

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:	<u>+100111</u>	subtrahend:	<u>-100111</u>	multiplier:	<u>× 101</u>
sum:	1010100	difference:	000110		1011
					0000
					<u>1011</u>
				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:

	<u>Integer quotient</u>		<u>Remainder</u>	<u>Coefficient</u>
$\frac{41}{2} =$	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: $(41)_{10} = (a_5 a_4 a_3 a_2 a_1 a_0)_2 = (101001)_2$

The arithmetic process can be manipulated more conveniently as follows:

<u>Integer</u>	<u>Remainder</u>
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:

	<u>Integer</u>		<u>Fraction</u>	<u>Coefficient</u>
$0.6875 \times 2 =$	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.

$$\begin{aligned}
 0.513 \times 8 &= 4.104 \\
 0.104 \times 8 &= 0.832 \\
 0.832 \times 8 &= 6.656 \\
 0.656 \times 8 &= 5.248 \\
 0.248 \times 8 &= 1.984 \\
 0.984 \times 8 &= 7.872
 \end{aligned}$$

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

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

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:

$$\left(\begin{array}{c} \boxed{10} \quad \boxed{110} \quad \boxed{001} \quad \boxed{101} \quad \boxed{011} \\ 2 \quad 6 \quad 1 \quad 5 \quad 3 \end{array} . \begin{array}{c} \boxed{111} \quad \boxed{100} \quad \boxed{000} \quad \boxed{110} \\ 7 \quad 4 \quad 0 \quad 6 \end{array} \right)_2 = (26153.7460)_8$$

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

$$\left(\begin{array}{c} \boxed{10} \quad \boxed{1100} \quad \boxed{0110} \quad \boxed{1011} \\ 2 \quad C \quad 6 \quad B \end{array} . \begin{array}{c} \boxed{1111} \quad \boxed{0010} \\ F \quad 2 \end{array} \right)_2 = (2C6B.F2)_{16}$$

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:

$$(673.124)_8 = \left(\begin{array}{c} \boxed{110} \quad \boxed{111} \quad \boxed{011} \\ 6 \quad 7 \quad 3 \end{array} . \begin{array}{c} \boxed{001} \quad \boxed{010} \quad \boxed{100} \\ 1 \quad 2 \quad 4 \end{array} \right)_2$$

$$(306.D)_{16} = \left(\begin{array}{c} \boxed{0011} \quad \boxed{0000} \quad \boxed{0110} \\ 3 \quad 0 \quad 6 \end{array} . \begin{array}{c} \boxed{1101} \\ D \end{array} \right)_2$$

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 $72532 - 3250$.

$$\begin{array}{rcl}
 M & = & 72532 \\
 \text{10's complement of } N & = & + \underline{96750} \\
 \text{Sum} & = & 169282 \\
 \text{Discard end carry } 10^5 & = & - \underline{100000} \\
 \text{Answer} & = & 69282
 \end{array}$$

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 $3250 - 72532$.

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

There is no end carry.

$$\text{Answer: } -(10\text{'s complement of } 30718) = -69282 \quad \blacksquare$$

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.

$$\begin{array}{r} \text{(a)} \quad X = \quad 1010100 \\ 2\text{'s complement of } Y = \quad + \underline{0111101} \\ \text{Sum} = \quad 10010001 \\ \text{Discard end carry } 2^7 = \quad - \underline{10000000} \\ \text{Answer: } X - Y = \quad 0010001 \end{array}$$

$$\begin{array}{r} \text{(b)} \quad Y = \quad 1000011 \\ 2\text{'s complement of } X = \quad + \underline{0101100} \\ \text{Sum} = \quad 1101111 \end{array}$$

There is no end carry.

$$\text{Answer: } Y - X = -(2\text{'s complement of } 1101111) = -0010001 \quad \blacksquare$$

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 Repeat Example 1-7 using 1's complement.

1-8

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

$$\begin{array}{r}
 X = \quad 1010100 \\
 \text{1's complement of } Y = \quad + \underline{0111100} \\
 \text{Sum} = \quad 10010000 \\
 \text{End-around carry} \quad \rightarrow + 1 \\
 \text{Answer: } X - Y = \quad 0010001
 \end{array}$$

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

$$\begin{array}{r}
 Y = \quad 1000011 \\
 \text{1's complement of } X = \quad + \underline{0101011} \\
 \text{Sum} = \quad 1101110
 \end{array}$$

There is no end carry.

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

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-

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 operator specifies a rule for obtaining a unique element of S . For example, the set of natural numbers $N = \{1, 2, 3, 4, \dots\}$ 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

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

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

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

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

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

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

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

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 constitute 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 \cdot 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) $0 + 0 = 0$ $0 + 1 = 1 + 0 = 1$
 - (b) $1 \cdot 1 = 1$ $1 \cdot 0 = 0 \cdot 1 = 0$
 which establishes the two *identity elements* 0 for $+$ and 1 for \cdot 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 \cdot 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 \cdot 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): $x + x = x$.

$$\begin{aligned}
x + x &= (x + x) \cdot 1 && \text{by postulate: } 2(b) \\
&= (x + x)(x + x') && 5(a) \\
&= x + xx' && 4(b) \\
&= x + 0 && 5(b) \\
&= x && 2(a)
\end{aligned}$$

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

$$\begin{aligned}
x \cdot x &= xx + 0 && \text{by postulate: } 2(a) \\
&= xx + xx' && 5(b) \\
&= x(x + x') && 4(a) \\
&= x \cdot 1 && 5(a) \\
&= x && 2(b)
\end{aligned}$$

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): $x + 1 = 1$.

$$\begin{aligned}
x + 1 &= 1 \cdot (x + 1) && \text{by postulate: } 2(b) \\
&= (x + x')(x + 1) && 5(a) \\
&= x + x' \cdot 1 && 4(b) \\
&= x + x' && 2(b) \\
&= 1 && 5(a)
\end{aligned}$$

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 .

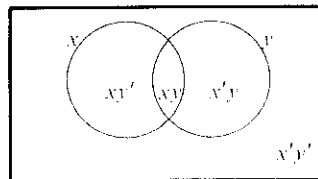


FIGURE 2-1
Venn diagram for two variables

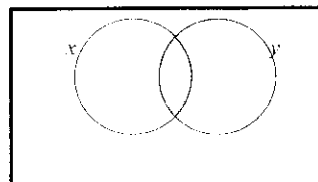
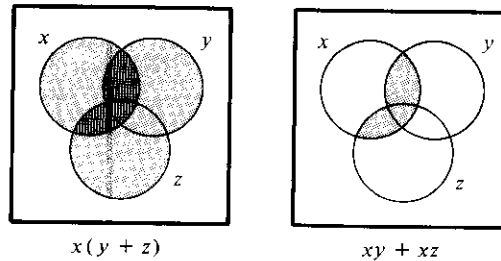


FIGURE 2-2
Venn diagram illustration $x = xy + x$

**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:

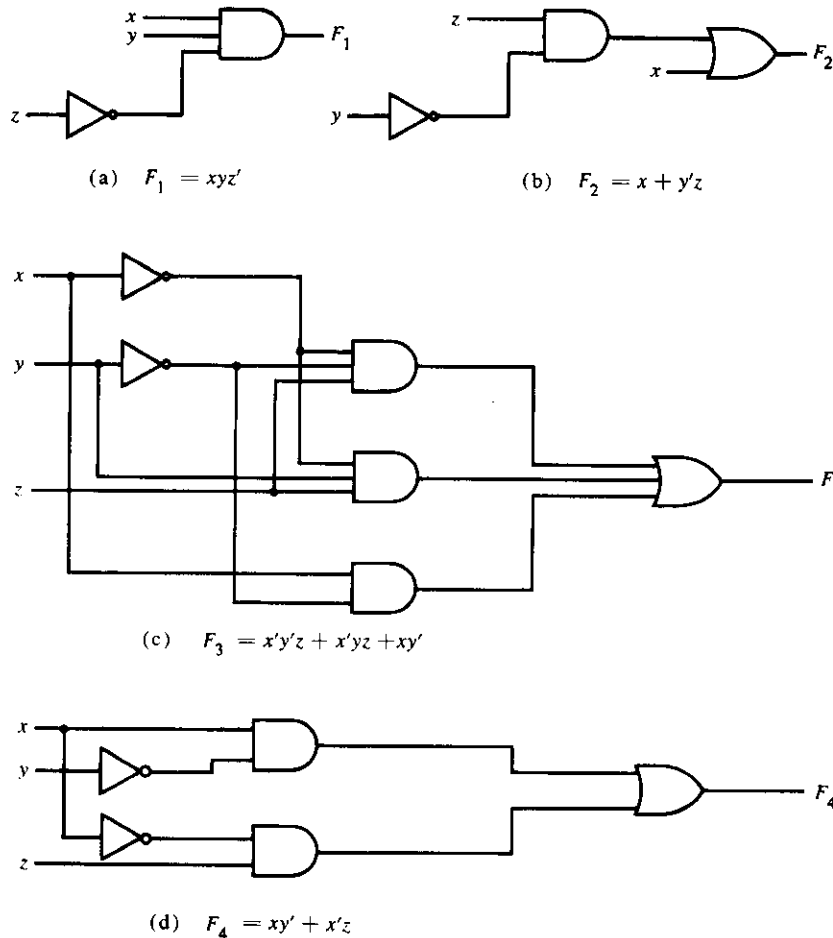
$$F_4 = xy' + x'z$$

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:

$$\begin{aligned}
 (A + B + C + D + \cdots + F)' &= A'B'C'D' \cdots F' \\
 (ABCD \cdots F)' &= A' + B' + C' + D' + \cdots + F'
 \end{aligned}$$

The generalized form of DeMorgan'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 $F_1 = x'y'z' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorem as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$

$$\begin{aligned} F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')' \cdot (yz)' \\ &= x' + (y + z)(y' + z') \end{aligned}$$

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 variables 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

<i>x</i>	<i>y</i>	<i>z</i>	Minterms		Maxterms	
			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

<i>x</i>	<i>y</i>	<i>z</i>	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:

$$\begin{aligned} f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z) \\ &= M_0 M_1 M_2 M_4 \end{aligned}$$

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 clarify 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:

$$\begin{aligned}
 A &= AB(C + C') + AB'(C + C') \\
 &= ABC + ABC' + AB'C + AB'C'
 \end{aligned}$$

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

$$\begin{aligned}
 F &= A + B'C \\
 &= ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C
 \end{aligned}$$

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

$$\begin{aligned}
 F &= A'B'C + AB'C' + AB'C + ABC' + ABC \\
 &= m_1 + m_4 + m_5 + m_6 + m_7
 \end{aligned}$$

■

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^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:

$$\begin{aligned} F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\ &= M_0 M_2 M_4 M_5 \end{aligned}$$

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 similarity 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			\cdot	$/$		$/$		\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
$F_{11} = x + y'$	$x \supset y$	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 \cdot 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 practical 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		$F = xy$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= x \odot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	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

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

and

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

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:

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

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

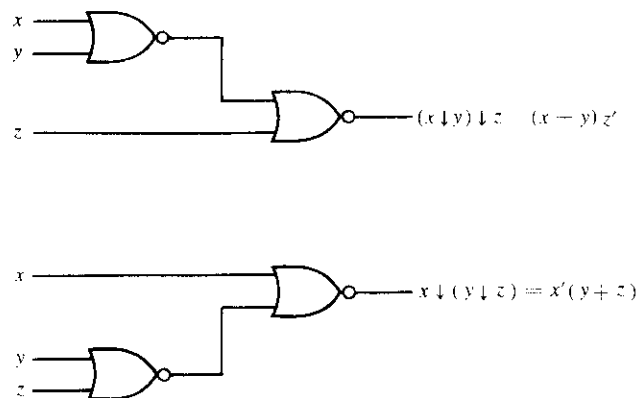


FIGURE 2-6

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. Moreover, 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.

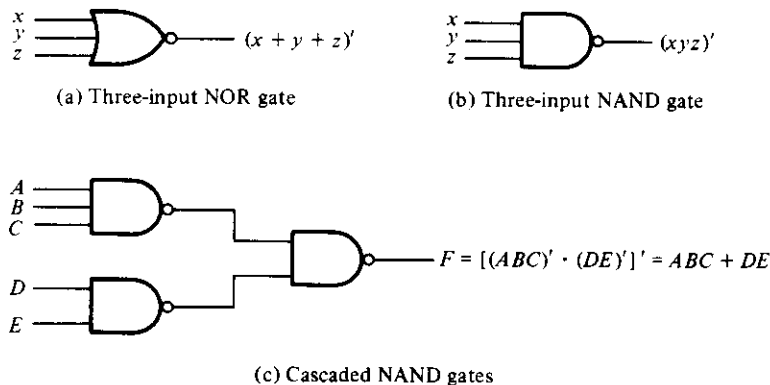


FIGURE 2-7

Multiple-input and cascaded NOR and NAND gates

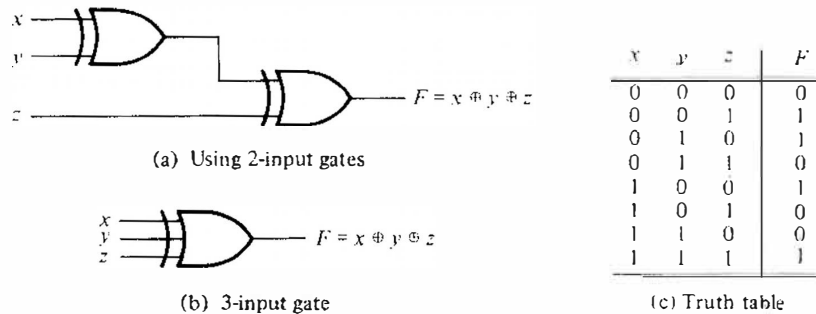


FIGURE 2-8
3-input exclusive-OR gate

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 $20 \times 8 \times 3$ millimeters. An entire microprocessor is enclosed within a 64-pin IC package with dimensions of $50 \times 15 \times 4$ 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.

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.

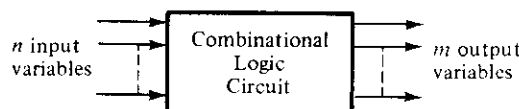


FIGURE 4-1

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

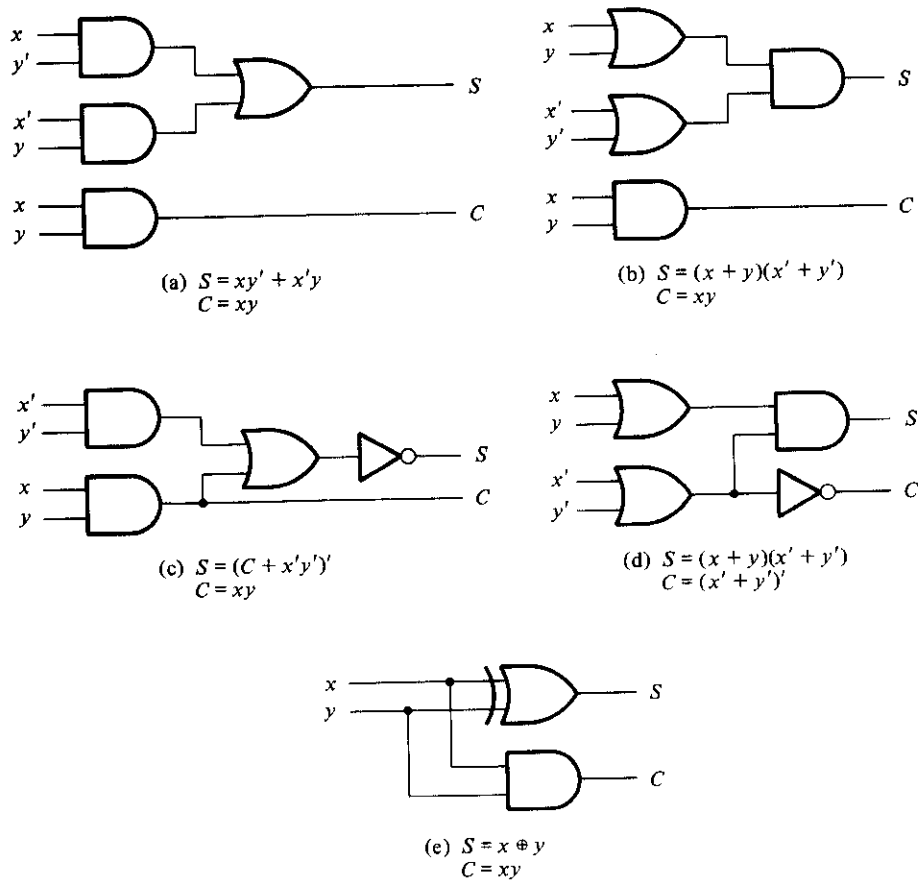
The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum.

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

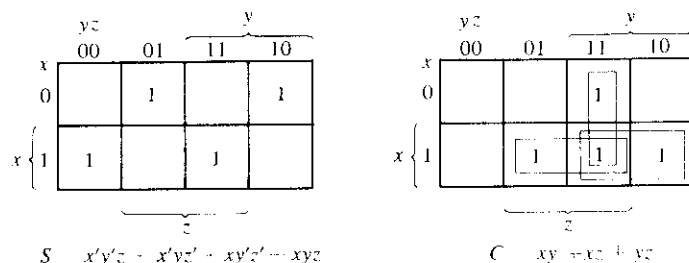
A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y , represent the two significant bits to be added. The third input, z , represents the carry from the previous lower significant 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**

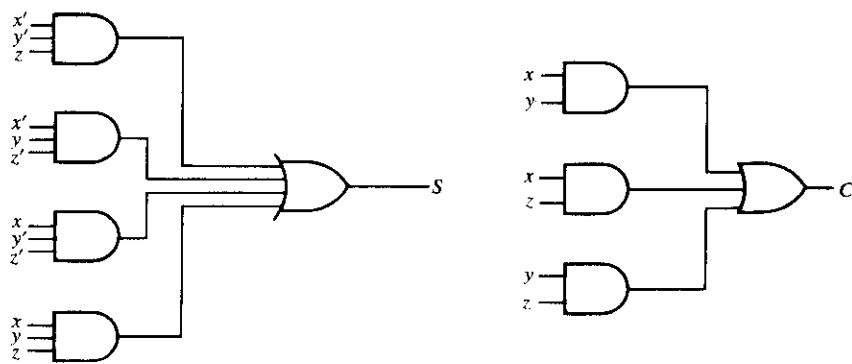
Maps for a full-adder

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:

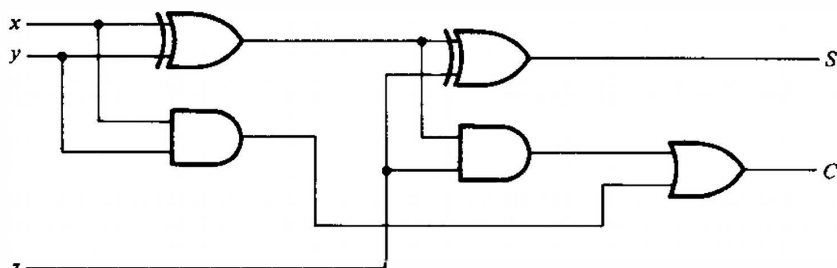
$$S = x'y'z + x'yz' + xy'z' + xyz$$

$$C = xy + xz + yz$$

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-