Combinational Logic

4-1 INTRODUCTION

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

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

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

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

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

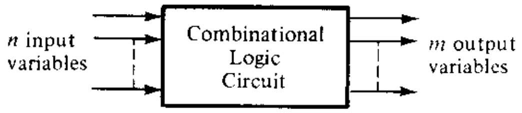


FIGURE 4-1
Block diagram of a combinational circuit

4-2 DESIGN PROCEDURE

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be easily obtained. The procedure involves the following steps:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.

3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

A truth table for a combinational circuit consists of input columns and output columns. The 1's and 0's in the input columns are obtained from the 2^n binary combinations available for n input variables. The binary values for the outputs are determined from examination of the stated problem. An output can be equal to either 0 or 1 for every valid input combination. However, the specifications may indicate that some input combinations will not occur. These combinations become don't-care conditions.

The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly into a truth table. Sometimes the designer must use intuition and experience to arrive at the correct interpretation. Word specifications are very seldom complete and exact. Any wrong interpretation that results in an incorrect truth table produces a combinational circuit that will not fulfill the stated requirements.

The output Boolean functions from the truth table are simplified by any available method, such as algebraic manipulation, the map method, or the tabulation procedure. Usually, there will be a variety of simplified expressions from which to choose. However, in any particular application, certain restrictions, limitations, and criteria will serve as a guide in the process of choosing a particular algebraic expression. A practical design method would have to consider such constraints as (1) minimum number of gates, (2) minimum number of inputs to a gate, (3) minimum propagation time of the signal through the circuit, (4) minimum number of interconnections, and (5) limitations of the driving capabilities of each gate. Since all these criteria cannot be satisfied simultaneously, and since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement as to what constitutes an acceptable simplification. In most cases, the simplification begins by satisfying an elementary objective, such as producing a simplified Boolean function in a standard form, and from that proceeds to meet any other performance criteria.

In practice, designers tend to go from the Boolean functions to a wiring list that shows the interconnections among various standard logic gates. In that case, the design need not go any further than the required simplified output Boolean functions. However, a logic diagram is helpful for visualizing the gate implementation of the expressions.

4-3 ADDERS

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely, $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 10$. The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half-adder*. One that performs the addition of three bits (two significant bits and a previous carry) is a *full-adder*. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder. The two adder circuits are the first combinational circuits we shall design.

Half-Adder

From the verbal explanation of a half-adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. It is necessary to specify two output variables because the result may consist of two binary digits. We arbitrarily assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.

Now that we have established the number and names of the input and output variables, we are ready to formulate a truth table to identify exactly the function of the half-adder. This truth table is

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

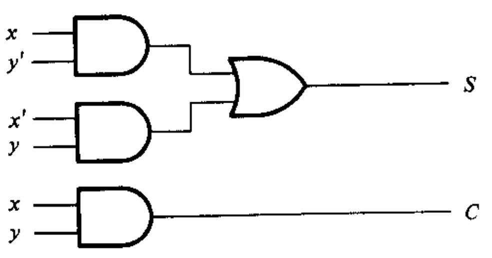
The carry output is 0 unless both inputs are 1. The S output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are

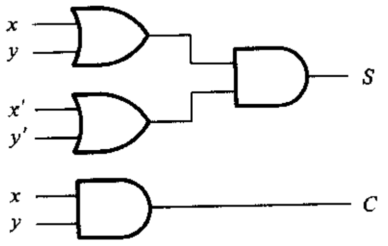
$$S = x'y + xy'$$

$$C = xy$$

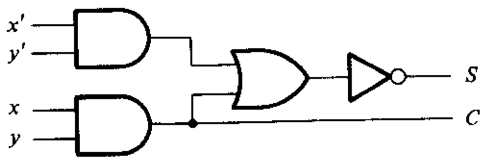
The logic diagram for this implementation is shown in Fig. 4-2(a), as are four other implementations for a half-adder. They all achieve the same result as far as the input–output behavior is concerned. They illustrate the flexibility available to the designer when implementing even a simple combinational logic function such as this.



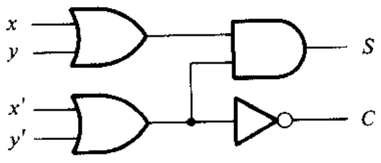
(a) $S = xy' + x'y$
$C = xy$



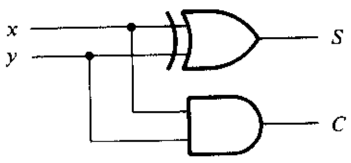
(b) $S = (x + y)(x' + y')$
$C = xy$



(c) $S = (C + x'y')'$
$C = xy$



(d) $S = (x + y)(x' + y')$
$C = (x' + y')'$



(e) $S = x \oplus y$
$C = xy$

FIGURE 4-2
Various implementations of a half-adder

Figure 4-2(a), as mentioned before, is the implementation of the half-adder in sum of products. Figure 4-2(b) shows the implementation in product of sums:

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

To obtain the implementation of Fig. 4-2(c), we note that S is the exclusive-OR of x and y. The complement of S is the equivalence of x and y (Section 2-6.):

$$S' = xy + x'y'$$

but $C = xy$, and, therefore, we have

$$S = (C + x'y')'$$

In Fig. 4-2(d), we use the product of sums implementation with C derived as follows:

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

The half-adder can be implemented with an exclusive-OR and an AND gate, as shown in Fig. 4-2(e). This form is used later to show that two half-adder circuits are needed to construct a full-adder circuit.

Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is

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

The eight rows under the input variables designate all possible combinations of 1's and 0's that these variables may have. The 1's and 0's for the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0's, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The input and output bits of the combinational circuit have different interpretations at various stages of the problem. Physically, the binary signals of the input wires are considered binary digits added arithmetically to form a two-digit sum at the output wires. On the other hand, the same binary values are considered variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. It is important to realize that two different interpretations are given to the values of the bits encountered in this circuit.

The input-output logical relationship of the full-adder circuit may be expressed in two Boolean functions, one for each output variable. Each output Boolean function re-

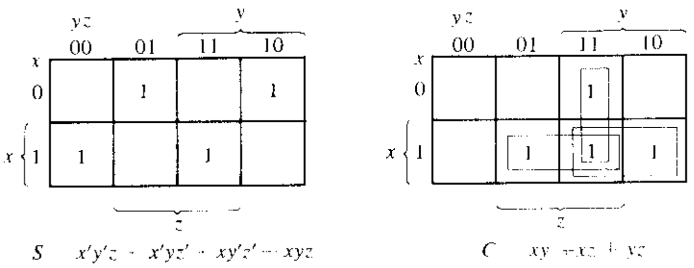


FIGURE 4-3
Maps for a full-adder

quires a unique map for its simplification. Each map must have eight squares, since each output is a function of three input variables. The maps of Fig. 4-3 are used for simplifying the two output functions. The 1's in the squares for the maps of S and C are determined directly from the truth table. The squares with 1's for the S output do not combine in adjacent squares to give a simplified expression in sum of products. The C output can be simplified to a six-literal expression. The logic diagram for the full-adder implemented in sum of products is shown in Fig. 4-4. This implementation uses the following Boolean expressions:

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

$$C = xy + xz + yz$$

Other configurations for a full-adder may be developed. The product of sums implementation requires the same number of gates as in Fig. 4-4, with the number of AND and OR gates interchanged. A full-adder can be implemented with two half-adders and

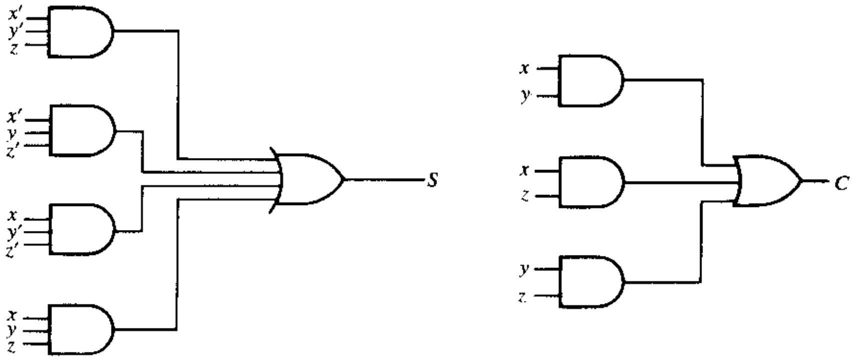


FIGURE 4-4
Implementation of a full-adder in sum of products

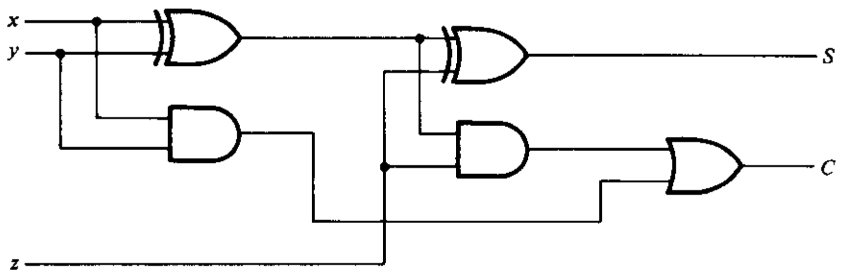


FIGURE 4-5
Implementation of a full-adder with two half-adders and an OR gate

one OR gate, as shown in Fig. 4-5. The S output from the second half-adder is the exclusive-OR of z and the output of the first half-adder, giving

$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

and the carry output is

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

4-4 SUBTRACTORS

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend (Section 1-5). By this method, the subtraction operation becomes an addition operation requiring full-adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. The fact that a 1 has been borrowed must be conveyed to the next higher pair of bits by means of a binary signal coming out (output) of a given stage and going into (input) the next higher stage. Just as there are half- and full-adders, there are half- and full-subtractors.

Half-Subtractor

A half-subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. Designate the minuend bit by x and the subtrahend bit by y. To perform $x - y$, we have to check the relative magnitudes of x and y. If $x \geqslant y$, we have three possibilities: $0 - 0 = 0$, $1 - 0 = 1$, and $1 - 1 = 0$. The result is called the *difference bit*. If $x < y$, we have $0 - 1$, and it is necessary to borrow a 1 from the next higher stage. The 1 borrowed from the next higher stage adds 2 to the minuend bit, just as in the decimal system a borrow adds 10 to a minuend digit. With the minuend equal to 2, the difference becomes $2 - 1 = 1$. The half-subtractor needs two outputs. One output generates the difference and will be designated by the symbol D. The second output, designated B for borrow, generates the binary signal that informs the next stage that a 1 has been borrowed. The truth table for the input–output relationships of a half-subtractor can now be derived as follows:

x	y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow B is a 0 as long as $x \geqslant y$. It is a 1 for $x = 0$ and $y = 1$. The D output is the result of the arithmetic operation $2B + x - y$.

The Boolean functions for the two outputs of the half-subtractor are derived directly from the truth table:

$$D = x'y + xy'$$

$$B = x'y$$

It is interesting to note that the logic for D is exactly the same as the logic for output S in the half-adder.

Full-Subtractor

A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 may have been borrowed by a lower significant stage. This circuit has three inputs and two outputs. The three inputs, x, y, and z, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and B, represent the difference and output borrow, respectively. The truth table for the circuit is

x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations of 1's and 0's that the binary variables may take. The 1's and 0's for the output variables are determined from the subtraction of $x - y - z$. The combinations having input borrow $z = 0$ reduce to the same four conditions of the half-adder. For $x = 0$, $y = 0$, and $z = 1$, we have to borrow a 1 from the next stage, which makes $B = 1$ and adds 2 to x. Since $2 - 0 - 1 = 1$, $D = 1$. For $x = 0$ and $yz = 11$, we need to borrow again, making $B = 1$ and $x = 2$. Since $2 - 1 - 1 = 0$, $D = 0$. For $x = 1$ and $yz = 01$, we have $x - y - z = 0$, which makes $B = 0$ and $D = 0$. Finally, for $x = 1$, $y = 1$, $z = 1$, we have to borrow 1, making $B = 1$ and $x = 3$, and $3 - 1 - 1 = 1$, making $D = 1$.

The simplified Boolean functions for the two outputs of the full-subtractor are derived in the maps of Fig. 4-6. The simplified sum of products output functions are

$$D = x'y'z + x'yz' + xy'z' + xyz$$
$$B = x'y + x'z + yz$$

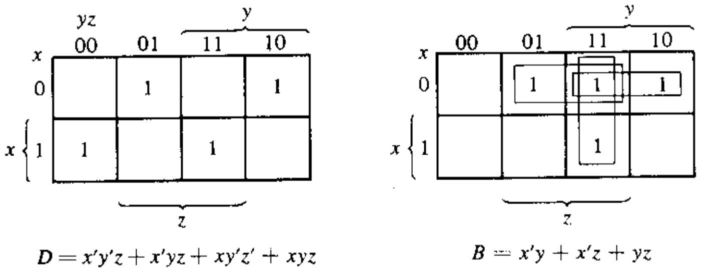


FIGURE 4-6
Maps for a full-subtractor

4-5 CODE CONVERSION

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. The design procedure of code converters will be illustrated by means of a specific example of conversion from the BCD to the excess-3 code.

The bit combinations for the BCD and excess-3 codes are listed in Table 1-2 (Section 1-7). Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. Let us designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y, and z. The truth table relating the input and output variables is shown in Table 4-1. The bit combinations for the inputs and their corresponding outputs are obtained directly from Table 1-2. We note that four binary variables may have 16 bit combinations, only 10 of which are listed in the truth table. The six bit combinations not listed for the *input* variables are don't-care combinations. Since they will never occur, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

The maps in Fig. 4-7 are drawn to obtain a simplified Boolean function for each output. Each of the four maps of Fig. 4-7 represents one of the four outputs of this circuit as a function of the four input variables. The 1's marked inside the squares are obtained

TABLE 4-1
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

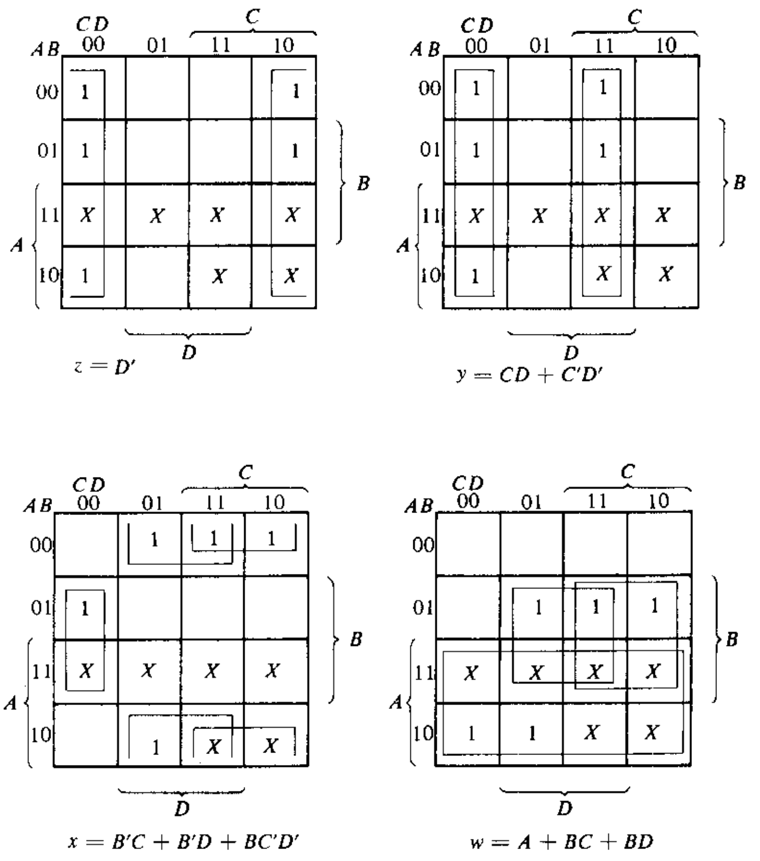


FIGURE 4-7
Maps for a BCD-to-excess-3-code converter

from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output z has five 1's; therefore, the map for z must have five 1's, each being in a square corresponding to the minterm that makes z equal to 1. The six don't-care combinations are marked by X's. One possible way to simplify the functions in sum of products is listed under the map of each variable.

A two-level logic diagram may be obtained directly from the Boolean expressions derived by the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. 4-7 may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown below, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates.

$$z = D'$$
$$y = CD + C'D' = CD + (C + D)'$$
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$
$$w = A + BC + BD = A + B(C + D)$$

The logic diagram that implements these expressions is shown in Fig. 4-8. In it we see that the OR gate whose output is $C + D$ has been used to implement partially each of three outputs.

Not counting input inverters, the implementation in sum of products requires seven AND gates and three OR gates. The implementation of Fig. 4-8 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first implementation will require inverters for variables B, C, and D, whereas the second implementation requires inverters for variables B and D.

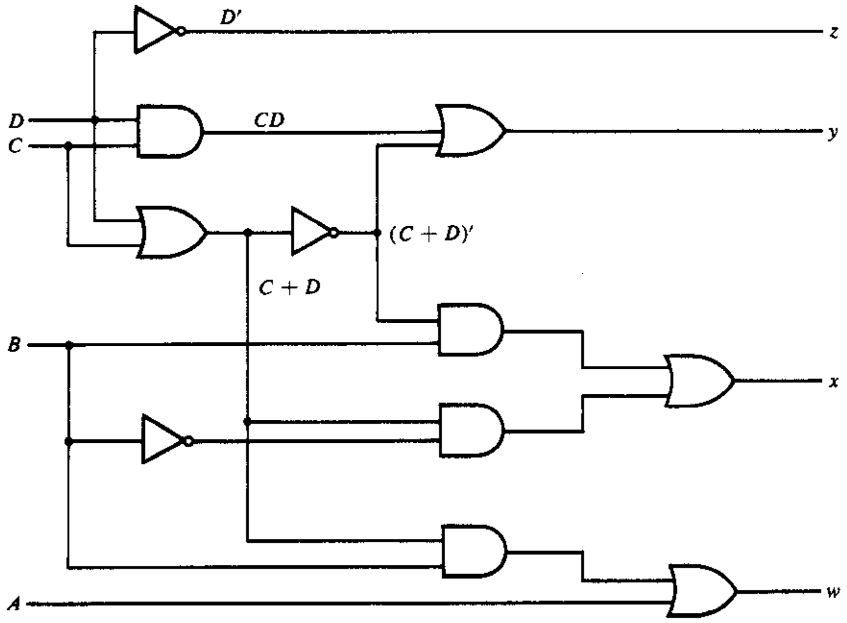


FIGURE 4-8
Logic diagram for a BCD-to-excess-3-code converter

4-6 ANALYSIS PROCEDURE

The design of a combinational circuit starts from the verbal specifications of a required function and culminates with a set of output Boolean functions or a logic diagram. The *analysis* of a combinational circuit is somewhat the reverse process. It starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation. If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function.

The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate. Feedback paths or memory ~~elements in a digital circuit define~~ a sequential circuit and must be analyzed according to procedures outlined in Chapter 6 or Chapter 9.

Once the logic diagram is verified as a combinational circuit, one can proceed to obtain the output Boolean functions and/or the truth table. If the circuit is accompanied by a verbal explanation of its function, then the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived truth table. The success of such investigation is enhanced if one has previous experience and familiarity with a wide variety of digital circuits. The ability to correlate a truth table with an information-processing task is an art one acquires with experience.

To obtain the output Boolean functions from a logic diagram, proceed as follows:

1. Label with arbitrary symbols all gate outputs that are a function of the input variables. Obtain the Boolean functions for each gate.
2. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables only.

Analysis of the combinational circuit in Fig. 4-9 illustrates the proposed procedure. We note that the circuit has three binary inputs, A, B, and C, and two binary outputs, F_1 and F_2. The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function of input variables only are F_2, T_1, and T_2. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

Next we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2' T_1$$

$$F_1 = T_3 + T_2$$

The output Boolean function F_2 just expressed is already given as a function of the in-

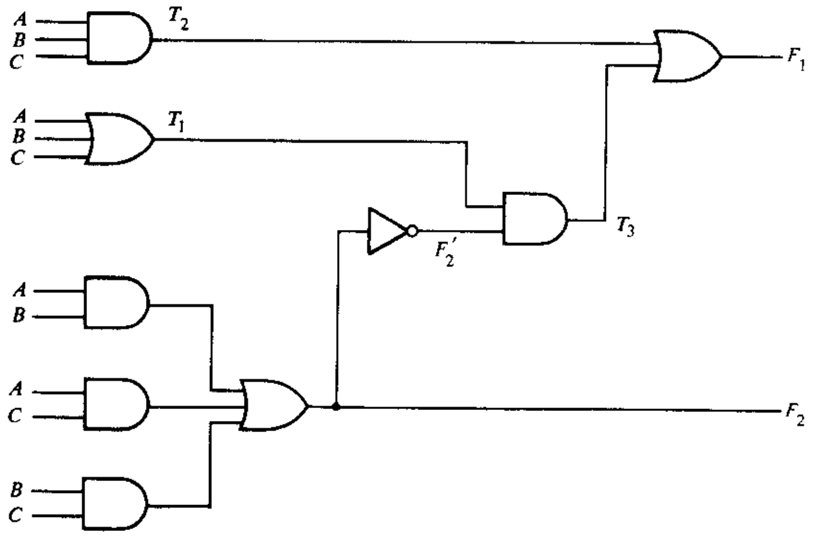


FIGURE 4-9
Logic diagram for analysis example

puts only. To obtain F_1 as a function of A, B, and C, form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F_2'T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If we want to pursue the investigation and determine the information-transformation task achieved by this circuit, we can derive the truth table directly from the Boolean functions and try to recognize a familiar operation. For this example, we note that the circuit is a full-adder, with F_1 being the sum output and F_2 the carry output. A, B, and C are the three inputs added arithmetically.

The derivation of the truth table for the circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, proceed as follows:

1. Determine the number of input variables to the circuit. For n inputs, form the 2^n possible input combinations of 1's and 0's by listing the binary numbers from 0 to $2^n - 1$.

2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

This process can be illustrated using the circuit of Fig. 4-9. In Table 4-2, we form the eight possible combinations for the three input variables. The truth table for F_2 is determined directly from the values of A, B, and C, with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F_2' is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T_3 are derived from T_1 and F_2': T_3 is equal to 1 when both T_1 and F_2' are equal to 1, and to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1. Inspection of the truth table combinations for A, B, C, F_1, and F_2 of Table 4-2 shows that it is identical to the truth table of the full-adder given in Section 4-3 for x, y, z, S, and C, respectively.

TABLE 4-2
Truth Table for the Logic Diagram of Fig. 4-9

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Consider now a combinational circuit that has don't-care input combinations. When such a circuit is designed, the don't-care combinations are marked by X's in the map and assigned an output of either 1 or 0, whichever is more convenient for the simplification of the output Boolean function. When a circuit with don't-care combinations is being analyzed, the situation is entirely different. Even though we assume that the don't-care input combinations will never occur, if any one of these combinations is applied to the inputs (intentionally or in error), a binary output will be present. The value of the output will depend on the choice for the X's taken during the design. Part of the analysis of such a circuit may involve the determination of the output values for the don't-care input combinations. As an example, consider the BCD-to-excess-3-code converter designed in Section 4-5. The outputs obtained when the six unused combinations of the BCD code are applied to the inputs are

Unused BCD Inputs				Outputs			
A	B	C	D	w	x	y	z
1	0	1	0	1	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	1	1	1	1
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	1
1	1	1	1	1	0	1	0

These outputs may be derived by means of the truth table analysis method as outlined in this section. In this particular case, the outputs may be obtained directly from the maps of Fig. 4-7. From inspection of the maps, we determine whether the X's in the corresponding minterm squares for each output have been included with the 1's or the 0's. For example, the square for minterm m_{10} (1010) has been included with the 1's for outputs w, x, and z, but not for y. Therefore, the outputs for m_{10} are $wxyz = 1101$, as listed in the previous table. We also note that the first three outputs in the table have no meaning in the excess-3 code, and the last three outputs correspond to decimal 5, 6, and 7, respectively. This coincidence is entirely a function of the choice for the X's taken during the design.

4-7 MULTILEVEL NAND CIRCUITS

Combinational circuits are more frequently constructed with NAND or NOR gates rather than AND and OR gates. NAND and NOR gates are more common from the hardware point of view because they are readily available in integrated-circuit form. Because of the prominence of NAND and NOR gates in the design of combinational circuits, it is important to be able to recognize the relationships that exist between circuits constructed with AND-OR gates and their equivalent NAND or NOR diagrams.

The implementation of two-level NAND and NOR logic diagrams was presented in Section 3-6. Here we consider the more general case of multilevel circuits. The procedure for obtaining NAND circuits is presented in this section, and for NOR circuits in the next section.

Universal Gate

The NAND gate is said to be a universal gate because any digital system can be implemented with it. Combinational circuits and sequential circuits as well can be constructed with this gate because the flip-flop circuit (the memory element most frequently used in sequential circuits) can be constructed from two NAND gates connected back to back, as shown in Section 6-2.

To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations AND, OR, and NOT can be implemented with

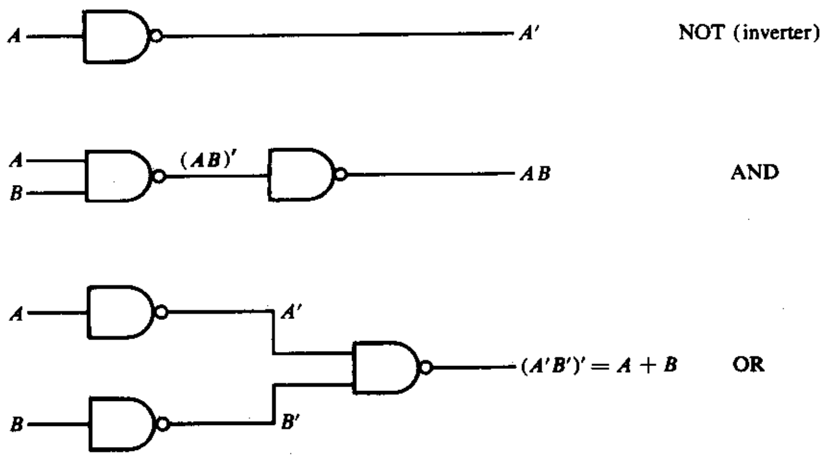


FIGURE 4-10
Implementation of NOT, AND, and OR by NAND gates

NAND gates. The implementation of the AND, OR, and NOT operations with NAND gates is shown in Fig. 4-10. The NOT operation is obtained from a one-input NAND gate, actually another symbol for an inverter circuit. The AND operation requires two NAND gates. The first produces the inverted AND and the second acts as an inverter to produce the normal output. The OR operation is achieved through a NAND gate with additional inverters in each input.

Boolean-Function Implementation

One possible way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit-manipulation techniques that change AND-OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to use the two alternate graphic symbols shown in Fig. 4-11. (These two graphic symbols for the NAND gate were introduced in Fig. 3-17(a) and are repeated here for convenience.) The AND-invert graphic symbol consists of an AND graphic symbol followed by a small circle. The invert-OR graphic symbol consists of an OR graphic symbol that is preceded by small circles in all the inputs. Either symbol can be used to represent a NAND gate.

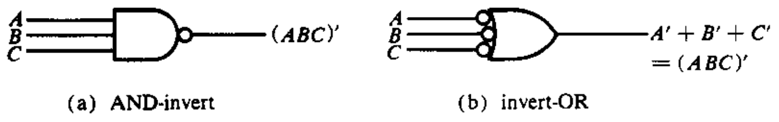


FIGURE 4-11
Two graphic symbols for a NAND gate

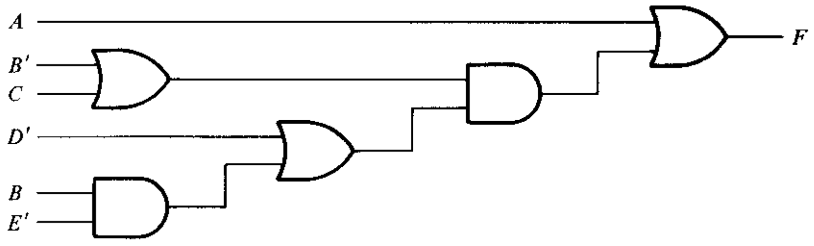
To obtain a multilevel NAND diagram from a Boolean expression, proceed as follows:

1. From the given Boolean expression, draw the logic diagram with AND, OR, and inverter gates. Assume that both the normal and complement inputs are available.
2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
3. Convert all OR gates to NAND gates with invert-OR graphic symbols.
4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same line, insert an inverter (one-input NAND gate) or complement the input variable.

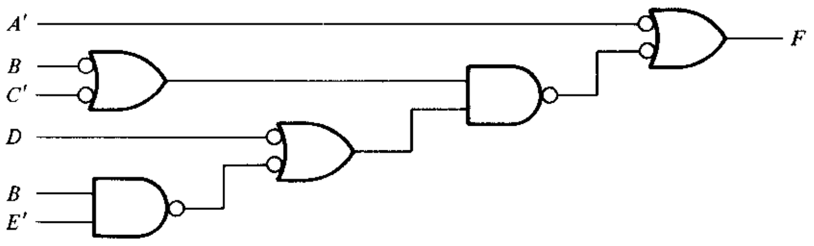
This procedure will be demonstrated with two examples. First, consider the Boolean function

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

Although it is possible to remove the parentheses and convert the expression into a standard sum of products form, we choose to implement it as a multilevel circuit for illustration. The AND-OR implementation is shown in Fig. 4-12(a). There are four levels of gating in the circuit. The first level has an AND and an OR gate. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level. A logic diagram with a pattern of alternate levels of AND and OR gates

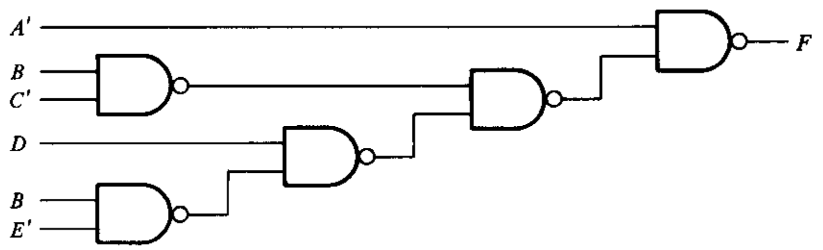


(a) AND-OR diagram



(b) NAND diagram using two graphic symbols

FIGURE 4-12
Implementing $F = A + (B' + C)(D' + BE')$ with NAND gates (continued on next page)



(c) NAND diagram using one graphic symbol

FIGURE 4-12 *(continued)*

can be easily converted into a NAND circuit. This is shown in Fig. 4-12(b). The procedure requires that we change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND-OR circuit as long as the complementing small circles do not change the value of the function. Any connection between an output of a gate that has a complementing circle and the input of another gate that also has a complementing circle represents double complementation and does not change the logic of the circuit. However, the small circles associated with inputs A, B', C, and D' cause extra complementations that are not compensated with other small circles along the same line. We can insert inverters after each of these inputs or, as shown in the figure, complement the literals to obtain A', B, C', and D.

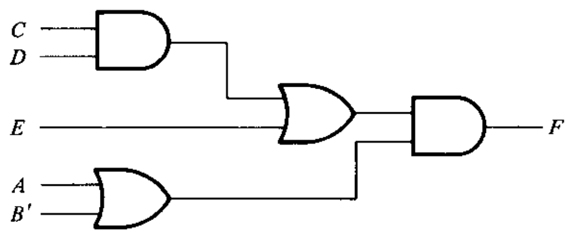
Because it does not matter whether we use the AND-invert or the invert-OR graphic symbol to represent a NAND gate, the diagram of Fig. 4-12(c) is identical to the NAND diagram of part (b). In fact, the diagram of Fig. 4-12(b) is preferable because it represents a clearer picture of the Boolean expression it implements.

As another example, consider the multilevel Boolean expression

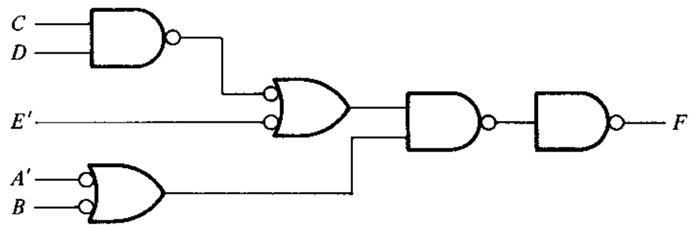
$$F = (CD + E)(A + B')$$

The AND-OR implementation is shown in Fig. 4-13(a) with three levels of gating. The conversion into a NAND circuit is presented in part (b) of the diagram. The three additional small circles associated with inputs E, A, and B' cause these three literals to be complemented to E', A', and B. The small circle in the last NAND gate complements the output, so we need to insert an inverter gate at the output in order to complement the signal again and obtain the original value.

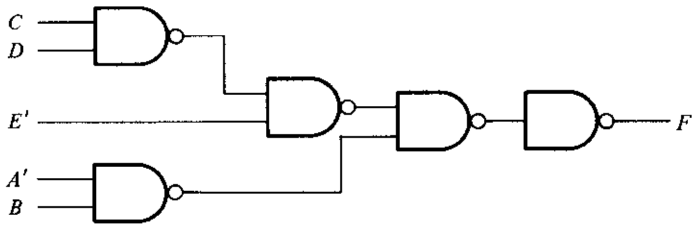
The number of NAND gates required to implement the expression of the first example is the same as the number of AND and OR gates in the AND-OR diagram. The number of NAND gates in the second example is equal to the number of AND-OR gates plus an additional inverter in the output. In general, the number of NAND gates required to implement a Boolean expression is equal to the number of AND-OR gates except for an occasional inverter. This is true provided both the normal and complement inputs are available, because the conversion forces certain input variables to be complemented.



(a) AND-OR diagram



(b) NAND diagram



(c) Alternate NAND diagram

FIGURE 4-13
Implementing $F = (CD + E)(A + B')$ with NAND gates

Analysis Procedure

The foregoing procedure considered the problem of deriving a NAND logic diagram from a given Boolean function. The reverse process is the analysis problem that starts with a given NAND logic diagram and culminates with a Boolean expression or a truth table. The analysis of NAND logic diagrams follows the same procedures presented in Section 4-6 for the analysis of combinational circuits. The only difference is that NAND logic requires a repeated application of DeMorgan's theorem. We shall now demonstrate the derivation of the Boolean function from a logic diagram. Then we will show the derivation of the truth table directly from the NAND logic diagram. Finally, a method will be presented for converting a NAND logic diagram to AND-OR logic diagram.

Derivation of the Boolean Function by Algebraic Manipulation

The procedure for deriving the Boolean function from a logic diagram is outlined in Section 4-6. This procedure is demonstrated for the NAND logic diagram shown in Fig. 4-14. First, all gate outputs are labeled with arbitrary symbols. Second, the Boolean functions for the outputs of gates that receive only external inputs are derived:

$$T_1 = (CD)' = C' + D'$$

$$T_2 = (BC')' = B' + C$$

The second form follows directly from DeMorgan's theorem and may, at times, be more convenient to use. Third, Boolean functions of gates that have inputs from previously derived functions are determined in consecutive order until the output is expressed in terms of input variables:

$$\begin{aligned} T_3 &= (B'T_1)' = (B'C' + B'D')' \\ &= (B + C)(B + D) = B + CD \\ T_4 &= (AT_3)' = [A(B + CD)]' \\ F &= (T_2T_4)' = \{(BC')'[A(B + CD)]'\}' \\ &= BC' + A(B + CD) \end{aligned}$$

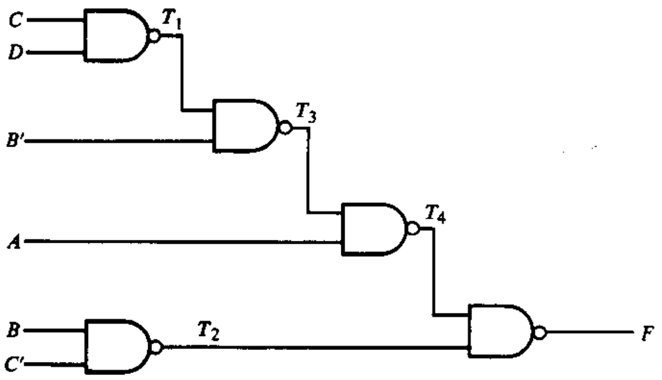


FIGURE 4-14
Analysis example

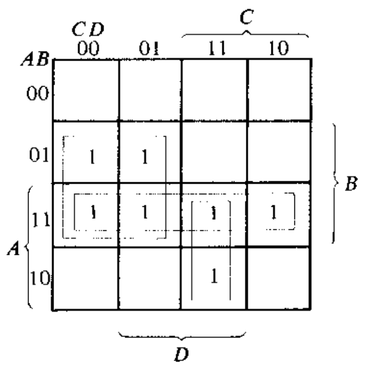
Derivation of the Truth Table

The procedure for obtaining the truth table directly from a logic diagram is also outlined in Section 4-6. This procedure is demonstrated for the NAND logic diagram of Fig. 4-14. First, the four input variables, together with their 16 combinations of 1's and 0's, are listed as in Table 4-3. Second, the outputs of all gates are labeled with arbitrary symbols as in Fig. 4-14. Third, we obtain the truth table for the outputs of those

TABLE 4-3
Truth Table for the Circuit of Figure 4-14

A	B	C	D	T_1	T_2	T_3	T_4	F
0	0	0	0	1	1	0	1	0
0	0	0	1	1	1	0	1	0
0	0	1	0	1	1	0	1	0
0	0	1	1	0	1	1	1	0
0	1	0	0	1	0	1	1	1
0	1	0	1	1	0	1	1	1
0	1	1	0	1	1	1	1	0
0	1	1	1	0	1	1	1	0
1	0	0	0	1	1	0	1	0
1	0	0	1	1	1	0	1	0
1	0	1	0	1	1	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	0	1	0	1	0	1
1	1	0	1	1	0	1	0	1
1	1	1	0	1	1	1	0	1
1	1	1	1	0	1	1	0	1

gates that are a function of the input variables only. These are T_1 and T_2. $T_1 = (CD)'$; so we mark 0's in those rows where both C and D are equal to 1 and fill the rest of the rows of T_1 with 1's. Also, $T_2 = (BC')'$; so we mark 0's in those rows where $B = 1$ and $C = 0$, and fill the rest of the rows of T_2 with 1's. We then proceed to obtain the truth table for the outputs of those gates that are a function of previously defined outputs until the column for the output F is determined. It is now possible to obtain an algebraic expression for the output from the derived truth table. The map shown in Fig. 4-15 is ob-



$F = AB + BC' + ACD$

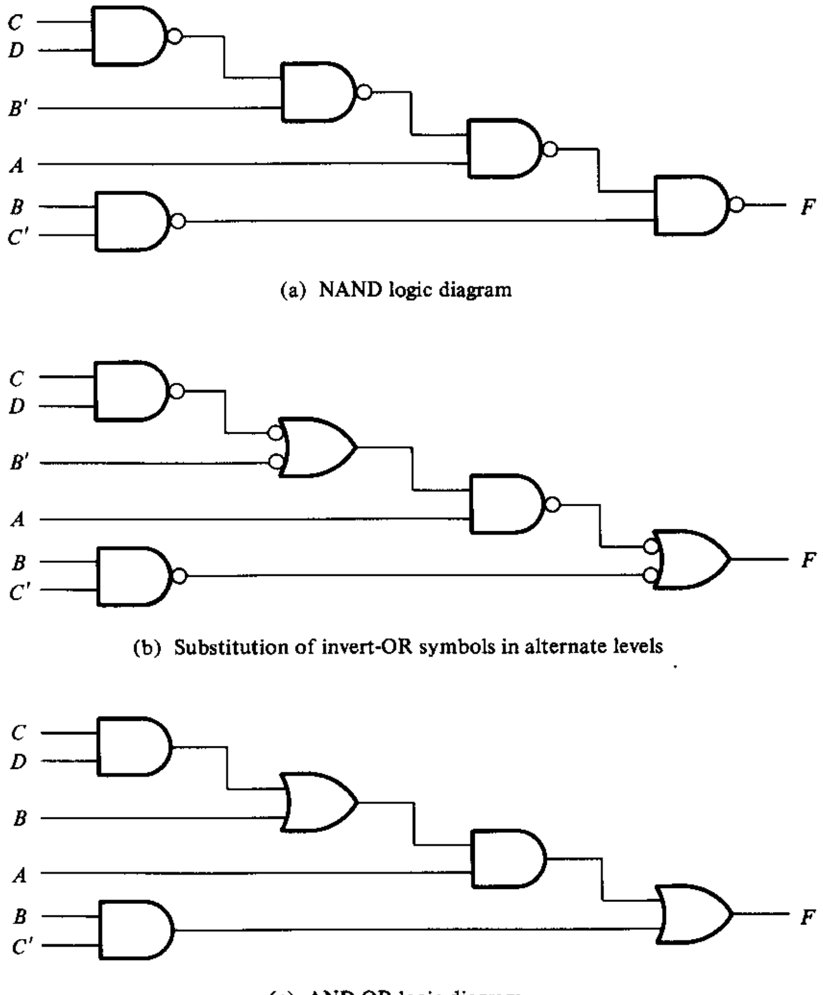
FIGURE 4-15
Derivation of F from Table 4-3

tained directly from Table 4-3 and has 1's in the squares of those minterms for which F is equal to 1. The simplified expression obtained from the map is

$$F = AB + ACD + BC' = A(B + CD) + BC'$$

Transformation to AND-OR Diagram

It is sometimes convenient to convert a NAND logic diagram to its equivalent AND-OR logic diagram to facilitate the analysis procedure. By doing so, the Boolean expression can be derived more easily from the diagram without employing DeMorgan's the-



(a) NAND logic diagram

(b) Substitution of invert-OR symbols in alternate levels

(c) AND-OR logic diagram

FIGURE 4-16
Conversion of NAND logic diagram to AND-OR

orem. The conversion is achieved through a change in graphic symbols from AND-invert to invert-OR in *alternate* levels in the gate structure. The first level to be changed to an invert-OR symbol should be the last level. These changes produce pairs of small circles along the same line, which are then removed since they represent double complementation. Any small circle associated with an input can be removed provided the input variable is complemented. A one-input AND or OR gate with a small circle in the input or output represents an inverter circuit.

The procedure is demonstrated in Fig. 4-16. The NAND logic diagram of Fig. 4-16(a) is to be converted to an equivalent AND-OR diagram. The graphic symbol of the NAND gate in the last level is changed to an invert-OR symbol. Looking for alternate levels, we find one more gate requiring a change of symbol, as shown in Fig. 4-16(b). Any two small circles along the same line are removed. The small circle connected to input B' is removed and the input variable is complemented. The required AND-OR logic diagram is shown in Fig. 4-16(c). The Boolean expression for F can be easily determined from the AND-OR diagram to be

$$F = BC' + A(B + CD)$$

4-8 MULTILEVEL NOR CIRCUITS

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules developed for NAND logic. This section enumerates various methods for NOR logic implementation and analysis by following the same list of topics used for NAND logic. However, less detailed explanation is included so as to avoid excessive repetition of the material in Section 4-7.

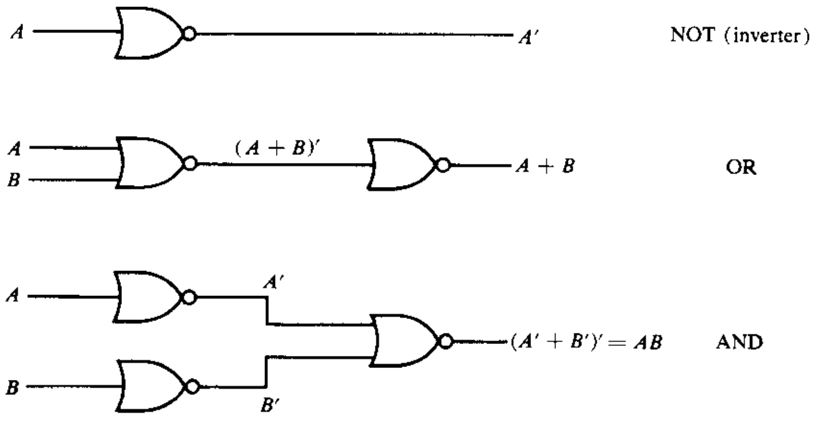


FIGURE 4-17
Implementation of NOT, OR, and AND by NOR gates

Universal Gate

The NOR gate is universal because any Boolean function can be implemented with it, including a flip-flop circuit, as shown in Section 6-2. The conversion of AND, OR, and NOT to NOR is shown in Fig. 4-17. The NOT operation is obtained from a one-input NOR gate, yet another symbol for an inverter circuit. The OR operation requires two NOR gates. The first produces the inverted-OR and the second acts as an inverter to obtain the normal output. The AND operation is achieved through a NOR gate with additional inverters at each input.

Boolean-Function Implementation

The two graphic symbols for the NOR gate are shown in Fig. 4-18. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.

The procedure for implementing a Boolean function with NOR gates is similar to the procedure outlined in the previous section for NAND gates.

1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.
2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
4. Any small circle that is not compensated by another small circle along the same line needs an inverter or the complementation of the input variable.

The procedure is illustrated in Fig. 4-19 for the Boolean function

$$F = (AB + E)(C + D)$$

The AND-OR implementation of the expression is shown in the logic diagram of Fig. 4-19(a). For each OR gate, we substitute a NOR gate with the OR-invert graphic symbol. For each AND gate, we substitute a NOR gate with the invert-AND graphic symbol. The two small circles associated with inputs A and B cause these two variables to be complemented to A' and B', respectively. The NOR diagram is shown in Fig. 4-19(b). The diagram of Fig. 4-19(c) is an alternate way of drawing the diagram using only one type of graphic symbol for the NOR gate.

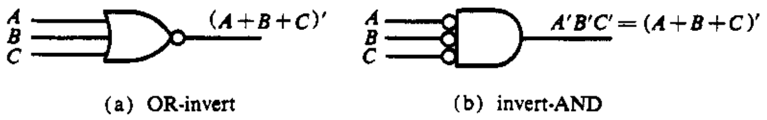
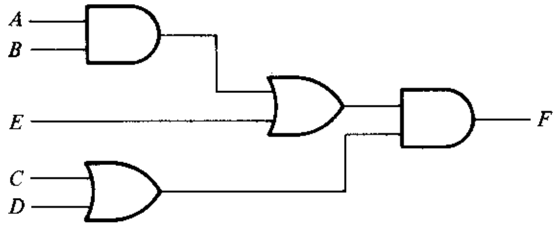
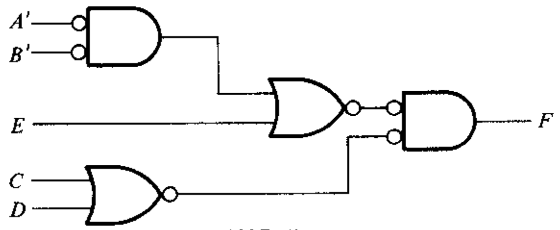


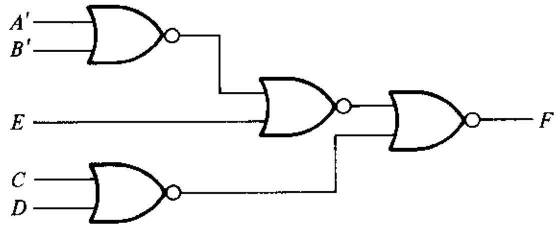
FIGURE 4-18
Two graphic symbols for a NOR gate



(a) AND-OR diagram



(b) NOR diagram



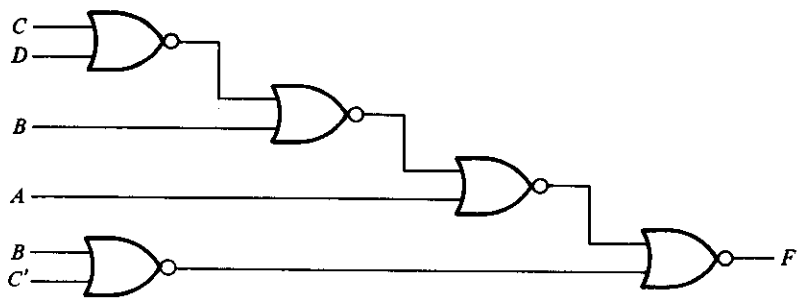
(c) Alternate NOR diagram

FIGURE 4-19
Implementing $F = (AB + E)(C + D)$ with NOR gates

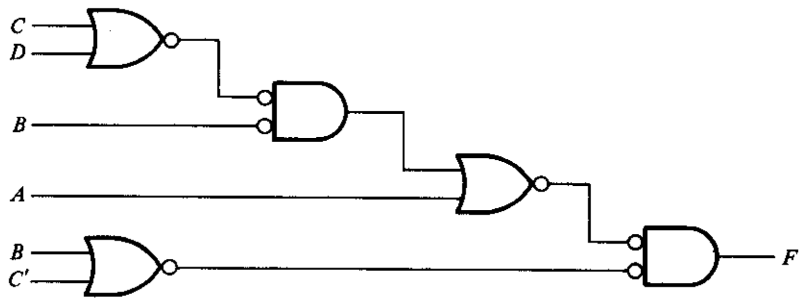
In general, the number of NOR gates required to implement a Boolean function will be the same as the number of gates in the AND-OR diagram. This is true provided both the normal and complement inputs are available, because the conversion may require that certain input variables be complemented.

Analysis Procedure

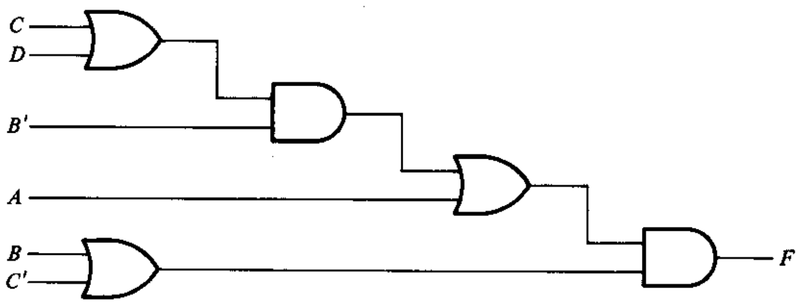
The analysis of NOR logic diagrams follows the same procedure presented in Section 4-6 for the analysis of combinational circuits. To derive the Boolean expression from a logic diagram, we mark the outputs of various gates with arbitrary symbols. By repetitive substitutions, we obtain the output variable as a function of the input variables.



(a) NOR logic diagram



(b) Sustituting invert-AND in alternate levels



(c) AND-OR logic diagram

FIGURE 4-20
Conversion of NOR diagram to AND-OR

To obtain the truth table from a logic diagram without first deriving the Boolean expression, we form a table with the n variables by listing the 2^n binary combinations. The truth table of selected NOR gate outputs is derived in succession until the output truth table is obtained. The output expression of a typical NOR gate is of the form $T = (A + B' + C)'$. By using DeMorgan's theorem, this can be expressed as $T = A'BC'$. The truth table for T is marked with 1's for those combinations where $ABC = 010$ and the rest of the rows are filled with 0's.

The conversion of a NOR logic diagram to an AND-OR diagram is achieved through a change of graphic symbols from OR-invert to invert-AND starting from the last logic level and in alternate levels. Pairs of small circles along the same line are removed. A one-input AND or OR gate is removed, but if it has a small circle at the input or output, it is converted to an inverter. Any small circle associated with an input is removed and the input variable is complemented.

This procedure is demonstrated in Fig. 4-20, where the NOR logic diagram in part (a) is converted to an AND-OR diagram. The graphic symbol of the gate in the last (fourth) logic level is changed to an invert-AND. Looking for alternate levels, we find a gate in level two that needs to undergo a symbol change, as shown in part (b). Any two circles along the same line are removed. The circle associated with external input B is removed and the input variable is changed to B'. The required AND-OR logic diagram is drawn in part (c). The Boolean expression for the circuit can be obtained by inspection and then manipulated into a product of sums form:

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

4-9 EXCLUSIVE-OR FUNCTION

The exclusive-OR (XOR) denoted by the symbol $\oplus$ is a logical operation that performs the following Boolean operation:

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

It is equal to 1 if only x is equal to 1 or if only y is equal to 1 but not when both are equal to 1. The exclusive-NOR, also known as equivalence, performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

It is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation.

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

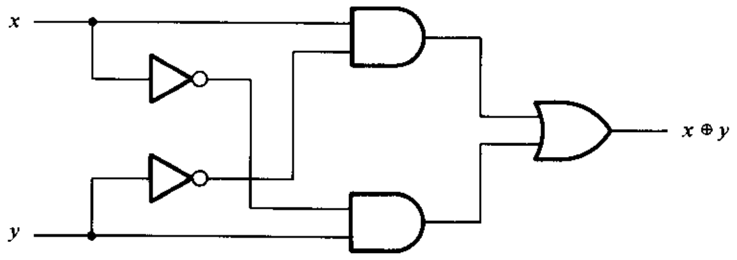
The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x \qquad x \oplus 1 = x'$$
$$x \oplus x = 0 \qquad x \oplus x' = 1$$
$$x \oplus y' = (x \oplus y)' \qquad x' \oplus y = (x \oplus y)'$$

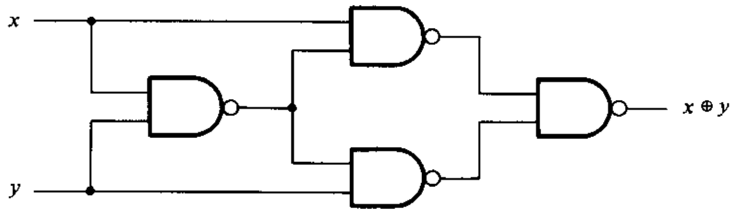
Any of these identities can be proven by using a truth table or by replacing the $\oplus$ operation by its equivalent Boolean expression. It can be shown also that the exclusive-OR operation is both commutative and associative.

$$A \oplus B = B \oplus A$$
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig. 4-21(a). Figure 4-21(b) shows the implementation of the exclusive-OR with four NAND gates.



(a) With AND-OR-NOT gates



(b) With NAND gates

FIGURE 4-21
Exclusive-OR implementations

The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit produces the sum of products of its inputs:

$$(x' + y')x + (x' + y')y = xy' + x'y = x \oplus y$$

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error-detection and correction circuits.

Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

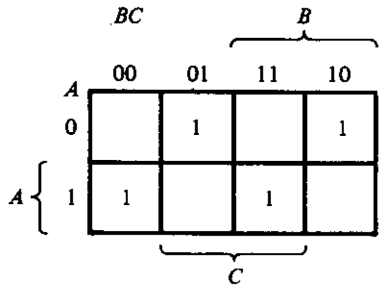
$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma\,(1, 2, 4, 7) \end{aligned}$$

The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, where only one variable must be equal to 1, in the three or more variable case, the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function*.

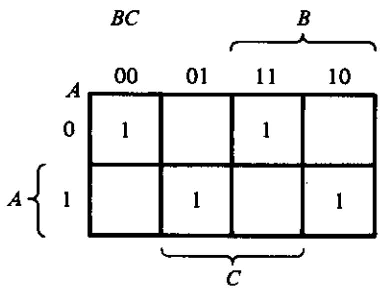
The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010, 100, and 111. Each of these binary numbers has an odd number of 1's. The other four minterms not included in the function are 000, 011, 101, and 110, and they have an even number of 1's in their binary numerical values. In general, an n-variable exclusive-OR function is an odd function defined as the logical sum of the $2^n/2$ minterms whose binary numerical values have an odd number of 1's.

The definition of an odd function can be clarified by plotting it in a map. Figure 4-22(a) shows the map for the three-variable exclusive-OR function. The four minterms of the function are a unit distance apart from each other. The odd function is identified from the four minterms whose binary values have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 4-22(b), the three-variable even function is equal to 1 when an even number of variables is equal to 1 (including the condition that none of the variables is equal to 1).

The 3-input odd function is implemented by means of 2-input exclusive-OR gates, as shown in Fig. 4-23(a). The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate, as shown in Fig. 4-23(b).



(a) Odd function
$F = A \oplus B \oplus C$



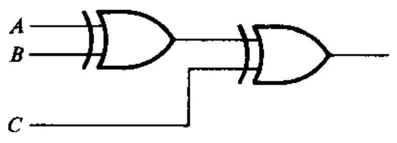
(b) Even function
$F = (A \oplus B \oplus C)'$

FIGURE 4-22
Map for a three-variable exclusive-OR function

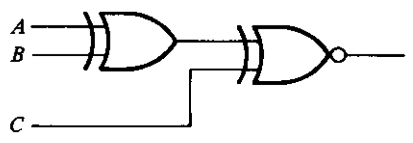
Consider now the the four-variable exclusive-OR operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

$$\begin{aligned} A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\ &= \Sigma\,(1, 2, 4, 7, 8, 11, 13, 14) \end{aligned}$$

There are 16 minterms for a four-variable Boolean function. Half of the minterms have binary numerical values with an odd number of 1's; the other half of the minterms have binary numerical values with an even number of 1's. When plotting the function in the map, the binary numerical value for a minterm is determined from the row and column numbers of the square that represents the minterm. The map of Fig. 4-24(a) is a plot of the four-variable exclusive-OR function. This is an odd function because the binary values of all the minterms have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 4-24(b), the four-variable even function is equal to 1 when an even number of variables is equal to 1.

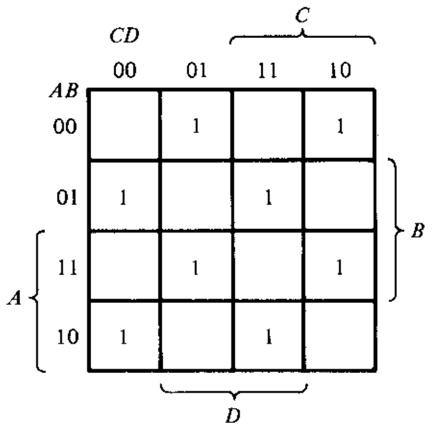


(a) 3-input odd function

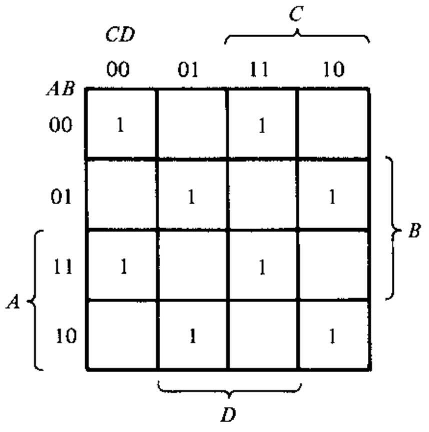


(b) 3-input even function

FIGURE 4-23
Logic diagram of odd and even functions



(a) Odd function
$F = A \oplus B \oplus C \oplus D$



(b) Even function
$F = (A \oplus B \oplus C \oplus D)'$

FIGURE 4-24
Map for a four-variable exclusive-OR function

Parity Generation and Checking

Exclusive-OR functions are very useful in systems requiring error-detection and correction codes. As discussed in Section 1-7, a parity bit is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator*. The circuit that checks the parity in the receiver is called a *parity checker*.

As an example, consider a 3-bit message to be transmitted together with an even parity bit. Table 4-4 shows the truth table for the parity generator. The three bits, x, y, and z, constitute the message and are the inputs to the circuit. The parity bit P is the output. For even parity, the bit P must be generated to make the total number of 1's even (including P). From the truth table, we see that P constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, P can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator is shown in Fig. 4-25(a).

The three bits in the message together with the parity bit are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the

TABLE 4-4
Even-Parity-Generator Truth Table

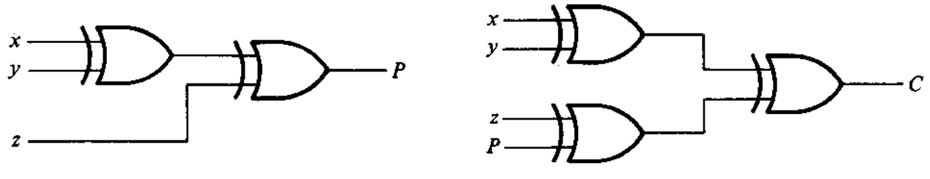
Three-Bit Message			Parity Bit
x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by C, will be equal to 1 if an error occurs, that is, if the four bits received have an odd number of 1's. Table 4-5 is the truth table for the even-parity checker. From it we see that the function C consists of the eight minterms with binary numerical values having an odd number of 1's. This corresponds to the map of Fig. 4-24(a), which represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker is shown in Fig. 4-25(b).

It is worth noting that the parity generator can be implemented with the circuit of Fig. 4-25(b) if the input P is connected to logic-0 and the output is marked with P. This is because $z \oplus 0 = z$, causing the value of z to pass through the gate unchanged. The advantage of this is that the same circuit can be used for both parity generation and checking.



(a) 3-bit even parity generator (b) 4-bit even parity checker

FIGURE 4-25
Logic diagram of a parity generator and checker

TABLE 4-5
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

It is obvious from the foregoing example that parity-generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

REFERENCES

1. HILL, F. J., and G. R. PETERSON, *Introduction to Switching Theory and Logical Design,* 3rd Ed. New York: John Wiley, 1981.
2. KOHAVI, Z., *Switching and Automata Theory,* 2nd Ed. New York: McGraw-Hill, 1978.
3. ROTH, C. H., *Fundamentals of Logic Design,* 3rd Ed. St. Paul, Minnesota: West Publishing Co., 1985.
4. BOOTH, T. L., *Introduction to Computer Engineering,* 3rd Ed. New York: John Wiley, 1984.
5. MANO, M. M., *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice-Hall, 1988.
6. FLETCHER, W. I., *An Engineering Approach to Digital Design.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

7. Ercegovac, M. D., and T. Lang, *Digital Systems and Hardware/Firmware Algorithms*. New York: John Wiley, 1985.

8. Mange, D., *Analysis and Synthesis of Logic Systems*. Norwood, MA: Artech House, 1986.

9. Shiva, S. G., *Introduction to Logic Design*. Glenview, IL: Scott, Foresman, 1988.

10. McCluskey, E. J., *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

PROBLEMS

4-1 Design a combinational circuit with three inputs and one output. The output is equal to logic-1 when the binary value of the input is less than 3. The output is logic-0 otherwise.

4-2 Design a combinational circuit with three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is one less than the input.

4-3 A majority function is generated in a combinational circuit when the output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise. Design a 3-input majority function.

4-4 Design a combinational circuit that adds one to a 4-bit binary number, $A_3A_2A_1A_0$. For example, if the input of the circuit is $A_3A_2A_1A_0 = 1101$, the output is 1110. The circuit can be designed using four half-adders.

4-5 A combinational circuit produces the binary sum of two 2-bit numbers, x_1x_0 and y_1y_0. The outputs are C, S_1, and S_0. Provide a truth table of the combinational circuit.

4-6 Design the circuit of Problem 4-5 using two full-adders.

4-7 Design a combinational circuit that multiplies two 2-bit numbers, a_1a_0 and b_1b_0, to produce a 4-bit product, $c_3c_2c_1c_0$. Use AND gates and half-adders.

4-8 Show that a full-subtractor can be constructed with two half-subtractors and an OR gate.

4-9 Design a combinational circuit with three inputs and six outputs. The output binary number should be the square of the input binary number.

4-10 Design a combinational circuit with four inputs that represent a decimal digit in BCD and four outputs that produce the 9's complement of the input digit. The six unused combinations can be treated as don't-care conditions.

4-11 Design a combinational circuit with four inputs and four outputs. The output generates the 2's complement of the input binary number.

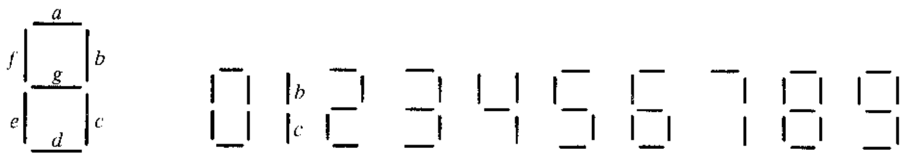
4-12 Design a combinational circuit that detects an error in the representation of a decimal digit in BCD. The output of the circuit must be equal to logic-1 when the inputs contain any one of the six unused bit combinations in the BCD code.

4-13 Design a code converter that converts a decimal digit from the 8 4 −2 −1 code to BCD (see Table 1-2.)

4-14 Design a combinational circuit that converts a decimal digit from the 2 4 2 1 code to the 8 4 −2 −1 code (see Table 1-2.)

4-15 Design a combinational circuit that converts a binary number of four bits to a decimal number in BCD. Note that the BCD number is the same as the binary number as long as the input is less than or equal to 9. The binary number from 1010 to 1111 converts into BCD numbers from 1 0000 to 1 0101.

4-16 A BCD-to-seven-segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in a display indicator used for displaying the decimal digit in a familiar form. The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display, as shown in Fig. P4-16(a). The numeric designation chosen to represent the decimal digit is shown in Fig. P4-16(b). Design the BCD-to-seven-segment decoder using a minimum number of NAND gates. The six invalid combinations should result in a blank display.



(a) Segment designation (b) Numerical designation for display

FIGURE P4-16

4-17 Analyze the two-output combinational circuit shown in Fig. P4-17. Find the Boolean functions for the two outputs as a function of the three inputs and explain the circuit operation.

4-18 Derive the truth table of the circuit shown in Fig. P4-17.

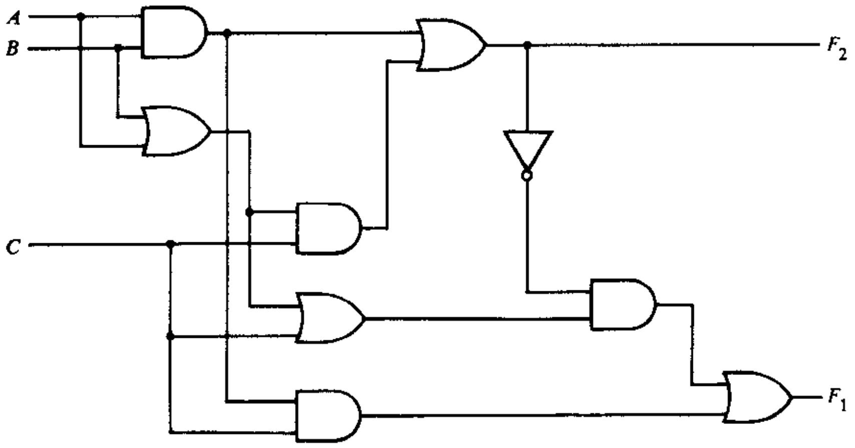


FIGURE P4-17

4-19 Draw the NAND logic diagram for each of the following expressions using multiple-level NAND gate circuits:
(a) $(AB' + CD')E + BC(A + B)$
(b) $w(x + y + z) + xyz$

4-20 Convert the logic diagram of the code converter shown in Fig. 4-8 to a multiple-level NAND circuit.

4-21 Determine the Boolean functions for outputs F and G as a function of four inputs, A, B, C, and D, in Fig. P4-21.

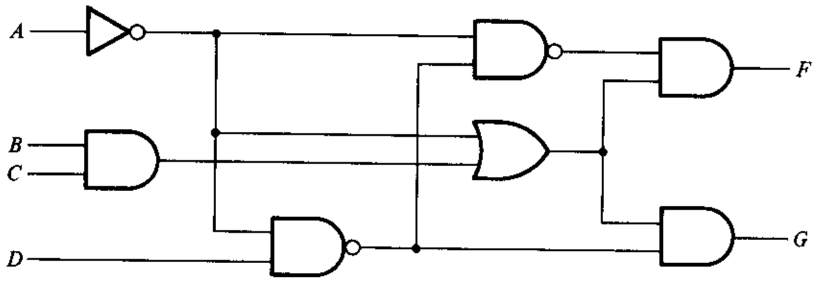


FIGURE P4-21

4-22 Verify that the circuit of Fig. P4-22 generates the exclusive-NOR function.

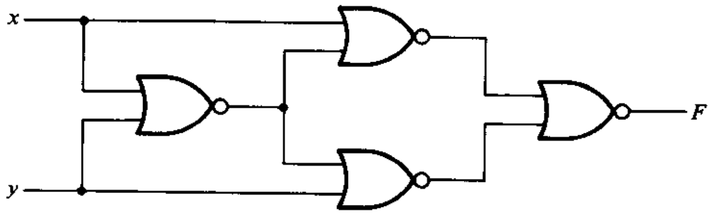


FIGURE P4-22

4-23 Convert the logic diagram of the code converter shown in Fig. 4-8 to a multiple-level NOR circuit.

4-24 Derive the truth table for the output of each NOR gate in Fig. 4-20(a).

4-25 Prove that $x' \oplus y = x \oplus y' = (x \oplus y)' = xy + x'y'$.

4-26 Prove that $x \oplus 1 = x'$ and $x \oplus 0 = x$.

4-27 Show that if $xy = 0$, then $x \oplus y = x + y$.

4-28 Design a combinational circuit that converts a 4-bit Gray code number (Table 1-4) to a 4-bit straight binary number. Implement the circuit with exclusive-OR gates.

4-29 Design the circuit of a 3-bit parity generator and the circuit of a 4-bit parity checker using an odd parity bit.

4-30 Manipulate the following Boolean expression in such a way so that it can be implemented using exclusive-OR and AND gates only.

$$AB'CD' + A'BCD' + AB'C'D + A'BC'D$$