CHAPTER 20

DIGITAL LOGIC

The operation of the digital computer is based on the storage and processing of binary data. Throughout this book, we have assumed the existence of storage elements that can exist in one of two stable states and of circuits than can operate on binary data under the control of control signals to implement the various computer functions. In this appendix, we suggest how these storage elements and circuits can be implemented in digital logic, specifically with combinational and sequential circuits. The appendix begins with a brief review of Boolean algebra, which is the mathematical foundation of digital logic. Next, the concept of a gate is introduced. Finally, combinational and sequential circuits, which are constructed from gates, are described.

20.1 BOOLEAN ALGEBRA

The digital circuitry in digital computers and other digital systems is designed, and its behavior is analyzed, with the use of a mathematical discipline known as *Boolean algebra*. The name is in honor of an English mathematician George Boole, who proposed the basic principles of this algebra in 1854 in his treatise, *An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities.* In 1938, Claude Shannon, a research assistant in the Electrical Engineering Department at M.I.T., suggested that Boolean algebra could be used to solve problems in relay-switching circuit design [SHAN38].[1] Shannon's techniques were subsequently used in the analysis and design of electronic digital circuits. Boolean algebra turns out to be a convenient tool in two areas:

- **Analysis:** It is an economical way of describing the function of digital circuitry.
- **Design:** Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

As with any algebra, Boolean algebra makes use of variables and operations. In this case, the variables and operations are logical variables and operations. Thus, a variable may take on the value 1 (TRUE) or 0 (FALSE). The basic logical operations are AND, OR, and NOT, which are symbolically represented by dot, plus sign, and overbar:[2]

$$A \text{ AND } B = A \cdot B$$
$$A \text{ OR } B = A + B$$
$$\text{NOT } A = \overline{A}$$

The operation AND yields true (binary value 1) if and only if both of its operands are true. The operation OR yields true if either or both of its operands are true. The unary operation NOT inverts the value of its operand. For example, consider the equation

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

D is equal to 1 if A is 1 or if both B = 0 and C = 1. Otherwise D is equal to 0.

---

[1] The paper is available at this book's Web site.
[2] Logical NOT is often indicated by an apostrophe: NOT A = A′.

Table 20.1 Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P $(\overline{P})$	P AND Q $(P \cdot Q)$	P OR Q $(P + Q)$	P NAND Q $(\overline{P \cdot Q})$	P NOR Q $(\overline{P + Q})$	P XOR Q $(P \oplus Q)$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, ...} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, ...} contains an odd number of ones.

Several points concerning the notation are needed. In the absence of parentheses, the AND operation takes precedence over the OR operation. Also, when no ambiguity will occur, the AND operation is represented by simple concatenation instead of the dot operator. Thus,

$$A + B \cdot C = A + (B \cdot C) = A + BC$$

all mean: Take the AND of B and C; then take the OR of the result and A.

Table 20.1a defines the basic logical operations in a form known as a *truth table,* which lists the value of an operation for every possible combination of values of operands. The table also lists three other useful operators: XOR, NAND, and NOR. The exclusive-or (XOR) of two logical operands is 1 if and only if exactly one of the operands has the value 1. The NAND function is the complement (NOT) of the AND function, and the NOR is the complement of OR:

$$A \text{ NAND } B = \text{NOT } (A \text{ AND } B) = \overline{AB}$$
$$A \text{ NOR } B = \text{NOT } (A \text{ OR } B) = \overline{A + B}$$

As we shall see, these three new operations can be useful in implementing certain digital circuits.

The logical operations, with the exception of NOT, can be generalized to more than two variables, as shown in Table 20.1b.

Table 20.2 summarizes key identities of Boolean algebra. The equations have been arranged in two columns to show the complementary, or dual, nature of the AND and OR operations. There are two classes of identities: basic rules (or *postulates*), which

Table 20.2   Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem

are stated without proof, and other identities that can be derived from the basic postulates. The postulates define the way in which Boolean expressions are interpreted. One of the two distributive laws is worth noting because it differs from what we would find in ordinary algebra:

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

The two bottommost expressions are referred to as DeMorgan's theorem. We can restate them as follows:

$$A \text{ NOR } B = \overline{A} \text{ AND } \overline{B}$$
$$A \text{ NAND } B = \overline{A} \text{ OR } \overline{B}$$

The reader is invited to verify the expressions in Table 20.2 by substituting actual values (1s and 0s) for the variables A, B, and C.

20.2 GATES

The fundamental building block of all digital logic circuits is the gate. Logical functions are implemented by the interconnection of gates.

A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR, and XOR. Figure 20.1 depicts these six gates. Each gate is defined in three ways: graphic symbol, algebraic notation, and truth table. The symbology used here and throughout the appendix is the IEEE standard, IEEE Std 91. Note that the inversion (NOT) operation is indicated by a circle.

Each gate shown in Figure 20.1 has one or two inputs and one output. However, as indicated in Table 20.1b, all of the gates except NOT can have more than two inputs. Thus, $(X + Y + Z)$ can be implemented with a single OR gate with three inputs. When one or more of the values at the input are changed, the correct output signal appears almost instantaneously, delayed only by the propagation time of
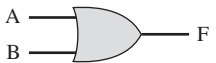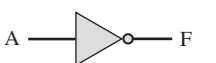
Name	Graphical Symbol	Algebraic Function	Truth Table
AND	A ——⊐ F B ——	$F = A \cdot B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A ——⊐ F B ——	$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT	A ——▷o—— F	$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND	A ——⊐o— F B ——	$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A ——⊐o— F B ——	$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR	A ——⊐ F B ——	$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

Figure 20.1 Basic Logic Gates

signals through the gate (known as the *gate delay*). The significance of this delay is discussed in Section 20.3. In some cases, a gate is implemented with two outputs, one output being the negation of the other output.

Here we introduce a common term: we say that to **assert** a signal is to cause signal line to make a transition from its logically false (0) state to its logically true (1) state. The true (1) state is either a high or low voltage state, depending on the type of electronic circuitry.

Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR
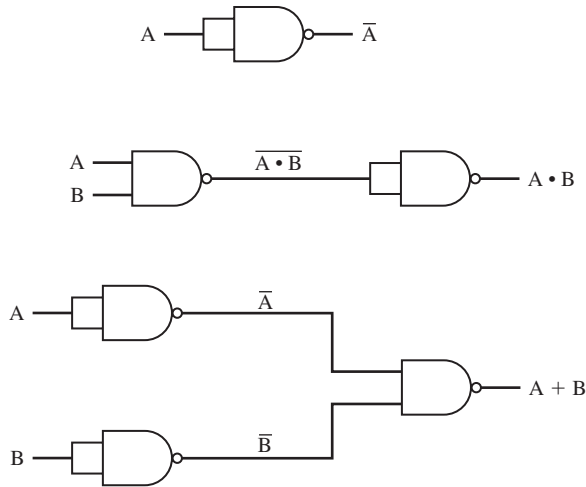
Figure 20.2   The Use of NAND Gates

It should be clear that AND, OR, and NOT gates constitute a functionally complete set, because they represent the three operations of Boolean algebra. For the AND and NOT gates to form a functionally complete set, there must be a way to synthesize the OR operation from the AND and NOT operations. This can be done by applying DeMorgan's theorem:

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

$$A \text{ OR } B = \text{NOT ((NOT A) AND (NOT B))}$$

Similarly, the OR and NOT operations are functionally complete because they can be used to synthesize the AND operation.

Figure 20.2 shows how the AND, OR, and NOT functions can be implemented solely with NAND gates, and Figure 20.3 shows the same thing for NOR gates. For
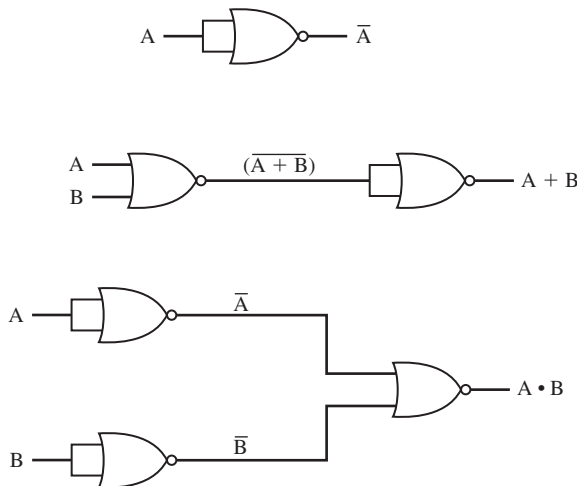
Figure 20.3   The Use of NOR Gates

this reason, digital circuits can be, and frequently are, implemented solely with NAND gates or solely with NOR gates.

    With gates, we have reached the most primitive circuit level of computer hardware. An examination of the transistor combinations used to construct gates departs from that realm and enters the realm of electrical engineering. For our purposes, however, we are content to describe how gates can be used as building blocks to implement the essential logical circuits of a digital computer.

20.3 COMBINATIONAL CIRCUITS

A combinational circuit is an interconnected set of gates whose output at any time is a function only of the input at that time. As with a single gate, the appearance of the input is followed almost immediately by the appearance of the output, with only gate delays.

    In general terms, a combinational circuit consists of n binary inputs and m binary outputs. As with a gate, a combinational circuit can be defined in three ways:

- **Truth table:** For each of the 2^n possible combinations of input signals, the binary value of each of the m output signals is listed.
- **Graphical symbols:** The interconnected layout of gates is depicted.
- **Boolean equations:** Each output signal is expressed as a Boolean function of its input signals.

Implementation of Boolean Functions

Any Boolean function can be implemented in electronic form as a network of gates. For any given function, there are a number of alternative realizations. Consider the Boolean function represented by the truth table in Table 20.3. We can express this function by simply itemizing the combinations of values of A, B, and C that cause F to be 1:

$$F = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C} \tag{20.1}$$

    There are three combinations of input values that cause F to be 1, and if any one of these combinations occurs, the result is 1. This form of expression, for self-evident

Table 20.3   A Boolean Function of Three Variables

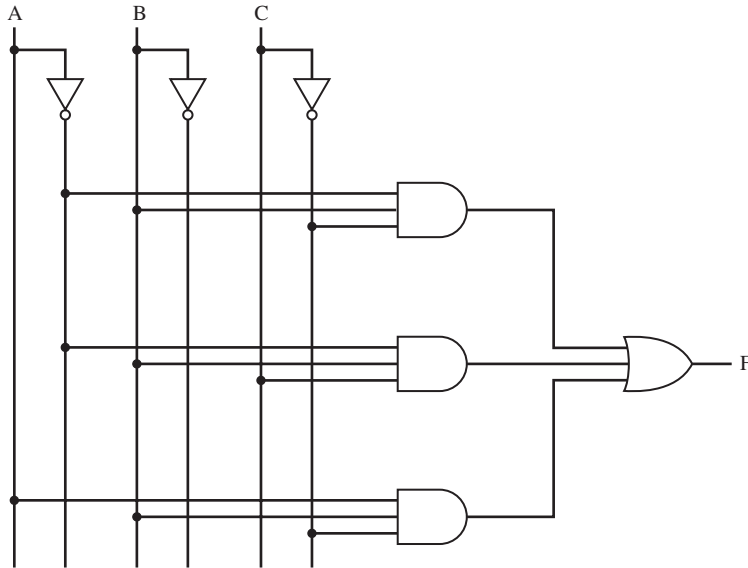| A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Figure 20.4   Sum-of-Products Implementation of Table 20.3

reasons, is known as the *sum of products* (SOP) form. Figure 20.4 shows a straightforward implementation with AND, OR, and NOT gates.

Another form can also be derived from the truth table. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true. We can also say that the output is 1 if none of the input combinations that produce 0 is true. Thus,

$$F = \overline{(\overline{A}\,\overline{B}\,\overline{C})} \cdot \overline{(\overline{A}\,\overline{B}\,C)} \cdot \overline{(A\,\overline{B}\,\overline{C})} \cdot \overline{(A\,\overline{B}\,C)} \cdot \overline{(ABC)}$$

This can be rewritten using a generalization of DeMorgan's theorem:

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

Thus,

$$
\begin{aligned}
F &= (\overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}}) \cdot (\overline{\overline{A}} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{A} + \overline{\overline{B}} + \overline{\overline{C}}) \cdot (\overline{A} + \overline{\overline{B}} + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C}) \\
&= (A + B + C) \cdot (A + B + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C})
\end{aligned}
\tag{20.2}
$$

This is in the *product of sums* (POS) form, which is illustrated in Figure 20.5. For clarity, NOT gates are not shown. Rather, it is assumed that each input signal and its complement are available. This simplifies the logic diagram and makes the inputs to the gates more readily apparent.

Thus, a Boolean function can be realized in either SOP or POS form. At this point, it would seem that the choice would depend on whether the truth table contains more 1s or 0s for the output function: The SOP has one term for each 1, and the POS has one term for each 0. However, there are other considerations:

- It is often possible to derive a simpler Boolean expression from the truth table than either SOP or POS.
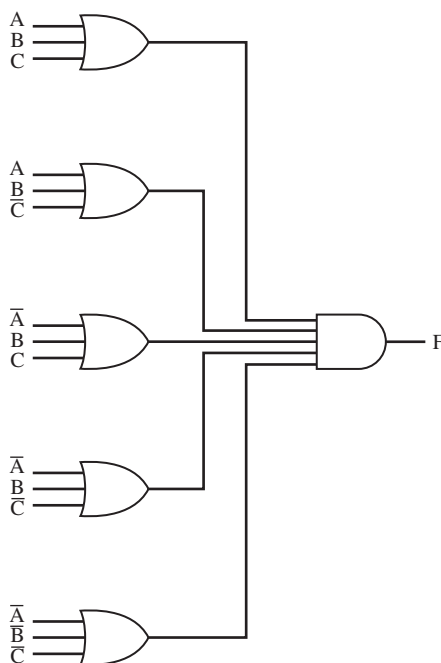
Figure 20.5 Product-of-Sums
Implementation of Table 20.3

- It may be preferable to implement the function with a single gate type (NAND or NOR).

The significance of the first point is that, with a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are

- Algebraic simplification
- Karnaugh maps
- Quine–McKluskey tables

ALGEBRAIC SIMPLIFICATION Algebraic simplification involves the application of the identities of Table 20.2 to reduce the Boolean expression to one with fewer elements. For example, consider again Equation (20.1). Some thought should convince the reader that an equivalent expression is

$$F = \overline{A}B + B\overline{C} \tag{20.3}$$

Or, even simpler,

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

This expression can be implemented as shown in Figure 20.6. The simplification of Equation (20.1) was done essentially by observation. For more complex expressions, some more systematic approach is needed.

Figure 20.6 Simplified Implementation of Table A.3

KARNAUGH MAPS For purposes of simplification, the Karnaugh map is a convenient way of representing a Boolean function of a small number (up to four) of variables. The map is an array of 2^n squares, representing all possible combinations of valu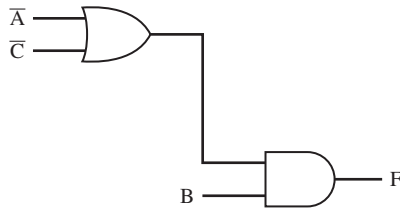es of n binary variables. Figure 20.7a shows the map of four squares for a function of two variables. It is essential for later purposes to list the combinations in the order 00, 01, 11, 10. Because the squares corresponding to the combinations are to be used for recording information, the combinations are customarily written above the squares. In the case of three variables, the representation is an arrangement of eight squares (Figure 20.7b), with the values for one of the variables to the left and for the other two variables above the squares. For four variables, 16 squares are needed, with the arrangement indicated in Figure 20.7c.

The map can be used to represent any Boolean function in the following way. Each square corresponds to a unique product in the sum-of-products form, with a 1 value corresponding to the variable and a 0 value corresponding to the NOT of that

(a) $F = A\overline{B} + \overline{A}B$

(b) $F = \overline{A}B\overline{C} + \overline{A}BC + AB\overline{C}$

(c) $F = \overline{A}\overline{B}CD + A\overline{B}\overline{C}D + AB\overline{C}\overline{D}$
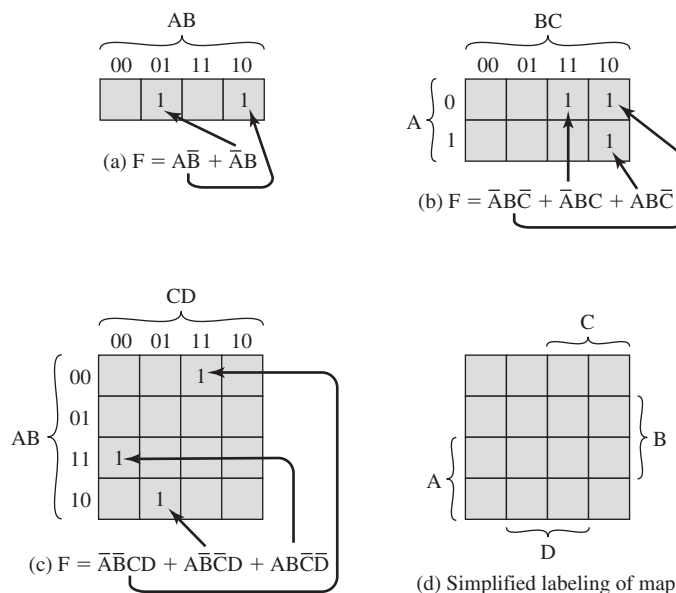
(d) Simplified labeling of map

Figure 20.7 The Use of Karnaugh Maps to Represent Boolean Functions

variable. Thus, the product $A\overline{B}$ corresponds to the fourth square in Figure 20.7a. For each such product in the function, 1 is placed in the corresponding square. Thus, for the two-variable example, the map corresponds to $A\overline{B} + \overline{A}B$. Given the truth table of a Boolean function, it is an easy matter to construct the map: for each combination of values of variables that produce a result of 1 in the truth table, fill in the corresponding square of the map with 1. Figure 20.7b shows the result for the truth table of Table 20.3. To convert from a Boolean expression to a map, it is first necessary to put the expression into what is referred to as *canonical* form: each term in the expression must contain each variable. So, for example, if we have Equation (20.3), we must first expand it into the full form of Equation (20.1) and then convert this to a map.

The labeling used in Figure 20.7d emphasizes the relationship between variables and the rows and columns of the map. Here the two rows embraced by the symbol A are those in which the variable A has the value 1; the rows not embraced by the symbol A are those in which A is 0; similarly for B, C, and D.

Once the map of a function is created, we can often write a simple algebraic expression for it by noting the arrangement of the 1s on the map. The principle is as follows. Any two squares that are adjacent differ in only one of the variables. If two adjacent squares both have an entry of one, then the corresponding product terms differ in only one variable. In such a case, the two terms can be merged by eliminating that variable. For example, in Figure 20.8a, the two adjacent squares correspond to the two terms $\overline{A}B\overline{C}D$ and $\overline{A}BCD$. Thus, the function expressed is

$$\overline{A}B\overline{C}D + \overline{A}BCD = \overline{A}BD$$

This process can be extended in several ways. First, the concept of adjacency can be extended to include wrapping around the edge of the map. Thus, the top square of a column is adjacent to the bottom square, and the leftmost square of a row is adjacent to the rightmost square. These conditions are illustrated in Figures 20.8b and c. Second, we can group not just 2 squares but 2^n adjacent squares (that is, 2, 4, 8, etc.). The next three examples in Figure 20.8 show groupings of 4 squares. Note that in this case, two of the variables can be eliminated. The last three examples show groupings of 8 squares, which allow three variables to be eliminated.

We can summarize the rules for simplification as follows:

1. Among the marked squares (squares with a 1), find those that belong to a unique largest block of 1, 2, 4, or 8 and circle those blocks.

2. Select additional blocks of marked squares that are as large as possible and as few in number as possible, but include every marked square at least once. The results may not be unique in some cases. For example, if a marked square combines with exactly two other squares, and there is no fourth marked square to complete a larger group, then there is a choice to be made as two which of the two groupings to choose. When you are circling groups, you are allowed to use the same 1 value more than once.

3. Continue to draw loops around single marked squares, or pairs of adjacent marked squares, or groups of four, eight, and so on in such a way that every marked square belongs to at least one loop; then use as few of these blocks as possible to include all marked squares.
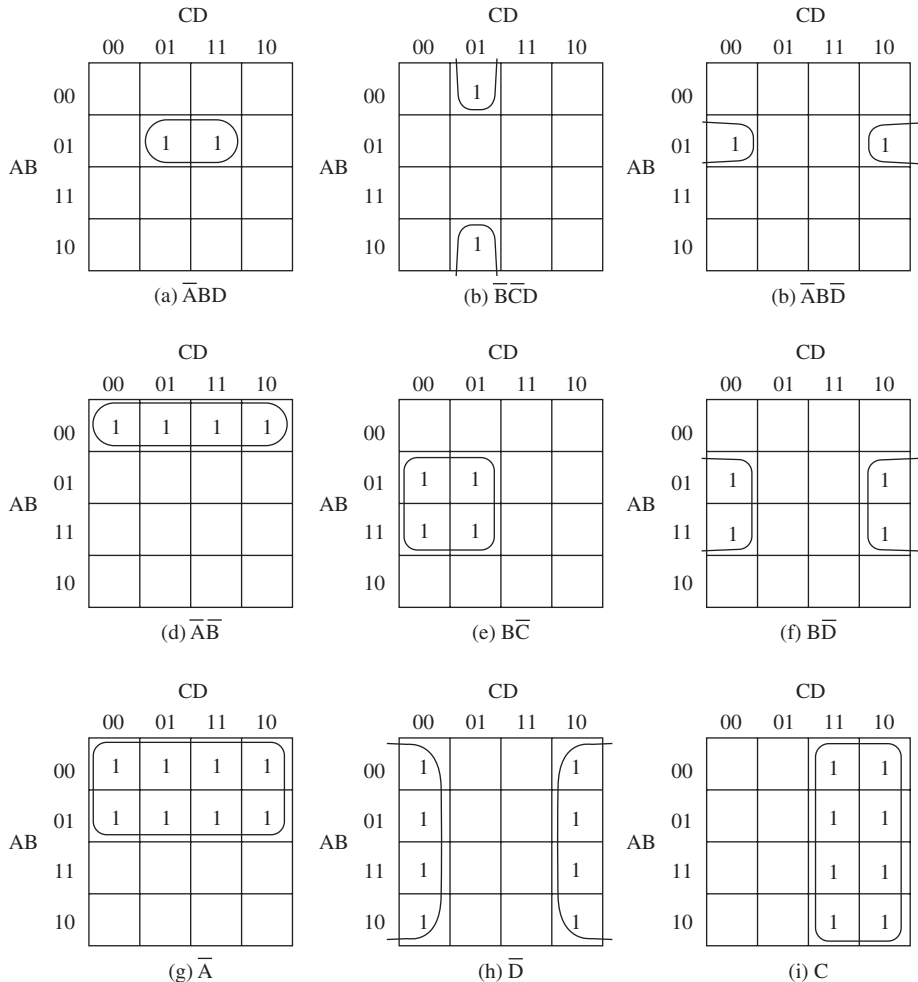
Figure 20.8   The Use of Karnaugh Maps

Figure 20.9a, based on Table 20.3, illustrates the simplification process. If any isolated 1s remain after the groupings, then each of these is circled as a group of 1s. Finally, before going from the map to a simplified Boolean expression, any group of 1s that is completely overlapped by other groups can be eliminated. This is shown in Figure 20.9b. In this case, the horizontal group is redundant and may be ignored in creating the Boolean expression.

One additional feature of Karnaugh maps needs to be mentioned. In some cases, certain combinations of values of variables never occur, and therefore the corresponding output never occurs. These are referred to as "don't care" conditions. For each such condition, the letter "d" is entered into the corresponding square of the map. In doing the grouping and simplification, each "d" can be treated as a 1 or 0, whichever leads to the simplest expression.
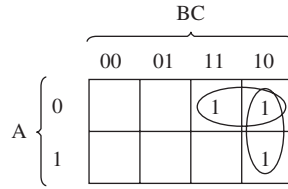
(a) F = $\overline{A}B + B\overline{C}$
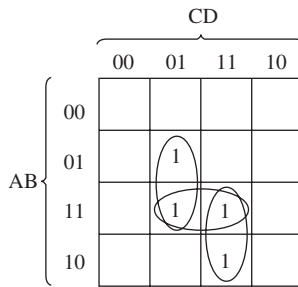
(b) F = $B\overline{C}D + ACD$

Figure 20.9   Overlapping Groups

An example, presented in [HAYE98], illustrates the points we have been discussing. We would like to develop the Boolean expressions for a circuit that adds 1 to a packed decimal digit. Recall from Section 9.2 that with packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way. Thus, 0 = 0000, 1 = 0001, ..., 8 = 1000, and 9 = 1001. The remaining 4-bit values, from 1010 to 1111, are not used. This code is also referred to as Binary Coded Decimal (BCD).

Table 20.4 shows the truth table for producing a 4-bit result that is one more than a 4-bit BCD input. The addition is modulo 10. Thus, 9 + 1 = 0. Also, note that six of the input codes produce "don't care" results, because those are not valid BCD inputs. Figure 20.10 shows the resulting Karnaugh maps for each of the output variables. The d squares are used to achieve the best possible groupings.

THE QUINE–McKLUSKEY METHOD For more than four variables, the Karnaugh map method becomes increasingly cumbersome. With five variables, two 16 × 16 maps are needed, with one map considered to be on top of the other in three dimensions to achieve adjacency. Six variables require the use of four 16 × 16 tables in four dimensions! An alternative approach is a tabular technique, referred to as the Quine–McKluskey method. The method is suitable for programming on a computer to give an automatic tool for producing minimized Boolean expressions.

The method is best explained by means of an example. Consider the following expression:

$$ABCD + AB\overline{C}D + AB\overline{C}\,\overline{D} + A\overline{B}CD + \overline{A}BCD + \overline{A}BC\overline{D} + \overline{A}B\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}D$$

Let us assume that this expression was derived from a truth table. We would like to produce a minimal expression suitable for implementation with gates.

Table 20.4 Truth Table for the One-Digit Packed Decimal Incrementer

	Input					Output			
Number	**A**	**B**	**C**	**D**	**Number**	**W**	**X**	**Y**	**Z**
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	2	0	0	1	0
2	0	0	1	0	3	0	0	1	1
3	0	0	1	1	4	0	1	0	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	6	0	1	1	0
6	0	1	1	0	7	0	1	1	1
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	9	1	0	0	1
9	1	0	0	1	0	0	0	0	0
	1	0	1	0		d	d	d	d
	1	0	1	1		d	d	d	d
Don't care condition	1	1	0	0		d	d	d	d
	1	1	0	1		d	d	d	d
	1	1	1	0		d	d	d	d
	1	1	1	1		d	d	d	d

The first step is to construct a table in which each row corresponds to one of the product terms of the expression. The terms are grouped according to the number of complemented variables. That is, we start with the term with no complements, if it exists, then all terms with one complement, and so on. Table 20.5 shows the list for our example expression, with horizontal lines used to indicate the grouping. For clarity,
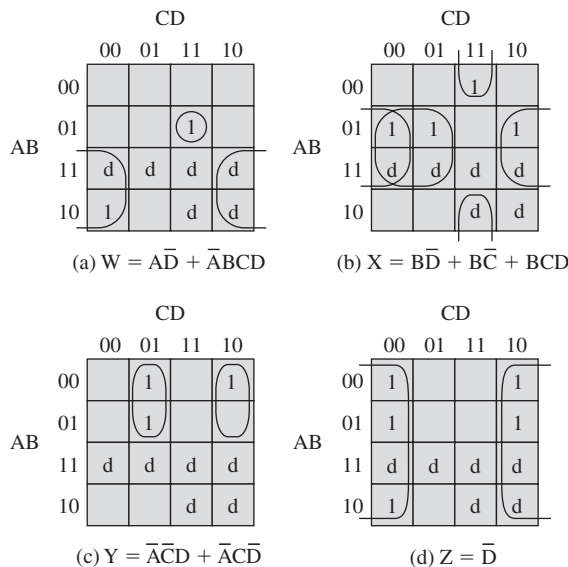

(a) $W = A\overline{D} + \overline{A}BCD$

(b) $X = B\overline{D} + B\overline{C} + BCD$

(c) $Y = \overline{A}\overline{C}D + \overline{A}C\overline{D}$

(d) $Z = \overline{D}$

Figure 20.10 Karnaugh Maps for the Incrementer

Table 20.5 First Stage of Quine-McKluskey Method
(for F = ABCD + AB$\overline{C}$D + ABC$\overline{D}$ + A$\overline{B}$CD + $\overline{A}$BCD + $\overline{A}$BC$\overline{D}$ + $\overline{A}$B$\overline{C}$D + $\overline{A}$ $\overline{B}$ $\overline{C}$D)

Product Term	Index	A	B	C	D	
$\overline{A}$ $\overline{B}$ $\overline{C}$D	1	0	0	0	1	✓
$\overline{A}$B$\overline{C}$D	5	0	1	0	1	✓
$\overline{A}$BC$\overline{D}$	6	0	1	1	0	✓
AB$\overline{C}$ $\overline{D}$	12	1	1	0	0	✓
$\overline{A}$BCD	7	0	1	1	1	✓
A$\overline{B}$CD	11	1	0	1	1	✓
AB$\overline{C}$D	13	1	1	0	1	✓
ABCD	15	1	1	1	1	✓

each term is represented by a 1 for each uncomplemented variable and a 0 for each complemented variable. Thus, we group terms according to the number of 1s they contain. The index column is simply the decimal equivalent and is useful in what follows.

The next step is to find all pairs of terms that differ in only one variable, that is, all pairs of terms that are the same except that one variable is 0 in one of the terms and 1 in the other. Because of the way in which we have grouped the terms, we can do this by starting with the first group and comparing each term of the first group with every term of the second group. Then compare each term of the second group with all of the terms of the third group, and so on. Whenever a match is found, place a check next to each term, combine the pair by eliminating the variable that differs in the two terms, and add that to a new list. Thus, for example, the terms $\overline{A}$BC$\overline{D}$ and $\overline{A}$BCD are combined to produce ABC. This process continues until the entire original table has been examined. The result is a new table with the following entries:

$\overline{A}$ $\overline{C}$D   AB$\overline{C}$ ABD $\sqrt{}$

$\phantom{\overline{A} \overline{C}D}$ B$\overline{C}$D$\sqrt{}$ ACD

$\phantom{\overline{A} \overline{C}D}$ $\overline{A}$BC   BCD $\sqrt{}$

$\phantom{\overline{A} \overline{C}D}$ $\overline{A}$BD$\sqrt{}$

The new table is organized into groups, as indicated, in the same fashion as the first table. The second table is then processed in the same manner as the first. That is, terms that differ in only one variable are checked and a new term produced for a third table. In this example, the third table that is produced contains only one term: BD.

In general, the process would proceed through successive tables until a table with no matches was produced. In this case, this has involved three tables.

Once the process just described is completed, we have eliminated many of the possible terms of the expression. Those terms that have not been eliminated are used to construct a matrix, as illustrated in Table 20.6. Each row of the matrix corresponds to one of the terms that have not been eliminated (has no check) in any of the tables used so far. Each column corresponds to one of the terms in the original expression. An X is placed at each intersection of a row and a column such that the row element is "compatible" with the column element. That is, the variables present in the row element have the same value as the variables present in the column element. Next, circle each X

Table 20.6 Last Stage of Quine-McKluskey Method
(for F = ABCD + AB$\overline{C}$D + AB$\overline{C}$ $\overline{D}$ + A$\overline{B}$CD + $\overline{A}$BCD + $\overline{A}$BC$\overline{D}$ + $\overline{A}$B$\overline{C}$D + $\overline{A}$ $\overline{B}$ $\overline{C}$D)

	ABCD	**AB$\overline{C}$D**	**AB$\overline{C}$ $\overline{D}$**	**A$\overline{B}$CD**	**$\overline{A}$BCD**	**$\overline{A}$BC$\overline{D}$**	**$\overline{A}$B$\overline{C}$D**	**$\overline{A}$ $\overline{B}$ $\overline{C}$D**
BD	X	X			X		X	
$\overline{A}$ $\overline{C}$D							☒	⊗
$\overline{A}$BC					☒	⊗		
AB$\overline{C}$		☒	⊗					
ACD	☒			⊗				

that is alone in a column. Then place a square around each X in any row in which there is a circled X. If every column now has either a squared or a circled X, then we are done, and those row elements whose Xs have been marked constitute the minimal expression. Thus, in our example, the final expression is

$$AB\overline{C} + ACD + \overline{A}BC + \overline{A}\,\overline{C}D$$

In cases in which some columns have neither a circle nor a square, additional processing is required. Essentially, we keep adding row elements until all columns are covered.

Let us summarize the Quine–McKluskey method to try to justify intuitively why it works. The first phase of the operation is reasonably straightforward. The process eliminates unneeded variables in product terms. Thus, the expression ABC + AB$\overline{C}$ is equivalent to AB, because

$$ABC + AB\overline{C} = AB(C + \overline{C}) = AB$$

After the elimination of variables, we are left with an expression that is clearly equivalent to the original expression. However, there may be redundant terms in this expression, just as we found redundant groupings in Karnaugh maps. The matrix layout assures that each term in the original expression is covered and does so in a way that minimizes the number of terms in the final expression.

NAND AND NOR IMPLEMENTATIONS Another consideration in the implementation of Boolean functions concerns the types of gates used. It is sometimes desirable to implement a Boolean function solely with NAND gates or solely with NOR gates. Although this may not be the minimum-gate implementation, it has the advantage of regularity, which can simplify the manufacturing process. Consider again Equation (20.3):

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

Because the complement of the complement of a value is just the original value,

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

Applying DeMorgan's theorem,

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

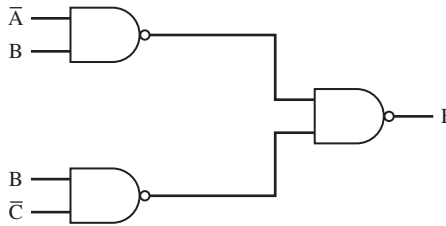which has three NAND forms, as illustrated in Figure 20.11.

Figure 20.11 NAND Implementation of Table 20.3

Multiplexers

The multiplexer connects multiple inputs to a single output. At any time, one of the inputs is selected to be passed to the output. A general block diagram representation is shown in Figure 20.12. This represents a 4-to-1 multiplexer. There are four input lines, labeled D0, D1, D2, and D3. One of these lines is selected to provide the output signal F. To select one of the four possible inputs, a 2-bit selection code is needed, and this is implemented as two select lines labeled S1 and S2.

An example 4-to-1 multiplexer is defined by the truth table in Table 20.7. This is a simplified form of a truth table. Instead of showing all possible combinations of input variables, it shows the output as data from line D0, D1, D2, or D3. Figure 20.13 shows an implementation using AND, OR, and NOT gates. S1 and S2 are connected to the AND gates in such a way that, for any combination of S1 and S2, three of the AND gates will output 0. The fourth AND gate will output the value of the selected line, which is either 0 or 1. Thus, three of the inputs to the OR gate are always 0, and the output of the OR gate will equal the value of the selected input gate. Using this regular organization, it is easy to construct multiplexers of size 8-to-1, 16-to-1, and so on.

Multiplexers are used in digital circuits to control signal and data routing. An example is the loading of the program counter (PC). The value to be loaded into the program counter may come from one of several different sources:

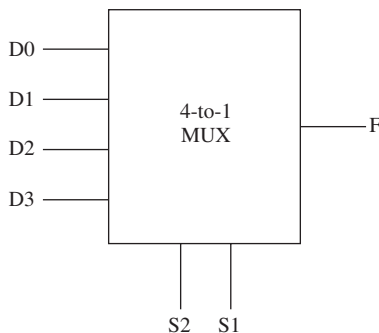- A binary counter, if the PC is to be incremented for the next instruction

Figure 20.12 4-to-1 Multiplexer Representation

Table 20.7   4-to-1 Multiplexer Truth Table

S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

- The instruction register, if a branch instruction using a direct address has just been executed
- The output of the ALU, if the branch instruction specifies the address using a displacement mode

These various inputs could be connected to the input lines of a multiplexer, with the PC connected to the output line. The select lines determine which value is loaded into the PC. Because the PC contains multiple bits, multiple multiplexers are used, one per bit. Figure 20.14 illustrates this for 16-bit addresses.

Decoders

A decoder is a combinational circuit with a number of output lines, only one of which is asserted at any time, dependent on the pattern of input lines. In general, a
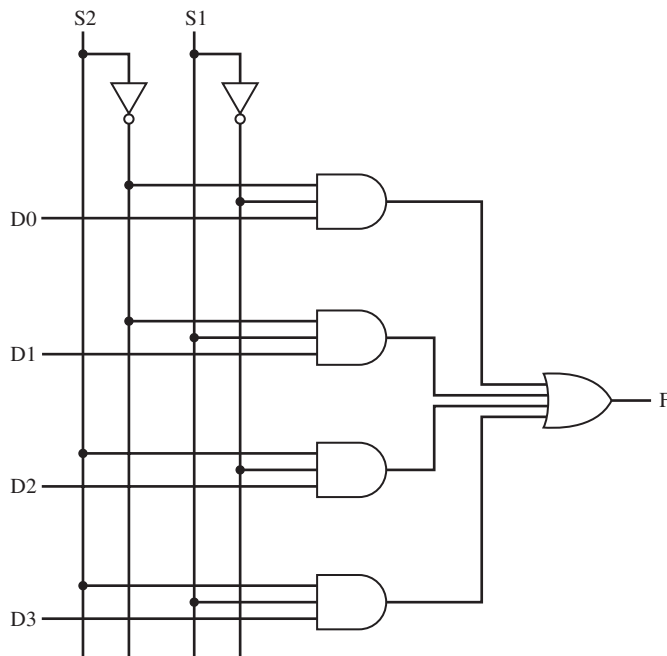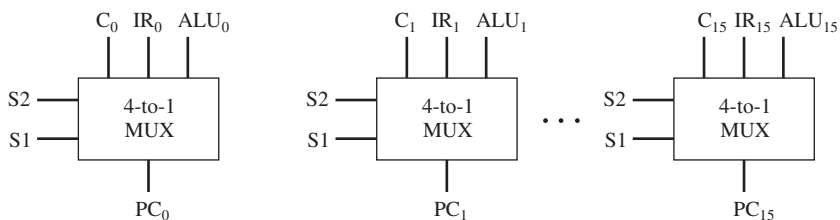

Figure 20.13   Multiplexer Implementation

Figure 20.14 Multiplexer Input to Program Counter

decoder has n inputs and 2^n outputs. Figure 20.15 shows a decoder with three inputs and eight outputs.

Decoders find many uses in digital computers. One example is address decoding. Suppose we wish to construct a 1K-byte memory using four 256×8–bit RAM chips. We want a single unified address space, which can be broken down as follows:

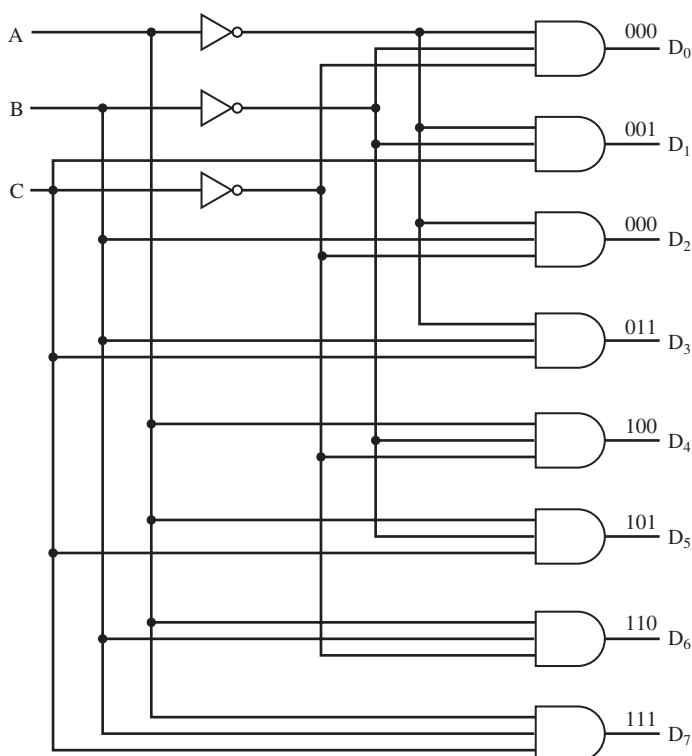| Address	Chip
0000–00FF	0
0100–01FF	1
0200–02FF	2
0300–03FF	3

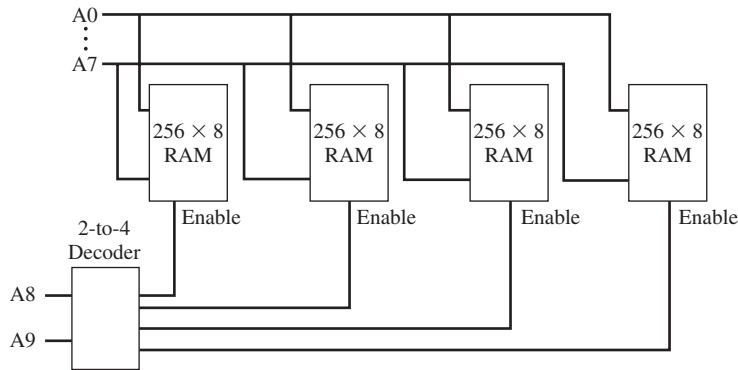**Figure 20.15**   Decoder with 3 Inputs and $2^3 = 8$ Outputs

Figure 20.16 Address Decoding

Each chip requires 8 address lines, and these are supplied by the lower-order 8 bits of the address. The higher-order 2 bits of the 10-bit address are used to select one of the four RAM chips. For this purpose, a 2-to-4 decoder is used whose output enables one of the four chips, as shown in Figure 20.16.

With an additional input line, a decoder can be used as a demultiplexer. The demultiplexer performs the inverse function of a multiplexer; it connects a single input to one of several outputs. This is shown in Figure 20.17. As before, n inputs are decoded to produce a single one of 2^n outputs. All of the 2^n output lines are ANDed with a data input line. Thus, the n inputs act as an address to select a particular output line, and the value on the data input line (0 or 1) is routed to that output line.

The configuration in Figure 20.17 can be viewed in another way. Change the label on the new line from *Data Input* to *Enable*. This allows for the control of the timing of the decoder. The decoded output appears only when the encoded input is present *and* the enable line has a value of 1.

Read–Only Memory

Combinational circuits are often referred to as "memoryless" circuits, because their output depends only on their current input and no history of prior inputs is retained.
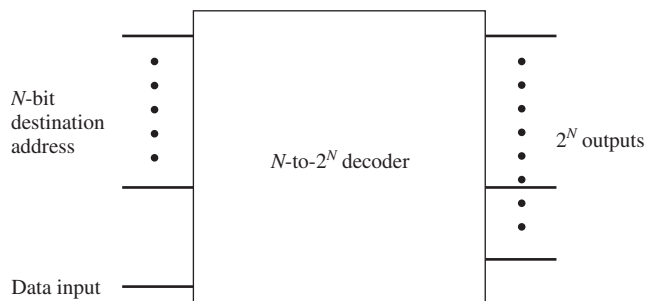

Figure 20.17 Implementation of a Demultiplexer Using a Decorder

Table 20.8   Truth Table for a ROM

Input				Output			
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

However, there is one sort of memory that is implemented with combinational circuits, namely *read-only memory* (ROM).

Recall that a ROM is a memory unit that performs only the read operation. This implies that the binary information stored in a ROM is permanent and was created during the fabrication process. Thus, a given input to the ROM (address lines) always produces the same output (data lines). Because the outputs are a function only of the present inputs, the ROM is in fact a combinational circuit.

A ROM can be implemented with a decoder and a set of OR gates. As an example, consider Table 20.8. This can be viewed as a truth table with four inputs and four outputs. For each of the 16 possible input values, the corresponding set of values of the outputs is shown. It can also be viewed as defining the contents of a 64-bit ROM consisting of 16 words of 4 bits each. The four inputs specify an address, and the four outputs specify the contents of the location specified by the address. Figure 20.18 shows how this memory could be implemented using a 4-to-16 decoder and four OR gates. As with the PLA, a regular organization is used, and the interconnections are made to reflect the desired result.

Adders

So far, we have seen how interconnected gates can be used to implement such functions as the routing of signals, decoding, and ROM. One essential area not yet addressed is that of arithmetic. In this brief overview, we will content ourselves with looking at the addition function.
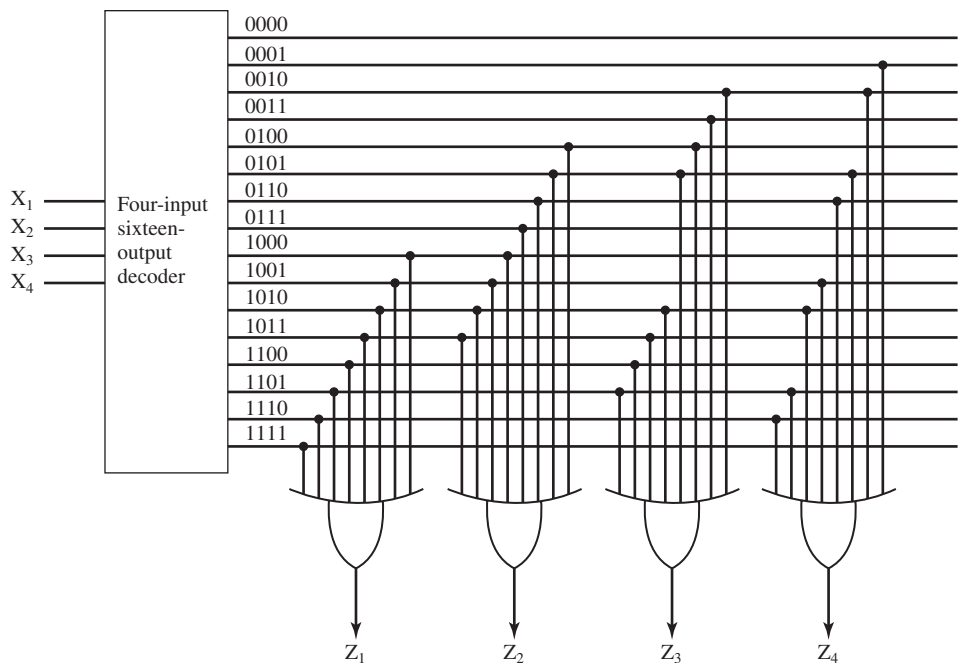
Figure 20.18 A 64-Bit ROM

Binary addition differs from Boolean algebra in that the result includes a carry term. Thus,

0	0	1	1
+ 0	+ 1	+ 0	+ 1
0	1	1	10

However, addition can still be dealt with in Boolean terms. In Table 20.9a, we show the logic for adding two input bits to produce a 1-bit sum and a carry bit. This truth table

Table 20.9 Binary Addition Truth Tables

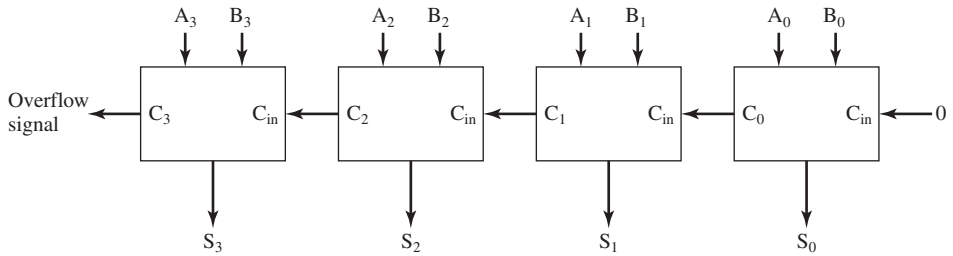| (a) Single-Bit Addition				(b) Addition with Carry Input				
A	**B**	**Sum**	**Carry**	C_{in}	**A**	**B**	**Sum**	C_{out}
0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	1	0
1	0	1	0	0	1	0	1	0
1	1	0	1	0	1	1	0	1
				1	0	0	1	0
				1	0	1	0	1
				1	1	0	0	1
				1	1	1	1	1

Figure 20.19   4-Bit Adder

could easily be implemented in digital logic. However, we are not interested in per-forming addition on just a single pair of bits. Rather, we wish to add two *n*-bit num-bers. This can be done by putting together a set of adders so that the carry from one adder is provided as input to the next. A 4-bit adder is depicted in Figure 20.19.

For a multiple-bit adder to work, each of the single-bit adders must have three inputs, including the carry from the next-lower-order adder. The revised truth table appears in Table 20.9b. The two outputs can be expressed:

$$\text{Sum} = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{B}\overline{C}$$

$$\text{Carry} = AB + AC + BC$$

Figure 20.20 is an implementation using AND, OR, and NOT gates.
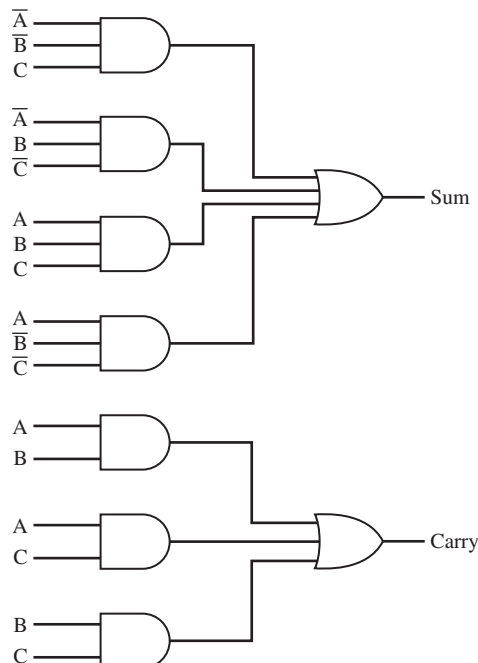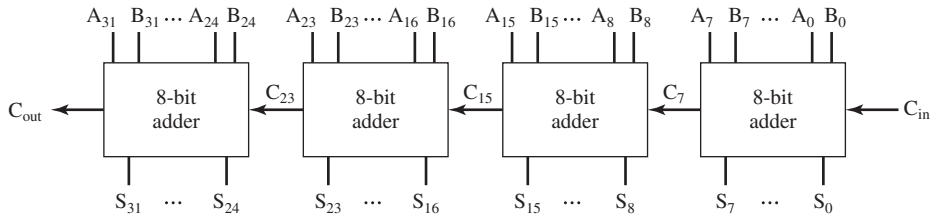

Figure 20.20   Implementation of an Adder

Figure 20.21 Construction of a 32-Bit Adder Using 8-Bit Adders

Thus we have the necessary logic to implement a multiple-bit adder such as shown in Figure 20.21. Note that because the output from each adder depends on the carry from the previous adder, there is an increasing delay from the least significant to the most significant bit. Each single-bit adder experiences a certain amount of gate delay, and this gate delay accumulates. For larger adders, the accumulated delay can become unacceptably high.

If the carry values could be determined without having to ripple through all the previous stages, then each single-bit adder could function independently, and delay would not accumulate. This can be achieved with an approach known as *carry lookahead.* Let us look again at the 4-bit adder to explain this approach.

We would like to come up with an expression that specifies the carry input to any stage of the adder without reference to previous carry values. We have

$$C_0 = A_0B_0 \tag{20.4}$$
$$C_1 = A_1B_1 + A_1A_0B_0 + B_1A_0B_0 \tag{20.5}$$

Following the same procedure, we get

$$C_2 = A_2B_2 + A_2A_1B_1 + A_2A_1A_0B_0 + A_2B_1A_0B_0 + B_2A_1B_1 \\ + B_2A_1A_0B_0 + B_2B_1A_0B_0$$

This process can be repeated for arbitrarily long adders. Each carry term can be expressed in SOP form as a function only of the original inputs, with no dependence on the carries. Thus, only two levels of gate delay occur regardless of the length of the adder.

For long numbers, this approach becomes excessively complicated. Evaluating the expression for the most significant bit of an n-bit adder requires an OR gate with $n - 1$ inputs and n AND gates with from 2 to $n + 1$ inputs. Accordingly, full carry lookahead is typically done only 4 to 8 bits at a time. Figure 20.21 shows how a 32-bit adder can be constructed out of four 8-bit adders. In this case, the carry must ripple through the four 8-bit adders, but this will be substantially quicker than a ripple through thirty-two 1-bit adders.

20.4 SEQUENTIAL CIRCUITS

Combinational circuits implement the essential functions of a digital computer. However, except for the special case of ROM, they provide no memory or state information, elements also essential to the operation of a digital computer. For the

latter purposes, a more complex form of digital logic circuit is used: the sequential circuit. The current output of a sequential circuit depends not only on the current input, but also on the past history of inputs. Another and generally more useful way to view it is that the current output of a sequential circuit depends on the current input and the current state of that circuit.

In this section, we examine some simple but useful examples of sequential circuits. As will be seen, the sequential circuit makes use of combinational circuits.

Flip–Flops

The simplest form of sequential circuit is the flip-flop. There are a variety of flip-flops, all of which share two properties:

- The flip-flop is a bistable device. It exists in one of two states and, in the absence of input, remains in that state. Thus, the flip-flop can function as a 1-bit memory.
- The flip-flop has two outputs, which are always the complements of each other. These are generally labeled Q and $\overline{Q}$.

THE S–R LATCH Figure 20.22 shows a common configuration known as the S–R flip-flop or S–R latch. The circuit has two inputs, S (Set) and R (Reset), and two outputs, Q and $\overline{Q}$, and consists of two NOR gates connected in a feedback arrangement.

First, let us show that the circuit is bistable. Assume that both S and R are 0 and that Q is 0. The inputs to the lower NOR gate are Q = 0 and S = 0. Thus, the output $\overline{Q}$ = 1 means that the inputs to the upper NOR gate are $\overline{Q}$ = 1 and R = 0, which has the output Q = 0. Thus, the state of the circuit is internally consistent and remains stable as long as S = R = 0. A similar line of reasoning shows that the state Q = 1, $\overline{Q}$ = 0 is also stable for R = S = 0.

Thus, this circuit can function as a 1-bit memory. We can view the output Q as the "value" of the bit. The inputs S and R serve to write the values 1 and 0, respectively, into memory. To see this, consider the state Q = 0, $\overline{Q}$ = 1, S = 0, R = 0. Suppose that S changes to the value 1. Now the inputs to the lower NOR gate are S = 1, Q = 0. After some time delay Δt, the output of the lower NOR gate will be $\overline{Q}$ = 0
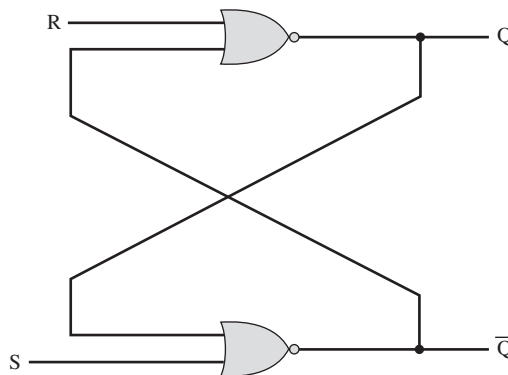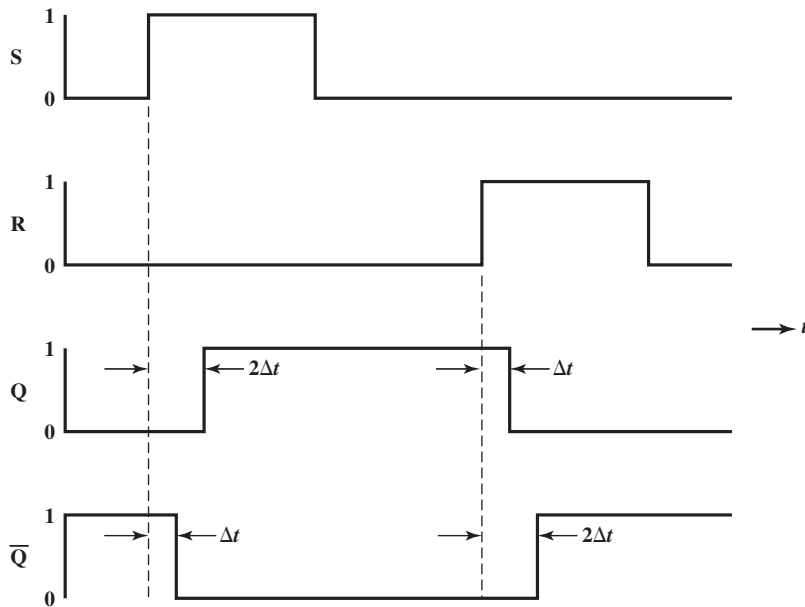

Figure 20.22 The S–R Latch Implemented with NOR Gates

Figure 20.23   NOR S–R Latch timing Diagram

(see Figure 20.23). So, at this point in time, the inputs to the upper NOR gate become R = 0, $\overline{Q}$ = 0. After another gate delay of Δt, the output Q becomes 1. This is again a stable state. The inputs to the lower gate are now S = 1, Q = 1, which maintain the output Q = 0. As long as S = 1 and R = 0, the outputs will remain Q = 1, $\overline{Q}$ = 0. Furthermore, if S returns to 0, the outputs will remain unchanged.

The R output performs the opposite function. When R goes to 1, it forces Q = 0, $\overline{Q}$ = 1 regardless of the previous state of Q and $\overline{Q}$. Again, a time delay of $2\Delta t$ occurs before the final state is established (Figure 20.23).

The S–R latch can be defined with a table similar to a truth table, called a *characteristic table*, which shows the next state or states of a sequential circuit as a function of current states and inputs. In the case of the S–R latch, the state can be defined by the value of Q. Table 20.10a shows the resulting characteristic table. Observe that the inputs S = 1, R = 1 are not allowed, because these would produce an inconsistent output (both Q and $\overline{Q}$ equal 0). The table can be expressed more compactly, as in Table 20.10b. An illustration of the behavior of the S–R latch is shown in Table 20.10c.

CLOCKED S–R FLIP–FLOP The output of the S–R latch changes, after a brief time delay, in response to a change in the input. This is referred to as asynchronous operation. More typically, events in the digital computer are synchronized to a clock pulse, so that changes occur only when a clock pulse occurs. Figure 20.24 shows this arrangement. This device is referred to as a *clocked S–R flip-flop*. Note that the R and S inputs are passed to the NOR gates only during the clock pulse.

D FLIP–FLOP One problem with S–R flip-flop is that the condition R = 1, S = 1 must be avoided. One way to do this is to allow just a single input. The D flip-flop accomplishes this. Figure 20.25 shows a gate implementation and the characteristic

Table 20.10 The S–R Latch

(a) Characteristic Table		
Current Inputs	Current State	Next State
SR	Q_n	Q_{n+1}
00	0	0
00	1	1
01	0	0
01	1	0
10	0	1
10	1	1
11	0	—
11	1	—

(b) Simplified Characteristic Table		
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	—

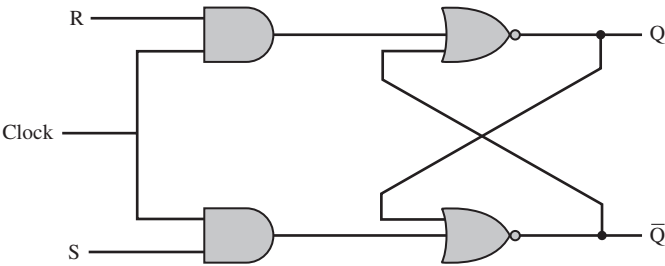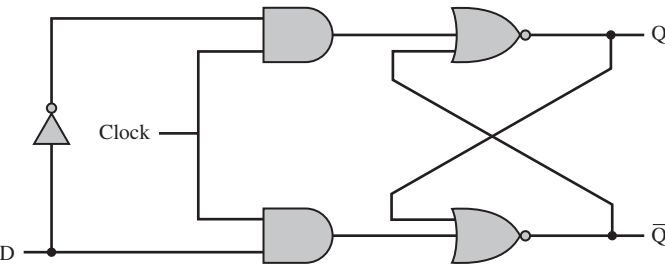| (c) Response to Series of Inputs										
t	0	1	2	3	4	5	6	7	8	9
S	1	0	0	0	0	0	0	0	1	0
R	0	0	0	1	0	0	1	0	0	0
Q_{n+1}	1	1	1	0	0	0	0	0	1	1

Figure 20.24 Clocked S–R Flip Flop
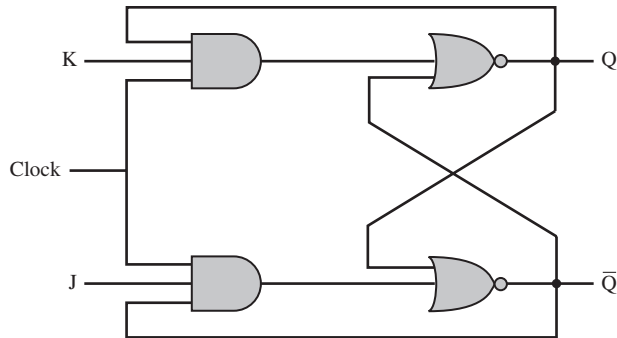

Figure 20.25 D Flip Flop

Figure 20.26 J–K Flip Flop

table of the D flip-flop. By using an inverter, the nonclock inputs to the two AND gates are guaranteed to be the opposite of each other.

The D flip-flop is sometimes referred to as the data flip-flop because it is, in effect, storage for one bit of data. The output of the D flip-flop is always equal to the most recent value applied to the input. Hence, it remembers and produces the last input. It is also referred to as the delay flip-flop, because it delays a 0 or 1 applied to its input for a single clock pulse. We can capture the logic of the D flip-flop in the following truth table:

D	Q_{n+1}
0	0
1	1

J–K FLIP–FLOP Another useful flip-flop is the J–K flip-flop. Like the S–R flip-flop, it has two inputs. However, in this case all possible combinations of input values are valid. Figure 20.26 shows a gate implementation of the J–K flip-flop, and Figure 20.27 shows its characteristic table (along with those for the S–R and D flip-flops). Note that the first three combinations are the same as for the S–R flip-flop. With no input asserted, the output is stable. If only the J input is asserted, the result is a set function, causing the output to be 1; if only the K in put is asserted, the result is a reset function, causing the output to be 0. When both J and K are 1, the function performed is referred to as the toggle function: the output is reversed. Thus, if Q is 1 and 1 is applied to J and K, then Q becomes 0. The reader should verify that the implementation of Figure 20.26 produces this characteristic function.

Registers

As an example of the use of flip-flops, let us first examine one of the essential elements of the CPU: the register. As we know, a register is a digital circuit used within the CPU to store one or more bits of data. Two basic types of registers are commonly used: parallel registers and shift registers.

PARALLEL REGISTERS A parallel register consists of a set of 1-bit memories that can be read or written simultaneously. It is used to store data. The registers that we have discussed throughout this book are parallel registers.
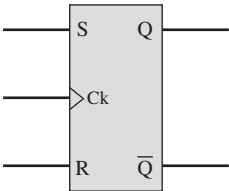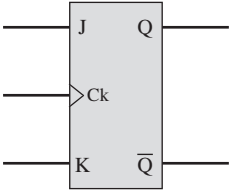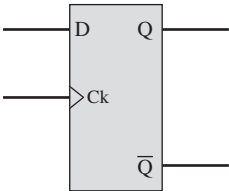
Name	Graphical Symbol	Truth Table
S–R	S — Q; Ck; R — $\overline{Q}$	S R Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 –
J–K	J — Q; Ck; K — $\overline{Q}$	J K Q_{n+1} 0 0 Q_n 0 1 0 1 0 1 1 1 $\overline{Q_n}$
D	D — Q; Ck; $\overline{Q}$	D Q_{n+1} 0 0 1 1

Figure 20.27   Basic Flip-Flops

The 8-bit register of Figure 20.28 illustrates the operation of a parallel register using D flip-flops. A control signal, labeled *load*, controls writing into the register from signal lines, D11 through D18. These lines might be the output of multiplexers, so that data from a variety of sources can be loaded into the register.
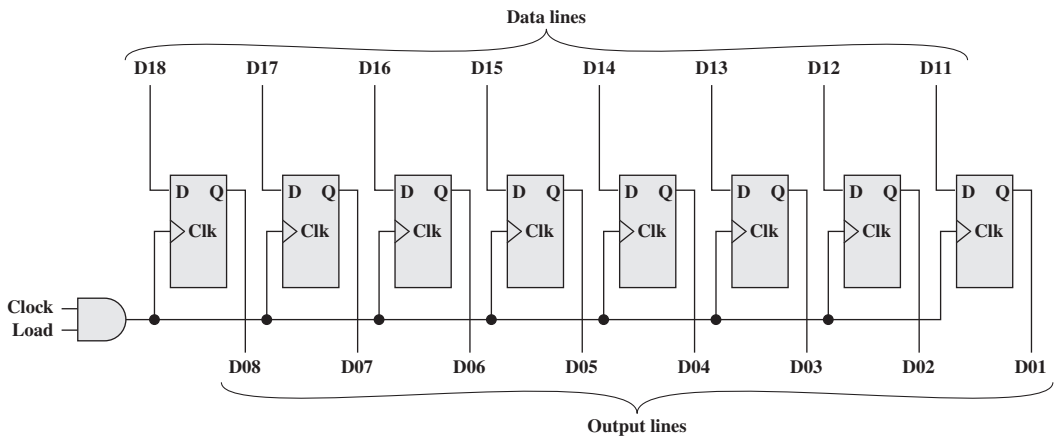

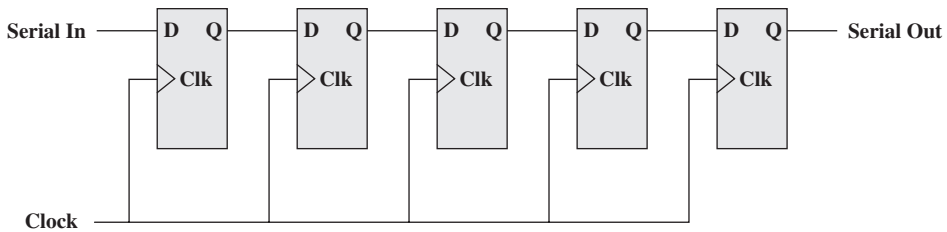**Figure 20.28**   8-Bit Parallel Register

Figure 20.29 5-Bit Shift Register

SHIFT REGISTER A shift register accepts and/or transfers information serially. Consider, for example, Figure 20.29, which shows a 5-bit shift register constructed from clocked D flip-flops. Data are input only to the leftmost flip-flop. With each clock pulse, data are shifted to the right one position, and the rightmost bit is transferred out.

Shift registers can be used to interface to serial I/O devices. In addition, they can be used within the ALU to perform logical shift and rotate functions. In this latter capacity, they need to be equipped with parallel read/write circuitry as well as serial.

Counters

Another useful category of sequential circuit is the counter. A counter is a register whose value is easily incremented by 1 modulo the capacity of the register; that is, after the maximum value is achieved the next increment sets the counter value to 0. Thus, a register made up of n flip-flops can count up to $2^n - 1$. An example of a counter in the CPU is the program counter.

Counters can be designated as asynchronous or synchronous, depending on the way in which they operate. Asynchronous counters are relatively slow because the output of one flip-flop triggers a change in the status of the next flip-flop. In a synchronous counter, all of the flip-flops change state at the same time. Because the latter type is much faster, it is the kind used in CPUs. However, it is useful to begin the discussion with a description of an asynchronous counter.

RIPPLE COUNTER An asynchronous counter is also referred to as a ripple counter, because the change that occurs to increment the counter starts at one end and "ripples" through to the other end. Figure 20.30 shows an implementation of a 4-bit counter using J–K flip-flops, together with a timing diagram that illustrates its behavior. The timing diagram is idealized in that it does not show the propagation delay that occurs as the signals move down the series of flip-flops. The output of the leftmost flip-flop (Q_0) is the least significant bit. The design could clearly be extended to an arbitrary number of bits by cascading more flip-flops.

In the illustrated implementation, the counter is incremented with each clock pulse. The J and K inputs to each flip-flop are held at a constant 1. This means that, when there is a clock pulse, the output at Q will be inverted (1 to 0; 0 to 1). Note that the change in state is shown as occurring with the falling edge of the clock pulse; this is known as an edge-triggered flip-flop. Using flip-flops that respond to the transition in a clock pulse rather than the pulse itself provides better timing control in complex circuits. If one looks at patterns of output for this counter, it can be seen that it cycles through $0000, 0001, \ldots, 1110, 1111, 0000$, and so on.
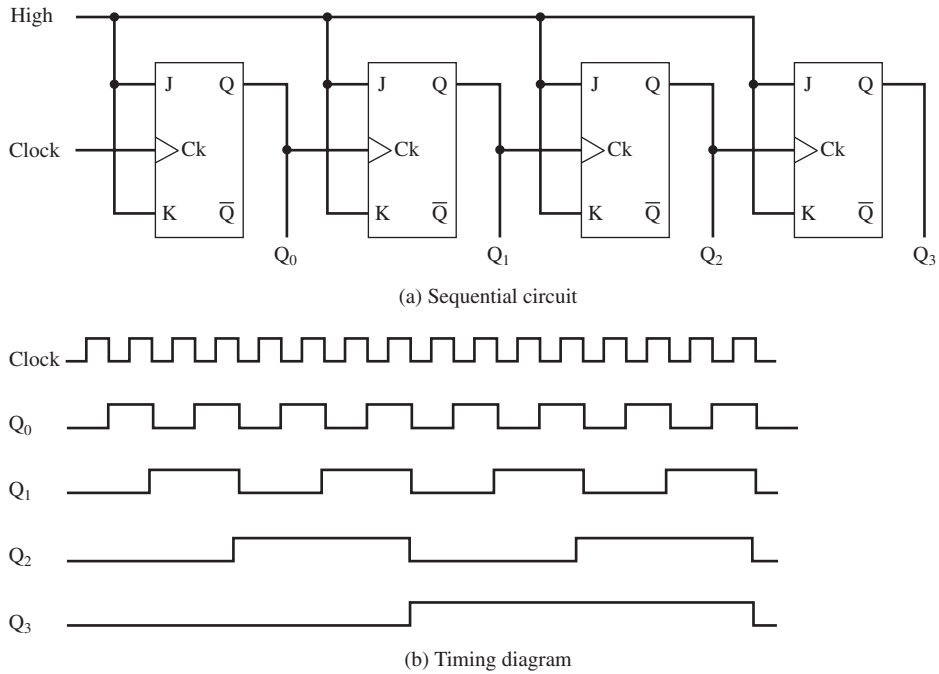
(a) Sequential circuit

(b) Timing diagram

Figure 20.30 Ripple Counter

SYNCHRONOUS COUNTERS The ripple counter has the disadvantage of the delay involved in changing value, which is proportional to the length of the counter. To overcome this disadvantage, CPUs make use of synchronous counters, in which all of the flip-flops of the counter change at the same time. In this subsection, we present a design for a 3-bit synchronous counter. In doing so, we illustrate some basic concepts in the design of a synchronous circuit.

For a 3-bit counter, three flip-flops will be needed. Let us use J–K flip-flops. Label the uncomplemented output of the three flip-flops A, B, C respectively, with C representing the least significant bit. The first step is to construct a truth table that relates the J–K inputs and outputs, to allow us to design the overall circuit. Such a truth table is shown in Figure 20.31a. The first three columns show the possible combinations of outputs A, B, and C. They are listed in the order that they will appear as the counter is incremented. Each row lists the current value of A, B, C and the inputs to the three flip-flops that will be required to reach the next value of A, B, C.

To understand the way in which the truth table of Figure 20.31a is constructed, it may be helpful to recast the characteristic table for the J–K flip-flop. Recall that this table was presented as follows:

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_{n+1}$

C	B	A	Jc	Kc	Jb	Kb	Ja	Ka
0	0	0	0	d	0	d	1	d
0	0	1	0	d	1	d	d	1
0	1	0	0	d	d	0	1	d
0	1	1	1	d	d	1	d	1
1	0	0	d	0	0	d	1	d
1	0	1	d	0	1	d	d	1
1	1	0	d	0	d	0	1	d
1	1	1	d	1	d	1	d	1

(a) Truth table

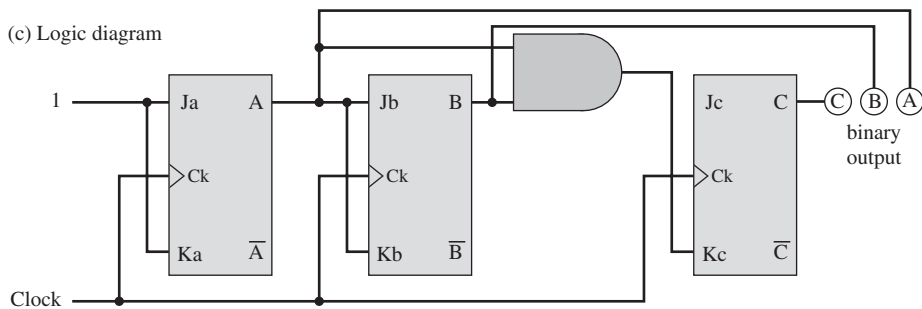(b) Karnaugh maps

(c) Logic diagram

Figure 20.31 Design of a Synchronous Counter

In this form, the table shows the effect that the J and K inputs have on the output. Now consider the following organization of the same information:

Q_n	J	K	Q_{n+1}
0	0	d	0
0	1	d	1
1	d	1	0
1	d	0	1

In this form, the table provides the value of the next output when the inputs and the present output are known. This is exactly the information needed to design the counter or, indeed, any sequential circuit. In this form, the table is referred to as an **excitation table**.

Let us return to Figure 20.31a. Consider the first row. We want the value of A to remain 0, the value of B to remain 0, and the value of C to go from 0 to 1 with the next application of a clock pulse. The excitation table shows that to maintain an output of 0, we must have inputs of J = 0 and don't care for K. To effect a transition from 0 to 1, the inputs must be J = 1 and K = d. These values are shown in the first row of the table. By similar reasoning, the remainder of the table can be filled in.

Having constructed the truth table of Figure 20.31a, we see that the table shows the required values of all of the J and K inputs as functions of the current values of A, B, and C. With the aid of Karnaugh maps, we can develop Boolean expressions for these six functions. This is shown in part b of the figure. For example, the Karnaugh map for the variable Ja (the J input to the flip-flop that produces the A output) yields the expression Ja = BC. When all six expressions are derived, it is a straightforward matter to design the actual circuit, as shown in part c of the figure.

20.5 PROGRAMMABLE LOGIC DEVICES

Thus far, we have treated individual gates as building blocks, from which arbitrary functions can be realized. The designer could pursue a strategy of minimizing the number of gates to be used by manipulating the corresponding Boolean expressions.

As the level of integration provided by integrated circuits increases, other considerations apply. Early integrated circuits, using small-scale integration (SSI), provided from one to ten gates on a chip. Each gate is treated independently, in the building-block approach described so far. Figure 20.32 is an example of some SSI chips. To construct a logic function, a number of these chips are laid out on a printed circuit board and the appropriate pin interconnections are made.

Increasing levels of integration made it possible to put more gates on a chip and to make gate interconnections on the chip as well. This yields the advantages of decreased cost, decreased size, and increased speed (because on-chip delays are of shorter duration than off-chip delays). A design problem arises, however. For each particular logic function or set of functions, the layout of gates and interconnections on the chip must be designed. The cost and time involved in such custom chip design is high. Thus, it becomes attractive to develop a general-purpose chip that can be readily adapted to specific purposes. This is the intent of the *programmable logic device* (PLD).

There are a number of different types of PLDs in commercial use. Table 20.11 lists some of the key terms and defines some of the most important types. In this section, we first look at one of the simplest such devices, the programmable logic array (PLA) and then introduce perhaps the most important and widely used type of PLD, the field-programmable gate array (FPGA).
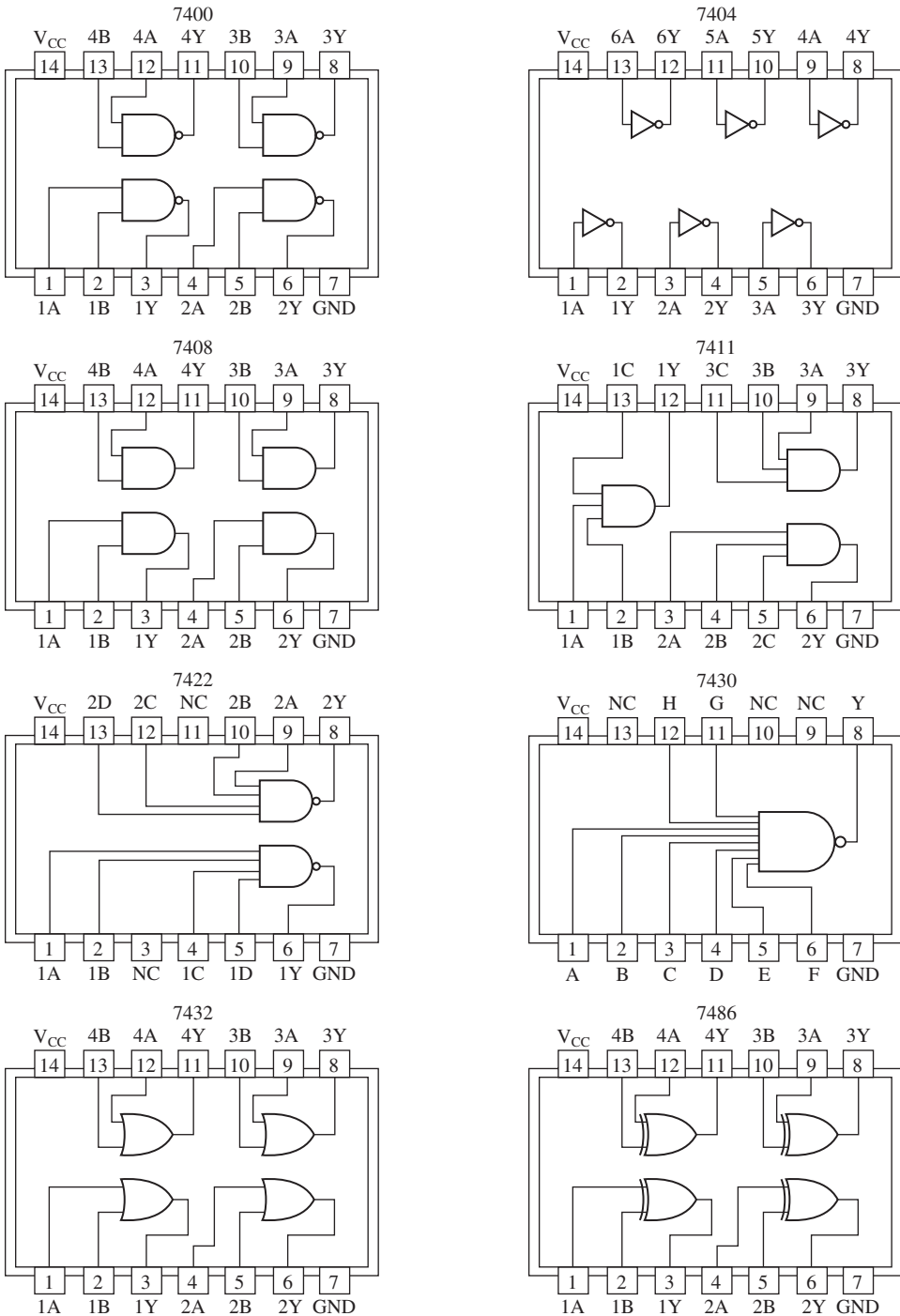
Figure 20.32 Some SSI Chips. Pin layouts from *The TTL Data Book for Design Engineers,* copyright © 1976 Texas Instrument Incorporated.

Table 20.11 PLD Terminology

> **Programmable Logic Device (PLD)**
>
> A general term that refers to any type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs. Programming of such a device often involves placing the chip into a special programming unit, but some chips can also be configured "in-system." Also referred to as a field-programmable device (FPD).
>
> **Programmable Logic Array (PLA)**
>
> A relatively small PLD that contains two levels of logic, an AND-plane and an OR-plane, where both levels are programmable.
>
> **Programmable Array Logic (PAL)**
>
> A relatively small PLD that has a programmable AND-plane followed by a fixed OR-plane.
>
> **Simple PLD (SPLD)**
>
> A PLA or PAL.
>
> **Complex PLD (CPLD)**
>
> A more complex PLD that consists of an arrangement of multiple SPLD-like blocks on a single chip.
>
> **Field-Programmable Gate Array (FPGA)**
>
> A PLD featuring a general structure that allows very high logic capacity. Whereas CPLDs feature logic resources with a wide number of inputs (AND planes), FPGAs offer more narrow logic resources. FPGAs also offer a higher ratio of flip-flops to logic resources than do CPLDs.
>
> **Logic Block**
>
> A relatively small circuit block that is replicated in an array in an FPD. When a circuit is implemented in an FPD, it is first decomposed into smaller sub-circuits that can each be mapped into a logic block. The term logic block is mostly used in the context of FPGAs, but it could also refer to a block of circuitry in a CPLD.
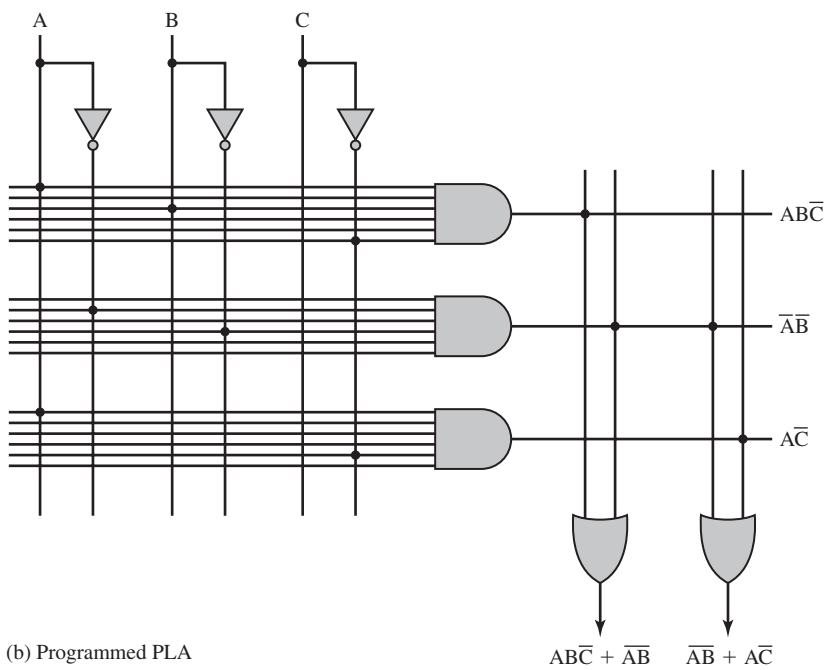
Programmable Logic Array

The PLA is based on the fact that any Boolean function (truth table) can be expressed in a sum-of-products (SOP) form, as we have seen. The PLA consists of a regular arrangement of NOT, AND, and OR gates on a chip. Each chip input is passed through a NOT gate so that each input and its complement are available to each AND gate. The output of each AND gate is available to each OR gate, and the output of each OR gate is a chip output. By making the appropriate connections, arbitrary SOP expressions can be implemented.

Figure 20.33a shows a PLA with three inputs, eight gates, and two outputs. On the left is a programmable AND array. The AND array is programmed by establishing a connection between any PLA input or its negation and any AND gate input by connecting the corresponding lines at their point of intersection. On the right is a programmable OR array, which involves connecting AND gate outputs to OR gate inputs. Most larger PLAs contain several hundred gates, 15 to 25 inputs, and 5 to 15 outputs. The connections from the inputs to the AND gates, and from the AND gates to the OR gates, are not specified until programming time.

PLAs are manufactured in two different ways to allow easy programming (making of connections). In the first, every possible connection is made through a fuse at
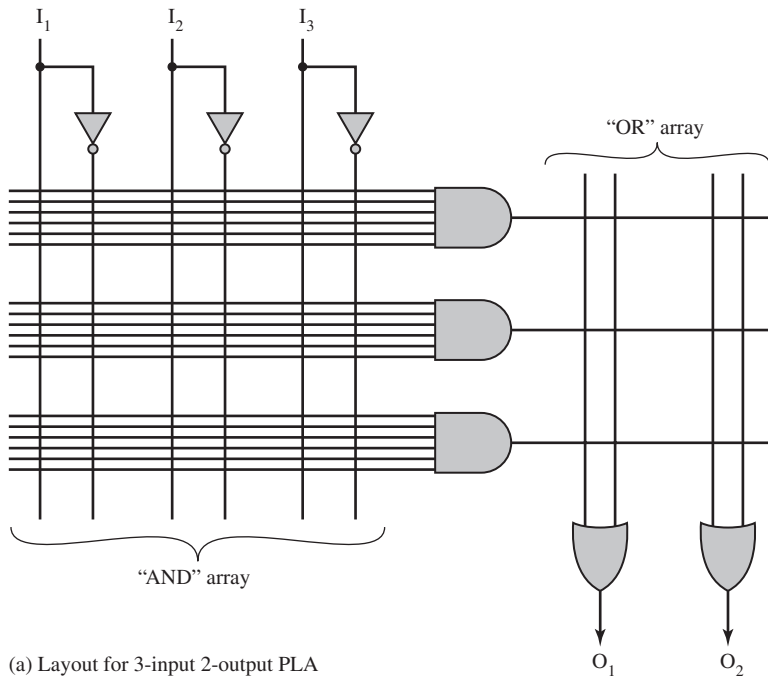
(a) Layout for 3-input 2-output PLA

(b) Programmed PLA

Figure 20.33 An Example of a Programmable Logic Array

every intersection point. The undesired connections can then be later removed by blowing the fuses. This type of PLA is referred to as a *field-programmable logic array.* Alternatively, the proper connections can be made during chip fabrication by using an appropriate mask supplied for a particular interconnection pattern. In either case, the PLA provides a flexible, inexpensive way of implementing digital logic functions.

Figure 20.33b shows a programmed PLA that realizes two Boolean expressions.

Field–Programmable Gate Array

The PLA is an example of a simple PLD (SPLD). The difficulty with increasing capacity of a strict SPLD architecture is that the structure of the programmable logic-planes grows too quickly in size as the number of inputs is increased. The only feasible way to provide large capacity devices based on SPLD architectures is then to integrate multiple SPLDs onto a single chip and provide interconnect to programmably connect the SPLD blocks together. Many commercial PLD products exist on the market today with this basic structure, and are collectively referred to as Complex PLDs (CPLDs). The most important type of CPLD is the FPGA.

An FPGA consists of an array of uncommitted circuit elements, called **logic blocks**, and interconnect resources. An illustration of a typical FPGA architecture is shown in Figure 20.34. The key components of an FGPA are;

- **Logic block:** The configurable logic blocks are where the computation of the user's circuit takes place.
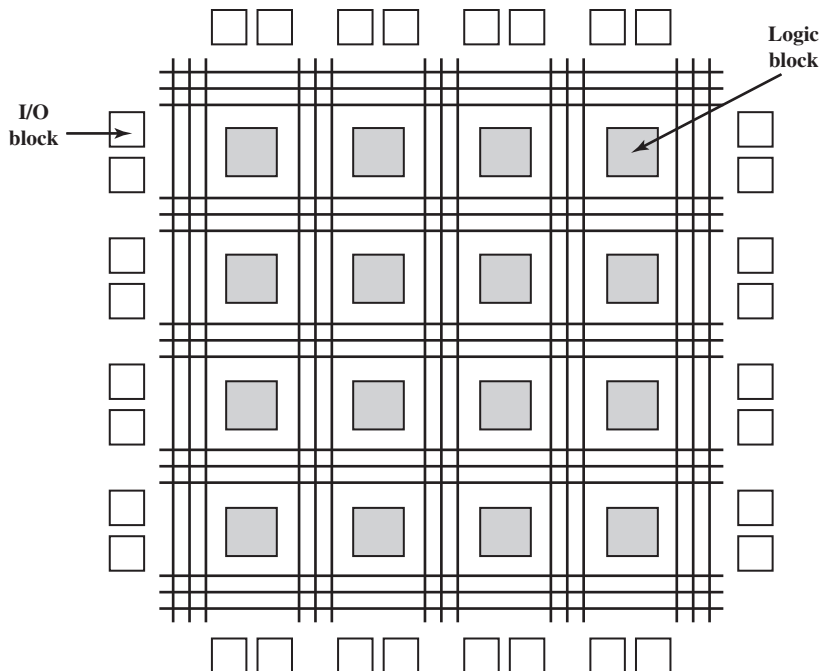- **I/O block:** The I/O blocks connect I/O pins to the circuitry on the chip.

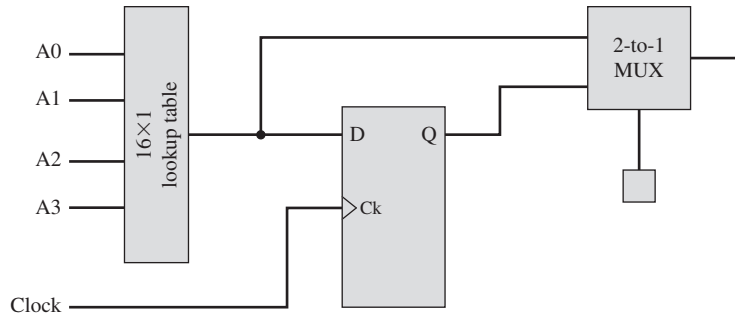**Figure 20.34**   Structure of an FPGA

Figure 20.35   A Simple FPGA Logic Block

- **Interconnect:** These are signal paths available for establishing connections among I/O blocks and logic blocks.

The logic block can be either a combinational circuit or a sequential circuit. In essence, the programming of a logic block is done by downloading the contents of a truth table for a logic function. Figure 20.35 Shows an example of a simple logic block consisting of a D flip-flop, a 2-to-1 multiplexer, and a 16-bit **lookup table**. The lookup table is a memory consisting of 16 1-bit elements, so that 4 input lines are required to select one of the 16 bits. Larger logic blocks have larger lookup tables and multiple interconnected lookup tables. The combinational logic realized by the lookup table can be output directly or stored in the D flip-flop and output synchronously. A separate one-bit memory controls the multiplexer to determine whether the output comes directly from the lookup table or from the flip-flop.

By interconnecting numerous logic blocks, very complex logic functions can be easily implemented.

20.6 RECOMMENDED READING AND WEB SITE

[GREG98] is an easy-to-read introduction to the concepts of this chapter. [STON96] is an excellent short introduction. A number of textbooks provide more in-depth treatment; these include [MANO04] and [FARH04].

[BROW96] is a worthwhile tutorial on programmable logic devices. [LEON08] looks at recent developments in FPGA devices, platforms, and applications.

BROW96   Brown, S., and Rose, S. "Architecture of FPGAs and CPLDs: A Tutorial." *IEEE Design and Test of Computers,* Vol. 13, No. 2, 1996.

FARH04   Farhat, H. *Digital Design and Computer Organization.* Boca Ratan: CRC Press, 2004.

GREG98   Gregg, J. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets.* New York: Wiley, 1998.

LEON08   Leong, p. "Recent Trends in FPGA Architectures and Applications." *Proceedings, 4th IEEE International symposium on Electronic Design, Test, and Applications*, 2008.

MANO04   Mano, M., and Kime, C. *Logic and Computer Design Fundamentals.* Upper Saddle River, NJ: Prentice Hall, 2004.

STON96   Stonham, T. *Digital Logic Techniques.* London: Chapman & Hall, 1996.

Recommended Web site:

- **Digital Logic:** Useful collection of circuit diagrams, with interactive features.

20.7 KEY TERMS AND PROBLEMS

Key Terms

adder	graphical symbol	programmable logic device
AND gate	J–K flip-flop	(PLD)
assert	Karnaugh map	Quine-McKluskey method
Boolean algebra	logic block	read-only memory (ROM)
clocked S–R flip-flop	lookup table	register
combinational circuit	multiplexer	ripple counter
complex PLD (CPLD)	NAND gate	sequential circuit
counter	NOR	shift register
decoder	OR gate	simple PLD (SPLD)
D flip-flop	parallel register	sum of products (SOP)
excitation table	product of sums (POS)	synchronous counter
field-programmable gate array	programmable array logic	S–R Latch
(FPGA)	(PAL)	truth table
flip-flop	programmable logic array	XOR gate
gates	(PLA)	

Problems

20.1 Construct a truth table for the following Boolean expressions:
a. $ABC + \overline{A}\,\overline{B}\,\overline{C}$
b. $ABC + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C}$
c. $A(B\overline{C} + \overline{B}C)$
d. $(A + B)(A + C)(\overline{A} + \overline{B})$

20.2 Simplify the following expressions according to the commutative law:
a. $A \cdot \overline{B} + \overline{B} \cdot A + C \cdot D \cdot E + \overline{C} \cdot D \cdot E + E \cdot \overline{C} \cdot D$
b. $A \cdot B + A \cdot C + B \cdot A$
c. $(L \cdot M \cdot N)(A \cdot B)(C \cdot D \cdot E)(M \cdot N \cdot L)$
d. $F \cdot (K + R) + S \cdot V + W \cdot \overline{X} + V \cdot S + \overline{X} \cdot W + (R + K) \cdot F$

20.3 Apply DeMorgan's theorem to the following equations:
a. $F = \overline{V + A + L}$
b. $F = \overline{A} + \overline{B} + \overline{C} + \overline{D}$

20.4 Simplify the following expressions:
a. $A = S \cdot T + V \cdot W + R \cdot S \cdot T$
b. $A = T \cdot U \cdot V + X \cdot Y + Y$
c. $A = F \cdot (E + F + G)$
d. $A = (P \cdot Q + R + S \cdot T)T \cdot S$
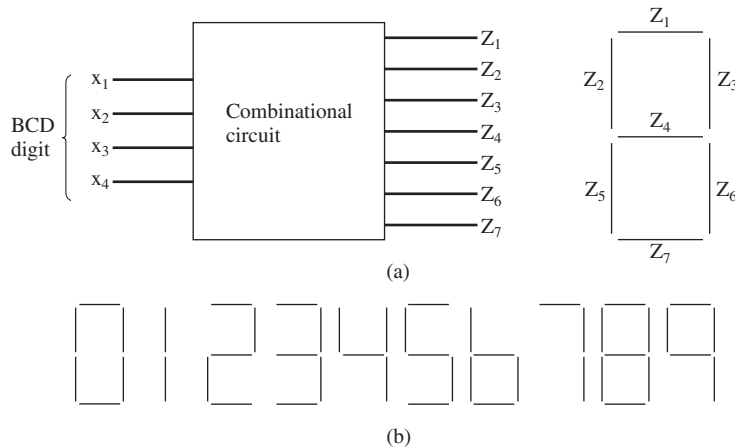e. $A = \overline{\overline{D} \cdot \overline{D} \cdot E}$

(a)

(b)

Figure 20.36 Seven-Segment LED Display Example

f. $A = Y \cdot (W + X + \overline{\overline{Y} \cdot \overline{Z}}) \cdot Z$
g. $A = (B \cdot E + C + F) \cdot C$

20.5   Construct the operation XOR from the basic Boolean operations AND, OR and NOT.

20.6   Given a NOR gate and NOT gates, draw a logic diagram that will perform the three-input AND function.

20.7   Write the Boolean expression for a four-input NAND gate.

20.8   A combinational circuit is used to control a seven-segment display of decimal digits, as shown in Figure 20.36. The circuit has four inputs, which provide the four-bit code used in packed decimal representation ($0_{10} = 0000, \ldots, 9_{10} = 1001$). The seven outputs define which segments will be activated to display a given decimal digit. Note that some combinations of inputs and outputs are not needed.
a.   Develop a truth table for this circuit.
b.   Express the truth table in SOP form.
c.   Express the truth table in POS form.
d.   Provide a simplified expression.

20.9   Design an 8-to-1 multiplexer.

20.10   Add an additional line to Figure 20.15 so that it functions as a demultiplexer.

20.11   The Gray code is a binary code for integers. It differs from the ordinary binary representation in that there is just a single bit change between the representations of any two numbers. This is useful for applications such as counters or analog-to-digital converters where a sequence of numbers is generated. Because only one bit changes at a time, there is never any ambiguity due to slight timing differences. The first eight elements of the code are

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Design a circuit that converts from binary to Gray code.

20.12   Design a 5×32 decoder using four 3×8 decoders (with enable inputs) and one 2×4 decoder.

20.13   Implement the full adder of Figure 20.20 with just five gates. (*Hint:* Some of the gates are XOR gates.)

20.14   Consider Figure 20.20. Assume that each gate produces a delay of 10 ns. Thus, the sum output is valid after 30 ns and the carry output after 0 ns. What is the total add time for a 32-bit adder
   a.   Implemented without carry lookahead, as in Figure 20.19?
   b.   Implemented with carry lookahead and using 8-bit adders, as in Figure 20.21?

20.15   An alternative form of the S–R latch has the same structure as Figure 20.22 but uses NAND gates instead of NOR gates.
   a.   Redo Table 20.10a and 20.10b for S–R latch implemented with NAND gates.
   b.   Complete the following table, similar to Table 20.10c

t	0	1	2	3	4	5	6	7	8	9
S	0	1	1	1	1	1	0	1	0	1
R	1	1	0	1	0	1	1	1	0	0

20.16   Consider the graphic symbol for the S–R flip-flop in Figure 20.27. Add additional lines to depict a D flip-flop wired from the S–R flip flop.

20.17   Show the structure of a PLA with three inputs (C, B, A) and four outputs (O_0, O_1, O_2, O_3), with the outputs defined as follows:

$$O_0 = \overline{A}\,\overline{B}C + A\overline{B} + AB\overline{C}$$
$$O_1 = \overline{A}\,\overline{B}C + AB\overline{C}$$
$$O_2 = C$$
$$O_3 = A\overline{B} + AB\overline{C}$$

20.18   An interesting application of a PLA is conversion from the old, obsolete punched cards character codes to ASCII codes. The standard punched cards that were so popular with computers in the past had 12 rows and 80 columns where holes could be punched. Each column corresponded to one character, so each character had a 12-bit code. However, only 96 characters were actually used. Consider an application that reads punched cards and converts the character codes to ASCII.
   a.   Describe a PLA implementation of this application.
   b.   Can this problem be solved with a ROM? Explain.