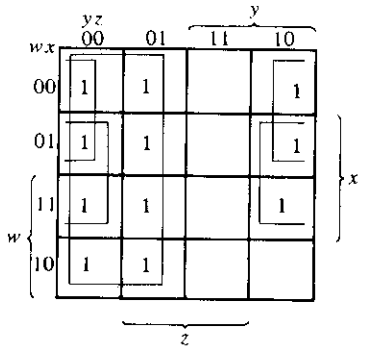


FIGURE 3-9
Map for Example 3-5; $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

Example 3-6 Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

The area in the map covered by this function consists of the squares marked with 1's in Fig. 3-10. This function has four variables and, as expressed, consists of three terms, each with three literals, and one term of four literals. Each term of three literals is represented in the map by two squares. For example, $A'B'C'$ is represented in squares 0000 and 0001. The function can be simplified in the map by taking the 1's in the four corners to give the term $B'D'$. This is possible because these four squares are adjacent when the map is drawn in a surface with top and bottom or left and right edges touching one another. The two left-hand 1's in the top row are combined with the two 1's in the bottom row to give the term $B'C'$. The remaining 1 may be combined in a two-square area to give the term $A'CD'$. The simplified function is

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

■

Prime Implicants

When choosing adjacent squares in a map, we must ensure that all the minterms of the function are covered when combining the squares. At the same time, it is necessary to minimize the number of terms in the expression and avoid any redundant terms whose minterms are already covered by other terms. Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of the terms referred to as prime implicant and essential prime implicant. A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential*. A more satisfactory definition of prime implicant is given in Section 3-10. Here we will use it to help us find all possible simplified expressions of a Boolean function by means of a map.

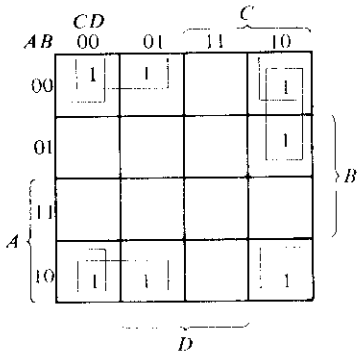


FIGURE 3-10
Map for Example 3-6; $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

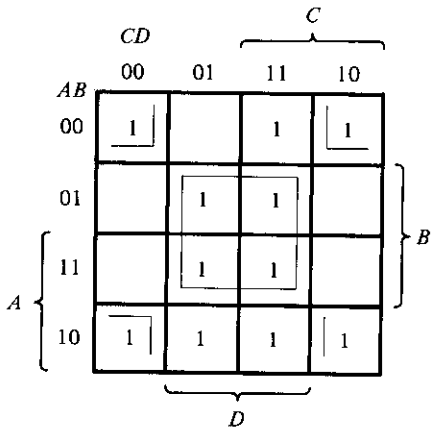
The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant provided they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

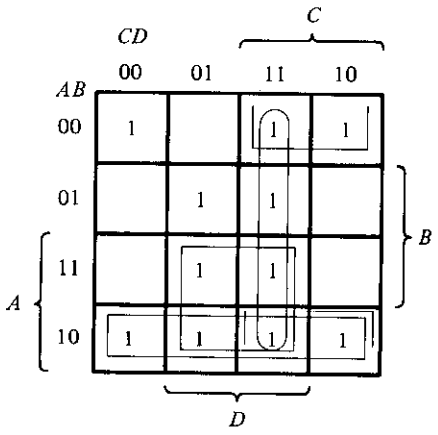
$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

The minterms of the function are marked with 1's in the maps of Fig. 3-11. Part (a) of the figure shows two essential prime implicants. One term is essential because there is only one way to include minterms m_0 within four adjacent squares. These four squares define the term $B'D'$. Similarly, there is only one way that minterm m_5 can be combined with four adjacent squares and this gives the second term BD. The two essential prime implicants cover eight minterms. The remaining three minterms, m_3, m_9, and m_{11}, must be considered next.

Figure 3-11(b) shows all possible ways that the three minterms can be covered with prime implicants. Minterm m_3 can be covered with either prime implicant CD or $B'C$. Minterm m_9 can be covered with either AD or AB'. Minterm m_{11} is covered with any one of the four prime implicants. The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m_3, m_9, and m_{11}. There are four possible ways that the function can be expressed with four product terms of two literals each:



(a) Essential prime implicants BD and $B'D'$



(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE 3-11
Simplification using prime implicants

$$F = BD + B'D' + CD + AD$$
$$= BD + B'D' + CD + AB'$$
$$= BD + B'D' + B'C + AD$$
$$= BD + B'D' + B'C + AB'$$

The above example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.

The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants. The simplified expression is obtained from the logical sum of all the essential prime implicants plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants. Occasionally, there may be more than one way of combining squares and each combination may produce an equally simplified expression.

3-4 FIVE-VARIABLE MAP

Maps for more than four variables are not as simple to use. A five-variable map needs 32 squares and a six-variable map needs 64 squares. When the number of variables becomes large, the number of squares becomes excessively large and the geometry for combining adjacent squares becomes more involved.

The five-variable map is shown in Fig. 3-12. It consists of 2 four-variable maps with variables A, B, C, D, and E. Variable A distinguishes between the two maps, as indicated on the top of the diagram. The left-hand four-variable map represents the 16

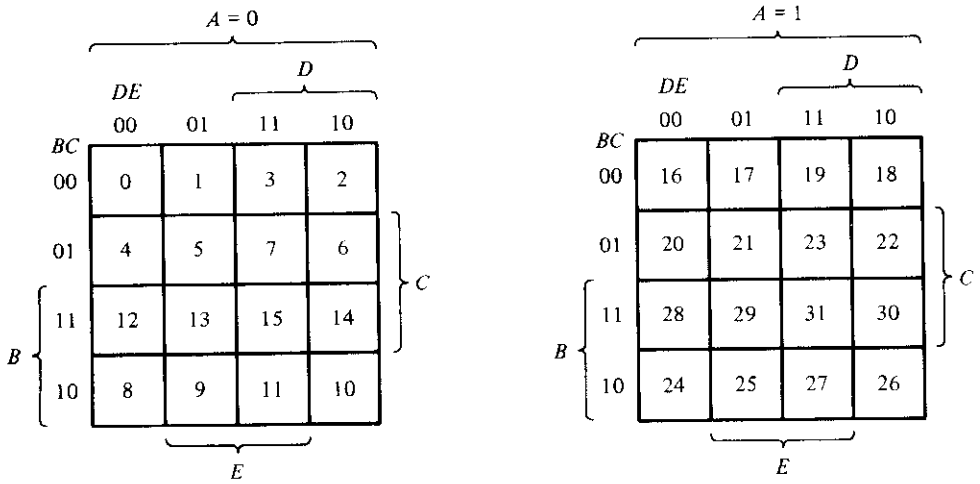


FIGURE 3-12
Five-variable map

squares where $A = 0$, and the other four-variable map represents the squares where $A = 1$. Minterms 0 through 15 belong with $A = 0$ and minterms 16 through 31 with $A = 1$. Each four-variable map retains the previously defined adjacency when taken separately. In addition, each square in the $A = 0$ map is adjacent to the corresponding square in the $A = 1$ map. For example, minterm 4 is adjacent to minterm 20 and minterm 15 to 31. The best way to visualize this new rule for adjacent squares is to consider the two half maps as being one on top of the other. Any two squares that fall one over the other are considered adjacent.

By following the procedure used for the five-variable map, it is possible to construct a six-variable map with 4 four-variable maps to obtain the required 64 squares. Maps with six or more variables need too many squares and are impractical to use. The alternative is to employ computer programs specifically written to facilitate the simplification of Boolean functions with a large number of variables.

From inspection, and taking into account the new definition of adjacent squares, it is possible to show that any 2^k adjacent squares, for $k = 0, 1, 2, \ldots, n$, in an n-variable map, will represent an area that gives a term of $n - k$ literals. For the above statement to have any meaning, n must be larger than k. When $n = k$, the entire area of the map is combined to give the identity function. Table 3-1 shows the relationship between the number of adjacent squares and the number of literals in the term. For example, eight adjacent squares combine an area in the five-variable map to give a term of two literals.

TABLE 3-1
The Relationship Between the Number of Adjacent Squares and the Number of Literals in the Term

	Number of adjacent squares	Number of literals in a term in an n-variable map					
k	2^k	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
0	1	2	3	4	5	6	7
1	2	1	2	3	4	5	6
2	4	0	1	2	3	4	5
3	8		0	1	2	3	4
4	16			0	1	2	3
5	32				0	1	2
6	64					0	1

Example 3-7 Simplify the Boolean function

$$F(A, B, C, D, E) = (0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$$

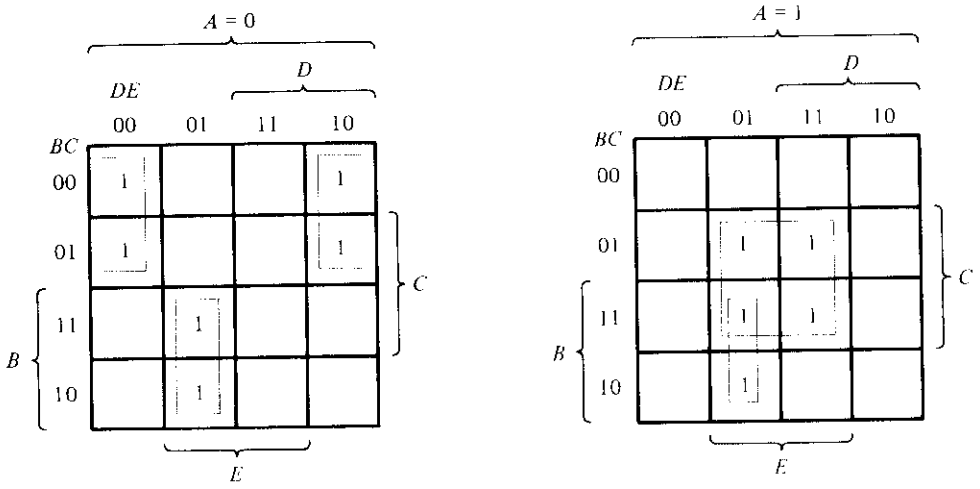


FIGURE 3-13
Map for Example 3-7; $F = A'B'E' + BD'E + ACE$

The five-variable map for this function is shown in Fig. 3-13. There are six minterms from 0 to 15 that belong to the part of the map with $A = 0$. The other five minterms belong with $A = 1$. Four adjacent squares in the $A = 0$ map are combined to give the three-literal term $A'B'E'$. Note that it is necessary to include A' with the term because all the squares are associated with $A = 0$. The two squares in column 01 and the last two rows are common to both parts of the map. Therefore, they constitute four adjacent squares and give the three-literal term $BD'E$. Variable A is not included here because the adjacent squares belong to both $A = 0$ and $A = 1$. The term ACE is obtained from the four adjacent squares that are entirely within the $A = 1$ map. The simplified function is the logical sum of the three terms:

$$F = A'B'E' + BD'E + ACE$$ ■

3-5 PRODUCT OF SUMS SIMPLIFICATION

The minimized Boolean functions derived from the map in all previous examples were expressed in the sum of products form. With a minor modification, the product of sums form can be obtained.

The procedure for obtaining a minimized function in product of sums follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the function denote the complement of the function. From this we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified expression of the complement of the function, i.e., of F'. The complement of F' gives us

back the function F. Because of the generalized DeMorgan's theorem, the function so obtained is automatically in the product of sums form. The best way to show this is by example.

Example 3-8

Simplify the following Boolean function in (a) sum of products and (b) product of sums.

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

The 1's marked in the map of Fig. 3-14 represent all the minterms of 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 1's gives the simplified function in sum of products:

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

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

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

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in Section 2-4), we obtain the simplified function in product of sums:

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

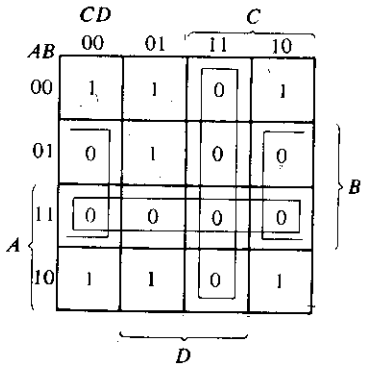
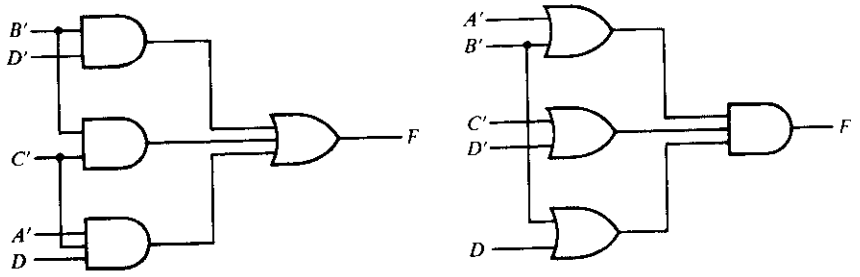


FIGURE 3-14
Map for Example 3-8; $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$

The implementation of the simplified expressions obtained in Example 3-8 is shown in Fig. 3-15. The sum of products expression is implemented in (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 (b) in its product of sums form with a group of OR gates, one for each OR term. The outputs of the OR



(a) $F = B'D' + B'C' + A'C'D$ (b) $F = (A' - B')(C' + D')(B' + D)$

FIGURE 3-15
Gate implementation of the function of Example 3-8

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 needed. The configuration pattern established in Fig. 3-15 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum of products; OR gates are connected to a single AND gate when in product of sums. Either configuration forms two levels of gates. Thus, the implementation of a function in a standard form is said to be a two-level implementation.

Example 3-8 showed the procedure for obtaining the product of sums simplification when the function is originally expressed in the sum of minterms canonical form. The procedure is also valid when the function is originally expressed in the product of maxterms canonical form. Consider, for example, the truth table that defines the function F in Table 3-2. In sum of minterms, this function is expressed as

$$F(x, y, z) = \Sigma(1, 3, 4, 6)$$

In product of maxterms, it is expressed as

$$F(x, y, z) = \Pi(0, 2, 5, 7)$$

TABLE 3-2
Truth Table of Function *F*

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

In other words, the 1's of the function represent the minterms, and the 0's represent the maxterms. The map for this function is shown in Fig. 3-16. One can start simplifying this function by first marking the 1's for each minterm that the function is a 1. The remaining squares are marked by 0's. If, on the other hand, the product of maxterms is initially given, one can start marking 0's in those squares listed in the function; the remaining squares are then marked by 1's. Once the 1's and 0's are marked, the function can be simplified in either one of the standard forms. For the sum of products, we combine the 1's to obtain

$$F = x'z + xz'$$

For the product of sums, we combine the 0's to obtain the simplified complemented function:

$$F' = xz + x'z'$$

which shows that the exclusive-OR function is the complement of the equivalence function (Section 2-6). Taking the complement of F', we obtain the simplified function in product of sums:

$$F = (x' + z')(x + z)$$

To enter a function expressed in product of sums in the map, take the complement of the function and from it find the squares to be marked by 0's. For example, the function

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

can be entered in the map by first taking its complement:

$$F' = ABC + B'D'$$

and then marking 0's in the squares representing the minterms of F'. The remaining squares are marked with 1's.

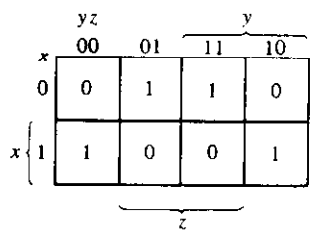


FIGURE 3-16
Map for the function of Table 3-2

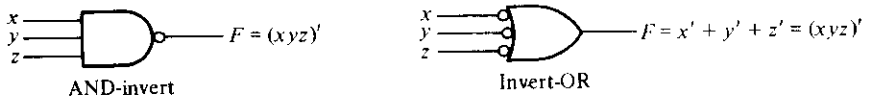
3-6 NAND AND NOR IMPLEMENTATION

Digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams. The procedure for two-level implementation is presented in this section. Multilevel implementation is discussed in Section 4-7.

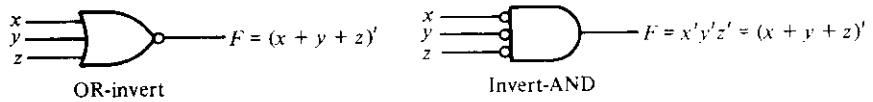
To facilitate the conversion to NAND and NOR logic, it is convenient to define two other graphic symbols for these gates. Two equivalent symbols for the NAND gate are shown in Fig. 3-17(a). The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle. Instead, it is possible to represent a NAND gate by an OR graphic symbol preceded by small circles in all the inputs. The invert-OR symbol for the NAND gate follows from DeMorgan's theorem and from the convention that small circles denote complementation.

Similarly, there are two graphic symbols for the NOR gate, as shown in Fig. 3-17(b). The OR-invert is the conventional symbol. The invert-AND is a convenient alternative that utilizes DeMorgan's theorem and the convention that small circles in the inputs denote complementation.

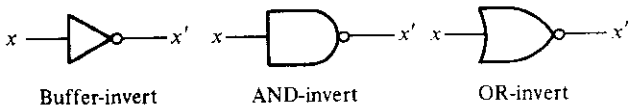
A one-input NAND or NOR gate behaves like an inverter. As a consequence, an inverter gate can be drawn in three different ways, as shown in Fig. 3-17(c). The small circles in all inverter symbols can be transferred to the input terminal without changing the logic of the gate.



(a) Two graphic symbols for NAND gate.



(b) Two graphic symbols for NOR gate.



(c) Three graphic symbols for inverter.

FIGURE 3-17
Graphic symbols for NAND and NOR gates

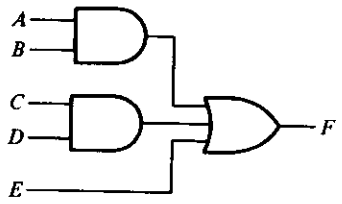
It should be pointed out that the alternate symbols for the NAND and NOR gates could be drawn with small triangles in all input terminals instead of the circles. A small triangle is a negative-logic polarity indicator (see Section 2-8 and Fig. 2-11). With small triangles in the input terminals, the graphic symbol denotes a negative-logic polarity for the inputs, but the output of the gate (not having a triangle) would have a positive-logic assignment. In this book, we prefer to stay with positive logic throughout and employ small circles when necessary to denote complementation.

NAND Implementation

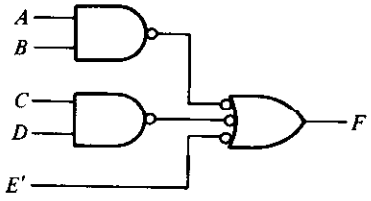
The implementation of a Boolean function with NAND gates requires that the function be simplified in the sum of products form. To see the relationship between a sum of products expression and its equivalent NAND implementation, consider the logic diagrams of Fig. 3-18. All three diagrams are equivalent and implement the function:

$$F = AB + CD + E$$

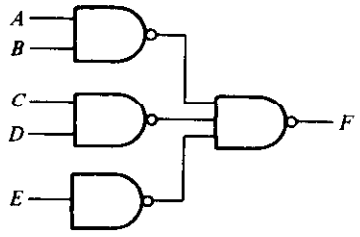
The function is implemented in Fig. 3-18(a) in sum of products form with AND and OR gates. In (b) the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an invert-OR symbol. The single variable E is complemented and applied to the second-level invert-OR gate. Remember that a small circle denotes complementation. Therefore, two circles on the same line represent double complementation and both can be removed. The complement of E goes through a small



(a) AND-OR



(b) NAND-NAND



(c) NAND-NAND

FIGURE 3-18
Three ways to implement $F = AB + CD + E$

circle that complements the variable again to produce the normal value of E. Removing the small circles in the gates of Fig. 3-18(b) produces the circuit in (a). Therefore, the two diagrams implement the same function and are equivalent.

In Fig. 3-18(c), the output NAND gate is redrawn with the conventional symbol. The one-input NAND gate complements variable E. It is possible to remove this inverter and apply E' directly to the input of the second-level NAND gate. The diagram in (c) is equivalent to the one in (b), which in turn is equivalent to the diagram in (a). Note the similarity between the diagrams in (a) and (c). The AND and OR gates have been changed to NAND gates, but an additional NAND gate has been included with the single variable E. When drawing NAND logic diagrams, the circuit shown in either (b) or (c) is acceptable. The one in (b), however, represents a more direct relationship to the Boolean expression it implements.

The NAND implementation in Fig. 3-18(c) can be verified algebraically. The NAND function it implements can be easily converted to a sum of products form by using DeMorgan's theorem:

$$F = [(AB)' \cdot (CD)' \cdot E']' = AB + CD + E$$

From the transformation shown in Fig. 3-18, we conclude that a Boolean function can be implemented with two levels of NAND gates. The rule for obtaining the NAND logic diagram from a Boolean function is as follows:

1. Simplify the function and express it in sum of products.
2. Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of the term. This constitutes a group of first-level gates.
3. Draw a single NAND gate (using the AND-invert or invert-OR graphic symbol) in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level or may be complemented and applied as an input to the second-level NAND gate.

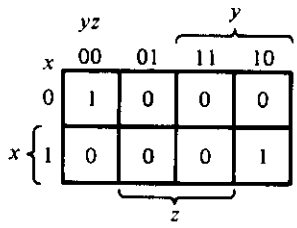
Before applying these rules to a specific example, it should be mentioned that there is a second way to implement a Boolean function with NAND gates. Remember that if we combine the 0's in a map, we obtain the simplified expression of the *complement* of the function in sum of products. The complement of the function can then be implemented with two levels of NAND gates using the rules stated above. If the normal output is desired, it would be necessary to insert a one-input NAND or inverter gate to generate the true value of the output variable. There are occasions where the designer may want to generate the complement of the function; so this second method may be preferable.

Example 3-9

Implement the following function with NAND gates:

$$F(x, y, z) = \Sigma(0, 6)$$

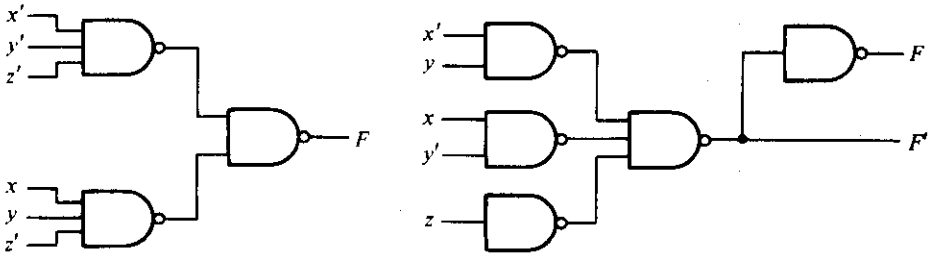
The first step is to simplify the function in sum of products form. This is attempted with the map shown in Fig. 3-19(a). There are only two 1's in the map, and they can-



$F = x'y'z' + xyz'$

$F' = x'y + xy' + z$

(a) Map simplification in sum of products.



(b) $F = x'y'z' + xyz'$ (c) $F' = x'y + xy' + z$

FIGURE 3-19
Implementation of the function of Example 3-9 with NAND gates

not be combined. The simplified function in sum of products for this example is

$$F = x'y'z' + xyz'$$

The two-level NAND implementation is shown in Fig. 3-19(b). Next we try to simplify the complement of the function in sum of products. This is done by combining the 0's in the map:

$$F' = x'y + xy' + z$$

The two-level NAND gate for generating F' is shown in Fig. 3-19(c). If output F is required, it is necessary to add a one-input NAND gate to invert the function. This gives a three-level implementation. In each case, it is assumed that the input variables are available in both the normal and complement forms. If they were available in only one form, it would be necessary to insert inverters in the inputs, which would add another level to the circuits. The one-input NAND gate associated with the single variable z can be removed provided the input is changed to z'. ■

NOR Implementation

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic.

The implementation of a Boolean function with NOR gates requires that the function be simplified in product of sums form. A product of sums expression specifies a group of OR gates for the sum terms, followed by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is depicted in Fig. 3-20. It is similar to the NAND transformation discussed previously, except that now we use the product of sums expression

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

The rule for obtaining the NOR logic diagram from a Boolean function can be derived from this transformation. It is similar to the three-step NAND rule, except that the simplified expression must be in the product of sums and the terms for the first-level NOR gates are the sum terms. A term with a single literal requires a one-input NOR or inverter gate or may be complemented and applied directly to the second-level NOR gate.

A second way to implement a function with NOR gates would be to use the expression for the complement of the function in product of sums. This will give a two-level implementation for F' and a three-level implementation if the normal output F is required.

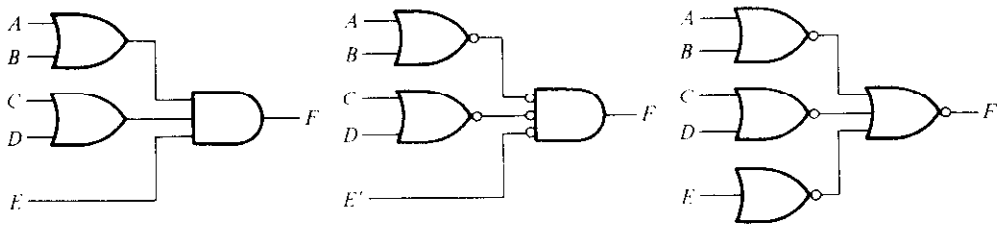
To obtain the simplified product of sums from a map, it is necessary to combine the 0's in the map and then complement the function. To obtain the simplified product of sums expression for the complement of the function, it is necessary to combine the 1's in the map and then complement the function. The following example demonstrates the procedure for NOR implementation.

Example 3-10 Implement the function of Example 3-9 with NOR gates.

The map of this function is drawn in Fig. 3-19(a). First, combine the 0's in the map to obtain

$$F' = x'y + xy' + z$$

This is the complement of the function in sum of products. Complement F' to obtain the simplified function in product of sums as required for NOR implementation:



(a) OR-AND (b) NOR-NOR (c) NOR-NOR

FIGURE 3-20
Three ways to implement $F = (A + B)(C + D)E$

$$F = (x + y')(x' + y)z'$$

The two-level implementation with NOR gates is shown in Fig. 3-21(a). The term with a single literal z' requires a one-input NOR or inverter gate. This gate can be removed and input z applied directly to the input of the second-level NOR gate.

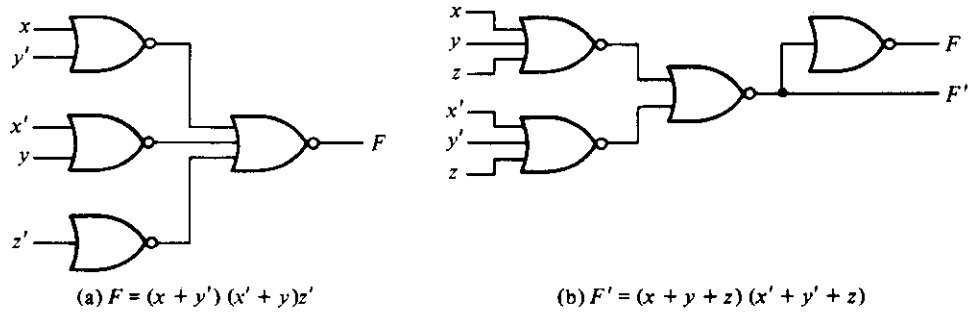
A second implementation is possible from the complement of the function in product of sums. For this case, first combine the 1's in the map to obtain

$$F = x'y'z' + xyz'$$

This is the simplified expression in sum of products. Complement this function to obtain the complement of the function in product of sums as required for NOR implementation:

$$F' = (x + y + z)(x' + y' + z)$$

The two-level implementation for F' is shown in Fig. 3-21(b). If output F is desired, it can be generated with an inverter in the third level. ■



(a) $F = (x + y')(x' + y)z'$ (b) $F' = (x + y + z)(x' + y' + z)$

FIGURE 3-21
Implementation with NOR gates

Table 3-3 summarizes the procedures for NAND or NOR implementation. One should not forget to always simplify the function in order to reduce the number of gates in the implementation. The standard forms obtained from the map-simplification procedures apply directly and are very useful when dealing with NAND or NOR logic.

TABLE 3-3
Rules for NAND and NOR Implementation

Case	Function to simplify	Standard form to use	How to derive	Implement with	Number of levels to F
(a)	F	Sum of products	Combine 1's in map	NAND	2
(b)	F'	Sum of products	Combine 0's in map	NAND	3
(c)	F	Product of sums	Complement F' in (b)	NOR	2
(d)	F'	Product of sums	Complement F in (a)	NOR	3

3-7 OTHER TWO-LEVEL IMPLEMENTATIONS

The types of gates most often found in integrated circuits are NAND and NOR. For this reason, NAND and NOR logic implementations are the most important from a practical point of view. Some NAND or NOR gates (but not all) allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function. This type of logic is called *wired logic*. For example, open-collector TTL NAND gates, when tied together, perform the wired-AND logic. (The open-collector TTL gate is shown in Chapter 10, Fig. 10-11.) The wired-AND logic performed with two NAND gates is depicted in Fig. 3-22(a). The AND gate is drawn with the lines going through the center of the gate to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated wired connection. The logic function implemented by the circuit of Fig. 3-22(a) is

$$F = (AB)' \cdot (CD)' = (AB + CD)'$$

and is called an AND-OR-INVERT function.

Similarly, the NOR output of ECL gates can be tied together to perform a wired-OR function. The logic function implemented by the circuit of Fig. 3-22(b) is

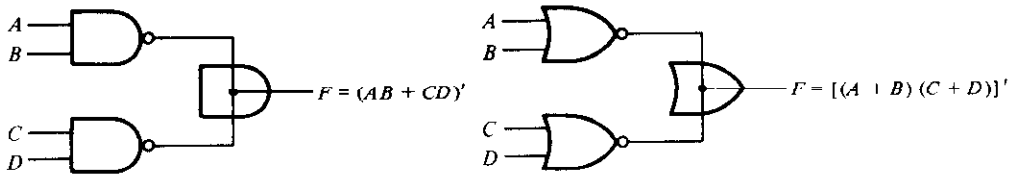
$$F = (A + B)' + (C + D)' = [(A + B)(C + D)]'$$

and is called an OR-AND-INVERT function.

A wired-logic gates does not produce a physical second-level gate since it is just a wire connection. Nevertheless, for discussion purposes, we will consider the circuits of Fig. 3-22 as two-level implementations. The first level consists of NAND (or NOR) gates and the second level has a single AND (or OR) gate. The wired connection in the graphic symbol will be omitted in subsequent discussions.

Nondegenerate Forms

It will be instructive from a theoretical point of view to find out how many two-level combinations of gates are possible. We consider four types of gates: AND, OR, NAND, and NOR. If we assign one type of gate for the first level and one type for the second



(a) Wired-AND in open-collector TTL NAND gates.

(AND-OR-INVERT)

(b) Wired-OR in ECL gates

(OR-AND-INVERT)

FIGURE 3-22
Wired logic

level, we find that there are 16 possible combinations of two-level forms. (The same type of gate can be in the first and second levels, as in NAND-NAND implementation.) Eight of these combinations are said to be *degenerate* forms because they degenerate to a single operation. This can be seen from a circuit with AND gates in the first level and an AND gate in the second level. The output of the circuit is merely the AND function of all input variables. The other eight *nondegenerate* forms produce an implementation in sum of products or product of sums. The eight nondegenerate forms are

AND-OR	OR-AND
NAND-NAND	NOR-NOR
NOR-OR	NAND-AND
OR-NAND	AND-NOR

The first gate listed in each of the forms constitutes a first level in the implementation. The second gate listed is a single gate placed in the second level. Note that any two forms listed in the same line are the duals of each other.

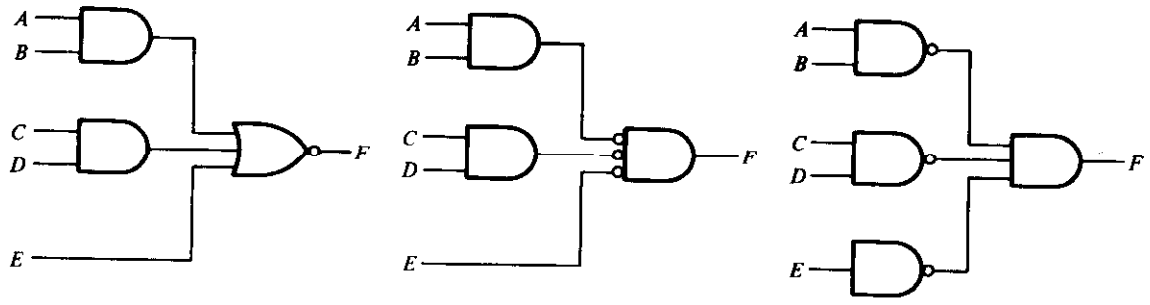
The AND-OR and OR-AND forms are the basic two-level forms discussed in Section 3-5. The NAND-NAND and NOR-NOR were introduced in Section 3-6. The remaining four forms are investigated in this section.

AND-OR-INVERT Implementation

The two forms NAND-AND and AND-NOR are equivalent forms and can be treated together. Both perform the AND-OR-INVERT function, as shown in Fig. 3-23. The AND-NOR form resembles the AND-OR form with an inversion done by the small circle in the output of the NOR gate. It implements the function

$$F = (AB + CD + E)'$$

By using the alternate graphic symbol for the NOR gate, we obtain the diagram of Fig. 3-23(b). Note that the single variable E is *not* complemented because the only change made is in the graphic symbol of the NOR gate. Now we move the circles from



(a) AND-NOR (b) AND-NOR (c) NAND-AND

FIGURE 3-23
AND-OR-INVERT circuits; $F = (AB + CD + E)'$

the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable to maintain the circle. Alternatively, the inverter can be removed provided input E is complemented. The circuit of Fig. 3-23(c) is a NAND-AND form and was shown in Fig. 3-22 to implement the AND-OR-INVERT function.

An AND-OR implementation requires an expression in sum of products. The AND-OR-INVERT implementation is similar except for the inversion. Therefore, if the *complement* of the function is simplified in sum of products (by combining the 0's in the map), it will be possible to implement F' with the AND-OR part of the function. When F' passes through the always present output inversion (the INVERT part), it will generate the ouput F of the function. An example for the AND-OR-INVERT implementation will be shown subsequently.

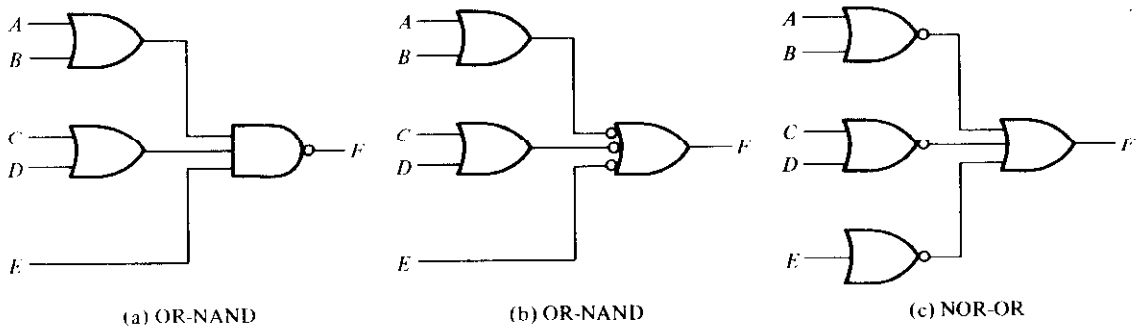
OR-AND-INVERT Implementation

The OR-NAND and NOR-OR forms perform the OR-AND-INVERT function. This is shown in Fig. 3-24. The OR-NAND form resembles the OR-AND form, except for the inversion done by the circle in the NAND gate. It implements the function

$$F = [(A + B)(C + D)E]'$$

By using the alternate graphic symbol for the NAND gate, we obtain the diagram of Fig. 3-24(b). The circuit in (c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates. The circuit of Fig. 3-24(c) is a NOR-OR form and was shown in Fig. 3-22 to implement the OR-AND-INVERT function.

The OR-AND-INVERT implementation requires an expression in product of sums. If the complement of the function is simplified in product of sums, we can implement F' with the OR-AND part of the function. When F' passes through the INVERT part, we obtain the complement of F', or F, in the output.



(a) OR-NAND (b) OR-NAND (c) NOR-OR

FIGURE 3-24
OR-AND-INVERT circuits; $F = [(A + B)(C + D)E]'$

Tabular Summary and Example

Table 3-4 summarizes the procedures for implementing a Boolean function in any one of the four two-level forms. Because of the INVERT part in each case, it is convenient to use the simplification of F' (the complement) of the function. When F' is implemented in one of these forms, we obtain the complement of the function in the AND-OR or OR-AND form. The four two-level forms invert this function, giving an output that is the complement of F'. This is the normal output F.

TABLE 3-4
Implementation with Other Two-Level Forms

Equivalent nondegenerate form		Implements the function	Simplify F' in	To get an output of
(a)	(b)*			
AND-NOR	NAND-AND	AND-OR-INVERT	Sum of products by combining 0's in the map	F
OR-NAND	NOR-OR	OR-AND-INVERT	Product of sums by combining 1's in the map and then complementing	F

*Form (b) requires a one-input NAND or NOR (inverter) gate for a single literal term.

Example 3-11

Implement the function of Fig. 3-19(a) with the four two-level forms listed in Table 3-4. The complement of the function is simplified in sum of products by combining the 0's in the map:

$$F' = x'y + xy' + z$$

The normal output for this function can be expressed as

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

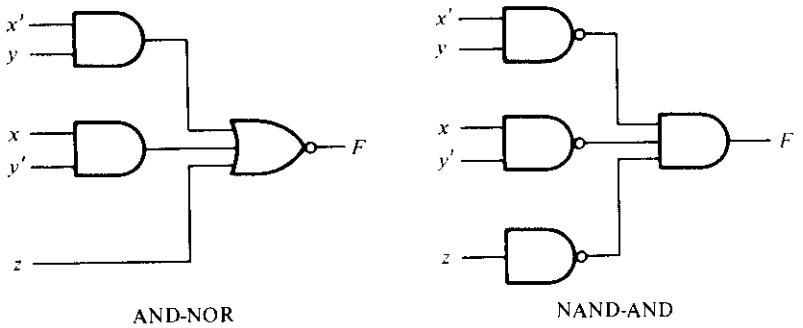
which is in the AND-OR-INVERT form. The AND-NOR and NAND-AND implementations are shown in Fig. 3-25(a). Note that a one-input NAND or inverter gate is needed in the NAND-AND implementation, but not in the AND-NOR case. The inverter can be removed if we apply the input variable z' instead of z.

The OR-AND-INVERT forms require a simplified expression of the complement of the function in product of sums. To obtain this expression, we must first combine the 1's in the map

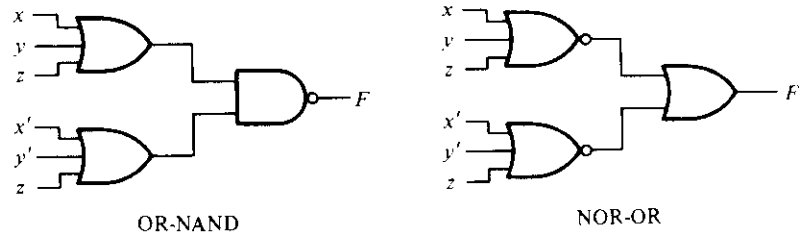
$$F = x'y'z' + xyz'$$

Then we take the complement of the function

$$F' = (x + y + z)(x' + y' + z)$$



(a) $F = (x'y + xy' + z)'$



(b) $F = [(x + y + z)(x' + y' + z)]'$

FIGURE 3-25
Other two-level implementations

The normal output F can now be expressed in the form

$$F = [(x + y + z)(x' + y' + z)]'$$

which is in the OR-AND-INVERT form. From this expression, we can implement the function in the OR-NAND and NOR-OR forms, as shown in Fig. 3-25(b). ■

3-8 DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This assumes that all the combinations of the values for the variables of the function are valid. In practice, there are some applications where the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered as unspecified. Functions that have unspecified outputs for some input combinations are called incompletely specified functions. In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function don't-care condi-

tions. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

It should be realized that a don't-care minterm is a combination of variables whose logical value is not specified. It cannot be marked with a 1 in the map because it would require that the function always be a 1 for such combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

When choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

Example 3-12

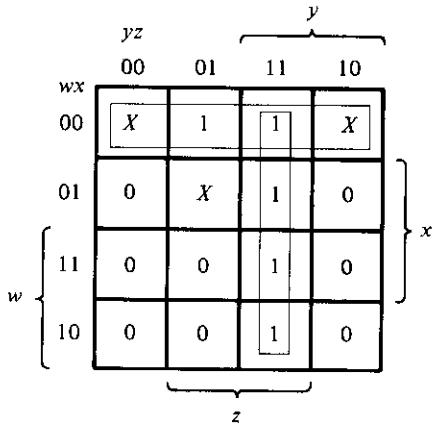
Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$$

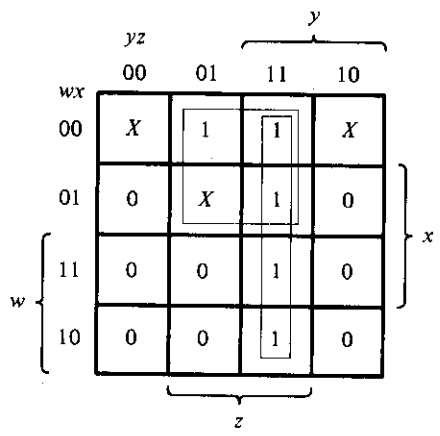
that has the don't-care conditions

$$d(w, x, y, z) = \Sigma(0, 2, 5)$$

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 3-26. The minterms of F are marked by 1's, those of d are marked by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum of products, we must include all the five 1's in the map, but



(a) $F = yz + w'x'$



(b) $F = yz + w'z$

FIGURE 3-26
Example with don't-care conditions

we may or may not include any of the X's, depending on the way the function is simplified. The term yz covers the four minterms in the third column. The remaining minterm m_1 can be combined with minterm m_3 to give the three-literal term $w'x'z$. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In part (a) of the diagram, don't-care minterms 0 and 2 are included with the 1's, which results in the simplified function

$$F = yz + w'x'$$

In part (b), don't-care minterm 5 is included with the 1's and the simplified function now is

$$F = yz + w'z$$

Either one of the above expressions satisfies the conditions stated for this example. ■

The above example has shown that the don't-care minterms in the map are initially marked with X's and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function so obtained will consist of a sum of minterms that includes those minterms that were initially unspecified and have been chosen to be included with the 1's. Consider the two simplified expressions obtained in Example 3-12:

$$F(w, x, y, z) = yz + w'x' = \Sigma(0, 1, 2, 3, 7, 11, 15)$$
$$F(w, x, y, z) = yz + w'z = \Sigma(1, 3, 5, 7, 11, 15)$$

Both expressions include minterms 1, 3, 7, 11, and 15 that make the function F equal to 1. The don't-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's. The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's. The two expressions represent two functions that are algebraically unequal. Both cover the specified minterms of the function, but each covers different don't-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable since the only difference is in the value of F for the don't-care minterms.

It is also possible to obtain a simplified product of sums expression for the function of Fig. 3-26. In this case, the only way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's to give a simplified complemented function:

$$F' = z' + wy'$$

Taking the complement of F' gives the simplified expression in product of sums:

$$F(w, x, y, z) = z(w' + y) = \Sigma(1, 3, 5, 7, 11, 15)$$

For this case, we include minterms 0 and 2 with the 0's and minterm 5 with the 1's.

3-9 THE TABULATION METHOD

The map method of simplification is convenient as long as the number of variables does not exceed five or six. As the number of variables increases, the excessive number of squares prevents a reasonable selection of adjacent squares. The obvious disadvantage of the map is that it is essentially a trial-and-error procedure that relies on the ability of the human user to recognize certain patterns. For functions of six or more variables, it is difficult to be sure that the best selection has been made.

The tabulation method overcomes this difficulty. It is a specific step-by-step procedure that is guaranteed to produce a simplified standard-form expression for a function. It can be applied to problems with many variables and has the advantage of being suitable for machine computation. However, it is quite tedious for human use and is prone to mistakes because of its routine, monotonous process. The tabulation method was first formulated by Quine and later improved by McCluskey. It is also known as the Quine–McCluskey method.

The tabular method of simplification consists of two parts. The first is to find by an exhaustive search all the terms that are candidates for inclusion in the simplified function. These terms are called *prime implicants*. The second operation is to choose among the prime implicants those that give an expression with the least number of literals.

3-10 DETERMINATION OF PRIME IMPLICANTS

The starting point of the tabulation method is the list of minterms that specify the function. The first tabular operation is to find the prime implicants by using a matching process. This process compares each minterm with every other minterm. If two minterms differ in only one variable, that variable is removed and a term with one less literal is found. This process is repeated for every minterm until the exhaustive search is completed. The matching-process cycle is repeated for those new terms just found. Third and further cycles are continued until a single pass through a cycle yields no further elimination of literals. The remaining terms and all the terms that did not match during the process comprise the prime implicants. This tabulation method is illustrated by the following example.

Example 3-13 Simplify the following Boolean function by using the tabulation method:

$$F = \Sigma(0, 1, 2, 8, 10, 11, 14, 15)$$

Step 1: Group binary representation of the minterms according to the number of 1's contained, as shown in Table 3-5, column (a). This is done by grouping the minterms into five sections separated by horizontal lines. The first section contains the number with no 1's in it. The second section contains those numbers that have only one 1. The

TABLE 3-5
Determination of Prime Implicants for Example 3-13

(a)		(b)		(c)	
	w x y z		*w x y z*		*w x y z*
0	0 0 0 0 ✓	0, 1	0 0 0 −	0, 2, 8, 10	− 0 − 0
		0, 2	0 0 − 0 ✓	0, 8, 2, 10	− 0 − 0
1	0 0 0 1 ✓	0, 8	− 0 0 0 ✓	10, 11, 14, 15	1 − 1 −
2	0 0 1 0 ✓			10, 14, 11, 15	1 − 1 −
8	1 0 0 0 ✓	2, 10	− 0 1 0 ✓		
		8, 10	1 0 − 0 ✓		
10	1 0 1 0 ✓				
		10, 11	1 0 1 − ✓		
11	1 0 1 1 ✓	10, 14	1 − 1 − ✓		
14	1 1 1 0 ✓				
		11, 15	1 − 1 1 ✓		
15	1 1 1 1 ✓	14, 15	1 1 1 − ✓		

third, fourth, and fifth sections contain those binary numbers with two, three, and four 1's, respectively. The decimal equivalents of the minterms are also carried along for identification.

Step 2: Any two minterms that differ from each other by only one variable can be combined, and the unmatched variable removed. Two minterm numbers fit into this category if they both have the same bit value in all positions except one. The minterms in one section are compared with those of the next section down only, because two terms differing by more than one bit cannot match. The minterm in the first section is compared with each of the three minterms in the second section. If any two numbers are the same in every position but one, a check is placed to the right of both minterms to show that they have been used. The resulting term, together with the decimal equivalents, is listed in column (b) of the table. The variable eliminated during the matching is denoted by a dash in its original position. In this case, m_0 (0000) combines with m_1 (0001) to form (000–). This combination is equivalent to the algebraic operation

$$m_0 + m_1 = w'x'y'z' + w'x'y'z = w'x'y'$$

Minterm m_0 also combines with m_2 to form (00–0) and with m_8 to form (–000). The result of this comparison is entered into the first section of column (b). The minterms of sections two and three of column (a) are next compared to produce the terms listed in the second section of column (b). All other sections of (a) are similarly compared and subsequent sections formed in (b). This exhaustive comparing process results in the four sections of (b).

Step 3: The terms of column (b) have only three variables. A 1 under the variable means it is unprimed, a 0 means it is primed, and a dash means the variable is not included in the term. The searching and comparing process is repeated for the terms in

column (b) to form the two-variable terms of column (c). Again, terms in each section need to be compared only if they have dashes in the same position. Note that the term (000–) does not match with any other term. Therefore, it has no check mark at its right. The decimal equivalents are written on the left-hand side of each entry for identification purposes. The comparing process should be carried out again in column (c) and in subsequent columns as long as proper matching is encountered. In the present example, the operation stops at the third column.

Step 4: The unchecked terms in the table form the prime implicants. In this example, we have the term $w'x'y'$ (000–) in column (b), and the terms $x'z'$ (–0–0) and wy (1–1–) in column (c). Note that each term in column (c) appears twice in the table, and as long as the term forms a prime implicant, it is unnecessary to use the same term twice. The sum of the prime implicants gives a simplified expression for the function. This is because each checked term in the table has been taken into account by an entry of a simpler term in a subsequent column. Therefore, the unchecked entries (prime implicants) are the terms left to formulate the function. For the present example, the sum of prime implicants gives the minimized function in sum of products:

$$F = w'x'y' + x'z' + wy$$ ■

It is worth comparing this answer with that obtained by the map method. Figure 3-27 shows the map simplification of this function. The combinations of adjacent squares give the three prime implicants of the function. The sum of these three terms is the simplified expression in sum of products.

It is important to point out that Example 3-13 was purposely chosen to give the simplified function from the sum of prime implicants. In most other cases, the sum of prime implicants does not necessarily form the expression with the minimum number of terms. This is demonstrated in Example 3-14.

The tedious manipulation that one must undergo when using the tabulation method is reduced if the comparing is done with decimal numbers instead of binary. A method will now be shown that uses subtraction of decimal numbers instead of the comparing and matching of binary numbers. We note that each 1 in a binary number represents the

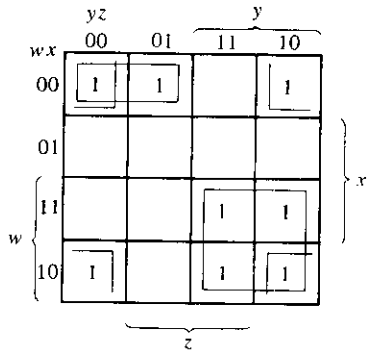


FIGURE 3-27
Map for the function of Example 3-13;
$F = w'x'y' + x'z' + wy$

coefficient multiplied by a power of 2. When two minterms are the same in every position except one, the minterm with the extra 1 must be larger than the number of the other minterm by a power of 2. Therefore, two minterms can be combined if the number of the first minterm differs by a power of 2 from a second larger number in the next section down the table. We shall illustrate this procedure by repeating Example 3-13.

As shown in Table 3-6, column (a), the minterms are arranged in sections as before, except that now only the decimal equivalents of the minterms are listed. The process of comparing minterms is as follows: Inspect every two decimal numbers in adjacent sections of the table. If the number in the section below is *greater* than the number in the section above by a power of 2 (i.e., 1, 2, 4, 8, 16, etc.), check both numbers to show that they have been used, and write them down in column (b). The pair of numbers transferred to column (b) includes a third number in parentheses that designates the power of 2 by which the numbers differ. The number in parentheses tells us the position of the dash in the binary notation. The results of all comparisons of column (a) are shown in column (b).

The comparison between adjacent sections in column (b) is carried out in a similar fashion, except that only those terms with the same number in parentheses are compared. The pair of numbers in one section must differ by a power of 2 from the pair of numbers in the next section. And the numbers in the next section below must be *greater* for the combination to take place. In column (c), write all four decimal numbers with the two numbers in parentheses designating the positions of the dashes. A comparison of Tables 3-5 and 3-6 may be helpful in understanding the derivations in Table 3-6.

TABLE 3-6
Determination of Prime Implicants of Example 3-13 with Decimal Notation

(a)		(b)		(c)	
0	√	0, 1 (1)		0, 2, 8, 10	(2, 8)
		0, 2 (2)	√	0, 2, 8, 10	(2, 8)
1	√	0, 8 (8)	√		
2	√			10, 11, 14, 15	(1, 4)
8	√	2, 10 (8)	√	10, 11, 14, 15	(1, 4)
		8, 10 (2)	√		
10	√				
		10, 11 (1)	√		
11	√	10, 14 (4)	√		
14	√				
		11, 15 (4)	√		
15	√	14, 15 (1)	√		

The prime implicants are those terms not checked in the table. These are the same as before, except that they are given in decimal notation. To convert from decimal notation to binary, convert all decimal numbers in the term to binary and then insert a dash in those positions designated by the numbers in parentheses. Thus 0, 1 (1) is converted to binary as 0000, 0001; a dash in the first position of either number results in (000–). Similarly, 0, 2, 8, 10 (2, 8) is converted to the binary notation from 0000, 0010, 1000, and 1010, and a dash inserted in positions 2 and 8, to result in (–0–0).

Example 3-14

Determine the prime implicants of the function

$$F(w, x, y, z) = \Sigma(1, 4, 6, 7, 8, 9, 10, 11, 15)$$

The minterm numbers are grouped in sections, as shown in Table 3-7, column (a). The binary equivalent of the minterm is included for the purpose of counting the number of

TABLE 3-7
Determination of Prime Implicants for Example 3-14

(a)			(b)			(c)
0001	1	√	1, 9	(8)		8, 9, 10, 11 (1, 2)
0100	4	√	4, 6	(2)		8, 9, 10, 11 (1, 2)
1000	8	√	8, 9	(1)	√	
			8, 10	(2)	√	
0110	6	√				
1001	9	√	6, 7	(1)		
1010	10	√	9, 11	(2)	√	
			10, 11	(1)	√	
0111	7	√				
1011	11	√	7, 15	(8)		
			11, 15	(4)		
1111	15	√				

Prime implicants

Decimal	Binary w	x	y	z	Term
1, 9 (8)	–	0	0	1	$x'y'z$
4, 6 (2)	0	1	–	0	$w'xz'$
6, 7 (1)	0	1	1	–	$w'xy$
7, 15 (8)	–	1	1	1	xyz
11, 15 (4)	1	–	1	1	wyz
8, 9, 10, 11 (1, 2)	1	0	–	–	wx'

1's. The binary numbers in the first section have only one 1; in the second section, two 1's; etc. The minterm numbers are compared by the decimal method and a match is found if the number in the section below is greater than that in the section above. If the number in the section below is smaller than the one above, a match is not recorded even if the two numbers differ by a power of 2. The exhaustive search in column (a) results in the terms of column (b), with all minterms in column (a) being checked. There are only two matches of terms in column (b). Each gives the same two-literal term recorded in column (c). The prime implicants consist of all the unchecked terms in the table. The conversion from the decimal to the binary notation is shown at the bottom of the table. The prime implicants are found to be $x'y'z$, $w'xz'$, $w'xy$, xyz, wyz, and wx'. ■

The sum of the prime implicants gives a valid algebraic expression for the function. However, this expression is not necessarily the one with the minimum number of terms. This can be demonstrated from inspection of the map for the function of Example 3-14. As shown in Fig. 3-28, the minimized function is recognized to be

$$F = x'y'z + w'xz' + xyz + wx'$$

which consists of the sum of four of the six prime implicants derived in Example 3-14. The tabular procedure for selecting the prime implicants that give the minimized function is the subject of the next section.

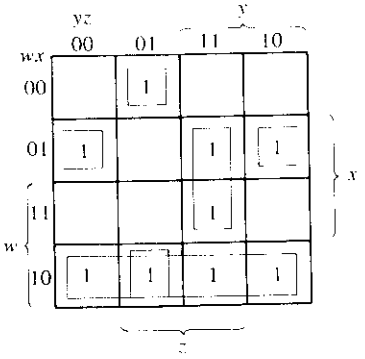


FIGURE 3-28
Map for the function of Example 3-14;
$F = x'y'z' + w'xz' + xyz + wx'$

3-11 SELECTION OF PRIME IMPLICANTS

The selection of prime implicants that form the minimized function is made from a prime implicant table. In this table, each prime implicant is represented in a row and each minterm in a column. X's are placed in each row to show the composition of

minterms that make the prime implicants. A minimum set of prime implicants is then chosen that covers all the minterms in the function. This procedure is illustrated in Example 3-15.

Example 3-15

Minimize the function of Example 3-14. The prime-implicant table for this example is shown in Table 3-8. There are six rows, one for each prime implicant (derived in Example 3-14), and nine columns, each representing one minterm of the function. X's are placed in each row to indicate the minterms contained in the prime implicant of that row. For example, the two X's in the first row indicate that minterms 1 and 9 are contained in the prime implicant $x'y'z$. It is advisable to include the decimal equivalent of the prime implicant in each row, as it conveniently gives the minterms contained in it. After all the X's have been marked, we proceed to select a minimum number of prime implicants.

The completed prime-implicant table is inspected for columns containing only a single X. In this example, there are four minterms whose columns have a single X: 1, 4, 8, and 10. Minterm 1 is covered by prime implicant $x'y'z$, i.e., the selection of prime implicant $x'y'z$ guarantees that minterm 1 is included in the function. Similarly, minterm 4 is covered by prime implicant $w'xz'$, and minterms 8 and 10, by prime implicant wx'. Prime implicants that cover minterms with a single X in their column are called *essential prime implicants*. To enable the final simplified expression to contain all the minterms, we have no alternative but to include essential prime implicants. A check mark is placed in the table next to the essential prime implicants to indicate that they have been selected.

Next we check each column whose minterm is covered by the selected essential prime implicants. For example, the selected prime implicant $x'y'z$ covers minterms 1 and 9. A check is inserted in the bottom of the columns. Similarly, prime implicant $w'xz'$ covers minterms 4 and 6, and wx' covers minterms 8, 9, 10, and 11. Inspection of the prime-implicant table shows that the selection of the essential prime implicants

TABLE 3-8
Prime Implicant Table for Example 3-15

		1	4	6	7	8	9	10	11	15
✓ $x'y'z$	1, 9	X					X			
✓ $w'xz'$	4, 6		X	X						
$w'xy$	6, 7			X	X					
xyz	7, 15				X					X
wyz	11, 15								X	X
✓ wx'	8, 9, 10, 11					X	X	X	X	
		✓	✓	✓		✓	✓	✓	✓	

covers all the minterms of the function except 7 and 15. These two minterms must be included by the selection of one or more prime implicants. In this example, it is clear that prime implicant xyz covers both minterms and is therefore the one to be selected. We have thus found the minimum set of prime implicants whose sum gives the required minimized function:

$$F = x'y'z + w'xz' + wx' + xyz$$ ■

The simplified expressions derived in the preceding examples were all in the sum of products form. The tabulation method can be adapted to give a simplified expression in product of sums. As in the map method, we have to start with the complement of the function by taking the 0's as the initial list of minterms. This list contains those minterms not included in the original function that are numerically equal to the maxterms of the function. The tabulation process is carried out with the 0's of the function and terminates with a simplified expression in sum of products of the complement of the function. By taking the complement again, we obtain the simplified product of sums expression.

A function with don't-care conditions can be simplified by the tabulation method after a slight modification. The don't-care terms are included in the list of minterms when the prime implicants are determined. This allows the derivation of prime implicants with the least number of literals. The don't-care terms are not included in the list of minterms when the prime implicant table is set up, because don't-care terms do not have to be covered by the selected prime implicants.

3-12 CONCLUDING REMARKS

Two methods of Boolean-function simplification were introduced in this chapter. The criterion for simplification was taken to be the minimization of the number of literals in sum of product or products of sums expressions. Both the map and the tabulation methods are restricted in their capabilities since they are useful for simplifying only Boolean functions expressed in the standard forms. Although this is a disadvantage of the methods, it is not very critical. Most applications prefer the standard forms over any other form. We have seen from Fig. 3-15 that the gate implementation of expressions in standard form consists of no more than two levels of gates. Expressions not in the standard form are implemented with more than two levels.

One should recognize that the Gray-code sequence chosen for the maps is not unique. It is possible to draw a map and assign a Gray-code sequence to the rows and columns different from the sequence employed here. As long as the binary sequence chosen produces a change in only one bit between adjacent squares, it will produce a valid and useful map.

Two alternate versions of the three-variable maps that are often found in the digital

logic literature are shown in Fig. 3-29. The minterm numbers are written in each square for reference. In (a), the assignment of the variables to the rows and columns is different from the one used in this book. In (b), the map has been rotated in a vertical position. The minterm number assignment in all maps remains in the order xyz. For example, the square for minterm 6 is found by assigning to the ordered variables the binary number $xyz = 110$. The square for this minterm is found in (a) from the column marked $xy = 11$ and the row with $z = 0$. The corresponding square in (b) belongs in the column marked with $x = 1$ and the row with $yz = 10$. The simplification procedure with these maps is exactly the same as described in this chapter except, of course, for the variations in minterm and variable assignment.

Two other versions of the four-variable map are shown in Fig. 3-30. The map in (a) is very popular and is used quite often in the literature. Here again, the difference is

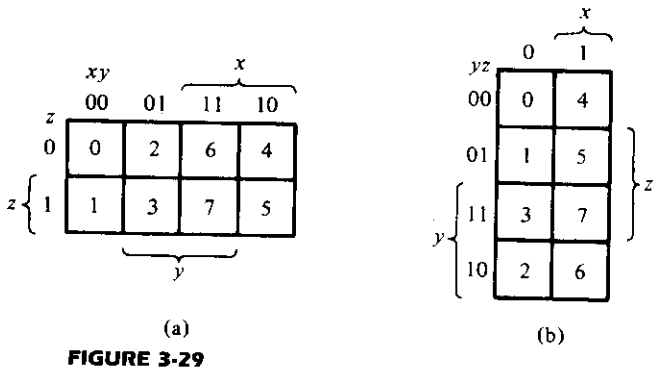


FIGURE 3-29
Variations of the three-variable map

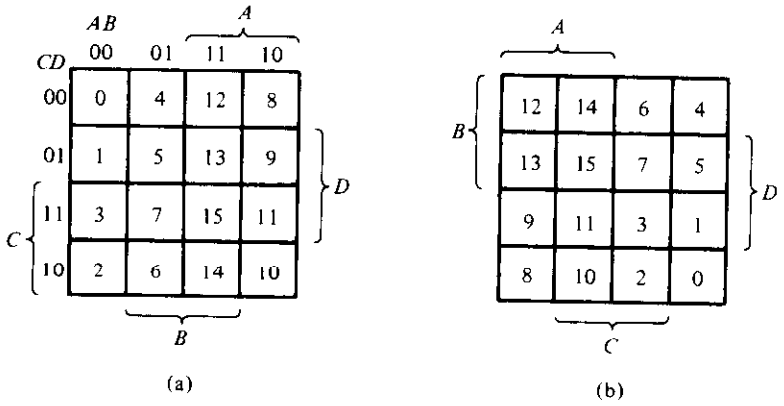


FIGURE 3-30
Variations of the four-variable map

slight and is manifested by a mere interchange of variable assignment from rows to columns and vice versa. The map in (b) is the original Veitch diagram that Karnaugh modified to the one shown in (a). Again, the simplification procedures do not change when these maps are used instead of the one employed in this book. There are also variations of the five-variable map. In any case, any map that looks different from the one used in this book, or is called by a different name, should be recognized merely as a variation of minterm assignment to the squares in the map.

As is evident from Examples 3-13 and 3-14, the tabulation method has the drawback that errors inevitably occur in trying to compare numbers over long lists. The map method would seem to be preferable, but for more than five variables, we cannot be certain that the best simplified expression has been found. The real advantage of the tabulation method lies in the fact that it consists of specific step-by-step procedures that guarantee an answer. Moreover, this formal procedure is suitable for computer mechanization.

In this chapter, we have considered the simplification of functions with many input variables and a single output variable. However, some digital circuits have more than one output. Such circuits are described by a set of Boolean functions, one for each output variable. A circuit with multiple outputs may sometimes have common terms among the various functions that can be utilized to form common gates during the implementation. This results in further simplification not taken into consideration when each function is simplified separately. There exists an extension of the tabulation method for multiple-output circuits. However, this method is too specialized and very tedious for human manipulation. It is of practical importance only if a computer program based on this method is available to the user.

REFERENCES

1. VEITCH, E. W., "A Chart Method for Simplifying Truth Functions." *Proc. ACM* (May 1952), 127–133.
2. KARNAUGH, M., "A Map Method for Synthesis of Combinational Logic Circuits." *Trans. AIEE, Comm. and Electron.*, **72,** Part I (November 1953), 593–599.
3. QUINE, W. V., "The Problem of Simplifying Truth Functions." *Am. Math. Monthly*, **59 (8)** (October 1952), 521–531.
4. MCCLUSKEY, E. J., "Minimization of Boolean Functions." *Bell Syst. Tech. J.*, **35 (6)** (November 1956), 1417–1444.
5. MCCLUSKEY, E. J., *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
6. KOHAVI, Z., *Switching and Automata Theory*, 2nd Ed. New York: McGraw-Hill, 1978.
7. HILL, F. J., and G. R. PETERSON, *Introduction to Switching Theory and Logical Design*, 3rd Ed. New York: John Wiley, 1981.
8. GIVONE, D. D., *Introduction to Switching Circuit Theory*. New York: McGraw-Hill, 1970.

PROBLEMS

3-1 Simplify the following Boolean functions using three-variable maps:
(a) $F(x, y, z) = \Sigma(0, 1, 5, 7)$
(b) $F(x, y, z) = \Sigma(1, 2, 3, 6, 7)$
(c) $F(x, y, z) = \Sigma(3, 5, 6, 7)$
(d) $F(A, B, C) = \Sigma(0, 2, 3, 4, 6)$

3-2 Simplify the following Boolean expressions using three-variable maps:
(a) $xy + x'y'z' + x'yz'$
(b) $x'y' + yz + x'yz'$
(c) $A'B + BC' + B'C'$

3-3 Simplify the following Boolean functions using four-variable maps:
(a) $F(A, B, C, D) = \Sigma(4, 6, 7, 15)$
(b) $F(w, x, y, z) = \Sigma(2, 3, 12, 13, 14, 15)$
(c) $F(A, B, C, D) = \Sigma(3, 7, 11, 13, 14, 15)$

3-4 Simplify the following Boolean functions using four-variable maps:
(a) $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 12, 14, 15)$
(b) $F(A, B, C, D) = \Sigma(0, 1, 2, 4, 5, 7, 11, 15)$
(c) $F(w, x, y, z) = \Sigma(2, 3, 10, 11, 12, 13, 14, 15)$
(d) $F(A, B, C, D) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

3-5 Simplify the following Boolean expressions using four-variable maps:
(a) $w'z + xz + x'y + wx'z$
(b) $B'D + A'BC' + AB'C + ABC'$
(c) $AB'C + B'C'D' + BCD + ACD' + A'B'C + A'BC'D$
(d) $wxy + yz + xy'z + x'y$

3-6 Find the minterms of the following Boolean expressions by first plotting each function in a map:
(a) $xy + yz + xy'z$
(b) $C'D + ABC' + ABD' + A'B'D$
(c) $wxy + x'z' + w'xz$

3-7 Simplify the following Boolean functions by first finding the essential prime implicants:
(a) $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
(b) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c) $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$

3-8 Simplify the following Boolean functions using five-variable maps:
(a) $F(A, B, C, D, E) = \Sigma(0, 1, 4, 5, 16, 17, 21, 25, 29)$
(b) $F(A, B, C, D, E) = \Sigma(0, 2, 3, 4, 5, 6, 7, 11, 15, 16, 18, 19, 23, 27, 31)$
(c) $F = A'B'CE' + A'B'C'D' + B'D'E' + B'CD' + CDE' + BDE'$

3-9 Simplify the following Boolean functions in product of sums:
(a) $F(w, x, y, z) = \Sigma(0, 2, 5, 6, 7, 8, 10)$
(b) $F(A, B, C, D) = \Pi(1, 3, 5, 7, 13, 15)$
(c) $F(x, y, z) = \Sigma(2, 3, 6, 7)$
(d) $F(A, B, C, D) = \Pi(0, 1, 2, 3, 4, 10, 11)$

3-10 Simplify the following expressions in (i) sum of products and (ii) products of sums:
(a) $x'z' + y'z' + yz' + xy$
(b) $AC' + B'D + A'CD + ABCD$
(c) $(A' + B' + D')(A + B' + C')(A' + B + D')(B + C' + D')$

3-11 Draw the AND-OR gate implementation of the following function after simplifying it in (a) sum of products and (b) product of sums:

$$F = (A, B, C, D) = \Sigma(0, 2, 5, 6, 7, 8, 10)$$

3-12 Simplify the following expressions and implement them with two-level NAND gate circuits:
(a) $AB' + ABD + ABD' + A'C'D' + A'BC'$
(b) $BD + BCD' + AB'C'D'$

3-13 Draw a NAND logic diagram that implements the complement of the following function:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 3, 4, 8, 9, 12)$$

3-14 Draw a logic diagram using only two-input NAND gates to implement the following expression:

$$(AB + A'B')(CD' + C'D)$$

3-15 Simplify the following functions and implement them with two-level NOR gate circuits:
(a) $F = wx' + y'z' + w'yz'$
(b) $F(w, x, y, z) = \Sigma(5, 6, 9, 10)$

3-16 Implement the functions of Problem 3-15 with three-level NOR gate circuits [similar to Fig. 3-21(b)].

3-17 Implement the expressions of Problem 3-12 with three-level NAND circuits [similar to Fig. 3-19(c)].

3-18 Give three possible ways to express the function F with eight or fewer literals.

$$F(A, B, C, D) = \Sigma(0, 2, 5, 7, 10, 13)$$

3-19 Find eight different two-level gate circuits to implement

$$F = xy'z + x'yz + w$$

3-20 Implement the function F with the following two-level forms: NAND-AND, AND-NOR, OR-NAND, and NOR-OR.

$$F(A,B,C,D) = \Sigma(0, 1, 2, 3, 4, 8, 9, 12)$$

3-21 List the eight degenerate two-level forms and show that they reduce to a single operation. Explain how the degenerate two-level forms can be used to extend the number of inputs to a gate.

3-22 Simplify the following Boolean function F together with the don't-care conditions d; then express the simplified function in sum of minterms.
(a) $F(x, y, z) = \Sigma(0, 1, 2, 4, 5)$
$d(x, y, z) = \Sigma(3, 6, 7)$

(b) $F(A, B, C, D) = \Sigma(0, 6, 8, 13, 14)$
$d(A, B, C, D) = \Sigma(2, 4, 10)$
(c) $F(A, B, C, D) = \Sigma(1, 3, 5, 7, 9, 15)$
$d(A, B, C, D) = \Sigma(4, 6, 12, 13)$

3-23 Simplify the Boolean function F together with the don't-care conditions d in (i) sum of products and (ii) product of sums.
(a) $F(w, x, y, z) = \Sigma(0, 1, 2, 3, 7, 8, 10)$
$d(w, x, y, z) = \Sigma(5, 6, 11, 15)$
(b) $F(A, B, C, D) = \Sigma(3, 4, 13, 15)$
$d(A, B, C, D) = \Sigma(1, 2, 5, 6, 8, 10, 12, 14)$

3-24 A logic circuit implements the following Boolean function:

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

It is found that the circuit input combination $A = C = 1$ can never occur. Find a simpler expression for F using the proper don't-care conditions.

3-25 Implement the following Boolean function F together with the don't-care conditions d using no more than two NOR gates. Assume that both the normal and complement inputs are available.

$$F(A, B, C, D) = \Sigma(0, 1, 2, 9, 11)$$

$$d(A, B, C, D) = \Sigma(8, 10, 14, 15)$$

3-26 Simplify the following Boolean function using the map presented in Fig. 3-30(a). Repeat using the map of Fig. 3-30(b).

$$F(A, B, C, D) = \Sigma(1, 2, 3, 5, 7, 9, 10, 11, 13, 15)$$

3-27 Simplify the following Boolean functions by means of the tabulation method:
(a) $P(A, B, C, D, E, F, G) = \Sigma(20, 28, 52, 60)$
(b) $P(A, B, C, D, E, F, G) = \Sigma(20, 28, 38, 39, 52, 60, 102, 103, 127)$
(c) $P(A, B, C, D, E, F) = \Sigma(6, 9, 13, 18, 19, 25, 27, 29, 41, 45, 57, 61)$

Combinational Logic

4-1 INTRODUCTION

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs. A combinational circuit performs a specific information-processing operation fully specified logically by a set of Boolean functions. Sequential circuits employ memory elements (binary cells) in addition to logic gates. Their outputs are a function of the inputs and the state of the memory elements. The state of memory elements, in turn, is a function of previous inputs. As a consequence, the outputs of a sequential circuit depend not only on present inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states. Sequential circuits are discussed in Chapter 6.

In Chapter 1, we learned to recognize binary numbers and binary codes that represent discrete quantities of information. These binary variables are represented by electric voltages or by some other signal. The signals can be manipulated in digital logic gates to perform required functions. In Chapter 2, we introduced Boolean algebra as a way to express logic functions algebraically. In Chapter 3, we learned how to simplify Boolean functions to achieve economical gate implementations. The purpose of this chapter is to use the knowledge acquired in previous chapters and formulate various systematic design and analysis procedures of combinational circuits. The solution of some typical examples will provide a useful catalog of elementary functions important for the understanding of digital computers and systems.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from the inputs and generate signals to the outputs. This process transforms binary information from the given input data to the required output data. Obviously, both input and output data are represented by binary signals, i.e., they exist in two possible values, one representing logic-1 and the other logic-0. A block diagram of a combinational circuit is shown in Fig. 4-1. The n input binary variables come from an external source; the m output variables go to an external destination. In many applications, the source and/or destination are storage registers (Section 1-7) located either in the vicinity of the combinational circuit or in a remote external device. By definition, an external register does not influence the behavior of the combinational circuit because, if it does, the total system becomes a sequential circuit.

For n input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

Each input variable to a combinational circuit may have one or two wires. When only one wire is available, it may represent the variable either in the normal form (unprimed) or in the complement form (primed). Since a variable in a Boolean expression may appear primed and/or unprimed, it is necessary to provide an inverter for each literal not available in the input wire. On the other hand, an input variable may appear in two wires, supplying both the normal and complement forms to the input of the circuit. If so, it is unnecessary to include inverters for the inputs. The type of binary cells used in most digital systems are flip-flop circuits (Chapter 6) that have outputs for both the normal and complement values of the stored binary variable. In our subsequent work, we shall assume that each input variable appears in two wires, supplying both the normal and complement values simultaneously. We must also realize that an inverter circuit can always supply the complement of the variable if only one wire is available.

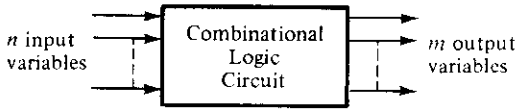


FIGURE 4-1
Block diagram of a combinational circuit

4-2 DESIGN PROCEDURE

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be easily obtained. The procedure involves the following steps:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.

3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

A truth table for a combinational circuit consists of input columns and output columns. The 1's and 0's in the input columns are obtained from the 2^n binary combinations available for n input variables. The binary values for the outputs are determined from examination of the stated problem. An output can be equal to either 0 or 1 for every valid input combination. However, the specifications may indicate that some input combinations will not occur. These combinations become don't-care conditions.

The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly into a truth table. Sometimes the designer must use intuition and experience to arrive at the correct interpretation. Word specifications are very seldom complete and exact. Any wrong interpretation that results in an incorrect truth table produces a combinational circuit that will not fulfill the stated requirements.

The output Boolean functions from the truth table are simplified by any available method, such as algebraic manipulation, the map method, or the tabulation procedure. Usually, there will be a variety of simplified expressions from which to choose. However, in any particular application, certain restrictions, limitations, and criteria will serve as a guide in the process of choosing a particular algebraic expression. A practical design method would have to consider such constraints as (1) minimum number of gates, (2) minimum number of inputs to a gate, (3) minimum propagation time of the signal through the circuit, (4) minimum number of interconnections, and (5) limitations of the driving capabilities of each gate. Since all these criteria cannot be satisfied simultaneously, and since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement as to what constitutes an acceptable simplification. In most cases, the simplification begins by satisfying an elementary objective, such as producing a simplified Boolean function in a standard form, and from that proceeds to meet any other performance criteria.

In practice, designers tend to go from the Boolean functions to a wiring list that shows the interconnections among various standard logic gates. In that case, the design need not go any further than the required simplified output Boolean functions. However, a logic diagram is helpful for visualizing the gate implementation of the expressions.

4-3 ADDERS

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits. This simple addition consists

of four possible elementary operations, namely, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the 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 a *full-adder*. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder. The two adder circuits are the first combinational circuits we shall design.

Half-Adder

From the verbal explanation of a half-adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. It is necessary to specify two output variables because the result may consist of two binary digits. We arbitrarily assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.

Now that we have established the number and names of the input and output variables, we are ready to formulate a truth table to identify exactly the function of the half-adder. This truth table is

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

The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are

$$S = x'y + xy'$$

$$C = xy$$

The logic diagram for this implementation is shown in Fig. 4-2(a), as are four other implementations for a half-adder. They all achieve the same result as far as the input–output behavior is concerned. They illustrate the flexibility available to the designer when implementing even a simple combinational logic function such as this.

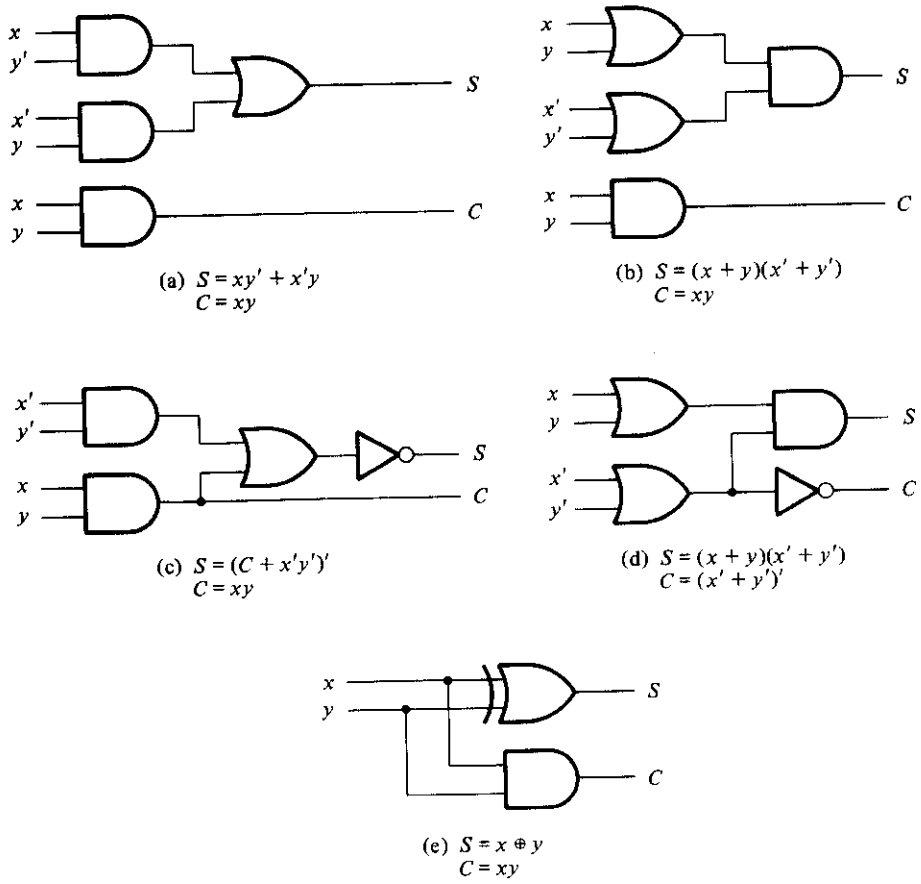


FIGURE 4-2
Various implementations of a half-adder

Figure 4-2(a), as mentioned before, is the implementation of the half-adder in sum of products. Figure 4-2(b) shows the implementation in product of sums:

$$S = (x + y)(x' + y')$$

$$C = xy$$

To obtain the implementation of Fig. 4-2(c), we note that S is the exclusive-OR of x and y. The complement of S is the equivalence of x and y (Section 2-6.):

$$S' = xy + x'y'$$

but $C = xy$, and, therefore, we have

$$S = (C + x'y')'$$

In Fig. 4-2(d), we use the product of sums implementation with C derived as follows:

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

The half-adder can be implemented with an exclusive-OR and an AND gate, as shown in Fig. 4-2(e). This form is used later to show that two half-adder circuits are needed to construct a full-adder circuit.

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

x	y	z	C	S
0	0	0	0	0
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

The eight rows under the input variables designate all possible combinations of 1's and 0's that these variables may have. The 1's and 0's for the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0's, 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 input and output bits of the combinational circuit have different interpretations at various stages of the problem. Physically, the binary signals of the input wires are considered binary digits added arithmetically to form a two-digit sum at the output wires. On the other hand, the same binary values are considered variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. It is important to realize that two different interpretations are given to the values of the bits encountered in this circuit.

The input-output logical relationship of the full-adder circuit may be expressed in two Boolean functions, one for each output variable. Each output Boolean function re-

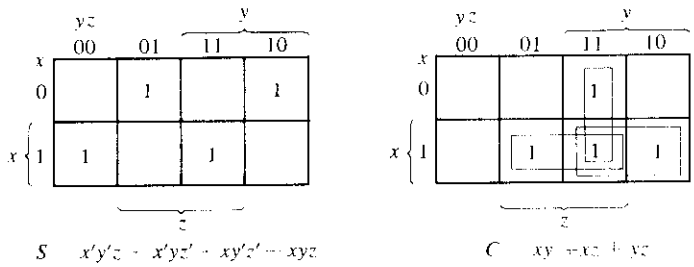


FIGURE 4-3
Maps for a full-adder

quires a unique map for its simplification. Each map must have eight squares, since each output is a function of three input variables. The maps of Fig. 4-3 are used for simplifying the two output functions. The 1's in the squares for the maps of S and C are determined directly from the truth table. The squares with 1's for the S output do not combine in adjacent squares to give a simplified expression in sum of products. The C output can be simplified to a six-literal expression. The logic diagram for the full-adder implemented in sum of products is shown in Fig. 4-4. This implementation uses the following Boolean expressions:

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

$$C = xy + xz + yz$$

Other configurations for a full-adder may be developed. The product of sums implementation requires the same number of gates as in Fig. 4-4, with the number of AND and OR gates interchanged. A full-adder can be implemented with two half-adders and

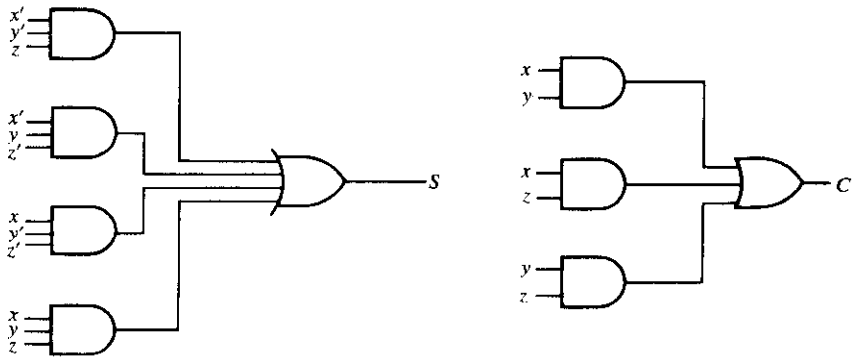


FIGURE 4-4
Implementation of a full-adder in sum of products

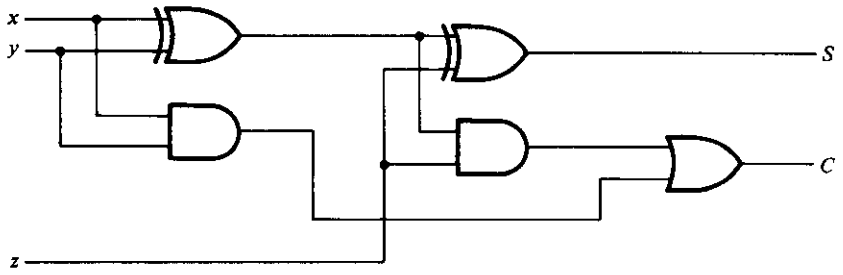


FIGURE 4-5
Implementation of a full-adder with two half-adders and an OR gate

one OR gate, as shown in Fig. 4-5. The S output from the second half-adder is the exclusive-OR of z and the output of the first half-adder, giving

$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

and the carry output is

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

4-4 SUBTRACTORS

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend (Section 1-5). By this method, the subtraction operation becomes an addition operation requiring full-adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. The fact that a 1 has been borrowed must be conveyed to the next higher pair of bits by means of a binary signal coming out (output) of a given stage and going into (input) the next higher stage. Just as there are half- and full-adders, there are half- and full-subtractors.

Half-Subtractor

A half-subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. Designate the min-

uend bit by x and the subtrahend bit by y. To perform $x - y$, we have to check the relative magnitudes of x and y. If $x \geq y$, we have three possibilities: $0 - 0 = 0$, $1 - 0 = 1$, and $1 - 1 = 0$. The result is called the *difference bit*. If $x < y$, we have $0 - 1$, and it is necessary to borrow a 1 from the next higher stage. The 1 borrowed from the next higher stage adds 2 to the minuend bit, just as in the decimal system a borrow adds 10 to a minuend digit. With the minuend equal to 2, the difference becomes $2 - 1 = 1$. The half-subtractor needs two outputs. One output generates the difference and will be designated by the symbol D. The second output, designated B for borrow, generates the binary signal that informs the next stage that a 1 has been borrowed. The truth table for the input–output relationships of a half-subtractor can now be derived as follows:

x	y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow B is a 0 as long as $x \geq y$. It is a 1 for $x = 0$ and $y = 1$. The D output is the result of the arithmetic operation $2B + x - y$.

The Boolean functions for the two outputs of the half-subtractor are derived directly from the truth table:

$$D = x'y + xy'$$

$$B = x'y$$

It is interesting to note that the logic for D is exactly the same as the logic for output S in the half-adder.

Full-Subtractor

A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 may have been borrowed by a lower significant stage. This circuit has three inputs and two outputs. The three inputs, x, y, and z, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and B, represent the difference and output borrow, respectively. The truth table for the circuit is

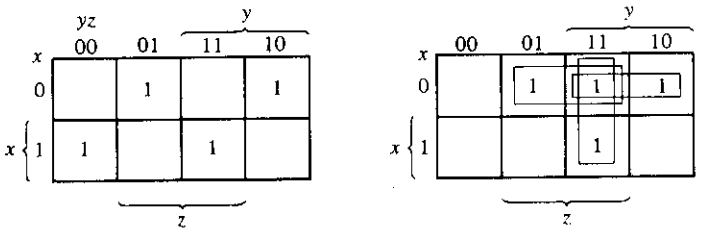
x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations of 1's and 0's that the binary variables may take. The 1's and 0's for the output variables are determined from the subtraction of $x - y - z$. The combinations having input borrow $z = 0$ reduce to the same four conditions of the half-adder. For $x = 0$, $y = 0$, and $z = 1$, we have to borrow a 1 from the next stage, which makes $B = 1$ and adds 2 to x. Since $2 - 0 - 1 = 1$, $D = 1$. For $x = 0$ and $yz = 11$, we need to borrow again, making $B = 1$ and $x = 2$. Since $2 - 1 - 1 = 0$, $D = 0$. For $x = 1$ and $yz = 01$, we have $x - y - z = 0$, which makes $B = 0$ and $D = 0$. Finally, for $x = 1$, $y = 1$, $z = 1$, we have to borrow 1, making $B = 1$ and $x = 3$, and $3 - 1 - 1 = 1$, making $D = 1$.

The simplified Boolean functions for the two outputs of the full-subtractor are derived in the maps of Fig. 4-6. The simplified sum of products output functions are

$$D = x'y'z + x'yz' + xy'z' + xyz$$

$$B = x'y + x'z + yz$$



$D = x'y'z + x'yz + xy'z' + xyz$ $\quad$ $B = x'y + x'z + yz$

FIGURE 4-6
Maps for a full-subtractor

4-5 CODE CONVERSION

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. The design procedure of code converters will be illustrated by means of a specific example of conversion from the BCD to the excess-3 code.

The bit combinations for the BCD and excess-3 codes are listed in Table 1-2 (Section 1-7). Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. Let us designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y, and z. The truth table relating the input and output variables is shown in Table 4-1. The bit combinations for the inputs and their corresponding outputs are obtained directly from Table 1-2. We note that four binary variables may have 16 bit combinations, only 10 of which are listed in the truth table. The six bit combinations not listed for the *input* variables are don't-care combinations. Since they will never occur, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

The maps in Fig. 4-7 are drawn to obtain a simplified Boolean function for each output. Each of the four maps of Fig. 4-7 represents one of the four outputs of this circuit as a function of the four input variables. The 1's marked inside the squares are obtained

TABLE 4-1
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

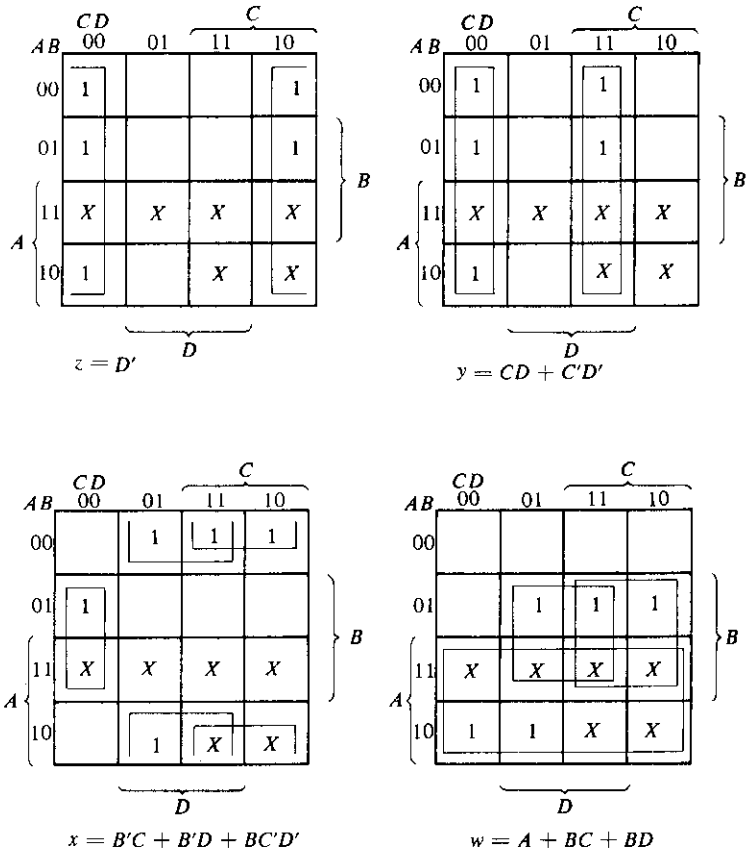


FIGURE 4-7
Maps for a BCD-to-excess-3-code converter

from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output z has five 1's; therefore, the map for z must have five 1's, each being in a square corresponding to the minterm that makes z equal to 1. The six don't-care combinations are marked by X's. One possible way to simplify the functions in sum of products is listed under the map of each variable.

A two-level logic diagram may be obtained directly from the Boolean expressions derived by the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. 4-7 may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown below, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates.

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$

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

$$w = A + BC + BD = A + B(C + D)$$

The logic diagram that implements these expressions is shown in Fig. 4-8. In it we see that the OR gate whose output is $C + D$ has been used to implement partially each of three outputs.

Not counting input inverters, the implementation in sum of products requires seven AND gates and three OR gates. The implementation of Fig. 4-8 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first implementation will require inverters for variables B, C, and D, whereas the second implementation requires inverters for variables B and D.

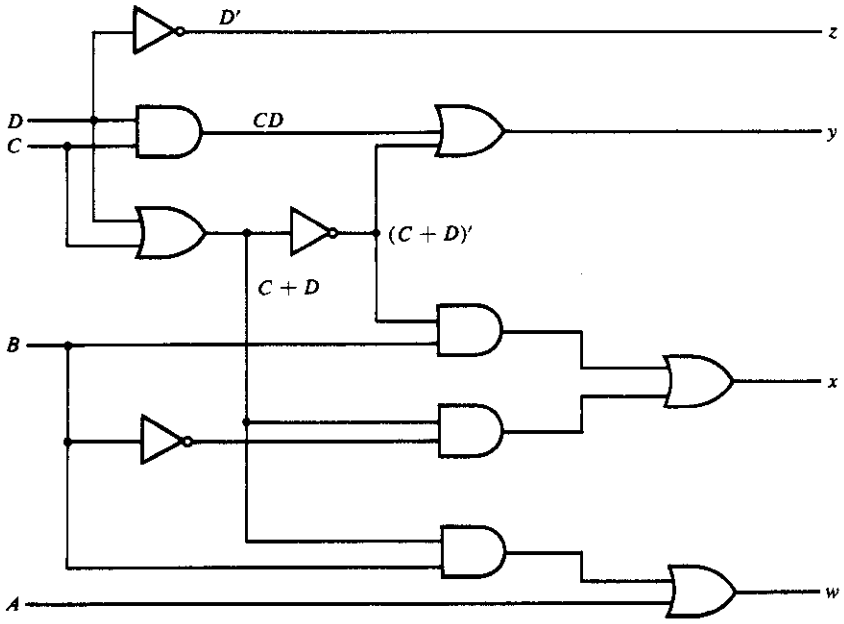


FIGURE 4-8
Logic diagram for a BCD-to-excess-3-code converter

4-6 ANALYSIS PROCEDURE

The design of a combinational circuit starts from the verbal specifications of a required function and culminates with a set of output Boolean functions or a logic diagram. The *analysis* of a combinational circuit is somewhat the reverse process. It starts with a

given logic diagram and culminates with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation. If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function.

The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate. Feedback paths or memory elements in a digital circuit define a sequential circuit and must be analyzed according to procedures outlined in Chapter 6 or Chapter 9.

Once the logic diagram is verified as a combinational circuit, one can proceed to obtain the output Boolean functions and/or the truth table. If the circuit is accompanied by a verbal explanation of its function, then the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived truth table. The success of such investigation is enhanced if one has previous experience and familiarity with a wide variety of digital circuits. The ability to correlate a truth table with an information-processing task is an art one acquires with experience.

To obtain the output Boolean functions from a logic diagram, proceed as follows:

1. Label with arbitrary symbols all gate outputs that are a function of the input variables. Obtain the Boolean functions for each gate.
2. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables only.

Analysis of the combinational circuit in Fig. 4-9 illustrates the proposed procedure. We note that the circuit has three binary inputs, A, B, and C, and two binary outputs, F_1 and F_2. The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function of input variables only are F_2, T_1, and T_2. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

Next we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2' T_1$$

$$F_1 = T_3 + T_2$$

The output Boolean function F_2 just expressed is already given as a function of the in-

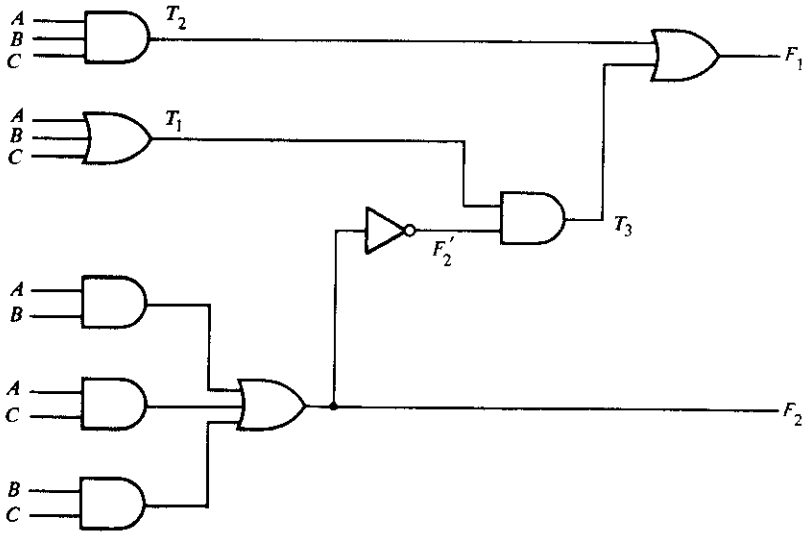


FIGURE 4-9
Logic diagram for analysis example

puts only. To obtain F_1 as a function of A, B, and C, form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F_2' T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If we want to pursue the investigation and determine the information-transformation task achieved by this circuit, we can derive the truth table directly from the Boolean functions and try to recognize a familiar operation. For this example, we note that the circuit is a full-adder, with F_1 being the sum output and F_2 the carry output. A, B, and C are the three inputs added arithmetically.

The derivation of the truth table for the circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, proceed as follows:

1. Determine the number of input variables to the circuit. For n inputs, form the 2^n possible input combinations of 1's and 0's by listing the binary numbers from 0 to $2^n - 1$.

2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

This process can be illustrated using the circuit of Fig. 4-9. In Table 4-2, we form the eight possible combinations for the three input variables. The truth table for F_2 is determined directly from the values of A, B, and C, with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F_2' is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T_3 are derived from T_1 and F_2': T_3 is equal to 1 when both T_1 and F_2' are equal to 1, and to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1. Inspection of the truth table combinations for A, B, C, F_1, and F_2 of Table 4-2 shows that it is identical to the truth table of the full-adder given in Section 4-3 for x, y, z, S, and C, respectively.

TABLE 4-2
Truth Table for the Logic Diagram of Fig. 4-9

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Consider now a combinational circuit that has don't-care input combinations. When such a circuit is designed, the don't-care combinations are marked by X's in the map and assigned an output of either 1 or 0, whichever is more convenient for the simplification of the output Boolean function. When a circuit with don't-care combinations is being analyzed, the situation is entirely different. Even though we assume that the don't-care input combinations will never occur, if any one of these combinations is applied to the inputs (intentionally or in error), a binary output will be present. The value of the output will depend on the choice for the X's taken during the design. Part of the analysis of such a circuit may involve the determination of the output values for the don't-care input combinations. As an example, consider the BCD-to-excess-3-code converter designed in Section 4-5. The outputs obtained when the six unused combinations of the BCD code are applied to the inputs are

Unused BCD Inputs				Outputs			
A	*B*	*C*	*D*	*w*	*x*	*y*	*z*
1	0	1	0	1	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	1	1	1	1
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	1
1	1	1	1	1	0	1	0

These outputs may be derived by means of the truth table analysis method as outlined in this section. In this particular case, the outputs may be obtained directly from the maps of Fig. 4-7. From inspection of the maps, we determine whether the X's in the corresponding minterm squares for each output have been included with the 1's or the 0's. For example, the square for minterm m_{10} (1010) has been included with the 1's for outputs w, x, and z, but not for y. Therefore, the outputs for m_{10} are $wxyz = 1101$, as listed in the previous table. We also note that the first three outputs in the table have no meaning in the excess-3 code, and the last three outputs correspond to decimal 5, 6, and 7, respectively. This coincidence is entirely a function of the choice for the X's taken during the design.

4-7 MULTILEVEL NAND CIRCUITS

Combinational circuits are more frequently constructed with NAND or NOR gates rather than AND and OR gates. NAND and NOR gates are more common from the hardware point of view because they are readily available in integrated-circuit form. Because of the prominence of NAND and NOR gates in the design of combinational circuits, it is important to be able to recognize the relationships that exist between circuits constructed with AND-OR gates and their equivalent NAND or NOR diagrams.

The implementation of two-level NAND and NOR logic diagrams was presented in Section 3-6. Here we consider the more general case of multilevel circuits. The procedure for obtaining NAND circuits is presented in this section, and for NOR circuits in the next section.

Universal Gate

The NAND gate is said to be a universal gate because any digital system can be implemented with it. Combinational circuits and sequential circuits as well can be constructed with this gate because the flip-flop circuit (the memory element most frequently used in sequential circuits) can be constructed from two NAND gates connected back to back, as shown in Section 6-2.

To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations AND, OR, and NOT can be implemented with

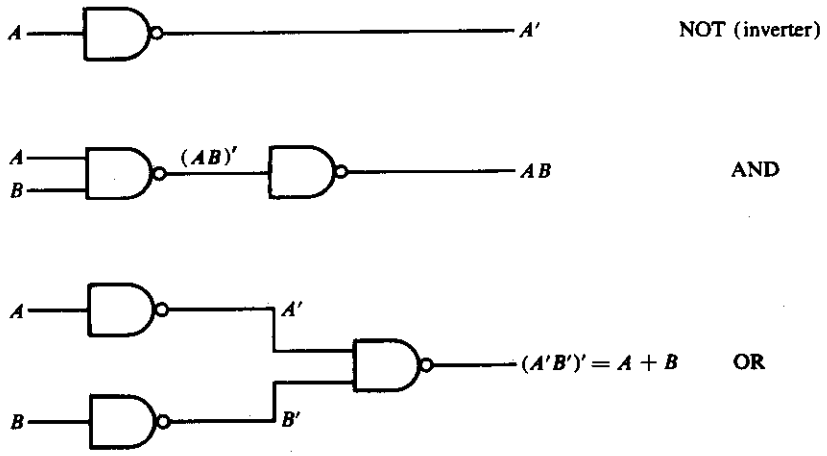


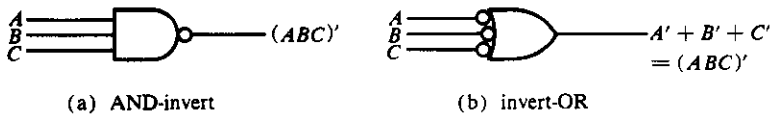
FIGURE 4-10
Implementation of NOT, AND, and OR by NAND gates

NAND gates. The implementation of the AND, OR, and NOT operations with NAND gates is shown in Fig. 4-10. The NOT operation is obtained from a one-input NAND gate, actually another symbol for an inverter circuit. The AND operation requires two NAND gates. The first produces the inverted AND and the second acts as an inverter to produce the normal output. The OR operation is achieved through a NAND gate with additional inverters in each input.

Boolean-Function Implementation

One possible way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit-manipulation techniques that change AND-OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to use the two alternate graphic symbols shown in Fig. 4-11. (These two graphic symbols for the NAND gate were introduced in Fig. 3-17(a) and are repeated here for convenience.) The AND-invert graphic symbol consists of an AND graphic symbol followed by a small circle. The invert-OR graphic symbol consists of an OR graphic symbol that is preceded by small circles in all the inputs. Either symbol can be used to represent a NAND gate.



(a) AND-invert (b) invert-OR

FIGURE 4-11
Two graphic symbols for a NAND gate

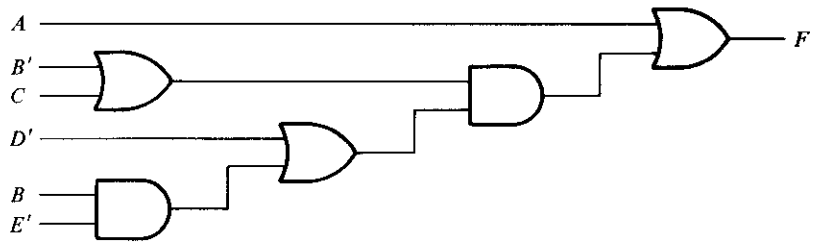
To obtain a multilevel NAND diagram from a Boolean expression, proceed as follows:

1. From the given Boolean expression, draw the logic diagram with AND, OR, and inverter gates. Assume that both the normal and complement inputs are available.
2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
3. Convert all OR gates to NAND gates with invert-OR graphic symbols.
4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same line, insert an inverter (one-input NAND gate) or complement the input variable.

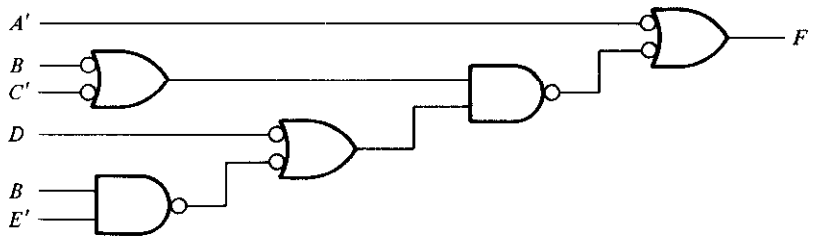
This procedure will be demonstrated with two examples. First, consider the Boolean function

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

Although it is possible to remove the parentheses and convert the expression into a standard sum of products form, we choose to implement it as a multilevel circuit for illustration. The AND-OR implementation is shown in Fig. 4-12(a). There are four levels of gating in the circuit. The first level has an AND and an OR gate. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level. A logic diagram with a pattern of alternate levels of AND and OR gates

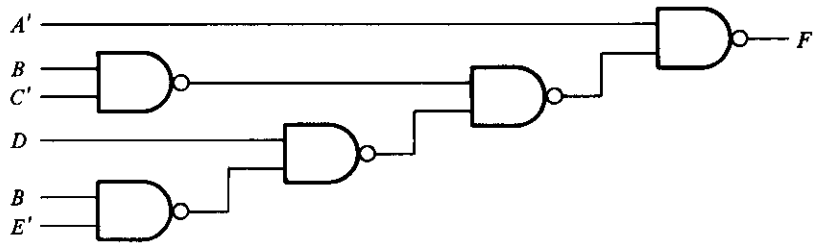


(a) AND-OR diagram



(b) NAND diagram using two graphic symbols

FIGURE 4-12
Implementing $F = A + (B' + C)(D' + BE')$ with NAND gates (continued on next page)



(c) NAND diagram using one graphic symbol

FIGURE 4-12 *(continued)*

can be easily converted into a NAND circuit. This is shown in Fig. 4-12(b). The procedure requires that we change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND-OR circuit as long as the complementing small circles do not change the value of the function. Any connection between an output of a gate that has a complementing circle and the input of another gate that also has a complementing circle represents double complementation and does not change the logic of the circuit. However, the small circles associated with inputs A, B', C, and D' cause extra complementations that are not compensated with other small circles along the same line. We can insert inverters after each of these inputs or, as shown in the figure, complement the literals to obtain A', B, C', and D.

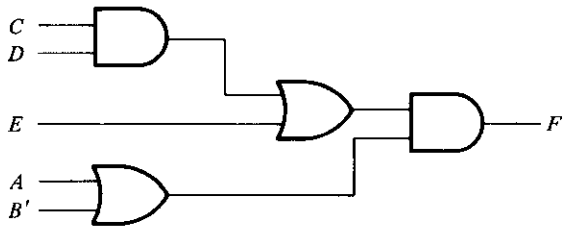
Because it does not matter whether we use the AND-invert or the invert-OR graphic symbol to represent a NAND gate, the diagram of Fig. 4-12(c) is identical to the NAND diagram of part (b). In fact, the diagram of Fig. 4-12(b) is preferable because it represents a clearer picture of the Boolean expression it implements.

As another example, consider the multilevel Boolean expression

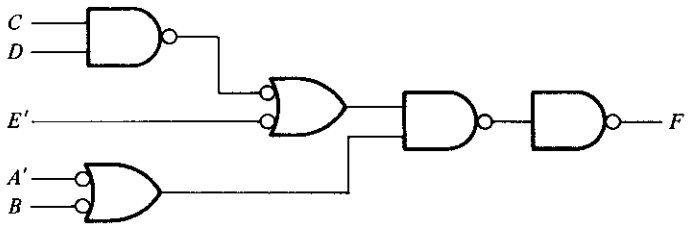
$$F = (CD + E)(A + B')$$

The AND-OR implementation is shown in Fig. 4-13(a) with three levels of gating. The conversion into a NAND circuit is presented in part (b) of the diagram. The three additional small circles associated with inputs E, A, and B' cause these three literals to be complemented to E', A', and B. The small circle in the last NAND gate complements the output, so we need to insert an inverter gate at the output in order to complement the signal again and obtain the original value.

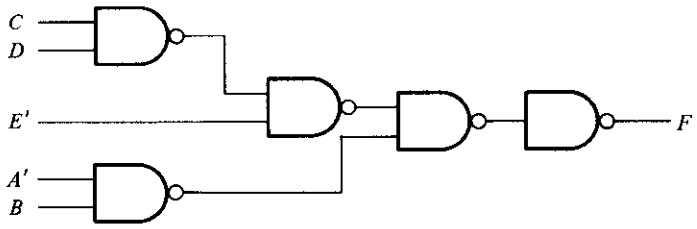
The number of NAND gates required to implement the expression of the first example is the same as the number of AND and OR gates in the AND-OR diagram. The number of NAND gates in the second example is equal to the number of AND-OR gates plus an additional inverter in the output. In general, the number of NAND gates required to implement a Boolean expression is equal to the number of AND-OR gates except for an occasional inverter. This is true provided both the normal and complement inputs are available, because the conversion forces certain input variables to be complemented.



(a) AND-OR diagram



(b) NAND diagram



(c) Alternate NAND diagram

FIGURE 4-13
Implementing $F = [CD + E][A + B']$ with NAND gates

Analysis Procedure

The foregoing procedure considered the problem of deriving a NAND logic diagram from a given Boolean function. The reverse process is the analysis problem that starts with a given NAND logic diagram and culminates with a Boolean expression or a truth table. The analysis of NAND logic diagrams follows the same procedures presented in Section 4-6 for the analysis of combinational circuits. The only difference is that NAND logic requires a repeated application of DeMorgan's theorem. We shall now demonstrate the derivation of the Boolean function from a logic diagram. Then we will show the derivation of the truth table directly from the NAND logic diagram. Finally, a method will be presented for converting a NAND logic diagram to AND-OR logic diagram.

Derivation of the Boolean Function by Algebraic Manipulation

The procedure for deriving the Boolean function from a logic diagram is outlined in Section 4-6. This procedure is demonstrated for the NAND logic diagram shown in Fig. 4-14. First, all gate outputs are labeled with arbitrary symbols. Second, the Boolean functions for the outputs of gates that receive only external inputs are derived:

$$T_1 = (CD)' = C' + D'$$

$$T_2 = (BC')' = B' + C$$

The second form follows directly from DeMorgan's theorem and may, at times, be more convenient to use. Third, Boolean functions of gates that have inputs from previously derived functions are determined in consecutive order until the output is expressed in terms of input variables:

$$\begin{aligned} T_3 &= (B'T_1)' = (B'C' + B'D')' \\ &= (B + C)(B + D) = B + CD \\ T_4 &= (AT_3)' = [A(B + CD)]' \\ F &= (T_2T_4)' = \{(BC')'[A(B + CD)]'\}' \\ &= BC' + A(B + CD) \end{aligned}$$

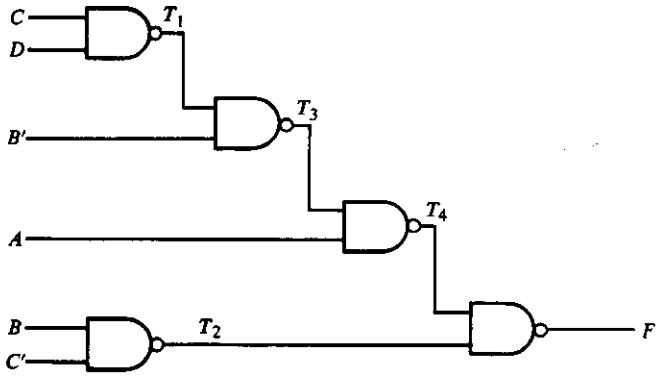


FIGURE 4-14
Analysis example

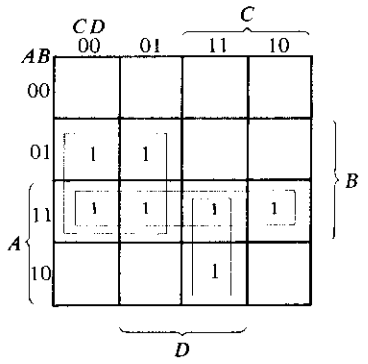
Derivation of the Truth Table

The procedure for obtaining the truth table directly from a logic diagram is also outlined in Section 4-6. This procedure is demonstrated for the NAND logic diagram of Fig. 4-14. First, the four input variables, together with their 16 combinations of 1's and 0's, are listed as in Table 4-3. Second, the outputs of all gates are labeled with arbitrary symbols as in Fig. 4-14. Third, we obtain the truth table for the outputs of those

TABLE 4-3
Truth Table for the Circuit of Figure 4-14

A	B	C	D	T_1	T_2	T_3	T_4	F
0	0	0	0	1	1	0	1	0
0	0	0	1	1	1	0	1	0
0	0	1	0	1	1	0	1	0
0	0	1	1	0	1	1	1	0
0	1	0	0	1	0	1	1	1
0	1	0	1	1	0	1	1	1
0	1	1	0	1	1	1	1	0
0	1	1	1	0	1	1	1	0
1	0	0	0	1	1	0	1	0
1	0	0	1	1	1	0	1	0
1	0	1	0	1	1	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	0	1	0	1	0	1
1	1	0	1	1	0	1	0	1
1	1	1	0	1	1	1	0	1
1	1	1	1	0	1	1	0	1

gates that are a function of the input variables only. These are T_1 and T_2. $T_1 = (CD)'$; so we mark 0's in those rows where both C and D are equal to 1 and fill the rest of the rows of T_1 with 1's. Also, $T_2 = (BC')'$; so we mark 0's in those rows where $B = 1$ and $C = 0$, and fill the rest of the rows of T_2 with 1's. We then proceed to obtain the truth table for the outputs of those gates that are a function of previously defined outputs until the column for the output F is determined. It is now possible to obtain an algebraic expression for the output from the derived truth table. The map shown in Fig. 4-15 is ob-



$F = AB + BC' + ACD$

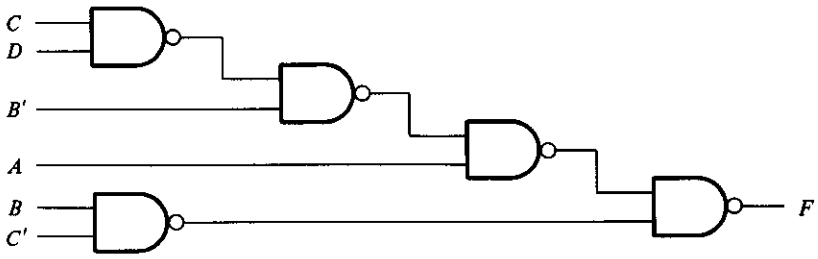
FIGURE 4-15
Derivation of F from Table 4-3

tained directly from Table 4-3 and has 1's in the squares of those minterms for which F is equal to 1. The simplified expression obtained from the map is

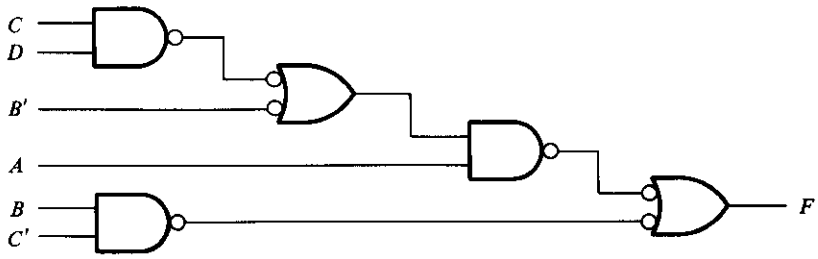
$$F = AB + ACD + BC' = A(B + CD) + BC'$$

Transformation to AND-OR Diagram

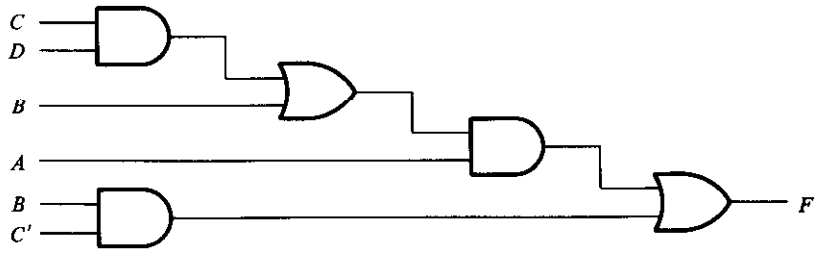
It is sometimes convenient to convert a NAND logic diagram to its equivalent AND-OR logic diagram to facilitate the analysis procedure. By doing so, the Boolean expression can be derived more easily from the diagram without employing DeMorgan's the-



(a) NAND logic diagram



(b) Substitution of invert-OR symbols in alternate levels



(c) AND-OR logic diagram

FIGURE 4-16
Conversion of NAND logic diagram to AND-OR

orem. The conversion is achieved through a change in graphic symbols from AND-invert to invert-OR in *alternate* levels in the gate structure. The first level to be changed to an invert-OR symbol should be the last level. These changes produce pairs of small circles along the same line, which are then removed since they represent double complementation. Any small circle associated with an input can be removed provided the input variable is complemented. A one-input AND or OR gate with a small circle in the input or output represents an inverter circuit.

The procedure is demonstrated in Fig. 4-16. The NAND logic diagram of Fig. 4-16(a) is to be converted to an equivalent AND-OR diagram. The graphic symbol of the NAND gate in the last level is changed to an invert-OR symbol. Looking for alternate levels, we find one more gate requiring a change of symbol, as shown in Fig. 4-16(b). Any two small circles along the same line are removed. The small circle connected to input B' is removed and the input variable is complemented. The required AND-OR logic diagram is shown in Fig. 4-16(c). The Boolean expression for F can be easily determined from the AND-OR diagram to be

$$F = BC' + A(B + CD)$$

4-8 MULTILEVEL NOR CIRCUITS

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules developed for NAND logic. This section enumerates various methods for NOR logic implementation and analysis by following the same list of topics used for NAND logic. However, less detailed explanation is included so as to avoid excessive repetition of the material in Section 4-7.

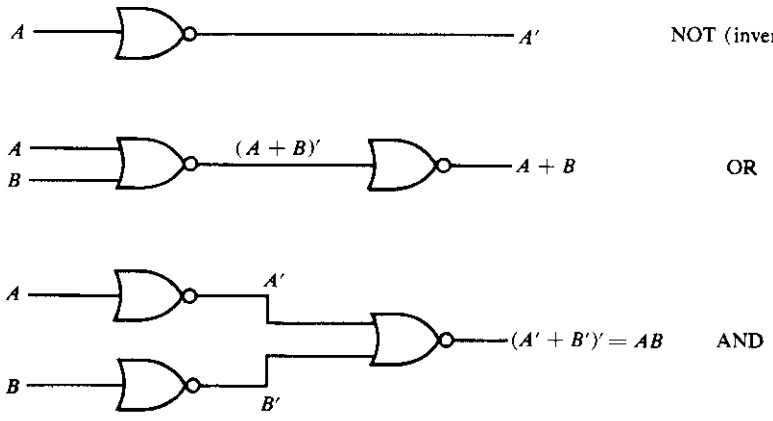


FIGURE 4-17
Implementation of NOT, OR, and AND by NOR gates

Universal Gate

The NOR gate is universal because any Boolean function can be implemented with it, including a flip-flop circuit, as shown in Section 6-2. The conversion of AND, OR, and NOT to NOR is shown in Fig. 4-17. The NOT operation is obtained from a one-input NOR gate, yet another symbol for an inverter circuit. The OR operation requires two NOR gates. The first produces the inverted-OR and the second acts as an inverter to obtain the normal output. The AND operation is achieved through a NOR gate with additional inverters at each input.

Boolean-Function Implementation

The two graphic symbols for the NOR gate are shown in Fig. 4-18. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.

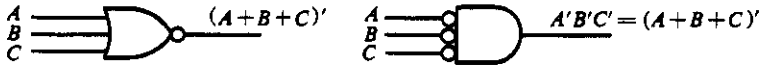
The procedure for implementing a Boolean function with NOR gates is similar to the procedure outlined in the previous section for NAND gates.

1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.
2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
4. Any small circle that is not compensated by another small circle along the same line needs an inverter or the complementation of the input variable.

The procedure is illustrated in Fig. 4-19 for the Boolean function

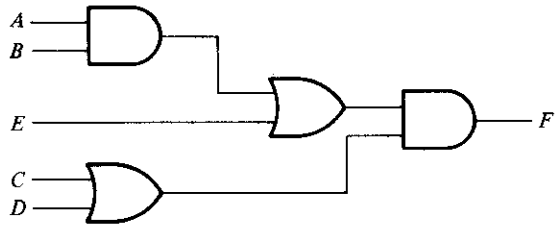
$$F = (AB + E)(C + D)$$

The AND-OR implementation of the expression is shown in the logic diagram of Fig. 4-19(a). For each OR gate, we substitute a NOR gate with the OR-invert graphic symbol. For each AND gate, we substitute a NOR gate with the invert-AND graphic symbol. The two small circles associated with inputs A and B cause these two variables to be complemented to A' and B', respectively. The NOR diagram is shown in Fig. 4-19(b). The diagram of Fig. 4-19(c) is an alternate way of drawing the diagram using only one type of graphic symbol for the NOR gate.

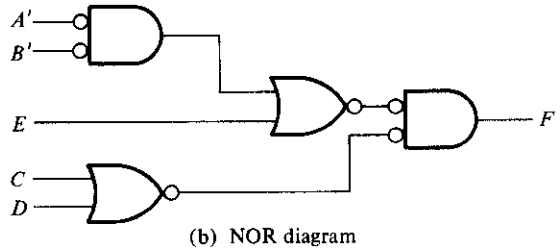


(a) OR-invert (b) invert-AND

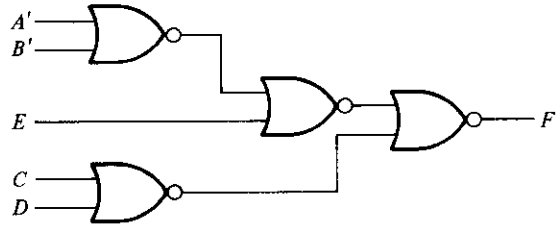
FIGURE 4-18
Two graphic symbols for a NOR gate



(a) AND-OR diagram



(b) NOR diagram



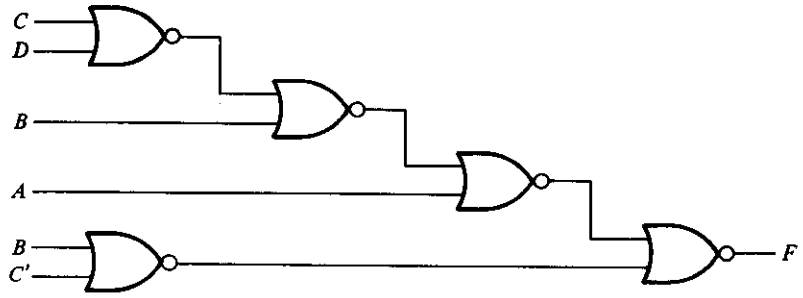
(c) Alternate NOR diagram

FIGURE 4-19
Implementing $F = [AB + E](C + D)$ with NOR gates

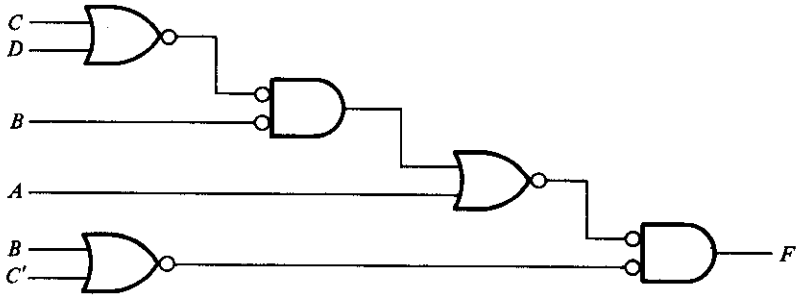
In general, the number of NOR gates required to implement a Boolean function will be the same as the number of gates in the AND-OR diagram. This is true provided both the normal and complement inputs are available, because the conversion may require that certain input variables be complemented.

Analysis Procedure

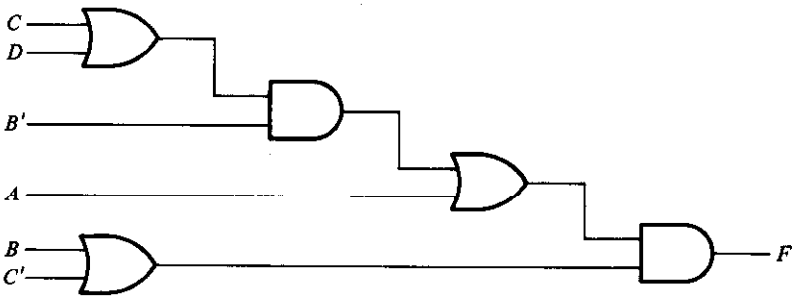
The analysis of NOR logic diagrams follows the same procedure presented in Section 4-6 for the analysis of combinational circuits. To derive the Boolean expression from a logic diagram, we mark the outputs of various gates with arbitrary symbols. By repetitive substitutions, we obtain the output variable as a function of the input variables.



(a) NOR logic diagram



(b) Sustituting invert-AND in alternate levels



(c) AND-OR logic diagram

FIGURE 4-20
Conversion of NOR diagram to AND-OR

To obtain the truth table from a logic diagram without first deriving the Boolean expression, we form a table with the n variables by listing the 2^n binary combinations. The truth table of selected NOR gate outputs is derived in succession until the output truth table is obtained. The output expression of a typical NOR gate is of the form $T = (A + B' + C)'$. By using DeMorgan's theorem, this can be expressed as $T = A'BC'$. The truth table for T is marked with 1's for those combinations where $ABC = 010$ and the rest of the rows are filled with 0's.

The conversion of a NOR logic diagram to an AND-OR diagram is achieved through a change of graphic symbols from OR-invert to invert-AND starting from the last logic level and in alternate levels. Pairs of small circles along the same line are removed. A one-input AND or OR gate is removed, but if it has a small circle at the input or output, it is converted to an inverter. Any small circle associated with an input is removed and the input variable is complemented.

This procedure is demonstrated in Fig. 4-20, where the NOR logic diagram in part (a) is converted to an AND-OR diagram. The graphic symbol of the gate in the last (fourth) logic level is changed to an invert-AND. Looking for alternate levels, we find a gate in level two that needs to undergo a symbol change, as shown in part (b). Any two circles along the same line are removed. The circle associated with external input B is removed and the input variable is changed to B'. The required AND-OR logic diagram is drawn in part (c). The Boolean expression for the circuit can be obtained by inspection and then manipulated into a product of sums form:

$$\begin{aligned} F &= [(C + D)B' + A](B + C') \\ &= (A + C + D)(A + B')(B + C') \end{aligned}$$

4-9 EXCLUSIVE-OR FUNCTION

The exclusive-OR (XOR) denoted by the symbol $\oplus$ is a logical operation that performs the following Boolean operation:

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

It is equal to 1 if only x is equal to 1 or if only y is equal to 1 but not when both are equal to 1. The exclusive-NOR, also known as equivalence, performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

It is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation.

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

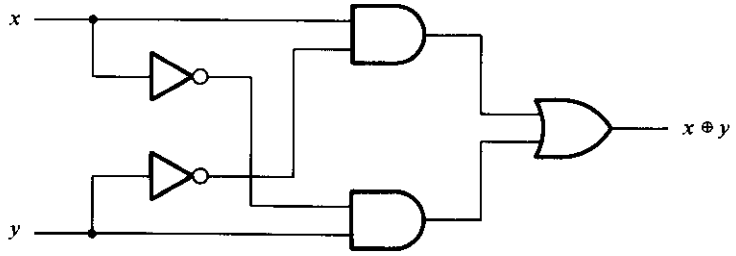
The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x \qquad x \oplus 1 = x'$$
$$x \oplus x = 0 \qquad x \oplus x' = 1$$
$$x \oplus y' = (x \oplus y)' \qquad x' \oplus y = (x \oplus y)'$$

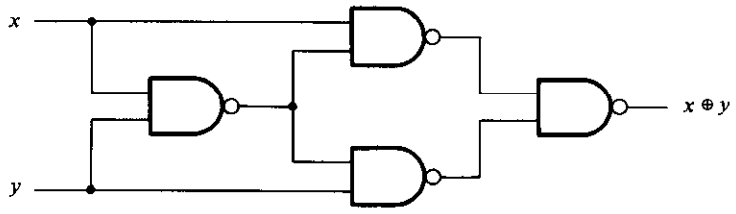
Any of these identities can be proven by using a truth table or by replacing the $\oplus$ operation by its equivalent Boolean expression. It can be shown also that the exclusive-OR operation is both commutative and associative.

$$A \oplus B = B \oplus A$$
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig. 4-21(a). Figure 4-21(b) shows the implementation of the exclusive-OR with four NAND gates.



(a) With AND-OR-NOT gates



(b) With NAND gates

FIGURE 4-21
Exclusive-OR implementations

The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit produces the sum of products of its inputs:

$$(x' + y')x + (x' + y')y = xy' + x'y = x \oplus y$$

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error-detection and correction circuits.

Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

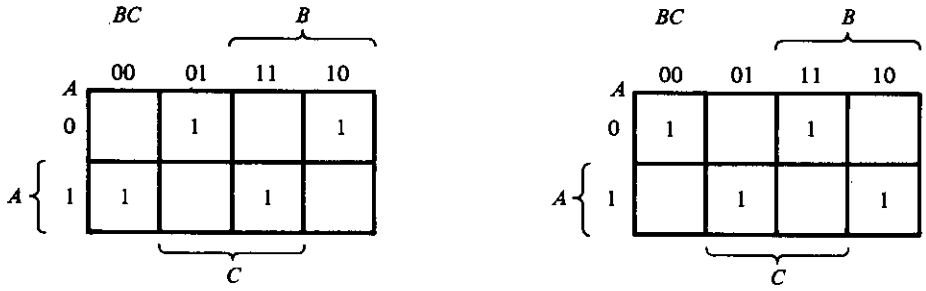
$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma\,(1, 2, 4, 7) \end{aligned}$$

The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, where only one variable must be equal to 1, in the three or more variable case, the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function*.

The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010, 100, and 111. Each of these binary numbers has an odd number of 1's. The other four minterms not included in the function are 000, 011, 101, and 110, and they have an even number of 1's in their binary numerical values. In general, an n-variable exclusive-OR function is an odd function defined as the logical sum of the $2^n/2$ minterms whose binary numerical values have an odd number of 1's.

The definition of an odd function can be clarified by plotting it in a map. Figure 4-22(a) shows the map for the three-variable exclusive-OR function. The four minterms of the function are a unit distance apart from each other. The odd function is identified from the four minterms whose binary values have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 4-22(b), the three-variable even function is equal to 1 when an even number of variables is equal to 1 (including the condition that none of the variables is equal to 1).

The 3-input odd function is implemented by means of 2-input exclusive-OR gates, as shown in Fig. 4-23(a). The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate, as shown in Fig. 4-23(b).



(a) Odd function
$F = A \oplus B \oplus C$

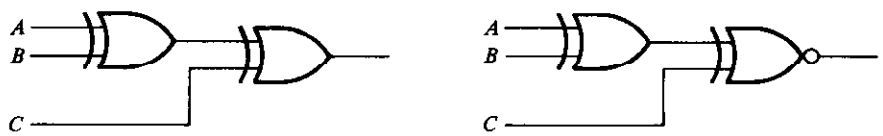
(b) Even function
$F = (A \oplus B \oplus C)'$

FIGURE 4-22
Map for a three-variable exclusive-OR function

Consider now the the four-variable exclusive-OR operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

$$\begin{aligned} A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\ &= \Sigma\,(1, 2, 4, 7, 8, 11, 13, 14) \end{aligned}$$

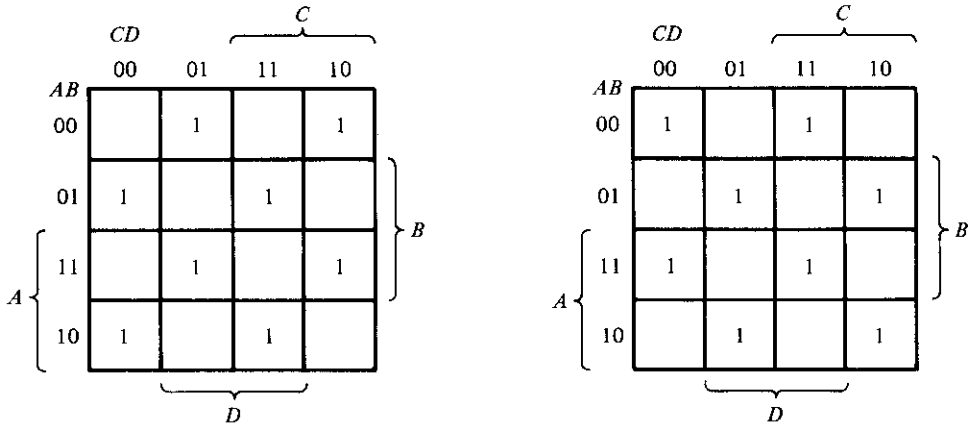
There are 16 minterms for a four-variable Boolean function. Half of the minterms have binary numerical values with an odd number of 1's; the other half of the minterms have binary numerical values with an even number of 1's. When plotting the function in the map, the binary numerical value for a minterm is determined from the row and column numbers of the square that represents the minterm. The map of Fig. 4-24(a) is a plot of the four-variable exclusive-OR function. This is an odd function because the binary values of all the minterms have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 4-24(b), the four-variable even function is equal to 1 when an even number of variables is equal to 1.



(a) 3-input odd function

(b) 3-input even function

FIGURE 4-23
Logic diagram of odd and even functions



(a) Odd function
$F = A \oplus B \oplus C \oplus D$

(b) Even function
$F = (A \oplus B \oplus C \oplus D)'$

FIGURE 4-24
Map for a four-variable exclusive-OR function

Parity Generation and Checking

Exclusive-OR functions are very useful in systems requiring error-detection and correction codes. As discussed in Section 1-7, a parity bit is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator*. The circuit that checks the parity in the receiver is called a *parity checker*.

As an example, consider a 3-bit message to be transmitted together with an even parity bit. Table 4-4 shows the truth table for the parity generator. The three bits, x, y, and z, constitute the message and are the inputs to the circuit. The parity bit P is the output. For even parity, the bit P must be generated to make the total number of 1's even (including P). From the truth table, we see that P constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, P can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator is shown in Fig. 4-25(a).

The three bits in the message together with the parity bit are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the

TABLE 4-4
Even-Parity-Generator Truth Table

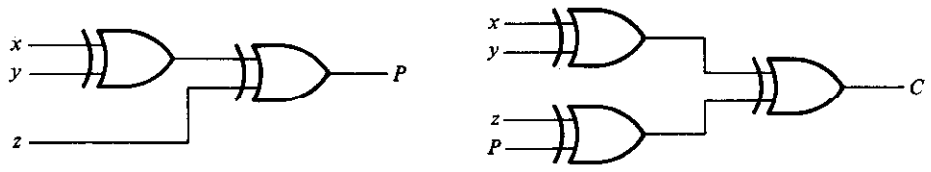
Three-Bit Message			Parity Bit
x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by C, will be equal to 1 if an error occurs, that is, if the four bits received have an odd number of 1's. Table 4-5 is the truth table for the even-parity checker. From it we see that the function C consists of the eight minterms with binary numerical values having an odd number of 1's. This corresponds to the map of Fig. 4-24(a), which represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker is shown in Fig. 4-25(b).

It is worth noting that the parity generator can be implemented with the circuit of Fig. 4-25(b) if the input P is connected to logic-0 and the output is marked with P. This is because $z \oplus 0 = z$, causing the value of z to pass through the gate unchanged. The advantage of this is that the same circuit can be used for both parity generation and checking.



(a) 3-bit even parity generator (b) 4-bit even parity checker

FIGURE 4-25
Logic diagram of a parity generator and checker

TABLE 4-5
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	y	z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

It is obvious from the foregoing example that parity-generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

REFERENCES

1. HILL, F. J., and G. R. PETERSON, *Introduction to Switching Theory and Logical Design*, 3rd Ed. New York: John Wiley, 1981.
2. KOHAVI, Z., *Switching and Automata Theory*, 2nd Ed. New York: McGraw-Hill, 1978.
3. ROTH, C. H., *Fundamentals of Logic Design*, 3rd Ed. St. Paul, Minnesota: West Publishing Co., 1985.
4. BOOTH, T. L., *Introduction to Computer Engineering*, 3rd Ed. New York: John Wiley, 1984.
5. MANO, M. M., *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
6. FLETCHER, W. I., *An Engineering Approach to Digital Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

7. ERCEGOVAC, M. D., and T. LANG, *Digital Systems and Hardware/Firmware Algorithms*. New York: John Wiley, 1985.
8. MANGE, D., *Analysis and Synthesis of Logic Systems*. Norwood, MA: Artech House, 1986.
9. SHIVA, S. G., *Introduction to Logic Design*. Glenview, IL: Scott, Foresman, 1988.
10. MCCLUSKEY, E. J., *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

PROBLEMS

4-1 Design a combinational circuit with three inputs and one output. The output is equal to logic-1 when the binary value of the input is less than 3. The output is logic-0 otherwise.

4-2 Design a combinational circuit with three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is one less than the input.

4-3 A majority function is generated in a combinational circuit when the output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise. Design a 3-input majority function.

4-4 Design a combinational circuit that adds one to a 4-bit binary number, $A_3A_2A_1A_0$. For example, if the input of the circuit is $A_3A_2A_1A_0 = 1101$, the output is 1110. The circuit can be designed using four half-adders.

4-5 A combinational circuit produces the binary sum of two 2-bit numbers, x_1x_0 and y_1y_0. The outputs are C, S_1, and S_0. Provide a truth table of the combinational circuit.

4-6 Design the circuit of Problem 4-5 using two full-adders.

4-7 Design a combinational circuit that multiplies two 2-bit numbers, a_1a_0 and b_1b_0, to produce a 4-bit product, $c_3c_2c_1c_0$. Use AND gates and half-adders.

4-8 Show that a full-subtractor can be constructed with two half-subtractors and an OR gate.

4-9 Design a combinational circuit with three inputs and six outputs. The output binary number should be the square of the input binary number.

4-10 Design a combinational circuit with four inputs that represent a decimal digit in BCD and four outputs that produce the 9's complement of the input digit. The six unused combinations can be treated as don't-care conditions.

4-11 Design a combinational circuit with four inputs and four outputs. The output generates the 2's complement of the input binary number.

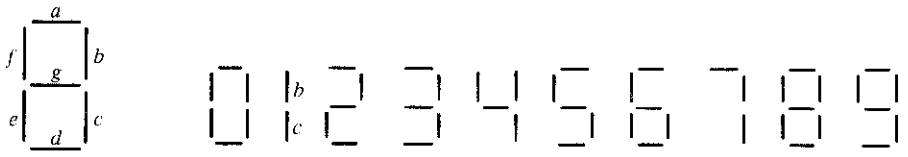
4-12 Design a combinational circuit that detects an error in the representation of a decimal digit in BCD. The output of the circuit must be equal to logic-1 when the inputs contain any one of the six unused bit combinations in the BCD code.

4-13 Design a code converter that converts a decimal digit from the 8 4 −2 −1 code to BCD (see Table 1-2.)

4-14 Design a combinational circuit that converts a decimal digit from the 2 4 2 1 code to the 8 4 −2 −1 code (see Table 1-2.)

4-15 Design a combinational circuit that converts a binary number of four bits to a decimal number in BCD. Note that the BCD number is the same as the binary number as long as the input is less than or equal to 9. The binary number from 1010 to 1111 converts into BCD numbers from 1 0000 to 1 0101.

4-16 A BCD-to-seven-segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in a display indicator used for displaying the decimal digit in a familiar form. The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display, as shown in Fig. P4-16(a). The numeric designation chosen to represent the decimal digit is shown in Fig. P4-16(b). Design the BCD-to-seven-segment decoder using a minimum number of NAND gates. The six invalid combinations should result in a blank display.



(a) Segment designation (b) Numerical designation for display

FIGURE P4-16

4-17 Analyze the two-output combinational circuit shown in Fig. P4-17. Find the Boolean functions for the two outputs as a function of the three inputs and explain the circuit operation.

4-18 Derive the truth table of the circuit shown in Fig. P4-17.

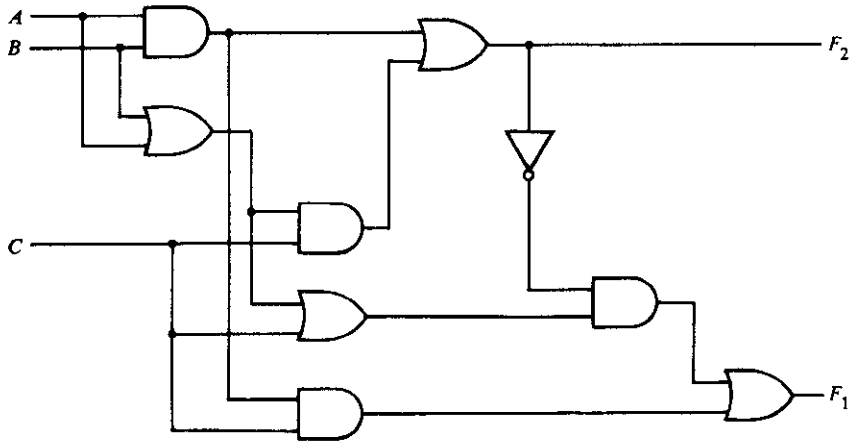


FIGURE P4-17

4-19 Draw the NAND logic diagram for each of the following expressions using multiple-level NAND gate circuits:
(a) $(AB' + CD')E + BC(A + B)$
(b) $w(x + y + z) + xyz$

4-20 Convert the logic diagram of the code converter shown in Fig. 4-8 to a multiple-level NAND circuit.

4-21 Determine the Boolean functions for outputs F and G as a function of four inputs, A, B, C, and D, in Fig. P4-21.

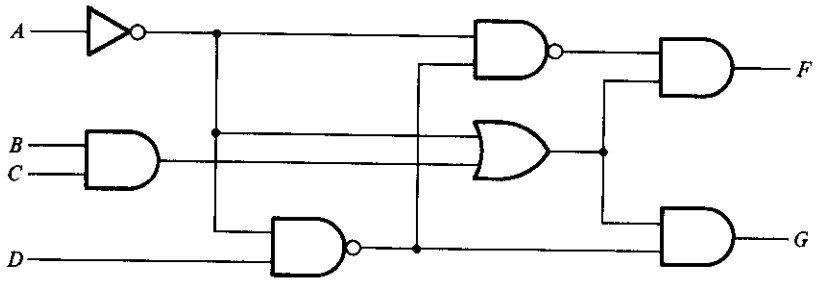


FIGURE P4-21

4-22 Verify that the circuit of Fig. P4-22 generates the exclusive-NOR function.

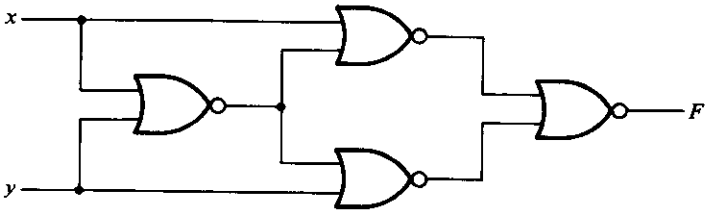


FIGURE P4-22

4-23 Convert the logic diagram of the code converter shown in Fig. 4-8 to a multiple-level NOR circuit.

4-24 Derive the truth table for the output of each NOR gate in Fig. 4-20(a).

4-25 Prove that $x' \oplus y = x \oplus y' = (x \oplus y)' = xy + x'y'$.

4-26 Prove that $x \oplus 1 = x'$ and $x \oplus 0 = x$.

4-27 Show that if $xy = 0$, then $x \oplus y = x + y$.

4-28 Design a combinational circuit that converts a 4-bit Gray code number (Table 1-4) to a 4-bit straight binary number. Implement the circuit with exclusive-OR gates.

4-29 Design the circuit of a 3-bit parity generator and the circuit of a 4-bit parity checker using an odd parity bit.

4-30 Manipulate the following Boolean expression in such a way so that it can be implemented using exclusive-OR and AND gates only.

$$AB'CD' + A'BCD' + AB'C'D + A'BC'D$$

MSI and PLD Components

5-1 INTRODUCTION

The purpose of Boolean-algebra simplification is to obtain an algebraic expression that, when implemented, results in a low-cost circuit. However, the criteria that determine a low-cost circuit must be defined if we are to evaluate the success of the achieved simplification. The design procedure for combinational circuits presented in Section 4-2 minimizes the number of gates required to implement a given function. This procedure assumes that given two circuits that perform the same function, the one that requires fewer gates is preferable because it will cost less. This is not necessarily true when integrated circuits are used.

The circuit complexity of integrated circuits (ICs) has been classified in Section 2-8 as having four levels of integration: small- (SSI), medium- (MSI), large- (LSI), and very large- (VLSI) scale integration. A combinational circuit designed with individual gates can be implemented with SSI circuits that contain several independent gates. The number of gates in an SSI circuit is limited by the number of pins in the package, typically 14 or 16. Since several gates are included in a single IC package, it becomes economical to use as many of the gates from an already used package even if, by doing so, we increase the total number of gates. Moreover, some interconnections among the gates in many ICs are internal to the chip and it is more economical to use as many internal interconnections as possible in order to minimize the number of wires between package pins. With integrated circuits, it is not the count of gates that determines the cost, but the number and types of ICs employed and the number of interconnections needed to implement the given digital circuit.

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as MSI components. MSI components perform specific digital functions commonly needed in the design of digital systems. In this chapter we introduce the most important combinational circuit-type MSI components that are readily available in IC packages. These are adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are also used as standard modules within more complex LSI and VLSI circuits. The MSI components presented here provide a catalog of elementary digital modules used extensively as basic building blocks in the design of digital computers and systems.

The components of a digital system can be classified as being specific to an application or as being standard circuits. Standard components are taken from a set that has been used in other systems. MSI components are standard circuits and their use results in a significant reduction in the total cost as compared to the cost of using SSI circuits. In contrast, specific components are particular to the system being implemented and are not commonly found among the standard components. The implementation of specific circuits with LSI chips can be done by means of ICs that can be programmed to provide the required logic.

A programmable logic device (PLD) is an integrated circuit with internal logic gates that are connected through electronic fuses. Programming the device involves the blowing of fuses along the paths that must be disconnected so as to obtain a particular configuration. The word "programming" here refers to a hardware procedure that specifies the internal configuration of the device. The gates in a PLD are divided into an AND array and an OR array that are connected together to provide an AND-OR sum of product implementation. The initial state of a PLD has all the fuses intact. Programming the device involves the blowing of internal fuses to achieve a desired logic function.

In this chapter we introduce three programmable logic devices and establish procedures for their use in the design of digital systems. The three types of PLDs differ in the placement of fuses in the AND-OR array. Figure 5-1 shows the fuse locations of the three PLDs. The programmable read-only memory (PROM) has a fixed AND array and programmable fuses for the output OR gates. The PROM implements Boolean functions in sum of minterms, as explained in Section 5-7. The programmable array logic (PAL) has a fused programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions that are logically summed in each OR gate. PALs are presented in Section 5-9. The most flexible PLD is the programmable logic array (PLA), where both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum of products implementation. The operation of the PLA is explained in Section 5-8.

The advantage of using PLDs in the design of digital systems is that they can be programmed to incorporate complex logic functions within one LSI circuit. The use of programmable logic devices is an alternative to another design technology called VLSI design. VLSI design refers to the design of digital systems that contain thousands of

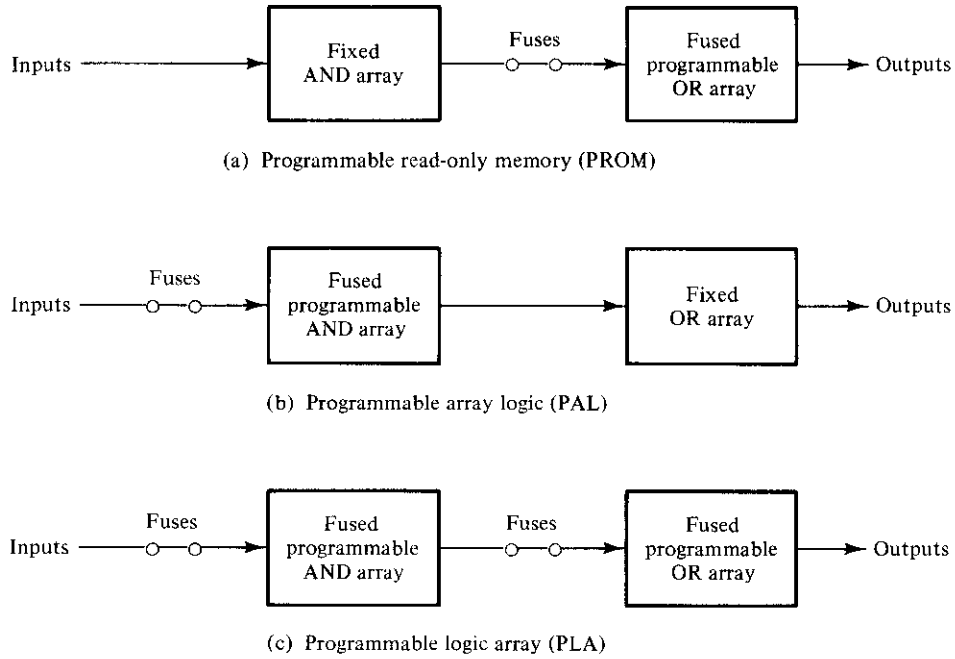


FIGURE 5-1
Basic configuration of three PLDs

gates within a single integrated-circuit chip. The basic component used in VLSI design is the *gate array*. A gate array consists of a pattern of gates fabricated in an area of silicon that is repeated thousands of times until the entire chip is covered with identical gates. Arrays of 1000 to 10,000 gates can be fabricated within a single integrated-circuit chip, depending on the technology used. The design with gate arrays requires that the designer specify the layout of the chip and the way that the gates are routed and connected. The first few levels of the fabrication process are common and independent of the final logic function. Additional fabrication levels are required to interconnect the gates in order to realize the desired function. This is usually done by means of computer-aided design methods. Both the gate array and the programmable logic device require extensive computer software tools to facilitate the design procedure.

5-2 BINARY ADDER AND SUBTRACTOR

The full-adder introduced in Section 4-3 forms the sum of two bits and a previous carry. Two binary numbers of n bits each can be added by means of this circuit. To demonstrate with a specific example, consider two binary numbers, $A = 1011$ and $B = 0011$, whose sum is $S = 1110$. When a pair of bits are added through a full-adder, the circuit produces a carry to be used with the pair of bits one significant position higher. This is shown in the following table:

Subscript i	4	3	2	1		Full-adder of Fig. 4-5
Input carry	0	1	1	0	C_i	z
Augend	1	0	1	1	A_i	x
Addend	0	0	1	1	B_i	y
Sum	1	1	1	0	S_i	S
Output carry	0	0	1	1	C_{i+1}	C

The bits are added with full-adders, starting from the least significant position (subscript 1), to form the sum bit and carry bit. The inputs and outputs of the full-adder circuit of Fig. 4-5 are also indicated. The input carry C_1 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full-adder. This value is transferred into the input carry of the full-adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

The sum of two n-bit binary numbers, A and B, can be generated in two ways: either in a serial fashion or in parallel. The serial addition method uses only one full-adder circuit and a storage device to hold the generated output carry. The pair of bits in A and B are transferred serially, one at a time, through the single full-adder to produce a string of output bits for the sum. The stored output carry from one pair of bits is used as an input carry for the next pair of bits. The parallel method uses n full-adder circuits, and all bits of A and B are applied simultaneously. The output carry from one full-adder is connected to the input carry of the full-adder one position to its left. As soon as the carries are generated, the correct sum bits emerge from the sum outputs of all full-adders.

Binary Parallel Adder

A binary parallel adder is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full-adders connected in a chain, with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

Figure 5-2(a) shows the interconnection of four full-adder (FA) circuits to provide a 4-bit binary parallel adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the adder is C_1 and the output carry is C_5. The S outputs generate the required sum bits. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augend bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries.

An n-bit parallel adder requires n full-adders. It can be constructed from 4-bit, 2-bit, and 1-bit full-adders ICs by cascading several packages. The output carry from one

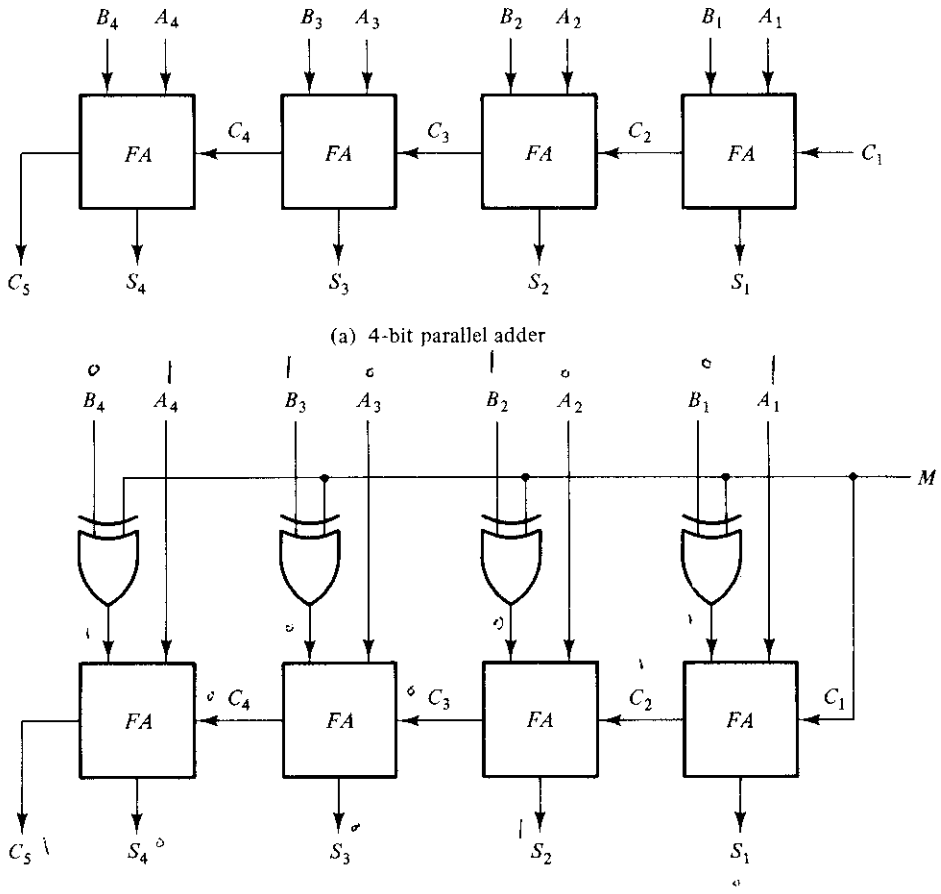


FIGURE 5-2
Adder and subtractor circuits

package must be connected to the input carry of the one with the next higher-order bits.

The 4-bit full-adder is a typical example of an MSI function. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading an already known function, we were able to obtain a simple and well-organized implementation.

Binary Adder-Subtractor

The subtraction of binary numbers can be done most conveniently by means of complements, as discussed in Section 1-5. 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 ob-

tained 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.

The circuit for subtracting $A - B$ consists of a parallel adder with inverters placed between each data input B and the corresponding input of the full-adder. The input carry C_1 must be equal to 1 when performing subtraction. The operation thus performed becomes A plus the 1's complement of B plus 1. This is equal to A plus the 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 $A < B$ (see Section 1-5). For signed numbers, the result is $A - B$ provided there is no overflow. (See Section 1-6.)

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 5-2(b). 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 $C_1 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and the addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate times the number of gate levels in the circuit. The longest propagation delay time in a parallel adder is the time it takes the carry to propagate through the full-adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i in any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. Consider output S_4 in Fig. 5-2(a). Inputs A_4 and B_4 reach a steady value as soon as input signals are applied to the adder. But input carry C_4 does not settle to its final steady-state value until C_3 is available in its steady-state value. Similarly, C_3 has to wait for C_2, and so on down to C_1. Thus, only after the carry propagates through all stages will the last output S_4 and carry C_5 settle to their final steady-state value.

The number of gate levels for the carry propagation can be found from the circuit of the full-adder. This circuit was derived in Fig. 4-5 and is redrawn in Fig. 5-3 for convenience. The input and output variables use the subscript i to denote a typical stage in the parallel adder. The signals at P_i and G_i settle to their steady-state values after the propagation through their respective gates. These two signals are common to all full-adders and depend only on the input augend and addend bits. The signal from the input carry, C_i, to the output carry, C_{i+1}, propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full-adders in the parallel adder, the

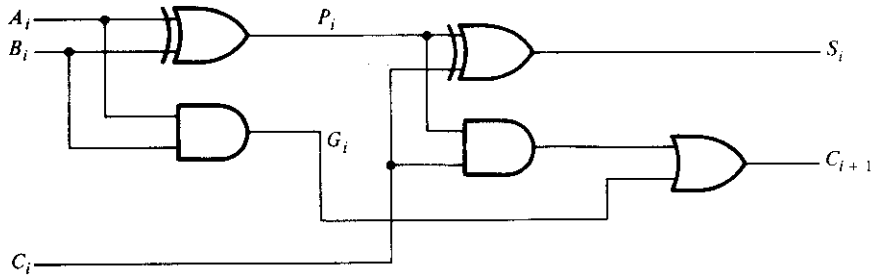


FIGURE 5-3
Full-adder circuit

output carry C_5 would have $2 \times 4 = 8$ gate levels from C_1 to C_5. The total propagation time in the adder would be the propagation time in one half-adder plus eight gate levels. For an n-bit parallel adder, there are $2n$ gate levels for the carry to propagate through.

The carry propagation time is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. But physical circuits have a limit to their capability. Another solution is to increase the equipment complexity in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *look-ahead* carry and is described below.

Consider the circuit of the full-adder shown in Fig. 5-3. If we define two new binary variables:

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

G_i is called a *carry generate* and it produces an output carry when both A_i and B_i are one, regardless of the input carry. P_i is called a *carry propagate* because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

We now write the Boolean function for the carry output of each stage and substitute for each C_i its value from the previous equations:

$$C_2 = G_1 + P_1C_1$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1C_1) = G_2 + P_2G_1 + P_2P_1C_1$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1C_1$$

Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND). The three Boolean functions for C_2, C_3, and C_4 are implemented in the look-ahead carry generator shown in Fig. 5-4. Note that C_4 does not have to wait for C_3 and C_2 to propagate; in fact, C_4 is propagated at the same time as C_2 and C_3.

The construction of a 4-bit parallel adder with a look-ahead carry scheme is shown in Fig. 5-5. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. All the P's and G's are generated in two gate levels. The carries are propagated through the look-ahead carry generator (similar to that in Fig. 5-4) and applied as in-

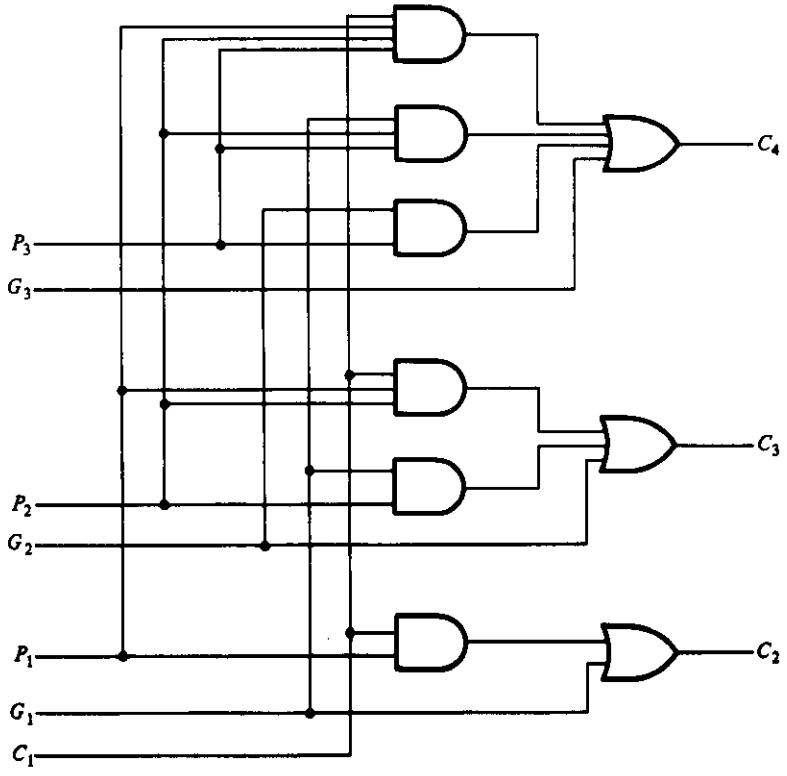


FIGURE 5-4
Logic diagram of a look-ahead carry generator

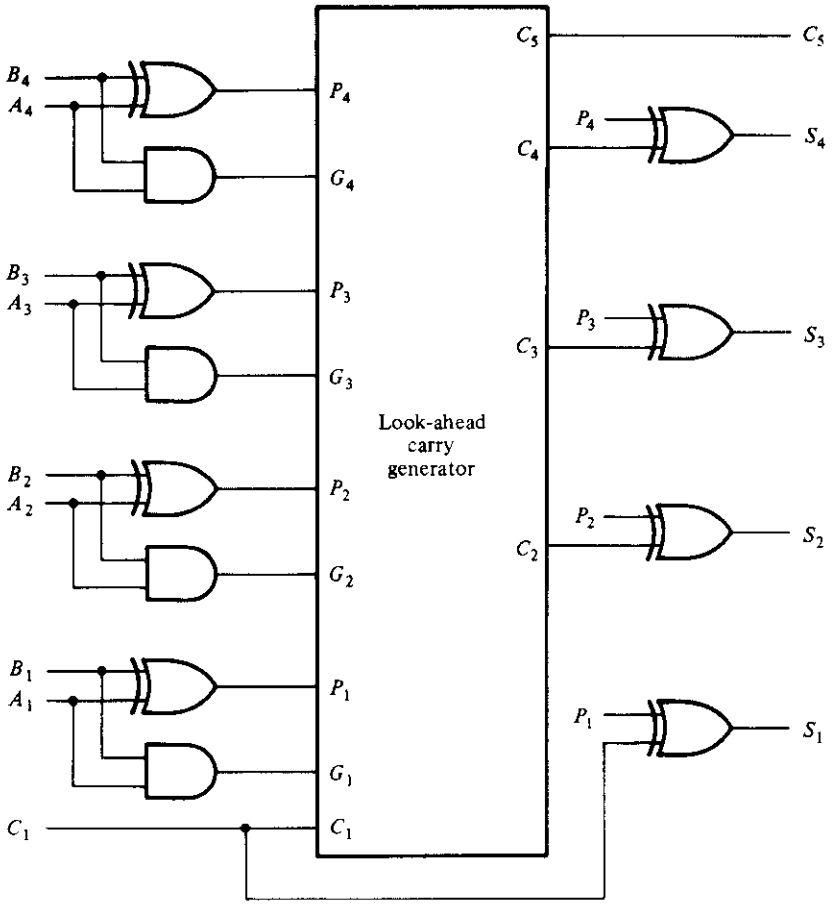


FIGURE 5-5
4-bit full-adders with look-ahead carry

puts to the second exclusive-OR gate. After the P and G signals settle into their steady-state values, all output carries are generated after a delay of two levels of gates. Thus, outputs S_2 through S_4 have equal propagation delay times. The two-level circuit for the output carry C_5 is not shown in Fig. 5-4. This circuit can be easily derived by the equation-substitution method, as done above (see Problem 5-8).

5-3 DECIMAL ADDER

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary-coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and

present results in the accepted code. For binary addition, it was sufficient to consider a pair of significant bits at a time, together with a previous carry. A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input carry and output carry. Of course, there is a wide variety of possible decimal adder circuits, dependent upon the code used to represent the decimal digits.

The design of a nine-input, five-output combinational circuit by the classical method requires a truth table with $2^9 = 512$ entries. Many of the input combinations are don't-care conditions, since each binary code input has six combinations that are invalid. The simplified Boolean functions for the circuit may be obtained by a computer-generated tabular method, and the result would probably be a connection of gates forming an irregular pattern. An alternate procedure is to add the numbers with full-adder circuits, taking into consideration the fact that six combinations in each 4-bit input are not used. The output must be modified so that only those binary combinations that are valid combinations of the decimal code are generated.

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 5-1 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal digits* must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD-digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

TABLE 5-1
Derivation of a BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

A *BCD adder* is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder, as shown in Fig. 5-6. The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

The BCD adder can be constructed with three IC packages. Each of the 4-bit adders is an MSI function and the three gates for the correction logic need one SSI package. However, the BCD adder is available in one MSI circuit. To achieve shorter propagation delays, an MSI BCD adder includes the necessary circuits for look-ahead carries. The adder circuit for the correction does not need all four full-adders, and this circuit can be optimized within the IC package.

A decimal parallel adder that adds n decimal digits needs n BCD adder stages. The

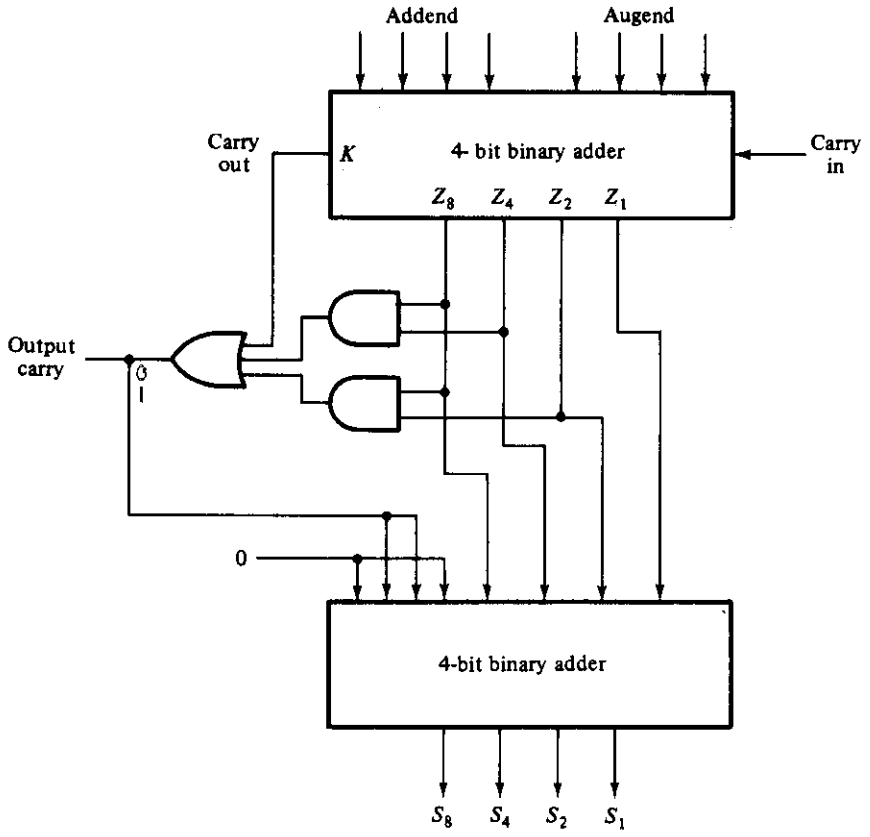


FIGURE 5-6
Block diagram of a BCD adder

output carry from one stage must be connected to the input carry of the next higher-order stage.

5-4 MAGNITUDE COMPARATOR

The comparison of two numbers is an operation that determines if one number is greater than, less than, or equal to the other number. A *magnitude comparator* is a combinational circuit that compares two numbers, A and B, and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

The circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome even with $n = 3$. On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions that possess an inherent well-defined regularity can usually be designed by means of an al-

gorithmic procedure if one is found to exist. An *algorithm* is a procedure that specifies a finite set of steps that, if followed, give the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a 4-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. Consider two numbers, A and B, with four digits each. Write the coefficients of the numbers with descending significance as follows:

$$A = A_3A_2A_1A_0$$

$$B = B_3B_2B_1B_0$$

where each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal, i.e., if $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$ and $A_0 = B_0$. When the numbers are binary, the digits are either 1 or 0 and the equality relation of each pair of bits can be expressed logically with an equivalence function:

$$x_i = A_iB_i + A_i'B_i' \qquad i = 0, 1, 2, 3$$

where $x_i = 1$ only if the pair of bits in position i are equal, i.e., if both are 1's or both are 0's.

The equality of the two numbers, A and B, is displayed in a combinational circuit by an output binary variable that we designate by the symbol $(A = B)$. This binary variable is equal to 1 if the input numbers, A and B, are equal, and it is equal to 0 otherwise. For the equality condition to exist, all x_i variables must be equal to 1. This dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

The *binary* variable $(A = B)$ is equal to 1 only if all pairs of digits of the two numbers are equal.

To determine if A is greater than or less than B, we inspect the relative magnitudes of pairs of significant digits starting from the most significant position. If the two digits are equal, we compare the next lower significant pair of digits. This comparison continues until a pair of unequal digits is reached. If the corresponding digit of A is 1 and that of B is 0, we conclude that $A > B$. If the corresponding digit of A is 0 and that of B is 1, we have that $A < B$. The sequential comparison can be expressed logically by the following two Boolean functions:

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

The symbols $(A > B)$ and $(A < B)$ are *binary* output variables that are equal to 1 when $A > B$ or $A < B$, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The "unequal" outputs can use the same gates that are needed to generate the "equal" output. The logic diagram of the 4-bit magnitude comparator is shown in Fig. 5-7. The four x outputs are generated with

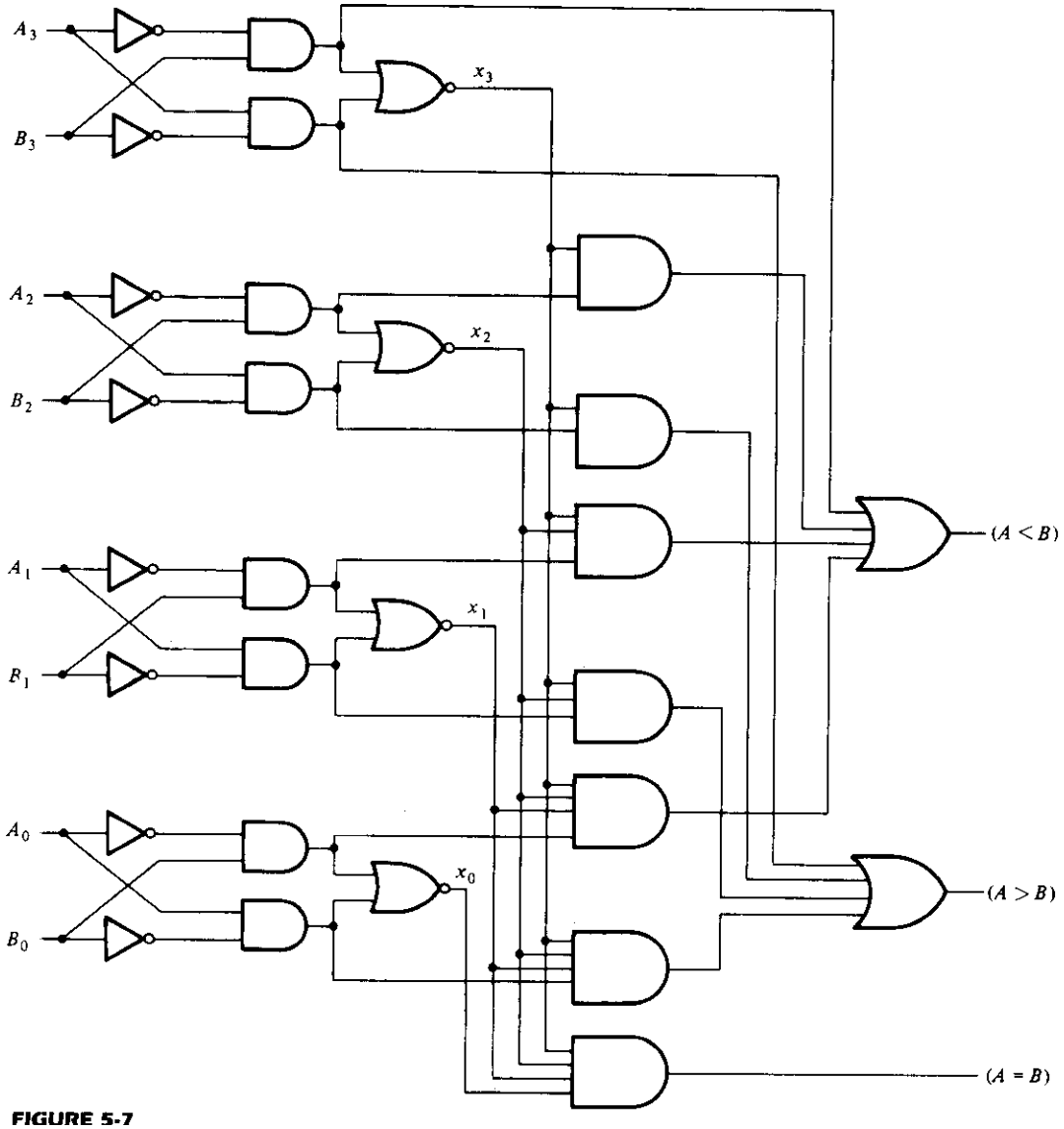


FIGURE 5-7
4-bit magnitude comparator

equivalence (exclusive-NOR) circuits and applied to an AND gate to give the output binary variable $(A = B)$. The other two outputs use the x variables to generate the Boolean functions listed before. This is a multilevel implementation and, as clearly seen, it has a regular pattern. The procedure for obtaining magnitude-comparator circuits for binary numbers with more than four bits should be obvious from this example. The same circuit can be used to compare the relative magnitudes of two BCD digits.

5-5 DECODERS AND ENCODERS

Discrete quantities of information are represented in digital systems 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 n input lines to a maximum of 2^n unique output lines. If the n-bit decoded information has unused or don't-care combinations, the decoder output will have fewer than 2^n outputs.

The decoders presented here are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. The name *decoder* is also used in conjunction with some code converters such as a BCD-to-seven-segment decoder.

As an example, consider the 3-to-8-line decoder circuit of Fig. 5-8. The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder would be a binary-to-octal conversion. The input variables may represent a binary number, and the outputs will then represent the eight digits in the octal number

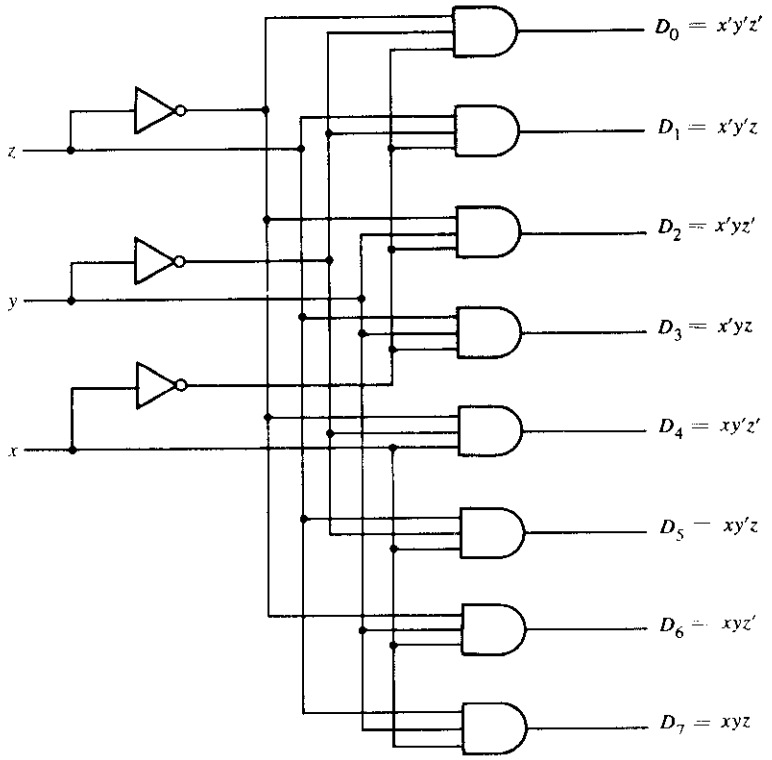


FIGURE 5-8
A 3-to-8 line decoder

TABLE 5-2
Truth Table of a 3-to-8-Line Decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

system. However, a 3-to-8-line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be further clarified from its input–output relationship, listed in Table 5-2. Observe that the output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

Combinational Logic Implementation

A decoder provides the 2^n minterm of n input variables. Since any Boolean function can be expressed in sum of minterms canonical form, one can use a decoder to generate the minterms and an external OR gate to form the sum. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n-line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean functions for the circuit be expressed in sum of minterms. This form can be easily obtained from the truth table or by expanding the functions to their sum of minterms (see Section 2-5). A decoder is then chosen that generates all the minterms of the n input variables. The inputs to each OR gate are selected from the decoder outputs according to the minterm list in each function.

Example 5-1

Implement a full-adder circuit with a decoder and two OR gates.

From the truth table of the full-adder (Section 4-3), we obtain the functions for this combinational circuit in sum of minterms:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a 3-to-8-line decoder. The implementation is shown in Fig. 5-9. The decoder generates the eight

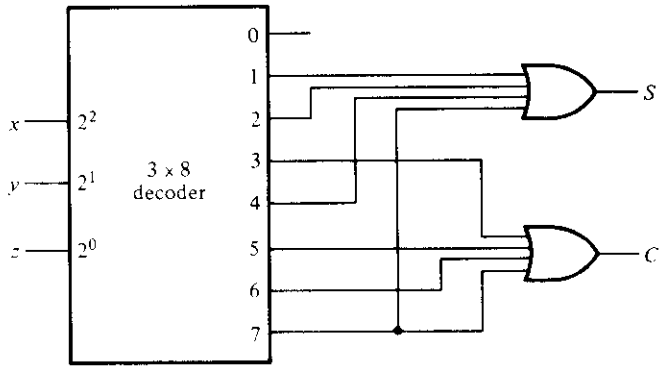


FIGURE 5-9
Implementation of a full-adder with a decoder

minterms for x, y, z. The OR gate for output S forms the sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the sum of minterms 3, 5, 6, and 7. ■

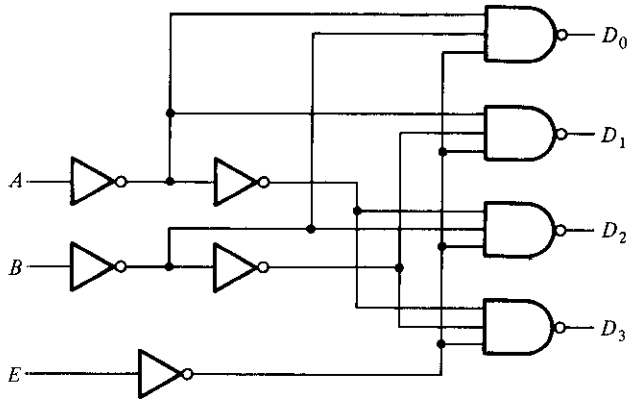
A function with a long list of minterms requires an OR gate with a large number of inputs. A function F having a list of k minterms can be expressed in its complemented form F' with $2^n - k$ minterms. If the number of minterms in a function is greater than $2^n/2$, then F' can be expressed with fewer minterms than required for F. In such a case, it is advantageous to use a NOR gate to sum the minterms of F'. The output of the NOR gate will generate the normal output F.

The decoder method can be used to implement any combinational circuit. However, its implementation must be compared with all other possible implementations to determine the best solution. In some cases, this method may provide the best implementation, especially if the combinational circuit has many outputs and if each output function (or its complement) is expressed with a small number of minterms.

Demultiplexers

Some IC decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Most, if not all, IC decoders include one or more *enable* inputs to control the circuit operation. A 2-to-4-line decoder with an enable input constructed with NAND gates is shown in Fig. 5-10. All outputs are equal to 1 if enable input E is 1, regardless of the values of inputs A and B. When the enable input is 0, the circuit operates as a decoder with complemented outputs. The truth table lists these conditions. The X's under A and B are don't-care conditions. Normal decoder operation occurs only with $E = 0$, and the outputs are selected when they are in the 0 state.

The block diagram of the decoder is shown in Fig. 5-11(a). The small circle at input E indicates that the decoder is enabled when $E = 0$. The small circles at the outputs indicate that all outputs are complemented.



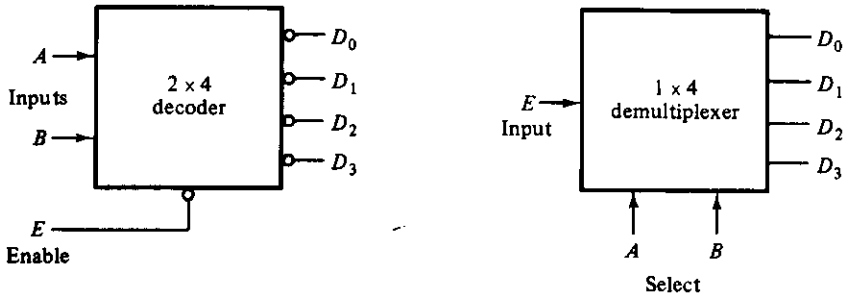
(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
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

(b) Truth table

FIGURE 5-10
A 2-to-4-line decoder with enable (E) input

A decoder with an enable input can function as a demultiplexer. A *demultiplexer* is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by the bit values of n selection lines. The decoder of Fig. 5-10 can function as a demultiplexer if the E line is taken as a data input line and lines A and B are taken as the selection lines. This is shown in Fig. 5-11(b). The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary value of the two selection lines, A and B. This can be verified from the truth table of this circuit, shown in Fig. 5-10(b). For example, if the selection lines $AB = 10$, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder/demultiplexer*. It is the enable input that makes the circuit a demultiplexer; the decoder itself can use AND, NAND, or NOR gates.



(a) Decoder with enable (b) Demultiplexer

FIGURE 5-11
Block diagrams for the circuit of Fig. 5-10

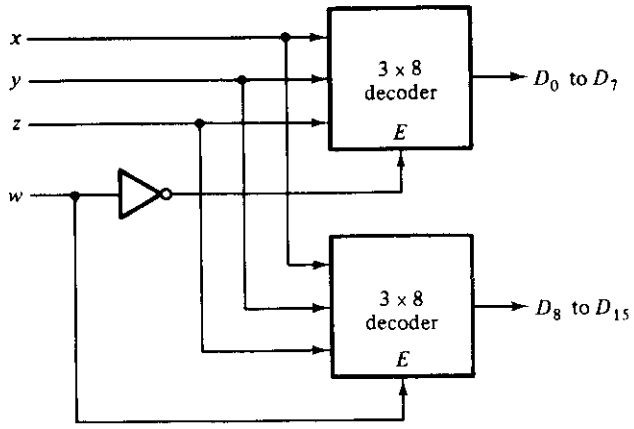


FIGURE 5-12
A 4 × 16 decoder constructed with two 3 × 8 decoders

Decoder/demultiplexer circuits can be connected together to form a larger decoder circuit. Figure 5-12 shows two 3 × 8 decoders with enable inputs connected to form a 4 × 16 decoder. When $w = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w = 1$, the enable conditions are reversed; the bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in ICs. In general, enable lines are a convenient feature for connecting two or more IC packages for the purpose of expanding the digital function into a similar function with more inputs and outputs.

Encoders

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output 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 5-3. 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.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following output Boolean functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

TABLE 5-3
Truth Table of Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder is implemented with three OR gates, as shown in Fig. 5-13.

The encoder defined in Table 5-3 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. This does not represent binary 3 nor binary 6. To resolve this ambiguity, encoder circuits must establish a priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0. The problem is that an output with all 0's is also generated when D_0 is equal to 1. This ambiguity can be resolved by providing an additional output that specifies the condition that none of the inputs are active.

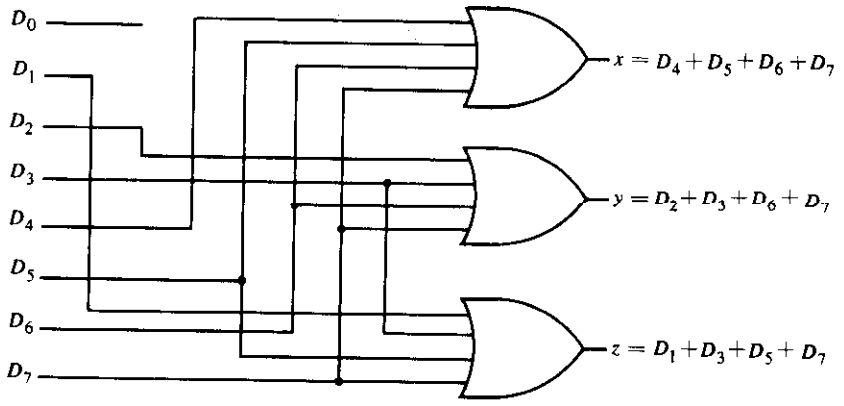


FIGURE 5-13
Octal-to-binary encoder

TABLE 5-4
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Priority Encoder

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 5-4. The X's are don't-care conditions that designate the fact that the binary value may be equal either to 0 or 1. Input D_3 has the highest priority; so regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$ provided that $D_3 = 0$, regardless of the values of the other two lower-priority inputs. The output for D_1 is generated only if higher-priority inputs are 0, and so on down the priority level. A *valid*-output indicator, designated by V, is set to 1 only when one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs of the circuit are not used.

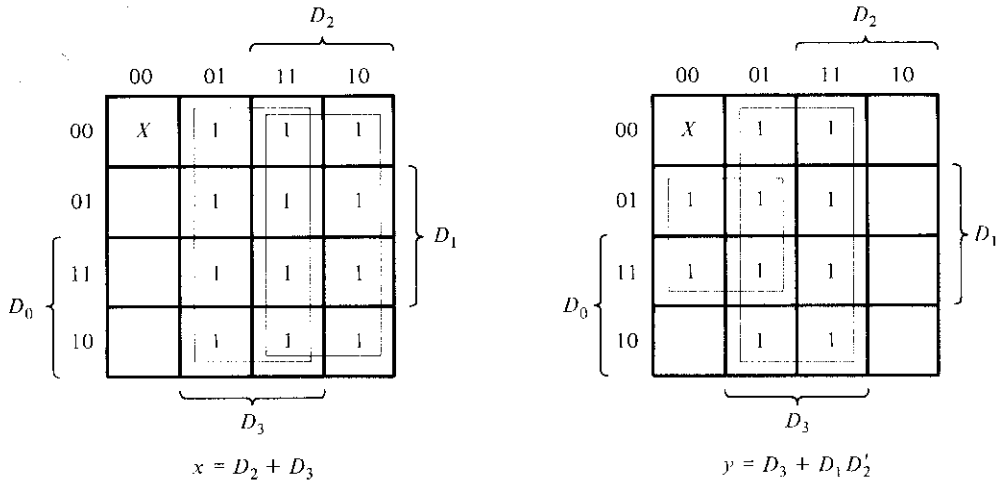


FIGURE 5-14
Maps for a priority encoder

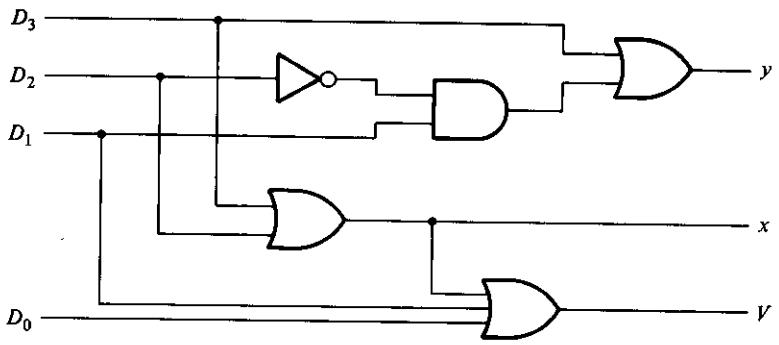


FIGURE 5-15
4-input priority encoder

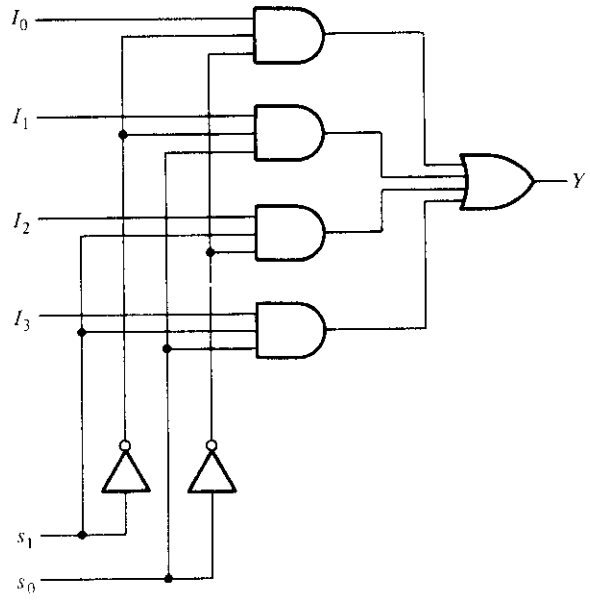
The maps for simplifying outputs x and y are shown in Fig. 5-14. The minterms for the two functions are derived from Table 5-4. Although the table has only five rows, when each don't-care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the third row in the table with $X100$ represents minterms 0100 and 1100 since X can be assigned either 0 or 1. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 5-15 according to the following Boolean functions:

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

5-6 MULTIPLEXERS

Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. A *digital multiplexer* is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

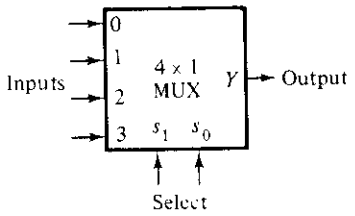
A 4-to-1-line multiplexer is shown in Fig. 5-16. Each of the four input lines, I_0 to I_3, is applied to one input of an AND gate. Selection lines s_1 and s_0 are decoded to select a particular AND gate. The function table, Fig. 5-16(b), lists the input-to-output path for each possible bit combination of the selection lines. When this MSI function is used in the design of a digital system, it is represented in block diagram form, as shown in Fig. 5-16(c). 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 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table



(c) Block diagram

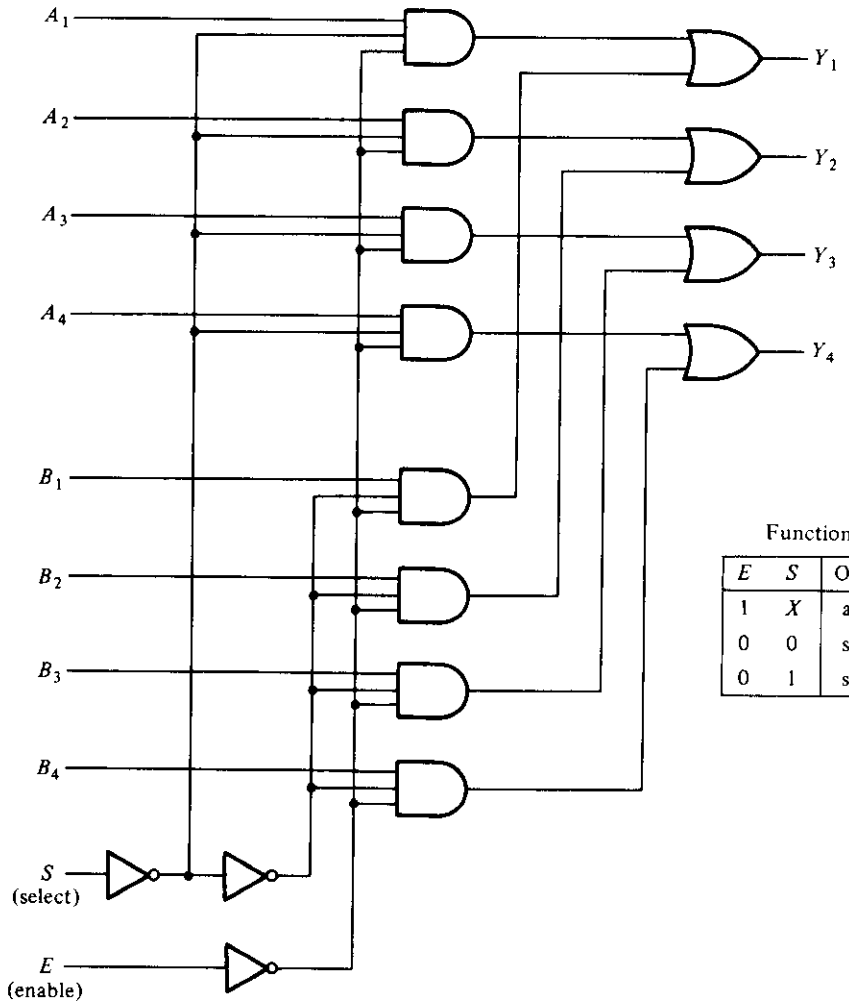
FIGURE 5-16
A 4-to-1-line multiplexer

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

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-1-line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1-line output. The size of a multiplexer is specified by the number 2^n of its input lines and the single output line. It is then implied that it also contains n selection lines. A multiplexer is often abbreviated as MUX.

As in decoders, multiplexer ICs may have an *enable* input to control the operation of the unit. When the enable input is in a given binary state, the outputs are disabled, and when it is in the other state (the enable state), the circuit functions as a normal multiplexer. The enable input (sometimes called *strobe*) can be used to expand two or more multiplexer ICs to a digital multiplexer with a larger number of inputs.

In some cases, two or more multiplexers are enclosed within one IC package. The selection and enable inputs in multiple-unit ICs may be common to all multiplexers. As an illustration, a quadruple 2-to-1-line multiplexer IC is shown in Fig. 5-17. It has four multiplexers, each capable of selecting one of two input lines. Output Y_1 can be selected



Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

FIGURE 5-17
Quadruple 2-to-1-line multiplexer

to be equal to either A_1 or B_1. Similarly, output Y_2 may have the value of A_2 or B_2, and so on. One input selection line, S, suffices to select one of two lines in all four multiplexers. The control input E enables the multiplexers in the 0 state and disables them in the 1 state. Although the circuit contains four multiplexers, we may think of it as a circuit that selects one in a pair of 4-input lines. As shown in the function table, the unit is selected when $E = 0$. Then, if $S = 0$, the four A inputs have a path to the outputs. On the other hand, if $S = 1$, the four B inputs are selected. The outputs have all 0's when $E = 1$, regardless of the value of S.

Boolean-Function Implementation

It was shown in the previous section that a decoder can be used to implement a Boolean function by employing an external OR gate. A quick reference to the multiplexer of Fig. 5-16 reveals that it is essentially a decoder with the OR gate already available. The minterms out of the decoder to be chosen can be controlled with the input lines. The minterms to be included with the function being implemented are chosen by making their corresponding input lines equal to 1; those minterms not included in the function are disabled by making their input lines equal to 0. This gives a method for implementing any Boolean function of n variables with a 2^n-to-1 multiplexer. However, it is possible to do better than that.

If we have a Boolen function of $n + 1$ variables, we take n of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the selection lines, one can implement any Boolean function with a multiplexer. In this way, it is possible to generate any function of $n + 1$ variables with a 2^n-to-1 multiplexer.

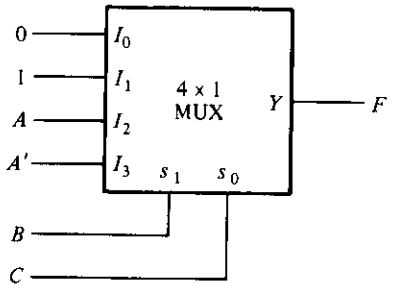
To demonstrate this procedure with a concrete example, consider the function of three variables:

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

The function can be implemented with a 4-to-1 multiplexer, as shown in Fig. 5-18. Two of the variables, B and C, are applied to the selection lines in that order, i.e., B is connected to s_1 and C to s_0. The inputs of the multiplexer are 0, 1, A, and A'. When $BC = 00$, output $F = 0$ since $I_0 = 0$. Therefore, both minterms $m_0 = A'B'C'$ and $m_4 = AB'C'$ produce a 0 output, since the output is 0 when $BC = 00$ regardless of the value of A. When $BC = 01$, output $F = 1$, since $I_1 = 1$. Therefore, both minterms $m_1 = A'B'C$ and $m_5 = AB'C$ produce a 1 output, since the output is 1 when $BC = 01$ regardless of the value of A. When $BC = 10$, input I_2 is selected. Since A is connected to this input, the output will be equal to 1 only for minterm $m_6 = ABC'$, but not for minterm $m_2 = A'BC'$, because when $A' = 1$, then $A = 0$, and since $I_2 = 0$, we have $F = 0$. Finally, when $BC = 11$, input I_3 is selected. Since A' is connected to this input, the output will be equal to 1 only for minterm $m_3 = A'BC$, but not for $m_7 = ABC$. This information is summarized in Fig. 5-18(b), which is the truth table of the function we want to implement.

This discussion shows by analysis that the multiplexer implements the required function. We now present a general procedure for implementing any Boolean function of n variables with a 2^{n-1}-to-1 multiplexer.

First, express the function in its sum of minterms form. Assume that the ordered sequence of variables chosen for the minterms is $ABCD \ldots$, where A is the leftmost variable in the ordered sequence of n variables and $BCD \ldots$ are the remaining $n - 1$ variables. Connect the $n - 1$ variables to the selection lines of the multiplexer, with B connected to the high-order selection line, C to the next lower selection line, and so on



(a) Multiplexer implementation

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

(c) Implementation table

FIGURE 5-18
Implementing $F(A, B, C) = \sum (1, 3, 5, 6)$ with a multiplexer

down to the last variable, which is connected to the lowest-order selection line s_0. Consider now the single variable A. Since this variable is in the highest-order position in the sequence of variables, it will be complemented in minterms 0 to $(2^n/2) - 1$, which comprise the first half in the list of minterms. The second half of the minterms will have their A variable uncomplemented. For a three-variable function, A, B, C, we have eight minterms. Variable A is complemented in minterms 0 to 3 and uncomplemented in minterms 4 to 7.

List the inputs of the multiplexer and under them list all the minterms in two rows. The first row lists all those minterms where A is complemented, and the second row all the minterms with A uncomplemented, as shown in Fig. 5-18(c). Circle all the minterms of the function and inspect each column separately.

If the two minterms in a column are not circled, apply 0 to the corresponding multiplexer input.

If the two minterms are circled, apply 1 to the corresponding multiplexer input.

If the bottom minterm is circled and the top is not circled, apply A to the corresponding multiplexer input.

If the top minterm is circled and the bottom is not circled, apply A' to the corresponding multiplexer input.

This procedure follows from the conditions established during the previous analysis.

Figure 5-18(c) shows the implementation table for the Boolean function

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

from which we obtain the multiplexer connections of Fig. 5-18(a). Note that B must be connected to s_1 and C to s_0.

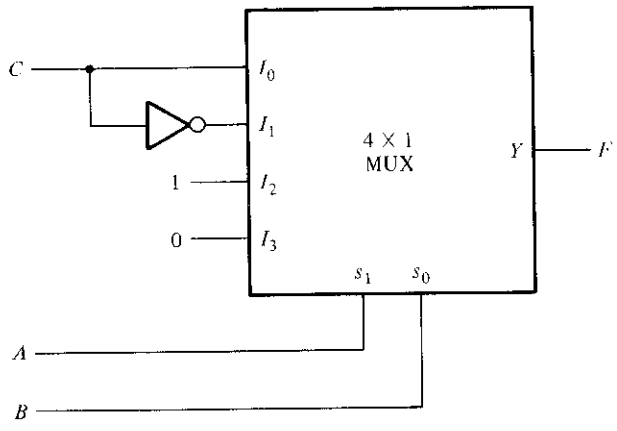
It is not necessary to choose the leftmost variable in the ordered sequence of a variable list for the data inputs of the multiplexer. In fact, any one of the variables can be chosen for the inputs, provided we modify the multiplexer implementation table. Moreover, it is possible to derive the multiplexer circuit directly from the truth table. Consider, for example, the following three-variable Boolean function:

$$F(A, B, C) = \Sigma(1, 2, 4, 5)$$

We wish to implement the function with a multiplexer, but in this case, we will connect variables A and B to selection inputs s_1 and s_0, respectively, and use the rightmost vari-

A	B	C	F	
0	0	0	0	$F = C$
0	0	1	1	
0	1	0	1	$F = C'$
0	1	1	0	
1	0	0	1	$F = 1$
1	0	1	1	
1	1	0	0	$F = 0$
1	1	1	0	

(a) Truth table



(b) Multiplexer implementation

	I_0	I_1	I_2	I_3
C'	0	(2)	(4)	6
C	(1)	3	(5)	7
	C	C'	1	0

(c) Implementation table

FIGURE 5-19
Implementing $F(A, B, C) = \sum (1, 2, 4, 5)$ with a multiplexer

able C for the data inputs of the multiplexer. Figure 5-19(a) is the truth table of the function. The table is divided into sections, with each section having identical values for variables A and B. We note that when $AB = 00$, output F is the same as input C. When $AB = 01$, F is the same as C'. When $AB = 10$, $F = 1$, and when $AB = 11$, $F = 0$. The multiplexer circuit of Fig. 5-19(b) can be derived directly from the truth table without the need of an implementation table. However, if an implementation table is desired, it must be modified to take into account the relationship between the minterms and the inputs of the multiplexer. As seen from the truth table, variable C is complemented in the even-numbered minterms 0, 2, 4, and 6, and uncomplemented in the odd-numbered minterms 1, 3, 5, and 7. The arrangement of the two rows in the implementation table must be as shown in Fig. 5-19(c). By circling the minterms of the function and using the rules stated before, we obtain the multiplexer inputs for implementing the function.

In a similar fashion, it is possible to choose any other variable of the function for the multiplexer data inputs. In any case, all input variables except one are applied to the selection inputs of the multiplexer. The remaining single variable, or its complement, or 0, or 1, is then applied to the data inputs of the multiplexer.

Example 5-2

Implement the following function with a multiplexer:

$$F(A, B, C, D) = \Sigma(0, 1, 3, 4, 8, 9, 15)$$

This is a four-variable function and, therefore, we need a multiplexer with three selection lines and eight inputs. We choose to apply variables B, C, and D to the selection lines. The implementation table is then as shown in Fig. 5-20. The first half of the

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	⓪	①	2	③	④	5	6	7
A	⑧	⑨	10	11	12	13	14	⑮
	1	1	0	A'	A'	0	0	A

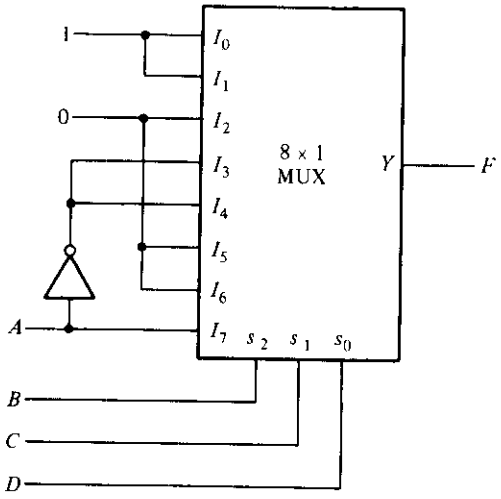


FIGURE 5-20
Implementing $F(A, B, C, D) = \sum (0, 1, 3, 4, 8, 9, 15)$

minterms are associated with A' and the second half with A. By circling the minterms of the function and applying the rules for finding values for the multiplexer inputs, we obtain the implementation shown. ■

Let us now compare the multiplexer method with the decoder method for implementing combinational circuits. The decoder method requires an OR gate for each output function, but only one decoder is needed to generate all minterms. The multiplexer method uses smaller-size units but requires one multiplexer for each output function. It would seem reasonable to assume that combinational circuits with a small number of outputs should be implemented with multiplexers. Combinational circuits with many output functions would probably use fewer ICs with the decoder method.

Although multiplexers and decoders may be used in the implementation of combinational circuits, it must be realized that decoders are mostly used for decoding binary information and multiplexers are mostly used to form a selected path between multiple sources and a single destination.

5-7 READ-ONLY MEMORY (ROM)

We saw in Section 5-5 that a decoder generates the 2^n minterms of the n input variables. By inserting OR gates to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration. The ROM is used to implement complex combinational circuits within one IC package or as permanent storage for binary information.

A ROM is essentially a memory (or storage) device in which permanent binary information is stored. The binary information must be specified by the designer and 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.

A block diagram of a ROM is shown in Fig. 5-21. It consists of n input lines and m output lines. Each bit combination of the input variables is called an *address*. Each bit combination that comes out of the output lines is called a *word*. The number of bits per word is equal to the number of output lines, m. An address is essentially a binary number that denotes one of the minterms of n variables. The number of distinct addresses possible with n input variables is 2^n. An output word can be selected by a unique address, and since there are 2^n distinct addresses in a ROM, there are 2^n distinct words that are said to be stored in the unit. The word available on the output lines at any given time depends on the address value applied to the input lines. A ROM is charac-

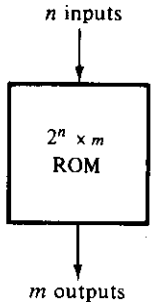


FIGURE 5-21
ROM block diagram

terized by the number of words 2^n and the number of bits per word m. This terminology is used because of the similarity between the read-only memory and the random-access memory, which is presented in Section 7-7.

Consider a 32 × 8 ROM. The unit consists of 32 words of 8 bits each. This means that there are eight output lines and that there are 32 distinct words stored in the unit, each of which may be applied to the output lines. The particular word selected that is presently available on the output lines is determined from the five input lines. There are only five inputs in a 32 × 8 ROM because $2^5 = 32$, and with five variables, we can specify 32 addresses or minterms. For each address input, there is a unique selected word. Thus, if the input address is 00000, word number 0 is selected and it appears on the output lines. If the input address is 11111, word number 31 is selected and applied to the output lines. In between, there are 30 other addresses that can select the other 30 words.

The number of addressed words in a ROM is determined from the fact that n input lines are needed to specify 2^n words. A ROM is sometimes specified by the total number of bits it contains, which is $2^n \times m$. For example, a 2048-bit ROM may be organized as 512 words of 4 bits each. This means that the unit has four output lines and nine input lines to specify $2^9 = 512$ words. The total number of bits stored in the unit is $512 \times 4 = 2048$.

Internally, the ROM is a combinational circuit with AND gates connected as a decoder and a number of OR gates equal to the number of outputs in the unit. Figure 5-22 shows the internal logic construction of a 32 × 4 ROM. The five input variables are decoded into 32 lines by means of 32 AND gates and 5 inverters. Each output of the decoder represents one of the minterms of a function of five variables. Each one of the 32 addresses selects one and only one output from the decoder. The address is a 5-bit number applied to the inputs, and the selected minterm out of the decoder is the one marked with the equivalent decimal number. The 32 outputs of the decoder are connected through fuses to each OR gate. Only four of these fuses are shown in the diagram, but actually each OR gate has 32 inputs and each input goes through a fuse that can be blown as desired.

The ROM is a two-level implementation in sum of minterms form. It does not have to be an AND-OR implementation, but it can be any other possible two-level minterm

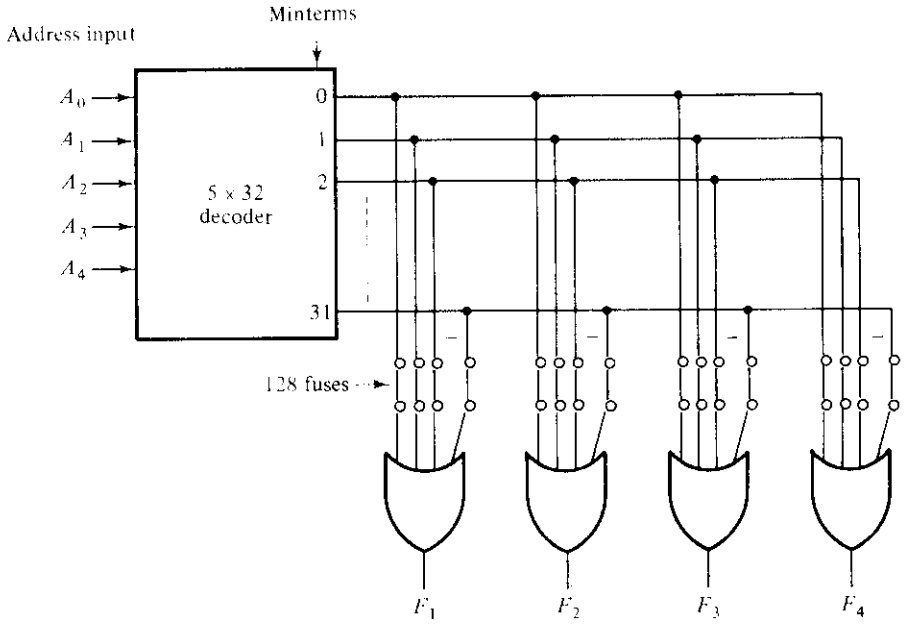


FIGURE 5-22
Logic construction of a 32 × 4 ROM

implementation. The second level is usually a wired-logic connection (see Section 3-7) to facilitate the blowing of fuses.

ROMs have many important applications in the design of digital computer systems. Their use for implementing complex combinational circuits is just one of these applications. Other uses of ROMs are presented in other parts of the book in conjunction with their particular applications.

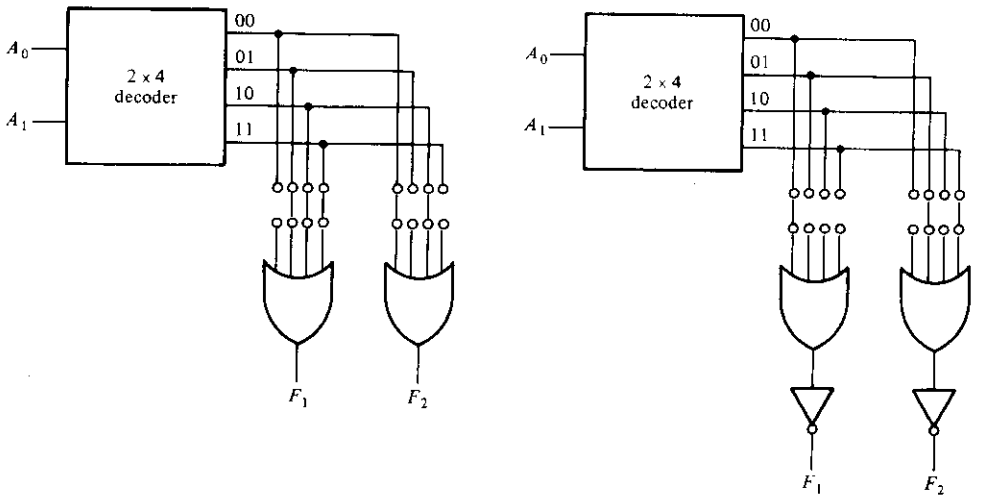
Combinational Logic Implementation

From the logic diagram of the ROM, it is clear that each output provides the sum of all the minterms of the n input variables. Remember that any Boolean function can be expressed in sum of minterms form. By breaking the links of those minterms not included in the function, each ROM output can be made to represent the Boolean function of one of the output variables in the combinational circuit. For an n-input, m-output combinational circuit, we need a $2^n \times m$ ROM. The blowing of the fuses is referred to as *programming* the ROM. The designer need only specify a ROM program table that gives the information for the required paths in the ROM. The actual programming is a hardware procedure that follows the specifications listed in the program table.

Let us clarify the process with a specific example. The truth table in Fig. 5-23(a) specifies a combinational circuit with two inputs and two outputs. The Boolean functions can be expressed in sum of minterms:

A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

(a) Truth table



(b) ROM with AND-OR gates

(c) ROM with AND-OR-INVERT gates

FIGURE 5-23
Combinational-circuit implementation with a 4 × 2 ROM

$$F_1(A_1, A_0) = \Sigma(1, 2, 3)$$
$$F_2(A_1, A_0) = \Sigma(0, 2)$$

When a combinational circuit is implemented by means of a ROM, the functions must be expressed in sum of minterms or, better yet, by a truth table. If the output functions are simplified, we find that the circuit needs only one OR gate and an inverter. Obviously, this is too simple a combinational circuit to be implemented with a ROM. The advantage of a ROM is in complex combinational circuits. This example merely demonstrates the procedure and should not be considered in a practical situation.

The ROM that implements the combinational circuit must have two inputs and two outputs; so its size must be 4 × 2. Figure 5-23(b) shows the internal construction of such a ROM. It is now necessary to determine which of the eight available fuses must be blown and which should be left intact. This can be easily done from the output functions listed in the truth table. Those minterms that specify an output of 0 should not have a path to the output through the OR gate. Thus, for this particular case, the truth table shows three 0's, and their corresponding fuses to the OR gates must be blown. It

is obvious that we must assume here that an open input to an OR gate behaves as a 0 input.

Some ROM units come with an inverter after each of the OR gates and, as a consequence, they are specified as having initially all 0's at their outputs. The programming procedure in such ROMs requires that we open the paths of the minterms (or addresses) that specify an output of 1 in the truth table. The output of the OR gate will then generate the complement of the function, but the inverter placed after the OR gate complements the function once more to provide the normal output. This is shown in the ROM of Fig. 5-23(c).

The previous example demonstrates the general procedure for implementing any combinational circuit with a ROM. From the number of inputs and outputs in the combinational circuit, we first determine the size of ROM required. Then we must obtain the programming truth table of the ROM; no other manipulation or simplification is required. The 0's (or 1's) in the output functions of the truth table directly specify those fuses that must be blown to provide the required combinational circuit in sum of minterms form.

In practice, when one designs a circuit by means of a ROM, it is not necessary to show the internal gate connections of fuses inside the unit, as was done in Fig. 5-23. This was shown there for demonstration purposes only. All the designer has to do is specify the particular ROM (or its designation number) and provide the ROM truth table, as in Fig. 5-23(a). The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the truth table.

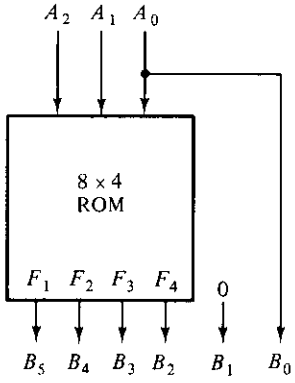
Example 5-3

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

The first step is to derive the truth table for the combinational circuit. In most cases, this is all that is needed. In some cases, we can fit a smaller truth table for the ROM by using certain properties in the truth table of the combinational circuit. Table 5-5 is the

TABLE 5-5
Truth Table for Circuit of Example 5-3

Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



(a) Block diagram

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

FIGURE 5-24
ROM implementation of Example 5-3

truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all possible numbers. We note that output B_0 is always equal to input A_0; so there is no need to generate B_0 with a ROM since it is equal to an input variable. Moreover, output B_1 is always 0, so this output is always known. We actually need to generate only four outputs with the ROM; the other two are easily obtained. The minimum-size ROM needed must have three inputs and four outputs. Three inputs specify eight words, so the ROM size must be 8 × 4. The ROM implementation is shown in Fig. 5-24. The three inputs specify eight words of four bits each. The other two outputs of the combinational circuit are equal to 0 and A_0. The truth table in Fig. 5-24 specifies all the information needed for programming the ROM, and the block diagram shows the required connections. ■

Types of ROMs

The required paths in a ROM may be programmed in two different ways. The first is called *mask programming* and is done by the manufacturer during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table the ROM is to satisfy. The truth table may be submitted on a special form provided by the manufacturer. More often, it is submitted in a computer input medium in the format specified on the data sheet of the particular ROM. The manufacturer makes the corresponding mask for the paths to produce the 1'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 a ROM. For this reason, mask programming is economical only if large quantities of the same ROM configuration are to be manufactured.

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 0's (or all 1's) in every bit of the stored words. The fuses in the PROM are blown by application of current pulses through the output terminals. A blown fuse defines one binary state and an unbroken link represents the other state. This allows the user to program the unit in the laboratory to achieve the desired relationship between input addresses and stored words. Special units 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 unit available is called *erasable PROM,* or EPROM. EPROMs can be restructured to the initial value (all 0's or all 1's) even though they have been changed previously. When an EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as contacts. After erasure, the ROM returns to its initial state and can be reprogrammed. Certain ROMs can be erased with electrical signals instead of ultraviolet light, and these are called *electrically erasable PROMs,* or EEPROMs.

The function of a ROM can be interpreted in two different ways. The first interpretation is of a unit that implements any combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed in sum of minterms. The second interpretation considers the ROM to be a storage unit having a fixed pattern of bit strings called *words.* From this point of view, the inputs specify an *address* to a specific stored word, which is then applied to the outputs. For example, the ROM of Fig. 5-24 has three address lines, which specify eight stored words as given by the truth table. Each word is four bits long. This is the reason why the unit is given the name *read-only memory. Memory* is commonly used to designate a storage unit. *Read* is commonly used to signify that the contents of a word specified by an address in a storage unit is placed at the output terminals. Thus, a ROM is a memory unit with a fixed word pattern that can be read out upon application of a given address. The bit pattern in the ROM is permanent and cannot be changed during normal operation.

ROMs are widely used to implement complex combinational circuits directly from their truth tables. They are useful for converting from one binary code to another (such as ASCII to EBCDIC and vice versa), for arithmetic functions such as multipliers, for display of characters in a cathode-ray tube, and in many other applications requiring a large number of inputs and outputs. They are also employed in the design of control units of digital systems. As such, they are used to store fixed bit patterns that represent the sequence of control variables needed to enable the various operations in the system. A control unit that utilizes a ROM to store binary control information is called a *microprogrammed control unit.*

5-8 PROGRAMMABLE LOGIC ARRAY (PLA)

A combinational circuit may occasionally have don't-care conditions. When implemented with a ROM, a don't-care condition becomes an address input that will never occur. The words at the don't-care addresses need not be programmed and may be left in their original state (all 0's or all 1's). The result is that not all the bit patterns available in the ROM are used, which may be considered a waste of available equipment.

Consider, for example, a combinational circuit that converts a 12-bit card code to a 6-bit internal alphanumeric code (see end of Section 1-7). The input card code consists of 12 lines designated by 0, 1, 2, . . . , 9, 11, 12. The size of the ROM for implementing the code converter must be 4096 × 6, since there are 12 inputs and 6 outputs. There are only 47 valid entries for the card code; all other input combinations are don't-care conditions. Thus, only 47 words of the 4096 available are used. The remaining 4049 words of ROM are not used and are thus wasted.

For cases where the number of don't-care conditions is excessive, it is more economical to use a second type of LSI component called a *programmable logic array*, or PLA. A PLA is similar to a ROM in concept; however, the PLA does not provide full decoding of the variables and does not generate all the minterms as in the ROM. In the PLA, the decoder is replaced by a group of AND gates, each of which can be programmed to generate a product term of the input variables. The AND and OR gates inside the PLA are initially fabricated with fuses among them. The specific Boolean functions are implemented in sum of products form by blowing appropriate fuses and leaving the desired connections.

A block diagram of the PLA is shown in Fig. 5-25. It consists of n inputs, m outputs, k product terms, and m sum terms. The product terms constitute a group of k AND gates and the sum terms constitute a group of m OR gates. Fuses are inserted between all n inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gates and the inputs of the OR gates. Another set of fuses in the output inverters allows the output function to be generated either in the AND-OR form or in the AND-OR-INVERT form. With the inverter fuse in place, the inverter is bypassed, giving an AND-OR implementation. With the fuse blown, the inverter becomes part of the circuit and the function is implemented in the AND-OR-INVERT form.

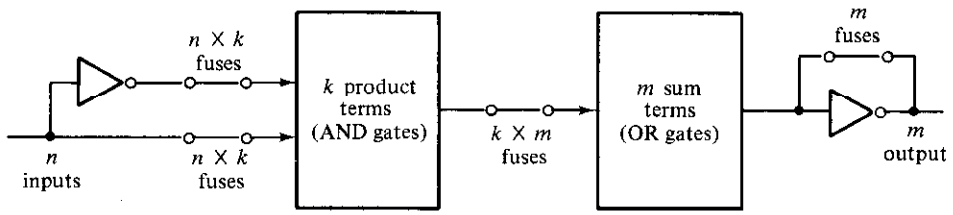


FIGURE 5-25
PLA block diagram

The size of the PLA is specified by the number of inputs, the number of product terms, and the number of outputs (the number of sum terms is equal to the number of outputs). A typical PLA has 16 inputs, 48 product terms, and 8 outputs. The number of programmed fuses is $2n \times k + k \times m + m$, whereas that of a ROM is $2^n \times m$.

Figure 5-26 shows the internal construction of a specific PLA. It has three inputs, three product terms, and two outputs. Such a PLA is too small to be available commercially; it is presented here merely for demonstration purposes. Each input and its complement are connected through fuses to the inputs of all AND gates. The outputs of the AND gates are connected through fuses to each input of the OR gates. Two more fuses are provided with the output inverters. By blowing selected fuses and leaving others intact, it is possible to implement Boolean functions in their sum of products form.

As with a ROM, the PLA may be mask-programmable or field-programmable. With a mask-programmable PLA, the customer must submit a PLA program table to the manufacturer. This table is used by the vendor to produce a custom-made PLA that has the required internal paths between inputs and outputs. A second type of PLA available is called a *field-programmable logic array*, or FPLA. The FPLA can be programmed by the user by means of certain recommended procedures. Commercial hardware programmer units are available for use in conjunction with certain FPLAs.

PLA Program Table

The use of a PLA must be considered for combinational circuits that have a large number of inputs and outputs. It is superior to a ROM for circuits that have a large number of don't-care conditions. The example to be presented demonstrates how a PLA is programmed. Bear in mind when going through the example that such a simple circuit will not require a PLA because it can be implemented more economically with SSI gates.

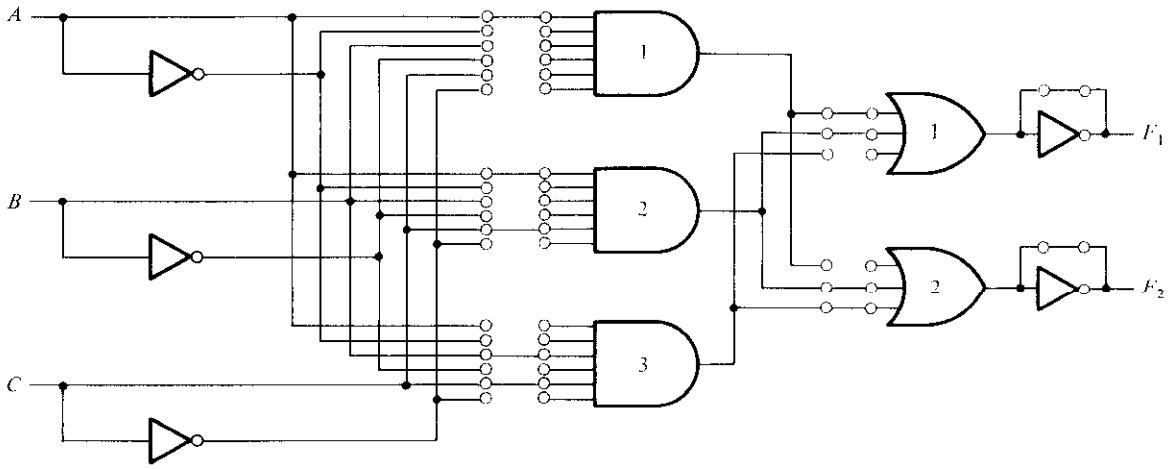
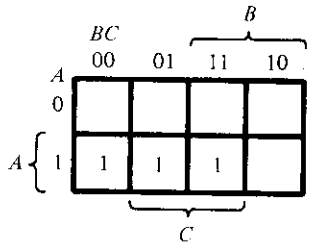


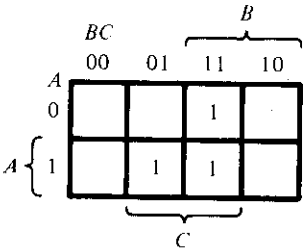
FIGURE 5-26
PLA with three inputs, three product terms, and two outputs; it implements the combinational circuit specified in Fig. 5-27

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

(a) Truth table



$F_1 = AB' + AC$



$F_2 = AC + BC$

(b) Map simplification

	Product term	Inputs A	B	C	Outputs F_1	F_2	
AB'	1	1	0	–	1	–	
AC	2	1	–	1	1	1	
BC	3	–	1	1	–	1	
					T	T	T/C

(c) PLA program table

FIGURE 5-27
Steps required in PLA implementation

Consider the truth table of the combinational circuit, shown in Fig. 5-27(a). Although a ROM implements a combinational circuit in its sum of minterms form, a PLA implements the functions in their sum of products form. Each product term in the expression requires an AND gate. Since the number of AND gates in a PLA is finite, it is necessary to simplify the function to a minimum number of product terms in order to minimize the number of AND gates used. The simplified functions in sum of products are obtained from the maps of Fig. 5-27(b):

$$F_1 = AB' + AC$$

$$F_2 = AC + BC$$

There are three distinct product terms in this combinational circuit: AB', AC, and BC. The circuit has three inputs and two outputs; so the PLA of Fig. 5-26 can be used to implement this combinational circuit.

Programming the PLA means that we specify the paths in its AND-OR-NOT pattern. A typical PLA program table is shown in Fig. 5-27(c). It consists of three columns. The first column lists the product terms numerically. The second column specifies the required paths between inputs and AND gates. The third column specifies the paths between the AND gates and the OR gates. Under each output variable, we write a T (for true) if the output inverter is to be bypassed, and C (for complement) if the function is to be complemented with the output inverter. The Boolean terms listed at the left are not part of the table; they are included for reference only.

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1. If it appears complemented (primed), the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash. Each product term is associated with an AND gate. The paths between the inputs and the AND gates are specified under the column heading *inputs*. A 1 in the input column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection. The appropriate fuses are blown and the ones left intact form the desired paths, as shown in Fig. 5-26. It is assumed that the open terminals in the AND gate behave like a 1 input.

The paths between the AND and OR gates are specified under the column heading *outputs*. The output variables are marked with 1's for all those product terms that formulate the function. In the example of Fig. 5-27, we have

$$F_1 = AB' + AC$$

so F_1 is marked with 1's for product terms 1 and 2 and with a dash for product term 3. Each product term that has a 1 in the output column requires a path from the corresponding AND gate to the output OR gate. Those marked with a dash specify no connection. Finally, a T (true) output dictates that the fuse across the output inverter remains intact, and a C (complement) specifies that the corresponding fuse be blown. The internal paths of the PLA for this circuit are shown in Fig. 5-26. It is assumed that an open terminal in an OR gate behaves like a 0, and that a short circuit across the output inverter does not damage the circuit.

When designing a digital system with a PLA, there is no need to show the internal connections of the unit, as was done in Fig. 5-26. All that is needed is a PLA program table from which the PLA can be programmed to supply the appropriate paths.

When implementing a combinational circuit with PLA, careful investigation must be undertaken in order to reduce the total number of distinct product terms, since a given PLA would have a finite number of AND terms. This can be done by simplifying each function to a minimum number of terms. The number of literals in a term is not important since we have available all input variables. Both the true value and the complement of the function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

Example 5-4

A combinational circuit is defined by the functions

$$F_1(A, B, C) = \Sigma(3, 5, 6, 7)$$

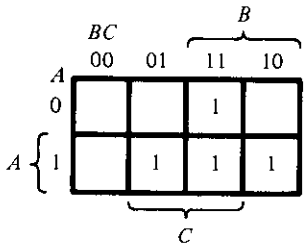
$$F_2(A, B, C) = \Sigma(0, 2, 4, 7)$$

Implement the circuit with a PLA having three inputs, four product terms, and two outputs.

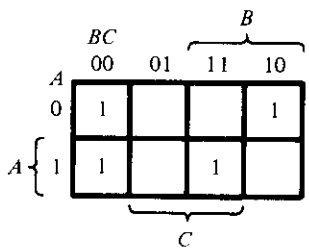
The two functions are simplified in the maps of Fig. 5-28. Both the true values and the complements of the functions are simplified. The combinations that give a minimum number of product terms are

$$F_1 = (B'C' + A'C' + A'B')'$$

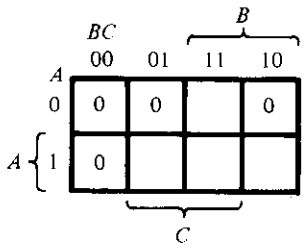
$$F_2 = B'C' + A'C' + ABC$$



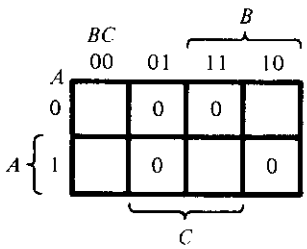
$F_1 = AC + AB + BC$



$F_2 = B'C' + A'C' + ABC$



$F_1' = B'C' + A'C' + A'B'$



$F_2' = B'C + A'C + ABC$

PLA program table

	Product term	Inputs A	B	C	Outputs F_1	F_2
$B'C'$	1	–	0	0	1	1
$A'C'$	2	0	–	0	1	1
$A'B'$	3	0	0	–	1	–
ABC	4	1	1	1	–	1
					C	T	T/C

FIGURE 5-28
Solution to Example 5-4

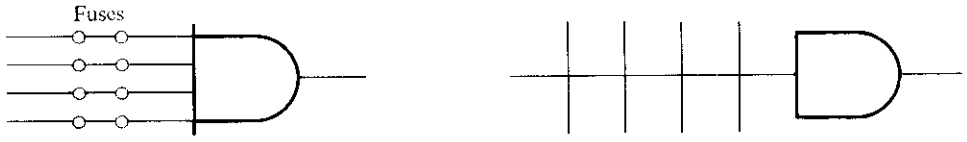
This gives only four distinct product terms: $B'C'$, $A'C'$, $A'B'$, and ABC. The PLA program table for this combination is shown in Fig. 5-28. Note that output F_1 is the normal (or true) output even though a C is marked under it. This is because F_1' is generated *prior* to the output inverter. The inverter complements the function to produce F_1 in the output. ■

The combinational circuit for this example is too small for practical implementation with a PLA. It was presented here merely for demonstration purposes. A typical commercial PLA would have over 10 inputs and about 50 product terms. The simplification of Boolean functions with so many variables should be carried out by means of a tabulation method or other computer-assisted simplification method. This is where a computer program may aid in the design of complex digital systems. The computer program should simplify each function of the combinational circuit and its complement to a minimum number of terms. The program then selects a minimum number of distinct terms that cover all functions in their true or complement form.

5-9 PROGRAMMABLE ARRAY LOGIC (PAL)

Programmable logic devices have hundreds of gates interconnected through hundreds of electronic fuses. It is sometimes convenient to draw the internal logic of such devices in a compact form referred to as *array logic*. Figure 5-29 shows the conventional and array logic symbols for a multiple-input AND gate. The conventional symbol is drawn with multiple lines showing the fuses connected to the inputs of the gate. The corresponding array logic symbol uses a single horizontal line connected to the gate input and multiple vertical lines to indicate the individual inputs. Each intersection between a vertical line and the common horizontal line has a fused connection. Thus, in Fig. 5-29(b), the AND gate has four inputs connected through fuses. In a similar fashion, we can draw the array logic for the OR gate or any other type of multiple-input gate.

The programmable array logic (PAL) is a programmable logic device with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program, but is not as flexible as the PLA. Figure 5-30 shows the array logic configuration of a typical PAL. It has four inputs and four out-



(a) Conventional symbol (b) Array logic symbol

FIGURE 5-29
Two graphic symbols for an AND gate

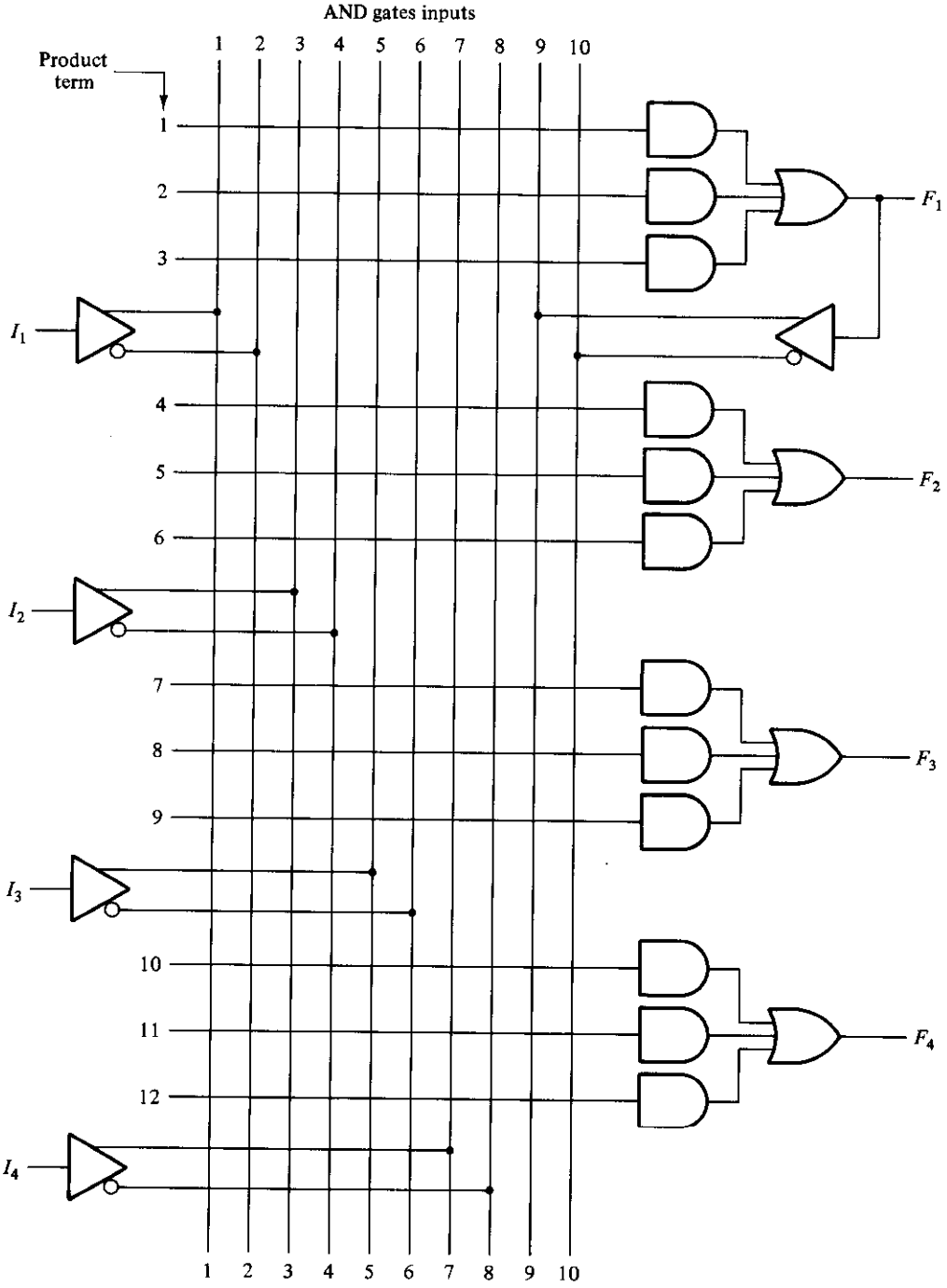


FIGURE 5-30
PAL with four inputs, four outputs, and three-wide AND-OR structure