2

NUMBER SYSTEMS

OBJECTIVES

1 An explanation of positional notation is given, and the idea of the base, or radix, of a number system is presented.

2 The binary number system is explained as well as how to add, subtract, multiply, and divide in this system. Then techniques for converting from binary to decimal and decimal to binary are given.

3 Negative numbers are represented in computers by using a sign bit, and this concept is explained. Negative numbers are often represented by using a complemented form rather than a signed-magnitude form. The two major complemented forms, true complement and radix minus one, are described.

4 The representation of decimal numbers using bistable devices can be accomplished with a binary-coded-decimal (BCD) system, and several of these are explained.

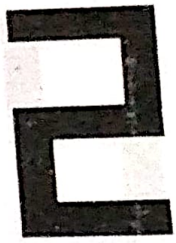
5 The octal and hexadecimal number systems are widely used in computer literature and manufacturer's manuals. These number systems are explained along with conversion techniques to and from decimal and binary.

DECIMAL SYSTEM

2.1 Our present system of numbers has 10 separate symbols, 0, 1, 2, 3, . . . , 9, which are called Arabic numerals. We would be forced to stop at 9 or to invent more symbols if it were not for the use of *positional notation*. An example of earlier types of notation can be found in Roman numerals, which are essentially additive: III = I + I + I, XXV = X + X + V. New symbols (X, C, M, etc.) were used as the numbers increased in value: thus V rather than IIII is equal to 5. The only importance of position in Roman numerals lies in whether a symbol precedes or follows another symbol (IV = 4, while VI = 6). The clumsiness of this system can be seen easily if we try to multiply XII by XIV. Calculating with roman numerals was so difficult that early mathematicians were forced to perform arithmetic operations almost entirely on abaci, or counting boards, translating their results back to Roman numeral form. Pencil-and-paper computations are unbelievably intricate and difficult in such systems. In fact, the ability to perform such operations as addition and multiplication was considered a great accomplishment in earlier civilizations.

Now the great beauty and simplicity of our number system can be seen. It is necessary to learn only the 10 basic numerals and the *positional notational system* in order to count to any desired figure. After memorizing the addition and multiplication tables and learning a few simple rules, we can perform all arithmetic operations. Notice the simplicity of multiplying 12 × 14 by using the present system:

$$\begin{array}{r} 14 \\ \times\ 12 \\ \hline 28 \\ 14 \\ \hline 168 \end{array}$$



The actual meaning of the number 168 can be seen more clearly if we notice that it is spoken as "one hundred and sixty-eight." Basically, the number is a contraction of $1 \times 100 + 6 \times 10 + 8$. The important point is that the value of each digit is determined by its position. For example, the 2 in 2000 has a different value than the 2 in 20. We show this verbally by saying "two thousand" and "twenty." Different verbal representations have been invented for numbers from 10 to 20 (eleven, twelve, . . .), but from 20 upward we break only at powers of 10 (hundreds, thousands, millions, billions). Written numbers are always contracted, however, and only the basic 10 numerals are used, regardless of the size of the integer written. The general rule for representing numbers in the decimal system by using positional notation is as follows: $a_{n-1}\ 10^{n-1} + a_{n-2}\ 10^{n-2} + \cdots + a_0$ is expressed as $a_{n-1}\ a_{n-2} \cdots a_0$, where n is the number of digits to the left of the decimal point.

The *base*, or *radix*, of a number system is defined as the number of different digits which can occur in each position in the number system. The decimal number system has a base, or radix, of 10. Thus the system has 10 different digits (0, 1, 2, . . . , 9), any one of which may be used in each position in a number. History records the use of several other number systems. The quinary system, which has 5 for its base, was prevalent among Eskimos and North American Indians. Examples of the duodecimal system (base 12) may be seen in clocks, inches and feet, and dozens or grosses.

BISTABLE DEVICES

2.2 The basic elements in early computers were relays and switches. The operation of a switch, or relay, can be seen to be essentially bistable, or binary in nature; that is, the switch is either on (1) or off (0). The principal circuit elements in more modern computers are transistors. The desire for reliability led designers to use these devices so that they were always in one of two states, fully conducting or nonconducting. A simple analogy may be made between this type of circuit and an electric light. At any given time the light (or transistor) is either on (conducting) or off (not conducting). Even after a bulb is old and weak, it is generally easy to tell whether it is on or off.

Because of the large number of electronic parts used in computers, it is highly desirable to utilize them in such a manner that slight changes in their characteristics will not affect their performance. The best way of accomplishing this is to use circuits which are basically *bistable* (having two possible states).

COUNTING IN THE BINARY SYSTEM

2.3 The same type of positional notation is used in the binary number system as in the decimal system. Table 2.1 lists the first 20 binary numbers.

Although the same positional notation system is used, the decimal system uses powers of 10, and the binary system powers of 2. As was previously explained, the number 125 actually means $1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$. In the binary system, the same number (125) is represented as 1111101, meaining $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

2

NUMBER SYSTEMS

TABLE 2.1

DECIMAL		BINARY	DECIMAL		BINARY
1	=	1	11	=	1011
2	=	10	12	=	1100
3	=	11	13	=	1101
4	=	100	14	=	1110
5	=	101	15	=	1111
6	=	110	16	=	10000
7	=	111	17	=	10001
8	=	1000	18	=	10010
9	=	1001	19	=	10011
10	=	1010	20	=	10100

To express the value of a binary number, therefore, $a_{n-1}\,2^{n-1} + a_{n-2}\,2^{n-2} + \cdots + a_0$ is represented as $a_{n-1}\,a_{n-1}\cdots a_0$, where a_i is either 1 or 0 and n is the number of digits to the left of the binary (radix) point.

The following examples illustrate the conversion of binary numbers to the decimal system:

$$\begin{aligned}
101 &= 1 \times 2^{3-1} + 0 \times 2^{3-2} + 1 \times 2^{3-3} \\
&= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 4 + 1 = 5 \\
1001 &= 1 \times 2^{4-1} + 0 \times 2^{4-2} + 0 \times 2^{4-3} + 1 \times 2^{4-4} \\
&= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 8 + 1 = 9 \\
11.011 &= 1 \times 2^{2-1} + 1 \times 2^{2-2} + 0 \times 2^{2-3} + 1 \times 2^{2-4} + 1 \times 2^{2-5} \\
&= 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\
&= 2 + 1 + \tfrac{1}{4} + \tfrac{1}{8} \\
&= 3\tfrac{3}{8}
\end{aligned}$$

Note that fractional numbers are formed in the same general way as in the decimal system. Just as

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

in the decimal system,

$$0.101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

in the binary system.

BINARY ADDITION AND SUBTRACTION

2.4 Binary addition is performed in the same manner as decimal addition. Actually, binary arithmetic is much simpler to learn. The complete table for binary addition is as follows:

BINARY MULTIPLICATION AND DIVISION

$$
\begin{aligned}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 \\
1 + 1 &= 0 \quad \text{plus a carry-over of 1}
\end{aligned}
$$

"Carry-overs" are performed in the same manner as in decimal arithmetic. Since 1 is the largest digit in the binary system, any sum greater than 1 requires that a digit be carried over. For instance, 100 plus 100 binary requires the addition of the two 1s in the third position to the left, with a carry-over. Since $1 + 1 = 0$ plus a carry-over of 1, the sum of 100 and 100 is 1000. Here are three more examples of binary addition:

DECIMAL	BINARY	DECIMAL	BINARY	DECIMAL	BINARY
5	101	15	1111	$3\frac{1}{4}$	11.01
6	110	20	10100	$5\frac{3}{4}$	101.11
11	1011	35	100011	9	1001.00

Subtraction is the inverse operation of addition. To subtract, it is necessary to establish a procedure for subtracting a larger from a smaller digit. The only case in which this occurs with binary numbers is when 1 is subtracted from 0. The remainder is 1, but it is necessary to borrow 1 from the next column to the left. This is the binary subtraction table.

$$
\begin{aligned}
0 - 0 &= 0 \\
1 - 0 &= 1 \\
1 - 1 &= 0 \\
0 - 1 &= 1 \quad \text{with a borrow of 1}
\end{aligned}
$$

A few examples will make the procedure for binary subtraction clear:

DECIMAL	BINARY	DECIMAL	BINARY	DECIMAL	BINARY
9	1001	16	10000	$6\frac{1}{4}$	110.01
−5	−101	−3	−11	$-4\frac{1}{2}$	−100.1
4	0100	13	1101	$1\frac{3}{4}$	1.11

BINARY MULTIPLICATION AND DIVISION

2.5 The table for binary multiplication is very short, with only four entries instead of the 100 necessary for decimal multiplication:

$$
\begin{aligned}
0 \times 0 &= 0 \\
1 \times 0 &= 0 \\
0 \times 1 &= 0 \\
1 \times 1 &= 1
\end{aligned}
$$

The following three examples of binary multiplication illustrate the simplicity of each operation. It is only necessary to copy the multiplicand if the digit in the multiplier is 1 and to copy all 0s if the digit in the multiplier is a 0. The ease with which each step of the operation is performed is apparent.

DECIMAL	BINARY	DECIMAL	BINARY	DECIMAL	BINARY
12	1100	102	1100110	1.25	1.01
×10	×1010	×8	×1000	×2.5	×10.1
120	0000	816	1100110000	625	101
	1100			250	1010
	0000			3.125	11.001
	1100				
	1111000				

Binary division is, again, very simple. As in the decimal system (or in any other), division by zero is meaningless. The complete table is

$$0 \div 1 = 0$$
$$1 \div 1 = 1$$

Here are two examples of division:

DECIMAL	BINARY
5	101
5)25	101)11001
	101
	101
	101

DECIMAL	BINARY
2.416···	10.011010101···
12)29.0000	1100)11101.00
24	1100
50	10100
48	1100
20	10000
12	1100
80	10000
72	1100
8	···

To convert the quotient obtained in the second example from binary to decimal, we would proceed as follows:

$$\begin{aligned}
10.011010101 = {} & 1 \times 2^{1} = 2.0 \\
& 0 \times 2^{0} = 0.0 \\
& 0 \times 2^{-1} = 0.0 \\
& 1 \times 2^{-2} = 0.25 \\
& 1 \times 2^{-3} = 0.125 \\
& 0 \times 2^{-4} = 0.0 \\
& 1 \times 2^{-5} = 0.03125 \\
& 0 \times 2^{-6} = 0.0 \\
& 1 \times 2^{-7} = 0.0078125 \\
& 0 \times 2^{-8} = 0.0 \\
& 1 \times 2^{-9} = 0.001953125 \\
& \qquad\qquad 2.416015625
\end{aligned}$$

Therefore, 10.011010101 binary equals approximately 2.416 decimal.

CONVERTING DECIMAL NUMBERS TO BINARY

2.6 There are several methods for converting a decimal number to a binary number. The first and most obvious method is simply to subtract all powers of 2 which can be subtracted from the decimal number until nothing remains. The highest power of 2 is subtracted first, then the second highest, etc. To convert the decimal integer 25 to the binary number system, first the highest power of 2 which can be subtracted from 25 is found. This is $2^4 = 16$. Then $25 - 16 = 9$. The highest power of 2 which can be subtracted from 9 is 2^3, or 8. The remainder after subtraction is 1, or 2^0. The binary representation for 25 is, therefore, 11001.

This is a laborious method for converting numbers. It is convenient for small numbers when it can be performed mentally, but is less used for larger numbers. Instead, the decimal number is repeatedly divided by 2, and the remainder after each division is used to indicate the coefficients of the binary number to be formed. Notice that the binary number derived is written from the bottom up.

$$\begin{aligned}
125 \div 2 &= 62 + \text{remainder of } 1\\
62 \div 2 &= 31 + \text{remainder of } 0\\
31 \div 2 &= 15 + \text{remainder of } 1\\
15 \div 2 &= 7 + \text{remainder of } 1\\
7 \div 2 &= 3 + \text{remainder of } 1\\
3 \div 2 &= 1 + \text{remainder of } 1\\
1 \div 2 &= 0 + \text{remainder of } 1
\end{aligned}$$

The binary representation of 125 is, therefore, 1111101. Checking this result gives

$$\begin{aligned}
1 \times 2^6 &= 64\\
1 \times 2^5 &= 32\\
1 \times 2^4 &= 16\\
1 \times 2^3 &= 8\\
1 \times 2^2 &= 4\\
0 \times 2^1 &= 0\\
1 \times 2^0 &= \underline{1}\\
&\quad 125
\end{aligned}$$

This method will not work for mixed numbers. If similar methods are to be used, first it is necessary to divide the number into its whole and fractional parts; that is, 102.247 would be divided into 102 and 0.247. The binary representation for each part is found, and then the two parts are added.

The conversion of decimal fractions to binary fractions may be accomplished by using several techniques. Again, the most obvious method is to subtract the highest negative power of 2 which may be subtracted from the decimal fraction. Then the next highest negative power of 2 is subtracted from the remainder of the first subtraction, and this process is continued until there is no remainder or to the desired precision.

$$\begin{aligned}
0.875 - 1 \times 2^{-1} &= 0.875 - 0.5 = 0.375\\
0.375 - 1 \times 2^{-2} &= 0.375 - 0.25 = 0.125\\
0.125 - 1 \times 2^{-3} &= 0.125 - 0.125 = 0
\end{aligned}$$

Therefore, 0.875 decimal is represented by 0.111 binary. A much simpler method for longer fractions consists of repeatedly "doubling" the decimal fraction. If a 1 appears to the left of the decimal point after a multiplication by 2 is performed, a 1 is added to the right of the binary fraction being formed. If after a multiplication by 2, a 0 remains to the left of the decimal point of the decimal number, a 0 is added to the right of the binary number. The following example illustrates the use of this technique in converting 0.4375 decimal to the binary system:

	BINARY REPRESENTATION
$2 \times 0.4375 = 0.8750$	0.0
$2 \times 0.875 = 1.750$	0.01
$2 \times 0.75 = 1.50$	0.011
$2 \times 0.5 = 1.0$	0.0111

The binary representation of 0.4375 is, therefore, 0.0111.

NEGATIVE NUMBERS

2.7 A standard convention adopted for writing negative numbers consists of placing a *sign symbol* before a number that is negative. For instance, negative 39 is written as -39. If -39 is to be added to $+70$, we write

$$+70 + (-39) = 31$$

When a negative number is subtracted from a positive number, we write $+70 - (-39) = +70 + 39 = 109$. The rules for handling negative numbers are well known and are not repeated here, but since negative numbers constitute an important part of our number system, the techniques used to represent negative numbers in digital machines are described.

In binary machines, numbers are represented by a set of bistable storage devices, each of which represents one binary digit. As an example, given a set of five switches, any number from 00000 to 11111 may be represented by the switches if we define a switch with its contacts closed as representing a 1 and a switch with open contacts as representing a 0. If we desire to increase the total range of numbers that we can represent so that it will include the negative numbers from 00000 to -11111, another bit (or switch) will be required. We then treat this bit as a *sign bit* and place it before the magnitude of the number to be represented.

Generally, the convention is adopted that when the sign bit is a 0, the number represented is positive, and when the sign bit is a 1, the number is negative. If the previous situation, where five switches are used to store the magnitude of a number, is extended so that both positive and negative numbers may be stored, then a sixth switch will be required. When the contacts of this switch are open, the number will be a positive number equal to the magnitude of the number stored in the other five switches; and if the switch for the sign bit is closed, the number represented by the six switches will be a negative number with a magnitude determined by the other five switches. An example is shown in Fig. 2.1.

Sets of storage devices which represent a number or are handled as an entity

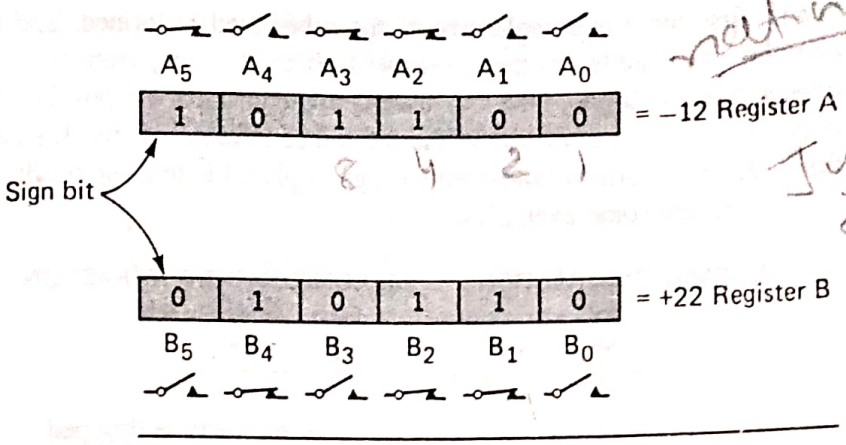


FIGURE 2.1

Example of negative-number representation.

are referred to as *registers*, and they are given names such as register A, register B, register C, etc. We can then write that register A contains −12 and register B contains +22. In writing a signed number in binary form, the sign bit is set apart from the magnitude of the number by means of an underscore, so that 0̲0111 represents +0111, or positive 7 decimal, and 1̲0111 represents −0111, or negative 7 decimal.

The use of the an underscore to mark the sign bit does not indicate that there is any difference between this bit and the other bits as the number is stored in a computer. Each and every binary bit is simply stored in a separate bistable device. A symbol other than the underscore could be used to separate the sign and magnitude bits, such as a hyphen, period, or a star. Then, −1011 (negative 11 decimal) could be written 1-1011 or 1*1011 or 1.1011 and +1100 as 0-1100 or 0*1100 or 0.1100. In fact, no marking whatever could be used, but it is felt that some indication makes for easier reading of signed numbers which use a sign bit.

USE OF COMPLEMENTS TO REPRESENT NEGATIVE NUMBERS

2.8 The convention of using a sign bit to indicate whether a stored number is negative or positive has been described. The magnitude of the number stored is not always represented in normal form, however, but quite often negative numbers are stored in *complemented form*. By using this technique, a machine can be made to both add and subtract, using only circuitry for adding. The actual technique involved is described in Chap. 5.

There are two basic types of complements which are useful in the binary and decimal number systems. In the decimal system the two types are referred to as the 10s *complement* and the 9s *complement*.

The 10s complement of any number may be formed by subtracting each digit of the number from 9 and then adding 1 to the least significant digit of the number thus formed. For instance, the 10s complement of 87 is 13, and the 10s complement of 23 is 77.

To subtract[1] one positive number (the minuend) from another (the subtra-

[1]The number of digits in each number must be the same. If one number has fewer digits than the other, then 0s can be added to the left until the number of digits is the same.

hend), first the 10s complement of the subtrahend is formed, and then this 10s complement is added to the minuend. If there is a carry from the addition of the most significant digits, then it is discarded, the difference is positive, and the result is correct. If there is no carry, the difference is negative, the 10s complement of this number is formed, and a minus sign is placed before the result.

Here are some examples.

NORMAL SUBTRACTION	10s COMPLEMENT SUBTRACTION
89 −23 66	89 89 −23 = +77 166 the carry is dropped
98 −87 11	98 98 −87 = +13 111 the carry is dropped
49 −62 −13	49 +38 87 no carry, so result is negative 10s complement, or −13
54 −81 −27	54 +19 73 no carry, so result is negative 10s complement, or −27

The 9s complement of a decimal number is formed by subtracting each digit of the number from 9. For instance, the 9s complement of 23 is 76, and the 9s complement of 87 is 12. When subtraction[2] is performed by using the 9s complement, the complement of the subtrahend is added as in 10s complement subtraction, but any carry generated must be added to the rightmost digit of the result. As is the case with 10s complement subtraction, if no carry is generated for the addition of the most significant digits, then the result is negative, the 9s complement of the result is formed, and a minus sign is placed before it.

NORMAL SUBTRACTION	9s COMPLEMENT SUBTRACTION
89 −23 66	89 89 −23 = +76 165 1 66
98 −87 11	98 98 −87 = +12 110 1 11

[2]The number of digits in the subtrahend and minuend must be the same. If one number has more digits than the other, zeros are added to the left of the shorter number until it has the same number of digits.

$$\begin{array}{r} 15 \\ -37 \\ \hline -22 \end{array} \qquad \begin{array}{r} 15 \\ -37 \\ \hline \end{array} = \begin{array}{r} 15 \\ +62 \\ \hline 77 \end{array}$$

no carry, so difference is negative 9s complement, which is -22

$$\begin{array}{r} 27 \\ -44 \\ \hline -17 \end{array} \qquad \begin{array}{r} 27 \\ -44 \\ \hline \end{array} = \begin{array}{r} 27 \\ +55 \\ \hline 82 \end{array}$$

no carry, so difference is negative 9s complement, which is -17

Complete rules for handling signs during the subtraction process and for handling all combinations of positive and negative numbers are explained in Chap. 5. If this seems, at first, to be an unwieldy technique, note that the majority of computers now being constructed subtract by using a complemented number.

COMPLEMENTS IN OTHER NUMBER SYSTEMS

2.9 There are two types of complements for each number system. Since now only binary and BCD machines are being constructed in quantity, only these number systems are explained in any detail. The two types of complements and the rules for obtaining them are as follows:

1 *True complement* This is formed by subtracting each digit of the number from the radix minus one of the number system and then adding 1 to the least significant digit of the number formed. The true complement of a number in the decimal system is referred to as the 10s complement and in the binary system as the 2s complement.

2 *Radix-minus-one complement* The radix minus one is 9 for the decimal system and 1 for the binary system. The complement in each system is formed by subtracting each digit of the number from the radix minus one. For instance, the radix-minus-one complement of decimal 72 is 27.

BINARY NUMBER COMPLEMENTS

2.10 According to the rule in the preceding section, the 2s complement of a binary number is formed by simply subtracting each digit (bit) of the number from the radix minus one and adding a 1 to the least significant bit. Since the radix in the binary number system is 2, each bit of the binary number is subtracted from 1. The application of this rule is actually very simple; every 1 in the number is changed to a 0 and every 0 to a 1. Then a 1 is added to the least significant bit of the number formed. For instance, the 2s complement of 10110 is 01010, and the 2s complement of 11010 is 00110. Subtraction using the 2s complement system involves forming the 2s complement of the subtrahend and then adding this "true

$$\begin{array}{r} 10110 \\ 01001 \\ +\quad\ 1 \\ \hline 01010 \end{array}$$

complement" to the minuend. For instance,

$$\begin{array}{r} 11011 \\ -10100 \\ \hline 00111 \end{array} = \begin{array}{r} 11011 \\ +01100 \\ \hline 1\ 00111 \\ \text{carry is dropped} \end{array} \quad \text{and} \quad \begin{array}{r} 11100 \\ -00100 \\ \hline 11000 \end{array} = \begin{array}{r} 11100 \\ +11100 \\ \hline 1\ 11000 \\ \text{dropped} \end{array}$$

Subtraction using the 1s complement system is also straightforward. The 1s complement of a binary number is formed by changing each 1 in the number to a 0 and each 0 in the number to a 1. For instance, the 1s complement of 10111 is 01000, and the 1s complement of 11000 is 00111.

When subtraction is performed in the 1s complement system, any end-around carry is added to the least significant bit. For instance,

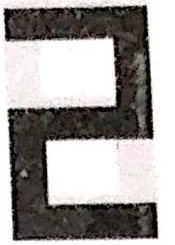
$$\begin{array}{r} 11001 \\ -10110 \\ \hline 00011 \end{array} = \begin{array}{r} 11001 \\ +01001 \\ \hline 1\ 00010 \\ \qquad\quad \rightarrow 1 \\ \hline 00011 \end{array} \quad \text{and} \quad \begin{array}{r} 11110 \\ -01101 \\ \hline 10001 \end{array} = \begin{array}{r} 11110 \\ +10010 \\ \hline 1\ 10000 \\ \qquad\quad \rightarrow 1 \\ \hline 10001 \end{array}$$

BINARY-CODED-DECIMAL NUMBER REPRESENTATION

2.11 Since most of the electronic circuit elements used to construct digital computers are inherently binary in operation, the binary number system is the most natural number system for a computer. Also, computers constructed with the binary number system require a smaller amount of circuitry and so are more efficient than machines operating in other number systems. However, the decimal system has been used for a long time, and there is a natural reaction to performing calculations in a binary number system. Also, since checks, bills, tax rates, prices, etc., are all figured in the decimal system, the values of most things must be converted from decimal to binary before computations can begin. For these and other reasons, most of the early machines operated in binary-coded-decimal number systems. In such systems a coded group of binary bits is used to represent each of the 10 decimal digits. For instance, an obvious and natural code is a simple *weighted binary code*, as shown in Table 2.2.

TABLE 2.2

BINARY CODE	DECIMAL DIGIT
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9



This is known as a *binary-coded-decimal 8, 4, 2, 1* code, or simply BCD. Notice that 4 binary bits are required for each decimal digit, and each bit is assigned a weight; for instance, the rightmost bit has a weight of 1, and the leftmost bit in each code group has a weight of 8. By adding the weights of the positions in which 1s appear, the decimal digit represented by a code group may be derived. This is somewhat uneconomical since $2^4 = 16$, and thus the 4 bits could actually represent 15 different values. But the next lesser choice, 3 bits, gives only 2^3, or 8, values, which are insufficient. If the decimal number 214 is to be represented in this type of code, 12 binary bits are required as follows: 0010 0001 0100. For the decimal number 1246 to be represented, 16 bits are required: 0001 0010 0100 0110.

This is a very useful code and has been much used. One difficulty with the code, however, lies in forming the complements of numbers in this system. It is common practice to perform subtraction in a computer by adding the complement of the subtrahend; however, when the BCD 8, 4, 2, 1 system is used, the most natural complement of the number stored is not useful because the most direct way for a computer to complement a number is simply to change each 0 to a 1 and each 1 to a 0. However, the natural complement of 0010 (2 decimal) is 1101, which is 13, and not an acceptable BCD character in this system. To get around this difficulty, several other codes have been used. One of the first, a code that was used in the early Mark machines built at Harvard and has been used a good deal since, is known as the *excess-3* code and is formed by adding 3 to the decimal number and then forming the binary-coded number in the normal weighted binary code. For instance, to form the excess-3 representation for 4, first 3 is added, yielding 7, and then the "normal" BCD is used, which is 0111. Therefore, 0111 is the excess-3 code for the decimal digit 4. Table 2.3 shows all 10 decimal digits and the code for each.

By complementing each digit of the binary code representing a decimal digit, the 9s complement of that digit may be formed. For instance, the complement of 0100 (1 decimal) is 1011, which is 8 decimal.

The decimal number 243 coded in the excess-3 system would be 0101 0111 0110, and the decimal number 347 would be 0110 0111 1010. The 9s complement of 243 is 756 decimal, or 1010 1000 1001 binary. Table 2.4 lists the excess-3 code representation for each of the 10 decimal digits along with the 9s complements

TABLE 2.3

EXCESS-3 CODE	
DECIMAL	BINARY CODE
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

TABLE 2.4

DECIMAL DIGIT	EXCESS-3 CODE	9s COMPLEMENT
0	0011	1100
1	0100	1011
2	0101	1010
3	0110	1001
4	0111	1000
5	1000	0111
6	1001	0110
7	1010	0101
8	1011	0100
9	1100	0011

TABLE 2.5 2, 4, 2, 1, CODE

	CODED BINARY WEIGHT OF BIT			
DECIMAL	2	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	0	1
8	1	1	1	0
9	1	1	1	1

listed. Note the 9s complement of each code group may be formed by changing each 0 to a 1 and each 1 to a 0 in the code group.

The excess-3 code is not a weighted code, however, because weights cannot be assigned to the bits so that their sum equals the decimal digits represented.

A weighted code in which the 9s complement may be formed by complementing each binary digit is the 2, 4, 2, 1 code (see Table 2.5). If each bit of a code group is complemented, the 9s complement of the decimal digit represented is formed. For instance, 0010 (2 decimal) complemented is 1101 (7 decimal), and 1011 (5 decimal) complemented is 0100 (4 decimal). This code is widely used in instruments and electronic calculators.

The following convention is generally adopted to distinguish binary from decimal. A binary number is identified by a subscript of 2 placed at the end of the number (00110_2), and a decimal number by the subscript 10 (for instance, decimal 948 may be written 948_{10}). So we may write 0111_2 as 7_{10}. We use this convention when necessary.

OCTAL AND HEXADECIMAL NUMBER SYSTEMS

2.12 Two other number systems are very useful in the computer industry: the octal number system and the hexadecimal number system.

The octal number system has a base, or radix, of 8; eight different symbols

TABLE 2.6

OCTAL	DECIMAL	OCTAL	DECIMAL
0	0	11	9
1	1	12	10
2	2	13	11
3	3	14	12
4	4	15	13
5	5	16	14
6	6	17	15
7	7	20	16
10	8	21	17

are used to represent numbers. These are commonly 0, 1, 2, 3, 4, 5, 6, and 7. We show the first octal numbers and their decimal equivalents in Table 2.6.

To convert an octal number to a decimal number, we use the same sort of polynomial as was used in the binary case, except that we now have a radix of 8 instead of 2. Therefore, 1213 in octal is $1 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 3 \times 8^0 = 512 + 128 + 8 + 3 = 651$ in decimal. Also, 1.123 in octal is $1 \times 8^0 + 1 \times 8^{-1} + 2 \times 8^{-2} + 3 \times 8^{-3}$, or $1 + \frac{1}{8} + \frac{2}{64} + \frac{3}{512} = 1\frac{83}{512}$ in decimal.

There is a simple trick for converting a binary number to an octal number. Simply group the binary digits into groups of 3, starting at the octal point, and read each set of three binary digits according to Table 2.7.

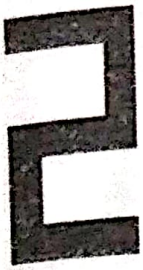
Let us convert the binary number 011101. First, we break it into 3s (thus 011 101). Then, converting each group of three binary digits, we get 35 in octal. Therefore 011101 binary = 35 octal. Here are several more examples:

$$\begin{aligned} 111110111_2 &= 767_8 \\ 110110101_2 &= 665_8 \\ 11011_2 &= 33_8 \\ 1001_2 &= 11_8 \\ 10101.11_2 &= 25.6_8 \\ 1100.111_2 &= 14.7_8 \\ 1011.1111_2 &= 13.74_8 \end{aligned}$$

Conversion from decimal to octal can be performed by repeatedly dividing the decimal number by 8 and using each remainder as a digit in the octal number

TABLE 2.7

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7



being formed. For instance, to convert 200_{10} to an octal representation, we divide as follows:

$$\begin{aligned} 200 \div 8 &= 25 \quad \text{remainder is } 0 \\ 25 \div 8 &= 3 \quad \text{remainder is } 1 \\ 3 \div 8 &= 0 \quad \text{remainder is } 3 \end{aligned}$$

Therefore, $200_{10} = 310_8$.

Notice that when the number to be divided is less than 8, we use 0 as the quotient and the number as the remainder. Let us check this:

$$310_8 = 3_{10} \times 8^2_{10} + 1_{10} \times 8^1_{10} + 0_{10} \times 8^0_{10} = 192_{10} + 8_{10} = 200_{10}$$

Here is another example. We wish to convert 3964_{10} to octal:

$$\begin{aligned} 3964 \div 8 &= 495 \quad \text{with a remainder of } 4 \\ 495 \div 8 &= 61 \quad \text{with a remainder of } 7 \\ 61 \div 8 &= 7 \quad \text{with a remainder of } 5 \\ 7 \div 8 &= 0 \quad \text{with a remainder of } 7 \end{aligned}$$

Therefore, $7574_8 = 3964_{10}$. Checking, we find

$$\begin{aligned} 7574_8 &= 7_{10} \times 8^3_{10} + 5_{10} \times 8^2_{10} + 7_{10} \times 8_{10} + 4_{10} \\ &= 7_{10} \times 512_{10} + 5_{10} \times 64_{10} + 7_{10} \times 8_{10} + 4_{10} \times 1_{10} \\ &= 3584_{10} + 320_{10} + 56_{10} + 4_{10} \\ &= 3964_{10} \end{aligned}$$

There are several other techniques for converting octal to decimal and decimal to octal, but they are not used very frequently manually, and tables prove to be of about as much value as anything in this process. Octal-to-decimal and decimal-to-octal tables are readily available in a number of places, including the manuals distributed by manufacturers of binary machines.

An important use for octal is in listings of programs and for memory "dumps" for binary machines, thus making the printouts more compact. The manuals for several of the largest manufacturers use octal numbers to represent binary numbers because of ease of conversion and compactness.

The hexadecimal number system is useful for the same reasons. Most minicomputers and microcomputers have their memories organized into sets of *bytes*, each consisting of eight binary digits. Each byte either is used as a single entity to represent a single alphanumeric character or is broken into two 4-bit pieces. (We examine the coding of alphanumeric characters using bytes in Chap. 7.) When the bytes are handled in two 4-bit pieces, the programmer is given the option of declaring each 4-bit character as a piece of a binary number or as two BCD numbers. For instance, the byte 00011000 can be declared a binary number, in which case it is equal to 24 decimal, or as two BCD characters, in which case it represents the decimal number 18.

When the machine is handling numbers in binary but in groups of four digits,

TABLE 2.8

BINARY	HEXADECIMAL	DECIMAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

it is convenient to have a code for representing each of these sets of four digits. Since 16 possible different numbers can be represented, the digits 0 through 9 will not suffice; so the letters A, B, C, D, E, and F are used also (see Table 2.8).

To convert binary to hexadecimal, we simply break a binary number into groups of four digits and convert each group of four digits according to the preceding code. Thus $10111011_2 = BB_{16}$, $10010101_2 = 95_{16}$, $11000111_2 = C7_{16}$, and $10001011_2 = 8B_{16}$. The mixture of letters and decimal digits may seem strange at first, but these are simply convenient symbols, just as decimal digits are.

The conversion of hexadecimal to decimal is straightforward but time-consuming. For instance, BB represents $B \times 16^1 + B \times 16^0 = 11 \times 16 + 11 \times 1 = 176 + 11 = 187$. Similarly,

$$\begin{aligned} AB6_{16} &= 10_{10} \times 16_{10}^2 + 11_{10} \times 16_{10} + 6_{10} \\ &= 10_{10} \times 256_{10} + 176_{10} + 6_{10} \\ &= 2560_{10} + 176_{10} + 6_{10} \\ &= 2742_{10} \end{aligned}$$

To convert, for instance, $3A6_{16}$ to decimal:

$$\begin{aligned} 3A6_{16} &= 3_{10} \times 16_{10}^2 + 10_{10} \times 16_{10} + 6_{10} \\ &= 3_{10} \times 256_{10} + 10_{10} \times 16_{10} + 6_{10} \\ &= 768_{10} + 160_{10} + 6_{10} \\ &= 934_{10} \end{aligned}$$

Again, tables are convenient for converting hexadecimal to decimal and decimal to hexadecimal. Table 2.9 is useful for converting in either direction.

The chief use of the hexadecimal system is in connection with byte-organized machines. And since most computers are now byte-organized, a knowledge of hexadecimal is essential to using manufacturers' manuals and to reading the current literature.

TABLE 2.9 HEXADECIMAL-TO-DECIMAL CONVERSION TABLE

A. INTEGER CONVERSION

EXAMPLE: 2322_{16} is $8192_{10} + 768_{10} + 32_{10} + 2_{10} = 8994.0$.

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15
Hexadecimal positions	4		3		2		1

B. FRACTIONAL CONVERSION

HEX	0123 DEC	HEX	4567 DECIMAL	HEX	0123 DECIMAL	HEX	4567 DECIMAL EQUIVALENT
.0	.0000	.00	.0000 0000	.000	.0000 0000 0000	.0000	.0000 0000 0000 0000
.1	.0625	.01	.0039 0625	.001	.0002 4414 0625	.0001	.0000 1525 8789 0625
.2	.1250	.02	.0078 1250	.002	.0004 8828 1250	.0002	.0000 3051 7578 1250
.3	.1875	.03	.0117 1875	.002	.0007 3242 1875	.0003	.0000 4577 6367 1875
.4	.2500	.04	.0156 2500	.004	.0009 7656 2500	.0004	.0000 6103 5156 2500
.5	.3125	.05	.0195 3125	.005	.0012 2070 3125	.0005	.0000 7629 3945 3125
.6	.3750	.06	.0234 3750	.006	.0014 6484 3750	.0006	.0000 9155 2734 3750
.7	.4375	.07	.0273 4375	.007	.0017 0898 4375	.0007	.0001 0681 1523 4375
.8	.5000	.08	.0312 5000	.008	.0019 5312 5000	.0008	.0001 2207 0312 5000
.9	.5625	.09	.0351 5625	.009	.0021 9726 5625	.0009	.0001 3732 9101 5625
.A	.6250	.0A	.0390 6250	.00A	.0024 4140 6250	.000A	.0001 5258 7890 6250
.B	.6875	.0B	.0429 6875	.00B	.0026 8554 6875	.000B	.0001 6784 6679 6875
.C	.7500	.0C	.0468 7500	.00C	.0029 2968 7500	.000C	.0001 8310 5468 7500
.D	.8125	.0D	.0507 8125	.00D	.0031 7382 8125	.000D	.0001 9836 4257 8125
.E	.8750	.0E	.0546 8750	.00E	.0034 1796 8750	.000E	.0002 1362 3046 8750
.F	.9375	.0F	.0585 9375	.00F	.0036 6210 9375	.000F	.0002 2888 1835 9375
Hexidecimal positions	1		2		3		4

Quite a large number of questions have been included for this chapter. For those desiring to study octal and hexadecimal number systems further, Questions 2.58 through 2.67 contain information and exercises on octal addition and multiplication, and Questions 2.68 through 2.72 can be used to supplement the study of the hexadecimal system.

QUESTIONS

SUMMARY

2.13 The individual memory cells used in computers are bistable in operation and capable of storing a single binary bit. Therefore, it is most practical to use the binary number system to represent numbers, and this system was explained along with conversion techniques to and from decimal.

Negative numbers are represented in computers by using a sign bit which is a 1 when the number is negative and a 0 for positive numbers. Negative numbers are often represented by using 1s or 2s complement form, and this was described along with examples showing how mixed numbers represented in that form can be added or subtracted.

The direct representation of decimal numbers can be accomplished by using a binary-coded-decimal (BCD) representation. This was explained, and examples were given.

The octal and hexidecimal number systems were described. These are useful in representing binary numbers in a compact form and to facilitate communication of values in written presentations. Computers are often organized with numbers represented in groups of 8 bits which makes hexadecimal particularly useful at this time.

QUESTIONS

2.1 Convert the following decimal numbers to equivalent binary numbers:

(*a*) 43 (*b*) 64 (*c*) 4096
(*d*) 0.375 (*e*) $\frac{27}{32}$ (*f*) 0.4375
(*g*) 512.5 (*h*) 131.5625 (*i*) 2048.0625

2.2 Convert the following numbers to the equivalent binary numbers:

(*a*) 14 (*b*) 0.25 (*c*) $2\frac{1}{8}$
(*d*) 6.25 (*e*) $2\frac{3}{8}$ (*f*) 0.625

2.3 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1101 (*b*) 11011 (*c*) 1011
(*d*) 0.1011 (*e*) 0.001101 (*f*) 0.001101101
(*g*) 111011.1011 (*h*) 1011011.001101 (*i*) 10110.0101011101

2.4 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1011 (*b*) 11000 (*c*) 100011
(*d*) 11011 (*e*) 111001 (*f*) 1011010

2.5 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 1011 (*b*) 100100 (*c*) 10011
(*d*) 0.1101 (*e*) 0.1001 (*f*) 0.0101
(*g*) 1011.0011 (*h*) 1001.1001 (*i*) 101.011

2.6 Convert the following binary numbers to equivalent decimal numbers:

(*a*) 0.111 (*b*) 0.11011 (*c*) 1.011
(*d*) 111.1011 (*e*) 0110.0101 (*f*) 101.101011

2.7 Perform the following additions and check by converting the binary numbers to decimal:

(*a*) 1001.1 + 1011.01 (*b*) 100101 + 100101
(*c*) 0.1011 + 0.1101 (*d*) 1011.01 + 1001.11

2.8 Perform the following additions and check by converting the binary numbers to decimal and adding:

(*a*) 1011 + 1110 (*b*) 1010 + 1111 (*c*) 10.11 + 10.011
(*d*) 1101.11 + 1.11 (*e*) 11111.1 + 10010.1 (*f*) 101.1 + 111.11

2.9 Perform the following additions and check by converting the binary numbers to decimal:

(*a*) 1101.1 + 1011.1 (*b*) 101101 + 1101101
(*c*) 0.0011 + 0.1110 (*d*) 1100.011 + 1011.011

2.10 Perform the following subtractions in binary and check by converting the numbers to decimal and subtracting:

(*a*) 1101 − 1000 (*b*) 1101 − 1001 (*c*) 1011.1 − 101.1
(*d*) 1101.01 − 1011.1 (*e*) 111.11 − 101.1 (*f*) 1101.1 − 1010.01

2.11 Perform the following subtractions in the binary number system:

(*a*) 64 − 32 (*b*) 127 − 63
(*c*) 93.5 − 42.75 (*d*) $84\frac{9}{32} - 48\frac{5}{16}$

2.12 Perform the following subtractions in the binary number system:

(*a*) 128 − 32 (*b*) $\frac{1}{8} - \frac{1}{16}$ (*c*) $2\frac{1}{8} - 4\frac{3}{32}$
(*d*) $31 - \frac{5}{8}$ (*e*) $62 - 31\frac{1}{16}$ (*f*) 129 − 35

2.13 Perform the following subtractions in the binary number system:

(*a*) 37 − 35 (*b*) 128 − 64
(*c*) 94.5 − 43.75 (*d*) 255 − 127

2.14 Perform the following multiplications and divisions in the binary number system:

(*a*) 16 × 8 (*b*) 31 × 14 (*c*) 23 × 3.525
(*d*) 15 × 8.625 (*e*) 6 ÷ 2 (*f*) 16 ÷ 8

2.15 Perform the following multiplications and divisions in the binary number system:

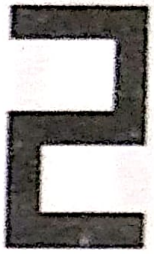
(*a*) 24 × 12 (*b*) 18 × 14 (*c*) 32 ÷ 8
(*d*) 27 ÷ 18 (*e*) 49.5 × 51.75 (*f*) 58.75 ÷ 23.5

2.16 Perform the following multiplications and divisions in the binary number system:

(*a*) 16 × 2.75 (*b*) 19 ÷ 6 (*c*) $256\frac{1}{2} \div 128\frac{1}{4}$
(*d*) 31.5 ÷ 15.75 (*e*) $3 \div \frac{5}{8}$ (*f*) $2\frac{5}{8} \times 1\frac{5}{8}$

2.17 Perform the following multiplications and divisions in the binary number system:

(*a*) 15 × 13 (*b*) 10 × 15 (*c*) 44 ÷ 11
(*d*) 42 ÷ 12 (*e*) 7.75 × 2.5 (*f*) 22.5 × 4.75



QUESTIONS

2.18 Convert the following decimal numbers to both their 9s and 10s complements:

(*a*) 9 (*b*) 19 (*c*) 8
(*d*) 24 (*e*) 25 (*f*) 99

2.19 Convert the following decimal numbers into both their 9s and 10s complements:

(*a*) 5436 (*b*) 1932 (*c*) 45.15 (*d*) 18.293

2.20 Convert the following decimal numbers into both their 9s and 10s complements:

(*a*) 95 (*b*) 79 (*c*) 0.83
(*d*) 0.16 (*e*) 298.64 (*f*) 332.52

2.21 Convert the following decimal numbers to both their 9s and 10s complements:

(*a*) 3654 (*b*) 2122 (*c*) 54.19 (*d*) 37.263

2.22 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1101 (*b*) 1010 (*c*) 1111
(*d*) 1110 (*e*) 1011 (*f*) 1011

2.23 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1011 (*b*) 11011 (*c*) 1011.01 (*d*) 11011.01

2.24 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 1011 (*b*) 1101 (*c*) 0.0111
(*d*) 0.101 (*e*) 11.101 (*f*) 101.011

2.25 Convert the following binary numbers to both their 1s and 2s complements:

(*a*) 101111 (*b*) 100100
(*c*) 10111.10 (*d*) 10011.11

2.26 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 8 − 4 (*b*) 16 − 8 (*c*) 198 − 124
(*d*) 28.5 − 23.4 (*e*) 27.6 − 23.4 (*f*) 0.55 − 0.42

2.27 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 948 − 234 (*b*) 347 − 263
(*c*) 349.5 − 245.3 (*d*) 412.7 − 409.2

2.28 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 14 − 9 (*b*) 15 − 9 (*c*) 0.5 − 40.24
(*d*) 0.41 − 0.4 (*e*) 0.434 − 0.33 (*f*) 1.2 − 0.34

2.29 Perform the following subtractions, using both 9s and 10s complements:

(*a*) 1024 − 913 (*b*) 249 − 137
(*c*) 24.1 − 13.4 (*d*) 239.3 − 119.4

2.30 Perform the following subtractions of binary numbers, using both 1s and 2s complements:

(*a*) 1010 − 1011 (*b*) 110 − 10 (*c*) 110 − 0.111
(*d*) 0.111 − 0.1001 (*e*) 0.1111 − 0.101 (*f*) 11.11 − 10.111

NUMBER SYSTEMS

2.31 Perform the following subtractions, using both 1s and 2s complements:
(*a*) 1011 − 101 (*b*) 11011 − 11001
(*c*) 10111.1 − 10011.1 (*d*) 11011 − 10011.11

2.32 How many different numbers can be stored in a set of four switches, each having three different positions (four three-position switches)?

2.33 How many different binary numbers can be stored in a register consisting of six switches?

2.34 How many different BCD numbers can be stored in 12 switches? (Assume two-position, or on-off switches.)

2.35 How many different BCD numbers can be stored in a register containing 12 switches using an 8, 4, 2, 1 code? Using an excess-3 code?

2.36 Write the first 12 numbers in the base 4 (or *quaternary*) number system.

2.37 Write the first 10 numbers in the quaternary number system, which has a base, or radix, of 4. Use the digits 0. 1, 2, and 3 to express these numbers.

2.38 Write the first 20 numbers in the base 12 (or *duodecimal*) number system. Use A for 10 and B for 11.

2.39 Write the first 25 numbers in a base 11 number system, using the digits 0, 1, 2, 3, 4, 5, 6, 7, 8; 9, and A to express the 25 numbers that you write. (Decimal 10 = A, for instance.)

2.40 Perform the following subtractions in the binary number system, using 1s complements:
(*a*) 1111 − 1001 (*b*) 1110 − 1011
(*c*) 101.11 − 101.01 (*d*) 111.1 − 100.1

2.41 Using the 1s complement number system, perform the following subtractions:
(*a*) 0.1001 − 0.0110 (*b*) 0.1110 − 0.0110
(*c*) 0.01111 − 0.01001 (*d*) 11011 − 11001
(*e*) 1110101 − 1010010

2.42 Perform the following subtractions in the binary number system, using 2s complements:
(*a*) 1111 − 110 (*b*) 1110 − 1100
(*c*) 1011.11 − 101.001 (*d*) 111.1 − 110.1

2.43 Using the 2s complement number system, perform the following subtractions and represent the answers as decimal fractions:
(*a*) 0.101010 − 0.010101 (*b*) 0.11001 − 0.00100
(*c*) 0.111000 − 0.000111 (*d*) 0.101100 − 0.010011

2.44 Convert the following hexadecimal numbers to decimal numbers:
(*a*) 15 (*b*) B8 (*c*) AB4
(*d*) 9.B (*e*) 9.1A

2.45 Convert the following hexadecimal numbers to decimal:
(*a*) B6C7 (*b*) 64AC (*c*) A492 (*d*) D2763

QUESTIONS

2.46 Convert the following octal numbers to decimal:
(*a*) 15 (*b*) 125 (*c*) 115
(*d*) 124 (*e*) 156 (*f*) 15.6

2.47 Convert the following octal numbers to decimal:
(*a*) 2376 (*b*) 2473 (*c*) 276431 (*d*) 22632

2.48 Convert the following binary numbers to octal:
(*a*) 110 (*b*) 111001 (*c*) 111.111
(*d*) 0.11111 (*e*) 10.11 (*f*) 1111.1101

2.49 Convert the following binary numbers to octal:
(*a*) 101101 (*b*) 101101110 (*c*) 10110111
(*d*) 110110.011 (*e*) 011.1011011

2.50 Convert the following octal numbers to binary:
(*a*) 54 (*b*) 44 (*c*) 232.2
(*d*) 232.4 (*e*) 453.45 (*f*) 31.234

2.51 Convert the following octal numbers to binary:
(*a*) 7423 (*b*) 3364 (*c*) 33762
(*d*) 3232.14 (*e*) 3146.52

2.52 Convert the following decimal numbers to octal:
(*a*) 17 (*b*) 8 (*c*) 19
(*d*) 0.55 (*e*) 0.625 (*f*) 2.125

2.53 Convert the following decimal numbers to octal:
(*a*) 932 (*b*) 332 (*c*) 545.375
(*d*) 632.97 (*e*) 4429.625

2.54 Convert the following hexadecimal numbers to binary:
(*a*) 9 (*b*) 1B (*c*) 0.A1
(*d*) 0.AB (*e*) AB (*f*) 12.B

2.55 Convert the following hexadecimal numbers to binary:
(*a*) CD (*b*) 6A9 (*c*) A14
(*d*) AA.1A (*e*) AB2.234

2.56 Convert the following binary numbers to hexadecimal:
(*a*) 1101.0110 (*b*) 11011110 (*c*) 1111
(*d*) 11101 (*e*) 11110.01011 (*f*) 1011.11010

2.57 Convert the following binary numbers to hexadecimal:
(*a*) 10110111 (*b*) 10011100 (*c*) 1001111
(*d*) 0.01111110 (*e*) 101101111010

2.58 A simple rule for multiplying two digits in any radix is simply to multiply the two digits in decimal. If the product is less than the radix, take it; if greater, divide (in decimal) by the radix and use the remainder as the first, or least significant, position and the quotient as the carry, or most significant, digit. In base 6, then $2 \times 2 = 4$, $3 \times 1 = 3$, etc.; however, $2 \times 4 = 8$, and

$$6\overline{)8}\ \ 1$$

So $2_6 \times 4_6 = 12_6$. Similarly, in base 7, then, $3 \times 4 = 12$ and

$$7\overline{)12}\quad 1,\ \text{remainder } 5$$

So $3_7 \times 4_7 = 15_7$. Using this rule, perform, in base 7, the following:

(*a*) $2_7 \times 3_7$ (*b*) $2_7 \times 2_7$
(*c*) $4_7 \times 4_7$ (*d*) $4_7 \times 3_7$

2.59 Using the rule in Question 2.58, perform:

(*a*) $3_6 \times 4_6$ (*b*) $3_6 \times 3_6$ (*c*) $3_9 \times 4_9$
(*d*) $4_9 \times 5_9$ (*e*) $5_9 \times 15_9$

2.60 An addition table for octal is as follows:

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

Using this table, we add in octal:

$$\begin{array}{r} 1\ 1 \longleftarrow \text{carries} \\ 1\ 2\ 6 \\ 3\ 5\ 7 \\ \hline 5\ 0\ 5 \end{array}$$

Perform the following additions:

(*a*) $7_8 + 7_8$ (*b*) $6_8 + 5_8$ (*c*) $7_8 + 16_8$
(*d*) $5_8 + 4_8$ (*e*) $5_8 + 14_8$

2.61 Using the table in Question 2.60, perform:

(*a*) $15_8 + 14_8$ (*b*) $24_8 + 36_8$ (*c*) $126_8 + 347_8$
(*d*) $67_8 + 45_8$ (*e*) $136_8 + 636_8$

2.62 Make up a hexadecimal addition table.

2.63 Using the table in Question 2.62, perform:

(*a*) $6_{16} + A1_{16}$ (*b*) $7_{16} + 17_{16}$ (*c*) $8_{16} + 28_{16}$
(*d*) $A16A_{16} + B16A_{16}$ (*e*) $A84_{16} + A83_{16}$

QUESTIONS

2.64 Perform the additions in Question 2.63 in binary and convert back to hexadecimal.

2.65 Perform the additions in Question 2.61 in binary and convert back to octal.

2.66 To multiply two numbers in octal, we use the rule given in Question 2.58 and then proceed as follows:

$$\begin{aligned} 6 \times 27 &= 6 \times 20 + 6 \times 7 \\ &= 140 + 52 = 212 \end{aligned}$$

Multiply the following in octal:

(*a*) 6×7 (*b*) 6×10 (*c*) 5×14

2.67 To multiply numbers of more than one digit, proceed as in Question 2.66 and then add in octal. Multiply the following octal numbers:

(*a*) 3×14 (*b*) 23×12 (*c*) 11×22
(*d*) 22×44 (*e*) 13×13 (*f*) 14×15

2.68 Perform the following multiplications of hexadecimal numbers:

(*a*) $A \times 8$ (*b*) 9×14 (*c*) $A1 \times 8$
(*d*) $A11 \times 9$ (*e*) $A12 \times 6$ (*f*) $A13 \times 2B$

2.69 Using the rule in Question 2.58, perform the following multiplications of hexadecimal numbers:

(*a*) $15 \times B$ (*b*) $14 \times B$ (*c*) $11 \times A$
(*d*) $142 \times A$ (*e*) 13×14

2.70 Perform the multiplications in Question 2.68 in binary, then convert back to hexadecimal.

2.71 Perform the multiplications in Question 2.69 in binary, then convert back to hexadecimal.

2.72 In converting decimal numbers to hexadecimal, it is convenient to go first to octal, then to binary, then to hexadecimal. For instance, to convert 412_{10} to hexadecimal, we go first to octal:

$$\begin{array}{r} 51 \\ 8\overline{)412} \\ \underline{40} \\ 12 \\ \underline{8} \\ 4 \end{array} \qquad \begin{array}{r} 6 \\ 8\overline{)51} \\ \underline{48} \\ 3 \end{array}$$

4 ← 3d digit; 3 ← 2d digit; 6 ← 1st digit

Now $412_{10} = 634_8$; converting this to binary gives

$$634_8 = \underbrace{110}_{6}\ \underbrace{011}_{3}\ \underbrace{100}_{4}$$

Then regrouping yields

$$\underbrace{1}_{1}\ \underbrace{1001}_{9}\ \underbrace{1100}_{C}$$

So

$$412_{10} = 19C$$

Convert the following decimal numbers to hexadecimal:

(*a*) 24 (*b*) 397 (*c*) 1343
(*d*) 513 (*e*) 262

3

BOOLEAN ALGEBRA AND GATE NETWORKS

There are several advantages in having a mathematical technique for the description of the internal workings of a computer. For one thing, it is often far more convenient to calculate with expressions used to represent switching circuits than it is to use schematic or even logical diagrams. Further, just as an ordinary algebraic expression may be simplified by means of the basic theorems, the expression describing a given switching circuit network may be reduced or simplified. This enables the logical designer to simplify the circuitry used, achieving economy of construction and reliability of operation. Boolean algebra also provides an economical and straightforward way of describing the circuitry used in computers. In all, a knowledge of boolean algebra is indispensable in the computing field.

OBJECTIVES

1 The design and maintenance of digital computers are greatly facilitated by the use of boolean algebra and block diagrams. Both of these are explained, as is their usage in designing networks using logic gates.

2 The major types of gates now in use are AND, OR, NOR, and NAND gates. These are explained, and design procedures using these gates are presented.

3 To simplify the construction of computers, the gate networks are simplified as much as possible. Both algebraic techniques and graphical (map) techniques exist which can be used; both are discussed with emphasis on the map minimization procedures.

4 Logic networks are generally laid out in a two-level form such as AND-OR, NAND-NAND, etc. These forms are described, and design procedures for the forms are presented.

5 There are several special characteristics of gates which can influence logic design (such as wired OR and wired AND gates). These are described, as are special forms such as NAND-AND and NOR-OR.

6 Integrated-circuit (IC) manufacturing techniques now make it possible to package many gates in a single IC package (chip). Often these arrays of gates are laid out in some regular form for large-scale integration and typical forms are presented, including those for programmable logic arrays, programmable array logic, and gate array logic.

FUNDAMENTAL CONCEPTS OF BOOLEAN ALGEBRA

3.1 When a variable is used in an algebraic formula, it is generally assumed that the variable may take any numerical value. For instance, in the formula $2X - 5Y = Z$, we assume that X, Y, and Z may range through the entire field of real numbers.

The variables used in boolean equations have a unique characteristic, however; they may assume only one of two possible values. These two values may be

†Or T and F, or + and −, etc. However, 0 and 1 are almost universally used in computer work.

represented by the symbols 0 and 1.† If an equation describing logical circuitry has several variables, it is still understood that each of the variables can assume only the value 0 or 1. For instance, in the equation $X + Y = Z$, each of the variables X, Y, and Z may have only the values 0 or 1.

This concept will become clearer if a symbol is defined, the + symbol. When the + symbol is placed between two variables, say X and Y, since both X and Y can take only the role 0 or 1, we can define the + symbol by listing all possible combinations for X and Y and the resulting values of $X + Y$.

The possible input and output combinations may be arranged as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

This is a *logical* addition table and could represent a standard binary addition table except for the last entry. When both X and Y represent 1s, the value of $X + Y$ is 1. The + symbol, therefore, does not have the "normal" meaning, but is a logical addition or logical OR symbol. The equation $X + Y = Z$ can be read "X or Y equals Z" or "X plus Y equals Z." This concept may be extended to any number of variables. For instance, in the equation $A + B + C + D = E$, even if A, B, C, and D all had the value of 1, E would represent only a 1.

To avoid ambiguity, a number of other symbols have been recommended as replacements for the + sign. Some of these[3] are ∪, v, and ∨. Computer people still use the + sign, however, which was the symbol originally proposed by Boole.

LOGICAL MULTIPLICATION

3.2 A second important operation in boolean algebra we call *logical multiplication* or the *logical* AND *operation*.[4] The rules for this operation can be given by simply listing all values that might occur:

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

Thus, for instance, if we write $Z = X \cdot Y$ and find $X = 0$ and $Y = 1$, then $Z = 0$. Only when X and Y are both 1s would Z be a 1.

Both + and · obey a mathematical rule called the *associative law*. This law says, for +, that $(X + Y) + Z = X + (Y + Z)$ and, for ·, that $X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$. This means that we can write $X + Y + Z$ without ambiguity, for no matter in what order the operation is performed, the result is the same. That is,

---

[3]The preceding equation might then be written $A \cup B \cup C \cup D = E$.

[4]It is necessary to know both the terms *logical addition* and *OR operation* for the + symbol and the terms *logical multiplication* and *AND operation* for the · symbol since all these terms are actively used in computer manuals, technical journals, and trade magazines. The term $X + Y$ is called a *sum term* or *OR term* in computer literature, for example.

3

BOOLEAN ALGEBRA AND GATE NETWORKS

ORing X and Y and then ORing Z gives the same result as ORing Y and Z and then ORing X. We can test this for both $+$ and $\cdot$ by trying all combinations.

Note that while either $+$'s or $\cdot$'s can be used freely, the two cannot be mixed without ambiguity in the absence of further rules. For instance, does $A \cdot B + C$ mean $(A \cdot B) + C$ or $A \cdot (B + C)$? The two form different values for $A = 0$, $B = 0$, and $C = 1$, for then we have $(0 \cdot 0) + 1 = 1$ and $0 \cdot (0 + 1) = 0$, which differ. (Always operating from left to right will alleviate this. This technique is used in some programming languages, but not usually by algebraists or computer designers or maintenance personnel.) The rule which is used is that $\cdot$ is always performed before $+$. Thus $X \cdot Y + Z$ is the same as $(X \cdot Y) + Z$, and $X \cdot Y + X \cdot Z$ means the same as $(X \cdot Y) + (X \cdot Z)$.

AND GATES AND OR GATES

3.3 The $+$ and $\cdot$ operations are physically realized by two types of electronic circuits, called *OR gates* and *AND gates*. We treat these as "black boxes," deferring until later any discussion of how the actual circuitry operates.

A *gate* is simply an electronic circuit which operates on one or more input signals to produce an output signal. One of the simplest and most frequently used gates is called the OR gate, and the block diagram symbol for the OR gate is shown in Fig. 3.1, as is the table of combinations for the inputs and outputs for the OR gate. Since the inputs X and Y are signals with values either 0 or 1 at any given time, the output signal Z can be described by simply listing all values for X and Y and the resulting value for Z. A study of the table in Fig. 3.1 indicates that the OR gate ORs or logically adds its inputs.

Similarly, the AND gate in Fig. 3.2 ANDs or logically multiplies input values, yielding an output Z with value $X \cdot Y$, so that Z is a 1 only when both X and Y are 1s.

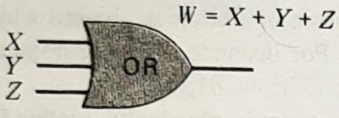
Just as the $+$ and $\cdot$ operations could be extended to several variables by using the associative law, OR gates and AND gates can have more than two inputs. Figure 3.3 shows three input OR and AND gates and the table of all input combinations for each. As might be hoped, the OR gate with input X, Y, and Z has a 1 output if X or Y or Z is a 1, so that we can write $X + Y + Z$ for its output.

FIGURE 3.1

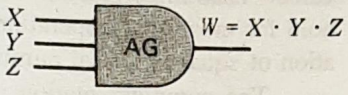
OR gate.

INPUT		OUTPUT
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

INPUT X	INPUT Y	OUTPUT Z	
0	0	0	0 · 0 = 0
0	1	0	0 · 1 = 0
1	0	0	1 · 0 = 0
1	1	1	1 · 1 = 1



INPUT X	INPUT Y	INPUT Z	OUTPUT W
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



INPUT X	INPUT Y	INPUT Z	OUTPUT W
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

FIGURE 3.3

Three-input OR and AND gates.

Also, the output of the AND gate with inputs, X, Y, and Z is a 1 only when all three of the inputs are 1s, so that we can write the output as $X \cdot Y \cdot Z$.

The above argument can be extended. A four-input OR gate has a 1 output when any of its inputs is a 1, and a four-input AND gate has a 1 output only when all four inputs are 1s.

It is often convenient to shorten $X \cdot Y \cdot Z$ to XYZ, and we sometimes use this convention.

COMPLEMENTATION AND INVERTERS

3.4 The two operations defined so far have been what algebraists would call *binary operations* in that they define an operation on two variables. There are also *singular* or *unary, operations*, which define an operation on a single variable. A

familiar example of unary operation is $-$, for we can write -5 or -10 or $-X$, meaning that we are to take the negative of these values. (The $-$ is also used as a binary operation symbol for subtraction, which makes it a familiar but ambiguous example.)

In boolean algebra we have an operation called *complementation*, and the symbol we use is $\overline{\ }$. Thus we write $\overline{X}$, meaning "take the complement of X," or $\overline{(X + Y)}$, meaning "take the complement of $X + Y$." The complement operation can be defined quite simply:

$$\overline{0} = 1$$
$$\overline{1} = 0$$

The complement of a value can be taken repeatedly. For instance, we can find $\overline{\overline{X}}$: For $X = 0$ it is $\overline{\overline{\overline{0}}} = \overline{\overline{1}} = \overline{0} = 1$, and for $X = 1$ it is $\overline{\overline{\overline{1}}} = \overline{\overline{0}} = \overline{1} = 0$.

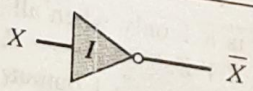
A useful rule is based on the fact that $\overline{\overline{X}} = X$. Checking, we find that $\overline{\overline{0}} = \overline{1} = 0$ and $\overline{\overline{1}} = \overline{0} = 1$. [This rule—that double complementation gives the original value—is an important characteristic of a boolean algebra which does not generally hold for most unary operations. For instance, the rule does not hold for the operation of squaring a real number: $(3^2)^2 = 81$, not 3.]

The complementation operation is physically realized by a gate or circuit called an *inverter*. Figure 3.4(*a*) shows an inverter and the table of combinations for its input and output. Figure 3.4(*b*) shows also that connecting two inverters in series gives an output equal to the input, and this is the gating counterpart to the law of double complementation, $\overline{\overline{X}} = X$.

Several other symbols have been used for the complementation symbol. For instance, $\sim$ is often used by logicians who write $\sim X$ and read this "the negation of X." The symbol $'$ has been used by mathematicians and computer people; thus X' is the complement of X in these systems. The overbar symbol is now used by the American National Standards Institute and military standards, as well as by most journals and manufacturers, and we use it.

FIGURE 3.4

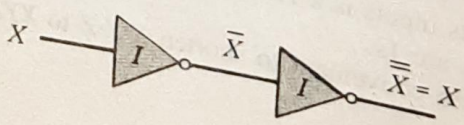
(*a*) Block diagram of an inverter. (*b*) Two inverters in series.



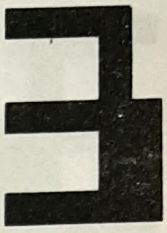
INPUT X	OUTPUT $\overline{X}$
0	1
1	0

$\overline{0} = 1$
$\overline{1} = 0$

(*a*)



(*b*)



3.5 The tables of values for the three operations just explained are sometimes called *truth tables*, or *tables of combinations*. To study a logical expression, it is very useful to construct a table of values for the variables and then to evaluate the expression for each of the possible combinations of variables in turn. Consider the expression $X + Y\overline{Z}$. There are three variables in this expression: X, Y, and Z, each of which can assume the value 0 or 1. The possible combinations of values may be arranged in ascending order,[5] as in Table 3.1.

One of the variables, Z, is complemented in the expression $X + Y\overline{Z}$. So a column is now added to the table listing values of $\overline{Z}$ (see Table 3.2).

A column is now added listing the values that $Y\overline{Z}$ assumes for each value of X, Y, and Z. This column will contain the value 1 only when both Y is a 1 and $\overline{Z}$ is a 1 (see Table 3.3).

Now the ORing, or logical addition, of the values of X to the values which have been calculated for $Y\overline{Z}$ is performed in a final column (see Table 3.4).

The final column contains the value of $X + Y\overline{Z}$ for each set of input values which X, Y, and Z may take. For instance, when $X = 1$, $Y = 0$, and $Z = 1$, the expression has the value of 1.

[5]Note that the variables in each row of this table may be combined into a binary number. The binary numbers will then count from 000 to 111 in binary, or from 0 to 7 decimal. Sometimes each row is numbered in decimal according to the number represented. Then reference may be made to the row by using the decimal number. For instance, row 0 has values of 0, 0, 0, for X, Y, and Z, row 6 has values of 1, 1, 0, and row 7 has values of 1, 1, 1.

TABLE 3.1

X	Y	Z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

TABLE 3.2

X	Y	Z	$\overline{Z}$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

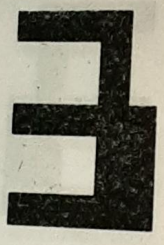


TABLE 3.3

X	Y	Z	$\overline{Z}$	$Y\overline{Z}$
0	0	0	1	0
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

TABLE 3.4

X	Y	Z	$\overline{Z}$	$Y\overline{Z}$	$X + Y\overline{Z}$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	1	1
1	1	1	0	0	1

EVALUATION OF AN EXPRESSION CONTAINING PARENTHESES

3.6 The following example illustrates the procedure for constructing a truth table for the expression $X + Y(\overline{X} + \overline{Y})$. There are only two variables in the expression, X and Y. First a table of the values which X and Y may assume is constructed (see Table 3.5).

Now, since the expression contains both $\overline{X}$ and $\overline{Y}$, two columns are added listing complements of the original values of the variables (see Table 3.6).

The various values of $\overline{X} + \overline{Y}$ are now calculated (see Table 3.7).

The values for $\overline{X} + \overline{Y}$ are now multiplied (ANDed) by the values of Y in the table, forming another column representing $Y(\overline{X} + \overline{Y})$ (see Table 3.8).

Finally the values for $Y(\overline{X} + \overline{Y})$ are added (ORed) to the values for X which are listed, forming the final column and completing the table (see Table 3.9).

Inspection of the final column of the table indicates that the values taken by the function $X + Y(\overline{X} + \overline{Y})$ are identical with the values found in the table for ORing X and Y. This indicates that the function $X + Y(\overline{X} + \overline{Y})$ is equivalent to

TABLE 3.5

X	Y
0	0
0	1
1	0
1	1

TABLE 3.6

X	Y	$\overline{X}$	$\overline{Y}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	0

TABLE 3.7

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} + \overline{Y}$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

TABLE 3.8

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} + \overline{Y}$	$Y(\overline{X} + \overline{Y})$
0	0	1	1	1	0
0	1	1	0	1	1
1	0	0	1	1	0
1	1	0	0	0	0

TABLE 3.9

X	Y	$\overline{X}$	$\overline{Y}$	$\overline{X} + \overline{Y}$	$Y(\overline{X} + \overline{Y})$	$X + Y(\overline{X} + \overline{Y})$
0	0	1	1	1	0	0
0	1	1	0	1	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	1

the function $X + Y$. This equivalence has been established by trying each possible combination of values in the variables and noting that both expressions then have the same value. This is called a *proof by perfect induction*. If a logic circuit were constructed for each of the two expressions, both circuits would perform the same function, yielding identical outputs for each combination of inputs.

BASIC LAWS OF BOOLEAN ALGEBRA

3.7 Some fundamental relations of boolean algebra have been presented. A complete set of the basic operations is listed below.[6] Although simple in appearance,

[6]Actually, a number of possible sets of postulates may be used to define the algebra. The particular treatment of boolean algebra given here is derived from that of E. V. Huntington and M. H. Stone. The author would also like to acknowledge the influence of I. S. Reed and S. H. Caldwell on this development of the concepts of the algebra.

N ALGEBRA
TE
KS

TABLE 3.10 BOOLEAN ALGEBRA RULES

1	$0 + X = X$
2	$1 + X = 1$
3	$X + X = X$
4	$X + \overline{X} = 1$
5	$0 \cdot X = 0$
6	$1 \cdot X = X$
7	$X \cdot X = X$
8	$X \cdot \overline{X} = 0$
9	$\overline{\overline{X}} = X$
10	$X + Y = Y + X$
11	$X \cdot Y = Y \cdot X$
12	$X + (Y + Z) = (X + Y) + Z$
13	$X(YZ) = (XY)Z$
14	$X(Y + Z) = XY + XZ$
15	$X + XZ = X$
16	$X(X + Y) = X$
17	$(X + Y)(X + \overline{Z}) = X + YZ$
18	$X + \overline{X}Y = X + Y$
19	$XY + YZ + \overline{Y}Z = XY + Z$

these rules may be used to construct a boolean algebra,[7] determining all the relations that follow:

$$\text{If } X \neq 0, \qquad \text{then } X = 1$$

and

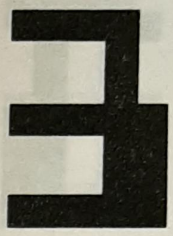
$$\text{If } X \neq 1, \qquad \text{then } X = 0$$

OR OPERATION (LOGICAL ADDITION)	AND OPERATION (LOGICAL MULTIPLICATION)	COMPLEMENT RULES
$0 + 0 = 0$	$0 \cdot 0 = 0$	$\overline{0} = 1$
$0 + 1 = 1$	$0 \cdot 0 = 0$	$\overline{1} = 0$
$1 + 0 = 1$	$1 \cdot 0 = 0$	
$1 + 1 = 1$	$1 \cdot 1 = 1$	

A list of useful relations is presented in Table 3.10. Most of the basic rules by which boolean algebra expressions may be manipulated are contained in this table. Each rule may be proved by using the proof by perfect induction. An example of this proof for rule 3 in Table 3.10 is as follows: The variable X can have only the value 0 or 1. If X has the value 0, then $0 + 0 = 0$; if X has the value 1, then $1 + 1 = 1$. Therefore $X + X = X$.

---

[7]These rules are used to construct an *example*, or *realization*, of a boolean algebra. We note that, strictly speaking, this boolean algebra consists of a set B of two elements which we call 0 and 1, an addition operation $+$, a multiplication operation $\cdot$, and a complement operation $\overline{\ }$. There are other boolean algebras (an infinite number), but this was Boole's original algebra. This algebra is sometimes called *switching algebra* to identify it more closely, but it is the same as propositional calculus, for instance.



The same basic technique may be used to prove the remainder of the rules. Rule 9 states that double complementation of a variable results in the original variable. If X equals 0, then the first complement is 1 and the second will be 0, the original value. If the original value for X is 1, then the first complement will be 0 and the second 1, the original value. Therefore $X = \overline{\overline{X}}$.

Rules 10 and 11, which are known as the *commutative laws*, express the fact that the order in which a combination of terms is performed does not affect the result of the combination. Rule 10 is the commutative law of addition, which states that the order of addition or ORing does not affect the sum ($X + Y = Y + X$). Rule 11 is the commutative law of multiplication ($XY = YX$), which states that the order of multiplication or ANDing does not affect the product.

Rules 12 and 13 are the *associative laws*. Rule 12 states that in the logical addition of several terms, the sum which will be obtained if the first term is added to the second and then the third term is added will be the same as the sum obtained if the second term is added to the third and then the first term is added [$X + (Y + Z) = (X + Y) + Z$]. Rule 13 is the associative law of logical multiplication, stating that in a product with three factors, *any* two may be multiplied, followed by the third [$X(YZ) = (XY)Z$].

Rule 14, the *distributive law*, states that the product of a variable (X) times a sum ($Y + Z$) is equal to the sum of the products of the variable multiplied by each term of the sum [$X(Y + Z) = XY + XZ$].

The three laws, commutative, associative, and distributive, may be extended to include any number of terms. For instance, the commutative law for logical addition states that $X + Y = Y + X$. This may be extended to

$$X + Y + Z + A = A + Y + Z + X$$

The commutative law for logical multiplication also may be extended: $XYZ = YZX$. These two laws are useful in rearranging the terms of an equation.

The terms also may be combined:

$$(X + Y) + (Z + A) = (A + Y) + (X + Z)$$

and $(XY)(ZA) = (XA)(ZY)$. These two laws are useful in regrouping the terms of an equation.

The distributive law may be extended in several ways:

$$X(Y + Z + A) = XY + XZ + XA$$

If two sums, such as $W + X$ and $Y + Z$, are to be multiplied, then one of the sums is treated as a single term and multiplied by the individual terms of the other sum. The results are then multiplied according to the distributive law. For instance.

$$(W + X)(Y + Z) = W(Y + Z) + X(Y + Z) = WY + WZ + XY + XZ$$

3.8 Notice that, among others, rule 17 does not apply to "normal" algebra. The rule may be obtained from the preceding rules as follows:

$$\begin{aligned}
(X + Y)(X + Z) &= XX + XZ + XY + YZ && \text{where } XX = X, \text{ rule 7} \\
&= X + XZ + XY + YZ \\
&= X + XY + XZ + YZ \\
&= X(1 + Y) + Z(X + Y) && \text{where } 1 + Y = 1, \text{ rule 2} \\
&= X + Z(X + Y) \\
&= X + XZ + YZ \\
&= X(1 + Z) + YZ && \text{where } 1 + Z = 1, \text{ rule 2} \\
&= X + YZ
\end{aligned}$$

Therefore

$$(X + Y)(X + Z) = X + YZ$$

Since rule 17 does not apply to normal algebra, it is interesting to test the rule by using the proof by perfect induction. It will be necessary to construct truth tables for the right-hand $(X + YZ)$ and left-hand $[(X + Y)(X + Z)]$ members of the equation and compare the results (see Tables 3.11 and 3.12).

The last column of the table for the function $X + YZ$ is identical with the last column of the table for $(X + Y)(X + Z)$. This proves (by means of the proof by perfect induction) that the expressions are equivalent.

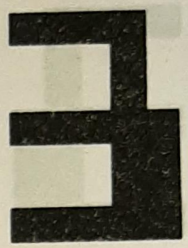
Rules 15 and 16 are also not rules in normal algebra. The following is a

TABLE 3.11

X	Y	Z	YZ	$X + YZ$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

TABLE 3.12

X	Y	Z	$X + Y$	$X + Z$	$(X + Y)(X + Z)$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1



proof of rule 15 using preceding rules:

$$X + XZ = X(1 + Z) \qquad \text{distributive law}$$

And since $1 + Z = 1$ by rule 2,

$$X + XZ = X(1) \qquad \text{and} \qquad X(1) = X \qquad \text{by rule 6}$$

Therefore

$$X + XZ = X$$

It is worthwhile to try to prove rule 15 by using the proof by perfect induction at this point. Here is a proof of rule 16 that uses rules that precede it:

$$\begin{aligned} X(X + Y) &= XX + XY && \text{distributive law} \\ &= X + XY && (\text{since } XX = X) \\ &= X(1 + Y) && \text{where } 1 + Y = 1, \text{ rule 2} \\ &= X \end{aligned}$$

It is instructive to prove this rule also by perfect induction at this point.

SIMPLIFICATION OF EXPRESSIONS

3.9 The rules given may be used to simplify boolean expressions, just as the rules of normal algebra may be used to simplify expressions. Consider the expression

$$(X + Y)(X + \overline{Y})(\overline{X} + Z)$$

The first two terms consist of $X + Y$ and $X + \overline{Y}$; these terms may be multiplied and, since $X + X\overline{Y} + XY = X$ and $Y\overline{Y} = 0$, reduced to X.

The expression has been reduced now to $X(\overline{X} + Z)$, which may be expressed as $X\overline{X} + XZ$ (rule 14). And since $X\overline{X}$ is equal to 0, the entire expression $(X + Y)(X + \overline{Y})(\overline{X} + Z)$ may be reduced to XZ.

Another expression that may be simplified is $XYZ + X\overline{Y}Z + XY\overline{Z}$. First the three terms $XYZ + X\overline{Y}Z + XY\overline{Z}$ may be written $X(YZ + \overline{Y}Z + Y\overline{Z})$, by rule 14. Then, by using rule 14 again, $X[Y(Z + \overline{Z}) + \overline{Y}Z]$; and since $Z + \overline{Z}$ equals 1, we have $X(Y + \overline{Y}Z)$.

The expression $X(Y + \overline{Y}Z)$ may be further reduced to $X(Y + Z)$ by using rule 18. The final expression can be written in two ways: $X(Y + Z)$ or $XY + XZ$. The first expression is generally preferable if the equation is to be constructed as an electronic circuit, because it requires only one AND circuit and one OR circuit.

DE MORGAN'S THEOREMS

3.10 The following two rules are known as De Morgan's theorems:

$$\begin{aligned} \overline{(X + Y)} &= \overline{X} \cdot \overline{Y} \\ \overline{(X \cdot Y)} &= \overline{X} + \overline{Y} \end{aligned}$$

The complement of any boolean expression, or a part of any expression, may be found by means of these theorems. In these rules, two steps are used to form a complement:

1. The + symbols are replaced with · symbols and · symbols with + symbols.
2. Each of the terms in the expression is complemented.

The use of De Morgan's theorem may be demonstrated by finding the complement of the expression $X + YZ$. First, note that a multiplication sign has been omitted and the expression could be written $X + (Y \cdot Z)$. To complement this, the addition symbol is replaced with a multiplication symbol and the two terms are complemented, giving $\overline{X} \cdot (\overline{Y \cdot Z})$; then the remaining term is complemented, $\overline{X}(\overline{Y} + \overline{Z})$. The following equivalence has been found: $\overline{(X + YZ)} = \overline{X}(\overline{Y} + \overline{Z})$.

The complement of $\overline{W}X + Y\overline{Z}$ may be formed by two steps:

1. The addition symbol is changed.
2. The complement of each term is formed:

$$(\overline{\overline{W} \cdot X})(\overline{Y \cdot \overline{Z}})$$

This becomes $(W + \overline{X})(\overline{Y} + Z)$.

Since W and Z were already complemented, they become uncomplemented by the theorem $\overline{\overline{X}} = X$.

It is sometimes necessary to complement both sides of an equation. This may be done in the same way as before:

$$WX + YZ = 0$$

Complementing both sides gives

$$\overline{(WX + YZ)} = \overline{0}$$
$$(\overline{W} + \overline{X})(\overline{Y} + \overline{Z}) = 1$$

BASIC DUALITY OF BOOLEAN ALGEBRA

3.11 De Morgan's theorem expresses a basic duality which underlies all boolean algebra. The postulates and theorems which have been presented can all be divided into pairs. For example, $(X + Y) + Z = X + (Y + Z)$ is the *dual* of $(XY)Z = X(YZ)$, and $X + 0 = X$ is the dual of $X \cdot 1 = X$.

Often the rules or theorems are listed in an order which illustrates the duality of the algebra. In proving the theorems or rules of the algebra, it is then necessary to prove only one theorem, and the dual of the theorem follows necessarily. For instance, if you prove that $X + XY = X$, you can immediately add the theorem $X(X + Y) = X$ to the list of theorems as the dual of the first expression.[8] In effect, all boolean algebra is predicated on this two-for-one basis.

---

[8]When the first expression, $X + XY = X$, has been complemented, $\overline{X}(\overline{X} + \overline{Y}) = \overline{X}$ is obtained. Then uncomplemented variables may be substituted on both sides of the equation without changing the basic equivalence of the expression.

TABLE 3.13

INPUTS		OUTPUT
X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	1

DERIVATION OF A BOOLEAN EXPRESSION

3.12 When designing a logical circuit, the logical designer works from two sets of known values: (1) the various states which the inputs to the logical network can take and (2) the desired outputs for each input condition. The logical expression is derived from these sets of values.

Consider a specific problem. A logical network has two inputs X and Y and an output Z. The relationship between inputs and outputs is to be as follows:

1 When both X and Y are 0s, the output Z is to be 1.

2 When X is 0 and Y is 1, the output Z is to be 0.

3 When X is 1 and Y is 0, the output Z is to be 1.

4 When X is 1 and Y is 1, the output Z is to be 1.

These relations may be expressed in tabular form, as shown in Table 3.13.

It is now necessary to add another column to the table. This column will consist of a list of *product terms* obtained from the values of the input variables. The new column will contain each of the input variables listed in each row of the table, with the letter representing the respective input complemented when the input value for this variable is 0 and not complemented when the input value is 1. The terms obtained in this manner are designated as product terms. With two input variables X and Y, each row of the table will contain a product term consisting of X and Y, with X or Y complemented or not, depending on the input values for that row (see Table 3.14).

Whenever Z is equal to 1, the X and Y product term from the same row is removed and formed into a *sum-of-products* expression. Therefore, the product terms from the first, third, and fourth rows are selected. These are $\overline{X}\overline{Y}$, $X\overline{Y}$, and XY.

TABLE 3.14

INPUTS		OUTPUT	PRODUCT TERMS
X	Y	Z	
0	0	1	$\overline{X}\overline{Y}$
0	1	0	$\overline{X}Y$
1	0	1	$X\overline{Y}$
1	1	1	XY

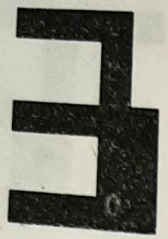


TABLE 3.15			
X	Y	$\overline{Y}$	$X + \overline{Y}$
0	0	1	1
0	1	0	0
1	0	1	1
1	1	0	1

There are now three terms, each the product of two variables. The logical sum of these products is equal to the expression desired. This type of expression is often referred to as a *canonical expansion* for the function. The complete expression in normal form is

$$\overline{X}\overline{Y} + X\overline{Y} + XY = Z$$

The left-hand side of the expression may be simplified as follows:

$$\begin{aligned} \overline{X}\overline{Y} + X\overline{Y} + XY &= Z \\ \overline{X}\overline{Y} + X(\overline{Y} + Y) &= Z \\ \overline{X}\overline{Y} + X &= Z \end{aligned}$$

and finally, by rule 18 in Table 3.10, $X + \overline{Y} = Z$.

The truth table may be constructed to check the function that has been derived (see Table 3.15). The last column of this table agrees with the last column of the truth table of the desired function, showing that the expressions are equivalent.

The expression $X + \overline{Y}$ may be constructed in one of two ways. If only the inputs X and Y are available, as might be the case if the inputs to the circuit were from another logical network or from certain types of storage devices, an inverter would be required to form $\overline{Y}$. Then the circuit would require an inverter plus an OR gate. Generally the complement of the Y input would be available, however, and only one OR gate would be required for the second way the expression would be constructed.

Another expression, with three inputs (designated X, Y, and Z), will be derived. Assume that the desired relationships between the inputs and the output have been determined, as shown in Table 3.16.

TABLE 3.16	
INPUTS	OUTPUT
When $X = 0$, $Y = 0$, $Z = 0$	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	1
1 1 1	0

DERIVATION OF A BOOLEAN EXPRESSION

TABLE 3.17

INPUTS			OUTPUT
X	Y	Z	A
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

1 A truth table is formed (see Table 3.17).

2 A column is added listing the inputs, X, Y, and Z according to their values in the input columns (see Table 3.18).

3 The product terms from each row in which the output is a 1 are collected ($\bar{X}\bar{Y}\bar{Z}$, $\bar{X}Y\bar{Z}$, $X\bar{Y}\bar{Z}$, and $XY\bar{Z}$), and the desired expression is the sum of these products ($\bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z}$). Therefore, the complete expression in standard form for the desired network is

$$\bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \dot{X}\bar{Y}\bar{Z} + XY\bar{Z} = A$$

This expression may be simplified as shown below:

$$\begin{aligned}
\bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z} &= A \\
\bar{X}(\bar{Y}\bar{Z} + Y\bar{Z}) + X(\bar{Y}\bar{Z} + Y\bar{Z}) &= A \\
\bar{X}\,[\bar{Z}(\bar{Y} + Y)] + X[\bar{Z}(\bar{Y} + Y)] &= A \\
\bar{X}\bar{Z} + X\bar{Z} &= A \\
\bar{Z} &= A
\end{aligned}$$

Thus the function can be performed by a single inverter connected to the Z input. Inspection of the truth table will indicate that the output A is always equal to the complement of the input variable Z.

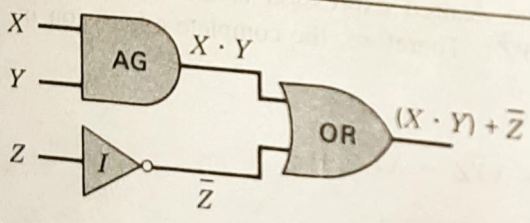
TABLE 3.18

INPUTS			OUTPUT	PRODUCT
X	Y	Z	A	TERMS
0	0	0	1	$\bar{X}\bar{Y}\bar{Z}$
0	0	1	0	$\bar{X}\bar{Y}Z$
0	1	0	1	$\bar{X}Y\bar{Z}$
0	1	1	0	$\bar{X}YZ$
1	0	0	1	$X\bar{Y}\bar{Z}$
1	0	1	0	$X\bar{Y}Z$
1	1	0	1	$XY\bar{Z}$
1	1	1	0	XYZ

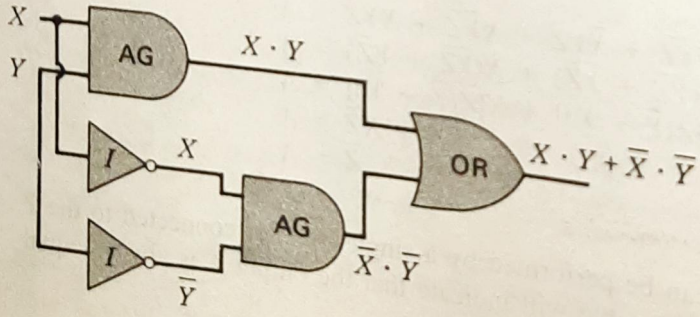
INTERCONNECTING GATES

3.13 The OR gates, AND gates, and inverters described in Secs. 3.3 and 3.4 can be interconnected to form *gating*, or *logic, networks*. (Those who study switching theory would also call these *combinational networks*.) The boolean algebra expression corresponding to a given gating network can be derived by systematically progressing from input to output on the gates. Figure 3.5(*a*) shows a gating network with three inputs X, Y, and Z and an output expression $(X \cdot Y) + \overline{Z}$. A network that forms $(X \cdot Y) + (\overline{X} \cdot \overline{Y})$ and another network that forms $(X + Y) \cdot (\overline{X} + \overline{Y})$ are shown in Fig. 3.5(*b*) and (*c*).

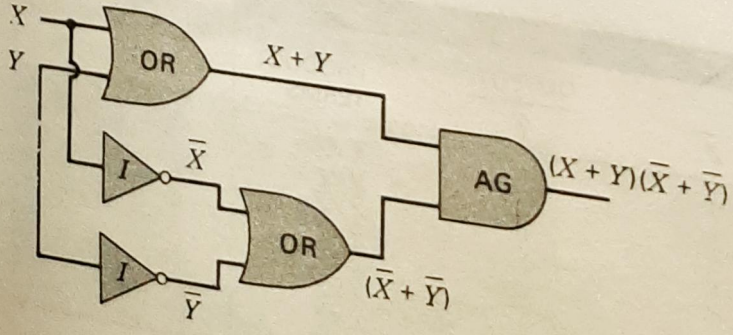
We can analyze the operation of these gating networks by using the boolean algebra expressions. For instance, in troubleshooting a computer, we can determine which gates have failed by examining the inputs to the gating network and the outputs and seeing whether the boolean operations are properly performed. The bookkeeping for computer circuitry is done by means of block diagrams, as in Fig.



(*a*)



(*b*)



(*c*)

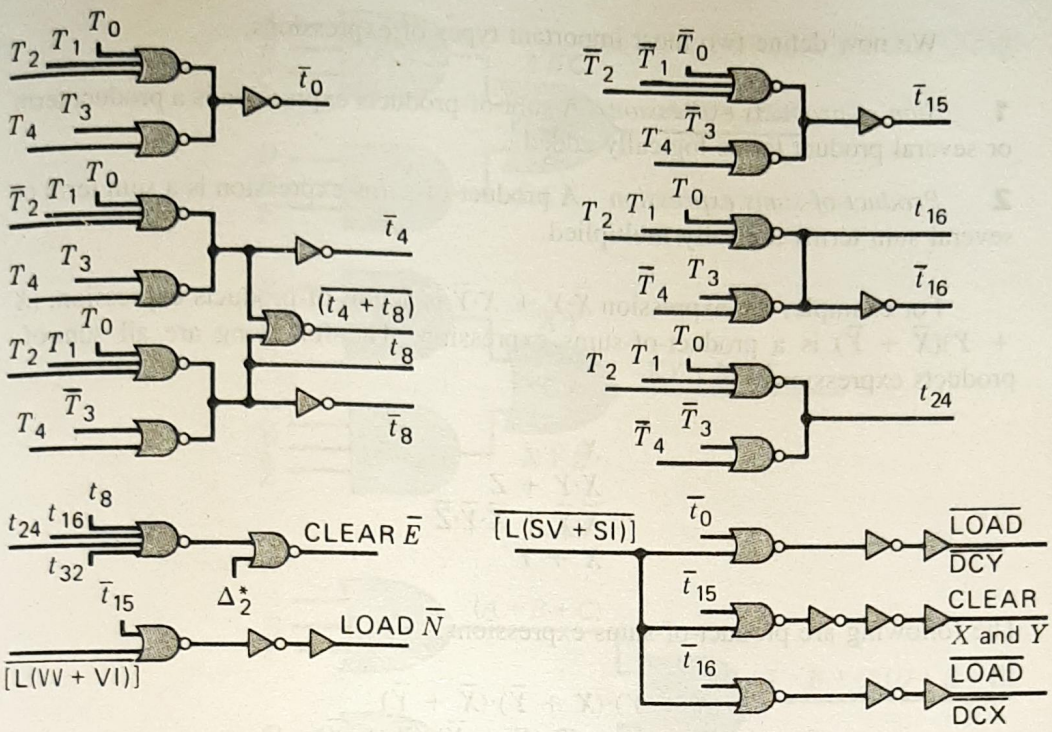


FIGURE 3.6

Block diagram from a computer.

3.6, which shows a typical print. The use of boolean algebra is spread completely throughout the computer industry.

SUM OF PRODUCTS AND PRODUCT OF SUMS

3.14 An important consideration in dealing with gating circuits and their algebraic counterparts is the *form* of the boolean algebra expression and the resulting form of the gating network. Certain types of boolean algebra expressions lead to gating networks which are more desirable from most implementation viewpoints. We now define the two most used and usable forms for boolean expressions.

First let us define terms:

1 *Product term* A product term is a single variable or the logical product of several variables. The variables may or may not be complemented.

2 *Sum term* A sum term is a single variable or the sum of several variables. The variables may or may not be complemented.

For example, the term $X \cdot Y \cdot Z$ is a product term; $X + Y$ is a sum term; X is both a product term and a sum term; $X + Y \cdot Z$ is neither a product term nor a sum term; $X + \overline{Y}$ is a sum term; $X \cdot \overline{Y} \cdot \overline{Z}$ is a product term; $\overline{Y}$ is a both a sum term and a product term. (*Comment:* Calling single variables sum terms and product terms is disagreeable but necessary. Since we must suffer with it, remember that some apples are red, round, and shiny, that is, more than one thing.)

We now define two most important types of expressions.

1 *Sum-of-products expression* A sum-of-products expression is a product term or several product terms logically added.

2 *Product-of-sums expression* A product-of-sums expression is a sum term or several sum terms logically multiplied.

For example, the expression $\overline{X}\cdot Y + X\cdot\overline{Y}$ is a sum-of-products expression; $(X + Y)(\overline{X} + \overline{Y})$ is a product-of-sums expression. The following are all sum-of-products expressions:

$$X$$
$$X\cdot Y + Z$$
$$\overline{X}\cdot\overline{Y} + \overline{X}\cdot\overline{Y}\cdot\overline{Z}$$
$$X + Y$$

The following are product-of-sums expressions:

$$(X + Y)\cdot(X + \overline{Y})\cdot(\overline{X} + \overline{Y})$$
$$(X + Y + Z)\cdot(X + \overline{Y})\cdot(\overline{X} + \overline{Y})$$
$$X + Z$$
$$\overline{X}$$
$$(X + Y)X$$

One prime reason for liking sum-of-products or product-of-sums expressions is their straightforward conversion to very nice gating networks. In their purest, nicest form they go into *two-level networks*, which are networks for which the longest path through which a signal must pass from input to output is two gates.

Note: In the following discussion we assume that when a variable X is available, its complement $\overline{X}$ is also available; that is, no inverters are required to complement inputs. This is quite important and quite realistic, since most signals come from flip-flops, which we study later, and which provide both an output and its complement.

Figure 3.7 shows several gating networks. Figure 3.7(*a*) shows sum-of-products networks, and Fig. 3.7(*b*) shows product-of-sums networks. The gating networks for sum-of-products expressions in "conventional" form—that is, expressions with at least two product terms and with at least two variables in each product term—go directly into an AND-to-OR gate network, while conventional product-of-sums expressions go directly into OR-to-AND gate networks, as shown in the figure.

DERIVATION OF PRODUCT-OF-SUMS EXPRESSIONS

3.15 The sequence of steps described in Sec. 3.12 derived a sum-of-products expression for a given circuit. Another technique, really a dual of the first, forms the required expression as a product-of-sums. The expression derived in this manner is made up, before simplification, of terms each consisting of sums of variables

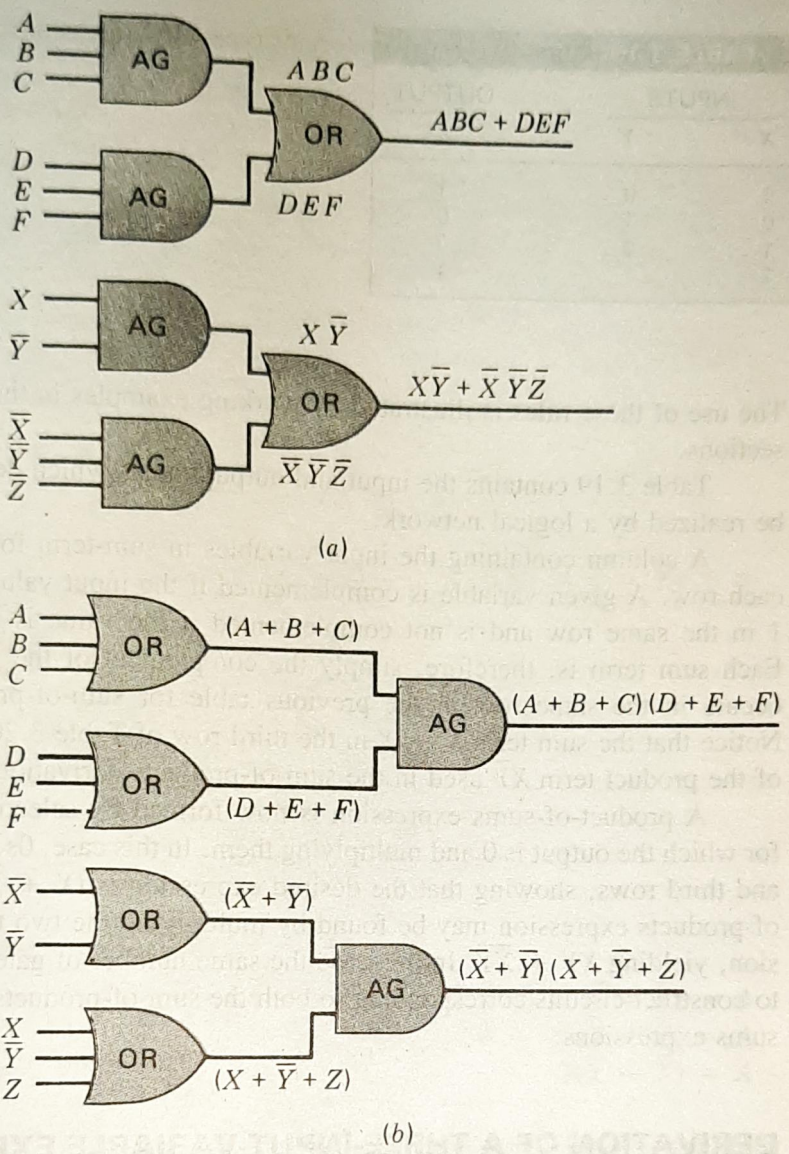


FIGURE 3.7

(*a*) AND-to-OR gate networks. (*b*) OR-to-AND gate networks.

such as $(X + Y + Z) \cdots$. The final expression is the product of these sum terms and has the form $(X + Y + Z)(X + Y + \overline{Z}) \cdots (\overline{X} + \overline{Y} + \overline{Z})$.

The method for arriving at the desired expression is as follows:

1 Construct a table of the input and output values.

2 Construct an additional column of sum terms containing complemented and uncomplemented variables (depending on the values in the input columns) for each row of the table. In each row of the table, a sum term is formed. However, in this case, if the input value for a given variable is 1, the variable will be complemented; and if 0, not complemented.

3 The desired expression is the product of the sum terms from the rows in which the output is 0.

TABLE 3.19

INPUTS		OUTPUT
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

The use of these rules is illustrated by working examples in this and the following sections.

Table 3.19 contains the input and output values which describe a function to be realized by a logical network.

A column containing the input variables in sum-term form is now added in each row. A given variable is complemented if the input value for the variable is 1 in the same row and is not complemented if the value is 0 (see Table 3.20). Each sum term is, therefore, simply the complement of the product term which occurs in the same row in the previous table for sum-of-products expressions. Notice that the sum term $\overline{X} + Y$ in the third row of Table 3.20 is the complement of the product term $X\overline{Y}$ used in the sum-of-products derivation.

A product-of-sums expression is now formed by selecting those sum terms for which the output is 0 and multiplying them. In this case, 0s appear in the second and third rows, showing that the desired expression is $(X + \overline{Y})(\overline{X} + Y)$. A sum-of-products expression may be found by multiplying the two terms of this expression, yielding $XY + \overline{X}\overline{Y}$. In this case the same number of gates would be required to construct circuits corresponding to both the sum-of-products and the product-of-sums expressions.

DERIVATION OF A THREE-INPUT-VARIABLE EXPRESSION

3.16 Consider Table 3.21, expressing an input-to-output relationship for which an expression is to be derived. Two columns will be added this time, one containing the sum-of-products terms and the other the product-of-sums terms (see Table 3.22). The two expressions may be written in the following way:

Sum-of-products:

$$\overline{X}Y\overline{Z} + \overline{X}YZ + XY\overline{Z} = A$$

TABLE 3.20

INPUTS		OUTPUT	SUM TERMS
X	Y	Z	
0	0	1	$X + Y$
0	1	0	$X + \overline{Y}$
1	0	0	$\overline{X} + Y$
1	1	1	$\overline{X} + \overline{Y}$

DERIVATION OF A THREE-INPUT-VARIABLE EXPRESSION

TABLE 3.21

INPUTS			OUTPUT
X	Y	Z	A
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Product-of-sums:

$$(X + Y + Z)(X + Y + \overline{Z})(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + \overline{Z}) = A$$

The two expressions may be simplified as shown:

SUM OF PRODUCTS

$$\begin{aligned}(\overline{X}Y\overline{Z}) + (\overline{X}YZ) + (XY\overline{Z}) &= A\\ \overline{X}(Y\overline{Z} + YZ) + (XY\overline{Z}) &= A\\ \overline{X}Y + XY\overline{Z} &= A\\ Y(\overline{X} + X\overline{Z}) &= A\\ \overline{X}Y + Y\overline{Z} &= A\end{aligned}$$

PRODUCT OF SUMS

$$\begin{aligned}(X + Y + Z)(X + Y + \overline{Z})(\overline{X} + Y + Z)(\overline{X} + Y + \overline{Z})(\overline{X} + \overline{Y} + \overline{Z}) &= A\\ (X + Y)(\overline{X} + Y)(\overline{X} + \overline{Z}) &= A\\ Y(\overline{X} + \overline{Z}) &= A\end{aligned}$$

The two final expressions clearly can be seen to be equivalent. Notice, however, that the shortest sum-of-products expression, which is $\overline{X}Y + Y\overline{Z}$, requires two AND gates and an OR gate (Fig. 3.8), while the shortest product-of-sums expression, $Y(\overline{X} + \overline{Z})$, requires only a single AND gate and a single OR gate. In some cases the minimal sum-of-products expression will require fewer logical elements to construct, and in other instances the construction of the minimal product

TABLE 3.22

INPUTS			OUTPUT	PRODUCT	SUM
X	Y	Z	A	TERMS	TERMS
0	0	0	0	$\overline{X}\,\overline{Y}\,\overline{Z}$	$X + Y + Z$
0	0	1	0	$\overline{X}\,\overline{Y}Z$	$X + Y + \overline{Z}$
0	1	0	1	$\overline{X}Y\overline{Z}$	$X + \overline{Y} + Z$
0	1	1	1	$\overline{X}YZ$	$X + \overline{Y} + \overline{Z}$
1	0	0	0	$X\overline{Y}\,\overline{Z}$	$\overline{X} + Y + Z$
1	0	1	0	$X\overline{Y}Z$	$\overline{X} + Y + \overline{Z}$
1	1	0	1	$XY\overline{Z}$	$\overline{X} + \overline{Y} + Z$
1	1	1	0	XYZ	$\overline{X} + \overline{Y} + \overline{Z}$

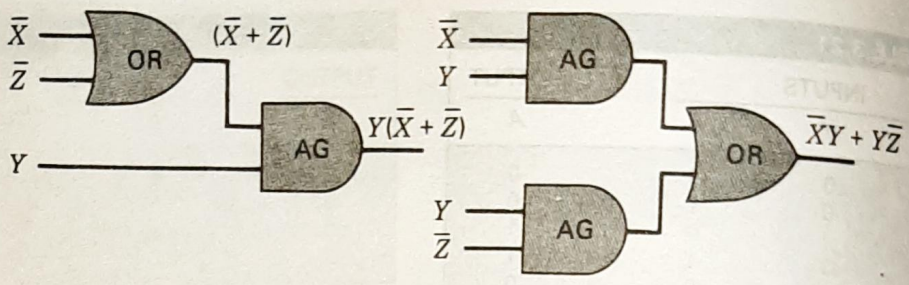


FIGURE 3.8

Networks for $Y(\overline{X} + \overline{Z})$ and $\overline{X}Y + Y\overline{Z}$.

of sums will require fewer elements. If the sole criterion is the number of logical elements, it is necessary to obtain both a minimal sum-of-products expression and a minimal product-of-sums expression to compare the two. It is possible first to derive the canonical expansion expression for the network to be designed in one of the forms—for instance, product of sums—to simplify the expression, and then to convert the simplified expression to the other form, by using the distributive laws. Then any additional simplification which is required can be performed. In this way, minimal expressions in each form may be obtained without deriving both canonical expansions, although this may be desirable.

The simplification techniques which have been described are algebraic and depend on judicious use of the theorems that have been presented. The problem of simplifying boolean expressions so that the shortest expression is always found is quite complex. However, it is possible, by means of the repeated use of certain algorithms, to derive minimal sum-of-products and product-of-sums expressions. We examine this problem in following sections.

NAND GATES AND NOR GATES

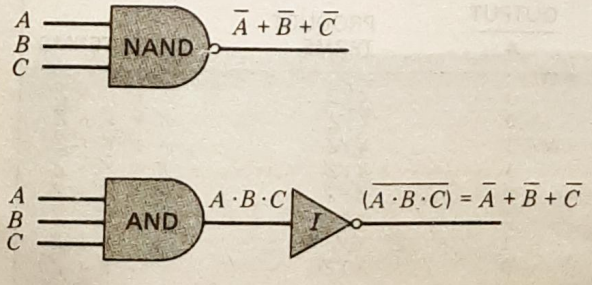
3.17 Two other types of gates, NAND gates and NOR gates, are often used in computers. It is fortunate that the boolean algebra which has been described can be easily used to analyze the operation of these gates.

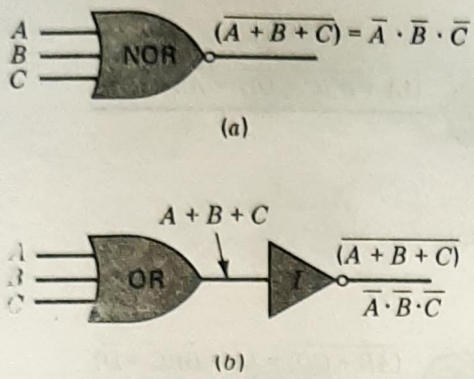
A NAND gate is shown in Fig. 3.9. The inputs are A, B, and C, and the output from the gate is written $\overline{A} + \overline{B} + \overline{C}$. The output will be a 1 if A is a 0 or B is a 0 or C is a 0, and the output will be a 0 only if A and B and C are all 1s.

The operation of the gate can be analyzed using the equivalent block diagram circuit shown in Fig. 3.9, which has an AND gate followed by an inverter. If the

FIGURE 3.9

NAND gate.





INPUT A	B	C	OUTPUT
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

FIGURE 3.10

(*a*) Block diagram symbol for a NOR gate. (*b*) OR gate and inverter equivalent circuit to NOR gate.

inputs are A, B, and C, the output of the AND gate will be $A \cdot B \cdot C$, and the complement of this is $(\overline{A \cdot B \cdot C}) = \overline{A} + \overline{B} + \overline{C}$, as shown in the figure.

The NOR gate can be analyzed in a similar manner. Figure 3.10 shows the NOR gate block diagram symbol with inputs, A, B, C and output $\overline{A}\,\overline{B}\,\overline{C}$. This shows the NOR gate's output will be a 1 only when all three inputs are 0s. If any input represents a 1, the output of a NOR gate will be a 0.

Below the NOR gate block diagram symbol in Fig. 3.10 is an equivalent circuit showing an OR gate and an inverter.[9] The inputs A, B, and C are ORed by the OR gate, giving $A + B + C$, which is complemented by the inverter, yielding $(\overline{A + B + C}) = \overline{A}\,\overline{B}\,\overline{C}$.

Multiple-input NAND gates can be analyzed similarly. A four-input NAND gate with inputs, A, B, C, and D has an output $\overline{A} + \overline{B} + \overline{C} + \overline{D}$, which says that the output will be a 1 if any one of the inputs is a 0 and will be a 0 only when all four inputs are 1s.

Similar reasoning will show that the output of a four-input NOR gate with inputs A, B, C, and D can be represented by the boolean algebra expression $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, which will be equal to 1 only when A, B, C, and D are all 0s.

If one of the two input lines to a two-input NAND gate contained the input $A + B$ and the other contained $C + D$, as shown in Fig. 3.11(*a*), the output from the NAND gate would be

$$\overline{[(A + B)(C + D)]} = \overline{A}\,\overline{B} + \overline{C}\,\overline{D}$$

We can show this by noting that the NAND gate first ANDs the inputs (in this case $A + B$ and $C + D$) and then complements this.

If one of the input lines to a two-input NOR gate contained the signal $A \cdot B$ and the other input line contained the signal $C \cdot D$, the output from the NOR gate would be $(\overline{A \cdot B + C \cdot D}) = (\overline{A} + \overline{B})(\overline{C} + \overline{D})$, as shown in Fig. 3.11(*b*).

[9] The "bubble," or small circle, on the output of the NAND and NOR gates represents complementation. The NAND can be seen to be an AND symbol followed by a complementer, and the NOR can be analyzed similarly.

BOOLEAN ALGEBRA AND GATE NETWORKS

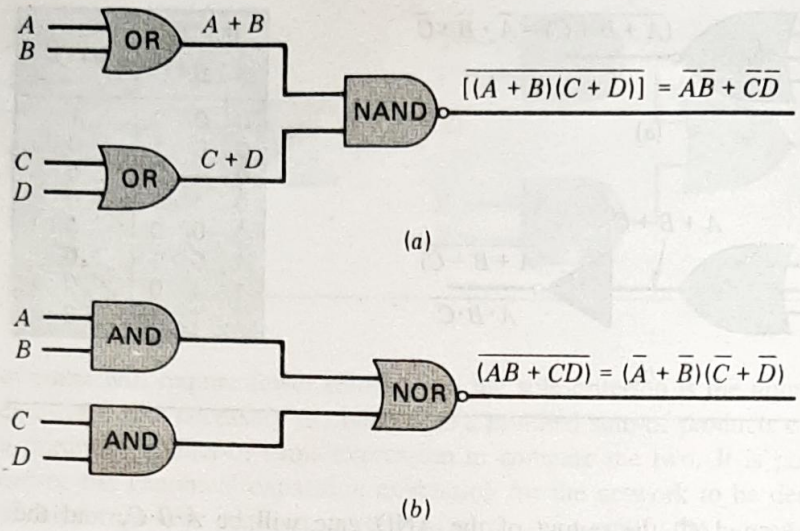


FIGURE 3.11

Two types of gating networks. (*a*) OR-to-NAND gate network. (*b*) AND-to-NOR gate network.

Notice that we can make an AND gate from two NAND gates, using the trick shown in Fig. 3.12, and a two-input OR gate from three NAND gates, as is also shown in the figure. A set of NAND gates can thus be used to make any combinational network by substituting the block diagrams shown in Fig. 3.12 for the AND and OR blocks. (Complementation of a variable, when needed, can be obtained from a single NAND gate by connecting the variable to all inputs.)

The NOR gate also can be used to form any boolean function which is desired, and the fundamental tricks are shown in Fig. 3.13.

Actually, it is not necessary to use the boxes shown in Figs. 3.12 and 3.13 to replace AND and OR gates singly, for a two-level NAND gate network yields the same function as a two-level AND-to-OR gate network, and a two-level NOR gate network yields the same function as a two-level OR-to-AND gate network. This is shown in Fig. 3.14. Compare the output of the NAND gate network with that in Fig. 3.7, for example. In Secs. 3.21 and 3.22 design procedures for NAND and NOR gate networks are given.

MAP METHOD FOR SIMPLIFYING EXPRESSIONS

***3.18**[10] We have examined the derivation of a boolean algebra expression for a given function by using a table of combinations to list desired function values. To derive a sum-of-products expression for the function, a set of product terms was listed, and those terms for which the function was to have a value 1 were selected and logically added to form the desired expression.

The table of combinations provides a nice, natural way to list all values of a boolean function. There are several other ways to represent or list function values, and the use of certain kinds of maps, which we will examine, also permits minimization of the expression formed in a nice graphic way.

The particular type of map we use is called the *Karnaugh map* after its originator.[11] Figure 3.15 shows the layouts for Karnaugh maps of two to four

[10]Sections with asterisks can be omitted in a first reading without the loss of continuity.

[11]Similar maps are sometimes called *Veitch diagrams*.

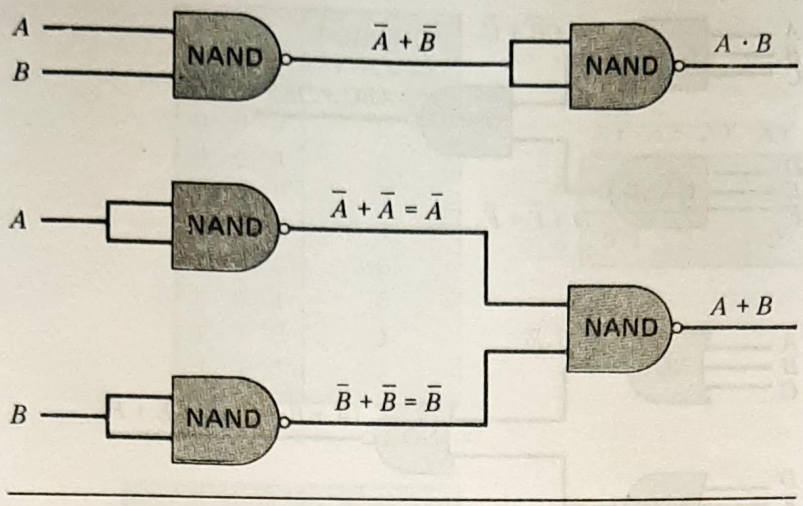


FIGURE 3.12

AND or OR operations from NAND gates.

variables. The diagram in each case lists the 2^n different product terms which can be formed in exactly n variables, each in a different square. For a function of n variables, a product term in exactly these n variables is called a *minterm*. Thus for three variables X, Y, and Z there are 2^3, or 8, different *minterms*, which are $\bar{X}\bar{Y}\bar{Z}$, $\bar{X}\bar{Y}Z$, $\bar{X}Y\bar{Z}$, $\bar{X}YZ$, $X\bar{Y}\bar{Z}$, $X\bar{Y}Z$, $XY\bar{Z}$, and XYZ. For four variables there are 2^4, or 16, terms; for five variables there are 32 terms; etc. As a result, a map of n variables will have 2^n squares, each representing a single minterm. The minterm in each box, or cell, of the map is the product of the variables listed at the abscissa and ordinate of the cell. Thus $\bar{X}YZ$ is at the intersection of $\bar{X}Y$ and Z.

Given a Karnaugh map form, the map is filled in by placing 1s in the squares, or cells, for each term which leads to a 1 output.

As an example, consider a function of three variables for which the following input values are to be 1:

$$X = 0, Y = 1, Z = 0$$
$$X = 0, Y = 1, Z = 1$$
$$X = 1, Y = 1, Z = 0$$
$$X = 1, Y = 1, Z = 1$$

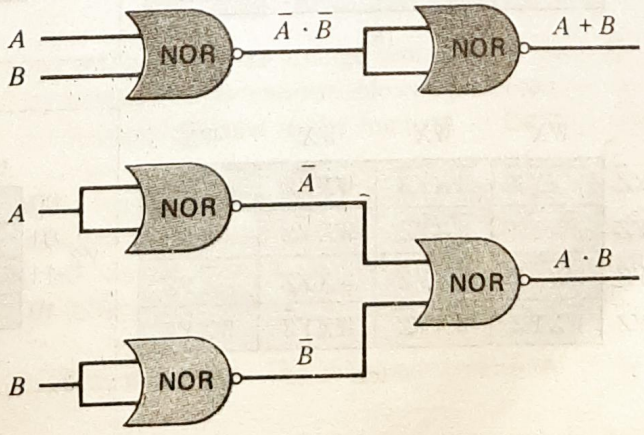
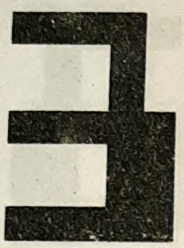


FIGURE 3.13

AND and OR gates from NOR gates.



BOOLEAN ALGEBRA AND GATE NETWORKS

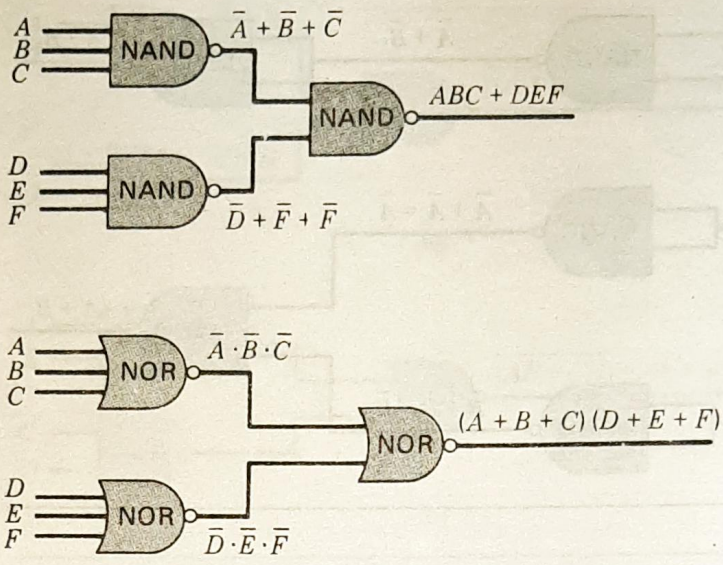


FIGURE 3.14

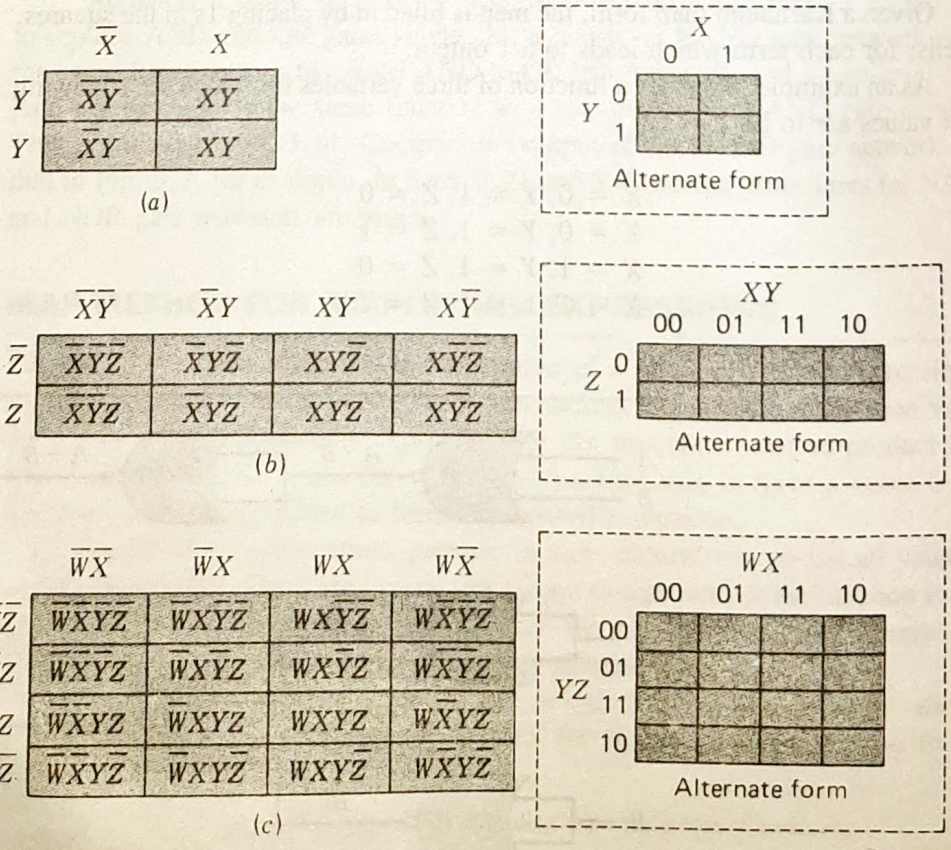
NAND and NOR gates in two-level networks.

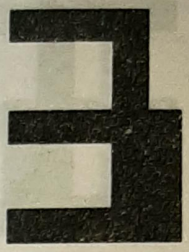
This function is shown in Fig. 3.16(*a*) in both table-of-combinations and Karnaugh map form. Another function of four variables is shown in Fig. 3.16(*b*).

As a means for displaying the values of a function, the Karnaugh map is convenient and provides some "feeling" for the function because of its graphic

FIGURE 3.15

Karnaugh maps for (*a*) two, (*b*) three, and (*c*) four variables.





MAP METHOD FOR SIMPLIFYING EXPRESSIONS

X	Y	Z	FUNCTION VALUES
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(*a*)

	$\overline{X}\,\overline{Y}$	$\overline{X}Y$	XY	$X\overline{Y}$
$\overline{Z}$	0	1	1	0
Z	0	1	1	0

W	X	Y	Z	FUNCTION VALUES
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*b*)

	$\overline{W}\,\overline{X}$	$\overline{W}X$	WX	$W\overline{X}$
$\overline{Y}\,\overline{Z}$	1	0	0	0
$\overline{Y}Z$	1	1	0	1
YZ	0	0	1	0
$Y\overline{Z}$	0	1	0	1

FIGURE 3.16

Two Karnaugh maps. (*a*) Map of boolean expression $\overline{X}Y\overline{Z} + \overline{X}YZ + XY\overline{Z} + XYZ$. (*b*) Map of four-variable function.

presentation. Its chief use, however, is due to the arrangement of cells. Each cell differs from its adjacent cell by having exactly one variable complemented in the minterm in one cell which is not complemented in the minterm in the adjacent cell.

As an example, consider the four-variable map in Fig. 3.16 and the minterm $\overline{W}X\overline{Y}Z$. There are four cells adjacent to the cell containing $\overline{W}X\overline{Y}Z$. These contain (1) $WX\overline{Y}Z$, which differs in the variable W; (2) $\overline{W}\,\overline{X}\,\overline{Y}Z$, which differs from $\overline{W}X\overline{Y}Z$ in X; (3) $\overline{W}XYZ$, which differs from $\overline{W}X\overline{Y}Z$ in Y; and (4) $\overline{W}X\overline{Y}\,\overline{Z}$, which differs from $\overline{W}X\overline{Y}Z$ in Z.

One trick should be noted at this point. The maps are considered to be

"rolled," or continuous, so that top and bottom edges or left and right side edges are touching. For the three-variable map, consider the left side edge and the right side edge to be touching, so that the map is considered to be rolled like a hoop horizontally on the page. This places the cell containing $\bar{X}\bar{Y}Z$ next to $X\bar{Y}\bar{Z}$, as well as to $\bar{X}\bar{Y}Z$ and $\bar{X}Y\bar{Z}$. Also, for this map it places $\bar{X}\bar{Y}Z$ next to $X\bar{Y}Z$, which touches because of the rolling, as well as to $\bar{X}YZ$ and $\bar{X}\bar{Y}\bar{Z}$.

For the four-variable map, the map is rolled so that the top edge touches the bottom edge, and the left side touches the right side. The touching of top and bottom places $\bar{W}X\bar{Y}\bar{Z}$ next to $\bar{W}XY\bar{Z}$, and the left side to the right side edges touching places $W\bar{X}YZ$ next to $\bar{W}\bar{X}YZ$.

A good rule to remember is that there are two minterms adjacent to a given minterm in a two-variable map; there are three minterms next to a given minterm in a three-variable map; there are four minterms next to a given minterm in a four-variable map; etc.

SUBCUBES AND COVERING

3.19 A *subcube* is a set of exactly 2^m adjacent cells containing 1s. For $m = 0$ the subcube consists of a single cell (and thus of a single minterm). For $m = 1$ a subcube consists of two adjacent cells; for instance, the cells containing $\bar{X}\bar{Y}Z$ and $\bar{X}YZ$ form a subcube, as shown in Fig. 3.17(*a*), as do $X\bar{Y}\bar{Z}$ and $\bar{X}\bar{Y}\bar{Z}$ (since the map is rolled).

For $m = 2$ a subcube has four adjacent cells, and several such subcubes are shown in Fig. 3.17(*c*). Notice that here we have omitted 0s for clarity and filled in only the 1s for the function. This policy will be continued.

Finally, subcubes containing eight cells (for $m = 3$) are shown in Fig. 3.17(*d*).

(It is sometimes convenient to call a subcube containing two cells a 2 cube, a subcube of four cells a 4 cube, a subcube of eight cells an 8 cube, etc., and this is done often.)

To demonstrate the use of maps and subcubes in minimizing boolean algebra expressions, we need to examine a rule of boolean algebra:

$$AX + A\bar{X} = A$$

In the above equation, A can stand for more than one variable. For instance, let $A = WY$; then we have

$$(WY)X + (WY)\bar{X} = WY$$

Or let $A = W\bar{Y}\bar{Z}$; then we have

$$W\bar{Y}\bar{Z}X + W\bar{Y}\bar{Z}X = W\bar{Y}\bar{Z}$$

The basic rule can be proved by factoring

$$AX + A\bar{X} = A(X + \bar{X})$$

SUBCUBES AND COVERING

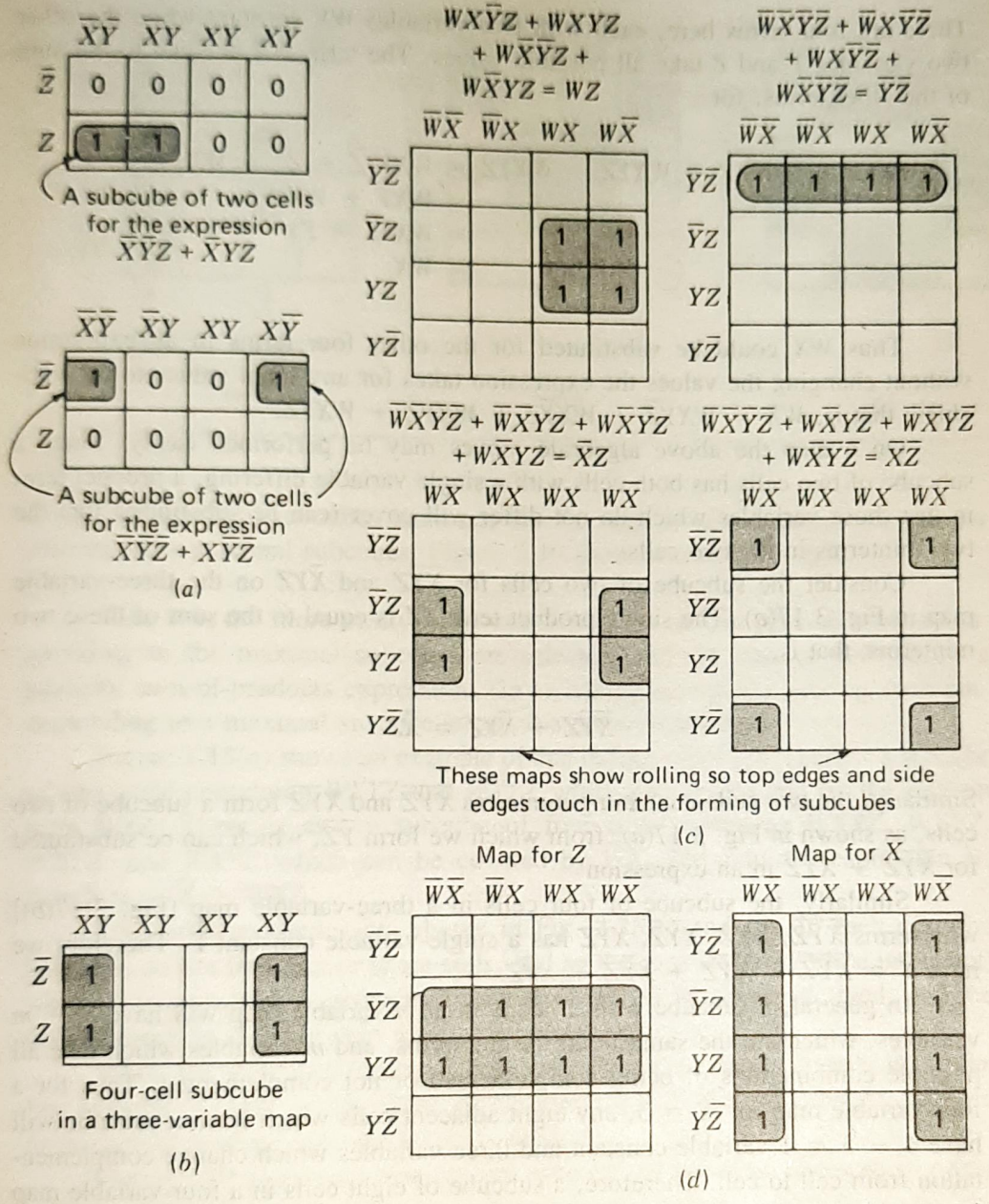


FIGURE 3.17

Subcubes with two, four, and eight cells. Blank cells are assumed to contain 0s.

Then since $X + \bar{X} = 1$, we have

$$AX + A\bar{X} = A(X + \bar{X}) = A \cdot 1 = A$$

Each of the examples given can be checked similarly; for instance,

$$W\bar{Y}\bar{Z}\bar{X} + W\bar{Y}\bar{Z}X = W\bar{Y}\bar{Z}(\bar{X} + X) = W\bar{Y}\bar{Z} \cdot 1 = W\bar{Y}\bar{Z}$$

This rule can be extended. Consider

$$WX\bar{Y}\bar{Z} + WX\bar{Y}Z + WXY\bar{Z} + WXYZ$$

There are four terms here, each with two variables WX constant while the other two variables Y and Z take all possible values. The term WX is equal to the sum of the other terms, for

$$\begin{aligned} W\bar{X}\bar{Y}\bar{Z} + W\bar{X}\bar{Y}Z + W\bar{X}Y\bar{Z} + W\bar{X}YZ &= W\bar{X}\bar{Y}(\bar{Z} + Z) + W\bar{X}Y(Z + \bar{Z}) \\ &= W\bar{X}\bar{Y} + W\bar{X}Y \\ &= W\bar{X}(\bar{Y} + Y) \\ &= W\bar{X} \end{aligned}$$

Thus WX could be substituted for the other four terms in an expression without changing the values the expression takes for any input values to the variables, that is, $WX = W\bar{X}\bar{Y}\bar{Z} + W\bar{X}\bar{Y}Z + W\bar{X}Y\bar{Z} + W\bar{X}YZ$.

On a map the above algebraic moves may be performed easily. Since a subcube of two cells has both cells with a single variable differing, a product term in just those variables which do not differ will cover (can be substituted for) the two minterms in the two cells.

Consider the subcube of two cells for $\bar{X}\bar{Y}Z$ and $\bar{X}YZ$ on the three-variable map in Fig. 3.17(a). The single product term $\bar{X}Z$ is equal to the sum of these two minterms; that is,

$$\bar{X}\bar{Y}Z + \bar{X}YZ = \bar{X}Z$$

Similarly, the two cells containing minterms $\bar{X}\bar{Y}\bar{Z}$ and $X\bar{Y}\bar{Z}$ form a subcube of two cells, as shown in Fig. 3.17(a), from which we form $\bar{Y}\bar{Z}$, which can be substituted for $\bar{X}\bar{Y}\bar{Z} + X\bar{Y}\bar{Z}$ in an expression.

Similarly, the subcube of four cells in a three-variable map [Fig. 3.17(b)] with terms $\bar{X}\bar{Y}\bar{Z}$, $\bar{X}\bar{Y}Z$, $X\bar{Y}\bar{Z}$, $X\bar{Y}Z$ has a single-variable constant $\bar{Y}$. Therefore we have $\bar{Y} = \bar{X}\bar{Y}\bar{Z} + \bar{X}\bar{Y}Z + X\bar{Y}\bar{Z} + X\bar{Y}Z$.

In general, a subcube with 2^m cells in an n-variable map will have $n - m$ variables, which are the same in all the minterms, and m variables which take all possible combinations of being complemented or not complemented. Thus for a four-variable map for $m = 3$, any eight adjacent cells which form a subcube will have $4 - 3 = 1$ variable constant and three variables which change complementation from cell to cell. Therefore, a subcube of eight cells in a four-variable map can be used to determine a single variable which can be substituted for the sum of the minterm in all eight cells.

As an example, in Fig. 3.17(d) we find a subcube of eight cells with the minterms $\bar{W}\bar{X}\bar{Y}Z$, $\bar{W}\bar{X}YZ$, $\bar{W}X\bar{Y}Z$, $\bar{W}XYZ$, $W\bar{X}\bar{Y}Z$, $W\bar{X}YZ$, $WX\bar{Y}Z$, and $WXYZ$. The sum of these will be found to be equivalent to Z.

The set of minterms in an expression does not necessarily form a single subcube, however, and there are two cases to be dealt with. Call a *maximal subcube* the largest subcube that can be found around a given minterm. Then the two cases are as follows:

1 All maximal subcubes are nonintersecting; that is, no cell in a maximal subcube is a part of another maximal subcube. Several examples are shown in Fig. 3.18.

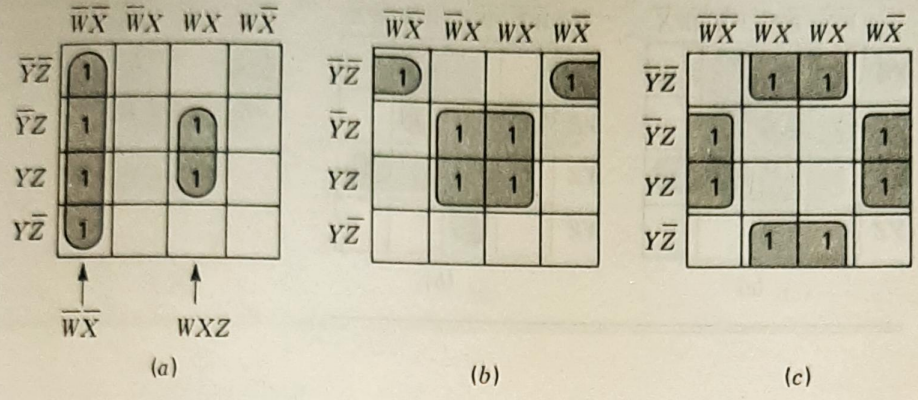


FIGURE 3.18

Maps with disjoint subcubes. (*a*) Map for $\overline{W}\overline{X} + WXZ$. (*b*) Map for $XZ + \overline{X}Y\overline{Z}$. (*c*) Map for $X\overline{Z} + \overline{X}Z$.

2 The maximal subcubes intersect; that is, cells in one maximal subcube are also in other maximal subcubes. Figure 3.19 shows examples of this.

Case 1 is the more easily dealt with. In this case, the product terms corresponding to the maximal subcubes are selected, and the sum of these forms a minimal sum-of-products expression. (In switching theory, the product term corresponding to a maximal subcube is called a *prime implicant*.)

Figure 3.18(*a*) shows an example of this in four variables. There is a subcube of two cells containing $WXYZ$ and $WX\overline{Y}Z$ which can be covered by the product term WXZ. There is also a subcube of four cells containing $\overline{W}X\overline{Y}\overline{Z}$, $\overline{W}XYZ$, $\overline{W}XY\overline{Z}$, and $\overline{W}\overline{X}YZ$ which can be covered by $\overline{W}\overline{X}$. The minimal expression is, therefore, $\overline{W}\overline{X} + WXZ$.

Two other examples are shown in Fig. 3.18(*b*) and (*c*). In each case the subcubes do not intersect or share cells, and so the product term (prime implicant) which corresponds to a given maximal subcube can be readily derived, and the sum of these for a given map forms the minimal expression.

When the subcubes intersect, the situation can be more complicated. The first principle to note is this: *Each cell containing a 1 (that is, each 1 cell) must be contained in some subcube which is selected.*

Figure 3.19(*a*) shows a map with an intersecting pair of subcubes plus another subcube. The minimal expression is, in this case, formed by simply adding the three product terms associated with the three maximal subcubes. Notice that a

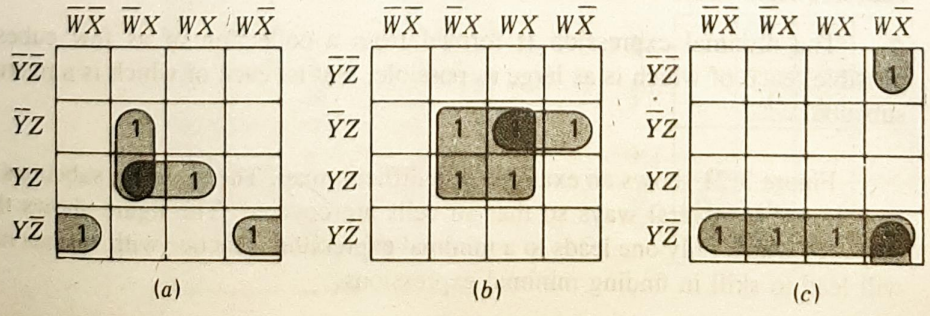
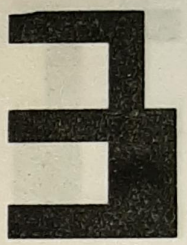


FIGURE 3.19

Intersecting subcubes. (*a*) $\overline{W}XZ + XYZ + \overline{X}Y\overline{Z}$. (*b*) $XZ + W\overline{Y}Z$. (*c*) $Y\overline{Z} + W\overline{X}\overline{Z}$.



the third row. These subcubes give the terms $\overline{Y}\overline{Z}$ and YZ; so the minimal sum-of-products expression is $\overline{Y}\overline{Z} + YZ$. Notice that if all the d's were made 0s, the solution would require more terms.

Another problem is worked in Fig. 3.23(b). For this problem the solution is $W\overline{Z} + W\overline{Y}$. Notice that two of the d's are made 0s. In effect, the d's are chosen so that they lead to the best solution.

DESIGN USING NAND GATES

3.21 Section 3.17 introduced NAND gates and showed the block diagram symbol for the NAND gate. NAND gates are widely used in modern computers, and an understanding of their use is invaluable.

Any NAND gate network can be analyzed by using boolean algebra, as previously indicated. Sometimes it is convenient, however, to substitute a functionally equivalent block diagram symbol for the conventional NAND gate symbol in order to analyze a block diagram. Figure 3.24 shows a gate symbol that consists of an OR gate symbol with "bubbles" (inverters) at each input. The two block diagram symbols in Fig. 3.24 perform the same function on inputs, as shown, for the NAND gate yields $\overline{A} + \overline{B} + \overline{C}$ on these inputs A, B, and C, as does the functionally equivalent gate.

As an example of the use of an equivalent symbol to simplify the analysis of a NAND gate network, we examine Fig. 3.25(a). This shows a two-level NAND-to-NAND gate network with inputs A, B, C, D, E, and F. Figure 3.25(b) shows the same network, but with the rightmost NAND gate replaced by the functionally equivalent block diagram symbol for a NAND gate previously shown in Fig. 3.24. Notice that the output function is the same for Fig. 3.25(b) as for Fig. 3.25(a), as it should be. Finally, an examination of the fact that the bubbles in Fig. 3.25(b) always occur in pairs, and so can be eliminated from the drawing from a functional viewpoint (since $\overline{\overline{X}} = X$), leads to Fig. 3.25($c$), which is an AND-to-OR gate network. This shows that the NAND-to-NAND gate network in Fig. 3.25(a) yields the same function as the AND-to-OR gate network in Fig. 3.25(c).

The substitution of the equivalent symbols followed by the removal of the "double bubbles" in Fig. 3.25 is a visual presentation of the following use of De Morgan's rule, which should be compared with the transformation in the figure:

$$\overline{\overline{(A \cdot B)} \cdot \overline{(C \cdot D)} \cdot \overline{(E \cdot F)}} = \overline{\overline{(A \cdot B)}} + \overline{\overline{(C \cdot D)}} + \overline{\overline{(E \cdot F)}} = A \cdot B + C \cdot D + E \cdot F$$

Study of the above will show that the same principle applies to NAND-to-NAND gates in general. As a further example, Fig. 3.26 shows another NAND-

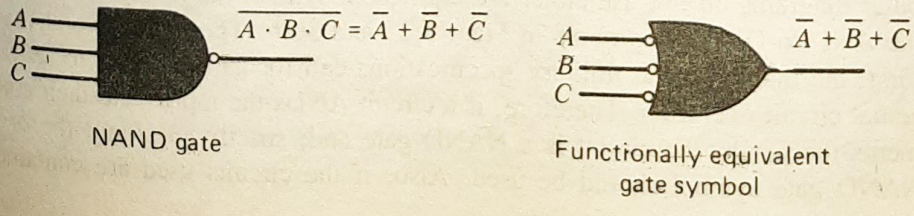
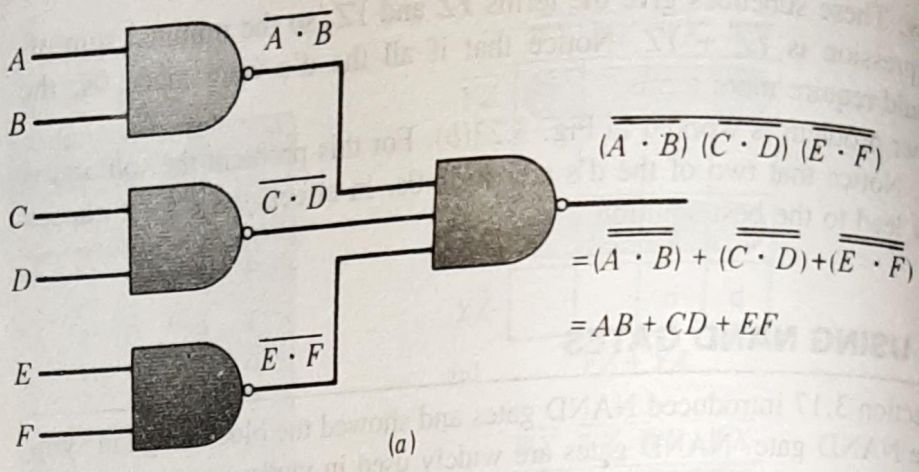
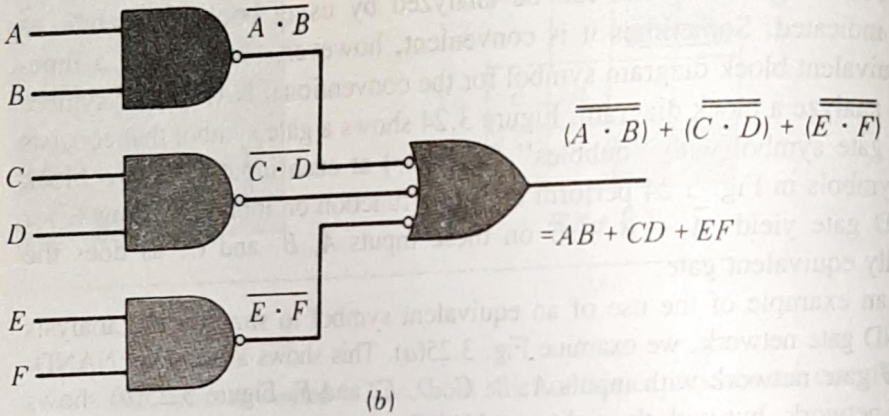


FIGURE 3.24

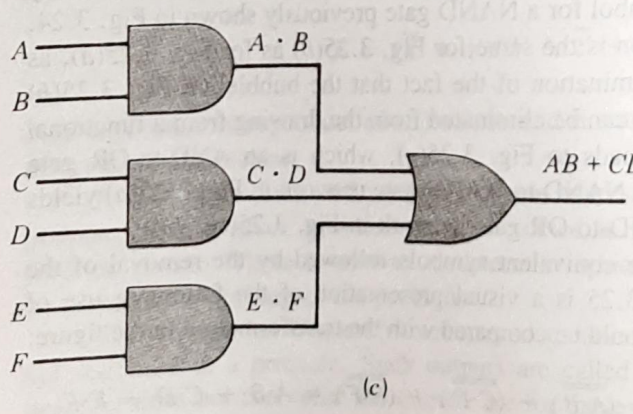
NAND gate and functionally equivalent



(a)



(b)



(c)

to-NAND gate network and the transformation to an AND-to-OR gate network. The algebraic moves equivalent to the symbology substitutions also are shown.

A question may arise as to why drawings of NAND gate networks in computer diagrams do not use either the equivalent symbol (as in Fig. 3.24) or even the AND-to-OR gate symbols in Figs. 3.25 and 3.26. There are several reasons. First, the industrial and military specifications call for gate symbols to reflect the actual circuit operation. Therefore, if a circuit ANDs the inputs and then complements the result, the circuit is a NAND gate and, strictly speaking, the original NAND gate symbol should be used. Also, if the circuits used are contained in

DESIGN USING NAND GATES

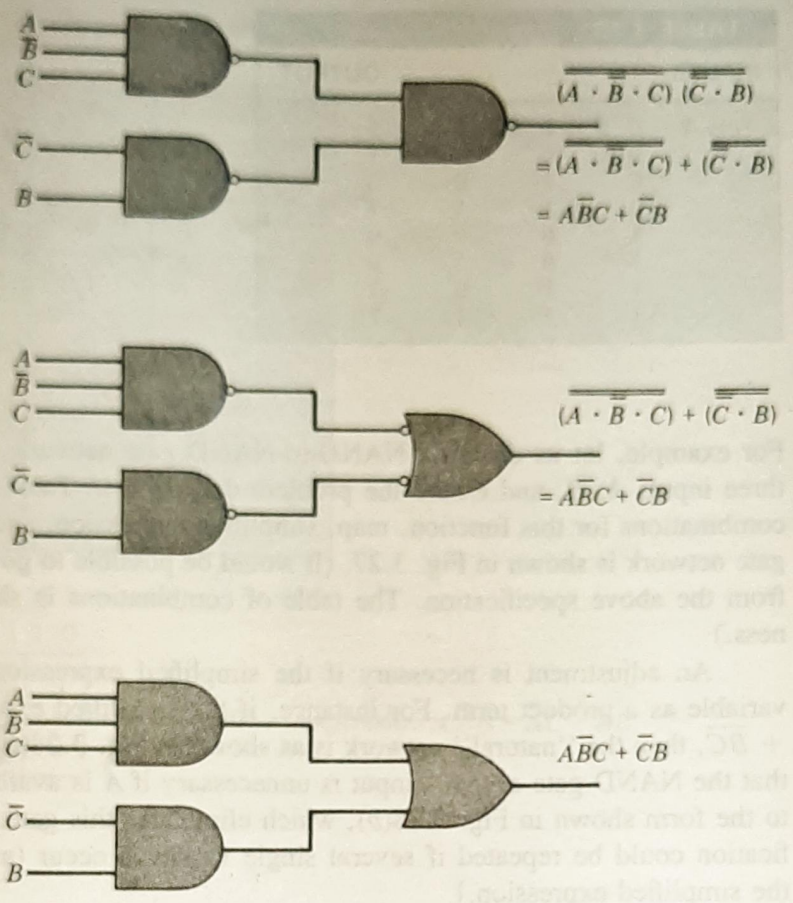


FIGURE 3.26

NAND-to-NAND and AND-to-OR gate transformation.

integrated-circuit packages and the computer drawing calls out the part number for the IC packages, an examination of the manufacturer's IC package drawings will show NAND gate symbols (if NAND gates are in the IC package). In the next chapter we show such packages and clarify this. In any case, substitution of symbols might easily lead to confusion, and it seems best to use the NAND gate symbol when NAND gates are used.

The above analysis of two-level NAND gate networks leads to a direct procedure for designing a NAND-to-NAND gate network.

DESIGN RULE

To design a two-level NAND-to-NAND gate network, use the table-of-combinations procedure for a sum-of-products expression. Simplify this sum-of-products expression by using maps, as has been shown. Finally, draw a NAND-to-NAND gate network in the two-level form, and write the same inputs as would have been used in an AND-to-OR gate network, except use NAND gates in place of the AND and OR gates.