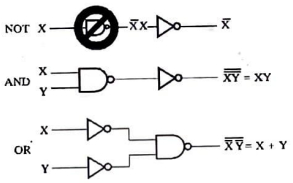
Name	Distinctive-Shape Graphics Symbol	Algebraic Equation	Truth Table
AND		$F = XY$	X Y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = X + Y$	X Y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT (inverter)		$F = \overline{X}$	X | F 0 | 1 1 | 0
NAND		$F = \overline{X \cdot Y}$	X Y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{X + Y}$	X Y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = X\overline{Y} + \overline{X}Y$ $= X \oplus Y$	X Y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR (XNOR)		$F = XY + \overline{X}\overline{Y}$ $= X \oplus Y$	X Y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

□ **FIGURE 2-3**
Commonly Used Logic Gates



□ **FIGURE 2-4**
Logical Operations with NAND Gates

bubble at the output of its graphics symbol. These gates indicate whether their two inputs are equal (XNOR) or not equal (XOR) to each other.

HDL Representations of Gates

While schematics using the basic logic gates are sufficient for describing small circuits, they are impractical for designing more complex digital systems. In contemporary computer systems design, HDL has become intrinsic to the design process. Consequently, we introduce HDLs early in the text. Initially, we justify such languages by describing their uses. We will then briefly discuss VHDL and Verilog[4], the most popular of these languages. At the end of this chapter and in Chapters 3 and 4, we will introduce them both in detail, although, in any given course, we expect that only one of them will be covered.

HDLs resemble programming languages, but are specifically oriented to describing hardware structures and behavior. They differ markedly from typical programming languages in that they represent extensive parallel operation, whereas most programming languages represent serial operation. An obvious use for an HDL is to provide an alternative to schematics. When a language is used in this fashion, it is referred to as a *structural description*, in which the language describes an interconnection of components. Such a structural description, referred to as a *netlist*, can be used as input to logic simulation just as a schematic is used. For this application, models for each of the primitive blocks are required. If an HDL is used, then these models can also be written in the HDL, providing a more uniform, portable representation for simulation input. Our use of HDLs in this chapter will be mainly limited to structural models. But as we will show later in the book, HDLs can represent much more than low-level behavior. In contemporary digital design, HDL models at a high level of abstraction can be automatically synthesized into optimized, working hardware.

To provide an initial introduction to HDLs, we start with features aimed at representing structural models. Table 2-2 shows the built-in Verilog primitives for the

□ **TABLE 2-2**
Verilog Primitives for Combinational Logic Gates

Gate primitive	Example instance
and	`and (F, X, Y);`
or	`or (F, X, Y);`
not	`not (F, X);`
nand	`nand (F, X, Y);`
nor	`nor (F, X, Y);`
xor	`xor (F, X, Y);`
xnor	`xnor (F, X, Y);`

□ **TABLE 2-3**
VHDL Predefined Logic Operators

VHDL logic operator	Example
not	`F <= not X;`
and	`F <= X and Y;`
or	`F <= X or Y;`
nand	`F <= X nand Y;`
nor	`F <= X nor Y;`
xor	`F <= X xor Y;`
xnor	`F <= X xnor Y;`

common logic gates from Figure 2-3. Each primitive declaration includes a list of signals that are its inputs and output. The first signal in the list is the output of the gate, and the remaining signals are the inputs. For the `not` gate, there can be only one input, but for the other gates, there can be two or more inputs. In Verilog, the gate primitives can be connected together to create structural models of logic circuits. VHDL does not have built-in logic gate primitives, but it does have logic operators that can be used to model the basic combinational gates, shown in Table 2-3. Verilog also has logic operators that can be used to model the basic combinational gates, shown in Table 2-4. Chapters 3 and 4 will show the necessary details to create fully simulation-ready models using these gate primitives and logic operators, but the reason for describing them at this point is to show that the HDLs provide an alternative for representing logic circuits. For small circuits, describing the input/output relationships with logic functions, truth tables, or schematics might be clear and feasible, but for larger, more complex circuits, HDLs are often more appropriate.

2-2 Boolean Algebra

The Boolean algebra we present is an algebra dealing with binary variables and logic operations. The variables are designated by letters of the alphabet, and the three basic logic operations are AND, OR, and NOT (complementation). A *Boolean expression* is an algebraic expression formed by using binary variables, the constants 0 and 1,

□ **TABLE 2-4**
Verilog Bitwise Logic Operators

Verilog operator symbol	Operator function	Example
~	Bitwise not	`F = ~X;`
&	Bitwise and	`F = X & Y;`
|	Bitwise or	`F = X | Y;`
^	Bitwise xor	`F = X ^ Y;`
^~, ~^	Bitwise xnor	`F = X ~^ Y;`

the logic operation symbols, and parentheses. A *Boolean function* can be described by a Boolean equation consisting of a binary variable identifying the function followed by an equals sign and a Boolean expression. Optionally, the function identifier is followed by parentheses enclosing a list of the function variables separated by commas. A *single-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to value 0 or 1. A *multiple-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to combinations of 0 and 1 on the function outputs.

EXAMPLE 2-1 Boolean Function Example–Power Windows

Consider an example Boolean equation representing electrical or electronic logic for control of the lowering of the driver's power window in a car.

$$L(D, X, A) = D\overline{X} + A$$

The window is raised or lowered by a motor driving a lever mechanism connected to the window. The function $L = 1$ means that the window motor is powered up to turn in the direction that lowers the window. $L = 0$ means the window motor is not powered up to turn in this direction. D is an output produced by pushing a panel switch on the inside of the driver's door. With $D = 1$, the lowering of the driver's window is requested, and with $D = 0$, this action is not requested. X is the output of a mechanical limit switch. $X = 1$ if the window is at a limit—in this case, in the fully down position. $X = 0$ if the window is not at its limit—i.e., not in the fully down position. $A = 1$ indicates automatic lowering of the window until it is in the fully down position. A is a signal generated by timing logic from D and X. Whenever D has been 1 for at least one-half second, A becomes 1 and remains at 1 until $X = 1$. If $D = 1$ for less than one-half second, $A = 0$. Thus, if the driver requests that the window be lowered for one-half second or longer, the window is to be lowered automatically to the fully down position.

The two parts of the expression, $D\overline{X}$ and A, are called *terms* of the expression for L. The function L is equal to 1 if term $D\overline{X}$ is equal to 1 or if term A is equal to 1. Otherwise, L is equal to 0. The complement operation dictates that if $\overline{X} = 1$, then $X = 0$. Therefore, we can say that $L = 1$ if $D = 1$, and $X = 0$ or if $A = 1$. So what does the equation for L say if interpreted in words? It says that the window will be

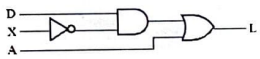
lowered if the window is not fully lowered ($X = 0$) and the switch D is being pushed ($D = 1$) or if the window is to be lowered automatically to fully down position ($A = 1$). ■

A Boolean equation expresses the logical relationship between binary variables. It is evaluated by determining the binary value of the expression for all possible combinations of values for the variables. A Boolean function can be represented by a truth table. A *truth table* for a function is a list of all combinations of 1s and 0s that can be assigned to the binary variables and a list that shows the value of the function for each binary combination. The truth tables for the logic operations given in Table 2-1 are special cases of truth tables for functions. The number of rows in a truth table is 2^n, where n is the number of variables in the function. The binary combinations for the truth table are the n-bit binary numbers that correspond to counting in decimal from 0 through $2^n - 1$. Table 2-5 shows the truth table for the function $L = D\overline{X} + A$. There are eight possible binary combinations that assign bits to the three variables D, X, and A. The column labeled L contains either 0 or 1 for each of these combinations. The table shows that the function L is equal to 1 if $D = 1$ and $X = 0$ or if $A = 1$. Otherwise, the function L is equal to 0.

An algebraic expression for a Boolean function can be transformed into a circuit diagram composed of logic gates that implements the function. The logic circuit diagram for function L is shown in Figure 2-5, with the equivalent Verilog and VHDL models for the circuit shown in Figures 2-6 and 2-7. An inverter on input X generates the complement, $\overline{X}$. An AND gate operates on $\overline{X}$ and D, and an OR gate combines $D\overline{X}$ and A. In logic circuit diagrams, the variables of the function F are taken as the inputs of the circuit, and the binary variable F is taken as the output of the circuit. If the circuit has a single output, F is a single output function. If the circuit has multiple

□ **TABLE 2-5**
Truth Table for the Function $L = D\overline{X} + A$

D	X	A	L
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



□ **FIGURE 2-5**
Logic Circuit Diagram for $L = D\overline{X} + A$


```
module fig2_5 (L, D, X, A);
    input D, X, A;
    output L;
    wire X_n, t2;

    not (X_n, X);
    and (t2, D, X_n);
    or (L, t2, A);
endmodule
```

□ **FIGURE 2-6**
Verilog Model for the Logic Circuit of Figure 2-5

```
library ieee, ldcf_vhdl;
use ieee.std_logic_1164.all,
ldcf_vhdl.func_prims.all;
entity fig2_5 is
  port (L: out std_logic;
  D, X, A: in std_logic);
end fig2_5;

architecture structural of fig2_5 is
 component NOT1
  port(in1: in std_logic;
    out1: out std_logic);
 end component;
 component AND2
  port(in1, in2: in std_logic;
    out1: out std_logic);
 end component;
 component OR2
  port(in1, in2: in std_logic;
    out1: out std_logic);
 end component;
signal X_n, t2: std_logic;

begin
    g0: NOT1 port map(X, X_n);
    g1: AND2 port map(D, X_n, t2);
    g3: OR2 port map(t2, A, L);
end structural;
```

□ **FIGURE 2-7**
VHDL Model for the Logic Circuit of Figure 2-5

outputs. function F is a multiple output function with multiple variables and equations required to represent its outputs. Circuit gates are interconnected by wires that carry logic signals. Logic circuits of this type are called *combinational logic circuits*, since the variables are "combined" by the logical operations. This is in contrast to the sequential logic to be treated in Chapter 4, in which variables are stored over time as well as being combined.

There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic equation form, it can be expressed in a variety of ways. The particular expression used to represent the function dictates the interconnection of gates in the logic circuit diagram. By manipulating a Boolean expression according to Boolean algebraic rules, it is often possible to obtain a simpler expression for the same function. This simpler expression reduces both the number of gates in the circuit and the numbers of inputs to the gates. To see how this is done, we must first study the basic rules of Boolean algebra.

Basic Identities of Boolean Algebra

Table 2-6 lists the most basic identities of Boolean algebra. The notation is simplified by omitting the symbol for AND whenever doing so does not lead to confusion. The first nine identities show the relationship between a single variable X, its complement $\overline{X}$, and the binary constants 0 and 1. The next five identities, 10 through 14, have counterparts in ordinary algebra. The last three, 15 through 17, do not apply in ordinary algebra, but are useful in manipulating Boolean expressions.

The basic rules listed in the table have been arranged into two columns that demonstrate the property of duality of Boolean algebra. The *dual* of an algebraic expression is obtained by interchanging OR and AND operations and replacing 1s by 0s and 0s by 1s. An equation in one column of the table can be obtained from the corresponding equation in the other column by taking the dual of the expressions on both sides of the equals sign. For example, relation 2 is the dual of relation 1 because the OR has been replaced by an AND and the 0 by 1. It is important to note that most of the time the dual of an expression is not equal to the original expression, so that an expression usually cannot be replaced by its dual.

□ **TABLE 2-6**
Basic Identities of Boolean Algebra

1. $X + 0 = X$	2. $X \cdot 1 = X$	
3. $X + 1 = 1$	4. $X \cdot 0 = 0$	
5. $X + X = X$	6. $X \cdot X = X$	
7. $X + \overline{X} = 1$	8. $X \cdot \overline{X} = 0$	
9. $\overline{\overline{X}} = X$		
10. $X + Y = Y + X$	11. $XY = YX$	Commutative
12. $X + (Y + Z) = (X + Y) + Z$	13. $X(YZ) = (XY)Z$	Associative
14. $X(Y + Z) = XY + XZ$	15. $X + YZ = (X + Y)(X + Z)$	Distributive
16. $\overline{X + Y} = \overline{X} \cdot \overline{Y}$	17. $\overline{X \cdot Y} = \overline{X} + \overline{Y}$	DeMorgan's

The nine identities involving a single variable can be easily verified by substituting each of the two possible values for X. For example, to show that $X + 0 = X$, let $X = 0$ to obtain $0 + 0 = 0$, and then let $X = 1$ to obtain $1 + 0 = 1$. Both equations are true according to the definition of the OR logic operation. Any expression can be substituted for the variable X in all the Boolean equations listed in the table. Thus, by identity 3 and with $X = AB + C$, we obtain

$$AB + C + 1 = 1$$

Note that identity 9 states that double complementation restores the variable to its original value. Thus, if $X = 0$, then $\overline{X} = 1$ and $\overline{\overline{X}} = 0 = X$.

Identities 10 and 11, the commutative laws, state that the order in which the variables are written will not affect the result when using the OR and AND operations. Identities 12 and 13, the associative laws, state that the result of applying an operation over three variables is independent of the order that is taken, and therefore, the parentheses can be removed altogether, as follows:

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

$$X(YZ) = (XY)Z = XYZ$$

These two laws and the first distributive law, identity 14, are well known from ordinary algebra, so they should not pose any difficulty. The second distributive law, given by identity 15, is the dual of the ordinary distributive law and does not hold in ordinary algebra. As illustrated previously, each variable in an identity can be replaced by a Boolean expression, and the identity still holds. Thus, consider the expression $(A + B)(A + CD)$. Letting $X = A$, $Y = B$, and $Z = CD$, and applying the second distributive law, we obtain

$$(A + B)(A + CD) = A + BCD$$

The last two identities in Table 2-6,

$$\overline{X + Y} = \overline{X} \cdot \overline{Y} \text{ and } \overline{X \cdot Y} = \overline{X} + \overline{Y}$$

are referred to as DeMorgan's theorem. This is a very important theorem and is used to obtain the complement of an expression and of the corresponding function. DeMorgan's theorem can be illustrated by means of truth tables that assign all the possible binary values to X and Y. Table 2-7 shows two truth tables that verify the

□ **TABLE 2-7**
Truth Tables to Verify DeMorgan's Theorem

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

4.3.23

first part of DeMorgan's theorem. In (a), we evaluate $\overline{X + Y}$ for all possible values of X and Y. This is done by first evaluating $X + Y$ and then taking its complement. In (b), we evaluate $\overline{X}$ and $\overline{Y}$ and then AND them together. The result is the same for the four binary combinations of X and Y, which verifies the identity of the equation.

Note the order in which the operations are performed when evaluating an expression. In part (b) of the table, the complement over a single variable is evaluated first, followed by the AND operation, just as in ordinary algebra with multiplication and addition. In part (a), the OR operation is evaluated first. Then, noting that the complement over an expression such as $X + Y$ is considered as specifying NOT $(X + Y)$, we evaluate the expression within the parentheses and take the complement of the result. It is customary to exclude the parentheses when complementing an expression, since a bar over the entire expression joins it together. Thus, $\overline{(X + Y)}$ is expressed as $\overline{X + Y}$ when designating the complement of $X + Y$.

DeMorgan's theorem can be extended to three or more variables. The general DeMorgan's theorem can be expressed as

$$\overline{X_1 + X_2 + \ldots + X_n} = \overline{X}_1\overline{X}_2 \ldots \overline{X}_n$$

$$\overline{X_1 X_2 \ldots X_n} = \overline{X}_1 + \overline{X}_2 + \ldots + \overline{X}_n$$

Observe that the logic operation changes from OR to AND or from AND to OR. In addition, the complement is removed from the entire expression and placed instead over each variable. For example,

$$\overline{A + B + C + D} = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$$

Algebraic Manipulation

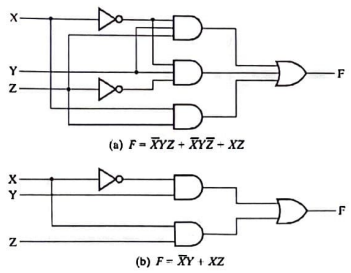
Boolean algebra is a useful tool for simplifying digital circuits. Consider, for example, the Boolean function represented by

$$F = \overline{X}YZ + X\overline{Y}\overline{Z} + XZ$$

The implementation of this equation with logic gates is shown in Figure 2-8(a). Input variables X and Z are complemented with inverters to obtain $\overline{X}$ and $\overline{Z}$. The three terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the terms. Now consider a simplification of the expression for F by applying some of the identities listed in Table 2-6:

$$\begin{aligned} F &= \overline{X}YZ + X\overline{Y}\overline{Z} + XZ \\ &= \overline{X}Y(Z + \overline{Z}) + XZ && \text{by identity 14} \\ &= \overline{X}Y \cdot 1 + XZ && \text{by identity 7} \\ &= \overline{X}Y + XZ && \text{by identity 2} \end{aligned}$$

The expression is reduced to only two terms and can be implemented with gates as shown in Figure 2-8(b). It is obvious that the circuit in (b) is simpler than the one in (a) yet, both implement the same function. It is possible to use a truth table to verify that the two implementations are equivalent. This is shown in Table 2-8. As



(a) $F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$

(b) $F = \overline{X}Y + XZ$

□ **FIGURE 2-8**
Implementation of Boolean Function with Gates

□ **TABLE 2-8**
Truth Table for Boolean Function

X	Y	Z	(a) F	(b) F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

expressed in Figure 2-8(a), the function is equal to 1 if $X = 0$, $Y = 1$, and $Z = 1$; if $X = 0$, $Y = 1$, and $Z = 0$; or if X and Z are both 1. This produces the four 1s for F in part (a) of the table. As expressed in Figure 2-8(b), the function is equal to 1 if $X = 0$ and $Y = 1$ or if $X = 1$ and $Z = 1$. This produces the same four 1s in part (b) of the table. Since both expressions produce the same truth table, they are equivalent. Therefore, the two circuits have the same output for all possible binary combinations of the three input variables. Each circuit implements the same function, but the one with fewer gates and/or fewer gate inputs is preferable because it requires fewer components.

When a Boolean equation is implemented with logic gates, each term requires a gate, and each variable within the term designates an input to the gate. We define a *literal* as a single variable within a term that may or may not be complemented. The

expression for the function in Figure 2-8(a) has three terms and eight literals; the one in Figure 2-8(b) has two terms and four literals. By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. Boolean algebra is applied to reduce an expression for the purpose of obtaining a simpler circuit. For highly complex functions, finding the best expression based on counts of terms and literals is very difficult, even by the use of computer programs. Certain methods, however, for reducing expressions are often included in computer tools for synthesizing logic circuits. These methods can obtain good, if not the best, solutions. The only manual method for the general case is a cut-and-try procedure employing the basic relations and other manipulations that become familiar with use. The following examples use identities from Table 2-6 to illustrate a few of the possibilities:

1. $X + XY = X \cdot 1 + XY = X(1 + Y) = X \cdot 1 = X$
2. $XY + X\overline{Y} = X(Y + \overline{Y}) = X \cdot 1 = X$
3. $X + \overline{X}Y = (X + \overline{X})(X + Y) = 1 \cdot (X + Y) = X + Y$

Note that the intermediate steps $X = X \cdot 1$ and $X \cdot 1 = X$ are often omitted because of their rudimentary nature. The relationship $1 + Y = 1$ is useful for eliminating redundant terms, as is done with the term XY in this same equation. The relation $Y + \overline{Y} = 1$ is useful for combining two terms, as is done in equation 2. The two terms being combined must be identical except for one variable, and that variable must be complemented in one term and not complemented in the other. Equation 3 is simplified by means of the second distributive law (identity 15 in Table 2-6). The following are three more examples of simplifying Boolean expressions:

4. $X(X + Y) = X \cdot X + X \cdot Y = X + XY = X(1 + Y) = X \cdot 1 = X$
5. $(X + Y)(X + \overline{Y}) = X + Y\overline{Y} = X + 0 = X$
6. $X(\overline{X} + Y) = X\overline{X} + XY = 0 + XY = XY$

The six equalities represented by the initial and final expressions are theorems of Boolean algebra proved by the application of the identities from Table 2-6. These theorems can be used along with the identities in Table 2-6 to prove additional results and to assist in performing simplification.

Theorems 4 through 6 are the duals of equations 1 through 3. Remember that the dual of an expression is obtained by changing AND to OR and OR to AND throughout (and 1s to 0s and 0s to 1s if they appear in the expression). The *duality principle* of Boolean algebra states that a Boolean equation remains valid if we take the dual of the expressions on both sides of the equals sign. Therefore, equations 4, 5, and 6 can be obtained by taking the dual of equations 1, 2, and 3, respectively.

Along with the results just given in equations 1 through 6, the following *consensus theorem* is useful when simplifying Boolean expressions:

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

The theorem shows that the third term, YZ, is redundant and can be eliminated. Note that Y and Z are associated with X and $\overline{X}$ in the first two terms and appear

together in the term that is eliminated. The proof of the consensus theorem is obtained by first ANDing YZ with $(X + \overline{X}) = 1$ and proceeds as follows:

$$\begin{aligned} XY + \overline{X}Z + YZ &= XY + \overline{X}Z + YZ(X + \overline{X}) \\ &= XY + \overline{X}Z + XYZ + \overline{X}YZ \\ &= XY + XYZ + \overline{X}Z + \overline{X}YZ \\ &= XY(1 + Z) + \overline{X}Z(1 + Y) \\ &= XY + \overline{X}Z \end{aligned}$$

The dual of the consensus theorem is

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

The following example shows how the consensus theorem can be applied in manipulating a Boolean expression:

$$\begin{aligned} (A + B)(\overline{A} + C) &= A\overline{A} + AC + \overline{A}B + BC \\ &= AC + \overline{A}B + BC \\ &= AC + \overline{A}B \end{aligned}$$

Note that $A\overline{A} = 0$ and $0 + AC = AC$. The redundant term eliminated in the last step by the consensus theorem is BC.

Complement of a Function

The complement representation for a function F, $\overline{F}$, is obtained from an interchange of 1s to 0s and 0s to 1s for the values of F in the truth table. The complement of a function can be derived algebraically by applying DeMorgan's theorem. The generalized form of this theorem states that the complement of an expression is obtained by interchanging AND and OR operations and complementing each variable and constant, as shown in Example 2-2.

EXAMPLE 2-2 Complementing Functions

Find the complement of each of the functions represented by the equations $F_1 = \overline{X}YZ + X\overline{Y}Z$ and $F_2 = X(\overline{Y}\,\overline{Z} + YZ)$. Applying DeMorgan's theorem as many times as necessary, we obtain the complements as follows:

$$\begin{aligned} \overline{F}_1 &= \overline{\overline{X}YZ + X\overline{Y}Z} = \overline{(\overline{X}YZ)} \cdot \overline{(X\overline{Y}Z)} \\ &= (X + \overline{Y} + Z)(X + Y + \overline{Z}) \\ \overline{F}_2 &= \overline{X(\overline{Y}\,\overline{Z} + YZ)} = \overline{X} + \overline{(\overline{Y}\,\overline{Z} + YZ)} \\ &= \overline{X} + \overline{\overline{Y}\,\overline{Z}} \cdot \overline{YZ} \\ &= \overline{X} + (Y + Z)(\overline{Y} + \overline{Z}) \end{aligned}$$

A simpler method for deriving the complement of a function is to take the dual of the function equation and complement each literal. This method follows from the

generalization of DeMorgan's theorem. Remember that the dual of an expression is obtained by interchanging AND and OR operations and 1s and 0s. To avoid confusion in handling complex functions, adding parentheses around terms before taking the dual is helpful, as illustrated in the next example. ■

EXAMPLE 2-3 Complementing Functions by Using Duals

Find the complements of the functions in Example 2-2 by taking the duals of their equations and complementing each literal.

We begin with

$$F_1 = \overline{X}YZ + X\overline{Y}Z = (\overline{X}YZ) + (X\overline{Y}Z)$$

The dual of F_1 is

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

Complementing each literal, we have

$$(X + \overline{Y} + Z)(\overline{X} + Y + \overline{Z}) = \overline{F}_1$$

Now,

$$F_2 = X(\overline{Y}\overline{Z} + YZ) = X((\overline{Y}\overline{Z}) + (YZ))$$

The dual of F_2 is

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

Complementing each literal yields

$$\overline{X} + (Y + Z)(\overline{Y} + \overline{Z}) = \overline{F}_2$$ ■

2-3 Standard Forms

A Boolean function expressed algebraically can be written in a variety of ways. There are, however, specific ways of writing algebraic equations that are considered to be standard forms. The standard forms facilitate the simplification procedures for Boolean expressions and, in some cases, may result in more desirable expressions for implementing logic circuits.

The standard forms contain *product terms* and *sum terms*. An example of a product term is $X\overline{Y}Z$. This is a logical product consisting of an AND operation among three literals. An example of a sum term is $X + Y + \overline{Z}$. This is a logical sum consisting of an OR operation among the literals. In Boolean algebra, the words "product" and "sum" do not imply arithmetic operations—instead, they specify the logical operations AND and OR, respectively.

Minterms and Maxterms

A truth table defines a Boolean function. An algebraic expression for the function can be derived from the table by finding a logical sum of product terms for which the function assumes the binary value 1. A product term in which all the variables appear

exactly once, either complemented or uncomplemented, is called a *minterm*. Its characteristic property is that it represents exactly one combination of binary variable values in the truth table. It has the value 1 for that combination and 0 for all others. There are 2^n distinct minterms for n variables. The four minterms for the two variables X and Y are $\overline{X}\overline{Y}$, $\overline{X}Y$, $X\overline{Y}$, and XY. The eight minterms for the three variables X, Y, and Z are listed in Table 2-9. The binary numbers from 000 to 111 are listed under the variables. For each binary combination, there is a related minterm. Each minterm is a product term of exactly n literals, where n is the number of variables. In this example, $n = 3$. A literal is a complemented variable if the corresponding bit of the related binary combination is 0 and is an uncomplemented variable if it is 1. A symbol m_j for each minterm is also shown in the table, where the subscript j denotes the decimal equivalent of the binary combination corresponding to the minterm. This list of minterms for any given n variables can be formed in a similar manner from a list of the binary numbers from 0 through $2^n - 1$. In addition, the truth table for each minterm is given in the right half of the table. These truth tables clearly show that each minterm is 1 for the corresponding binary combination and 0 for all other combinations. Such truth tables will be helpful later in using minterms to form Boolean expressions.

A sum term that contains all the variables in complemented or uncomplemented form is called a *maxterm*. Again, it is possible to formulate 2^n maxterms with n variables. The eight maxterms for three variables are listed in Table 2-10. Each maxterm is a logical sum of the three variables, with each variable being complemented if the corresponding bit of the binary number is 1 and uncomplemented if it is 0. The symbol for a maxterm is M_j, where j denotes the decimal equivalent of the binary combination corresponding to the maxterm. In the right half of the table, the truth table for each maxterm is given. Note that the value of the maxterm is 0 for the corresponding combination and 1 for all other combinations. It is now clear where the terms "minterm" and "maxterm" come from: a minterm is a function, not equal to 0, having the minimum number of 1s in its truth table; a maxterm is a function, not equal to 1, having the maximum of 1s in its truth table. Note from Table 2-9

□ **TABLE 2-9**
Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\overline{Y}\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}Y\overline{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

and Table 2-10 that a minterm and maxterm with the same subscript are the complements of each other; that is, $M_j = \overline{m}_j$ and $m_j = \overline{M}_j$. For example, for $j = 3$, we have

$$M_3 = X + \overline{Y} + \overline{Z} + = \overline{X}\overline{Y}Z = \overline{m}_3$$

A Boolean function can be represented algebraically from a given truth table by forming the logical sum of all the minterms that produce a 1 in the function. This expression is called a *sum of minterms*. Consider the Boolean function F in Table 2-11(a). The function is equal to 1 for each of the following binary combinations of the variables X, Y, and Z: 000, 010, 101, and 111. These combinations correspond to minterms 0, 2, 5, and 7. By examining Table 2-11 and the truth tables for these minterms in Table 2-9, it is evident that the function F can be expressed algebraically as the logical sum of the stated minterms:

$$F = \overline{X}\overline{Y}\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

This can be further abbreviated by listing only the decimal subscripts of the minterms:

$$F(X, Y, Z) = \Sigma m(0, 2, 5, 7)$$

□ **TABLE 2-10**
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	0	0	$X + Y + Z$	M_0	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \overline{Z}$	M_1	1	0	1	1	1	1	1	1
0	1	0	$X + \overline{Y} + Z$	M_2	1	1	0	1	1	1	1	1
0	1	1	$X + \overline{Y} + \overline{Z}$	M_3	1	1	1	0	1	1	1	1
1	0	0	$\overline{X} + Y + Z$	M_4	1	1	1	1	0	1	1	1
1	0	1	$\overline{X} + Y + \overline{Z}$	M_5	1	1	1	1	1	0	1	1
1	1	0	$\overline{X} + \overline{Y} + Z$	M_6	1	1	1	1	1	1	0	1
1	1	1	$\overline{X} + \overline{Y} + \overline{Z}$	M_7	1	1	1	1	1	1	1	0

□ **TABLE 2-11**
Boolean Functions of Three Variables

(a) X	Y	Z	F	$\overline{F}$	(b) X	Y	Z	E
0	0	0	1	0	0	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	1	0	0	1	0	1
0	1	1	0	1	0	1	1	0
1	0	0	0	1	1	0	0	1
1	0	1	1	0	1	0	1	1
1	1	0	0	1	1	1	0	0
1	1	1	1	0	1	1	1	0

The symbol Σ stands for the logical sum (Boolean OR) of the minterms. The numbers following it represent the minterms of the function. The letters in parentheses following F form a list of the variables in the order taken when the minterms are converted to product terms.

Now consider the complement of a Boolean function. The binary values of $\overline{F}$ in Table 2-11(a) are obtained by changing 1s to 0s and 0s to 1s in the values of F. Taking the logical sum of minterms of $\overline{F}$, we obtain

$$\overline{F}(X,Y,Z) = \overline{X}\overline{Y}Z + \overline{X}YZ + X\overline{Y}\overline{Z} + XY\overline{Z} = m_1 + m_3 + m_4 + m_6$$

or, in abbreviated form,

$$\overline{F}(X, Y, Z) = \Sigma m(1, 3, 4, 6)$$

Note that the minterm numbers for $\overline{F}$ are the ones missing from the list of the minterm numbers of F. We now take the complement of $\overline{F}$ to obtain F:

$$\begin{aligned} F &= \overline{m_1 + m_3 + m_4 + m_6} = \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_6} \\ &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \text{ (since } \overline{m_j} = M_j) \\ &= (X + Y + \overline{Z})(X + \overline{Y} + \overline{Z})(\overline{X} + Y + Z)(\overline{X} + \overline{Y} + Z) \end{aligned}$$

This shows the procedure for expressing a Boolean function as a *product of maxterms*. The abbreviated form for this product is

$$F(X, Y, Z) = \prod M(1, 3, 4, 6)$$

where the symbol Π denotes the logical product (Boolean AND) of the maxterms whose numbers are listed in parentheses. Note that the decimal numbers included in the product of maxterms will always be the same as the minterm list of the complemented function, such as (1, 3, 4, 6) in the foregoing example. Maxterms are seldom used directly when dealing with Boolean functions, since we can always replace them with the minterm list of $\overline{F}$.

The following is a summary of the most important properties of minterms:

1. There are 2^n minterms for n Boolean variables. These minterms can be generated from the binary numbers from 0 to $2^n - 1$.
2. Any Boolean function can be expressed as a logical sum of minterms.
3. The complement of a function contains those minterms not included in the original function.
4. A function that includes all the 2^n minterms is equal to logic 1.

A function that is not in the sum-of-minterms form can be converted to that form by means of a truth table, since the truth table always specifies the minterms of the function. Consider, for example, the Boolean function

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

The expression is not in sum-of-minterms form, because each term does not contain all three variables X, Y, and Z. The truth table for this function is listed in Table 2-11(b).

From the table, we obtain the minterms of the function:

$$E(X, Y, Z) = \Sigma m(0, 1, 2, 4, 5)$$

The minterms for the complement of E are given by

$$\overline{E}(X, Y, Z) = \Sigma m(3, 6, 7)$$

Note that the total number of minterms in E and $\overline{E}$ is equal to eight, since the function has three variables, and three variables produce a total of eight minterms. With four variables, there will be a total of 16 minterms, and for two variables, there will be four minterms. An example of a function that includes all the minterms is

$$G(X, Y) = \Sigma m(0, 1, 2, 3) = 1$$

Since G is a function of two variables and contains all four minterms, it is always equal to logic 1.

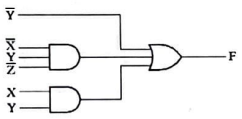
Sum of Products

The sum-of-minterms form is a standard algebraic expression that is obtained directly from a truth table. The expression so obtained contains the maximum number of literals in each term and usually has more product terms than necessary. This is because, by definition, each minterm must include all the variables of the function, complemented or uncomplemented. Once the sum of minterms is obtained from the truth table, the next step is to try to simplify the expression to see whether it is possible to reduce the number of product terms and the number of literals in the terms. The result is a simplified expression in *sum-of-products* form. This is an alternative standard form of expression that contains product terms with up to n literals. An example of a Boolean function expressed as a sum of products is

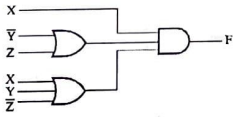
$$F = \overline{Y} + \overline{X}Y\overline{Z} + XY$$

The expression has three product terms, the first with one literal, the second with three literals, and the third with two literals.

The logic diagram for a sum-of-products form consists of a group of AND gates followed by a single OR gate, as shown in Figure 2-9. Each product term requires an AND gate, except for a term with a single literal. The logical sum is formed with an OR gate that has single literals and the outputs of the AND gates as inputs. Often,



□ **FIGURE 2-9**
Sum-of-Products Implementation



□ **FIGURE 2-11**
Product-of-Sums Implementation

The gate structure of the product-of-sums expression consists of a group of OR gates for the sum terms (except for a single literal term), followed by an AND gate. This is shown in Figure 2-11 for the preceding function F. As with the sum of products, this standard type of expression results in a two-level gating structure.

2-4 Two-Level Circuit Optimization

The complexity of a logic circuit that implements a Boolean function is directly related to the algebraic expression from which the function is implemented. Although the truth-table representation of a function is unique, when expressed algebraically, the function appears in many different forms. Boolean expressions may be simplified by algebraic manipulation, as discussed in Section 2-2. However, this procedure of simplification is awkward, because it lacks specific rules to predict each succeeding step in the manipulative process and it is difficult to determine whether the simplest expression has been achieved. By contrast, the map method provides a straightforward procedure for optimizing Boolean functions of up to four variables. Maps for five and six variables can be drawn as well, but are more cumbersome to use. The map is also known as the *Karnaugh map*, or *K-map*. The map is a diagram made up of squares, with each square representing one row of a truth table, or correspondingly, one minterm of a single output function. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map by those squares for which the function has value 1, or correspondingly, whose minterms are included in the function. From a more complex view, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. Among these ways are the optimum sum-of-products standard forms for the function. The optimized expressions produced by the map are always in sum-of-products or product-of-sums form. Thus, maps handle optimization for two-level implementations, but do not apply directly to possible simpler implementations for the general case with three or more levels. Initially, this section covers sum-of-products optimization and, later, applies it to performing product-of-sums optimization.

Cost Criteria

In the prior section, counting literals and terms was mentioned as a way of measuring the simplicity of a logic circuit. We introduce two cost criteria to formalize this concept.

we assumed that the input variables are directly available in their complemented and uncomplemented forms, so inverters are not included in the diagram. The AND gates followed by the OR gate form a circuit configuration referred to as a *two-level implementation* or *two-level circuit*.

If an expression is not in sum-of-products-form, it can be converted to the standard form by means of the distributive laws. Consider the expression

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

This is not in sum-of-products form, because the term $D + E$ is part of a product, not a single literal. The expression can be converted to a sum of products by applying the appropriate distributive law as follows:

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

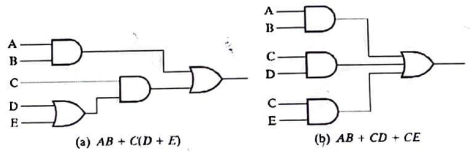
The function F is implemented in a nonstandard form in Figure 2-10(a). This requires two AND gates and two OR gates. There are three levels of gating in the circuit. F is implemented in sum-of-products form in Figure 2-10(b). This circuit requires three AND gates and an OR gate and uses two levels of gating. The decision as to whether to use a two-level or multiple-level (three levels or more) implementation is complex. Among the issues involved are the number of gates, number of gate inputs, and the amount of delay between the time the input values are set and the time the resulting output values appear. Two-level implementations are the natural form for certain implementation technologies, as we will see in Chapter 5.

Product of Sums

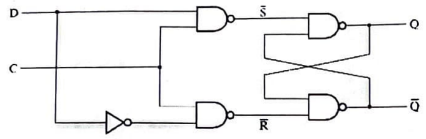
Another standard form of expressing Boolean functions algebraically is the *product of sums*. This form is obtained by forming a logical product of sum terms. Each logical sum term may have any number of distinct literals. An example of a function expressed in product-of-sums form is

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

This expression has sum terms of one, two, and three literals. The sum terms perform an OR operation, and the product is an AND operation.



□ **FIGURE 2-10**
Three-Level and Two-Level Implementation



(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

□ **FIGURE 4-8**
D Latch

control input goes back to 0, one cannot conclusively determine the next state, since the $\overline{S}\,\overline{R}$ latch sees inputs (0, 0) followed by (1, 1). The *SR* latch with control input is an important circuit, because other latches and flip-flops are constructed from it. Sometimes the *SR* latch with control input is referred to as an *SR* (or *RS*) flip-flop—however, according to our terminology, it does not qualify as a flip-flop, since the circuit does not fulfill the flip-flop requirements presented in the next section.

D Latch

One way to eliminate the undesirable undefined state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done in the *D* latch, shown in Figure 4-8. This latch has only two inputs: *D* (data) and *C* (control). The complement of the *D* input goes directly to the $\overline{S}$ input, and *D* is applied to the $\overline{R}$ input. As long as the control input is 0, the $\overline{S}\,\overline{R}$ latch has both inputs at the 1 level, and the circuit cannot change state regardless of the value of *D*. The *D* input is sampled when $C = 1$. If *D* is 1, the *Q* output goes to 1, placing the circuit in the set state. If *D* is 0, output *Q* goes to 0, placing the circuit in the reset state.

The *D* latch receives its designation from its ability to hold *data* in its internal storage. The binary information present at the data input of the *D* latch is transferred to the *Q* output when the control input is enabled (1). The output follows changes in the data input, as long as the control input is enabled. When the control input is disabled (0), the binary information that was present at the data input at the time the transition in *C* occurred is retained at the *Q* output until the control input *C* is enabled again.

4-3 Flip-Flops

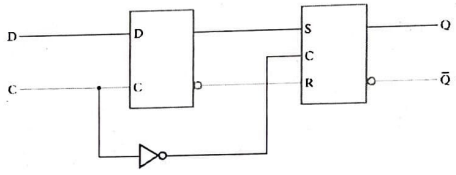
A change in value on the control input allows the state of a latch in a flip-flop to switch. This change is called a *trigger*, and it enables, or triggers, the flip-flop. The *D*

latch with clock pulses on its control input is triggered every time a pulse to the logic-1 level occurs. As long as the pulse remains at the active (1) level, any changes in the data input will change the state of the latch. In this sense, the latch is *transparent*, since its input value can be seen from the outputs while the control input is 1.

As the block diagram of Figure 4-3 shows, a sequential circuit has a feedback path from the outputs of the flip-flops to the combination circuit. As a consequence, the data inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch may appear at its output while the pulse is still active. This output is connected to the inputs of some of the latches through a combinational circuit. If the inputs applied to the latches change while the clock pulse is still in the logic-1 level, the latches will respond to *new state values* of other latches instead of the *original state values*, and a succession of changes of state instead of a single one may occur. The result is an unpredictable situation, since the state may keep changing and continue to change until the clock returns to 0. The final state depends on how long the clock pulse stays at the logic-1 level. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a single clock signal.

Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a single clock. Note that the problem with the latch is that it is transparent: As soon as an input changes, shortly thereafter the corresponding output changes to match it. This transparency is what allows a change on a latch output to produce additional changes at other latch outputs while the clock pulse is at logic 1. The key to the proper operation of flip-flops is to prevent them from being transparent. In a flip-flop, before an output can change, the path from its inputs to its outputs is broken. So a flip-flop cannot "see" the change of its output or of the outputs of other, similar flip-flops at its input during the same clock pulse. Thus, the new state of a flip-flop depends only on the immediately preceding state, and the flip-flops do not go through multiple changes of state.

A common way to create a flip-flop is to connect two latches as shown in Figure 4-9, which is often referred to as a *master–slave* flip-flop. The left latch, the master, changes its value based upon the input while the clock is high. That value is



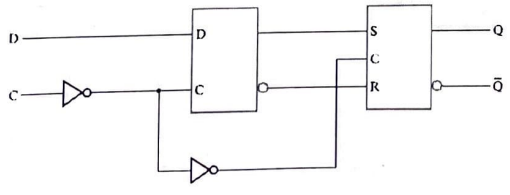
□ **FIGURE 4-9**
Negative-Edge-Triggered D Flip-Flop

then transferred to the right latch, the slave, when the clock changes to low. Depending upon the type of latch that is used to construct the master–slave flip-flop, there are two possible ways that the flip-flop can respond to changes in the clock. One way is to combine two latches such that (1) the inputs presented to the flip-flop when a clock pulse is present control its state and (2) the state of the flip-flop changes only when a clock pulse is not present. Such a circuit is called a *pulse-triggered* flip-flop. A master–slave flip-flop constructed with SR latches is a pulse-triggered flip-flop, because changes on either the S or R inputs of the master during the clock pulse can change the master's output value. Thus a master–slave SR flip-flop depends on the input values throughout the entire high clock pulse.

In contrast, another way is to produce a flip-flop that triggers only during a signal *transition* from 0 to 1 (or from 1 to 0) on the clock and that is disabled at all other times, including for the duration of the clock pulse. Such a circuit is said to be an *edge-triggered* flip-flop. Edge-triggered flip-flops tend to be faster and have easier design constraints than pulse-triggered flip-flops, so they are much more commonly used. It is necessary to consider the SR flip-flop to illustrate the pulse-triggering approach, which is presented in the online Companion Website due to its lesser prevalence in contemporary design. The edge-triggered D flip-flop is currently the most common flip-flop, so its implementation is presented next.

Edge-Triggered Flip-Flop

An *edge-triggered* flip-flop ignores the clock pulse while it is at a constant level and triggers only during a *transition* of the clock signal. Some edge-triggered flip-flops trigger on the positive edge (0-to-1 transition), whereas others trigger on the negative edge (1-to-0 transition). The logic diagram of a negative-edge-triggered D flip-flop is shown in Figure 4-9. The logic diagram of a *D*-type positive-edge-triggered flip-flop to be analyzed in detail here appears in Figure 4-10. This flip-flop is a master-slave flip-flop, with the master a *D* latch and the slave an *SR* latch or a *D* latch. Also, an inverter is added to the clock input. Because the master latch is a *D* latch, the flip-flop exhibits edge-triggered rather than pulse-triggered behavior. For the clock input equal to 0, the master latch is enabled and transparent and follows the *D* input value. The slave latch is disabled and holds the state of the flip-flop fixed.

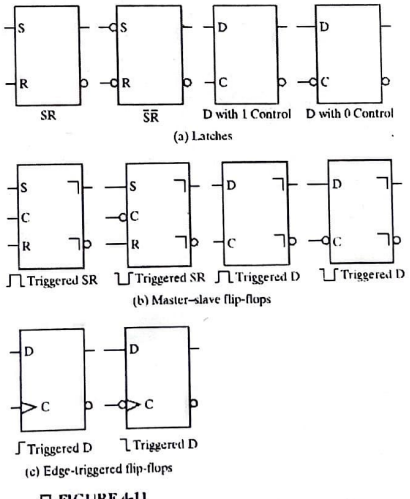


□ **FIGURE 4-10**
Positive-Edge-Triggered D Flip-Flop

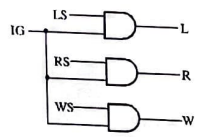
When the positive edge occurs, the clock input changes to 1. This disables the master latch so that its value is fixed and enables the slave latch so that it copies the state of the master latch. The state of the master latch to be copied is the state that is present at the positive edge of the clock. Thus, the behavior appears to be edge triggered. With the clock input equal to 1, the master latch is disabled and cannot change, so the state of both the master and the slave remain unchanged. Finally, when the clock input changes from 1 to 0, the master is enabled and begins following the *D* value. But during the 1-to-0 transition, the slave is disabled before any change in the master can reach it. Thus, the value stored in the slave remains unchanged during this transition. An alternative implementation that requires fewer gates is given in Problem 4-3 at the end of the chapter.

Standard Graphics Symbols

The standard graphics symbols for the different types of latches and flip-flops are shown in Figure 4-11. A flip-flop or latch is designated by a rectangular block with



□ **FIGURE 4-11**
Standard Graphics Symbols for Latches and Flip-Flop



□ **FIGURE 3-11**
Car Electrical Control Using Enabling

columns are used to represent product terms that are not minterms. For example, 0XXX represents the product term $\overline{IS}$. Just as with minterms, each variable is complemented if the corresponding bit in the input combination from the table is 0 and is not complemented if the bit is 1. If the corresponding bit in the input combination is an X, then the variable does not appear in the product term. When the ignition switch *IS* is on (1), then the accessories are controlled by their respective switches. When IS is off (0), all accessories are off. So *IS* replaces the normal values of the outputs *L*, *R*, and *W* with a fixed value 0 and meets the definition of an *ENABLE* signal. The resulting circuit is given in Figure 3-11. ■

3-5 DECODING

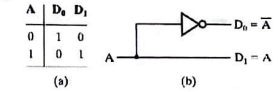
In digital computers, discrete quantities of information are represented by binary codes. An *n*-bit binary code is capable of representing up to 2^n distinct elements of coded information. Decoding is the conversion of an *n*-bit input code to an *m*-bit output code with $n \leq m \leq 2^n$, such that each valid input code word produces a unique output code. Decoding is performed by a *decoder*, a combinational circuit with an *n*-bit binary code applied to its inputs and an *m*-bit binary code appearing at the outputs. The decoder may have unused bit combinations on its inputs for which no corresponding *m*-bit code appears at the outputs. Among all of the specialized functions defined here, decoding is the most important, since this function and the corresponding functional blocks are incorporated into many of the other functions and functional blocks defined here.

In this section, the functional blocks that implement decoding are called *n*–to–*m*-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms from the *n* input variables. For $n = 1$ and $m = 2$, we obtain the 1–to–2-line decoding function with input A and outputs D_0 and D_1. The truth table for this decoding function is given in Figure 3-12(a). If $A = 0$, then $D_0 = 1$ and $D_1 = 0$. If $A = 1$, then $D_0 = 0$ and $D_1 = 1$. From this truth table, $D_0 = \overline{A}$ and $D_1 = A$, giving the circuit shown in Figure 3-12(b).

A second decoding function for $n = 2$ and $m = 4$ with the truth table given in Figure 3-13(a) better illustrates the general nature of decoders. This table has 2-variable minterms as its outputs, with each row containing one output value equal to 1 and three output values equal to 0. Output D_i is equal to 1 whenever the two input values on A_1 and A_0 are the binary code for the number *i*. As a consequence, the circuit implements the four possible minterms of two variables, one minterm for

A	D_0	D_1
0	1	0
1	0	1

(a)



(b)

□ **FIGURE 3-12**
A 1–to–2-Line Decoder

each output. In the logic diagram in Figure 3-13(b), each minterm is implemented by a 2-input AND gate. These AND gates are connected to two 1–to–2-line decoders, one for each of the lines driving the AND gate inputs.

Large decoders can be constructed by simply implementing each minterm function using a single AND gate with more inputs. Unfortunately, as decoders become larger, this approach gives a high gate-input cost. In this section, we give a procedure that uses design hierarchy and collections of AND gates to construct any decoder with *n* inputs and 2^n outputs. The resulting decoder has the same or a lower gate-input cost than the one constructed by simply enlarging each AND gate.

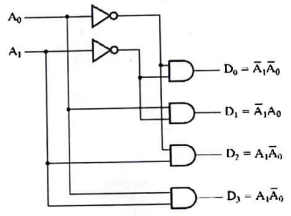
To construct a 3–to–8-line decoder ($n = 3$), we can use a 2–to–4-line decoder and a 1–to–2-line decoder feeding eight 2-input AND gates to form the minterms. Hierarchically, the 2–to–4-line decoder can be implemented using two 1–to–2-line decoders feeding four 2-input AND gates, as observed in Figure 3-13. The resulting structure is shown in Figure 3-14.

The general procedure is as follows:

1. Let $k = n$.
2. If k is even, divide k by 2 to obtain $k/2$. Use 2^k AND gates driven by two decoders of output size $2^k/2$. If k is odd, obtain $(k + 1)/2$ and $(k - 1)/2$. Use 2^k AND

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



(b)

□ **FIGURE 3-13**
A 2–to–4-Line Decoder

□ **TABLE 3-5**
Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

3-6 ENCODING

An encoder is a digital function that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and *n* output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 3-5. This encoder has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time, so that the table has only eight rows with specified output values. For the remaining 56 rows, all of the outputs are don't cares.

From the truth table, we can observe that A_i is 1 for the columns in which D_j is 1 only if subscript *j* has a binary representation with a 1 in the *i*th position. For example, output $A_0 = 1$ if the input is 1 or 3 or 5 or 7. Since all of these values are odd, they have a 1 in the 0 position of their binary representation. This approach can be used to find the truth table. From the table, the encoder can be implemented with *n* OR gates, one for each output variable A_i. Each OR gate combines the input variables D_j having a 1 in the rows for which A_i has value 1. For the 8–to–3-line encoder, the resulting output equations are

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

which can be implemented with three 4-input OR gates.

The encoder just defined has the limitation that only one input can be active at any given time: if two inputs are active simultaneously, the output produces an incorrect combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111, because all the three outputs are equal to 1. This represents neither a binary 3 nor a binary 6. To resolve this ambiguity, some encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for

inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110, because D_6 has higher priority than D_3. Another ambiguity in the octal-to-binary encoder is that an output of all 0s is generated when all the inputs are 0, but this output is the same as when D_0 is equal to 1. This discrepancy can be resolved by providing a separate output to indicate that at least one input is equal to 1.

Priority Encoder

A priority encoder is a combinational circuit that implements a priority function. As mentioned in the preceding paragraph, the operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority takes precedence. The truth table for a four-input priority encoder is given in Table 3-6. With the use of Xs, this condensed truth table with just five rows represents the same information as the usual 16-row truth table. Whereas Xs in output columns represent don't-care conditions, Xs in input columns are used to represent product terms that are not minterms. For example, 001X represents the product term $\overline{D}_3\overline{D}_2D_1$. Just as with minterms, each variable is complemented if the corresponding bit in the input combination from the table is 0 and is not complemented if the bit is 1. If the corresponding bit in the input combination is an X, then the variable does not appear in the product term. Thus, for 001X, the variable D_0, corresponding to the position of the X, does not appear in $\overline{D}_3\overline{D}_2D_1$.

The number of rows of a full truth table represented by a row in the condensed table is 2^p, where *p* is the number of Xs in the row. For example, in Table 3-6, 1XXX represents $2^3 = 8$ truth-table rows, all having the same value for all outputs. In forming a condensed truth table, we must include each minterm in at least one of the rows in the sense that the minterm can be obtained by filling in 1s and 0s for the Xs. Also, a minterm must never be included in more than one row such that the rows in which it appears have one or more conflicting output values.

We form Table 3-6 as follows: Input D_3 has the highest priority; so, regardless of the values of the other inputs, when this input is 1, the output for A_1A_0 is 11 (binary 3). From this we obtain the last row of the table. D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the lower-priority inputs. From this, we obtain the fourth row of the table. The output for D_1 is generated only if all inputs with higher priority are 0, and so on down the priority levels. From this, we obtain the remaining rows of the table. The valid output designated by *V* is set to 1 only when one or more of the inputs are equal to 1. If all inputs

□ **TABLE 3-6**
Truth Table of Priority Encoder

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

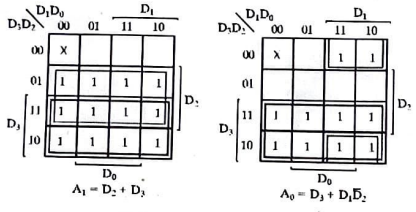
are 0, V is equal to 0, and the other two outputs of the circuit are not used and are specified as don't-care conditions in the output part of the table.

The maps for simplifying outputs A_1 and A_0 are shown in Figure 3-22. The minterms for the two functions are derived from Table 3-6. The output values in the table can be transferred directly to the maps by placing them in the squares covered by the corresponding product term represented in the table. The optimized equation for each function is listed under the map for the function. The equation for output V is an OR function of all the input variables. The priority encoder is implemented in Figure 3-23 according to the following Boolean functions:

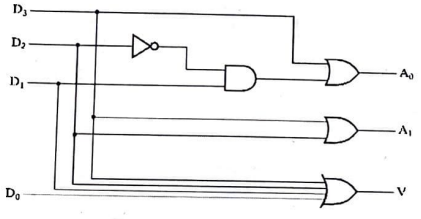
$$A_0 = D_3 + D_1\overline{D}_2$$
$$A_1 = D_2 + D_3$$
$$V = D_0 + D_1 + D_2 + D_3$$

Encoder Expansion

Thus far, we have considered only small encoders. Encoders can be expanded to larger numbers of inputs by expanding OR gates. In the implementation of



□ **FIGURE 3-22**
Maps for Priority Encoder



□ **FIGURE 3-23**
Logic Diagram of a 4-Input Priority Encoder

decoders, the use of multiple-level circuits with OR gates beyond the output levels shared in implementing the more significant bits in the output codes reduces the gate input cost as n increases for $n \geq 5$. For $n \geq 3$, multiple-level circuits result from technology mapping anyway, due to limited gate fan-in. Designing multiple-level circuits with shared gates reduces the cost of the encoders after technology mapping.

3-7 SELECTING

Selection of information to be used in a computer is a very important function, not only in communicating between the parts of the system, but also within the parts as well. Circuits that perform selection typically have a set of inputs from which selections are made, a single output, and a set of control lines for making the selection. First we consider selection using multiplexers; then we briefly examine selection circuits implemented by using three-state drivers.

Multiplexers

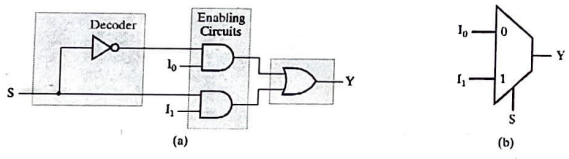
A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs the information to a single output line. The selection of a particular input line is controlled by a set of input variables, called *selection inputs*.

Normally, there are 2^n input lines and n selection inputs whose bit combinations determine which input is selected. We begin with $n = 1$, a 2-to-1-line multiplexer. This function has two information inputs, I_0 and I_1, and a single select input S. The truth table for the circuit is given in Table 3-7. Examining the table, if the select input $S = 0$, the output of the multiplexer takes on the values of I_0, and, if input $S = 1$, the output of the multiplexer takes on the values of I_1. Thus, S selects either input I_0 or input I_1 to appear at output Y. From this discussion, we can see that the equation for the 2-to-1-line multiplexer output Y is

$$Y = \overline{S}I_0 + SI_1$$

□ **TABLE 3-7**
Truth Table for 2-to-1-Line Multiplexer

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



□ **FIGURE 3-24**
(a) Single-Bit 2-to-1-Line Multiplexer; (b) common Symbol for a Multiplexer

This same equation can be obtained by using a 3-variable K-map. As shown in Figure 3-24(a), the implementation of the preceding equation can be decomposed into a 1-to-2-line decoder, two enabling circuits, and a 2-input OR gate. A common symbol for a 2-to-1 multiplexer is shown in Figure 3-24(b), with a trapezoid signifying the selection of the output on the short parallel side from among the 2^n information inputs on the long parallel side.

Suppose that we wish to design a 4-to-1-line multiplexer. In this case, the function Y depends on four inputs I_0, I_1, I_2, and I_3 and two select inputs S_1 and S_0. By placing the values of I_0 through I_3 in the Y column, we can form Table 3-8, a condensed truth table for this multiplexer. In this table, the information variables do not appear as input columns of the table but appear in the output column. Each row represents multiple rows of the full truth table. In Table 3-8, the row 00 I_0 represents all rows in which $(S_1, S_0) = 00$. For $I_0 = 1$ it gives $Y = 1$, and for $I_0 = 0$ it gives $Y = 0$. Since there are six variables, and only S_1 and S_0 are fixed, this single row represents 16 rows of the corresponding full truth table. From the table, we can write the equation for Y as

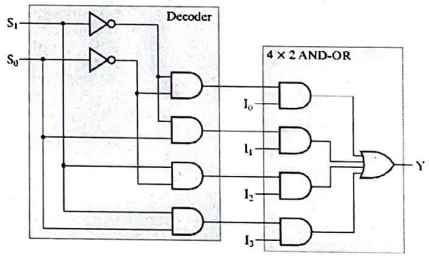
$$Y = \overline{S}_1\overline{S}_0I_0 + \overline{S}_1S_0I_1 + S_1\overline{S}_0I_2 + S_1S_0I_3$$

If this equation is implemented directly, two inverters, four 3-input AND gates, and a 4-input OR gate are required, giving a gate-input cost of 18. A different implementation can be obtained by factoring the AND terms to give

$$Y = (\overline{S}_1\overline{S}_0)I_0 + (\overline{S}_1S_0)I_1 + (S_1\overline{S}_0)I_2 + (S_1S_0)I_3$$

□ **TABLE 3-8**
Condensed Truth Table for 4-to-1-Line Multiplexer

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



□ **FIGURE 3-25**
A Single-Bit 4-to-1-Line Multiplexer

This implementation can be constructed by combining a 2-to-4-line decoder, four AND gates used as enabling circuits, and a 4-input OR gate, as shown in Figure 3-25. We will refer to the combination of AND gates and OR gates as an $m \times 2$ AND-OR, where m is the number of AND gates and 2 is the number of inputs to the AND gates. This resulting circuit has a gate input cost of 22, which is more costly. Nevertheless, it provides a structural basis for constructing larger n-to-2^n-line multiplexers by expansion.

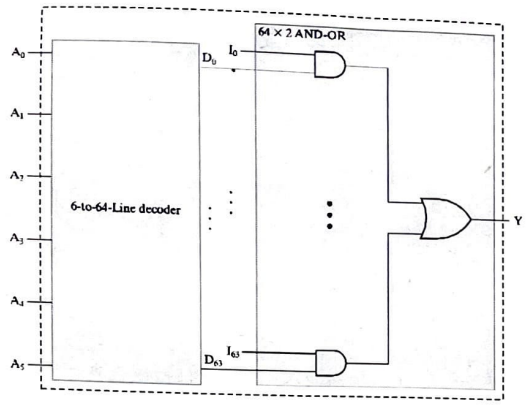
A multiplexer is also called a *data selector*, since it selects one of many information inputs and steers the binary information to the output line. The term "multiplexer" is often abbreviated as "MUX."

Multiplexers can be expanded by considering vectors of input bits for larger values of n. Expansion is based upon the circuit structure given in Figure 3-24(a), consisting of a decoder, enabling circuits, and an OR gate. Multiplexer design is illustrated in Examples 3-10 and 3-11.

EXAMPLE 3-10 64-to-1-Line Multiplexer

A multiplexer is to be designed for $n = 6$. This will require a 6-to-64-line decoder as given in Figure 3-15, and a 64×2 AND-OR gate. The resulting structure is shown in Figure 3-26. This structure has a gate-input cost of $182 + 128 + 64 = 374$.

In contrast, if the decoder and the enabling circuit were replaced by inverters plus 7-input AND gates, the gate-input cost would be $6 + 448 + 64 = 518$. For single-bit multiplexers such as this one, combining the AND gate generating D_i with the AND gate driven by D_i into a single 3-input AND gate for every $i = 0$ through 63 reduces the gate-input cost to 310. For multiple-bit multiplexers, this

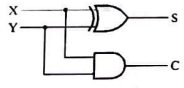


☐ **FIGURE 3-26**
A 64–to–1-Line Multiplexer

reduction to 3-input ANDs cannot be performed without replicating the output ANDs of the decoders. As a result, in almost all cases, the original structure has a lower gate-input cost. The next example illustrates the expansion to a multiple-bit multiplexer. ■

EXAMPLE 3-11 4–to–1-Line Quad Multiplexer

A quad 4–to–1-line multiplexer, which has two selection inputs and each information input replaced by a vector of four inputs, is to be designed. Since the information inputs are a vector, the output Y also becomes a four-element vector. The implementation for this multiplexer requires a 2–to–4-line decoder, as given in Figure 3-13, and four 4 × 2 AND-OR gates. The resulting structure is shown in Figure 3-27. This structure has a gate-input cost of 10 + 32 + 16 = 58. In contrast, if four 4-input multiplexers implemented with 3-input gates were placed side by side, the gate-input cost would be 76. So, by sharing the decoder, we reduced the gate-input cost.



☐ **FIGURE 3-40**
Logic Diagram of Half Adder

The half adder can be implemented with one exclusive-OR gate and one AND gate, as shown in Figure 3-40.

Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. Besides the three inputs, it has 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 bits ranges in value from 0 to 3, and binary 2 and 3 need two digits for their representation. Again, the two outputs are designated by the symbols S for "sum" and C for "carry": the binary variable S gives the value of the bit of the sum, and the binary variable C gives the output carry. The truth table of the full adder is listed in Table 3-12. The values for the outputs are determined from the arithmetic sum of the three input bits. When all the input bits are 0, the outputs are 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 is a carry of 1 if two or three inputs are equal to 1. The maps for the two outputs of the full adder are shown in Figure 3-41. The simplified sum-of-product functions for the two outputs are

$$S = \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$$

$$C = XY + XZ + YZ$$

The two-level implementation requires seven AND gates and two OR gates. However, the map for output S is recognized as an odd function, as discussed in

☐ **TABLE 3-12**
Truth Table of Full Adder

Inputs			Outputs	
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

cell. In particular, the number of connections used and their functions can affect both the cost and speed of an iterative circuit.

In the next section, we will define cells for performing addition in individual bit positions and then define a binary adder as an iterative array of cells.

3-9 Binary Adders

An arithmetic circuit is a combinational circuit that performs arithmetic operations such as addition, subtraction, multiplication, and division with binary numbers or with decimal numbers in a binary code. We will develop arithmetic circuits by means of hierarchical, iterative design. We begin at the lowest level by finding a circuit that performs the addition of two binary digits. This simple addition consists of four possible elementary operations: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum requiring a one-bit representation, but when both the augend and addend are equal to 1, the binary sum requires two bits. Because of this case, the result is always represented by two bits, the carry and the sum. 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 called a *full adder*. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder. The half adder and the full adder are basic arithmetic blocks with which other arithmetic circuits are designed.

Half Adder

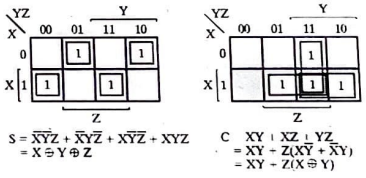
A half adder is an arithmetic circuit that generates the sum of two binary digits. The circuit has two inputs and two outputs. The input variables are the augend and addend bits to be added, and the output variables produce the sum and carry. We assign the symbols X and Y to the two inputs and S (for "sum") and C (for "carry") to the outputs. The truth table for the half adder is listed in Table 3-11. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs, easily obtained from the truth table, are

$$S = \overline{X}Y + X\overline{Y} = X \oplus Y$$

$$C = XY$$

☐ **TABLE 3-11**
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



☐ **FIGURE 3-41**
Maps for Full Adder

Section 2-6. Furthermore, the C output function can be manipulated to include the exclusive-OR of X and Y. The Boolean functions for the full adder in terms of exclusive-OR operations can then be expressed as

$$S = (X \oplus Y) \oplus Z$$

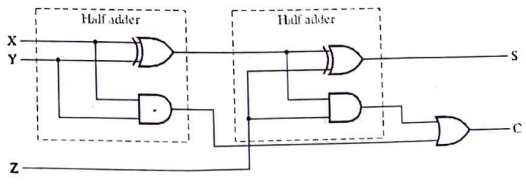
$$C = XY + Z(X \oplus Y)$$

The logic diagram for this multiple-level implementation is shown in Figure 3-42. It consists of two half adders and an OR gate.

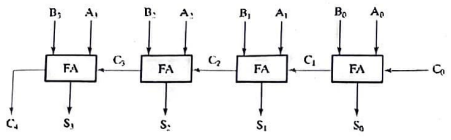
Binary Ripple Carry Adder

A parallel binary adder is a digital circuit that produces the arithmetic sum of two binary numbers using only combinational logic. The parallel adder uses n full adders in parallel, with all input bits applied simultaneously to produce the sum.

The full adders are connected in cascade, with the carry output from one full adder connected to the carry input of the next full adder. Since a 1 carry may appear near the least significant bit of the adder and yet propagate through many full

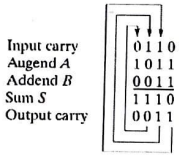


☐ **FIGURE 3-42**
Logic Diagram of Full Adder



□ **FIGURE 3-43**
4-Bit Ripple Carry Adder

adders to the most significant bit, just as a wave ripples outward from a pebble dropped in a pond, the parallel adder is referred to as a *ripple carry adder*. Figure 3-43 shows the interconnection of four full-adder blocks to form a 4-bit ripple carry adder. The augend bits of A and the addend bits of B are designated by subscripts in increasing order from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the parallel adder is C_0, and the output carry is C_4. An n-bit ripple carry adder requires n full adders, with each output carry connected to the input carry of the next-higher-order full adder. For example, consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum, $S = 1110$, is formed with a 4-bit ripple carry adder as follows:



The input carry in the least significant position is 0. Each full adder receives the corresponding bits of A and B and the input carry, and generates the sum bit for S and the output carry. The output carry in each position is the input carry of the next-higher-order position, as indicated by the blue lines.

The 4-bit adder is a typical example of a digital component that can be used as a building block. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the usual method would require a truth table with 512 entries, since there are nine inputs to the circuit. By cascading the four instances of the known full adders, it is possible to obtain a simple and straightforward implementation without directly solving this larger problem. This is an example of the power of iterative circuits and circuit reuse in design.

3-10 BINARY SUBTRACTION

In Chapter 1, we briefly examined the subtraction of unsigned binary numbers. Although beginning texts cover only signed-number addition and subtraction, to the complete exclusion of the unsigned alternative, unsigned-number arithmetic plays an important role in computation and computer hardware design. It is used in floating-point units, in signed-magnitude addition and subtraction algorithms, and in extending the precision of fixed-point numbers. For these reasons, we will treat unsigned-number addition and subtraction here. We also, however, choose to treat it first so that we can clearly justify, in terms of hardware cost, an approach that otherwise appears bizarre and often is accepted on faith, namely, the use of complement representations in arithmetic.

In Section 1-3, subtraction is performed by comparing the subtrahend with the minuend and subtracting the smaller from the larger. The use of a method containing this comparison operation results in inefficient and costly circuitry. As an alternative, we can simply subtract the subtrahend from the minuend. Using the same numbers as in a subtraction example from Section 1-3, we have

Borrows into:	1100
Minuend:	10011
Subtrahend:	−1100110
Difference:	10101
Correct Difference:	− 01011

If no borrow occurs into the most significant position, then we know that the subtrahend is not larger than the minuend and that the result is positive and correct. If a borrow does occur into the most significant position, as indicated in blue, then we know that the subtrahend is larger than the minuend. The result must then be negative, and so we need to correct its magnitude. We can do this by examining the result of the calculation when a borrow occurs:

$$M - N + 2^n$$

Note that the added 2^n represents the value of the borrow into the most significant position. Instead of this result, the desired magnitude is $N - M$. This can be obtained by subtracting the preceding formula from 2^n:

$$2^n - (M - N + 2^n) = N - M$$

In the previous example, $100000 - 10101 = 01011$, which is the correct magnitude.

In general, the subtraction of two n-digit numbers, $M - N$, in base 2 can be done as follows:

1. Subtract the subtrahend N from the minuend M.
2. If no end borrow occurs, then $M \geq N$, and the result is nonnegative and correct.
3. If an end borrow occurs, then $N > M$, and the difference, $M - N + 2^n$, is subtracted from 2^n, and a minus sign is appended to the result.

Subtraction of a binary number from 2^n to obtain an n-digit result is called taking the *2s complement* of the number. So in step 3, we are taking the 2s complement of the difference $M - N + 2^n$. Use of the 2s complement in subtraction is illustrated by the following example.

EXAMPLE 3-19 Unsigned Binary Subtraction by 2s Complement Subtract

Perform the binary subtraction 01100100 − 10010110. We have

Borrows into:	10011110
Minuend:	01100100
Subtrahend:	−10010110
Initial Result:	11001110

The end borrow of 1 implies correction:

2^8	100000000
−Initial Result:	−11001110
Final Result:	− 00110010

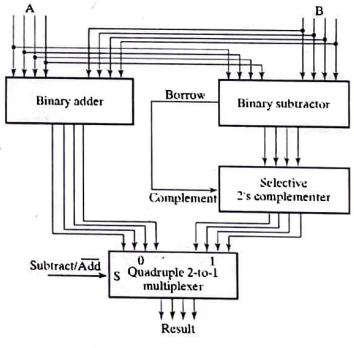
■

To perform subtraction using this method requires a subtractor for the initial subtraction. In addition, when necessary, either the subtractor must be used a second time to perform the correction, or a separate 2s complementer circuit must be provided. So, thus far, we require a subtractor, an adder, and possibly a 2s complementer to perform both addition and subtraction. The block diagram for a 4-bit adder–subtractor using these functional blocks is shown in Figure 3-44. The inputs are applied to both the adder and the subtractor, so both operations are performed in parallel. If an end borrow value of 1 occurs in the subtraction, then the selective 2s complementer receives a value of 1 on its *complement* input. This circuit then takes the 2s complement of the output of the subtractor. If the end borrow has value of 0, the selective 2s complementer passes the output of the subtractor through unchanged. If subtraction is the operation, then a 1 is applied to S of the multiplexer that selects the output of the complementer. If addition is the operation, then a 0 is applied to S, thereby selecting the output of the adder.

As we will see, this circuit is more complex than necessary. To reduce the amount of hardware, we would like to share logic between the adder and the subtractor. This can also be done using the notion of the complement. So before considering the combined adder–subtractor further, we will take a more careful look at complements.

Complements

There are two types of complements for each base-r system: the *radix complement*, which we saw earlier for base 2, and the *diminished radix complement*. The first is referred to as the *r's complement* and the second as the *(r − 1)'s complement*. When the value of the base r is substituted in the names, the two types are referred to as the



□ **FIGURE 3-44**
Block Diagram of Binary Adder–Subtractor

2s and 1s complements for binary numbers and the 10s and 9s complements for decimal numbers, respectively. Since our interest for the present is in binary numbers and operations, we will deal with only 1s and 2s complements.

Given a number N in binary having n digits, the *1s complement of N* is defined as $(2^n - 1) - N$. 2^n is represented by a binary number that consists of a 1 followed by n 0s. $2^n - 1$ is a binary number represented by n 1s. For example, if $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus, the 1s complement of a binary number is obtained by subtracting each digit from 1. When subtracting binary digits from 1, we can have either $1 - 0 = 1$ or $1 - 1 = 0$, which causes the original bit to change from 0 to 1 or from 1 to 0, respectively. Therefore, the 1s complement of a binary number is formed by changing all 1s to 0s and all 0s to 1s—that is, applying the NOT or complement operation to each of the bits. Following are two numerical examples:

The 1s complement of 1011001 is 0100110.
The 1s complement of 0001111 is 1110000.

In similar fashion, the 9s complement of a decimal number, the 7s complement of an octal number, and the 15s complement of a hexadecimal number are obtained by subtracting each digit from 9, 7, and F (decimal 15), respectively.

Given an n-digit number N in binary, the *2s complement of N* is defined as $2^n - N$ for $N \neq 0$ and 0 for $N = 0$. The reason for the special case of $N = 0$ is that the result must have n bits, and subtraction of 0 from 2^n gives an $(n + 1)$-bit result, $100\ldots0$. This special case is achieved by using only an n-bit subtractor or otherwise dropping the 1 in the extra position. Comparing with the 1s complement, we note that the 2s complement can be obtained by adding 1 to the 1s complement, since

$2^n - N = \{[(2^n - 1) - N] + 1\}$. For example, the 2s complement of binary 101100 is 010011 + 1 = 010100 and is obtained by adding 1 to the 1s complement value. Again, for $N = 0$, the result of this addition is 0, achieved by ignoring the carry out of the most significant position of the addition. These concepts hold for other bases as well. As we will see later, they are very useful in simplifying 2s complement and subtraction hardware.

Also, the 2s complement can be formed by leaving all least significant 0s and the first 1 unchanged and then replacing 1s with 0s and 0s with 1s in all other higher significant bits. Thus, the 2s complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0s and the first 1 unchanged and then replacing 1s with 0s and 0s with 1s in the other four most significant bits. In other bases, the first nonzero digit is subtracted from the base r, and the remaining digits to the left are replaced with $r - 1$ minus their values.

It is also worth mentioning that the complement of the complement restores the number to its original value. To see this, note that the 2s complement of N is $2^n - N$, and the complement of the complement is $2^n - (2^n - N) = N$, giving back the original number.

Subtraction Using 2s Complement

Earlier, we expressed a desire to simplify hardware by sharing adder and subtractor logic. Armed with complements, we are prepared to define a binary subtraction procedure that uses addition and the corresponding complement logic. The subtraction of two n-digit unsigned numbers, $M - N$, in binary can be done as follows:

1. Add the 2s complement of the subtrahend N to the minuend M. This performs $M + (2^n - N) = M - N + 2^n$.
2. If $M \geq N$, the sum produces an end carry, 2^n. Discard the end carry, leaving result $M - N$.
3. If $M < N$, the sum does not produce an end carry, since it is equal to $2^n - (N - M)$, the 2s complement of $N - M$. Perform a correction, taking the 2s complement of the sum and placing a minus sign in front to obtain the result $-(N - M)$.

The examples that follow further illustrate the foregoing procedure. Note that, although we are dealing with unsigned numbers, there is no way to get an unsigned result for the case in step 3. When working with paper and pencil, we recognize, by the absence of the end carry, that the answer must be changed to a negative number. If the minus sign for the result is to be preserved, it must be stored separately from the corrected n-bit result.

EXAMPLE 3-20 Unsigned Binary Subtraction by 2s Complement Addition

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction $X - Y$ and $Y - X$ using 2s complement operations. We have

$X =$	1010100
2s complement of $Y =$	0111101
Sum =	10010001
Discard end carry $2^7 =$	−10000000
Answer: $X - Y =$	0010001
$Y =$	1000011
2s complement of $X =$	0101100
Sum =	1101111

There is no end carry.

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

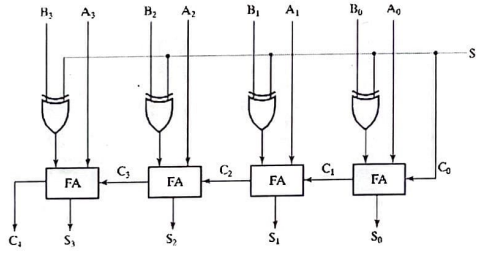
While subtraction of unsigned numbers also can be done by means of the 1s complement, it is little used in modern designs, so will not be covered here.

3-11 Binary Adder-Subtractors

Using the 2s complement, we have eliminated the subtraction operation and need only the complementer and an adder. When performing a subtraction we complement the subtrahend N, and when performing an addition we do not complement N. These operations can be accomplished by using a selective complementer and adder interconnected to form an adder–subtractor. We have used 2s complement, since it is most prevalent in modern systems. The 2s complement can be obtained by taking the 1s complement and adding 1 to the least significant bit. The 1s complement can be implemented easily with inverter circuits, and we can add 1 to the sum by making the input carry of the parallel adder equal to 1. Thus, by using 1s complement and an unused adder input, the 2s complement is obtained inexpensively. In 2s complement subtraction, as a correction step after adding, we complement the result and append a minus sign if an end carry does not occur. The correction operation is performed by using either the adder–subtractor a second time with $M = 0$ or a selective complementer as in Figure 3-44.

The circuit for subtracting $A - B$ consists of a parallel adder as shown in Figure 3-43, with inverters placed between each B terminal and the corresponding full-adder input. The input carry C_0 must be equal to 1. The operation that is performed becomes A plus the 1s complement of B plus 1. This is equal to A plus the 2s complement of B. For unsigned numbers, it gives $A - B$ if $A \geq B$ or the 2s complement of $B - A$ if $A < B$.

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each



□ **FIGURE 3-45**
Adder-Subtractor Circuit

full adder. A 4-bit adder–subtractor circuit is shown in Figure 3-45. Input S controls the operation. When $S = 0$ the circuit is an adder, and when $S = 1$ the circuit becomes a subtractor. Each exclusive-OR gate receives input S and one of the inputs of B, B_i. When $S = 0$, we have $B_i \oplus 0$. If the full adders receive the value of B, and the input carry is 0, the circuit performs A plus B. When $S = 1$, we have $B_i \oplus 1 = \overline{B}_i$ and $C_0 = 1$. In this case, the circuit performs the operation A plus the 2s complement of B.

Signed Binary Numbers

In the previous section, we dealt with the addition and subtraction of unsigned numbers. We will now extend this approach to signed numbers, including a further use of complements that eliminates the correction step.

Positive integers and the number zero can be represented as unsigned numbers. To represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1s and 0s, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the most significant position of an n-bit number. The convention is to make the sign bit 0 for positive numbers and 1 for negative numbers.

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

number or −9 when considered as a signed number. The latter is because the 1 in the leftmost position designates a minus sign and the remaining four bits represent binary 9. Usually, there is no confusion in identifying the bits because the type of number representation is known in advance. The representation of signed numbers just discussed is referred to as the *signed-magnitude* system. In this system, the number consists of a magnitude and a symbol (+ or −) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic.

In implementing signed-magnitude addition and subtraction for n-bit numbers, the single sign bit in the leftmost position and the $n - 1$ magnitude bits are processed separately. The magnitude bits are processed as unsigned binary numbers. Thus, subtraction involves the correction step. To avoid this step, we use a different system for representing negative numbers, referred to as a *signed-complement* system. In this system, a negative number is represented by its complement. While the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement. Since positive numbers always start with 0 (representing a plus sign) in the leftmost position, their complements will always start with a 1, indicating a negative number. The signed-complement system can use either the 1s or the 2s complement, but the latter is the most common. As an example, consider the number 9, represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent of 9, to give 00001001. Note that all eight bits must have a value, and therefore, 0s are inserted between the sign bit and the first 1. Although there is only one way to represent +9, we have two different ways to represent −9 using eight bits:

In signed-magnitude representation:	10001001
In signed 2s complement representation:	11110111

In signed magnitude, −9 is obtained from +9 by changing the sign bit in the leftmost position from 0 to 1. The signed 2s complement representation of −9 is obtained by taking the 2s complement of the positive number, including the 0 sign bit.

Table 3-13 lists all possible 4-bit signed binary numbers in two representations. The equivalent decimal number is also shown. Note that the positive numbers in both representations are identical and have 0 in the leftmost position. The signed 2s complement system has only one representation for 0, which is always positive. The signed-magnitude system has a positive 0 and a negative 0, which is something not encountered in ordinary arithmetic. Note that both negative numbers have a 1 in the leftmost bit position; this is the way we distinguish them from positive numbers. With four bits, we can represent 16 binary numbers. In the signed-magnitude representation, there are seven positive numbers and seven negative numbers, and two signed zeros. In the 2s complement representation, there are seven positive numbers, one zero, and eight negative numbers.

The signed-magnitude system is used in ordinary arithmetic, but is awkward when employed in computer arithmetic due to the separate handling of the sign and the correction step required for subtraction. Therefore, the signed complement is normally used. The following discussion of signed binary arithmetic deals exclusively

□ **TABLE 3-13**
Signed Binary Numbers

Decimal	Signed 2s Complement	Signed Magnitude
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	—	1000
−1	1111	1001
−2	1110	1010
−3	1101	1011
−4	1100	1100
−5	1011	1101
−6	1010	1110
−7	1001	1111
−8	1000	—

with the signed 2s complement representation of negative numbers, because it prevails in actual use.

Signed Binary Addition and Subtraction

The addition of two numbers, $M + N$, in the signed-magnitude system follows the rules of ordinary arithmetic: If the signs are the same, we add the two magnitudes and give the sum the sign of M. If the signs are different, we subtract the magnitude of N from the magnitude of M. The absence or presence of an end borrow then determines the sign of the result, based on the sign of M, and determines whether or not a 2s complement correction is performed. For example, since the signs are different, (00011001) + (10100101) causes 0100101 to be subtracted from 0011001. The result is 1110100, and an end borrow of 1 occurs. The end borrow indicates that the magnitude of M is smaller than that of N. So the sign of the result is opposite to that of M and is therefore a minus. The end borrow indicates that the magnitude of the result, 1110100, must be corrected by taking its 2s complement. Combining the sign and the corrected magnitude of the result, we obtain 1 0001100.

In contrast to this signed-magnitude case, the rule for adding numbers in the signed 2s complement system does not require comparison or subtraction, but only addition. The procedure is simple and can be stated as follows for binary numbers:

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

Numerical examples of signed binary addition are given in Example 3-21. Note that negative numbers will already be in 2s complement form and that the sum obtained after the addition, if negative, is left in that same form.

EXAMPLE 3-21 Signed Binary Addition Using 2s Complement

+6	00000110	−6	11111010	+6	00000110	−6	11111010
+13	00001101	+13	00001101	−13	11110011	−13	11110011
+19	00010011	+7	00000111	−7	11111001	−19	11101101

In each of the four cases, the operation performed is addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2s complement form. ■

The complement form for representing negative numbers is unfamiliar to people accustomed to the signed-magnitude system. To determine the value of a negative number in signed 2s complement, it is necessary to convert the number to a positive number in order to put it in a more familiar form. For example, the signed binary number 11111001 is negative, because the leftmost bit is 1. Its 2s complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original number to be equal to −7.

The subtraction of two signed binary numbers when negative numbers are in 2s complement form is very simple and can be stated as follows:

> Take the 2s complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. That is,

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

But changing a positive number to a negative number is easily done by taking its 2s complement. The reverse is also true, because the complement of a negative number that is already in complement form produces the corresponding positive number. Numerical examples are shown in Example 3-22.

EXAMPLE 3-22 Signed Binary Subtraction Using 2s Complement

−6	11111010	11111010	+6	00000110	00000110
−(−13)	−11110011	+00001101	−(−13)	−11110011	+00001101
+7		00000111	+19		00010011

The end carry is discarded. ■

6-1 REGISTERS AND LOAD ENABLE

A register includes a set of flip-flops. Since each flip-flop is capable of storing one bit of information, an *n*-bit register, composed of *n* flip-flops, is capable of storing *n* bits of binary information. By the broadest definition, a *register* consists of a set of flip-flops, together with gates that implement their state transitions. This broad definition includes the various sequential circuits considered in Chapter 4. More commonly, the term *register* is applied to a set of flip-flops, possibly with added combinational gates, that perform data-processing tasks. The flip-flops hold data, and the gates determine the new or transformed data to be transferred into the flip-flops.

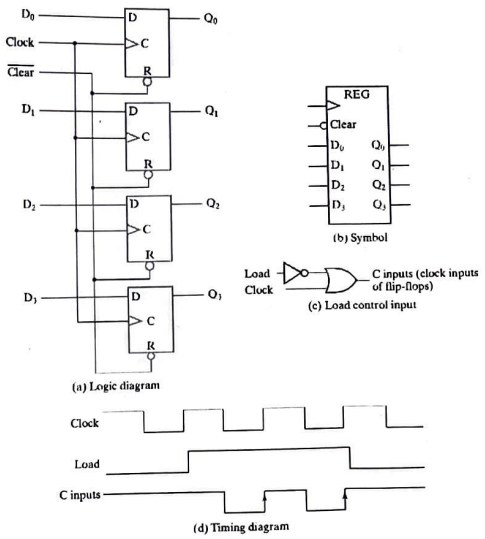
A *counter* is a register that goes through a predetermined sequence of states upon the application of clock pulses. The gates in the counter are connected in a way that produces the prescribed sequence of binary states. Although counters are a special type of registers, it is common to differentiate them from registers.

Registers and counters are sequential functional blocks that are used extensively in the design of digital systems in general and in digital computers in particular. Registers are useful for storing and manipulating information; counters are employed in circuits that sequence and control operations in a digital system.

The simplest register is one that consists of only flip-flops without external gates. Figure 6-1(a) shows such a register constructed from four *D*-type flip-flops. The common *Clock* input triggers all flip-flops on the rising edge of each pulse, and the binary information available at the four *D* inputs is transferred into the 4-bit register. The four *Q* outputs can be sampled to obtain the binary information stored in the register. The $\overline{Clear}$ input goes to the $\overline{R}$ inputs of all four flip-flops and is used to clear the register to all 0s prior to its clocked operation. This input is labeled $\overline{Clear}$ rather than *Clear*, since a 0 must be applied to reset the flip-flops asynchronously. Activation of the asynchronous $\overline{R}$ inputs to flip-flops during normal clocked operation can lead to circuit designs that are highly delay dependent and that can, therefore, malfunction. Thus, we maintain $\overline{Clear}$ at logic 1 during normal clocked operation, allowing it to be logic 0 only when a system reset is desired. We note that the ability to clear a register to all 0s is optional; whether a clear operation is provided depends upon the use of the register in the system.

The transfer of new information into a register is referred to as *loading* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done in parallel. A positive clock transition applied to the *Clock* input of the register of Figure 6-1(a) loads all four *D* inputs into the flip-flops in parallel.

Figure 6-1(b) shows a symbol for the register in Figure 6-1(a). This symbol permits the use of the register in a design hierarchy. It has all inputs to the logic circuit on its left and all outputs from the circuit on the right. The inputs include the clock input with the dynamic indicator to represent positive-edge triggering of the flip-flops. We note that the name *Clear* appears inside the symbol, with a bubble in the signal line on the outside of the symbol. This notation indicates that application of a logic 0 to the signal line activates the clear operation on the flip-flops in the register. If the signal line were labeled outside the symbol, the label would be $\overline{Clear}$.



□ **FIGURE 6-1**
4-Bit Register

Register with Parallel Load

Most digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. In effect, the master clock acts like a heart that supplies a constant beat to all parts of the system. For the design in Figure 6-1(a), the clock can be prevented from reaching the clock input to the circuit if the contents of the register are to be left unchanged. Thus, a separate control signal is used to control the clock cycles during which clock pulses are to have an effect on the register. The clock pulses are prevented from reaching the register when its content is not to be changed. This approach can be implemented with a load control input *Load* combined with the clock, as shown in

Figure 6-1(c). The output of the OR gate is applied to the *C* inputs of the register flip-flops. The equation for the logic shown is

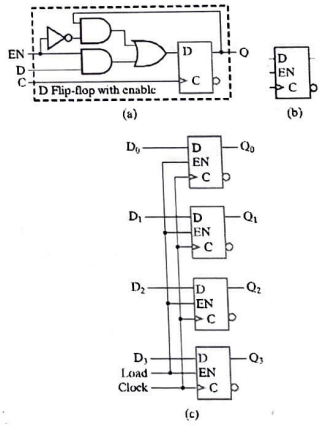
$$C\ inputs = \overline{Load} + Clock$$

When the *Load* signal is 1, *C inputs* = *Clock*, so the register is clocked normally, and new information is transferred into the register on the positive transitions of the clock. When the *Load* signal is 0, *C inputs* = 1. With this constant input applied, there are no positive transitions on *C inputs*, so the contents of the register remain unchanged. The effect of the *Load* signal on the signal *C inputs* is shown in Figure 6-1(d). Note that the clock pulses that appear on *C inputs* are pulses to 0, which end with the positive edge that triggers the flip-flops. These pulses and edges appear when *Load* is 1 and are replaced by a constant 1 when *Load* is 0. In order for this circuit to work correctly, *Load* must be constant at the correct value, either 0 or 1, throughout the interval when *Clock* is 0. One situation in which this occurs is if *Load* comes from a flip-flop that is triggered on a positive edge of *Clock*, a normal circumstance if all flip-flops in the system are positive-edge triggered. Since the clock is turned on and off at the register *C* inputs by the use of a logic gate, the technique is referred to as *clock gating*.

Inserting gates in the clock pulse path produces different propagation delays between *Clock* and the inputs of flip-flops with and without clock gating. If the clock signals arrive at different flip-flops or registers at different times, *clock skew* is said to exist. But to have a truly synchronous system, we must ensure that all clock pulses arrive simultaneously throughout the system so that all flip-flops trigger at the same time. For this reason, in routine designs, control of the operation of the register without using clock gating is advisable. Otherwise, delays must be controlled to drive the clock skew as close to zero as possible. This is applicable in aggressive low-power or high-speed designs.

A 4-bit register with a control input *Load* that is directed through gates into the *D* inputs of the flip-flops, instead of through the *C* inputs, is shown in Figure 6-2(c). This register is based on a bit cell shown in Figure 6-2(a) consisting of a 2-to-1 multiplexer and a *D* flip-flop. The signal *EN* selects between the data bit *D* entering the cell and the value *Q* at the output of the cell. For *EN* = 1, *D* is selected and the cell is loaded. For *EN* = 0, *Q* is selected and the output is loaded back into the flip-flop, preserving its current state. The feedback connection from output to input of the flip-flop is necessary because the *D* flip-flop, unlike other flip-flop types, does not have a "no change" input condition: With each clock pulse, the *D* input determines the next state of the output. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output. The logic in Figure 6-2(a) can be viewed as a new type of *D* flip-flop, a *D flip-flop with enable*, having the symbol shown in Figure 6-2(b).

The register is implemented by placing four *D* flip-flops with enables in parallel and connecting the *Load* input to the *EN* inputs. When *Load* is 1, the data on the four inputs is transferred into the register with the next positive clock edge. When *Load* is 0, the current value remains in the register at the next positive clock edge. Note that the clock pulses are applied continuously to the *C* inputs. *Load* determines whether the next pulse accepts new information or leaves the information in the register intact. The transfer of information from inputs to register is



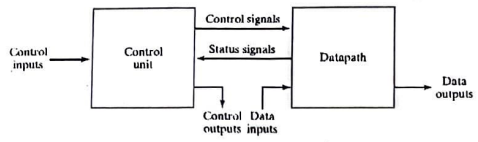
□ **FIGURE 6-2**
4-Bit Register with Parallel Load

done simultaneously for all four bits during a single positive pulse transition. This method of transfer is traditionally preferred over clock gating, since it avoids clock skew and the potential for malfunctions of the circuit.

6-2 REGISTER TRANSFERS

A digital system is a sequential circuit made up of interconnected flip-flops and gates. In Chapter 4, we learned that sequential circuits can be specified by means of state tables. To specify a large digital system with state tables is very difficult, if not impossible, because the number of states is prohibitively large. To overcome this difficulty, digital systems are designed using a modular, hierarchical approach. The system is partitioned into subsystems or modules, each of which performs some functional task. The modules are constructed hierarchically from functional blocks such as registers, counters, decoders, multiplexers, buses, arithmetic elements, flip-flops, and primitive gates. The various subsystems communicate with data and control signals to form a digital system.

In most digital system designs, we partition the system into two types of modules: a *datapath*, which performs data-processing operations, and a *control unit*,



□ **FIGURE 6-3**
Interaction Between Datapath and Control Unit

which determines the sequence of those operations. Figure 6-3 shows the general relationship between a datapath and a control unit. *Control signals* are binary signals that activate the various data-processing operations. To activate a sequence of such operations, the control unit sends the proper sequence of control signals to the datapath. The control unit, in turn, receives status bits from the datapath. These status bits describe aspects of the state of the datapath. The status bits are used by the control unit in defining the specific sequence of the operations to be performed. Note that the datapath and control unit may also interact with other parts of a digital system, such as memory and input–output logic, through the paths labeled data inputs, data outputs, control inputs, and control outputs.

Datapaths are defined by their registers and the operations performed on binary data stored in the registers. Examples of register operations are load, clear, shift, and count. The registers are assumed to be basic components of the digital system. The movement of the data stored in registers and the processing performed on the data are referred to as *register transfer operations*. The register transfer operations of digital systems are specified by the following three basic components:

1. the set of registers in the system,
2. the operations that are performed on the data stored in the registers, and
3. the control that supervises the sequence of operations in the system.

A register has the capability to perform one or more *elementary operations* such as load, count, add, subtract, and shift. For example, a right-shift register is a register that can shift data to the right. A counter is a register that increments a number by one. A single flip-flop is a 1-bit register that can be set or cleared. In fact, by this definition, the flip-flops and closely associated gates of any sequential circuit can be called registers.

An elementary operation performed on data stored in registers is called a *microoperation*. Examples of microoperations are loading the contents of one register into another, adding the contents of two registers, and incrementing the contents of a register. A microoperation is usually, but not always, performed in parallel on a vector of bits during one clock cycle. The result of the microoperation may replace the previous binary data in the register. Alternatively, the result may be transferred to another register, leaving the previous data unchanged. The

sequential functional blocks introduced in this chapter are registers that implement one or more microoperations.

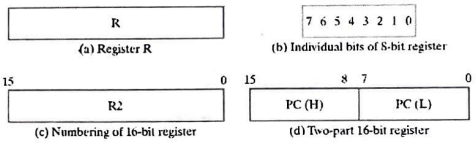
The control unit provides signals that sequence the microoperations in a prescribed manner. The results of a current microoperation may determine both the sequence of control signals and the sequence of future microoperations to be executed. Note that the term "microoperation," as used here, does not refer to any particular way of producing the control signals; specifically, it does not imply that the control signals are generated by a control unit based on a technique called microprogramming.

This chapter introduces registers, their implementations and register transfers using a simple register transfer language (RTL) to represent registers and specify the operations on their contents. The register transfer language uses a set of expressions and statements that resemble statements used in HDLs and programming languages. This notation can concisely specify part or all of a complex digital system such as a computer. The specification then serves as a basis for a more detailed design of the system.

6-3 Register Transfer Operations

We denote the registers in a digital system by uppercase letters (sometimes followed by numerals) that indicate the function of the register. For example, a register that holds an address for the memory unit is usually called an address register and can be designated by the name *AR*. Other designations for registers are *PC* for program counter, *IR* for instruction register, and *R2* for register 2. The individual flip-flops in an n-bit register are typically numbered in sequence from 0 to $n - 1$, starting with 0 in the least significant (often the rightmost) position and increasing toward the most significant position. Since the 0 bit is on the right, this order can be referred to as *little-endian*. The reverse order, with bit 0 on the left, is referred to as *big-endian*.

Figure 6-4 shows representations of registers in block-diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in part (a) of the figure. The individual bits can be identified as in part (b). The numbering of bits represented by just the leftmost and rightmost values at the top of a register box is illustrated by a 16-bit register *R2* in part (c). A 16-bit program counter, *PC*, is partitioned into two sections in part (d) of the figure. In this case, bits 0 through 7 are assigned the symbol *L* (for low-order byte), and bits 8 through 15 are assigned the symbol *H* (for high-order byte). The label *PC*(*L*), which



□ **FIGURE 6-4**
Block Diagrams of Registers

may also be written *PC*(7:0), refers to the low-order byte of the register, and *PC*(*H*) or *PC*(15:8) refers to the high-order byte.

Data transfer from one register to another is designated in symbolic form by means of the replacement operator (←). Thus, the statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register *R1* into register *R2*. Specifically, the statement designates the copying of the contents of *R1* into *R2*. The register *R1* is referred to as the *source* of the transfer and the register *R2* as the *destination*. By definition, the contents of the source register do not change as a result of the transfer—only the contents of the destination register, *R2*, change.

A statement that specifies a register transfer implies that datapath circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals. This can be specified by a *conditional statement*, symbolized by the *if-then* form

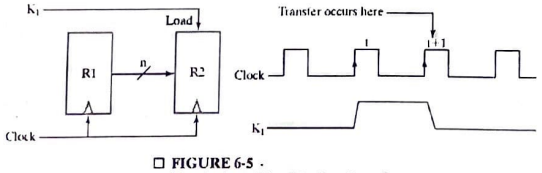
$$\text{if } (K_1 = 1) \text{ then } (R2 \leftarrow R1)$$

where K_1 is a control signal generated in the control unit. In fact, K_1 can be any Boolean function that evaluates to 0 or 1. A more concise way of writing the if-then form is

$$K_1\text{: } R2 \leftarrow R1$$

This control condition, terminated with a colon, symbolizes the requirement that the transfer operation be executed by the hardware only if $K_1 = 1$.

Every statement written in register-transfer notation presupposes a hardware construct for implementing the transfer. Figure 6-5 shows a block diagram that depicts the transfer from *R1* to *R2*. The n outputs of register *R1* are connected to the n inputs of register *R2*. The letter n is used to indicate the number of bits in the register-transfer path from *R1* to *R2*. When the width of the path is known, n is replaced by an actual number. Register *R2* has a load control input that is activated by the control signal K_1. It is assumed that the signal is synchronized with the same clock as the one applied to the register. The flip-flops are assumed to be positive-edge



□ **FIGURE 6-5**
Transfer from *R1* to *R2* when $K_1 = 1$

□ **TABLE 6-1**
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*AR*, *R2*, *DR*, *IR*
Parentheses	Denotes a part of a register	*R2*(1), *R2*(7:0), *AR*(*L*)
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2$, $R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$

triggered by this clock. As shown in the timing diagram, K_1 is set to 1 on the rising edge of a clock pulse at time t. The next positive transition of the clock at time $t + 1$ finds $K_1 = 1$, and the inputs of *R2* are loaded into the register in parallel. In this case, K_1 returns to 0 on the positive clock transition at time $t + 1$, so that only a single transfer from *R1* to *R2* occurs.

Note that the clock is not included as a variable in the register-transfer statements. It is assumed that all transfers occur in response to a clock transition. Even though the control condition K_1 becomes active at time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock, at time $t + 1$.

The basic symbols we use in register-transfer notation are listed in Table 6-1. Registers are denoted by an uppercase letter, possibly followed by one or more uppercase letters and numerals. Parentheses are used to denote a part of a register by specifying the range of bits in the register or by giving a symbolic name to a portion of the register. The left-pointing arrow denotes a transfer of data and the direction of transfer. A comma is used to separate two or more register transfers that are executed at the same time. For example, the statement

$$K_3\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers simultaneously for a positive clock edge at which $K_3 = 1$. Such an exchange is possible with registers made of flip-flops but presents a difficult timing problem with registers made of latches. Square brackets are used in conjunction with a memory transfer. The letter *M* designates a memory word, and the register enclosed inside the square brackets provides the address of the word in memory. This is explained in more detail in Chapter 8.

6-4 Register Transfers in VHDL and Verilog

Although there are some similarities, the register-transfer language used here differs from both VHDL and Verilog. In particular, different notation is used in each of the three languages. Table 6-2 compares the notation for many identical or similar register-transfer operations in the three languages. As you study this chapter and others to follow, this table will assist you in relating descriptions in the text RTL to the corresponding descriptions in VHDL or Verilog.