

Introduction

Number Systems and Conversion

1.1 Digital Systems and Switching Circuits

Digital systems are used extensively in computation and data processing, control systems, communications, and measurement. Because digital systems are capable of greater accuracy and reliability than analog systems, many tasks formerly done by analog systems are now being performed digitally.

In a digital system, the physical quantities or signals can assume only discrete values, while in analog systems the physical quantities or signals may vary continuously over a specified range. For example, the output voltage of a digital system might be constrained to take on only two values such as 0 volts and 5 volts, while the output voltage from an analog system might be allowed to assume any value in the range -10 volts to $+10$ volts.

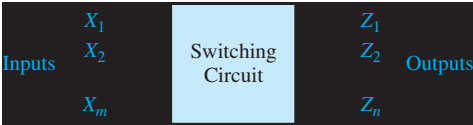
Because digital systems work with discrete quantities, in many cases they can be designed so that for a given input, the output is exactly correct. For example, if we multiply two 5-digit numbers using a digital multiplier, the 10-digit product will be correct in all 10 digits. On the other hand, the output of an analog multiplier might have an error ranging from a fraction of one percent to a few percent depending on the accuracy of the components used in construction of the multiplier. Furthermore, if we need a product which is correct to 20 digits rather than 10, we can redesign the digital multiplier to process more digits and add more digits to its input. A similar improvement in the accuracy of an analog multiplier would not be possible because of limitations on the accuracy of the components.

The design of digital systems may be divided roughly into three parts—system design, logic design, and circuit design. System design involves breaking the overall system into subsystems and specifying the characteristics of each subsystem. For example, the system design of a digital computer could involve specifying the number and type of memory units, arithmetic units, and input-output devices as well as the interconnection and control of these subsystems. Logic design involves determining how to interconnect basic logic building blocks to perform a specific function. An example of logic design is determining the interconnection of logic gates and flip-flops required to perform binary addition. Circuit design involves specifying the interconnection of specific components such as resistors, diodes, and

transistors to form a gate, flip-flop, or other logic building block. Most contemporary circuit design is done in integrated circuit form using appropriate computer-aided design tools to lay out and interconnect the components on a chip of silicon. This book is largely devoted to a study of logic design and the theory necessary for understanding the logic design process. Some aspects of system design are treated in Units 18 and 20. Circuit design of logic gates is discussed briefly in Appendix A.

Many of a digital system’s subsystems take the form of a switching circuit (Figure 1-1). A switching circuit has one or more inputs and one or more outputs which take on discrete values. In this text, we will study two types of switching circuits—combinational and sequential. In a combinational circuit, the output values depend only on the present value of the inputs and not on past values. In a sequential circuit, the outputs depend on both the present and past input values. In other words, in order to determine the output of a sequential circuit, a sequence of input values must be specified. The sequential circuit is said to have memory because it must “remember” something about the past sequence of inputs, while a combinational circuit has no memory. In general, a sequential circuit is composed of a combinational circuit with added memory elements. Combinational circuits are easier to design than sequential circuits and will be studied first.

FIGURE 1-1
Switching Circuit



The basic building blocks used to construct combinational circuits are logic gates. The logic designer must determine how to interconnect these gates in order to convert the circuit input signals into the desired output signals. The relationship between these input and output signals can be described mathematically using Boolean algebra. Units 2 and 3 of this text introduce the basic laws and theorems of Boolean algebra and show how they can be used to describe the behavior of circuits of logic gates.

Starting from a given problem statement, the first step in designing a combinational logic circuit is to derive a table or the algebraic logic equations which describe the circuit outputs as a function of the circuit inputs (Unit 4). In order to design an economical circuit to realize these output functions, the logic equations which describe the circuit outputs generally must be simplified. Algebraic methods for this simplification are described in Unit 3, and other simplification methods (Karnaugh map and Quine-McCluskey procedure) are introduced in Units 5 and 6. Implementation of the simplified logic equations using several types of gates is described in Unit 7, and alternative design procedures using programmable logic devices are developed in Unit 9.

The basic memory elements used in the design of sequential circuits are called flip-flops (Unit 11). These flip-flops can be interconnected with gates to form counters and registers (Unit 12). Analysis of more general sequential circuits using

timing diagrams, state tables, and graphs is presented in Unit 13. The first step in designing a sequential switching circuit is to construct a state table or graph which describes the relationship between the input and output sequences (Unit 14). Methods for going from a state table or graph to a circuit of gates and flip-flops are developed in Unit 15. Methods of implementing sequential circuits using programmable logic are discussed in Unit 16. In Unit 18, combinational and sequential design techniques are applied to the realization of systems for performing binary addition, multiplication, and division. The sequential circuits designed in this text are called synchronous sequential circuits because they use a common timing signal, called a clock, to synchronize the operation of the memory elements.

Use of a hardware description language, VHDL, in the design of combinational logic, sequential logic, and digital systems is introduced in Units 10, 17, and 20. VHDL is used to describe, simulate, and synthesize digital hardware. After writing VHDL code, the designer can use computer-aided design software to compile the hardware description and complete the design of the digital logic. This allows the completion of complex designs without having to manually work out detailed circuit descriptions in terms of gates and flip-flops.

The switching devices used in digital systems are generally two-state devices, that is, the output can assume only two different discrete values. Examples of switching devices are relays, diodes, and transistors. A relay can assume two states—closed or open—depending on whether power is applied to the coil or not. A diode can be in a conducting state or a nonconducting state. A transistor can be in a cut-off or saturated state with a corresponding high or low output voltage. Of course, transistors can also be operated as linear amplifiers with a continuous range of output voltages, but in digital applications greater reliability is obtained by operating them as two-state devices. Because the outputs of most switching devices assume only two different values, it is natural to use binary numbers internally in digital systems. For this reason binary numbers and number systems will be discussed first before proceeding to the design of switching circuits.

1.2 Number Systems and Conversion

When we write decimal (base 10) numbers, we use a positional notation; each digit is multiplied by an appropriate power of 10 depending on its position in the number. For example,

$$953.78_{10} = 9 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$$

Similarly, for binary (base 2) numbers, each binary digit is multiplied by the appropriate power of 2:

$$\begin{aligned} 1011.11_2 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= 8 + 0 + 2 + 1 + \frac{1}{2} + \frac{1}{4} = 11\frac{3}{4} = 11.75_{10} \end{aligned}$$

Note that the binary point separates the positive and negative powers of 2 just as the decimal point separates the positive and negative powers of 10 for decimal numbers.

Any positive integer R ($R > 1$) can be chosen as the *radix* or *base* of a number system. If the base is R , then R digits $(0, 1, \dots, R-1)$ are used. For example, if $R = 8$, then the required digits are 0, 1, 2, 3, 4, 5, 6, and 7. A number written in positional notation can be expanded in a power series in R . For example,

$$\begin{aligned} N &= (a_4 a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3})_R \\ &= a_4 \times R^4 + a_3 \times R^3 + a_2 \times R^2 + a_1 \times R^1 + a_0 \times R^0 \\ &\quad + a_{-1} \times R^{-1} + a_{-2} \times R^{-2} + a_{-3} \times R^{-3} \end{aligned}$$

where a_i is the coefficient of R^i and $0 \leq a_i \leq R-1$. If the arithmetic indicated in the power series expansion is done in base 10, then the result is the decimal equivalent of N . For example,

$$\begin{aligned} 147.3_8 &= 1 \times 8^2 + 4 \times 8^1 + 7 \times 8^0 + 3 \times 8^{-1} = 64 + 32 + 7 + \\ &= 103.375_{10} \end{aligned}$$

The power series expansion can be used to convert to any base. For example, converting 147_{10} to base 3 would be written as

$$147_{10} = 1 \times (101)^2 + (11) \times (101)^1 + (21) \times (101)^0$$

where all the numbers on the right-hand side are base 3 numbers. (*Note:* In base 3, 10 is 101, 7 is 21, etc.) To complete the conversion, base 3 arithmetic would be used. Of course, this is not very convenient if the arithmetic is being done by hand. Similarly, if 147_{10} is being converted to binary, the calculation would be

$$147_{10} = 1 \times (1010)^2 + (100) \times (1010)^1 + (111) \times (1010)^0$$

Again this is not convenient for hand calculation but it could be done easily in a computer where the arithmetic is done in binary. For hand calculation, use the power series expansion when converting from some base *into* base 10.

For bases greater than 10, more than 10 symbols are needed to represent the digits. In this case, letters are usually used to represent digits greater than 9. For example, in hexadecimal (base 16), A represents 10_{10} , B represents 11_{10} , C represents 12_{10} , D represents 13_{10} , E represents 14_{10} , and F represents 15_{10} . Thus,

$$A2F_{16} = 10 \times 16^2 + 2 \times 16^1 + 15 \times 16^0 = 2560 + 32 + 15 = 2607_{10}$$

Next, we will discuss conversion of a decimal *integer* to base R using the division method. The base R equivalent of a decimal integer N can be represented as

$$N = (a_n a_{n-1} \dots a_2 a_1 a_0)_R = a_n R^n + a_{n-1} R^{n-1} + \dots + a_2 R^2 + a_1 R^1 + a_0$$

If we divide N by R , the remainder is a_0 :

$$\frac{N}{R} = a_n R^{n-1} + a_{n-1} R^{n-2} + \cdots + a_2 R^1 + a_1 = Q_1, \text{ remainder } a_0$$

Then we divide the quotient Q_1 by R :

$$\frac{Q_1}{R} = a_n R^{n-2} + a_{n-1} R^{n-3} + \cdots + a_3 R^1 + a_2 = Q_2, \text{ remainder } a_1$$

Next we divide Q_2 by R :

$$\frac{Q_2}{R} = a_n R^{n-3} + a_{n-1} R^{n-4} + \cdots + a_3 = Q_3, \text{ remainder } a_2$$

This process is continued until we finally obtain a_n . Note that the remainder obtained at each division step is one of the desired digits and the least significant digit is obtained first.

Example

Convert 53_{10} to binary.

$$\begin{array}{rcl} 2 \overline{)53} & & \\ 2 \overline{)26} & \text{rem.} = 1 = a_0 & \\ 2 \overline{)13} & \text{rem.} = 0 = a_1 & \\ 2 \overline{)6} & \text{rem.} = 1 = a_2 & 53_{10} = 110101_2 \\ 2 \overline{)3} & \text{rem.} = 0 = a_3 & \\ 2 \overline{)1} & \text{rem.} = 1 = a_4 & \\ 0 & \text{rem.} = 1 = a_5 & \end{array}$$

Conversion of a decimal *fraction* to base R can be done using successive *multiplications* by R . A decimal fraction F can be represented as

$$F = (.a_{-1} a_{-2} a_{-3} \cdots a_{-m})_R = a_{-1} R^{-1} + a_{-2} R^{-2} + a_{-3} R^{-3} + \cdots + a_{-m} R^{-m}$$

Multiplying by R yields

$$FR = a_{-1} + a_{-2} R^{-1} + a_{-3} R^{-2} + \cdots + a_{-m} R^{-m+1} = a_{-1} + F_1$$

where F_1 represents the fractional part of the result and a_{-1} is the integer part. Multiplying F_1 by R yields

$$F_1 R = a_{-2} + a_{-3} R^{-1} + \cdots + a_{-m} R^{-m+2} = a_{-2} + F_2$$

Next, we multiply F_2 by R :

$$F_2R = a_{-3} + \cdots + a_{-m}R^{-m+3} = a_{-3} + F_3$$

This process is continued until we have obtained a sufficient number of digits. Note that the integer part obtained at each step is one of the desired digits and the most significant digit is obtained first.

Example

Convert 0.625_{10} to binary.

$F = .625$	$F_1 = .250$	$F_2 = .500$	$.625_{10} = .101_2$
$\times 2$	$\times 2$	$\times 2$	
<u>1.250</u>	<u>0.500</u>	<u>1.000</u>	
$(a_{-1} = 1)$	$(a_{-2} = 0)$	$(a_{-3} = 1)$	

This process does not always terminate, but if it does not terminate, the result is a repeating fraction.

Example

Convert 0.7_{10} to binary.

.7	
<u>2</u>	
(1).4	
<u>2</u>	
(0).8	
<u>2</u>	
(1).6	
<u>2</u>	
(1).2	
<u>2</u>	
(0).4	← process starts repeating here because 0.4 was previously obtained
<u>2</u>	
(0).8	

$0.7_{10} = 0.1 \underline{0110} \underline{0110} \underline{0110} \dots_2$

Conversion between two bases other than decimal can be done directly by using the procedures given; however, the arithmetic operations would have to be carried out using a base other than 10. It is generally easier to convert to decimal first and then convert the decimal number to the new base.

Example

Convert 231.3_4 to base 7.

$$231.3_4 = 2 \times 16 + 3 \times 4 + 1 + \frac{3}{4} = 45.75_{10}$$

$7 \overline{)45}$		$.75$	
$7 \overline{)6}$	rem. 3	$\underline{7}$	
0	rem. 6	(5) .25	$45.75_{10} = 63.5151 \dots_7$
		$\underline{7}$	
		(1) .75	
		$\underline{7}$	
		(5) .25	
		$\underline{7}$	
		(1) .75	

Conversion from binary to hexadecimal (and conversely) can be done by inspection because each hexadecimal digit corresponds to exactly four binary digits (bits). Starting at the binary point, the bits are divided into groups of four, and each group is replaced by a hexadecimal digit:

$$1001101.010111_2 = \frac{0100}{4} \frac{1101}{D} . \frac{0101}{5} \frac{1100}{C} = 4D.5C_{16} \quad (1-1)$$

As shown in Equation (1-1), extra 0's are added at each end of the bit string as needed to fill out the groups of four bits.

1.3 Binary Arithmetic

Arithmetic operations in digital systems are usually done in binary because design of logic circuits to perform binary arithmetic is much easier than for decimal. Binary arithmetic is carried out in much the same manner as decimal, except the addition and multiplication tables are much simpler.

The addition table for binary numbers is

$0 + 0 = 0$	
$0 + 1 = 1$	
$1 + 0 = 1$	
$1 + 1 = 0$	and carry 1 to the next column

Carrying 1 to a column is equivalent to adding 1 to that column.

Example

Add 13_{10} and 11_{10} in binary.

$$\begin{array}{r} 1111 \leftarrow \text{carries} \\ 13_{10} = 1101 \\ 11_{10} = \underline{1011} \\ 11000 = 24_{10} \end{array}$$

The subtraction table for binary numbers is

$$\begin{array}{l} 0 - 0 = 0 \\ 0 - 1 = 1 \quad \text{and borrow 1 from the next column} \\ 1 - 0 = 1 \\ 1 - 1 = 0 \end{array}$$

Borrowing 1 from a column is equivalent to subtracting 1 from that column.

Examples
of Binary
Subtraction

(a)	$1 \leftarrow$ (indicates a borrow from the 3rd column)	(b)	$1111 \leftarrow$ borrows	(c)	$111 \leftarrow$ borrows
	$\begin{array}{r} 11101 \\ -10011 \\ \hline 1010 \end{array}$		$\begin{array}{r} 10000 \\ - \quad 11 \\ \hline 1101 \end{array}$		$\begin{array}{r} 111001 \\ - \quad 1011 \\ \hline 101110 \end{array}$

Note how the borrow propagates from column to column in the second example. In order to borrow 1 from the second column, we must in turn borrow 1 from the third column, etc. An alternative to binary subtraction is the use of 2's complement arithmetic, as discussed in Section 1.4.

Binary subtraction sometimes causes confusion, perhaps because we are so used to doing decimal subtraction that we forget the significance of the borrowing process. Before doing a detailed analysis of binary subtraction, we will review the borrowing process for decimal subtraction.

If we number the columns (digits) of a decimal integer from right to left (starting with 0), and then we borrow 1 from column n , what we mean is that we subtract 1 from column n and add 10 to column $n - 1$. Because $1 \times 10^n = 10 \times 10^{n-1}$, the value of the decimal number is unchanged, but we can proceed with the subtraction. Consider, for example, the following decimal subtraction problem:

$$\begin{array}{r} \text{column 2} \quad \swarrow \quad \searrow \text{column 1} \\ 205 \\ - 18 \\ \hline 187 \end{array}$$

A detailed analysis of the borrowing process for this example, indicating first a borrow of 1 from column 1 and then a borrow of 1 from column 2, is as follows:

$$\begin{aligned}
 205 - 18 &= [2 \times 10^2 + 0 \times 10^1 + 5 \times 10^0] \\
 &\quad - [\qquad \qquad 1 \times 10^1 + 8 \times 10^0] \\
 &\quad \quad \quad \downarrow \qquad \qquad \downarrow \quad \text{note borrow from column 1} \\
 &= [2 \times 10^2 + (0 - 1) \times 10^1 + (10 + 5) \times 10^0] \\
 &\quad - [\qquad \qquad 1 \times 10^1 + \qquad \qquad 8 \times 10^0] \\
 &\quad \quad \quad \downarrow \qquad \qquad \downarrow \quad \text{note borrow from column 2} \\
 &= [(2 - 1) \times 10^2 + (10 + 0 - 1) \times 10^1 + 15 \times 10^0] \\
 &\quad - [\qquad \qquad \qquad \qquad 1 \times 10^1 + 8 \times 10^0] \\
 &= [1 \times 10^2 \qquad + \qquad 8 \times 10^1 \qquad + \qquad 7 \times 10^0] = 187
 \end{aligned}$$

The analysis of borrowing for binary subtraction is exactly the same, except that we work with powers of 2 instead of powers of 10. Thus for a binary number, borrowing 1 from column n is equivalent to subtracting 1 from column n and adding 2 (10_2) to column $n - 1$. The value of the binary number is unchanged because $1 \times 2^n = 2 \times 2^{n-1}$.

A detailed analysis of binary subtraction example (c) follows. Starting with the rightmost column, $1 - 1 = 0$. To subtract in the second column, we must borrow from the third column. Rather than borrow immediately, we place a 1 over the third column to indicate that a borrow is necessary, and we will actually do the borrowing when we get to the third column. (This is similar to the way borrow signals might propagate in a computer.) Now because we have borrowed 1, the second column becomes 10, and $10 - 1 = 1$. In order to borrow 1 from the third column, we must borrow 1 from the fourth column (indicated by placing a 1 over column 4). Column 3 then becomes 10, subtracting off the borrow yields 1, and $1 - 0 = 1$. Now in column 4, we subtract off the borrow leaving 0. In order to complete the subtraction, we must borrow from column 5, which gives 10 in column 4, and $10 - 1 = 1$.

The multiplication table for binary numbers is

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

The following example illustrates multiplication of 13_{10} by 11_{10} in binary:

$$\begin{array}{r}
 1101 \\
 1011 \\
 \hline
 1101 \\
 1101 \\
 0000 \\
 1101 \\
 \hline
 10001111 = 143_{10}
 \end{array}$$

Note that each partial product is either the multiplicand (1101) shifted over the appropriate number of places or is zero.

When adding up long columns of binary numbers, the sum of the bits in a single column can exceed 11_2 , and therefore the carry to the next column can be greater than 1. For example, if a single column of bits contains five 1's, then adding up the 1's gives 101_2 , which means that the sum bit for that column is 1, and the carry to the next column is 10_2 . When doing binary multiplication, a common way to avoid carries greater than 1 is to add in the partial products one at a time as illustrated by the following example:

1111	multiplicand
<u>1101</u>	multiplier
1111	first partial product
<u>0000</u>	second partial product
(01111)	sum of first two partial products
<u>1111</u>	third partial product
(1001011)	sum after adding third partial product
<u>1111</u>	fourth partial product
11000011	final product (sum after adding fourth partial product)

The following example illustrates division of 145_{10} by 11_{10} in binary:

	1101	
1011	<u>10010001</u>	
	1011	
	1110	
	<u>1011</u>	
	1101	The quotient is 1101 with a remainder
	<u>1011</u>	of 10.
	10	

Binary division is similar to decimal division, except it is much easier because the only two possible quotient digits are 0 and 1. In the above example, if we start by comparing the divisor (1011) with the upper four bits of the dividend (1001), we find that we cannot subtract without a negative result, so we move the divisor one place to the right and try again. This time we can subtract 1011 from 10010 to give 111 as a result, so we put the first quotient bit of 1 above 10010. We then bring down the next dividend bit (0) to get 1110 and shift the divisor right. We then subtract 1011 from 1110 to get 11, so the second quotient bit is 1. When we bring down the next dividend bit, the result is 110, and we cannot subtract the shifted divisor, so the third quotient bit is 0. We then bring down the last dividend bit and subtract 1011 from 1101 to get a final remainder of 10, and the last quotient bit is 1.

1.4 Representation of Negative Numbers

Up to this point we have been working with unsigned positive numbers. In most computers, in order to represent both positive and negative numbers the first bit in a word is used as a sign bit, with 0 used for plus and 1 used for minus. Several representations of negative binary numbers are possible. The *sign and magnitude* system is similar to that which people commonly use. For an n -bit word, the first bit is the sign and the remaining $n - 1$ bits represent the magnitude of the number. Thus an n -bit word can represent any one of 2^{n-1} positive integers or 2^{n-1} negative integers. Table 1-1 illustrates this for $n = 4$. For example, 0011 represents +3 and 1011 represents -3. Note that 1000 represents minus zero in the sign and magnitude system and -8 in the 2's complement system.

The design of logic circuits to do arithmetic with sign and magnitude binary numbers is awkward; therefore, other representations are often used. The 2's complement and 1's complement are commonly used because arithmetic units are easy to design using these systems. For the 2's complement number system, a positive number, N , is represented by a 0 followed by the magnitude as in the sign and magnitude system; however, a negative number, $-N$, is represented by its 2's complement, N^* . If the word length is n bits, the 2's complement of a positive integer N is defined as for a word length of n bits.

$$N^* = 2^n - N \tag{1-2}$$

For $n = 4$, $-N$ is represented by $16 - N$ as shown in Table 1-1. For example, -3 is represented by $16 - 3 = 13 = 1101_2$. As is the case for sign and magnitude numbers, all negative 2's complement numbers have a 1 in the position furthest to the left (sign bit).

For the 1's complement system a negative number, $-N$, is represented by its 1's complement, \bar{N} . The 1's complement of a positive integer N is defined as

$$\bar{N} = (2^n - 1) - N \tag{1-3}$$

TABLE 1-1
Signed Binary
Integers (word
length: $n = 4$)

$+N$	Positive Integers (all systems)	$-N$	Negative Integers		
			Sign and Magnitude	2's Complement N^*	1's Complement \bar{N}
+0	0000	-0	1000	—	1111
+1	0001	-1	1001	1111	1110
+2	0010	-2	1010	1110	1101
+3	0011	-3	1011	1101	1100
+4	0100	-4	1100	1100	1011
+5	0101	-5	1101	1011	1010
+6	0110	-6	1110	1010	1001
+7	0111	-7	1111	1001	1000
		-8	—	1000	—

Note that 1111 represents minus zero, and -8 has no representation in a 4-bit system. An alternate way to form the 1's complement is to simply complement N bit-by-bit by replacing 0's with 1's and 1's with 0's. This is equivalent to the definition, Equation (1-3), because $2^n - 1$ consists of all 1's, and subtracting a bit from 1 is the same as complementing the bit. No borrows occur in this subtraction. For example, if $n = 6$ and $N = 010101$,

$$\begin{array}{r} 2^n - 1 = 111111 \\ N = 010101 \\ \hline \bar{N} = 101010 \end{array}$$

From Equations (1-2) and (1-3).

$$N^* = 2^n - N = (2^n - 1 - N) + 1 = \bar{N} + 1$$

so the 2's complement can be formed by complementing N bit-by-bit and then adding 1. An easier way to form the 2's complement of N is to start at the right and complement all bits to the left of the first 1. For example, if

$$N = 0101100, \text{ then } N^* = 1010100$$

From Equations (1-2) and (1-3),

$$N = 2^n - N^* \quad \text{and} \quad N = (2^n - 1) - \bar{N}$$

Therefore, given a negative integer represented by its 2's complement (N^*), we can obtain the magnitude of the integer by taking the 2's complement of N^* . Similarly, to get the magnitude of a negative integer represented by its 1's complement (\bar{N}), we can take the 1's complement of \bar{N} .

In the 2's complement system the number of negative integers which can be represented is one more than the number of positive integers (not including 0). For example, in Table 1-1, 1000 represents -8 , because a sign bit of 1 indicates a negative number, and if $N = 8$, $N^* = 10000 - 1000 = 1000$. In general, in a 2's complement system with a word length of n bits, the number $100 \dots 000$ (1 followed by $n - 1$ 0's) represents a negative number with a magnitude of

$$2^n - 2^{n-1} = 2^{n-1}$$

This special case occurs only for 2's complement. However, -0 has no representation in 2's complement, but -0 is a special case for 1's complement as well as for the sign and magnitude system.

Addition of 2's Complement Numbers

The addition of n -bit signed binary numbers is straightforward using the 2's complement system. The addition is carried out just as if all the numbers were positive, and any carry from the sign position is ignored. This will always yield the correct result except when an overflow occurs. When the word length is n bits, we say that an

overflow has occurred if the correct representation of the sum (including sign) requires more than n bits. The different cases which can occur are illustrated below for $n = 4$.

1. Addition of two positive numbers, $\text{sum} < 2^{n-1}$

$$\begin{array}{r} +3 \quad 0011 \\ +4 \quad 0100 \\ \hline +7 \quad 0111 \end{array} \quad (\text{correct answer})$$

2. Addition of two positive numbers, $\text{sum} \geq 2^{n-1}$

$$\begin{array}{r} +5 \quad 0101 \\ +6 \quad 0110 \\ \hline 1011 \end{array} \quad \leftarrow \text{wrong answer because of overflow (+11 requires 5 bits including sign)}$$

3. Addition of positive and negative numbers (negative number has greater magnitude)

$$\begin{array}{r} +5 \quad 0101 \\ -6 \quad 1010 \\ \hline -1 \quad 1111 \end{array} \quad (\text{correct answer})$$

4. Same as case 3 except positive number has greater magnitude

$$\begin{array}{r} -5 \quad 1011 \\ +6 \quad 0110 \\ \hline +1 \quad (1)0001 \end{array} \quad \leftarrow \text{correct answer when the carry from the sign bit is ignored (this is *not* an overflow)}$$

5. Addition of two negative numbers, $|\text{sum}| \leq 2^{n-1}$

$$\begin{array}{r} -3 \quad 1101 \\ -4 \quad 1100 \\ \hline -7 \quad (1)1001 \end{array} \quad \leftarrow \text{correct answer when the last carry is ignored (this is *not* an overflow)}$$

6. Addition of two negative numbers, $|\text{sum}| > 2^{n-1}$

$$\begin{array}{r} -5 \quad 1011 \\ -6 \quad 1010 \\ \hline (1)0101 \end{array} \quad \leftarrow \text{wrong answer because of overflow (-11 requires 5 bits including sign)}$$

Note that an overflow condition (cases 2 and 6) is easy to detect because in case 2 the addition of two positive numbers yields a negative result, and in case 6 the addition of two negative numbers yields a positive answer (for four bits).

The proof that throwing away the carry from the sign bit always gives the correct answer follows for cases 4 and 5:

Case 4: $-A + B$ (where $B > A$)

$$A^* + B = (2^n - A) + B = 2^n + (B - A) > 2^n$$

Throwing away the last carry is equivalent to subtracting 2^n , so the result is $(B - A)$, which is correct.

Case 5: $-A - B$ (where $A + B \leq 2^{n-1}$)

$$A^* + B^* = (2^n - A) + (2^n - B) = 2^n + 2^n - (A + B)$$

Discarding the last carry yields $2^n - (A + B) = (A + B)^*$, which is the correct representation of $-(A + B)$.

Addition of 1's Complement Numbers

The addition of 1's complement numbers is similar to 2's complement except that instead of discarding the last carry, it is added to the n -bit sum in the position furthest to the right. This is referred to as an *end-around* carry. The addition of positive numbers is the same as illustrated for cases 1 and 2 under 2's complement. The remaining cases are illustrated below ($n = 4$).

3. Addition of positive and negative numbers (negative number with greater magnitude)

$$\begin{array}{r} +5 \quad 0101 \\ -6 \quad 1001 \\ \hline -1 \quad 1110 \end{array} \quad (\text{correct answer})$$

4. Same as case 3 except positive number has greater magnitude

$$\begin{array}{r} -5 \quad 1010 \\ +6 \quad 0110 \\ \hline (1) \quad 0000 \\ \quad \quad \quad \hookrightarrow 1 \quad \text{(end-around carry)} \\ \quad \quad \quad 0001 \quad \text{(correct answer, no overflow)} \\ \hline \end{array}$$

- 5.** Addition of two negative numbers, $|\text{sum}| < 2^{n-1}$

$$\begin{array}{r} -3 \quad 1100 \\ -4 \quad 1011 \\ \hline (1) \quad 0111 \\ \quad \hookrightarrow 1 \quad \text{(end-around carry)} \\ \hline \quad 1000 \quad \text{(correct answer, no overflow)} \end{array}$$

- 6.** Addition of two negative numbers, $|\text{sum}| \geq 2^{n-1}$

$$\begin{array}{r} -5 \quad 1010 \\ -6 \quad 1001 \\ \hline (1) \quad 0011 \\ \quad \hookrightarrow 1 \quad (\text{end-around carry}) \\ \quad \quad 0100 \\ \hline \end{array}$$

(wrong answer because of overflow)

Again, note that the overflow in case 6 is easy to detect because the addition of two negative numbers yields a positive result.

The proof that the end-round carry method gives the correct result follows for cases 4 and 5:

$$\begin{aligned} \text{Case 4: } & -A + B \quad (\text{where } B > A) \\ \overline{A} + B &= (2^n - 1 - A) + B = 2^n + (B - A) - 1 \end{aligned}$$

The end-around carry is equivalent to subtracting 2^n and adding 1, so the result is $(B - A)$, which is correct.

$$\begin{aligned} \text{Case 5: } & -A - B \quad (A + B < 2^{n-1}) \\ \overline{A} + \overline{B} &= (2^n - 1 - A) + (2^n - 1 - B) = 2^n + [2^n - 1 - (A + B)] - 1 \end{aligned}$$

After the end-around carry, the result is $2^n - 1 - (A + B) = \overline{(A + B)}$ which is the correct representation for $-(A + B)$.

The following examples illustrate the addition of 1's and 2's complement numbers for a word length of $n = 8$:

1. Add -11 and -20 in 1's complement.

$$+11 = 00001011 \qquad +20 = 00010100$$

taking the bit-by-bit complement,

-11 is represented by 11110100 and -20 by 11101011

$$\begin{array}{r} 11110100 \quad (-11) \\ 11101011 \quad +(-20) \\ \hline (1) 11011111 \\ \xrightarrow{\quad} 1 \quad (\text{end-around carry}) \\ \hline 11100000 = -31 \end{array}$$

2. Add -8 and $+19$ in 2's complement

$$+8 = 00001000$$

complementing all bits to the left of the first 1, -8 , is represented by 11111000

$$\begin{array}{r} 11111000 \quad (-8) \\ 00010011 \quad +19 \\ \hline (1) 00001011 = +11 \\ \uparrow \quad (\text{discard last carry}) \end{array}$$

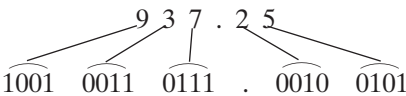
Note that in both cases, the addition produced a carry out of the furthest left bit position, but there is no overflow because the answer can be correctly

represented by eight bits (including sign). A general rule for detecting overflow when adding two n -bit signed binary numbers (1's or 2's complement) to get an n -bit sum is:

An overflow occurs if adding two positive numbers gives a negative answer or if adding two negative numbers gives a positive answer.

1.5 Binary Codes

Although most large computers work internally with binary numbers, the input-output equipment generally uses decimal numbers. Because most logic circuits only accept two-valued signals, the decimal numbers must be coded in terms of binary signals. In the simplest form of binary code, each decimal digit is replaced by its binary equivalent. For example, 937.25 is represented by



This representation is referred to as binary-coded-decimal (BCD) or more explicitly as 8-4-2-1 BCD. Note that the result is quite different than that obtained by converting the number as a whole into binary. Because there are only ten decimal digits, 1010 through 1111 are not valid BCD codes.

Table 1-2 shows several possible sets of binary codes for the ten decimal digits. Many other possibilities exist because the only requirement for a

TABLE 1-2
Binary Codes for
Decimal Digits

Decimal Digit	8-4-2-1 Code (BCD)	6-3-1-1 Code	Excess-3 Code	2-out-of-5 Code	Gray Code
0	0000	0000	0011	00011	0000
1	0001	0001	0100	00101	0001
2	0010	0011	0101	00110	0011
3	0011	0100	0110	01001	0010
4	0100	0101	0111	01010	0110
5	0101	0111	1000	01100	1110
6	0110	1000	1001	10001	1010
7	0111	1001	1010	10010	1011
8	1000	1011	1011	10100	1001
9	1001	1100	1100	11000	1000

valid code is that each decimal digit be represented by a distinct combination of binary digits. To translate a decimal number to coded form, each decimal digit is replaced by its corresponding code. Thus 937 expressed in excess-3 code is 1100 0110 1010. The 8-4-2-1 (BCD) code and the 6-3-1-1 code are examples of weighted codes. A 4-bit weighted code has the property that if the weights are w_3 , w_2 , w_1 , and w_0 , the code $a_3a_2a_1a_0$ represents a decimal number N , where

$$N = w_3a_3 + w_2a_2 + w_1a_1 + w_0a_0$$

For example, the weights for the 6-3-1-1 code are $w_3 = 6$, $w_2 = 3$, $w_1 = 1$, and $w_0 = 1$. The binary code 1011 thus represents the decimal digit

$$N = 6 \cdot 1 + 3 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 = 8$$

The excess-3 code is obtained from the 8-4-2-1 code by adding 3 (0011) to each of the codes. The 2-out-of-5 code has the property that exactly 2 out of the 5 bits are 1 for every valid code combination. This code has useful error-checking properties because if any one of the bits in a code combination is changed due to a malfunction of the logic circuitry, the number of 1 bits is no longer exactly two. The table shows one example of a Gray code. A Gray code has the property that the codes for successive decimal digits differ in exactly one bit. For example, the codes for 6 and 7 differ only in the fourth bit, and the codes for 9 and 0 differ only in the first bit. A Gray code is often used when translating an analog quantity, such as a shaft position, into digital form. In this case, a small change in the analog quantity will change only one bit in the code, which gives more reliable operation than if two or more bits changed at a time. The Gray and 2-out-of-5 codes are *not* weighted codes. In general, the decimal value of a coded digit *cannot* be computed by a simple formula when a non-weighted code is used.

Many applications of computers require the processing of data which contains numbers, letters, and other symbols such as punctuation marks. In order to transmit such alphanumeric data to or from a computer or store it internally in a computer, each symbol must be represented by a binary code. One common alphanumeric code is the ASCII code (American Standard Code for Information Interchange). This is a 7-bit code, so 2^7 (128) different code combinations are available to represent letters, numbers, and other symbols. Table 1-3 shows a portion of the ASCII code; the code combinations not listed are used for special control functions such as “form feed” or “end of transmission.” The word “*Start*” is represented in ASCII code as follows:

1010011	1110100	1100001	1110010	1110100
S	t	a	r	t

TABLE 1-3 ASCII Code

ASCII Code								ASCII Code								ASCII Code							
Character	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	Character	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	Character	A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀
space	0	1	0	0	0	0	0	@	1	0	0	0	0	0	0	'	1	1	0	0	0	0	0
!	0	1	0	0	0	0	1	A	1	0	0	0	0	0	1	a	1	1	0	0	0	0	1
"	0	1	0	0	0	1	0	B	1	0	0	0	0	1	0	b	1	1	0	0	0	1	0
#	0	1	0	0	0	1	1	C	1	0	0	0	0	1	1	c	1	1	0	0	0	1	1
\$	0	1	0	0	1	0	0	D	1	0	0	0	1	0	0	d	1	1	0	0	1	0	0
%	0	1	0	0	1	0	1	E	1	0	0	0	1	0	1	e	1	1	0	0	1	0	1
&	0	1	0	0	1	1	0	F	1	0	0	0	1	1	0	f	1	1	0	0	1	1	0
'	0	1	0	0	1	1	1	G	1	0	0	0	1	1	1	g	1	1	0	0	1	1	1
(0	1	0	1	0	0	0	H	1	0	0	1	0	0	0	h	1	1	0	1	0	0	0
)	0	1	0	1	0	0	1	I	1	0	0	1	0	0	1	i	1	1	0	1	0	0	1
*	0	1	0	1	0	1	0	J	1	0	0	1	0	1	0	j	1	1	0	1	0	1	0
+	0	1	0	1	0	1	1	K	1	0	0	1	0	1	1	k	1	1	0	1	0	1	1
,	0	1	0	1	1	0	0	L	1	0	0	1	1	0	0	l	1	1	0	1	1	0	0
—	0	1	0	1	1	0	1	M	1	0	0	1	1	0	1	m	1	1	0	1	1	0	1
.	0	1	0	1	1	1	0	N	1	0	0	1	1	1	0	n	1	1	0	1	1	1	0
/	0	1	0	1	1	1	1	O	1	0	0	1	1	1	1	o	1	1	0	1	1	1	1
0	0	1	1	0	0	0	0	P	1	0	1	0	0	0	0	p	1	1	1	0	0	0	0
1	0	1	1	0	0	0	1	Q	1	0	1	0	0	0	1	q	1	1	1	0	0	0	1
2	0	1	1	0	0	1	0	R	1	0	1	0	0	1	0	r	1	1	1	0	0	1	0
3	0	1	1	0	0	1	1	S	1	0	1	0	0	1	1	s	1	1	1	0	0	1	1
4	0	1	1	0	1	0	0	T	1	0	1	0	1	0	0	t	1	1	1	0	1	0	0
5	0	1	1	0	1	0	1	U	1	0	1	0	1	0	1	u	1	1	1	0	1	0	1
6	0	1	1	0	1	1	0	V	1	0	1	0	1	1	0	v	1	1	1	0	1	1	0
7	0	1	1	0	1	1	1	W	1	0	1	0	1	1	1	w	1	1	1	0	1	1	1
8	0	1	1	1	0	0	0	X	1	0	1	1	0	0	0	x	1	1	1	1	0	0	0
9	0	1	1	1	0	0	1	Y	1	0	1	1	0	0	1	y	1	1	1	1	0	0	1
:	0	1	1	1	0	1	0	Z	1	0	1	1	0	1	0	z	1	1	1	1	0	1	0
;	0	1	1	1	0	1	1	[1	0	1	1	0	1	1	{	1	1	1	1	0	1	1
<	0	1	1	1	1	0	0	\	1	0	1	1	1	0	0		1	1	1	1	1	0	0
=	0	1	1	1	1	0	1]	1	0	1	1	1	0	1	}	1	1	1	1	1	0	1
>	0	1	1	1	1	1	0	^	1	0	1	1	1	1	0	~	1	1	1	1	1	1	0
?	0	1	1	1	1	1	1	—	1	0	1	1	1	1	1	delete	1	1	1	1	1	1	1

Problems

- 1.1 Convert to hexadecimal and then to binary:
(a) 757.25_{10} (b) 123.17_{10} (c) 356.89_{10} (d) 1063.5_{10}
- 1.2 Convert to octal. Convert to hexadecimal. Then convert both of your answers to decimal, and verify that they are the same.
(a) 111010110001.011_2 (b) 10110011101.11_2

- 1.3** Convert to base 6: $3BA.25_{14}$ (do all of the arithmetic in decimal).
- 1.4** (a) Convert to hexadecimal: 1457.11_{10} . Round to two digits past the hexadecimal point.
 (b) Convert your answer to binary, and then to octal.
 (c) Devise a scheme for converting hexadecimal directly to base 4 and convert your answer to base 4.
 (d) Convert to decimal: $DEC.A_{16}$.
- 1.5** Add, subtract, and multiply in binary:
 (a) 1111 and 1010 (b) 110110 and 11101 (c) 100100 and 10110
- 1.6** Subtract in binary. Place a 1 over each column from which it was necessary to borrow.
 (a) $11110100 - 1000111$ (b) $1110110 - 111101$ (c) $10110010 - 111101$
- 1.7** Add the following numbers in binary using 2's complement to represent negative numbers. Use a word length of 6 bits (including sign) and indicate if an overflow occurs.
 (a) $21 + 11$ (b) $(-14) + (-32)$ (c) $(-25) + 18$
 (d) $(-12) + 13$ (e) $(-11) + (-21)$
 Repeat (a), (c), (d), and (e) using 1's complement to represent negative numbers.
- 1.8** A computer has a word length of 8 bits (including sign). If 2's complement is used to represent negative numbers, what range of integers can be stored in the computer? If 1's complement is used? (Express your answers in decimal.)
- 1.9** Construct a table for 7-3-2-1 weighted code and write 3659 using this code.
- 1.10** Convert to hexadecimal and then to binary.
 (a) 1305.375_{10} (b) 111.33_{10} (c) 301.12_{10} (d) 1644.875_{10}
- 1.11** Convert to octal. Convert to hexadecimal. Then convert both of your answers to decimal, and verify that they are the same.
 (a) 101111010100.101_2 (b) 100001101111.01_2
- 1.12** (a) Convert to base 3: 375.54_8 (do all of the arithmetic in decimal).
 (b) Convert to base 4: 384.74_{10} .
 (c) Convert to base 9: $A52.A4_{11}$ (do all of the arithmetic in decimal).
- 1.13** Convert to hexadecimal and then to binary: 544.1_9 .
- 1.14** Convert the decimal number 97.7_{10} into a number with exactly the same value represented in the following bases. The exact value requires an infinite repeating part in the fractional part of the number. Show the steps of your derivation.
 (a) binary (b) octal (c) hexadecimal (d) base 3 (e) base 5
- 1.15** Devise a scheme for converting base 3 numbers directly to base 9. Use your method to convert the following number to base 9: 1110212.20211_3

- 1.16** Convert the following decimal numbers to octal and then to binary:
 (a) $2983^{63}/_{64}$ (b) 93.70 (c) $1900^{31}/_{32}$ (d) 109.30
- 1.17** Add, subtract, and multiply in binary:
 (a) 1111 and 1001 (b) 1101001 and 110110 (c) 110010 and 11101
- 1.18** Subtract in binary. Place a 1 over each column from which it was necessary to borrow.
 (a) $10100100 - 01110011$ (b) $10010011 - 01011001$
 (c) $11110011 - 10011110$
- 1.19** Divide in binary:
 (a) $11101001 \div 101$ (b) $110000001 \div 1110$ (c) $1110010 \div 1001$
 Check your answers by multiplying out in binary and adding the remainder.
- 1.20** Divide in binary:
 (a) $10001101 \div 110$ (b) $110000011 \div 1011$ (c) $1110100 \div 1010$
- 1.21** Assume three digits are used to represent positive integers and also assume the following operations are correct. Determine the base of the numbers. Did any of the additions overflow?
 (a) $654 + 013 = 000$
 (b) $024 + 043 + 013 + 033 = 223$
 (c) $024 + 043 + 013 + 033 = 201$
- 1.22** What is the lowest number of bits (digits) required in the binary number approximately equal to the decimal number 0.6117_{10} so that the binary number has the same or better precision?
- 1.23** Convert $0.363636..._{10}$ to its exact equivalent base 8 number.
- 1.24** (a) Verify that a number in base b can be converted to base b^3 by partitioning the digits of the base b number into groups of three consecutive digits starting at the radix point and proceeding both left and right and converting each group into a base b^3 digit. (*Hint:* Represent the base b number using the power series expansion.)
 (b) Verify that a number in base b^3 can be converted to base b by expanding each digit of the base b^3 number into three consecutive digits starting at the radix point and proceeding both left and right.
- 1.25** Construct a table for 4-3-2-1 weighted code and write 9154 using this code.
- 1.26** Is it possible to construct a 5-3-1-1 weighted code? A 6-4-1-1 weighted code? Justify your answers.
- 1.27** Is it possible to construct a 5-4-1-1 weighted code? A 6-3-2-1 weighted code? Justify your answers.

- 1.28** Construct a 6-2-2-1 weighted code for decimal digits. What number does 1100 0011 represent in this code?
- 1.29** Construct a 5-2-2-1 weighted code for decimal digits. What numbers does 1110 0110 represent in this code?
- 1.30** Construct a 7-3-2-1 code for base 12 digits. Write B4A9 using this code.
- 1.31** (a) It is possible to have negative weights in a weighted code for the decimal digits, e.g., 8, 4, -2 , and -1 can be used. Construct a table for this weighted code.
 (b) If d is a decimal digit in this code, how can the code for $9 - d$ be obtained?
- 1.32** Convert to hexadecimal, and then give the ASCII code for the resulting hexadecimal number (including the code for the hexadecimal point):
 (a) 222.22_{10} (b) 183.81_{10}
- 1.33** Repeat 1.7 for the following numbers:
 (a) $(-10) + (-11)$ (b) $(-10) + (-6)$ (c) $(-8) + (-11)$
 (d) $11 + 9$ (e) $(-11) + (-4)$
- 1.34** Because $A - B = A + (-B)$, the subtraction of signed numbers can be accomplished by adding the complement. Subtract each of the following pairs of 5-bit binary numbers by adding the complement of the subtrahend to the minuend. Indicate when an overflow occurs. Assume that negative numbers are represented in 1's complement. Then repeat using 2's complement.
- | | | | | |
|---------------|---------------|---------------|---------------|---------------|
| (a) 01001 | (b) 11010 | (c) 10110 | (d) 11011 | (e) 11100 |
| <u>-11010</u> | <u>-11001</u> | <u>-01101</u> | <u>-00111</u> | <u>-10101</u> |
- 1.35** Work Problem 1.34 for the following pairs of numbers:
- | | | | |
|---------------|---------------|---------------|---------------|
| (a) 11010 | (b) 01011 | (c) 10001 | (d) 10101 |
| <u>-10100</u> | <u>-11000</u> | <u>-01010</u> | <u>-11010</u> |
- 1.36** (a) $A = 101010$ and $B = 011101$ are 1's complement numbers. Perform the following operations and indicate whether overflow occurs.
 (i) $A + B$ (ii) $A - B$
 (b) Repeat Part (a) assuming the numbers are 2's complement numbers.
- 1.37** (a) Assume the integers below are 1's complement integers. Find the 1's complement of each number, and give the decimal values of the original number and of its complement.
 (i) 0000000 (ii) 1111111 (iii) 00110011 (iv) 1000000
 (b) Repeat, assuming the numbers are 2's complement numbers and finding the 2's complement of them.

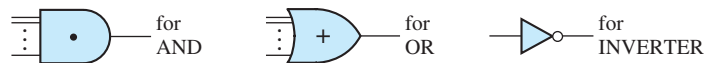
Objectives

A list of 15 laws and theorems of Boolean algebra is given on page 55 of this unit. When you complete this unit, you should be familiar with and be able to use any of the first 12 of these. Specifically, you should be able to:

1. Understand the basic operations and laws of Boolean algebra.
2. Relate these operations and laws to circuits composed of AND gates, OR gates, and INVERTERS. Also relate these operations and laws to circuits composed of switches.
3. Prove any of these laws using a truth table.
4. Apply these laws to the manipulation of algebraic expressions including:
 - a. Multiplying out an expression to obtain a sum of products (SOP).
 - b. Factoring an expression to obtain a product of sums (POS).
 - c. Simplifying an expression by applying one of the laws.
 - d. Finding the complement of an expression.

Study Guide

1. In this unit you will study Boolean algebra, the basic mathematics needed for the logic design of digital systems. Just as when you first learned ordinary algebra, you will need a fair amount of practice before you can use Boolean algebra effectively. However, by the end of the course, you should be just as comfortable with Boolean algebra as with ordinary algebra. Fortunately, many of the rules of Boolean algebra are the same as for ordinary algebra, but watch out for some surprises!
2. Study Sections 2.1 and 2.2, *Introduction and Basic Operations*.
 - (a) How does the meaning of the symbols 0 and 1 as used in this unit differ from the meaning as used in Unit 1?
 - (b) Two commonly used notations for the inverse or complement of A are \bar{A} and A' . The latter has the advantage that it is much easier for typists, printers, and computers. (Have you ever tried to get a computer to print a bar over a letter?) We will use A' for the complement of A . You may use either notation in your work, but please do not mix notations in the same equation. Most engineers use $+$ for OR and \cdot (or no symbol) for AND, and we will follow this practice. An alternative notation, often used by mathematicians, is \vee for OR and \wedge for AND.
 - (c) Many different symbols are used for AND, OR, and INVERTER logic blocks. Initially we will use

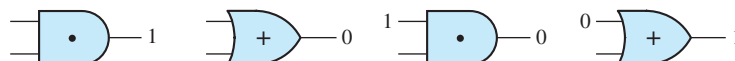


The shapes of these symbols conform to those commonly used in industrial practice. We have added the $+$ and \cdot for clarity. These symbols point in the direction of signal flow. This makes it easier to read the circuit diagrams in comparison with the square or round symbols used in some books.

- (d) Determine the output of each of the following gates:



- (e) Determine the unspecified inputs to each of the following gates if the outputs are as shown:

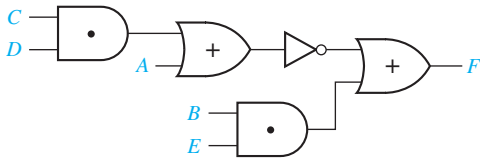


3. Study Section 2.3, *Boolean Expressions and Truth Tables*.

- (a) How many *variables* does the following expression contain?
How many *literals*?

$$A'BC'D + AB + B'CD + D'$$

- (b) For the following circuit, if $A = B = 0$ and $C = D = E = 1$, indicate the output of each gate (0 or 1) on the circuit diagram:



- (c) Derive a Boolean expression for the circuit output. Then substitute $A = B = 0$ and $C = D = E = 1$ into your expression and verify that the value of F obtained in this way is the same as that obtained on the circuit diagram in (b).

- (d) Write an expression for the output of the following circuit and complete the truth table:



A	B	A'	A'B	(A'B)'

- (e) When filling in the combinations of values for the variables on the left side of a truth table, always list the combinations of 0's and 1's in binary order. For example, for a three-variable truth table, the first row should be 000, the next row 001, then 010, 011, 100, 101, 110, and 111. Write an expression for the output of the following circuit and complete the truth table:



A	B	C	B'	A+B'	C(A+B')

- (f) Draw a gate circuit which has an output

$$Z = [BC' + F(E + AD')]'$$

(Hint: Start with the innermost parentheses and draw the circuit for AD' first.)

4. Study Section 2.4, *Basic Theorems*.

- (a) Prove each of the Theorems (2-4) through (2-8D) by showing that it is valid for both $X = 0$ and $X = 1$.
- (b) Determine the output of each of these gates:



- (c) State which of the basic theorems was used in simplifying each of the following expressions:

$$(AB' + C) \cdot 0 = 0$$

$$A(B + C') + 1 = 1$$

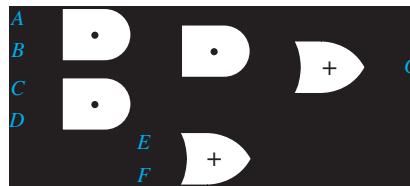
$$(BC' + A)(BC' + A) = BC' + A \quad X(Y' + Z) + [X(Y' + Z)]' = 1$$

$$(X' + YZ)(X' + YZ)' = 0$$

$$D'(E' + F) + D'(E' + F) = D'(E' + F)$$

5. Study Section 2.5, *Commutative, Associative, and Distributive Laws*.

- (a) State the associative law for OR.
- (b) State the commutative law for AND.
- (c) Simplify the following circuit by using the associative laws. Your answer should require only two gates.



- (d) For each gate determine the value of the unspecified input(s):



- (e) Using a truth table, verify the distributive law, Equation (2-11).

- (f) Illustrate the distributive laws, Equations (2-11) and (2-11D), using AND and OR gates.
- (g) Verify Equation (2-3) using the second distributive law.
- (h) Show how the second distributive law can be used to factor $RS + T'$.
6. Study Section 2.6, *Simplification Theorems*.
- (a) By completing the truth table, prove that $XY' + Y = X + Y$.

X	Y	XY'	$XY' + Y$	$X + Y$
0	0			
0	1			
1	0			
1	1			

- (b) Which one of Theorems (2-12) through (2-14D) was applied to simplify each of the following expressions? Identify X and Y in each case.

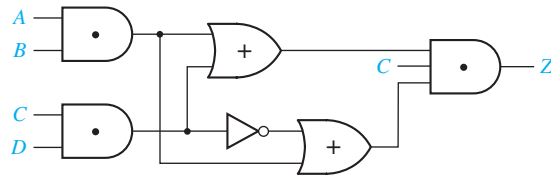
$$(A + B)(DE)' + DE = A + B + DE$$

$$AB' + AB'C'D = AB'$$

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

$$(A + BC' + D'E)(A + D'E) = A + D'E$$

- (c) Simplify the following circuit to a single gate:



- (d) Work Problems 2.1, 2.2, 2.3, and 2.4.

7. Study Section 2.7, *Multiplying Out and Factoring*.

- (a) Indicate which of the following expressions are in the product-of-sums form, sum-of-products form, or neither:

$$AB' + D'EF' + G$$

$$(A + B'C')(A' + BC)$$

$$AB'(C' + D + E')(F' + G)$$

$$X'Y + WX(X' + Z) + A'B'C'$$

Your answer to this question should include one product-of-sums, one sum-of-products, and two neither, not necessarily in that order.

- (b) When multiplying out an expression, why should the second distributive law be applied before the ordinary distributive law when possible?

- (c) Factor as much as possible using the ordinary distributive law:

$$AD + B'CD + B'DE$$

Now factor your result using the second distributive law to obtain a product of sums.

- (d) Work Problems 2.5, 2.6, and 2.7.

8. Probably the most difficult part of the unit is using the second distributive law for factoring or multiplying out an expression. If you have difficulty with Problems 2.5 or 2.6, or you cannot work them *quickly*, study the examples in Section 2.7 again, and then work the following problems.

Multiply out:

- (a) $(B' + D + E)(B' + D + A)(AE + C')$

(b) $(A + C')(B' + D)(C' + D')(C + D)E$

As usual, when we say multiply out, we do not mean to multiply out by brute force, but rather to use the second distributive law whenever you can to cut down on the amount of work required.

The answer to (a) should be of the following form: $XX + XX + XX$ and (b) of the form: $XXX + XXXX$, where each X represents a single variable or its complement.

Now factor your answer to (a) to see that you can get back the original expression.

9. Study Section 2.8, *DeMorgan's Laws*.

10. Find the complement of each of the following expressions as indicated. In your answer, the complement operation should be applied only to single variables.

(a) $(ab'c')' =$

(b) $(a' + b + c + d')' =$

(c) $(a' + bc)' =$

(d) $(a'b' + cd)' =$

(e) $[a(b' + c'd)]' =$

11. Because $(X')' = X$, if you complement each of your answers to 10, you should get back the original expression. Verify that this is true.

(a)

(b)

(c)

(d)

(e)

12. Given that $F = a'b + b'c$, $F' =$

Complete the following truth table and verify that your answer is correct:

a	b	c	$a'b$	$b'c$	$a'b + b'c$	$(a + b')$	$(b + c')$	F'
0	0	0						
0	0	1						
0	1	0						
0	1	1						
1	0	0						
1	0	1						
1	1	0						
1	1	1						

13. A fully simplified expression should have nothing complemented except the individual variables. For example, $F = (X + Y)'(W + Z)$ is *not* a minimum product of sums. Find the minimum product of sums for F .
14. Work Problems 2.8 and 2.9.
15. Find the dual of $(M + N')P'$.
16. Review the first 12 laws and theorems on page 55. Make sure that you can recognize when to apply them even if an expression has been substituted for a variable.
17. Reread the objectives of this unit. If you are satisfied that you can meet these objectives, take the readiness test.
[Note: You will be provided with a copy of the theorem sheet (page 55) when you take the readiness test this time. However, by the end of Unit 3, you should know all the theorems by memory.]



Boolean Algebra

2.1 Introduction

The basic mathematics needed for the study of the logic design of digital systems is Boolean algebra. Boolean algebra has many other applications including set theory and mathematical logic, but we will restrict ourselves to its application to switching circuits in this text. Because all of the switching devices which we will use are essentially two-state devices (such as a transistor with high or low output voltage), we will study the special case of Boolean algebra in which all of the variables assume only one of two values. This two-valued Boolean algebra is often referred to as switching algebra. George Boole developed Boolean algebra in 1847 and used

it to solve problems in mathematical logic. Claude Shannon first applied Boolean algebra to the design of switching circuits in 1939.

We will use a Boolean variable, such as X or Y , to represent the input or output of a switching circuit. We will assume that each of these variables can take on only two different values. The symbols “0” and “1” are used to represent these two different values. Thus, if X is a Boolean (switching) variable, then either $X = 0$ or $X = 1$.

The symbols “0” and “1” used in Boolean algebra do not have a numeric value; instead they represent two different states in a logic circuit and are the two values of a switching variable. In a logic gate circuit, 0 (usually) represents a range of low voltages, and 1 represents a range of high voltages. In a switch circuit, 0 (usually) represents an open switch, and 1 represents a closed circuit. In general, 0 and 1 can be used to represent the two states in any binary-valued system.

2.2 Basic Operations

The basic operations of Boolean algebra are AND, OR, and complement (or inverse). The complement of 0 is 1, and the complement of 1 is 0. Symbolically, we write

$$0' = 1 \quad \text{and} \quad 1' = 0$$

where the prime (') denotes complementation. If X is a switching variable,

$$X' = 1 \text{ if } X = 0 \quad \text{and} \quad X' = 0 \text{ if } X = 1$$

An alternate name for complementation is inversion, and the electronic circuit which forms the inverse of X is referred to as an inverter. Symbolically, we represent an inverter by



where the circle at the output indicates inversion. If a logic 0 corresponds to a low voltage and a logic 1 corresponds to a high voltage, a low voltage at the inverter input produces a high voltage at the output and vice versa. Complementation is sometimes referred to as the NOT operation because $X = 1$ if X is *not* equal to 0.

The AND operation can be defined as follows:

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

where “ \cdot ” denotes AND. (Although this looks like binary multiplication, it is not, because 0 and 1 here are Boolean constants rather than binary numbers.) If we write the Boolean expression $C = A \cdot B$, then given the values of A and B , we can determine C from the following table:

$A \ B$	$C = A \cdot B$
0 0	0
0 1	0
1 0	0
1 1	1

Note that $C = 1$ iff (if and only if) A and B are both 1, hence, the name AND operation. A logic gate which performs the AND operation is represented by



The dot symbol (\cdot) is frequently omitted in a Boolean expression, and we will usually write AB instead of $A \cdot B$. The AND operation is also referred to as logical (or Boolean) multiplication.

The OR operation can be defined as follows:

$$0 + 0 = 0 \quad 0 + 1 = 1 \quad 1 + 0 = 1 \quad 1 + 1 = 1$$

where “ $+$ ” denotes OR. If we write $C = A + B$, then given the values of A and B , we can determine C from the following table:

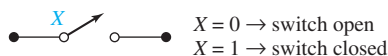
A	B	$C = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Note that $C = 1$ iff A or B (or both) is 1, hence, the name OR operation. This type of OR operation is sometimes referred to as inclusive-OR. A logic gate which performs the OR operation is represented by

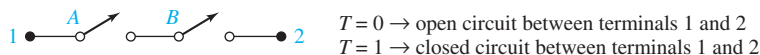


The OR operation is also referred to as logical (or Boolean) addition. Electronic circuits which realize inverters and AND and OR gates are described in Appendix A.

Next, we will apply switching algebra to describe circuits containing switches. We will label each switch with a variable. If switch X is open, then we will define the value of X to be 0; if switch X is closed, then we will define the value of X to be 1.



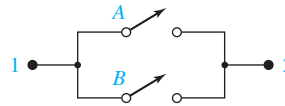
Now consider a circuit composed of two switches in a series. We will define the transmission between the terminals as $T = 0$ if there is an open circuit between the terminals and $T = 1$ if there is a closed circuit between the terminals.



Now we have a closed circuit between terminals 1 and 2 ($T = 1$) iff (if and only if) switch A is closed and switch B is closed. Stating this algebraically,

$$T = A \cdot B$$

Next consider a circuit composed of two switches in parallel.



In this case, we have a closed circuit between terminals 1 and 2 iff switch A is closed *or* switch B is closed. Using the same convention for defining variables as above, an equation which describes the behavior of this circuit is

$$T = A + B$$

Thus, switches in a series perform the AND operation and switches in parallel perform the OR operation.

2.3 Boolean Expressions and Truth Tables

Boolean expressions are formed by application of the basic operations to one or more variables or constants. The simplest expressions consist of a single constant or variable, such as 0, X, or Y' . More complicated expressions are formed by combining two or more other expressions using AND or OR, or by complementing another expression. Examples of expressions are

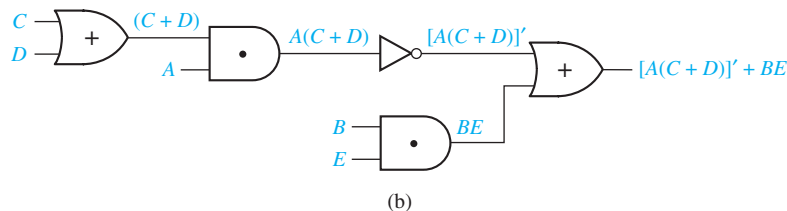
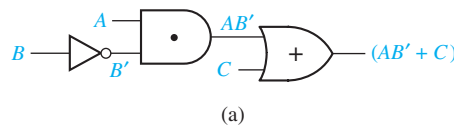
$$AB' + C \quad (2-1)$$

$$[A(C + D)]' + BE \quad (2-2)$$

Parentheses are added as needed to specify the order in which the operations are performed. When parentheses are omitted, complementation is performed first followed by AND and then OR. Thus in Expression (2-1), B' is formed first, then AB' , and finally $AB' + C$.

Each expression corresponds directly to a circuit of logic gates. Figure 2-1 gives the circuits for Expressions (2-1) and (2-2).

FIGURE 2-1
Circuits for
Expressions (2-1)
and (2-2)



An expression is evaluated by substituting a value of 0 or 1 for each variable. If $A = B = C = 1$ and $D = E = 0$, the value of Expression (2-2) is

$$[A(C + D)]' + BE = [1(1 + 0)]' + 1 \cdot 0 = [1(1)]' + 0 = 0 + 0 = 0$$

Each appearance of a variable or its complement in an expression will be referred to as a *literal*. Thus, the following expression, which has three variables, has 10 literals:

$$ab'c + a'b + a'bc' + b'c'$$

When an expression is realized using logic gates, each literal in the expression corresponds to a gate input.

A *truth table* (also called a table of combinations) specifies the values of a Boolean expression for every possible combination of values of the variables in the expression. The name truth table comes from a similar table which is used in symbolic logic to list the truth or falsity of a statement under all possible conditions. We can use a truth table to specify the output values for a circuit of logic gates in terms of the values of the input variables. The output of the circuit in Figure 2-2(a) is $F = A' + B$. Figure 2-2(b) shows a truth table which specifies the output of the circuit for all possible combinations of values of the inputs A and B . The first two columns list the four combinations of values of A and B , and the next column gives the corresponding values of A' . The last column, which gives the values of $A' + B$, is formed by ORing together corresponding values of A' and B in each row.

FIGURE 2-2
Two-Input Circuit
and Truth Table



A	B	A'	F = A' + B
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Next, we will use a truth table to specify the value of Expression (2-1) for all possible combinations of values of the variables A , B , and C . On the left side of Table 2-1, we list the values of the variables A , B , and C . Because each of the three variables can assume the value 0 or 1, there are $2 \times 2 \times 2 = 8$ combinations of values of the variables. These combinations are easily obtained by listing the binary numbers 000, 001, ..., 111. In the next three columns of the truth table, we compute B' , AB' , and $AB' + C$, respectively.

Two expressions are equal if they have the same value for every possible combination of the variables. The expression $(A + C)(B' + C)$ is evaluated using the last three columns of Table 2-1. Because it has the same value as $AB' + C$ for all eight combinations of values of the variables A , B , and C , we conclude

TABLE 2-1

A	B	C	B'	AB'	AB' + C	A + C	B' + C	(A + C)(B' + C)
0	0	0	1	0	0	0	1	0
0	0	1	1	0	1	1	1	1
0	1	0	0	0	0	0	0	0
0	1	1	0	0	1	1	1	1
1	0	0	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
1	1	0	0	0	0	1	0	0
1	1	1	0	0	1	1	1	1

$$AB' + C = (A + C)(B' + C) \quad (2-3)$$

If an expression has n variables, and each variable can have the value 0 or 1, the number of different combinations of values of the variables is

$$\underbrace{2 \times 2 \times 2 \times \dots}_{n \text{ times}} = 2^n$$

Therefore, a truth table for an n -variable expression will have 2^n rows.

2.4 Basic Theorems

The following basic laws and theorems of Boolean algebra involve only a single variable:
Operations with 0 and 1:

$$X + 0 = X \quad (2-4) \quad X \cdot 1 = X \quad (2-4D)$$

$$X + 1 = 1 \quad (2-5) \quad X \cdot 0 = 0 \quad (2-5D)$$

Idempotent laws

$$X + X = X \quad (2-6) \quad X \cdot X = X \quad (2-6D)$$

Involution law

$$(X')' = X \quad (2-7)$$

Laws of complementarity

$$X + X' = 1 \quad (2-8) \quad X \cdot X' = 0 \quad (2-8D)$$

Each of these theorems is easily proved by showing that it is valid for both of the possible values of X . For example, to prove $X + X' = 1$, we observe that if

$$X = 0, \quad 0 + 0' = 0 + 1 = 1, \quad \text{and if } X = 1, \quad 1 + 1' = 1 + 0 = 1$$

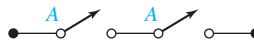
Any expression can be substituted for the variable X in these theorems. Thus, by Theorem (2-5),

$$(AB' + D)E + 1 = 1$$

and by Theorem (2-8D),

$$(AB' + D)(AB' + D)' = 0$$

We will illustrate some of the basic theorems with circuits of switches. As before, 0 will represent an open circuit or open switch, and 1 will represent a closed circuit or closed switch. If two switches are both labeled with the variable A , this means that both switches are open when $A = 0$ and both are closed when $A = 1$. Thus the circuit



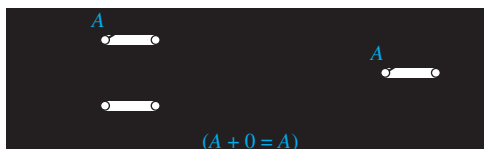
can be replaced with a single switch:



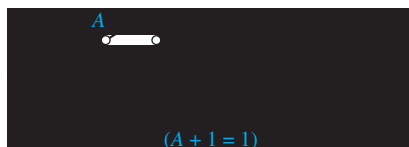
This illustrates the theorem $A \cdot A = A$. Similarly,



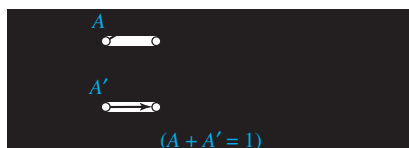
which illustrates the theorem $A + A = A$. A switch in parallel with an open circuit is equivalent to the switch alone



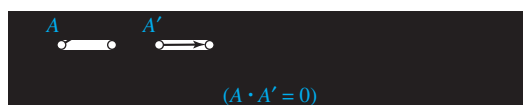
while a switch in parallel with a short circuit is equivalent to a short circuit.



If a switch is labeled A' , then it is open when A is closed and conversely. Hence, A in parallel with A' can be replaced with a closed circuit because one or the other of the two switches is always closed.



Similarly, switch A in series with A' can be replaced with an open circuit (why?).



2.5 Commutative, Associative, and Distributive Laws

Many of the laws of ordinary algebra, such as the commutative and associative laws, also apply to Boolean algebra. The commutative laws for AND and OR, which follow directly from the definitions of the AND and OR operations, are

$$XY = YX \quad (2-9) \qquad X + Y = Y + X \quad (2-9D)$$

This means that the order in which the variables are written will not affect the result of applying the AND and OR operations.

The associative laws also apply to AND and OR:

$(XY)Z = X(YZ) = XYZ$ (2-10)

$(X + Y) + Z = X + (Y + Z) = X + Y + Z$ (2-10D)

When forming the AND (or OR) of three variables, the result is independent of which pair of variables we associate together first, so parentheses can be omitted as indicated in Equations (2-10) and (2-10D).

We will prove the associative law for AND by using a truth table (Table 2-2). On the left side of the table, we list all combinations of values of the variables X, Y, and Z. In the next two columns of the truth table, we compute XY and YZ for each combination of values of X, Y, and Z. Finally, we compute (XY)Z and X(YZ). Because (XY)Z and X(YZ) are equal for all possible combinations of values of the variables, we conclude that Equation (2-10) is valid.

TABLE 2-2

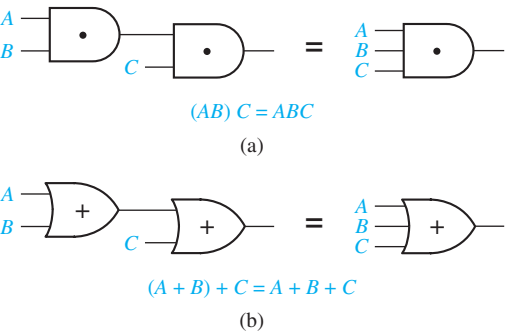
Proof of Associative Law for AND

X	Y	Z	XY	YZ	(XY)Z	X(YZ)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Figure 2-3 illustrates the associative laws using AND and OR gates. In Figure 2-3(a) two two-input AND gates are replaced with a single three-input AND gate. Similarly, in Figure 2-3(b) two two-input OR gates are replaced with a single three-input OR gate.

FIGURE 2-3

Associative Laws for AND and OR



When two or more variables are ANDed together, the value of the result will be 1 iff all of the variables have the value 1. If any of the variables have the value 0, the result of the AND operation will be 0. For example,

$XYZ = 1 \text{ iff } X = Y = Z = 1$

When two or more variables are ORed together, the value of the result will be 1 if any of the variables have the value 1. The result of the OR operation will be 0 iff all of the variables have the value 0. For example,

$$X + Y + Z = 0 \text{ iff } X = Y = Z = 0$$

Using a truth table, it is easy to show that the distributive law is valid:

$$X(Y + Z) = XY + XZ \quad (2-11)$$

In addition to the ordinary distributive law, a second distributive law is valid for Boolean algebra but not for ordinary algebra:

$$X + YZ = (X + Y)(X + Z) \quad (2-11D)$$

Proof of the second distributive law follows:

$$\begin{aligned} (X + Y)(X + Z) &= X(X + Z) + Y(X + Z) = XX + XZ + YX + YZ \\ &\quad \text{(by (2-11))} \\ &= X + XZ + XY + YZ = X \cdot 1 + XZ + XY + YZ \\ &\quad \text{(by (2-6D) and (2-4D))} \\ &= X(1 + Z + Y) + YZ = X \cdot 1 + YZ = X + YZ \\ &\quad \text{(by (2-11), (2-5), and (2-4D))} \end{aligned}$$

The ordinary distributive law states that the AND operation distributes over OR, while the second distributive law states that OR distributes over AND. This second law is very useful in manipulating Boolean expressions. In particular, an expression like $A + BC$, which cannot be factored in ordinary algebra, is easily factored using the second distributive law:

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

2.6 Simplification Theorems

The following theorems are useful in simplifying Boolean expressions:

$$XY + XY' = X \quad (2-12) \qquad (X + Y)(X + Y') = X \quad (2-12D)$$

$$X + XY = X \quad (2-13) \qquad X(X + Y) = X \quad (2-13D)$$

$$(X + Y')Y = XY \quad (2-14) \qquad XY' + Y = X + Y \quad (2-14D)$$

In each case, one expression can be replaced by a simpler one. Because each expression corresponds to a circuit of logic gates, simplifying an expression leads to simplifying the corresponding logic circuit.

Each of the preceding theorems can be proved by using a truth table, or they can be proved algebraically starting with the basic theorems.

Proof of (2-13): $X + XY = X \cdot 1 + XY = X(1 + Y) = X \cdot 1 = X$

Proof of (2-13D): $X(X + Y) = XX + XY = X + XY = X$
(by (2-6D) and (2-13))

Proof of (2-14D): $Y + XY' = (Y + X)(Y + Y') = (Y + X)1 = Y + X$
(by (2-11 D) and (2-8))

The proof of the remaining theorems is left as an exercise.

We will illustrate Theorem (2-14D), using switches. Consider the following circuit:



Its transmission is $T = Y + XY'$ because there is a closed circuit between the terminals if switch Y is closed *or* switch X is closed and switch Y' is closed. The following circuit is equivalent because if Y is closed ($Y = 1$) both circuits have a transmission of 1; if Y is open ($Y' = 1$) both circuits have a transmission of X .

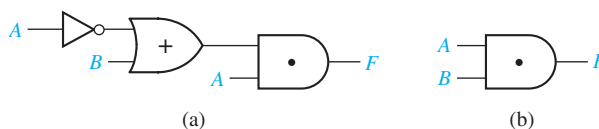


The following example illustrates simplification of a logic gate circuit using one of the theorems. In Figure 2-4, the output of circuit (a) is

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

By Theorem (2-14), the expression for F simplifies to AB . Therefore, circuit (a) can be replaced with the equivalent circuit (b).

FIGURE 2-4
Equivalent Gate
Circuits



Any expressions can be substituted for X and Y in the theorems.

Simplify $Z = A'BC + A'$

This expression has the same form as (2-13) if we let $X = A'$ and $Y = BC$. Therefore, the expression simplifies to $Z = X + XY = X = A'$.

Example 1

Example 2

$$\text{Simplify } Z = \underbrace{[A + B'C + D + EF]}_{X} \underbrace{[A + B'C + (D + EF)']}_{Y'}$$

$$\text{Substituting: } Z = [X + Y'] [X + Y']$$

Then, by (2-12D), the expression reduces to

$$Z = X = A + B'C$$

Example 3

$$\text{Simplify } Z = \underbrace{(AB + C)}_{Y'} \underbrace{(B'D + C'E')}_X + \underbrace{(AB + C)'}_Y$$

$$\text{Substituting: } Z = Y'X + Y$$

$$\text{By, (2-14D): } Z = X + Y = B'D + C'E' + (AB + C)'$$

Note that in this example we let $Y = (AB + C)'$ rather than $(AB + C)$ in order to match the form of (2-14D).

2.7 Multiplying Out and Factoring

The two distributive laws are used to multiply out an expression to obtain a sum-of-products (SOP) form. An expression is said to be in *sum-of-products* form when all products are the products of single variables. This form is the end result when an expression is fully multiplied out. It is usually easy to recognize a sum-of-products expression because it consists of a sum of product terms:

$$AB' + CD'E + AC'E' \quad (2-15)$$

However, in degenerate cases, one or more of the product terms may consist of a single variable. For example,

$$ABC' + DEFG + H \quad (2-16)$$

and

$$A + B' + C + D'E \quad (2-17)$$

are still considered to be in sum-of-products form. The expression

$$(A + B)CD + EF$$

is not in sum-of-products form because the $A + B$ term enters into a product but is not a single variable.

When multiplying out an expression, apply the second distributive law first when possible. For example, to multiply out $(A + BC)(A + D + E)$ let

$$X = A, \quad Y = BC, \quad Z = D + E$$

Then

$$(X + Y)(X + Z) = X + YZ = A + BC(D + E) = A + BCD + BCE$$

Of course, the same result could be obtained the hard way by multiplying out the original expression completely and then eliminating redundant terms:

$$\begin{aligned}(A + BC)(A + D + E) &= A + AD + AE + ABC + BCD + BCE \\ &= A(1 + D + E + BC) + BCD + BCE \\ &= A + BCD + BCE\end{aligned}$$

You will save yourself a lot of time if you learn to apply the second distributive law instead of doing the problem the hard way.

Both distributive laws can be used to factor an expression to obtain a product-of-sums form. An expression is in *product-of-sums* (POS) form when all sums are the sums of single variables. It is usually easy to recognize a product-of-sums expression since it consists of a product of sum terms:

$$(A + B')(C + D' + E)(A + C' + E') \quad (2-18)$$

However, in degenerate cases, one or more of the sum terms may consist of a single variable. For example,

$$(A + B)(C + D + E)F \quad (2-19)$$

and

$$AB'C(D' + E) \quad (2-20)$$

are still considered to be in product-of-sums form, but $(A + B)(C + D) + EF$ is not. An expression is fully factored iff it is in product-of-sums form. Any expression not in this form can be factored further.

The following examples illustrate how to factor using the second distributive law:

Example 1

Factor $A + B'CD$. This is of the form $X + YZ$ where $X = A$, $Y = B'$, and $Z = CD$, so

$$A + B'CD = (X + Y)(X + Z) = (A + B')(A + CD)$$

$A + CD$ can be factored again using the second distributive law, so

$$A + B'CD = (A + B')(A + C)(A + D)$$

Example 2

Factor $AB' + C'D$.

$$\begin{aligned}AB' + C'D &= (AB' + C')(AB' + D) \quad \leftarrow \text{note how } X + YZ = (X + Y)(X + Z) \text{ was applied here} \\ &= (A + C')(B' + C')(A + D)(B' + D) \quad \leftarrow \text{the second distributive law was applied again to each term}\end{aligned}$$

Example 3Factor $C'D + C'E' + G'H$.

$$C'D + C'E' + G'H = C'(D + E') + G'H$$

← first apply the ordinary distributive law, $XY + XZ = X(Y + Z)$

$$= (C' + G'H)(D + E' + G'H)$$

← then apply the second distributive law

$$= (C' + G')(C' + H)(D + E' + G')(D + E' + H)$$

← now identify X, Y, and Z in each expression and complete the factoring

As in Example 3, the ordinary distributive law should be applied before the second law when factoring an expression.

A sum-of-products expression can always be realized directly by one or more AND gates feeding a single OR gate at the circuit output. Figure 2-5 shows the circuits for Equations (2-15) and (2-17). Inverters required to generate the complemented variables have been omitted.

A product-of-sums expression can always be realized directly by one or more OR gates feeding a single AND gate at the circuit output. Figure 2-6 shows the circuits for Equations (2-18) and (2-20). Inverters required to generate the complements have been omitted.

The circuits shown in Figures 2-5 and 2-6 are often referred to as two-level circuits because they have a maximum of two gates in series between an input and the circuit output.

FIGURE 2-5
Circuits for
Equations (2-15)
and (2-17)

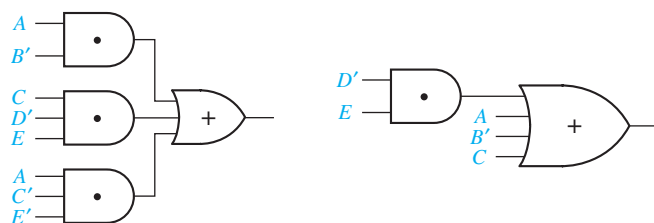
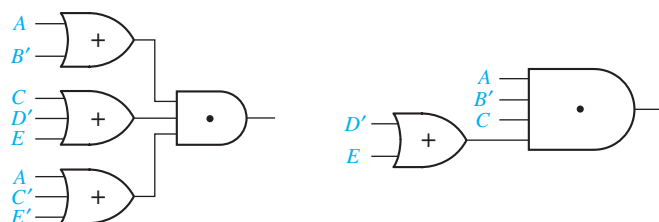


FIGURE 2-6
Circuits for
Equations (2-18)
and (2-20)



2.8 DeMorgan's Laws

The inverse or complement of any Boolean expression can easily be found by successively applying the following theorems, which are frequently referred to as DeMorgan's laws:

$$(X + Y)' = X' Y' \quad (2-21)$$

$$(XY)' = X' + Y' \quad (2-22)$$

We will verify these laws using a truth table:

XY	$X' Y'$	$X + Y$	$(X + Y)'$	$X' Y'$	XY	$(XY)'$	$X' + Y'$
0 0	1 1	0	1	1	0	1	1
0 1	1 0	1	0	0	0	1	1
1 0	0 1	1	0	0	0	1	1
1 1	0 0	1	0	0	1	0	0

DeMorgan's laws are easily generalized to n variables:

$$(X_1 + X_2 + X_3 + \dots + X_n)' = X_1' X_2' X_3' \dots X_n' \quad (2-23)$$

$$(X_1 X_2 X_3 \dots X_n)' = X_1' + X_2' + X_3' + \dots + X_n' \quad (2-24)$$

For example, for $n = 3$,

$$(X_1 + X_2 + X_3)' = (X_1 + X_2)' X_3' = X_1' X_2' X_3'$$

Referring to the OR operation as the logical sum and the AND operation as logical product, DeMorgan's laws can be stated as

The complement of the product is the sum of the complements.

The complement of the sum is the product of the complements.

To form the complement of an expression containing both OR and AND operations, DeMorgan's laws are applied alternately.

Example 1

To find the complement of $(A' + B)C'$, first apply (2-22) and then (2-21).

$$[(A' + B)C']' = (A' + B)' + (C')' = AB' + C$$

Example 2

$$\begin{aligned}
 [(AB' + C)D' + E]' &= [(AB' + C)D']'E' && \text{(by (2-21))} \\
 &= [(AB' + C)' + D]E' && \text{(by (2-22))} \\
 &= [(AB')'C' + D]E' && \text{(by (2-21))} \\
 &= [(A' + B)C' + D]E' && \text{(by (2-22))} \quad (2-25)
 \end{aligned}$$

Note that in the final expressions, the complement operation is applied only to single variables.

The inverse of $F = A'B + AB'$ is

$$\begin{aligned} F' &= (A'B + AB')' = (A'B)'(AB')' = (A + B')(A' + B) \\ &= AA' + AB + B'A' + BB' = A'B' + AB \end{aligned}$$

We will verify that this result is correct by constructing a truth table for F and F' :

$A B$	$A'B$	AB'	$F = A'B + AB'$	$A'B'$	AB	$F' = A'B' + AB$
0 0	0	0	0	1	0	1
0 1	1	0	1	0	0	0
1 0	0	1	1	0	0	0
1 1	0	0	0	0	1	1

In the table, note that for every combination of values of A and B for which $F = 0$, $F' = 1$; and whenever $F = 1$, $F' = 0$.

Given a Boolean expression, the *dual* is formed by replacing AND with OR, OR with AND, 0 with 1, and 1 with 0. Variables and complements are left unchanged. The dual of AND is OR and the dual of OR is AND:

$$(XYZ \dots)^D = X + Y + Z + \dots \quad (X + Y + Z + \dots)^D = XYZ \dots \quad (2-26)$$

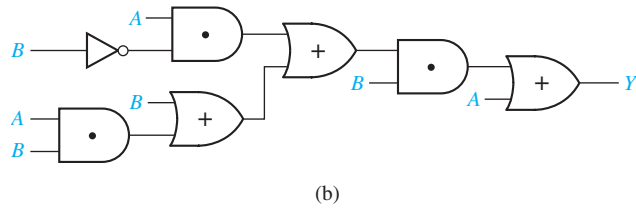
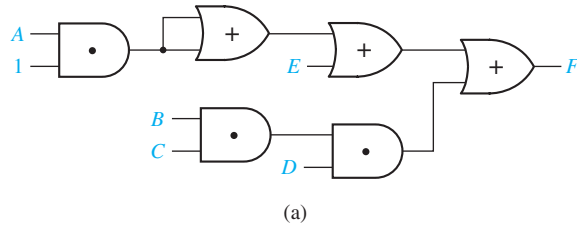
The dual of an expression may be found by complementing the entire expression and then complementing each individual variable. For example, to find the dual of $AB' + C$,

$$(AB' + C)' = (AB')'C' = (A' + B)C', \quad \text{so} \quad (AB' + C)^D = (A + B')C$$

The laws and theorems of Boolean algebra on page 55 are listed in dual pairs. For example, Theorem 11 is $(X + Y')Y = XY$ and its dual is $XY' + Y = X + Y$ (Theorem 11D).

Problems

- 2.1 Prove the following theorems algebraically:
- (a) $X(X' + Y) = XY$ (b) $X + XY = X$
(c) $XY + XY' = X$ (d) $(A + B)(A + B') = A$
- 2.2 Illustrate the following theorems using circuits of switches:
- (a) $X + XY = X$ (b) $X + YZ = (X + Y)(X + Z)$
In each case, explain why the circuits are equivalent.
- 2.3 Simplify each of the following expressions by applying *one* of the theorems. State the theorem used (see page 55).
- (a) $X'Y'Z + (X'Y'Z)'$ (b) $(AB' + CD)(B'E + CD)$
(c) $ACF + AC'F$ (d) $A(C + D'B) + A'$
(e) $(A'B + C + D)(A'B + D)$ (f) $(A + BC) + (DE + F)(A + BC)'$
- 2.4 For each of the following circuits, find the output and design a simpler circuit having the same output. (*Hint*: Find the circuit output by first finding the output of each gate, going from left to right, and simplifying as you go.)



2.5 Multiply out and simplify to obtain a sum of products:

(a) $(A + B)(C + B)(D' + B)(ACD' + E)$

(b) $(A' + B + C')(A' + C' + D)(B' + D')$

2.6 Factor each of the following expressions to obtain a product of sums:

(a) $AB + C'D'$

(b) $WX + WY'X + ZYX$

(c) $A'BC + EF + DEF'$

(d) $XYZ + W'Z + XQ'Z$

(e) $ACD' + C'D' + A'C$

(f) $A + BC + DE$

(The answer to (f) should be the product of four terms, each a sum of three variables.)

2.7 Draw a circuit that uses only one AND gate and one OR gate to realize each of the following functions:

(a) $(A + B + C + D)(A + B + C + E)(A + B + C + F)$

(b) $WXYZ + VXYZ + UXYZ$

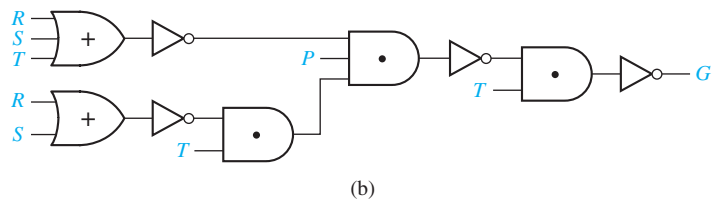
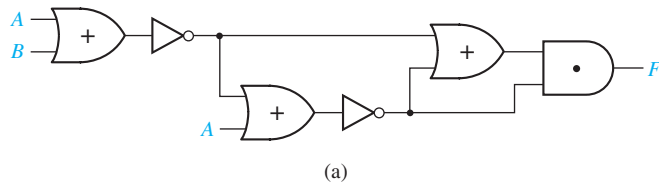
2.8 Simplify the following expressions to a minimum sum of products.

(a) $[(AB)' + C'D']'$

(b) $[A + B(C' + D)]'$

(c) $((A + B')C')(A + B)(C + A)'$

2.9 Find F and G and simplify:



2.10 Illustrate the following equations using circuits of switches:

- (a) $XY + XY' = X$ (b) $(X + Y')Y = XY$
 (c) $X + X'ZY = X + YZ$ (d) $(A + B)C + (A + B)C' = A + B$
 (e) $(X + Y)(X + Z) = X + YZ$ (f) $X(X + Y) = X$

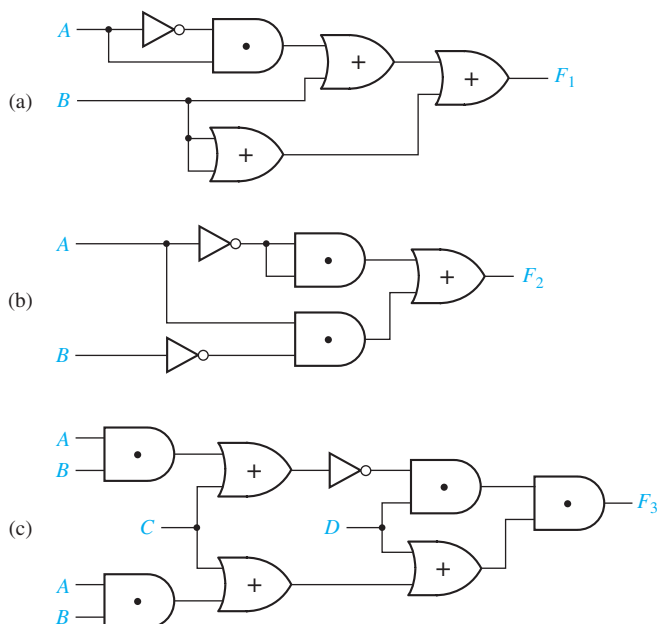
2.11 Simplify each of the following expressions by applying *one* of the theorems. State the theorem used.

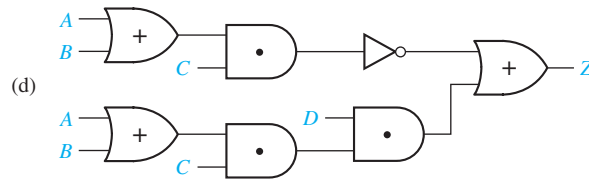
- (a) $(A' + B' + C)(A' + B' + C)'$ (b) $AB(C' + D) + B(C' + D)$
 (c) $AB + (C' + D)(AB)'$ (d) $(A'BF + CD')(A'BF + CEG)$
 (e) $[AB' + (C + D)' + E'F](C + D)$ (f) $A'(B + C)(D'E + F)' + (D'E + F)$

2.12 Simplify each of the following expressions by applying *one* of the theorems. State the theorem used.

- (a) $(X + Y'Z) + (X + Y'Z)'$
 (b) $[W + X'(Y + Z)][W' + X'(Y + Z)]$
 (c) $(V'W + UX)'(UX + Y + Z + V'W)$
 (d) $(UV' + W'X)(UV' + W'X + Y'Z)$
 (e) $(W' + X)(Y + Z') + (W' + X)'(Y + Z')$
 (f) $(V' + U + W)[(W + X) + Y + UZ'] + [(W + X) + UZ' + Y]$

2.13 For each of the following circuits, find the output and design a simpler circuit that has the same output. (*Hint*: Find the circuit output by first finding the output of each gate, going from left to right, and simplifying as you go).





2.14 Draw a circuit that uses only one AND gate and one OR gate to realize each of the following functions:

(a) $ABCF + ACEF + ACDF$

(b) $(V + W + Y + Z)(U + W + Y + Z)(W + X + Y + Z)$

2.15 Use *only* DeMorgan's relationships and Involution to find the complements of the following functions:

(a) $f(A, B, C, D) = [A + (BCD)'][(AD)' + B(C' + A)]$

(b) $f(A, B, C, D) = AB'C + (A' + B + D)(ABD' + B')$

2.16 Using *just* the definition of the dual of a Boolean algebra expression, find the duals of the following expressions:

(a) $f(A, B, C, D) = [A + (BCD)'][(AD)' + B(C' + A)]$

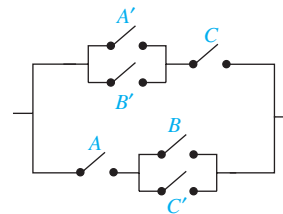
(b) $f(A, B, C, D) = AB'C + (A' + B + D)(ABD' + B')$

2.17 For the following switching circuit, find the logic function expression describing the circuit by the three methods indicated, simplify each expression, and show they are equal.

(a) subdividing it into series and parallel connections of subcircuits until single switches are obtained

(b) finding all paths through the circuit (sometimes called *tie sets*), forming an AND term for each path and ORing the AND terms together

(c) finding all ways of breaking all paths through the circuit (sometimes called *cut sets*), forming an OR term for each cut set and ANDing the OR terms together.



2.18 For each of the following Boolean (or switching) algebra expressions, indicate which, if any, of the following terms describe the expression: product term, sum-of-products, sum term, and product-of-sums. (More than one may apply.)

(a) $X'Y$

(b) $XY' + YZ$

(c) $(X' + Y)(WX + Z)$

(d) $X + Z$

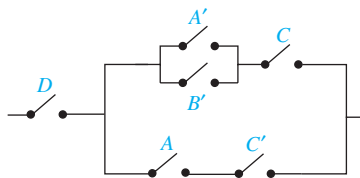
(e) $(X' + Y)(W + Z)(X + Y' + Z')$

- 2.19** Construct a gate circuit using AND, OR, and NOT gates that corresponds one to one with the following switching algebra expression. Assume that inputs are available only in uncomplemented form. (Do not change the expression.)

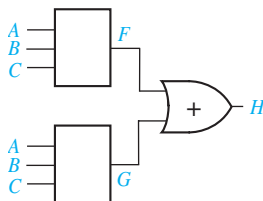
$$(WX' + Y)[(W + Z)' + XYZ']$$

- 2.20** For the following switch circuit:

- derive the switching algebra expression that corresponds one to one with the switch circuit.
- derive an equivalent switch circuit with a structure consisting of a parallel connection of groups of switches connected in series. (Use 9 switches.)
- derive an equivalent switch circuit with a structure consisting of a series connection of groups of switches connected in parallel. (Use 6 switches.)



- 2.21** In the following circuit, $F = (A' + B)C$. Give a truth table for G so that H is as specified in its truth table. If G can be either 0 or 1 for some input combination, leave its value unspecified.



A	B	C	H
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- 2.22** Factor each of the following expressions to obtain a product of sums:

- $A'B' + A'CD + A'DE'$
- $H'I' + JK$
- $A'BC + A'B'C + CD'$
- $A'B' + (CD' + E)$
- $A'B'C + B'CD' + EF'$
- $WX'Y + W'X' + W'Y'$

- 2.23** Factor each of the following expressions to obtain a product of sums:

- $W + U'YV$
- $TW + UY' + V$
- $A'B'C + B'CD' + B'E'$
- $ABC + ADE' + ABF'$

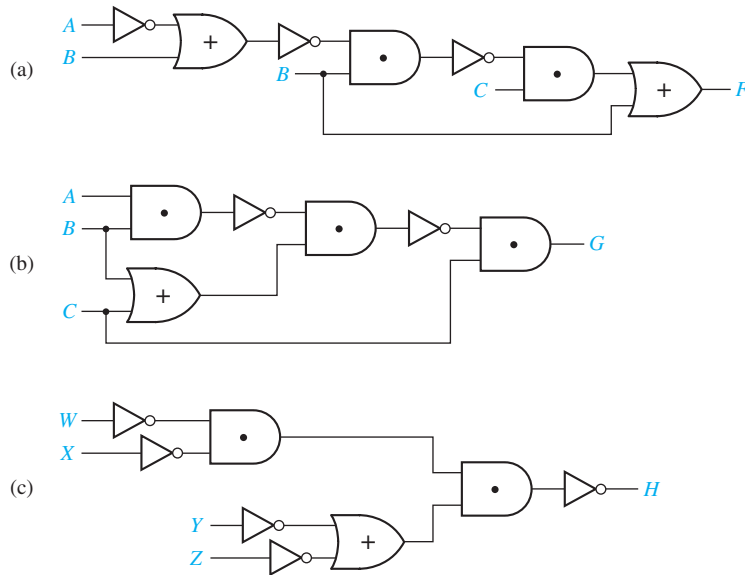
- 2.24** Simplify the following expressions to a minimum sum of products. Only individual variables should be complemented.

- $[(XY)' + (X' + Y)Z]$
- $(X + (Y'(Z + W)'))'$
- $[(A' + B')' + (A'B'C)' + C'D]'$
- $(A + B)CD + (A + B)'$

2.25 For each of the following functions find a sum-of-products expression for F' .

- (a) $F(P, Q, R, S) = (R' + PQ)S$
- (b) $F(W, X, Y, Z) = X + YZ(W + X')$
- (c) $F(A, B, C, D) = A' + B' + ACD$

2.26 Find F , G , and H , and simplify:



2.27 Draw a circuit that uses two OR gates and two AND gates to realize the following function:

$$F = (V + W + X)(V + X + Y)(V + Z)$$

2.28 Draw a circuit to realize the function:

$$F = ABC + A'BC + AB'C + ABC'$$

- (a) using one OR gate and three AND gates. The AND gates should have two inputs.
- (b) using two OR gates and two AND gates. All of the gates should have two inputs.

2.29 Prove the following equations using truth tables:

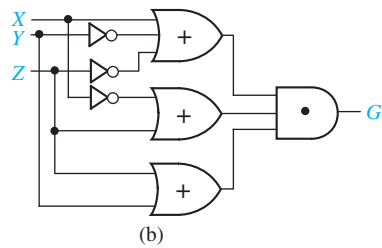
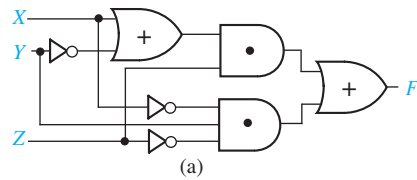
- (a) $(X + Y)(X' + Z) = XZ + X'Y$
- (b) $(X + Y)(Y + Z)(X' + Z) = (X + Y)(X' + Z)$
- (c) $XY + YZ + X'Z = XY + X'Z$

(d) $(A + C)(AB + C') = AB + AC'$

(e) $W'XY + WZ = (W' + Z)(W + XY)$

(Note: Parts (a), (b), and (c) are theorems that will be introduced in Unit 3.)

2.30 Show that the following two gate circuits realize the same function.



Laws and Theorems of Boolean Algebra

Operations with 0 and 1:

$$1. X + 0 = X$$

$$1D. X \cdot 1 = X$$

$$2. X + 1 = 1$$

$$2D. X \cdot 0 = 0$$

Idempotent laws:

$$3. X + X = X$$

$$3D. X \cdot X = X$$

Involution law:

$$4. (X')' = X$$

Laws of complementarity:

$$5. X + X' = 1$$

$$5D. X \cdot X' = 0$$

Commutative laws:

$$6. X + Y = Y + X$$

$$6D. XY = YX$$

Associative laws:

$$7. (X + Y) + Z = X + (Y + Z) \\ = X + Y + Z$$

$$7D. (XY)Z = X(YZ) = XYZ$$

Distributive laws:

$$8. X(Y + Z) = XY + XZ$$

$$8D. X + YZ = (X + Y)(X + Z)$$

Simplification theorems:

$$9. XY + XY' = X$$

$$9D. (X + Y)(X + Y') = X$$

$$10. X + XY = X$$

$$10D. X(X + Y) = X$$

$$11. (X + Y')Y = XY$$

$$11D. XY' + Y = X + Y$$

DeMorgan's laws:

$$12. (X + Y + Z + \dots)' = X'Y'Z' \dots$$

$$12D. (XYZ \dots)' = X' + Y' + Z' + \dots$$

Duality:

$$13. (X + Y + Z + \dots)^D = XYZ \dots$$

$$13D. (XYZ \dots)^D = X + Y + Z + \dots$$

Theorem for multiplying out and factoring:

$$14. (X + Y)(X' + Z) = XZ + X'Y$$

$$14D. XY + X'Z = (X + Z)(X' + Y)$$

Consensus theorem:

$$15. XY + YZ + X'Z = XY + X'Z$$

$$15D. (X + Y)(Y + Z)(X' + Z) \\ = (X + Y)(X' + Z)$$

Objectives

When you complete this unit, you should know from memory and be able to use any of the laws and theorems of Boolean algebra listed at the end of Unit 2. Specifically, you should be able to

1. Apply these laws and theorems to the manipulation of algebraic expressions including:
 - a. Simplifying an expression.
 - b. Finding the complement of an expression.
 - c. Multiplying out and factoring an expression.
2. Prove any of the theorems using a truth table or give an algebraic proof if appropriate.
3. Define the exclusive-OR and equivalence operations. State, prove, and use the basic theorems that concern these operations.
4. Use the consensus theorem to delete terms from and add terms to a switching expression.
5. Given an equation, prove algebraically that it is valid or show that it is not valid.

Study Guide

1. Study Section 3.1, *Multiplying Out and Factoring Expressions*.

- (a) List three laws or theorems which are useful when multiplying out or factoring expressions.

- (b) Use Equation (3-3) to factor each of the following:

$$ab'c + bd =$$

$$abc + (ab)'d =$$

- (c) In the following example, first group the terms so that (3-2) can be applied two times.

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

After applying (3-2), apply (3-3) and then finish multiplying out by using (3-1).

If we did not use (3-2) and (3-3) and used only (3-1) on the original F_1 expression, we would generate many more terms:

$$\begin{aligned} F_1 &= (w'x + w'y' + w'z + \cancel{xx'} + x'y' + x'z + xy + \cancel{yy'} + yz) \\ &\quad (\cancel{ww'} + w'x + w'y' + wy + xy + \cancel{yy'} + wz' + xz' + y'z') \\ &= \underbrace{(w'x + w'xy' + w'xz + \cdots + yzy'z')}_{49 \text{ terms in all}} \end{aligned}$$

This is obviously a *very inefficient* way to proceed! The moral to this story is to first group the terms and apply (3-2) and (3-3) where possible.

- (d) Work Programmed Exercise 3.1. Then work Problem 3.6, being careful not to introduce any unnecessary terms in the process.
- (e) In Unit 2 you learned how to factor a Boolean expression, using the two distributive laws. In addition, this unit introduced use of the theorem

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

in the factoring process. Careful choice of the order in which these laws and theorems are applied may cut down the amount of work required to

factor an expression. When factoring, it is best to apply Equation (3-1) first, using as X the variable or variables which appear most frequently. Then Equations (3-2) and (3-3) can be applied in either order, depending on circumstances.

(f) Work Programmed Exercise 3.2. Then work Problem 3.7.

2. Checking your answers:

A good way to partially check your answers for correctness is to substitute 0's or 1's for some of the variables. For example, if we substitute $A = 1$ in the first and last expression in Equation (3-5), we get

$$\begin{aligned} 1 \cdot C + 0 \cdot BD' + 0 \cdot BE + 0 \cdot C'DE &= (1 + B + C')(1 + B + D) \\ &\quad \cdot (1 + B + E)(1 + D' + E)(0 + C) \\ C &= 1 \cdot 1 \cdot 1 \cdot 1 \cdot C \quad \checkmark \end{aligned}$$

Similarly, substituting $A = 0, B = 0$ we get

$$\begin{aligned} 0 + 0 + 0 + C'DE &= (0 + C')(0 + D)(0 + E)(D' + E)(1 + C) \\ &= C'DE \quad \checkmark \end{aligned}$$

Verify that the result is also correct when $A = 0$ and $B = 1$.

3. The *method* which you use to get your answer is very important in this unit. If it takes you two pages of algebra and one hour of time to work a problem that can be solved in 10 minutes with three lines of work, you have not learned the material in this unit! Even if you get the correct answer, your work is not satisfactory if you worked the problem by an excessively long and time-consuming method. It is important that you learn to solve simple problems in a simple manner—otherwise, when you are asked to solve a complex problem, you will get bogged down and never get the answer. When you are given a problem to solve, do not just plunge in, but first ask yourself, “What is the easiest way to work this problem?” For example, when you are asked to multiply out an expression, do not just multiply it out by brute force, term by term. Instead, ask yourself, “How can I group the terms and which theorems should I apply first in order to reduce the amount of work?” (See Study Guide Part 1.) After you have worked out Problems 3.6 and 3.7, compare your solutions with those in the solution book. If your solution required substantially more work than the one in the solution book, rework the problem and try to get the answer in a more straightforward manner.

4. Study Section 3.2, *Exclusive-OR and Equivalence Operations*.

- (a) Prove Theorems (3-8) through (3-13). You should be able to prove these both algebraically and by using a truth table.

- (b) Show that $(xy' + x'y)' = xy + x'y'$. Memorize this result.

- (c) Prove Theorem (3-15).

- (d) Show that $(x \equiv 0) = x'$, $(x \equiv x) = 1$, and $(x \equiv y)' = (x \equiv y')$.

- (e) Express $(x \equiv y)'$ in terms of exclusive OR.

- (f) Work Problems 3.8 and 3.9.

5. Study Section 3.3, *The Consensus Theorem*. The consensus theorem is an important method for simplifying switching functions.

- (a) In each of the following expressions, find the consensus term and eliminate it:

$$\begin{aligned} &abc'd + a'be + bc'de \\ &(a' + b + c)(a + d)(b + c + d) \\ &ab'c + a'bd + bcd' + a'bc \end{aligned}$$

- (b) Eliminate two terms from the following expression by applying the consensus theorem:

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

(Hint: First, compare the first term with each of the remaining terms to see if a consensus exists, then compare the second term with each of the remaining terms, etc.)

- (c) Study the example given in Equations (3-22) and (3-23) carefully. Now let us start with the four-term form of the expression (Equation 3-22):

$$A'C'D + A'BD + ABC + ACD'$$

Can this be reduced directly to three terms by the application of the consensus theorem? Before we can reduce this expression, we must add another term. Which term can be added by applying the consensus theorem?

Add this term, and then reduce the expression to three terms. After this reduction, can the term which was added be removed? Why not?

- (d) Eliminate two terms from the following expression by applying the dual consensus theorem:

$$(a' + c' + d)(a' + b + c)(a + b + d)(a' + b + d)(b + c' + d)$$

Use brackets to indicate how you formed the consensus terms. (*Hint: First, find the consensus of the first two terms and eliminate it.*)

- (e) Derive Theorem (3-3) by using the consensus theorem.
- (f) Work Programmed Exercise 3.3. Then work Problem 3.10.
6. Study Section 3.4, *Algebraic Simplification of Switching Expressions*.
- (a) What theorems are used for:

Combining terms?

Eliminating terms?

Eliminating literals?

Adding redundant terms?

Factoring or multiplying out?

- (b) Note that in the example of Equation (3-27), the redundant term WZ' was added and then was eliminated later after it had been used to eliminate another term. Why was it possible to eliminate WZ' in this example?

If a term has been added by the consensus theorem, it may not always be possible to eliminate the term later by the consensus theorem. Why?

- (c) You will need considerable practice to develop skill in simplifying switching expressions. Work through Programmed Exercises 3.4 and 3.5.
- (d) Work Problem 3.11.
- (e) When simplifying an expression using Boolean algebra, two frequently asked questions are
 - (1) Where do I begin?
 - (2) How do I know when I am finished?

In answer to (1), it is generally best to try simple techniques such as combining terms or eliminating terms and literals before trying more complicated things such as using the consensus theorem or adding redundant terms. Question (2) is generally difficult to answer because it may be impossible to simplify some expressions without first adding redundant terms. We will usually tell you how many terms to expect in the minimum solution so that you will not have to waste time trying to simplify an expression which is already minimized. In Units 5 and 6, you will learn systematic techniques which will guarantee finding the minimum solution.

7. Study Section 3.5, *Proving Validity of an Equation*.

- (a) When attempting to prove that an equation is valid, is it permissible to add the same expression to both sides? Explain.
- (b) Work Problem 3.12.
- (c) Show that (3-33) and (3-34) are true by considering both $x = 0$ and $x = 1$.

- (d) Given that $a'(b + d') = a'(b + e')$, the following “proof” shows that $d = e$:

$$a'(b + d') = a'(b + e')$$

$$a + b'd = a + b'e$$

$$b'd = b'e$$

$$d = e$$

State two things that are wrong with the “proof.” Give a set of values for a , b , d , and e that demonstrates that the result is incorrect.

- 8.** Reread the objectives of this unit. When you take the readiness test, you will be expected to know from memory the laws and theorems listed at the end of Unit 2. Where appropriate, you should know them “forward and backward”; that is, given either side of the equation, you should be able to supply the other. Test yourself to see if you can do this. When you are satisfied that you can meet the objectives, take the readiness test.

Boolean Algebra (Continued)

In this unit we continue our study of Boolean algebra to learn additional methods for manipulating Boolean expressions. We introduce another theorem for multiplying out and factoring that facilitates conversion between sum-of-products and product-of-sums expressions. These algebraic manipulations allow us to realize a switching function in a variety of forms. The exclusive-OR and equivalence operations are introduced along with examples of their use. The consensus theorem provides a useful method for simplifying an expression. Then methods for algebraic simplification are reviewed and summarized. The unit concludes with methods for proving the validity of an equation.

3.1 Multiplying Out and Factoring Expressions

Given an expression in product-of-sums form, the corresponding sum-of-products expression can be obtained by multiplying out, using the two distributive laws:

$$X(Y + Z) = XY + XZ \quad (3-1)$$

$$(X + Y)(X + Z) = X + YZ \quad (3-2)$$

In addition, the following theorem is very useful for factoring and multiplying out:

$$(X + Y)(\overline{X'} + Z) = XZ + X'Y \quad (3-3)$$

Note that the variable that is paired with X on one side of the equation is paired with X' on the other side, and vice versa.

Proof:

If $X = 0$, (3-3) reduces to $Y(1 + Z) = 0 + 1 \cdot Y$ or $Y = Y$.

If $X = 1$, (3-3) reduces to $(1 + Y)Z = Z + 0 \cdot Y$ or $Z = Z$.

Because the equation is valid for both $X = 0$ and $X = 1$, it is always valid.

The following example illustrates the use of Theorem (3-3) for factoring:

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

Note that the theorem can be applied when we have two terms, one which contains a variable and another which contains its complement.

Theorem (3-3) is very useful for multiplying out expressions. In the following example, we can apply (3-3) because one factor contains the variable Q , and the other factor contains Q' .

$$(Q + \overline{AB'})(C'D + Q') = QC'D + Q'AB'$$

If we simply multiplied out by using the distributive law, we would get four terms instead of two:

$$(Q + AB')(C'D + Q') = QC'D + QQ' + AB'C'D + AB'Q'$$

Because the term $AB'C'D$ is difficult to eliminate, it is much better to use (3-3) instead of the distributive law.

In general, when we multiply out an expression, we should use (3-3) along with (3-1) and (3-2). To avoid generating unnecessary terms when multiplying out, (3-2) and (3-3) should generally be applied before (3-1), and terms should be grouped to expedite their application.

Example

$$\begin{aligned} & (A + \overline{B + C'})(\overline{A + B} + D)(A + B + E)(\overline{A + D' + E})(A' + C) \\ &= (\overline{A + B + C'D})(\overline{A + B} + E)[AC + A'(D' + E)] \\ &= (\overline{A + B + C'DE})(AC + A'D' + A'E) \\ &= AC + \cancel{ABC} + A'BD' + A'BE + A'C'DE \end{aligned} \quad (3-4)$$

What theorem was used to eliminate ABC ? (*Hint: let $X = AC$.*)

In this example, if the ordinary distributive law (3-1) had been used to multiply out the expression by brute force, 162 terms would have been generated, and 158 of these terms would then have to be eliminated.

The same theorems that are useful for multiplying out expressions are useful for factoring. By repeatedly applying (3-1), (3-2), and (3-3), any expression can be converted to a product-of-sums form.

Example of Factoring

$$\begin{aligned} & AC + A'BD' + A'BE + A'C'DE \\ &= \overbrace{AC}^{XZ} + A'(\overbrace{BD' + BE + C'DE}^{Y}) \\ &= (A + \overbrace{BD' + BE + C'DE}^{Y})(A' + C) \\ &= [\overbrace{A + C'DE}^{X} + \overbrace{B(D' + E)}^{YZ}](A' + C) \end{aligned}$$

$$\begin{aligned}
&= (A + B + C'DE)(A + C'DE + D' + E)(A' + C) \\
&= (A + B + C')(A + B + D)(A + B + E)(A + D' + E)(A' + C) \quad (3-5)
\end{aligned}$$

This is the same expression we started with in (3-4).

3.2 Exclusive-OR and Equivalence Operations

The *exclusive-OR* operation (\oplus) is defined as follows:

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

The truth table for $X \oplus Y$ is

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

From this table, we can see that $X \oplus Y = 1$ iff $X = 1$ or $Y = 1$, but *not* both. The ordinary OR operation, which we have previously defined, is sometimes called inclusive OR because $X + Y = 1$ iff $X = 1$ or $Y = 1$, or both.

Exclusive OR can be expressed in terms of AND and OR. Because $X \oplus Y = 1$ iff X is 0 and Y is 1 or X is 1 and Y is 0, we can write

$$X \oplus Y = X'Y + XY' \quad (3-6)$$

The first term in (3-6) is 1 if $X = 0$ and $Y = 1$; the second term is 1 if $X = 1$ and $Y = 0$. Alternatively, we can derive Equation (3-6) by observing that $X \oplus Y = 1$ iff $X = 1$ or $Y = 1$ *and* X and Y are not both 1. Thus,

$$X \oplus Y = (X + Y)(XY)' = (X + Y)(X' + Y') = X'Y + XY' \quad (3-7)$$

In (3-7), note that $(XY)' = 1$ if X and Y are not both 1.

We will use the following symbol for an exclusive-OR gate:



The following theorems apply to exclusive OR:

$$X \oplus 0 = X \quad (3-8)$$

$$X \oplus 1 = X' \quad (3-9)$$

$$X \oplus X = 0 \quad (3-10)$$

$$X \oplus X' = 1 \quad (3-11)$$

$$X \oplus Y = Y \oplus X \text{ (commutative law)} \quad (3-12)$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z \text{ (associative law)} \quad (3-13)$$

$$X(Y \oplus Z) = XY \oplus XZ \text{ (distributive law)} \quad (3-14)$$

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y' \quad (3-15)$$

Any of these theorems can be proved by using a truth table or by replacing $X \oplus Y$ with one of the equivalent expressions from Equation (3-7). Proof of the distributive law follows:

$$\begin{aligned} XY \oplus XZ &= XY(XZ)' + (XY)'XZ = XY(X' + Z') + (X' + Y')XZ \\ &= XYZ' + XY'Z \\ &= X(YZ' + Y'Z) = X(Y \oplus Z) \end{aligned}$$

The *equivalence* operation (\equiv) is defined by

$$\begin{aligned} (0 \equiv 0) &= 1 & (0 \equiv 1) &= 0 \\ (1 \equiv 0) &= 0 & (1 \equiv 1) &= 1 \end{aligned} \quad (3-16)$$

The truth table for $X \equiv Y$ is

X	Y	$X \equiv Y$
0	0	1
0	1	0
1	0	0
1	1	1

From the definition of equivalence, we see that $(X \equiv Y) = 1$ iff $X = Y$. Because $(X \equiv Y) = 1$ iff $X = Y = 1$ or $X = Y = 0$, we can write

$$(X \equiv Y) = XY + X'Y' \quad (3-17)$$

Equivalence is the complement of exclusive-OR:

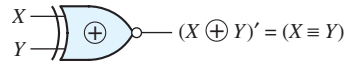
$$\begin{aligned} (X \oplus Y)' &= (X'Y + XY')' = (X + Y')(X' + Y) \\ &= XY + X'Y' = (X \equiv Y) \end{aligned} \quad (3-18)$$

Just as for exclusive-OR, the equivalence operation is commutative and associative.

We will use the following symbol for an equivalence gate:



Because equivalence is the complement of exclusive-OR, an alternate symbol for the equivalence gate is an exclusive-OR gate with a complemented output:



The equivalence gate is also called an exclusive-NOR gate.

In order to simplify an expression which contains AND and OR as well as exclusive OR and equivalence, it is usually desirable to first apply (3-6) and (3-17) to eliminate the \oplus and \equiv operations. As an example, we will simplify

$$F = (A'B \equiv C) + (B \oplus AC')$$

By (3-6) and (3-17),

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

When manipulating an expression that contains several exclusive-OR or equivalence operations, it is useful to note that

$$(XY' + X'Y)' = XY + X'Y' \quad (3-19)$$

For example,

$$\begin{aligned} A' \oplus B \oplus C &= [A'B' + (A')'B] \oplus C \\ &= (A'B' + AB)C' + (A'B' + AB)'C && \text{(by (3-6))} \\ &= (A'B' + AB)C' + (A'B + AB')C && \text{(by (3-19))} \\ &= A'B'C' + ABC' + A'BC + AB'C \end{aligned}$$

3.3 The Consensus Theorem

The consensus theorem is very useful in simplifying Boolean expressions. Given an expression of the form $XY + X'Z + YZ$, the term YZ is redundant and can be eliminated to form the equivalent expression $XY + X'Z$.

The term that was eliminated is referred to as the *consensus term*. Given a pair of terms for which a variable appears in one term and the complement of that variable in another, the consensus term is formed by multiplying the two original terms together, leaving out the selected variable and its complement. For example, the consensus of ab and $a'c$ is bc ; the consensus of abd and $b'de'$ is $(ad)(de') = ade'$. The consensus of terms $ab'd$ and $a'bd'$ is 0.

The consensus theorem can be stated as follows:

$$XY + X'Z + YZ = XY + X'Z \quad (3-20)$$

Proof:

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

The consensus theorem can be used to eliminate redundant terms from Boolean expressions. For example, in the following expression, $b'c$ is the consensus of $a'b'$ and ac , and ab is the consensus of ac and bc' , so both consensus terms can be eliminated:

$$a'b' + ac + bc' + b'c + ab = a'b' + ac + bc'$$

The brackets indicate how the consensus terms are formed.

The dual form of the consensus theorem is

$$(X + Y)(X' + Z)(Y + Z) = (X + Y)(X' + Z) \quad (3-21)$$

Note again that the key to recognizing the consensus term is to first find a pair of terms, one of which contains a variable and the other its complement. In this case, the consensus is formed by adding this pair of terms together leaving out the selected variable and its complement. In the following expression, $(a + b + d')$ is a consensus term and can be eliminated by using the dual consensus theorem:

$$(a + b + c')(a + b + d')(b + c + d') = (a + b + c')(b + c + d')$$

The final result obtained by application of the consensus theorem may depend on the order in which terms are eliminated.

Example

$$A'C'D + A'BD + BCD + ABC + ACD' \quad (3-22)$$

First, we eliminate BCD as shown. (Why can it be eliminated?)

Now that BCD has been eliminated, it is no longer there, and it *cannot* be used to eliminate another term. Checking all pairs of terms shows that no additional terms can be eliminated by the consensus theorem.

Now we start over again:

$$A'C'D + A'BD + BCD + ABC + ACD' \quad (3-23)$$

This time, we do not eliminate BCD ; instead we eliminate two other terms by the consensus theorem. After doing this, observe that BCD can no longer be eliminated. Note that the expression reduces to four terms if BCD is eliminated first, but that it can be reduced to three terms if BCD is not eliminated.

Sometimes it is impossible to directly reduce an expression to a minimum number of terms by simply eliminating terms. It may be necessary to first add a term using the consensus theorem and *then* use the added term to eliminate other terms. For example, consider the expression

$$F = ABCD + B'CDE + A'B' + BCE'$$

If we compare every pair of terms to see if a consensus term can be formed, we find that the only consensus terms are $ACDE$ (from $ABCD$ and $B'CDE$) and $A'CE'$ (from $A'B'$ and BCE'). Because neither of these consensus terms appears in the original expression, we cannot directly eliminate any terms using the consensus theorem. However, if we first add the consensus term $ACDE$ to F , we get

$$F = \overbrace{ABCD + B'CDE} + A'B' + \overbrace{BCE' + ACDE}$$

Then, we can eliminate $ABCD$ and $B'CDE$ using the consensus theorem, and F reduces to

$$F = A'B' + BCE' + ACDE$$

The term $ACDE$ is no longer redundant and cannot be eliminated from the final expression.

3.4 Algebraic Simplification of Switching Expressions

In this section we review and summarize methods for simplifying switching expressions, using the laws and theorems of Boolean algebra. This is important because simplifying an expression reduces the cost of realizing the expression using gates. Later, we will learn graphical methods for simplifying switching functions, but we will learn algebraic methods first. In addition to multiplying out and factoring, three basic ways of simplifying switching functions are combining terms, eliminating terms, and eliminating literals.

1. *Combining terms.* Use the theorem $XY + XY' = X$ to combine two terms. For example,

$$abc'd' + abcd' = abd' \quad [X = abd', Y = c] \quad (3-24)$$

When combining terms by this theorem, the two terms to be combined should contain exactly the same variables, and exactly one of the variables should appear complemented in one term and not in the other. Because $X + X = X$, a given term may be duplicated and combined with two or more other terms. For example,

$$ab'c + abc + a'bc = ab'c + abc + abc + a'bc = ac + bc$$

The theorem still can be used, of course, when X and Y are replaced with more complicated expressions. For example,

$$(a + bc)(d + e') + a'(b' + c')(d + e') = d + e' \\ [X = d + e', Y = a + bc, Y' = a'(b' + c')]$$

2. *Eliminating terms.* Use the theorem $X + XY = X$ to eliminate redundant terms if possible; then try to apply the consensus theorem ($XY + X'Z + YZ = XY + X'Z$) to eliminate any consensus terms. For example,

$$\begin{aligned} a'b + a'bc &= a'b & [X = a'b] \\ a'bc' + bcd + a'bd &= a'bc' + bcd & [X = c, Y = bd, Z = a'b] \end{aligned} \quad (3-25)$$

3. *Eliminating literals.* Use the theorem $X + X'Y = X + Y$ to eliminate redundant literals. Simple factoring may be necessary before the theorem is applied.

Example

$$\begin{aligned} A'B + A'B'C'D' + ABCD' &= A'(B + B'C'D') + ABCD' \\ &= A'(B + C'D') + ABCD' \\ &= B(A' + ACD') + A'C'D' \\ &= B(A' + CD') + A'C'D' \\ &= A'B + BCD' + A'C'D' \end{aligned} \quad (3-26)$$

The expression obtained after applying steps 1, 2, and 3 will not necessarily have a minimum number of terms or a minimum number of literals. If it does not and no further simplification can be made using steps 1, 2, and 3, the deliberate introduction of redundant terms may be necessary before further simplification can be made.

4. *Adding redundant terms.* Redundant terms can be introduced in several ways such as adding xx' , multiplying by $(x + x')$, adding yz to $xy + x'z$, or adding xy to x . When possible, the added terms should be chosen so that they will combine with or eliminate other terms.

Example

$$\begin{aligned} WX + XY + X'Z' + WY'Z' & \quad (\text{add } WZ' \text{ by consensus theorem}) \\ = WX + XY + X'Z' + WY'Z' + WZ' & \quad (\text{eliminate } WY'Z') \\ = WX + XY + X'Z' + WZ' & \quad (\text{eliminate } WZ') \\ = WX + XY + X'Z' & \quad (3-27) \end{aligned}$$

The following comprehensive example illustrates the use of all four methods:

Example

$$\begin{aligned} & \underbrace{A'B'C'D' + A'BC'D'}_{\textcircled{1} A'C'D'} + A'BD + \underbrace{A'BC'D}_{\textcircled{2}} + ABCD + ACD' + B'CD' \\ &= A'C'D' + BD(A' + AC) + ACD' + B'CD' \\ &= A'C'D' + A'BD + \underbrace{BCD + ACD'}_{\textcircled{3}} + B'CD' \\ & \quad \quad \quad + ABC \textcircled{4} \end{aligned}$$

$$\begin{aligned}
 &= A'C'D' + \underbrace{A'BD + \cancel{BCD} + \cancel{ACD'} + B'CD' + ABC}_{\text{consensus } BCD} \\
 &= A'C'D' + A'BD + B'CD' + ABC
 \end{aligned} \tag{3-28}$$

What theorems were used in steps 1, 2, 3, and 4?

If the simplified expression is to be left in a product-of-sums form instead of a sum-of-products form, the duals of the preceding theorems should be applied.

Example

$$\begin{aligned}
 &(\underbrace{A' + B' + C'}_{\textcircled{1} (A' + B')})(A' + B' + C)(B' + C)(A + C)(\underbrace{\overline{A + B + C}}_{\textcircled{2}}) \\
 &= (A' + B')(\underbrace{\overline{B' + C}}_{\textcircled{3}})(A + C) = (A' + B')(A + C)
 \end{aligned} \tag{3-29}$$

What theorems were used in steps 1, 2, and 3?

In general, there is no easy way of determining when a Boolean expression has a minimum number of terms or a minimum number of literals. Systematic methods for finding minimum sum-of-products and minimum product-of-sums expressions will be discussed in Units 5 and 6.

3.5 Proving Validity of an Equation

Often we will need to determine if an equation is valid for all combinations of values of the variables. Several methods can be used to determine if an equation is valid:

1. Construct a truth table and evaluate both sides of the equation for all combinations of values of the variables. (This method is rather tedious if the number of variables is large, and it certainly is not very elegant.)
2. Manipulate one side of the equation by applying various theorems until it is identical with the other side.
3. Reduce both sides of the equation independently to the same expression.
4. It is permissible to perform the same operation on both sides of the equation provided that the operation is reversible. For example, it is all right to complement both sides of the equation, but it is *not* permissible to multiply both sides of the equation by the same expression. (Multiplication is not reversible because division is not defined for Boolean algebra.) Similarly, it is *not* permissible to add the same term to both sides of the equation because subtraction is not defined for Boolean algebra.

To prove that an equation is *not* valid, it is sufficient to show one combination of values of the variables for which the two sides of the equation have different values. When using method 2 or 3 above to prove that an equation is valid, a useful strategy is to

1. First reduce both sides to a sum of products (or a product of sums).
2. Compare the two sides of the equation to see how they differ.
3. Then try to add terms to one side of the equation that are present on the other side.
4. Finally try to eliminate terms from one side that are not present on the other.

Whatever method is used, frequently compare both sides of the equation and let the difference between them serve as a guide for what steps to take next.

Example 1

Show that

$$A'BD' + BCD + ABC' + AB'D = BC'D' + AD + A'BC$$

Starting with the left side, we first add consensus terms, then combine terms, and finally eliminate terms by the consensus theorem.

$$\begin{aligned}
 & A'BD' + BCD + ABC' + AB'D \\
 &= A'BD' + BCD + ABC' + AB'D + BC'D' + A'BC + ABD \\
 &\quad \text{(add consensus of } A'BD' \text{ and } ABC') \quad \nearrow \\
 &\quad \text{(add consensus of } A'BD' \text{ and } BCD) \quad \nearrow \\
 &\quad \text{(add consensus of } BCD \text{ and } ABC') \quad \nearrow \\
 &= AD + A'BD' + BCD + ABC' + BC'D' + A'BC = BC'D' + AD + A'BC \\
 &\quad \quad \quad \nearrow \quad \quad \quad \nearrow \quad \quad \quad \nearrow \\
 &\quad \quad \quad \text{(eliminate consensus of } BC'D' \text{ and } AD) \\
 &\quad \quad \quad \text{(eliminate consensus of } AD \text{ and } A'BC) \\
 &\quad \quad \quad \text{(eliminate consensus of } BC'D' \text{ and } A'BC) \quad (3-30)
 \end{aligned}$$

Example 2

Show that the following equation is valid:

$$\begin{aligned}
 & A'BC'D + (A' + BC)(A + C'D') + BC'D + A'BC' \\
 &= ABCD + A'C'D' + ABD + ABCD' + BC'D
 \end{aligned}$$

First, we will reduce the left side:

$$\begin{aligned}
 & A'BC'D + (A' + BC)(A + C'D') + BC'D + A'BC' \\
 & \quad \quad \quad \text{(eliminate } A'BC'D \text{ using (2-13))} \\
 &= (A' + BC)(A + C'D') + BC'D + A'BC' \\
 & \quad \quad \quad \text{(multiply out using (3-3))} \\
 &= ABC + A'C'D' + BC'D + A'BC' \\
 & \quad \quad \quad \text{(eliminate } A'BC' \text{ by consensus)} \\
 &= ABC + A'C'D' + BC'D
 \end{aligned}$$

Now we will reduce the right side:

$$\begin{aligned}
 &= ABCD + A'C'D' + ABD + ABCD' + BC'D \\
 &\text{(combine } ABCD \text{ and } ABCD') \\
 &= ABC + A'C'D' + ABD + BC'D \\
 &\text{(eliminate } ABD \text{ by consensus)} \\
 &= ABC + A'C'D' + BC'D
 \end{aligned}$$

Because both sides of the original equation were independently reduced to the same expression, the original equation is valid.

As we have previously observed, some of the theorems of Boolean algebra are not true for ordinary algebra. Similarly, some of the theorems of ordinary algebra are *not* true for Boolean algebra. Consider, for example, the cancellation law for ordinary algebra:

$$\text{If } x + y = x + z, \quad \text{then} \quad y = z \quad (3-31)$$

The cancellation law is *not* true for Boolean algebra. We will demonstrate this by constructing a counterexample in which $x + y = x + z$ but $y \neq z$. Let $x = 1$, $y = 0$, $z = 1$. Then,

$$1 + 0 = 1 + 1 \text{ but } 0 \neq 1$$

In ordinary algebra, the cancellation law for multiplication is

$$\text{If } xy = xz, \quad \text{then} \quad y = z \quad (3-32)$$

This law is valid provided $x \neq 0$.

In Boolean algebra, the cancellation law for multiplication is also *not* valid when $x = 0$. (Let $x = 0$, $y = 0$, $z = 1$; then $0 \cdot 0 = 0 \cdot 1$, but $0 \neq 1$). Because $x = 0$ about half of the time in switching algebra, the cancellation law for multiplication cannot be used.

Even though Statements (3-31) and (3-32) are generally false for Boolean algebra, the converses

$$\text{If } y = z, \quad \text{then} \quad x + y = x + z \quad (3-33)$$

$$\text{If } y = z, \quad \text{then} \quad xy = xz \quad (3-34)$$

are true. Thus, we see that although adding the same term to both sides of a Boolean equation leads to a valid equation, the reverse operation of canceling or subtracting a term from both sides generally does not lead to a valid equation. Similarly, multiplying both sides of a Boolean equation by the same term leads to a valid equation, but not conversely. When we are attempting to prove that an equation is valid, it is *not* permissible to add the same expression to both sides of the equation or to multiply both sides by the same expression, because these operations are not reversible.

Programmed Exercise 3.1

Cover the answers to this exercise with a sheet of paper and slide it down as you check your answers. Write your answer in the space provided before looking at the correct answer.

The following expression is to be multiplied out to form a sum of products:

$$(A + B + C')(A' + B' + D)(A' + C + D')(A + C' + D)$$

First, find a pair of sum terms which have two literals in common and apply the second distributive law. Also, apply the same law to the other pair of terms.

Answer

$$(A + C' + BD)[A' + (B' + D)(C + D')]$$

(Note: This answer was obtained by using $(X + Y)(X + Z) = X + YZ$.)

Next, find a pair of sum terms which have a variable in one and its complement in the other. Use the appropriate theorem to multiply these sum terms together without introducing any redundant terms. Apply the same theorem a second time.

Answer

$$(A + C' + BD)(A' + B'D' + CD) = A(B'D' + CD) + A'(C' + BD) \text{ or } A(B' + D)(C + D') + A'(C' + BD) = A(B'D' + CD) + A'(C' + BD)$$

(Note: This answer was obtained using $(X + Y)(X' + Z) = XZ + X'Y$.)

Complete the problem by multiplying out using the ordinary distributive law.

Final Answer

$$AB'D' + ACD + A'C' + A'BD$$

Programmed Exercise 3.2

Cover the answers to this exercise with a sheet of paper and slide it down as you check your answers. Write your answer in the space provided before looking at the correct answer.

The following expression is to be factored to form a product of sums:

$$WXY' + W'X'Z + WY'Z + W'YZ'$$

First, factor as far as you can using the ordinary distributive law.

Answer $WY'(X + Z) + W'(X'Z + YZ')$

Next, factor further by using a theorem which involves a variable and its complement. Apply this theorem twice.

Answer $(W + X'Z + YZ')[W' + Y'(X + Z)]$
 $= [W + (X' + Z')(Y + Z)][W' + Y'(X + Z)]$
 or $WY'(X + Z) + W'(X' + Z')(Y + Z)$
 $= [W + (X' + Z')(Y + Z)][W' + Y'(X + Z)]$
 [Note: This answer was obtained by using $AB + A'C = (A + C)(A' + B)$.]

Now, complete the factoring by using the second distributive law.

Final answer $(W + X' + Z')(W + Y + Z)(W' + Y')(W' + X + Z)$

Programmed Exercise 3.3

Cover the answers to this exercise with a sheet of paper and slide it down as you check your answers. Write your answer in the space provided before looking at the correct answer.

The following expression is to be simplified using the consensus theorem:

$$AC' + AB'D + A'B'C + A'CD' + B'C'D'$$

First, find all of the consensus terms by checking all pairs of terms.

Answer The consensus terms are indicated.

$$AC' + AB'D + \overbrace{A'B'D'}^{\text{consensus}} + \underbrace{B'CD}_{\text{consensus}} + \underbrace{A'B'D'}_{\text{consensus}} + \underbrace{AB'C'}_{\text{consensus}} + B'C'D'$$

Can the original expression be simplified by the direct application of the consensus theorem?

Answer No, because none of the consensus terms appears in the original expression.

Now add the consensus term $B'CD$ to the original expression. Compare the added term with each of the original terms to see if any consensus exists. Eliminate as many of the original terms as you can.

Answer

$$\overbrace{AC' + \cancel{AB'D} + \cancel{A'B'C} + A'CD' + \underbrace{B'C'D' + B'CD}_{(A'B'C)}}^{(AB'DD)}$$

Now that we have eliminated two terms, can $B'CD$ also be eliminated? What is the final reduced expression?

Answer No, because the terms used to form $B'CD$ are gone. Final answer is

$$AC' + A'CD' + B'C'D' + B'CD$$

Programmed Exercise 3.4

Keep the answers to this exercise covered with a sheet of paper and slide it down as you check your answers.

Problem: The following expression is to be simplified

$$ab'cd'e + acd + acf'gh' + abcd'e + acde' + e'h'$$

State a theorem which can be used to combine a pair of terms and apply it to combine two of the terms in the above expression.

Answer Apply $XY + XY' = X$ to the terms $ab'cd'e$ and $abcd'e$, which reduces the expression to

$$acd'e + acd + acf'gh' + acde' + e'h'$$

Now state a theorem (other than the consensus theorem) which can be used to eliminate terms and apply it to eliminate a term in this expression.

Answer Apply $X + XY = X$ to eliminate $acde'$. (What term corresponds to X ?) The result is $acd'e + acd + acf'gh' + e'h'$

Now state a theorem that can be used to eliminate literals and apply it to eliminate a literal from one of the terms in this expression. (*Hint*: It may be necessary to factor out some common variables from a pair of terms before the theorem can be applied.)

Answer Use $X + X'Y = X + Y$ to eliminate a literal from $acd'e$. To do this, first factor ac out of the first two terms: $acd'e + acd = ac(d + d'e)$. After eliminating d' , the resulting expression is

$$ace + acd + acf'gh' + e'h'$$

- Can any term be eliminated from this expression by the direct application of the consensus theorem?
- If not, add a redundant term using the consensus theorem, and use this redundant term to eliminate one of the other terms.
- Finally, reduce your expression to three terms.

Answer

- No
- Add the consensus of ace and $e'h'$:

$$ace + acd + acf'gh' + e'h' + ach'$$
 Now eliminate $acf'gh'$ (by $X + XY = X$)

$$ace + acd + e'h' + ach'$$
- Now eliminate ach' by the consensus theorem. The final answer is

$$ace + acd + e'h'$$

Programmed Exercise 3.5

Keep the answers to this exercise covered with a sheet of paper and slide it down as you check your answers.

$$Z = (A + C' + F' + G)(A + C' + F + G)(A + B + C' + D' + G) \\ (A + C + E + G)(A' + B + G)(B + C' + F + G)$$

This is to be simplified to the form

$$(X + X + X)(X + X + X)(X + X + X)$$

where each X represents a literal.

State a theorem which can be used to combine the first two sum terms of Z and apply it. (*Hint:* The two sum terms differ in only one variable.)

Answer

$$(X + Y)(X + Y') = X \\ Z = (A + C' + G)(A + B + C' + D' + G)(A + C + E + G)(A' + B + G) \\ (B + C' + F + G)$$

Now state a theorem (other than the consensus theorem) which can be used to eliminate a sum term and apply it to this expression.

Answer

$$X(X + Y) = X \\ Z = (A + C' + G)(A + C + E + G)(A' + B + G)(B + C' + F + G)$$

Next, eliminate one literal from the second term, leaving the expression otherwise unchanged. (*Hint:* This cannot be done by the direct application of one theorem; it will be necessary to partially multiply out the first two sum terms before eliminating the literal.)

Answer

$$(A + C' + G)(A + C + E + G) = A + G + C'(C + E) = A + G + C'E \\ \text{Therefore,} \\ Z = (A + C' + G)(A + E + G)(A' + B + G)(B + C' + F + G)$$

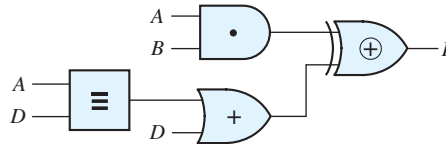
- (a) Can any term be eliminated from this expression by the direct application of the consensus theorem?
- (b) If not, add a redundant sum term using the consensus theorem, and use this redundant term to eliminate one of the other terms.
- (c) Finally, reduce your expression to a product of three sum terms.

Answer

- (a) No
- (b) Add $B + C' + G$ (consensus of $A + C' + G$ and $A' + B + G$).
Use $X(X + Y) = X$, where $X = B + C' + G$, to eliminate $B + C' + F + G$.
- (c) Now eliminate $B + C' + G$ by consensus. The final answer is
$$Z = (A + C' + G)(A + E + G)(A' + B + G)$$

Problems

- 3.6 In each case, multiply out to obtain a sum of products: (Simplify where possible.)
 - (a) $(W + X' + Z')(W' + Y')(W' + X + Z')(W + X')(W + Y + Z)$
 - (b) $(A + B + C + D)(A' + B' + C + D')(A' + C)(A + D)(B + C + D)$
- 3.7 Factor to obtain a product of sums. (Simplify where possible.)
 - (a) $BCD + C'D' + B'C'D + CD$
 - (b) $A'C'D' + ABD' + A'CD + B'D$
- 3.8 Write an expression for F and simplify.



- 3.9 Is the following distributive law valid? $A \oplus BC = (A \oplus B)(A \oplus C)$ Prove your answer.
- 3.10
 - (a) Reduce to a minimum sum of products (three terms):
 $(X + W)(Y \oplus Z) + XW'$
 - (b) Reduce to a minimum sum of products (four terms):
 $(A \oplus BC) + BD + ACD$
 - (c) Reduce to a minimum product of sums (three terms):
 $(A' + C' + D')(A' + B + C')(A + B + D)(A + C + D)$

3.11 Simplify algebraically to a minimum sum of products (five terms):

$$(A + B' + C + E') (A + B' + D' + E) (B' + C' + D' + E')$$

3.12 Prove algebraically that the following equation is valid:

$$A'CD'E + A'B'D' + ABCE + ABD = A'B'D' + ABD + BCD'E$$

3.13 Simplify each of the following expressions:

(a) $KL MN' + K'L' MN + MN'$

(b) $KL'M' + MN' + LM'N'$

(c) $(K + L')(K' + L' + N)(L' + M + N')$

(d) $(K' + L + M' + N)(K' + M' + N + R)(K' + M' + N + R')KM$

3.14 Factor to obtain a product of sums:

(a) $K'L'M + KM'N + KLM + LM'N'$ (four terms)

(b) $KL + K'L' + L'M'N' + LMN'$ (four terms)

(c) $KL + K'L'M + L'M'N + LM'N'$ (four terms)

(d) $K'M'N + KL'N' + K'MN' + LN$ (four terms)

(e) $WXY + WX'Y + WYZ + XYZ'$ (three terms)

3.15 Multiply out to obtain a sum of products:

(a) $(K' + M' + N)(K' + M)(L + M' + N')(K' + L + M)(M + N)$ (three terms)

(b) $(K' + L' + M')(K + M + N')(K + L)(K' + N)(K' + M + N)$

(c) $(K' + L' + M)(K + N')(K' + L + N')(K + L)(K + M + N')$

(d) $(K + L + M)(K' + L' + N')(K' + L' + M')(K + L + N)$

(e) $(K + L + M)(K + M + N)(K' + L' + M')(K' + M' + N')$

3.16 Eliminate the exclusive-OR, and then factor to obtain a minimum product of sums:

(a) $(KL \oplus M) + M'N'$

(b) $M'(K \oplus N') + MN + K'N$

3.17 Algebraically prove identities involving the equivalence (exclusive-NOR) operation:

(a) $x \equiv 0 = x'$

(b) $x \equiv 1 = x$

(c) $x \equiv x = 1$

(d) $x \equiv x' = 0$

(e) $x \equiv y = y \equiv x$

(f) $(x \equiv y) \equiv z = x \equiv (y \equiv z)$

(g) $(x \equiv y)' = x' \equiv y = x \equiv y'$

3.18 Algebraically prove identities involving the exclusive-OR operation:

(a) $x \oplus 0 = x$

(b) $x \oplus 1 = x'$

(c) $x \oplus x = 0$

(d) $x \oplus x' = 1$

(e) $x \oplus y = y \oplus x$

(f) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

(g) $(x \oplus y)' = x' \oplus y = x \oplus y'$

3.19 Algebraically prove the following identities:

- (a) $x + y = x \oplus y \oplus xy$
- (b) $x + y = x \equiv y \equiv xy$

3.20 Algebraically prove or disprove the following distributive identities:

- (a) $x(y \oplus z) = xy \oplus xz$
- (b) $x + (y \oplus z) = (x + y) \oplus (x + z)$
- (c) $x(y \equiv z) = xy \equiv xz$
- (d) $x + (y \equiv z) = (x + y) \equiv (x + z)$

3.21 Simplify each of the following expressions using only the consensus theorem (or its dual):

- (a) $BC'D' + ABC' + AC'D + AB'D + A'BD'$ (reduce to three terms)
- (b) $W'Y' + WYZ + XY'Z + WX'Y$ (reduce to three terms)
- (c) $(B + C + D)(A + B + C)(A' + C + D)(B' + C' + D')$
- (d) $W'XY + WXZ + WY'Z + W'Z'$
- (e) $A'BC' + BC'D' + A'CD + B'CD + A'BD$
- (f) $(A + B + C)(B + C' + D)(A + B + D)(A' + B' + D')$

3.22 Factor $Z = ABC + DE + ACF + AD' + AB'E'$ and simplify it to the form $(X + X)(X + X)(X + X + X + X)$ (where each X represents a literal). Now express Z as a minimum sum of products in the form:

$$XX + XX + XX + XX$$

3.23 Repeat Problem 3.22 for $F = A'B + AC + BC'D' + BEF + BDF$

3.24 Factor to obtain a product of four terms and then reduce to three terms by applying the consensus theorem: $X'Y'Z' + XYZ$

3.25 Simplify each of the following expressions:

- (a) $xy + x'yz' + yz$
- (b) $(xy' + z)(x + y')z$
- (c) $xy' + z + (x' + y)z'$
- (d) $a'd(b' + c) + a'd'(b + c') + (b' + c)(b + c')$
- (e) $w'x' + x'y' + yz + w'z'$
- (f) $A'BCD + A'BC'D + B'EF + CDE'G + A'DEF + A'B'EF$ (reduce to a sum of three terms)
- (g) $[(a' + d' + b'c)(b + d + ac')]' + b'c'd' + a'c'd$ (reduce to three terms)

3.26 Simplify to a sum of three terms:

- (a) $A'C'D' + AC' + BCD + A'CD' + A'BC + AB'C'$
- (b) $A'B'C' + ABD + A'C + A'CD' + AC'D + AB'C'$

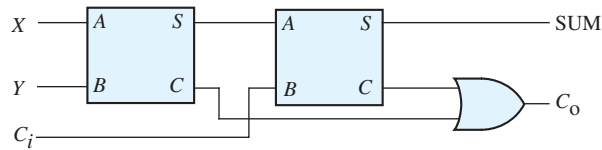
3.27 Reduce to a minimum sum of products:

$$F = WXY' + (W'Y' \equiv X) + (Y \oplus WZ).$$

3.28 Determine which of the following equations are always valid (give an algebraic proof):

- (a) $a'b + b'c + c'a = ab' + bc' + ca'$
- (b) $(a + b)(b + c)(c + a) = (a' + b')(b' + c')(c' + a')$
- (c) $abc + ab'c' + b'cd + bc'd + ad = abc + ab'c' + b'cd + bc'd$
- (d) $xy' + x'z + yz' = x'y + xz' + y'z$
- (e) $(x + y)(y + z)(x + z) = (x' + y')(y' + z')(x' + z')$
- (f) $abc' + ab'c + b'c'd + bcd = ab'c + abc' + ad + bcd + b'c'd$

3.29 The following circuit is implemented using two half-adder circuits. The expressions for the half-adder outputs are $S = A \oplus B$ where \oplus represents the exclusive-OR function, and $C = AB$. Derive simplified sum-of-products expressions for the circuit outputs SUM and C_o . Give the truth table for the outputs.



3.30 The output of a majority circuit is 1 if a majority (more than half) of its inputs are equal to 1, and the output is 0 otherwise. Construct a truth table for a three-input majority circuit and derive a simplified sum-of-products expression for its output.

3.31 Prove algebraically:

- (a) $(X' + Y')(X \equiv Z) + (X + Y)(X \oplus Z) = (X \oplus Y) + Z'$
- (b) $(W' + X + Y')(W + X' + Y)(W + Y' + Z) = X'Y' + WX + XYZ + W'YZ$
- (c) $ABC + A'C'D' + A'BD' + ACD = (A' + C)(A + D')(B + C' + D)$

3.32 Which of the following statements are always true? Justify your answers.

- (a) If $A + B = C$, then $AD' + BD' = CD'$
- (b) If $A'B + A'C = A'D$, then $B + C = D$
- (c) If $A + B = C$, then $A + B + D = C + D$
- (d) If $A + B + C = C + D$, then $A + B = D$

3.33 Find all possible terms that could be added to each expression using the consensus theorem. Then reduce to a minimum sum of products.

- (a) $A'C' + BC + AB' + A'BD + B'C'D' + ACD'$
- (b) $A'C'D' + BC'D + AB'C' + A'BC$

3.34 Simplify the following expression to a sum of two terms and then factor the result to obtain a product of sums: $abd'f' + b'cegh' + abd'f + acd'e + b'ce$

3.35 Multiply out the following expression and simplify to obtain a sum-of-products expression with three terms: $(a + c)(b' + d)(a + c' + d')(b' + c' + d')$

- 3.36** Factor and simplify to obtain a product-of-sums expression with four terms:
 $abc' + d'e + ace + b'c'd'$
- 3.37** (a) Show that $x \oplus y = (x \equiv y)'$
(b) Realize $a'b'c' + a'bc + ab'c + abc'$ using only two-input equivalence gates.

Applications of Boolean Algebra

Minterm and Maxterm Expansions

Objectives

1. Given a word description of the desired behavior of a logic circuit, write the output of the circuit as a function of the input variables. Specify this function as an algebraic expression or by means of a truth table, as is appropriate.
2. Given a truth table, write the function (or its complement) as both a minterm expansion (standard sum of products) and a maxterm expansion (standard product of sums). Be able to use both alphabetic and decimal notation.
3. Given an algebraic expression for a function, expand it algebraically to obtain the minterm or maxterm form.
4. Given one of the following: minterm expansion for F , minterm expansion for F' , maxterm expansion for F , or maxterm expansion for F' , find any of the other three forms.
5. Write the general form of the minterm and maxterm expansion of a function of n variables.
6. Explain why some functions contain don't-care terms.
7. Explain the operation of a full adder and a full subtracter and derive logic equations for these modules. Draw a block diagram for a parallel adder or subtracter and trace signals on the block diagram.

Study Guide

In the previous units, we placed a dot (\bullet) inside the AND-gate symbol, a plus sign ($+$) inside the OR-gate symbol, and a \oplus inside the Exclusive-OR. Because you are now familiar with the relationship between the shape of the gate symbol and the logic function performed, we will omit the \bullet , $+$, and \oplus and use the standard gate symbols for AND, OR, and Exclusive-OR in the rest of the book.

1. Study Section 4.1, *Conversion of English Sentences to Boolean Equations*.

- (a) Use braces to identify the phrases in each of the following sentences:
 - (1) The tape reader should stop if the manual stop button is pressed, if an error occurs, or if an end-of-tape signal is present.
 - (2) He eats eggs for breakfast if it is not Sunday and he has eggs in the refrigerator.
 - (3) Addition should occur iff an add instruction is given and the signs are the same, or if a subtract instruction is given and the signs are not the same.
- (b) Write a Boolean expression which represents each of the sentences in (a). Assign a variable to each phrase, and use a complemented variable to represent a phrase which contains “not”.

(Your answers should be in the form $F = S'E$, $F = AB + SB'$, and $F = A + B + C$, but not necessarily in that order.)

- (c) If X represents the phrase “ N is greater than 3”, how can you represent the phrase “ N is less than or equal to 3”?
 - (d) Work Problems 4.1 and 4.2.
- 2.** Study Section 4.2, *Combinational Logic Design Using a Truth Table*. Previously, you have learned how to go from an algebraic expression for a function to a truth table; in this section you will learn how to go from a truth table to an algebraic expression.

- (a) Write a product term which is 1 iff $a = 0$, $b = 0$, and $c = 1$.
- (b) Write a sum term which is 0 iff $a = 0$, $b = 0$, and $c = 1$.
- (c) Verify that your answers to (a) and (b) are complements.

(d) Write a product term which is 1 iff $a = 1$, $b = 0$, $c = 0$, and $d = 1$.

(e) Write a sum term which is 0 iff $a = 0$, $b = 0$, $c = 1$, and $d = 1$.

(f) For the given truth table, write F as a sum of four product terms which correspond to the four 1's of F

a	b	c	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(g) From the truth table write F as a product of four sum terms which correspond to the four 0's of F

(h) Verify that your answers to both (f) and (g) reduce to $F = b'c' + a'c$.

3. Study Section 4.3, *Minterm and Maxterm Expansions*.

(a) Define the following terms:
minterm (for n variables)

maxterm (for n variables)

(b) Study Table 4-1 and observe the relation between the values of A , B , and C and the corresponding minterms and maxterms.

If $A = 0$, then does A or A' appear in the minterm?

In the maxterm?

If $A = 1$, then does A or A' appear in the minterm?

In the maxterm?

What is the relation between minterm, m_i , and the corresponding maxterm, M_i ?

(c) For the table given in Study Guide Question 2(f), write the minterm expansion for F in m -notation and in decimal notation.

For the same table, write the maxterm expansion for F in M -notation and in decimal notation.

Check your answers by converting your answer to 2(f) to m -notation and your answer to 2(g) to M -notation.

- (d) Given a sum-of-products expression, how do you expand it to a standard sum of products (minterm expansion)?
- (e) Given a product-of-sums expression, how do you expand it to a standard product of sums (maxterm expansion)?
- (f) In Equation (4-11), what theorems were used to factor f to obtain the maxterm expansion?
- (g) Why is the following expression not a maxterm expansion?

$$f(A, B, C, D) = (A + B' + C + D)(A' + B + C')(A' + B + C + D')$$

- (h) Assuming that there are three variables (A, B, C), identify each of the following as a minterm expansion, maxterm expansion, or neither:

- | | |
|--------------------------|---------------------------------|
| (1) $AB + B'C'$ | (2) $(A' + B + C')(A + B' + C)$ |
| (3) $A + B + C$ | (4) $(A' + B)(B' + C)(A' + C)$ |
| (5) $A'BC' + AB'C + ABC$ | (6) $AB'C'$ |

Note that it is possible for a minterm or maxterm expansion to have only one term.

4. (a) Given a minterm in terms of its variables, the procedure for conversion to decimal notation is
 - (1) Replace each complemented variable with a _____ and replace each uncomplemented variable with a _____.
 - (2) Convert the resulting binary number to decimal.
- (b) Convert the minterm $AB'C'DE$ to decimal notation.
- (c) Given that m_{13} is a minterm of the variables A, B, C, D , and E , write the minterm in terms of these variables.
- (d) Given a maxterm in terms of its variables, the procedure for conversion to decimal notation is
 - (1) Replace each complemented variable with a _____ and replace each uncomplemented variable with a _____.
 - (2) Group these 0's and 1's to form a binary number and convert to decimal.
- (e) Convert the maxterm $A' + B + C + D' + E'$ to decimal notation.
- (f) Given that M_{13} is a maxterm of the variables A, B, C, D , and E , write the maxterm in terms of these variables.
- (g) Check your answers to (b), (c), (e), and (f) by using the relation $M_i = m_i'$.
- (h) Given $f(a, b, c, d, e) = \Pi M(0, 10, 28)$, express f in terms of a, b, c, d , and e . (Your answer should contain only five complemented variables.)

5. Study Section 4.4, *General Minterm and Maxterm Expansions*. Make sure that you understand the notation here and can follow the algebra in all of the equations. If you have difficulty with this section, ask for help *before* you take the readiness test.

- (a) How many different functions of four variables are possible?
- (b) Explain why there are 2^{2^n} functions of n variables.
- (c) Write the function of Figure 4-1 in the form of Equation (4-13) and show that it reduces to Equation (4-3).
- (d) For Equation (4-19), write out the indicated summations in full for the case $n = 2$.

- (e) Study Tables 4-3 and 4-4 carefully and make sure you understand why each table entry is valid. Use the truth table for f and f' (Figure 4-1) to verify the entries in Table 4-4. If you understand the relationship between Table 4-3 and the truth table for f and f' , you should be able to perform the conversions without having to memorize the table.

- (f) Given that $f(A, B, C) = \sum m(0, 1, 3, 4, 7)$

The maxterm expansion for f is _____

The minterm expansion for f' is _____

The maxterm expansion for f' is _____

- (g) Work Problem 4.3 and 4.4.

6. Study Section 4.5, *Incompletely Specified Functions*.

- (a) State two reasons why some functions have don't-care terms.

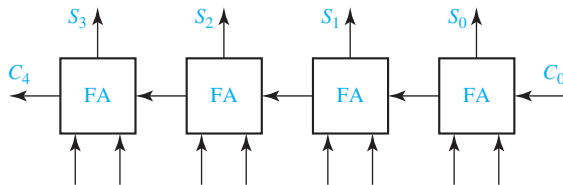
- (b) Given the following table, write the minterm expansion for Z in decimal form.

A	B	C	Z
0	0	0	1
0	0	1	X
0	1	0	0
0	1	1	X
1	0	0	X
1	0	1	1
1	1	0	0
1	1	1	0

- (c) Write the maxterm expansion in decimal form.

- (d) Work Problems 4.5 and 4.6.

7. Study Section 4.6, *Examples of Truth Table Construction*. Finding the truth table from the problem statement is probably the most difficult part of the process of designing a switching circuit. Make sure that you understand how to do this.
8. Work Problems 4.7 through 4.10.
9. Study Section 4.7, *Design of Binary Adders*.
 - (a) For the given parallel adder, show the 0's and 1's at the full adder (FA) inputs and outputs when the following unsigned numbers are added: $11 + 14 = 25$. Verify that the result is correct if $C_4S_3S_2S_1S_0$ is taken as a 5-bit sum. If the sum is limited to 4 bits, explain why this is an overflow condition.



- (b) Review Section 1.4, *Representation of Negative Numbers*. If we use the 2's complement number system to add $(-5) + (-2)$, verify that the FA inputs and outputs are exactly the same as in Part (a). However, for 2's complement, the interpretation of the results is quite different. After discarding C_4 , verify that the resultant 4-bit sum is correct, and therefore no overflow has occurred.
- (c) If we use the 1's complement number system to add $(-5) + (-2)$, show the FA inputs and outputs on the diagram below before the end-around carry is added in. Assume that C_0 is initially 0. Then add the end-around carry (C_4) to the rightmost FA, add the new carry (C_1) into the next cell, and continue until no further changes occur. Verify that the resulting sum is the correct 1's complement representation of -7 .



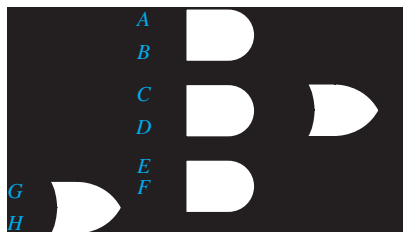
10. (a) Work the following subtraction example. As you subtract each column, place a 1 over the next column if you have to borrow, otherwise place a 0. For each column, as you compute $x_i - y_i - b_i$, fill in the corresponding values of b_{i+1} and d_i in the truth table. If you have done this correctly, the resulting table should match the full subtracter truth table (Table 4-6).

	← borrows	x_i	y_i	b_i	b_{i+1}	d_i
1 1 0 0 0 1 1 0	← X	0	0	0		
-0 1 0 1 1 0 1 0	← Y	0	0	1		
<u> </u>	← difference	0	1	0		
		0	1	1		
		1	0	0		
		1	0	1		
		1	1	0		
		1	1	1		


- (b) Work Problems 4.11 and 4.12.
11. Read the following and then work Problem 4.13 or 4.14 as assigned:
When looking at an expression to determine the required number of gates, keep in mind that the number of required gates is generally *not* equal to the number of AND and OR operations which appear in the expression. For example,

$$AB + CD + EF(G + H)$$

contains four AND operations and three OR operations, but it only requires three AND gates and two OR gates:



12. *Simulation Exercise.* (Must be completed before you take the readiness test.) One purpose of this exercise is to acquaint you with the simulator that you will be using later in more complex design problems. Follow the instructions on the Unit 4 lab assignment sheet.
13. Reread the objectives of this unit. Make sure that you understand the difference in the procedures for converting maxterms and minterms from decimal to algebraic notation. When you are satisfied that you can meet the objectives, take the readiness test. When you come to take the readiness test, turn in a copy of your solution to assigned simulation exercise.



Applications of Boolean Algebra

Minterm and Maxterm Expansions

In this unit you will learn how to design a combinational logic circuit starting with a word description of the desired circuit behavior. The first step is usually to translate the word description into a truth table or into an algebraic expression. Given the truth table for a Boolean function, two standard algebraic forms of the function can be derived—the standard sum of products (minterm expansion) and the standard product of sums (maxterm expansion). Simplification of either of these standard forms leads directly to a realization of the circuit using AND and OR gates.

4.1 Conversion of English Sentences to Boolean Equations

The three main steps in designing a single-output combinational switching circuit are

1. Find a switching function that specifies the desired behavior of the circuit.
2. Find a simplified algebraic expression for the function.
3. Realize the simplified function using available logic elements.

For simple problems, it may be possible to go directly from a word description of the desired behavior of the circuit to an algebraic expression for the output function. In other cases, it is better to first specify the function by means of a truth table and then derive an algebraic expression from the truth table.

Logic design problems are often stated in terms of one or more English sentences. The first step in designing a logic circuit is to translate these sentences into Boolean equations. In order to do this, we must break down each sentence into phrases and associate a Boolean variable with each phrase. If a phrase can have a value of true or false, then we can represent that phrase by a Boolean variable. Phrases such as “she goes to the store” or “today is Monday” can be either true or false, but a command like “go to the store” has no truth value. If a sentence has several phrases, we will mark each phrase with a brace. The following sentence has three phrases:

Mary watches TV if it is Monday night and she has finished her homework.

The “if” and “and” are not included in any phrase; they show the relationships among the phrases.

We will define a two-valued variable to indicate the truth or falsity of each phrase:

$F = 1$ if “Mary watches TV” is true; otherwise, $F = 0$.

$A = 1$ if “it is Monday night” is true; otherwise, $A = 0$.

$B = 1$ if “she has finished her homework” is true; otherwise $B = 0$.

Because F is “true” if A and B are both “true”, we can represent the sentence by $F = A \cdot B$

The following example illustrates how to go from a word statement of a problem directly to an algebraic expression which represents the desired circuit behavior. An alarm circuit is to be designed which operates as follows:

The alarm will ring iff the alarm switch is turned on and the door is not closed, or it is after 6 P.M. and the window is not closed.

The first step in writing an algebraic expression which corresponds to the above sentence is to associate a Boolean variable with each phrase in the sentence. This variable will have a value of 1 when the phrase is true and 0 when it is false. We will use the following assignment of variables:

$\underbrace{\text{The alarm will ring}}_{Z}$	iff	$\underbrace{\text{the alarm switch is on}}_{A}$	and
$\underbrace{\text{the door is not closed}}_{B'}$	or	$\underbrace{\text{it is after 6 P.M.}}_{C}$	and
$\underbrace{\text{the window is not closed.}}_{D'}$			

This assignment implies that if $Z = 1$, the alarm will ring. If the alarm switch is turned on, $A = 1$, and if it is after 6 P.M., $C = 1$. If we use the variable B to represent the phrase “the door is closed”, then B' represents “the door is not closed”. Thus, $B = 1$ if the door is closed, and $B' = 1$ ($B = 0$) if the door is not closed. Similarly, $D = 1$ if the window is closed, and $D' = 1$ if the window is not closed. Using this assignment of variables, the above sentence can be translated into the following Boolean equation:

$$Z = AB' + CD'$$

This equation corresponds to the following circuit:

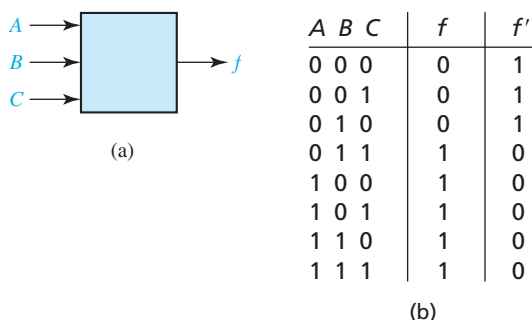


In this circuit, A is a signal which is 1 when the alarm switch is on, C is a signal from a time clock which is 1 when it is after 6 P.M., B is a signal from a switch on the door which is 1 when the door is closed, and similarly D is 1 when the window is closed. The output Z is connected to the alarm so that it will ring when $Z = 1$.

4.2 Combinational Logic Design Using a Truth Table

The next example illustrates logic design using a truth table. A switching circuit has three inputs and one output, as shown in Figure 4-1(a). The inputs A , B , and C represent the first, second, and third bits, respectively, of a binary number N . The output of the circuit is to be $f = 1$ if $N \geq 011_2$ and $f = 0$ if $N < 011_2$. The truth table for f is shown in Figure 4-1(b).

FIGURE 4-1
Combinational
Circuit with Truth
Table



Next, we will derive an algebraic expression for f from the truth table by using the combinations of values of A , B , and C for which $f = 1$. The term $A'BC$ is 1 only if $A = 0$, $B = 1$, and $C = 1$. Similarly, the term $AB'C'$ is 1 only for the combination 100, $AB'C$ is 1 only for 101, ABC' is 1 only for 110, and ABC is 1 only for 111. ORing these terms together yields

$$f = A'BC + AB'C' + AB'C + ABC' + ABC \quad (4-1)$$

This expression equals 1 if A , B , and C take on any of the five combinations of values 011, 100, 101, 110, or 111. If any other combination of values occurs, f is 0 because all five terms are 0.

Equation (4-1) can be simplified by first combining terms and then eliminating A' :

$$f = A'BC + AB' + AB = A'BC + A = A + BC \quad (4-2)$$

Equation (4-2) leads directly to the following circuit:



Instead of writing f in terms of the 1's of the function, we may also write f in terms of the 0's of the function. The function defined by Figure 4-1 is 0 for three combinations of input values. Observe that the term $A + B + C$ is 0 only if $A = B = C = 0$. Similarly, $A + B + C'$ is 0 only for the input combination 001, and $A + B' + C$ is 0 only for the combination 010. ANDing these terms together yields

$$f = (A + B + C)(A + B + C')(A + B' + C) \quad (4-3)$$

This expression equals 0 if A , B , and C take on any of the combinations of values 000, 001, or 010. For any other combination of values, f is 1 because all three terms are 1. Because Equation (4-3) represents the same function as Equation (4-1) they must both reduce to the same expression. Combining terms and using the second distributive law, Equation (4-3) simplifies to

$$f = (A + B)(A + B' + C) = A + B(B' + C) = A + BC \quad (4-4)$$

which is the same as Equation (4-2).

Another way to derive Equation (4-3) is to first write f' as a sum of products, and then complement the result. From Figure 4-1, f' is 1 for input combinations $ABC = 000$, 001, and 010, so

$$f' = A'B'C' + A'B'C + A'BC'$$

Taking the complement of f' yields Equation (4-3).

4.3 Minterm and Maxterm Expansions

Each of the terms in Equation (4-1) is referred to as a minterm. In general, a *minterm* of n variables is a product of n literals in which each variable appears exactly once in either true or complemented form, but not both. (A *literal* is a variable or its complement.) Table 4-1 lists all of the minterms of the three variables A , B , and C . Each minterm has a value of 1 for exactly one combination of values of the variables A , B , and C . Thus if $A = B = C = 0$, $A'B'C' = 1$; if $A = B = 0$ and $C = 1$, $A'B'C = 1$; and so forth. Minterms are often written in abbreviated form— $A'B'C'$ is designated m_0 , $A'B'C$ is designated m_1 , etc. In general, the minterm which corresponds to row i of the truth table is designated m_i (i is usually written in decimal).

TABLE 4-1
Minterms and
Maxterms for
Three Variables

Row No.	A B C	Minterms	Maxterms
0	0 0 0	$A'B'C' = m_0$	$A + B + C = M_0$
1	0 0 1	$A'B'C = m_1$	$A + B + C' = M_1$
2	0 1 0	$A'BC' = m_2$	$A + B' + C = M_2$
3	0 1 1	$A'BC = m_3$	$A + B' + C' = M_3$
4	1 0 0	$AB'C' = m_4$	$A' + B + C = M_4$
5	1 0 1	$AB'C = m_5$	$A' + B + C' = M_5$
6	1 1 0	$ABC' = m_6$	$A' + B' + C = M_6$
7	1 1 1	$ABC = m_7$	$A' + B' + C' = M_7$

When a function f is written as a sum of minterms as in Equation (4-1), this is referred to as a *minterm expansion* or a *standard sum of products*.¹ If $f = 1$ for row i of the truth table, then m_i must be present in the minterm expansion because $m_i = 1$ only for the combination of values of the variables corresponding to row i of the table. Because the minterms present in f are in one-to-one correspondence with the 1's of f in the truth table, the minterm expansion for a function f is unique. Equation (4-1) can be rewritten in terms of m -notation as

$$f(A, B, C) = m_3 + m_4 + m_5 + m_6 + m_7 \quad (4-5)$$

This can be further abbreviated by listing only the decimal subscripts in the form

$$f(A, B, C) = \Sigma m(3, 4, 5, 6, 7) \quad (4-5a)$$

Each of the sum terms (or factors) in Equation (4-3) is referred to as a *maxterm*. In general, a maxterm of n variables is a sum of n literals in which each variable appears exactly once in either true or complemented form, but not both. Table 4-1 lists all of the maxterms of the three variables A , B , and C . Each maxterm has a value of 0 for exactly one combination of values for A , B , and C . Thus, if $A = B = C = 0$, $A + B + C = 0$; if $A = B = 0$ and $C = 1$, $A + B + C' = 0$; and so forth. Maxterms are often written in abbreviated form using M -notation. The maxterm which corresponds to row i of the truth table is designated M_i . Note that each maxterm is the complement of the corresponding minterm, that is, $M_i = m'_i$.

When a function f is written as a product of maxterms, as in Equation (4-3), this is referred to as a *maxterm expansion* or *standard product of sums*. If $f = 0$ for row i of the truth table, then M_i must be present in the maxterm expansion because $M_i = 0$ only for the combination of values of the variables corresponding to row i of the table. Note that the maxterms are multiplied together so that if any one of them is 0, f will be 0. Because the maxterms are in one-to-one correspondence with the 0's of f in the truth table, the maxterm expansion for a function f is unique. Equation (4-3) can be rewritten in M -notation as

$$f(A, B, C) = M_0 M_1 M_2 \quad (4-6)$$

This can be further abbreviated by listing only the decimal subscripts in the form

$$f(A, B, C) = \Pi M(0, 1, 2) \quad (4-6a)$$

where Π means a product.

Because if $f \neq 1$ then $f = 0$, it follows that if m_i is *not* present in the minterm expansion of f , then M_i is present in the maxterm expansion. Thus, given a minterm expansion of an n -variable function f in decimal notation, the maxterm expansion is obtained by listing those decimal integers ($0 \leq i \leq 2^n - 1$) not in the minterm list. Using this method, Equation (4-6a) can be obtained directly from Equation (4-5a).

¹Other names used in the literature for standard sum of products are canonical sum of products and disjunctive normal form. Similarly, a standard product of sums may be called a canonical product of sums or a conjunctive normal form.

Given the minterm or maxterm expansions for f , the minterm or maxterm expansions for the complement of f are easy to obtain. Because f' is 1 when f is 0, the minterm expansion for f' contains those minterms not present in f . Thus, from Equation (4-5),

$$f' = m_0 + m_1 + m_2 = \Sigma m(0, 1, 2) \quad (4-7)$$

Similarly, the maxterm expansion for f' contains those maxterms not present in f . From Equation (4-6),

$$f' = \Pi M(3, 4, 5, 6, 7) = M_3 M_4 M_5 M_6 M_7 \quad (4-8)$$

Because the complement of a minterm is the corresponding maxterm, Equation (4-8) can be obtained by complementing Equation (4-5):

$$f' = (m_3 + m_4 + m_5 + m_6 + m_7)' = m_3' m_4' m_5' m_6' m_7' = M_3 M_4 M_5 M_6 M_7$$

Similarly, Equation (4-7) can be obtained by complementing Equation (4-6):

$$f' = (M_0 M_1 M_2)' = M_0' + M_1' + M_2' = m_0 + m_1 + m_2$$

A general switching expression can be converted to a minterm or maxterm expansion either using a truth table or algebraically. If a truth table is constructed by evaluating the expression for all different combinations of the values of the variables, the minterm and maxterm expansions can be obtained from the truth table by the methods just discussed. Another way to obtain the minterm expansion is to first write the expression as a sum of products and then introduce the missing variables in each term by applying the theorem $X + X' = 1$.

Example

Find the minterm expansion of $f(a, b, c, d) = a'(b' + d) + acd'$.

$$\begin{aligned} f &= a'b' + a'd + acd' \\ &= a'b'(c + c')(d + d') + a'd(b + b')(c + c') + acd'(b + b') \\ &= a'b'c'd' + a'b'c'd + a'b'cd' + a'b'cd + a'b'c'd + a'b'cd' + a'b'cd + a'b'cd' \\ &\quad + a'bc'd + a'bcd + abcd' + ab'cd' \end{aligned} \quad (4-9)$$

Duplicate terms have been crossed out, because $X + X = X$. This expression can then be converted to decimal notation:

$$\begin{aligned} f &= a'b'c'd' + a'b'c'd + a'b'cd' + a'b'cd + a'bc'd + a'bcd + abcd' + ab'cd' \\ &\quad 0000 \quad 0001 \quad 0010 \quad 0011 \quad 0101 \quad 0111 \quad 1110 \quad 1010 \\ f &= \Sigma m(0, 1, 2, 3, 5, 7, 10, 14) \end{aligned} \quad (4-10)$$

The maxterm expansion for f can then be obtained by listing the decimal integers (in the range 0 to 15) which do not correspond to minterms of f :

$$f = \Pi M(4, 6, 8, 9, 11, 12, 13, 15)$$

An alternate way of finding the maxterm expansion is to factor f to obtain a product of sums, introduce the missing variables in each sum term by using $XX' = 0$, and then factor again to obtain the maxterms. For Equation (4-9),

$$\begin{aligned}
 f &= a'(b' + d) + acd' \\
 &= (a' + cd')(a + b' + d) = (a' + c)(a' + d')(a + b' + d) \\
 &= (a' + bb' + c + dd')(a' + bb' + cc' + d')(a + b' + cc' + d) \\
 &= (a' + bb' + c + d)(a' + bb' + c + d')(a + b' + cc' + d) \\
 &\quad (a' + bb' + c' + d')(a + b' + cc' + d) \\
 &= (a' + b + c + d)(a' + b' + c + d)(a' + b + c + d')(a' + b' + c + d') \\
 &\quad \begin{array}{cccc}
 1000 & 1100 & 1001 & 1101 \\
 (a' + b + c' + d')(a' + b' + c' + d')(a + b' + c + d)(a + b' + c' + d) \\
 1011 & 1111 & 0100 & 0110
 \end{array} \\
 &= \Pi M(4, 6, 8, 9, 11, 12, 13, 15) \quad (4-11)
 \end{aligned}$$

Note that when translating the maxterms to decimal notation, a primed variable is first replaced with a 1 and an unprimed variable with a 0.

Because the terms in the minterm expansion of a function F correspond one-to-one with the rows of the truth table for which $F = 1$, the minterm expansion of F is unique. Thus, we can prove that an equation is valid by finding the minterm expansion of each side and showing that these expansions are the same.

Example

Show that $a'c + b'c' + ab = a'b' + bc + ac'$.

We will find the minterm expansion of each side by supplying the missing variables. For the left side,

$$\begin{aligned}
 &a'c(b + b') + b'c'(a + a') + ab(c + c') \\
 &= a'bc + a'b'c + ab'c' + a'b'c' + abc + abc' \\
 &= m_3 + m_1 + m_4 + m_0 + m_7 + m_6
 \end{aligned}$$

For the right side,

$$\begin{aligned}
 &a'b'(c + c') + bc(a + a') + ac'(b + b') \\
 &= a'b'c + a'b'c' + abc + a'bc + abc' + ab'c' \\
 &= m_1 + m_0 + m_7 + m_3 + m_6 + m_4
 \end{aligned}$$

Because the two minterm expansions are the same, the equation is valid.

4.4 General Minterm and Maxterm Expansions

Table 4-2 represents a truth table for a general function of three variables. Each a_i is a constant with a value of 0 or 1. To completely specify a function, we must assign values to all of the a_i 's. Because each a_i can be specified in two ways, there are 2^8

TABLE 4-2
General Truth Table
for Three Variables

A	B	C	F
0	0	0	a_0
0	0	1	a_1
0	1	0	a_2
0	1	1	a_3
1	0	0	a_4
1	0	1	a_5
1	1	0	a_6
1	1	1	a_7

ways of filling the F column of the truth table; therefore, there are 256 different functions of three variables (this includes the degenerate cases, F identically equal to 0 and F identically equal to 1). For a function of n variables, there are 2^n rows in the truth table, and because the value of F can be 0 or 1 for each row, there are 2^{2^n} possible functions of n variables.

From Table 4-2, we can write the minterm expansion for a general function of three variables as follows:

$$F = a_0m_0 + a_1m_1 + a_2m_2 + \cdots + a_7m_7 = \sum_{i=0}^7 a_im_i \quad (4-12)$$

Note that if $a_i = 1$, minterm m_i is present in the expansion; if $a_i = 0$, the corresponding minterm is not present. The maxterm expansion for a general function of three variables is

$$F = (a_0 + M_0)(a_1 + M_1)(a_2 + M_2) \cdots (a_7 + M_7) = \prod_{i=0}^7 (a_i + M_i) \quad (4-13)$$

Note that if $a_i = 1$, $a_i + M_i = 1$, and M_i drops out of the expansion; however, M_i is present if $a_i = 0$.

From Equation (4-13), the minterm expansion of F' is

$$F' = \left[\prod_{i=0}^7 (a_i + M_i) \right]' = \sum_{i=0}^7 a'_i M'_i = \sum_{i=0}^7 a'_i m_i \quad (4-14)$$

Note that all minterms which are not present in F are present in F' .

From Equation (4-12), the maxterm expansion of F' is

$$F' = \left[\sum_{i=0}^7 a_im_i \right]' = \prod_{i=0}^7 (a'_i + m'_i) = \prod_{i=0}^7 (a'_i + M_i) \quad (4-15)$$

Note that all maxterms which are not present in F are present in F' . Generalizing Equations (4-12), (4-13), (4-14), and (4-15) to n variables, we have

$$F = \sum_{i=0}^{2^n-1} a_im_i = \prod_{i=0}^{2^n-1} (a_i + M_i) \quad (4-16)$$

$$F' = \sum_{i=0}^{2^n-1} a'_i m_i = \prod_{i=0}^{2^n-1} (a'_i + M_i) \tag{4-17}$$

Given two different minterms of n variables, m_i and m_j , at least one variable appears complemented in one of the minterms and uncomplemented in the other. Therefore, if $i \neq j$, $m_i m_j = 0$. For example, for $n = 3$, $m_1 m_3 = (A'B'C)(A'BC) = 0$. Given minterm expansions for two functions

$$f_1 = \sum_{i=0}^{2^n-1} a_i m_i \qquad f_2 = \sum_{j=0}^{2^n-1} b_j m_j \tag{4-18}$$

the product is

$$\begin{aligned} f_1 f_2 &= \left(\sum_{i=0}^{2^n-1} a_i m_i \right) \left(\sum_{j=0}^{2^n-1} b_j m_j \right) = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} a_i b_j m_i m_j \\ &= \sum_{i=0}^{2^n-1} a_i b_i m_i \quad (\text{because } m_i m_j = 0 \text{ unless } i = j) \end{aligned} \tag{4-19}$$

Note that all of the cross-product terms ($i \neq j$) drop out so that $f_1 f_2$ contains only those minterms which are present in both f_1 and f_2 . For example, if

$$\begin{aligned} f_1 &= \Sigma m(0, 2, 3, 5, 9, 11) \qquad \text{and} \qquad f_2 = \Sigma m(0, 3, 9, 11, 13, 14) \\ f_1 f_2 &= \Sigma m(0, 3, 9, 11) \end{aligned}$$

Table 4-3 summarizes the procedures for conversion between minterm and maxterm expansions of F and F' , assuming that all expansions are written as lists of decimal numbers. When using this table, keep in mind that the truth table for an n -variable function has 2^n rows so that the minterm (or maxterm) numbers range from 0 to $2^n - 1$. Table 4-4 illustrates the application of Table 4-3 to the three-variable function given in Figure 4-1.

TABLE 4-3
Conversion of
Forms

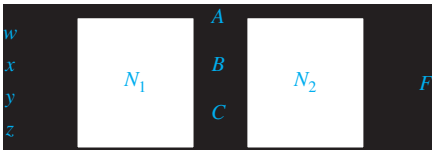
		DESIRED FORM			
		Minterm Expansion of F	Maxterm Expansion of F	Minterm Expansion of F'	Maxterm Expansion of F'
GIVEN FORM	Minterm Expansion of F	_____	maxterm nos. are those nos. not on the minterm list for F	list minterms not present in F	maxterm nos. are the same as minterm nos. of F
	Maxterm Expansion of F	minterm nos. are those nos. not on the maxterm list for F	_____	minterm nos. are the same as maxterm nos. of F	list maxterms not present in F

TABLE 4-4
Application of
Table 4.3

		DESIRED FORM		
GIVEN FORM		Minterm Expansion of f	Maxterm Expansion of f	Minterm Expansion of f'
	$f =$ $\Sigma m(3, 4, 5, 6, 7)$	_____	$\Pi M(0, 1, 2)$	$\Sigma m(0, 1, 2)$
	$f =$ $\Pi M(0, 1, 2)$	$\Sigma m(3, 4, 5, 6, 7)$	_____	$\Sigma m(0, 1, 2)$
				$\Pi M(3, 4, 5, 6, 7)$

4.5 Incompletely Specified Functions

A large digital system is usually divided into many subcircuits. Consider the following example in which the output of circuit N_1 drives the input of circuit N_2 .



Let us assume that the output of N_1 does not generate all possible combinations of values for A, B , and C . In particular, we will assume that there are no combinations of values for w, x, y , and z which cause A, B , and C to assume values of 001 or 110. Hence, when we design N_2 , it is not necessary to specify values of F for $ABC = 001$ or 110 because these combinations of values can never occur as inputs to N_2 . For example, F might be specified by Table 4-5.

The X's in the table indicate that we don't care whether the value of 0 or 1 is assigned to F for the combinations $ABC = 001$ or 110. In this example, we don't care what the value of F is because these input combinations never occur anyway. The function F is then *incompletely specified*. The minterms $A'B'C$ and ABC' are referred to as don't-care minterms, since we don't care whether they are present in the function or not.

TABLE 4-5
Truth Table with
Don't-Cares

$A B C$	F
0 0 0	1
0 0 1	X
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	X
1 1 1	1

When we realize the function, we must specify values for the don't-cares. It is desirable to choose values which will help simplify the function. If we assign the value 0 to both X's, then

$$F = A'B'C' + A'BC + ABC = A'B'C' + BC$$

If we assign 1 to the first X and 0 to the second, then

$$F = A'B'C' + A'B'C + A'BC + ABC = A'B' + BC$$

If we assign 1 to both X's, then

$$F = A'B'C' + A'B'C + A'BC + ABC' + ABC = A'B' + BC + AB$$

The second choice of values leads to the simplest solution.

We have seen one way in which incompletely specified functions can arise, and there are many other ways. In the preceding example, don't-cares were present because certain combinations of circuit inputs did not occur. In other cases, all input combinations may occur, but the circuit output is used in such a way that we do not care whether it is 0 or 1 for certain input combinations.

When writing the minterm expansion for an incompletely specified function, we will use m to denote the required minterms and d to denote the don't-care minterms. Using this notation, the minterm expansion for Table 4-5 is

$$F = \Sigma m(0, 3, 7) + \Sigma d(1, 6)$$

For each don't-care minterm there is a corresponding don't-care maxterm. For example, if $F = X$ (don't-care) for input combination 001, m_1 is a don't-care minterm and M_1 is a don't-care maxterm. We will use D to represent a don't-care maxterm, and we write the maxterm expansion of the function in Table 4-5 as

$$F = \Pi M(2, 4, 5) \bullet \Pi D(1, 6)$$

which implies that maxterms M_2 , M_4 , and M_5 are present in F and don't-care maxterms M_1 and M_6 are optional.

4.6 Examples of Truth Table Construction

Example 1

We will design a simple binary adder that adds two 1-bit binary numbers, a and b , to give a 2-bit sum. The numeric values for the adder inputs and output are as follows:

a	b	Sum
0	0	00 (0 + 0 = 0)
0	1	01 (0 + 1 = 1)
1	0	01 (1 + 0 = 1)
1	1	10 (1 + 1 = 2)

We will represent inputs to the adder by the logic variables A and B and the 2-bit sum by the logic variables X and Y , and we construct a truth table:

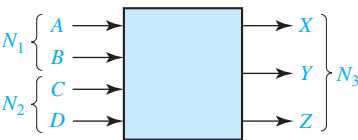
A	B	X	Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Because a numeric value of 0 is represented by a logic 0 and a numeric value of 1 by a logic 1, the 0's and 1's in the truth table are exactly the same as in the previous table. From the truth table,

$$X = AB \text{ and } Y = A'B + AB' = A \oplus B$$

Example 2

An adder is to be designed which adds two 2-bit binary numbers to give a 3-bit binary sum. Find the truth table for the circuit. The circuit has four inputs and three outputs as shown:



TRUTH TABLE:

N_1		N_2		N_3		
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Inputs A and B taken together represent a binary number N_1 . Inputs C and D taken together represent a binary number N_2 . Outputs X , Y , and Z taken together represent a binary number N_3 , where $N_3 = N_1 + N_2$ (+ of course represents ordinary addition here).

In this example we have used A , B , C , and D to represent both numeric values and logic values, but this should not cause any confusion because the numeric and

logic values are the same. In forming the truth table, the variables were treated like binary numbers having *numeric* values. Now we wish to derive the switching functions for the output variables. In doing so, we will treat $A, B, C, D, X, Y,$ and Z as switching variables having nonnumeric values 0 and 1. (Remember that in this case the 0 and 1 may represent low and high voltages, open and closed switches, etc.)

From inspection of the table, the output functions are

$$X(A, B, C, D) = \Sigma m(7, 10, 11, 13, 14, 15)$$

$$Y(A, B, C, D) = \Sigma m(2, 3, 5, 6, 8, 9, 12, 15)$$

$$Z(A, B, C, D) = \Sigma m(1, 3, 4, 6, 9, 11, 12, 14)$$

Example 3

Design an error detector for 6-3-1-1 binary-coded-decimal digits. The output (F) is to be 1 iff the four inputs (A, B, C, D) represent an invalid code combination.

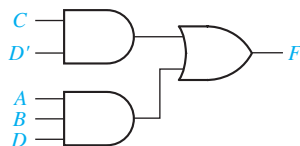
The valid 6-3-1-1 code combinations are listed in Table 1-2. If any other combination occurs, this is not a valid 6-3-1-1 binary-coded-decimal digit, and the circuit output should be $F = 1$ to indicate that an error has occurred. This leads to the following truth table:

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

The corresponding output function is

$$\begin{aligned}
 F &= \Sigma m(2, 6, 10, 13, 14, 15) \\
 &= A'B'CD' + A'BCD' + AB'CD' + ABCD' + ABC'D + ABCD \\
 &= \underline{A'CD'} + \underline{ACD'} + ABD = CD' + ABD
 \end{aligned}$$

The realization using AND and OR gates is



Example 4

The four inputs to a circuit (A, B, C, D) represent an 8-4-2-1 binary-coded-decimal digit. Design the circuit so that the output (Z) is 1 iff the decimal number represented by the inputs is exactly divisible by 3. Assume that only valid BCD digits occur as inputs.

The digits 0, 3, 6, and 9 are exactly divisible by 3, so $Z = 1$ for the input combinations $ABCD = 0000, 0011, 0110$, and 1001 . The input combinations 1010, 1011, 1100, 1101, 1110, and 1111 do not represent valid BCD digits and will never occur, so Z is a don't-care for these combinations. This leads to the following truth table:

A	B	C	D	Z
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

The corresponding output function is

$$Z = \sum m(0, 3, 6, 9) + \sum d(10, 11, 12, 13, 14, 15)$$

In order to find the simplest circuit which will realize Z , we must choose some of the don't-cares (X's) to be 0 and some to be 1. The easiest way to do this is to use a Karnaugh map as described in Unit 5.

4.7 Design of Binary Adders and Subtractors

In this section, we will design a parallel adder that adds two 4-bit unsigned binary numbers and a carry input to give a 4-bit sum and a carry output (see Figure 4-2). One approach would be to construct a truth table with nine inputs and five outputs and then derive and simplify the five output equations. Because each equation would be a function of nine variables before simplification, this approach would be very difficult, and the resulting logic circuit would be very complex. A better method is to design a logic module that adds two bits and a carry, and then connect four of these modules together to form a 4-bit adder as shown in Figure 4-3. Each of the modules is called a *full adder*. The carry output from the first full adder serves as the carry input to the second full adder, etc.

FIGURE 4-2
Parallel Adder
for 4-Bit Binary
Numbers

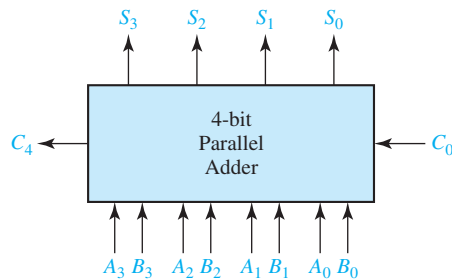
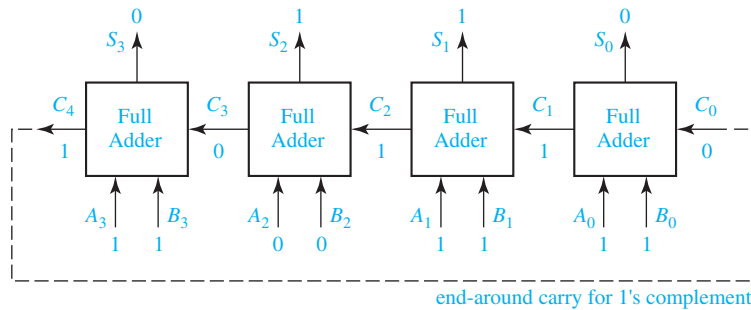


FIGURE 4-3
Parallel Adder
Composed of Four
Full Adders



In the example of Figure 4-3, we perform the following addition:

$$\begin{array}{r}
 10110 \quad (\text{carries}) \\
 1011 \\
 + 1011 \\
 \hline
 10110
 \end{array}$$

The full adder to the far right adds $A_0 + B_0 + C_0 = 1 + 1 + 0$ to give a sum of 10_2 , which gives a sum $S_0 = 0$ and a carry out of $C_1 = 1$. The next full adder adds $A_1 + B_1 + C_1 = 1 + 1 + 1 = 11_2$, which gives a sum $S_1 = 1$ and a carry $C_2 = 1$. The carry continues to propagate from right to left until the left cell produces a final carry of $C_4 = 1$.

FIGURE 4-4
Truth Table for a
Full Adder

X	Y	C_{in}	C_{out}	Sum
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

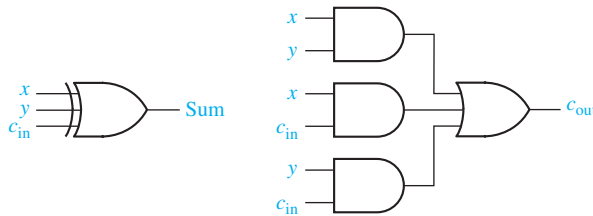
Figure 4-4 gives the truth table for a full adder with inputs X , Y , and C_{in} . The outputs for each row of the table are found by adding up the input bits ($X + Y + C_{in}$) and splitting the result into a carry out (C_{i+1}) and a sum bit (S_i). For example, in the 101 row $1 + 0 + 1 = 10_2$, so $C_{i+1} = 1$ and $S_i = 0$. Figure 4-5 shows the implementation of the full adder using gates. The logic equations for the full adder derived from the truth table are

$$\begin{aligned}
 Sum &= X'Y'C_{in} + X'YC'_{in} + XY'C'_{in} + XYC_{in} \\
 &= X'(Y'C_{in} + YC'_{in}) + X(Y'C'_{in} + YC_{in}) \\
 &= X'(Y \oplus C_{in}) + X(Y \oplus C_{in})' = X \oplus Y \oplus C_{in}
 \end{aligned} \tag{4-20}$$

$$\begin{aligned}
 C_{out} &= X'YC_{in} + XY'C_{in} + XYC'_{in} + XYC_{in} \\
 &= (X'YC_{in} + XYC_{in}) + (XY'C_{in} + XYC_{in}) + (XYC'_{in} + XYC_{in}) \\
 &= YC_{in} + XC_{in} + XY
 \end{aligned} \tag{4-21}$$

Note that the term XYC_{in} was used three times in simplifying C_{out} . Figure 4-5 shows the logic circuit for Equations (4-20) and (4-21).

FIGURE 4-5
Implementation of
Full Adder



Although designed for unsigned binary numbers, the parallel adder of Figure 4-3 can also be used for signed binary numbers with negative numbers expressed in complement form. When 2's complement is used, the last carry (C_4) is discarded, and there is no carry into the first cell. Because $C_0 = 0$, the equations for the first cell may be simplified to

$$S_0 = A_0 \oplus B_0 \text{ and } C_1 = A_0 B_0$$

When 1's complement is used, the end-around carry is accomplished by connecting C_4 to the C_0 input, as shown by the dashed line in Figure 4-3.

When adding signed binary numbers with negative numbers expressed in complement form, the sign bit of the sum is wrong when an overflow occurs. That is, an overflow has occurred if adding two positive numbers gives a negative result, or adding two negative numbers gives a positive result. We will define a signal V that is

1 when an overflow occurs. For Figure 4-3, we can use the sign bits of A , B , and S (the sum) to determine the value of V :

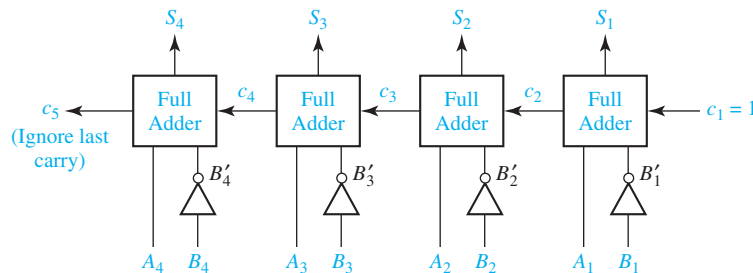
$$V = A_3'B_3'S_3 + A_3B_3S_3' \quad (4-22)$$

If the number of bits is large, a parallel binary adder of the type shown in Figure 4-4 may be rather slow because the carry generated in the first cell might have to propagate all of the way to the last cell. Other types of adders, such as a carry-look-ahead adder,² may be used to speed up the carry propagation.

Subtraction of binary numbers is most easily accomplished by adding the complement of the number to be subtracted. To compute $A - B$, add the complement of B to A . This gives the correct answer because $A + (-B) = A - B$. Either 1's or 2's complement is used depending on the type of adder employed.

The circuit of Figure 4-6 may be used to form $A - B$ using the 2's complement representation for negative numbers. The 2's complement of B can be formed by first finding the 1's complement and then adding 1. The 1's complement is formed by inverting each bit of B , and the addition of 1 is effectively accomplished by putting a 1 into the carry input of the first full adder.

FIGURE 4-6
Binary Subtractor
Using Full Adders



Example

$A = 0110 \quad (+6)$
 $B = 0011 \quad (+3)$

The adder output is

$$\begin{array}{r} 0110 \quad (+6) \\ + 1100 \quad (1\text{'s complement of } 3) \\ + \quad 1 \quad (\text{first carry input}) \\ \hline (1) \quad 0011 = 3 = 6 - 3 \end{array}$$

Alternatively, direct subtraction can be accomplished by employing a full subtracter in a manner analogous to a full adder. A block diagram for a parallel subtracter which subtracts Y from X is shown in Figure 4-7. The first two bits are subtracted in the rightmost cell to give a difference d_1 , and a borrow signal ($b_2 = 1$) is generated if it is necessary to borrow from the next column. A typical cell (cell i) has inputs x_i , y_i , and b_i , and outputs b_{i+1} and d_i . An input $b_i = 1$ indicates that we must borrow 1 from x_i in that cell, and borrowing 1 from x_i is equivalent to subtracting 1 from x_i . In cell i , bits b_i and

²See, for example, J. F. Wakerly, *Digital Design Principles and Practices*, 4th ed (Prentice Hall, 2006).

FIGURE 4-7
Parallel Subtractor

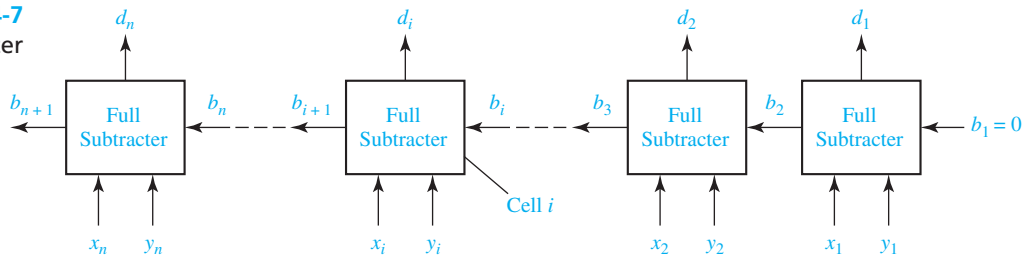


TABLE 4-6
Truth Table for
Binary Full
Subtractor

x_i	y_i	b_i	b_{i+1}	d_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

y_i are subtracted from x_i to form the difference d_i , and a borrow signal ($b_{i+1} = 1$) is generated if it is necessary to borrow from the next column.

Table 4-6 gives the truth table for a binary full subtractor. Consider the following case, where $x_i = 0$, $y_i = 1$ and $b_i = 1$:

	Column i Before Borrow	Column i After Borrow
x_i	0	10
$-b_i$	-1	-1
$-y_i$	-1	-1
d_i		0 ($b_{i+1} = 1$)

Note that in column i , we cannot immediately subtract y_i and b_i from x_i . Hence, we must borrow from column $i + 1$. Borrowing 1 from column $i + 1$ is equivalent to setting b_{i+1} to 1 and adding 10 (2_{10}) to x_i . We then have $d_i = 10 - 1 - 1 = 0$. Verify that Table 4-6 is correct for the other input combinations and use it to work out several examples of binary subtraction.

Problems

- 4.1 Represent each of the following sentences by a Boolean equation.
- (a) The company safe should be unlocked only when Mr. Jones is in the office or Mr. Evans is in the office, and only when the company is open for business, and only when the security guard is present.

- (b) You should wear your overshoes if you are outside in a heavy rain and you are wearing your new suede shoes, or if your mother tells you to.
- (c) You should laugh at a joke if it is funny, it is in good taste, and it is not offensive to others, or if it is told in class by your professor (regardless of whether it is funny and in good taste) and it is not offensive to others.
- (d) The elevator door should open if the elevator is stopped, it is level with the floor, and the timer has not expired, or if the elevator is stopped, it is level with the floor, and a button is pressed.

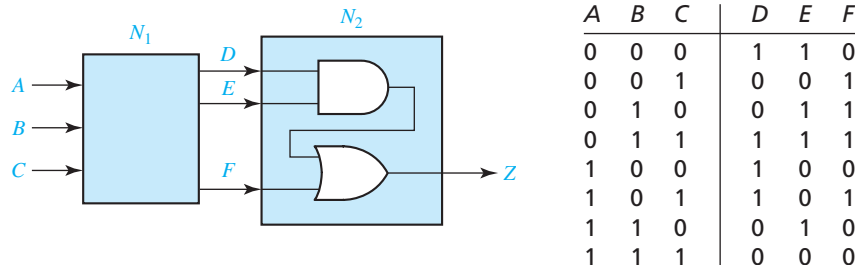
4.2 A flow rate sensing device used on a liquid transport pipeline functions as follows. The device provides a 5-bit output where all five bits are zero if the flow rate is less than 10 gallons per minute. The first bit is 1 if the flow rate is at least 10 gallons per minute; the first and second bits are 1 if the flow rate is at least 20 gallons per minute; the first, second, and third bits are 1 if the flow rate is at least 30 gallons per minute; and so on. The five bits, represented by the logical variables A , B , C , D , and E , are used as inputs to a device that provides two outputs Y and Z .

- (a) Write an equation for the output Y if we want Y to be 1 iff the flow rate is less than 30 gallons per minute.
- (b) Write an equation for the output Z if we want Z to be 1 iff the flow rate is at least 20 gallons per minute but less than 50 gallons per minute.

4.3 Given $F_1 = \sum m(0, 4, 5, 6)$ and $F_2 = \sum m(0, 3, 6, 7)$ find the minterm expression for $F_1 + F_2$. State a general rule for finding the expression for $F_1 + F_2$ given the minterm expansions for F_1 and F_2 . Prove your answer by using the general form of the minterm expansion.

- 4.4** (a) How many switching functions of two variables (x and y) are there?
 (b) Give each function in truth table form and in reduced algebraic form.

4.5 A combinational circuit is divided into two subcircuits N_1 and N_2 as shown. The truth table for N_1 is given. Assume that the input combinations $ABC = 110$ and $ABC = 010$ never occur. Change as many of the values of D , E , and F to don't-cares as you can without changing the value of the output Z .



4.6 Work (a) and (b) with the following truth table:

A	B	C	F	G
0	0	0	1	0
0	0	1	X	1
0	1	0	0	X
0	1	1	0	1
1	0	0	0	0
1	0	1	X	1
1	1	0	1	X
1	1	1	1	1

- (a) Find the simplest expression for F , and specify the values of the don't-cares that lead to this expression.
 (b) Repeat (a) for G . (*Hint*: Can you make G the same as one of the inputs by properly choosing the values for the don't-care?)

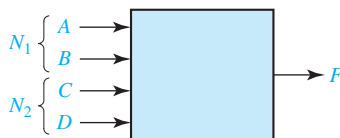
4.7 Each of three coins has two sides, heads and tails. Represent the heads or tails status of each coin by a logical variable (A for the first coin, B for the second coin, and C for the third) where the logical variable is 1 for heads and 0 for tails. Write a logic function $F(A, B, C)$ which is 1 iff exactly one of the coins is heads after a toss of the coins. Express F

- (a) as a minterm expansion.
 (b) as a maxterm expansion.

4.8 A switching circuit has four inputs as shown. A and B represent the first and second bits of a binary number N_1 . C and D represent the first and second bits of a binary number N_2 . The output is to be 1 only if the product $N_1 \times N_2$ is less than or equal to 2.

- (a) Find the minterm expansion for F .
 (b) Find the maxterm expansion for F .

Express your answers in both decimal notation and algebraic form.



4.9 Given: $F(a, b, c) = abc' + b'$.

- (a) Express F as a minterm expansion. (Use m -notation.)
 (b) Express F as a maxterm expansion. (Use M -notation.)
 (c) Express F' as a minterm expansion. (Use m -notation.)
 (d) Express F' as a maxterm expansion. (Use M -notation.)

4.10 Work Problem 4.9 using:

$$F(a, b, c, d) = (a + b + d) (a' + c) (a' + b' + c') (a + b + c' + d')$$

- 4.11** (a) Implement a full subtracter using a minimum number of gates.
 (b) Compare the logic equations for the full adder and full subtracter. What is the relation between s_i and d_i ? Between c_{i+1} and b_{i+1} ?

4.12 Design a circuit which will perform the following function on three 4-bit numbers:

$$(X_3X_2X_1X_0 + Y_3Y_2Y_1Y_0) - Z_3Z_2Z_1Z_0$$

It will give a result $S_3S_2S_1S_0$, a carry, and a borrow. Use eight full adders and any other type of gates. Assume that negative numbers are represented in 2's complement.

4.13 A combinational logic circuit has four inputs (A , B , C , and D) and one output Z . The output is 1 iff the input has three consecutive 0's or three consecutive 1's. For example, if $A = 1$, $B = 0$, $C = 0$, and $D = 0$, then $Z = 1$, but if $A = 0$, $B = 1$, $C = 0$, and $D = 0$, then $Z = 0$. Design the circuit using one four-input OR gate and four three-input AND gates.

4.14 Design a combinational logic circuit which has one output Z and a 4-bit input $ABCD$ representing a binary number. Z should be 1 iff the input is at least 5, but is no greater than 11. Use one OR gate (three inputs) and three AND gates (with no more than three inputs each).

4.15 A logic circuit realizing the function f has four inputs A , B , C , and D . The three inputs A , B , and C are the binary representation of the digits 0 through 7 with A being the most-significant bit. The input D is an odd-parity bit, i.e., the value of D is such that A , B , C , and D always contain an odd number of 1's. (For example, the digit 1 is represented by $ABC = 001$ and $D = 0$, and the digit 3 is represented by $ABCD = 0111$.) The function f has value 1 if the input digit is a prime number. (A number is prime if it is divisible only by itself and 1; 1 is considered to be prime and 0 is not.)

- (a) List the minterms and don't-care minterms of f in algebraic form.
 (b) List the maxterms and don't-care maxterms of f in algebraic form.

4.16 A priority encoder circuit has four inputs, x_3 , x_2 , x_1 , and x_0 . The circuit has three outputs: z , y_1 , and y_0 . If one of the inputs is 1, z is 1 and y_1 and y_0 represent a 2-bit, binary number whose value equals the index of the highest numbered input that is 1. For example, if x_2 is 1 and x_3 is 0, then the outputs are $z = 1$ and $y_1 = 1$ and $y_0 = 0$. If all inputs are 0, $z = 0$ and y_1 and y_0 are don't-cares.

- (a) List in decimal form the minterms and don't-care minterms of each output.
 (b) List in decimal form the maxterms and don't-care maxterms of each output.

4.17 The 9's complement of a decimal digit d (0 to 9) is defined to be $9 - d$. A logic circuit produces the 9's complement of an input digit where the input and output

digits are represented in BCD. Label the inputs A , B , C , and D , and label the outputs W , X , Y and Z .

- Determine the minterms and don't-care minterms for each of the outputs.
- Determine the maxterms and don't-care maxterms for each of the outputs.

4.18 Repeat Problem 4.17 for the case where the input and output digits are represented using the 4-2-2-1 weighted code. (If only one weight of 2 is required for decimal digits less than 5, select the rightmost 2. In addition, select the codes so that $W = A'$, $X = B'$, $Y = C'$, and $Z = D'$. (There are two possible codes with these restrictions.)

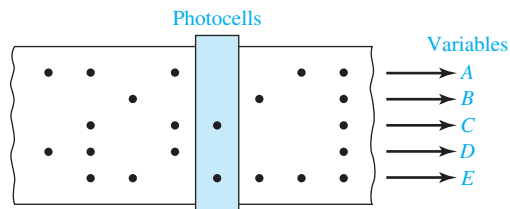
4.19 Each of the following sentences has two possible interpretations depending on whether the AND or OR is done first. Write an equation for each interpretation.

- The buzzer will sound if the key is in the ignition switch, and the car door is open, or the seat belts are not fastened.
- You will gain weight if you eat too much, or you do not exercise enough, and your metabolism rate is too low.
- The speaker will be damaged if the volume is set too high, and loud music is played, or the stereo is too powerful.
- The roads will be very slippery if it snows, or it rains, and there is oil on the road.

4.20 A bank vault has three locks with a different key for each lock. Each key is owned by a different person. To open the door, at least two people must insert their keys into the assigned locks. The signal lines A , B , and C are 1 if there is a key inserted into lock 1, 2, or 3, respectively. Write an equation for the variable Z which is 1 iff the door should open.

4.21 A paper tape reader used as an input device to a computer has five rows of holes as shown. A hole punched in the tape indicates a logic 1, and no hole indicates a logic 0. As each hole pattern passes under the photocells, the pattern is translated into logic signals on lines A , B , C , D , and E . All patterns of holes indicate a valid character with two exceptions. A pattern consisting of none of the possible holes punched is not used because it is impossible to distinguish between this pattern and the unpunched space between patterns. An incorrect pattern punched on the tape is erased by punching all five holes in that position. Therefore, a valid character punched on the tape will have at least one hole but will not have all five holes punched.

- Write an equation for a variable Z which is 1 iff a valid character is being read.
- Write an equation for a variable Y which is 1 iff the hole pattern being read has holes punched only in rows C and E .



4.22 A computer interface to a line printer has seven data lines that control the movement of the paper and the print head and determine which character to print. The data lines are labeled A, B, C, D, E, F , and G , and each represents a binary 0 or 1. When the data lines are interpreted as a 7-bit binary number with line A being the most significant bit, the data lines can represent the numbers 0 to 127_{10} . The number 13_{10} is the command to return the print head to the beginning of a line, the number 10_{10} means to advance the paper by one line, and the numbers 32_{10} to 127_{10} represent printing characters.

- Write an equation for the variable X which is 1 iff the data lines indicate a command to return the print head to the beginning of the line.
- Write an equation for the variable Y which is 1 iff there is an advance paper command on the data lines.
- Write an equation for the variable Z which is 1 iff the data lines indicate a printable character. (*Hint:* Consider the binary representations of the numbers 0–31 and 32–127 and write the equation for Z with only two terms.)

4.23 Given $F_1 = \prod M(0, 4, 5, 6)$ and $F_2 = \prod M(0, 4, 7)$, find the maxterm expansion for $F_1 F_2$. State a general rule for finding the maxterm expansion of $F_1 F_2$ given the maxterm expansions of F_1 and F_2 .

Prove your answer by using the general form of the maxterm expansion.

4.24 Given $F_1 = \prod M(0, 4, 5, 6)$ and $F_2 = \prod M(0, 4, 7)$, find the maxterm expansion for $F_1 + F_2$.

State a general rule for finding the maxterm expansion of $F_1 + F_2$, given the maxterm expansions of F_1 and F_2 .

Prove your answer by using the general form of the maxterm expansion.

4.25 Four chairs are placed in a row:



Each chair may be occupied (1) or empty (0). Give the minterm and maxterm expansion for each logic function described.

- $F(A, B, C, D)$ is 1 iff there are no adjacent empty chairs.
- $G(A, B, C, D)$ is 1 iff the chairs on the ends are both empty.
- $H(A, B, C, D)$ is 1 iff at least three chairs are full.
- $J(A, B, C, D)$ is 1 iff there are more people sitting in the left two chairs than in the right two chairs.

4.26 Four chairs (A, B, C , and D) are placed in a circle: A next to B , B next to C , C next to D , and D next to A . Each chair may be occupied (1) or empty (0). Give the minterm and maxterm expansion for each of the following logic functions:

- $F(A, B, C, D)$ is 1 iff there are no adjacent empty chairs.
- $G(A, B, C, D)$ is 1 iff there are at least three adjacent empty chairs.

- (c) $H(A, B, C, D)$ is 1 iff at least three chairs are full.
 (d) $J(A, B, C, D)$ is 1 iff there are more people sitting in chairs A and B than chairs C and D .

4.27 Given $f(a, b, c) = a(b + c')$.

- (a) Express f as a minterm expansion (use m -notation).
 (b) Express f as maxterm expansion (use M -notation).
 (c) Express f' as a minterm expansion (use m -notation).
 (d) Express f' as a maxterm expansion (use M -notation).

4.28 Work Problem 4.27 using $f(a, b, c, d) = acd + bd' + a'c'd + ab'cd + a'b'cd'$.

4.29 Find both the minterm expansion and maxterm expansion for the following functions, using *algebraic manipulations*:

- (a) $f(A, B, C, D) = AB + A'CD$
 (b) $f(A, B, C, D) = (A + B + D')(A' + C)(C + D)$

4.30 Given $F'(A, B, C, D) = \sum m(0, 1, 2, 6, 7, 13, 15)$.

- (a) Find the minterm expansion for F (both decimal and algebraic form).
 (b) Find the maxterm expansion for F (both decimal and algebraic form).

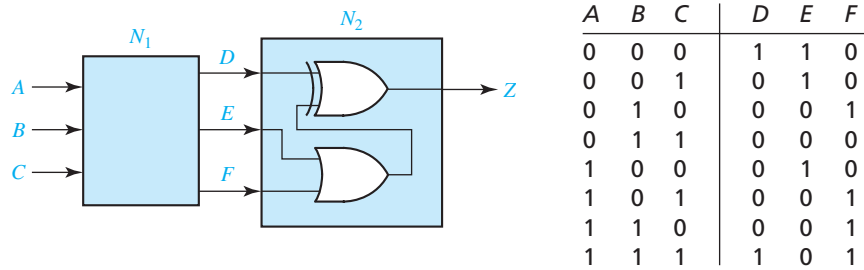
4.31 Repeat Problem 4.30 for $F'(A, B, C, D) = \sum m(1, 2, 5, 6, 10, 15)$.

4.32 Work parts (a) through (d) with the given truth table.

A	B	C	F_1	F_2	F_3	F_4
0	0	0	1	1	0	1
0	0	1	X	0	0	0
0	1	0	0	1	X	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	X	0	1	0
1	1	0	0	X	X	X
1	1	1	1	X	1	X

- (a) Find the simplest expression for F_1 , and specify the values for the don't-cares that lead to this expression.
 (b) Repeat for F_2 .
 (c) Repeat for F_3 .
 (d) Repeat for F_4 .

4.33 Work Problem 4.5 using the following circuits and truth table. Assume that the input combinations of $ABC = 011$ and $ABC = 110$ will never occur.



- 4.34** Work Problem 4.7 for the following logic functions:
- (a) $G_1(A, B, C)$ is 1 iff all the coins landed on the same side (heads or tails).
 - (b) $G_2(A, B, C)$ is 1 iff the second coin landed on the same side as the first coin.
- 4.35** A combinational circuit has four inputs (A, B, C, D) and three outputs (X, Y, Z). XYZ represents a binary number whose value equals the number of 1's at the input. For example if $ABCD = 1011$, $XYZ = 011$.
- (a) Find the minterm expansions for X, Y , and Z .
 - (b) Find the maxterm expansions for Y and Z .
- 4.36** A combinational circuit has four inputs (A, B, C, D) and four outputs (W, X, Y, Z). $WXYZ$ represents an excess-3 coded number whose value equals the number of 1's at the input. For example, if $ABCD = 1101$, $WXYZ = 0110$.
- (a) Find the minterm expansions for X, Y , and Z .
 - (b) Find the maxterm expansions for Y and Z .
- 4.37** A combinational circuit has four inputs (A, B, C, D), which represent a binary-coded-decimal digit. The circuit has two groups of four outputs— S, T, U, V , and W, X, Y, Z . Each group represents a BCD digit. The output digits represent a decimal number which is five times the input number. For example, if $ABCD = 0111$, the outputs are 0011 0101. Assume that invalid BCD digits do not occur as inputs.
- (a) Construct the truth table.
 - (b) Write down the minimum expressions for the outputs by inspection of the truth table. (*Hint:* Try to match output columns in the table with input columns.)
- 4.38** Work Problem 4.37 where the BCD outputs represent a decimal number that is 1 more than four times the input number. For example, if $ABCD = 0011$, the outputs are 0001 0011.
- 4.39** Design a circuit which will add a 4-bit binary number to a 5-bit binary number. Use five full adders. Assume negative numbers are represented in 2's complement. (*Hint:* How do you make a 4-bit binary number into a 5-bit binary number, without making a negative number positive or a positive number negative? Try writing

down the representation for -3 as a 3-bit 2's complement number, a 4-bit 2's complement number, and a 5-bit 2's complement number. Recall that one way to find the 2's complement of a binary number is to complement *all* bits to the left of the first 1.)

- 4.40** A half adder is a circuit that adds two bits to give a sum and a carry. Give the truth table for a half adder, and design the circuit using only two gates. Then design a circuit which will find the 2's complement of a 4-bit binary number. Use four half adders and any additional gates. (*Hint:* Recall that one way to find the 2's complement of a binary number is to complement *all* bits, and then add 1.)
- 4.41** (a) Write the switching function $f(x, y) = x + y$ as a sum of minterms and as a product of maxterms.
- (b) Consider the Boolean algebra of four elements $\{0, 1, a, b\}$ specified by the following operation tables and the Boolean function $f(x, y) = ax + by$ where a and b are two of the elements in the Boolean algebra. Write $f(x, y)$ in a sum-of-minterms form.
- (c) Write the Boolean function of part (b) in a product-of-maxterms form.
- (d) Give a table of combinations for the Boolean function of Part (b). (*Note:* The table of combinations has 16 rows, not just 4.)
- (e) Which four rows of the table of combinations completely specify the function of Part (b)? Verify your answer.

	'	+	0	1	a	b	•	0	1	a	b
0	1	0	0	1	a	b	0	0	0	0	0
1	0	1	1	1	1	1	1	0	1	a	b
a	b	a	a	1	a	1	a	0	a	a	0
b	a	b	b	1	1	b	b	0	b	0	b

- 4.42** (a) If m_1 and m_2 are minterms of n variables, prove that $m_1 + m_2 = m_1 \oplus m_2$.
- (b) Prove that any switching function can be written as the exclusive-OR sum of products where each product does not contain a complemented literal. [*Hint:* Start with the function written as a sum of minterms and use Part (a).]

Karnaugh Maps

Objectives

1. Given a function (completely or incompletely specified) of three to five variables, plot it on a Karnaugh map. The function may be given in minterm, maxterm, or algebraic form.
2. Determine the essential prime implicants of a function from a map.
3. Obtain the minimum sum-of-products or minimum product-of-sums form of a function from the map.
4. Determine all of the prime implicants of a function from a map.
5. Understand the relation between operations performed using the map and the corresponding algebraic operations.

Study Guide

In this unit we will study the Karnaugh (pronounced “car-no”) map. Just about any type of algebraic manipulation we have done so far can be facilitated by using the map, provided the number of variables is small.

1. Study Section 5.1, *Minimum Forms of Switching Functions*.

(a) Define a minimum sum of products.

(b) Define a minimum product of sums.

2. Study Section 5.2, *Two- and Three-Variable Karnaugh Maps*.

(a) Plot the given truth table on the map. Then, loop two pairs of 1's on the map and write the simplified form of F .

P	Q	F
0	0	1
0	1	1
1	0	0
1	1	1

	0	1
0		
1		

Now simplify F algebraically and verify that your answer is correct.

(b) $F(a, b, c)$ is plotted below. Find the truth table for F .

	0	1
00	0	1
01	1	1
11	0	1
10	1	0

a	b	c	F
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

- (c) Plot the following functions on the given Karnaugh maps:

$$F_1(R, S, T) = \sum m(0, 1, 5, 6) \quad F_2(R, S, T) = \prod M(2, 3, 4, 7)$$

	0	1		0	1
00					
01					
11					
10					

Why are the two maps the same?

- (d) Plot the following function on the given map:

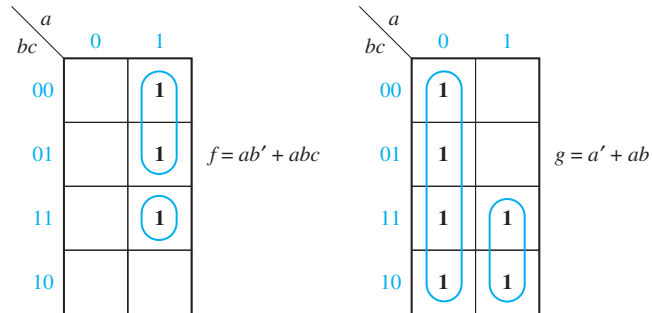
$$f(x, y, z) = z' + x'z + yz$$

Do *not* make a minterm expansion or a truth table before plotting.

	x	
yz	0	1
00		
01		
11		
10		

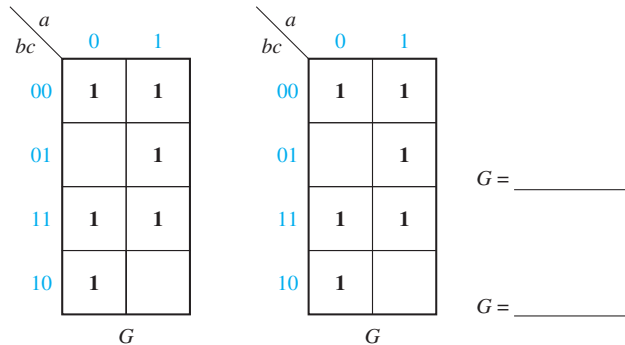
- (e) For a three-variable map, which squares are “adjacent” to square 2?
- (f) _____
What theorem is used when two terms in adjacent squares are combined?
- (g) What law of Boolean algebra justifies using a given 1 on a map in two or more loops?

(h) Each of the following solutions is *not* minimum.



In each case, change the looping on the map so that the minimum solution is obtained.

- (i) Work Problem 5.3.
 (j) Find two different minimum sum-of-products expressions for the function G , which is plotted below.



3. Study Section 5.3, *Four-Variable Karnaugh Maps*.

- (a) Note the locations of the minterms on three- and four-variable maps [Figures 5-3(b) and 5-10]. Memorize this ordering. This will save you a lot of time when you are plotting Karnaugh maps.

This ordering is valid only for the order of the variables given. If we label the maps as shown below, fill in the locations of the minterms:

	00	01	11	10		00	01	11	10
0						00			
1						01			
						11			
						10			

- (b) Given the following map, write the minterm and maxterm expansions for F in decimal form:

	00	01	11	10
00				
01				
11				
10				

- (c) Plot the following functions on the given maps:

$$(1) f(w, x, y, z) = \sum m(0, 1, 2, 5, 7, 8, 9, 10, 13, 14)$$

$$(2) f(w, x, y, z) = x'z' + y'z + w'xz + wyz'$$

	00	01	11	10		00	01	11	10
00						00			
01						01			
11						11			
10						10			

Your answers to (1) and (2) should be the same.

- (d) For a four-variable map, which squares are adjacent to square 14? _____

To square 8? _____

- (e) When we combine two adjacent 1's on a map, this corresponds to applying the theorem $xy' + xy = x$ to eliminate the variable in which the two terms differ. Thus, looping the two 1's as indicated on the following map is equivalent to combining the corresponding minterms algebraically:

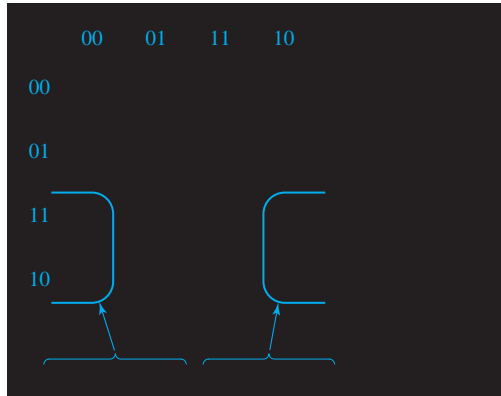
ab cd	00	01	11	10
00			1	
01	1			1
11	1	1		
10			1	

$a'b'c'd + ab'c'd = b'c'd$

[The term $b'c'd$ can be read directly from the map because it spans the first and last columns (b') and because it is in the second row ($c'd$).]

Loop two other pairs of adjacent 1's on this map and state the algebraic equivalent of looping these terms. Now read the loops directly off the map and check your algebra.

- (f) When we combine four adjacent 1's on a map (either four in a line or four in a square) this is equivalent to applying $xy + xy' = x$ three times:



Loop the other four 1's on the map and state the algebraic equivalent.

- (g) For each of the following maps, loop a minimum number of terms which will cover all of the 1's.

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00		1	1	
	01		1	1	1
	11	1	1		
	10		1		

f_1

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00				1
	01		1		
	11	1	1	1	1
	10	1			1

f_2

(For each part you should have looped two groups of four 1's and two groups of two 1's).

Write down the minimum sum-of-products expression for f_1 and f_2 from these maps.

$f_1 =$ _____

$f_2 =$ _____

- (h) Why is it not possible to combine three or six minterms together rather than just two, four, eight, etc.?

- (i) Note the procedure for deriving the minimum *product of sums* from the map. You will probably make fewer mistakes if you write down f' as a sum of products first and then complement it, as illustrated by the example in Figure 5-14.
- (j) Work Problems 5.4 and 5.5.
4. Study Section 5.4, *Determination of Minimum Expressions Using Essential Prime Implicants*.
- (a) For the map of Figure 5-15, list three implicants of F other than those which are labeled.

For the same map, is $ac'd'$ a prime implicant of F ?
Why or why not?

- (b) For the given map, are any of the circled terms prime implicants?
Why or why not?

AB \ CD	00	01	11	10
00				1
01		1	1	1
11		1	1	
10		1	1	

5. Study Figure 5-18 carefully and then answer the following questions for the given map:

- (a) How many 1's are adjacent to m_0 ?
- (b) Are all these 1's covered by a single prime implicant?
- (c) From your answer to (b), can you determine whether $B'C'$ is essential?
- (d) How many 1's are adjacent to m_9 ?
- (e) Are all of these 1's covered by a single prime implicant?
- (f) From your answer to (e), is $B'C'$ essential?
- (g) How many 1's are adjacent to m_7 ?
- (h) Why is $A'C$ essential?
- (i) Find two other essential prime implicants and tell which minterm makes them essential.

AB \ CD	00	01	11	10
00	1 ₀	1 ₄		1 ₈
01	1 ₁			1 ₉
11	1 ₃	1 ₇		
10	1 ₂	1 ₆		1 ₁₀

6. (a) How do you determine if a prime implicant is essential using a Karnaugh map?

- (b) For the following map, why is $A'B'$ not essential?

Why is BD' essential?

Is $A'D'$ essential? Why?

Is BC' essential? Why?

Is $B'CD$ essential? Why?

Find the minimum sum of products.

CD \ AB				
	00	01	11	10
00	1	1	1	
01	1	1	1	
11	1			1
10	1	1	1	

- (c) Work Programmed Exercise 5.1.
 (d) List all 1's and X's that are adjacent to 1_0 .

	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Why is $A'C'$ an essential prime implicant?

List all 1's and X's adjacent to 1_{15} .

Based on this list, why can you not find an essential prime implicant that covers 1_{15} ?

Does this mean that there is no essential prime implicant that covers 1_{15} ?

What essential prime implicant covers 1_{11} ?

Can you find an essential prime implicant that covers 1_{12} ? Explain.

Find two prime implicants that cover 1_{12} .

Give two minimum expressions for F .

- (e) Work Problem 5.6.
- (f) If you have a copy of the *LogicAid* program available, use the Karnaugh map tutorial mode to help you learn to find minimum solutions from Karnaugh maps. This program will check your work at each step to make sure that you loop the terms in the correct order. It also will check your final answer. Work Problem 5.7 using the Karnaugh map tutor.
7. (a) In Example 4, page 103, we derived the following function:

$$Z = \sum m(0, 3, 6, 9) + \sum d(10, 11, 12, 13, 14, 15)$$

Plot Z on the given map using X's to represent don't-care terms.



- (b) Show that the minimum sum of products is

$$Z = A'B'C'D' + B'CD + AD + BCD'$$

Which *four* don't-care minterms were assigned the value 1 when forming your solution?

- (c) Show that the minimum product of sums for Z is

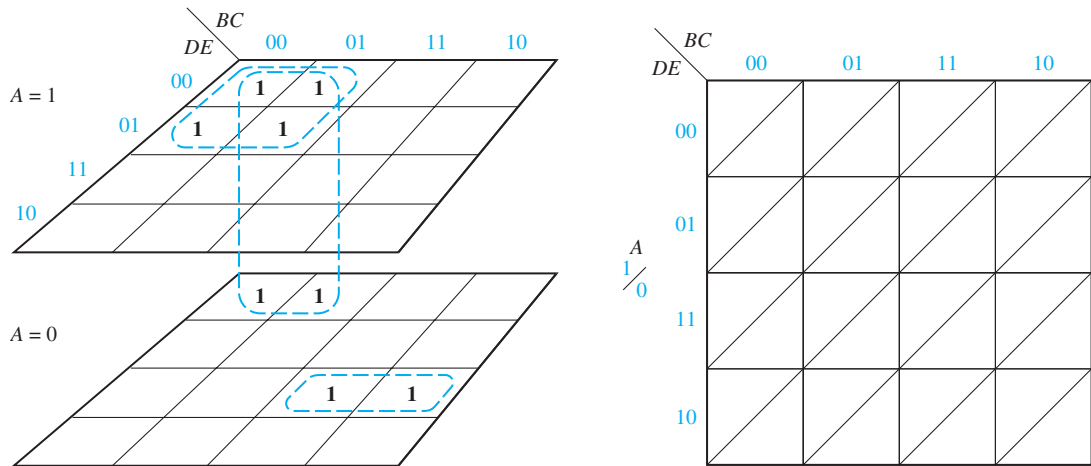
$$Z = (B' + C)(B' + D')(A' + D)(A + C + D')(B + C' + D)$$

Which *one* don't-care term of Z was assigned the value 1 when forming your solution?

- (d) Work Problem 5.8.

8. Study Section 5.5, *Five-Variable Karnaugh Maps*.

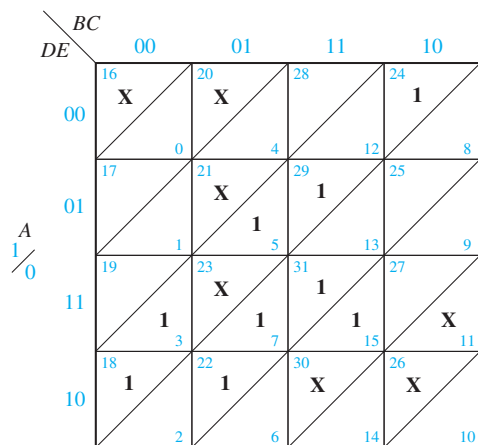
- (a) The figure below shows a three-dimensional five-variable map. Plot the 1's and loops on the corresponding two-dimensional map, and give the minimum sum-of-products expression for the function.



$F =$ _____

- (b) On a five-variable map (Figure 5-21), what are the five minterms adjacent to minterm 24?
- (c) Work through all of the examples in this section carefully and make sure that you understand all of the steps.
- (d) Two minimum solutions are given for Figure 5-24. There is a third minimum sum-of-products solution. What is it?
- (e) Work Programmed Exercise 5.2.

(f)



Find the three 1's and X's adjacent to 1_{18} . Can these all be looped with a single loop?

Find the 1's and X's adjacent to 1_{24} . Loop the essential prime implicant that covers 1_{24} .

Find the 1's and X's adjacent to 1_3 . Loop the essential prime implicant that covers 1_3 .

Can you find an essential prime implicant that covers 1_{22} ? Explain.

Find and loop two more essential prime implicants.

Find three ways to cover the remaining 1 on the map and give the corresponding minimum solutions.

- (g) If you have the *LogicAid* program available, work Problem 5.9, using the Karnaugh map tutor.
9. Study Section 5.6, *Other Uses of Karnaugh Maps*. Refer to Figure 5-8 and note that a consensus term exists if there are two adjacent, but nonoverlapping prime implicants. Observe how this principle is applied in Figure 5-26.
 10. Work Problems 5.10, 5.11, 5.12, and 5.13 When deriving the minimum solution from the map, always write down the *essential* prime implicants first. If you do not, it is quite likely that you will not get the minimum solution. In addition, make sure you can find *all* of the prime implicants from the map [see Problem 5.10(b)].
 11. Review the objectives and take the readiness test.



Karnaugh Maps

Switching functions can generally be simplified by using the algebraic techniques described in Unit 3. However, two problems arise when algebraic procedures are used:

1. The procedures are difficult to apply in a systematic way.
2. It is difficult to tell when you have arrived at a minimum solution.

The Karnaugh map method studied in this unit and the Quine-McCluskey procedure studied in Unit 6 overcome these difficulties by providing systematic methods for simplifying switching functions. The Karnaugh map is an especially useful tool for simplifying and manipulating switching functions of three or four variables, but it can be extended to functions of five or more variables. Generally, you will find the Karnaugh map method is faster and easier to apply than other simplification methods.

5.1 Minimum Forms of Switching Functions

When a function is realized using AND and OR gates, the cost of realizing the function is directly related to the number of gates and gate inputs used. The Karnaugh map techniques developed in this unit lead directly to minimum cost two-level circuits composed of AND and OR gates. An expression consisting of a sum of product terms corresponds directly to a two-level circuit composed of a group of AND gates feeding a single OR gate (see Figure 2-5). Similarly, a product-of-sums expression corresponds to a two-level circuit composed of OR gates feeding a single AND gate (see Figure 2-6). Therefore, to find minimum cost two-level AND-OR gate circuits, we must find minimum expressions in sum-of-products or product-of-sums form.

A *minimum sum-of-products* expression for a function is defined as a sum of product terms which (a) has a minimum number of terms and (b) of all those expressions which have the same minimum number of terms, has a minimum number of literals. The minimum sum of products corresponds directly to a minimum two-level gate circuit which has (a) a minimum number of gates and (b) a minimum

number of gate inputs. Unlike the minterm expansion for a function, the minimum sum of products is not necessarily unique; that is, a given function may have two different minimum sum-of-products forms, each with the same number of terms and the same number of literals. Given a minterm expansion, the minimum sum-of-products form can often be obtained by the following procedure:

1. Combine terms by using $XY' + XY = X$. Do this repeatedly to eliminate as many literals as possible. A given term may be used more than once because $X + X = X$.
2. Eliminate redundant terms by using the consensus theorem or other theorems.

Unfortunately, the result of this procedure may depend on the order in which terms are combined or eliminated so that the final expression obtained is not necessarily minimum.

Example

Find a minimum sum-of-products expression for

$$\begin{aligned}
 F(a, b, c) &= \Sigma m(0, 1, 2, 5, 6, 7) \\
 F &= a'b'c' + a'b'c + a'bc' + ab'c + abc' + abc \\
 &= \underbrace{a'b'c' + a'b'c}_{a'b'} + \underbrace{a'bc' + ab'c}_{b'c} + \underbrace{abc' + abc}_{bc'} + \underbrace{ab'c + abc}_{ab}
 \end{aligned} \tag{5-1}$$

None of the terms in the above expression can be eliminated by consensus. However, combining terms in a different way leads directly to a minimum sum of products:

$$\begin{aligned}
 F &= a'b'c' + a'b'c + a'bc' + ab'c + abc' + abc \\
 &= \underbrace{a'b'c' + a'b'c}_{a'b'} + \underbrace{a'bc' + ab'c}_{bc'} + \underbrace{abc' + abc}_{ac}
 \end{aligned} \tag{5-2}$$

A *minimum product-of-sums* expression for a function is defined as a product of sum terms which (a) has a minimum number of factors, and (b) of all those expressions which have the same number of factors, has a minimum number of literals. Unlike the maxterm expansion, the minimum product-of-sums form of a function is not necessarily unique. Given a maxterm expansion, the minimum product of sums can often be obtained by a procedure similar to that used in the minimum sum-of-products case, except that the theorem $(X + Y)(X + Y') = X$ is used to combine terms.

Example

$$\begin{aligned}
 &(A + B' + C + D')(A + B' + C' + D')(A + B' + C' + D)(A' + B' + C' + D)(A + B + C' + D)(A' + B + C' + D) \\
 &= (A + B' + D') \quad (A + B' + C') \quad (B' + C' + D) \quad (B + C' + D) \\
 &= (A + B' + D') \quad \underbrace{(A + B' + C')}_{\text{eliminate by consensus}} \quad (C' + D) \\
 &= (A + B' + D')(C' + D)
 \end{aligned} \tag{5-3}$$

5.2 Two- and Three-Variable Karnaugh Maps

Just like a truth table, the Karnaugh map of a function specifies the value of the function for every combination of values of the independent variables. A two-variable Karnaugh map is shown. The values of one variable are listed across the top of the map, and the values of the other variable are listed on the left side. Each square of the map corresponds to a pair of values for A and B as indicated.

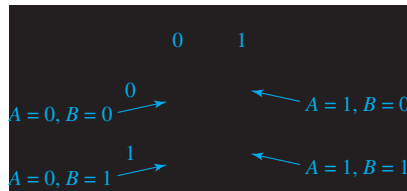


Figure 5-1 shows the truth table for a function F and the corresponding Karnaugh map. Note that the value of F for $A = B = 0$ is plotted in the upper left square, and the other map entries are plotted in a similar way in Figure 5-1(b). Each 1 on the map corresponds to a minterm of F . We can read the minterms from the map just like we can read them from the truth table. A 1 in square 00 of Figure 5-1(c) indicates that $A'B'$ is a minterm of F . Similarly, a 1 in square 01 indicates that $A'B$ is a minterm. Minterms in adjacent squares of the map can be combined since they differ in only one variable. Thus, $A'B'$ and $A'B$ combine to form A' , and this is indicated by looping the corresponding 1's on the map in Figure 5-1(d).

FIGURE 5-1

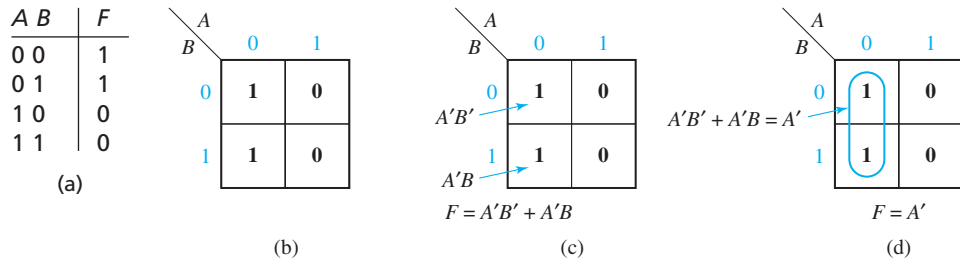


Figure 5-2 shows a three-variable truth table and the corresponding Karnaugh map (see Figure 5-27 for an alternative way of labeling maps). The value of one variable (A) is listed across the top of the map, and the values of the other two variables (B , C) are listed along the side of the map. The rows are labeled in the sequence 00, 01, 11, 10 so that values in adjacent rows differ in only one variable. For each combination of values of the variables, the value of F is read from the truth table and plotted in the appropriate map square. For example, for the input combination $ABC = 001$, the value $F = 0$ is plotted in the square for which $A = 0$ and $BC = 01$. For the combination $ABC = 110$, $F = 1$ is plotted in the $A = 1$, $BC = 10$ square.

FIGURE 5-2
Truth Table and
Karnaugh Map for
Three-Variable
Function

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

(a)

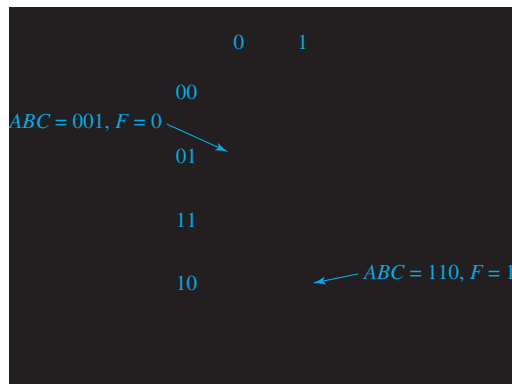
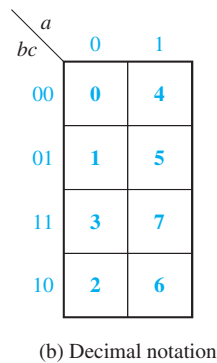
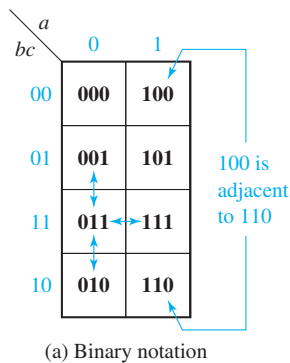


Figure 5-3 shows the location of the minterms on a three-variable map. Minterms in adjacent squares of the map differ in only one variable and therefore can be combined using the theorem $XY' + XY = X$. For example, minterm 011 ($a'bc$) is adjacent to the three minterms with which it can be combined—001 ($a'b'c$), 010 ($a'bc'$), and 111 (abc). In addition to squares which are physically adjacent, the top and bottom rows of the map are defined to be adjacent because the corresponding minterms in these rows differ in only one variable. Thus 000 and 010 are adjacent, and so are 100 and 110.

FIGURE 5-3
Location of
Minterms on
a Three-Variable
Karnaugh Map



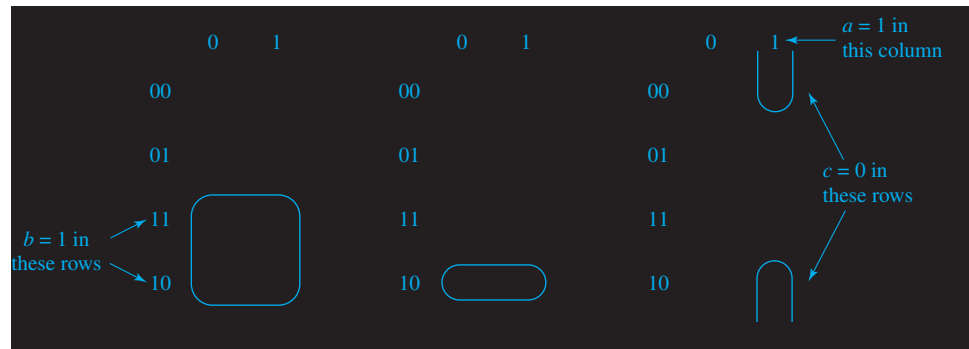
Given the minterm expansion of a function, it can be plotted on a map by placing 1's in the squares which correspond to minterms of the function and 0's in the remaining squares (the 0's may be omitted if desired). Figure 5-4 shows the plot of $F(a, b, c) = m_1 + m_3 + m_5$. If F is given as a maxterm expansion, the map is plotted by placing 0's in the squares which correspond to the maxterms and then by filling in the remaining squares with 1's. Thus, $F(a, b, c) = M_0 M_2 M_4 M_6 M_7$ gives the same map as Figure 5-4.

FIGURE 5-4
Karnaugh Map of
 $F(a, b, c) =$
 $\Sigma m(1, 3, 5) =$
 $\Pi M(0, 2, 4, 6, 7)$

	0	1
00	0	4
01	1	5
11	3	7
10	2	6

Figure 5-5 illustrates how product terms can be plotted on Karnaugh maps. To plot the term b , 1's are entered in the four squares of the map where $b = 1$. The term bc' is 1 when $b = 1$ and $c = 0$, so 1's are entered in the two squares in the $bc = 10$ row. The term ac' is 1 when $a = 1$ and $c = 0$, so 1's are entered in the $a = 1$ column in the rows where $c = 0$.

FIGURE 5-5
Karnaugh Maps for
Product Terms



If a function is given in algebraic form, it is unnecessary to expand it to minterm form before plotting it on a map. If the algebraic expression is converted to sum-of-products form, then each product term can be plotted directly as a group of 1's on the map. For example, given that

$$f(a, b, c) = abc' + b'c + a'$$

we would plot the map as follows:

1. The term abc' is 1 when $a = 1$ and $bc = 10$, so we place a 1 in the square which corresponds to the $a = 1$ column and the $bc = 10$ row of the map.
2. The term $b'c$ is 1 when $bc = 01$, so we place 1's in both squares of the $bc = 01$ row of the map.
3. The term a' is 1 when $a = 0$, so we place 1's in all the squares of the $a = 0$ column of the map. (Note: Since there already is a 1 in the $abc = 001$ square, we do not have to place a second 1 there because $x + x = x$.)

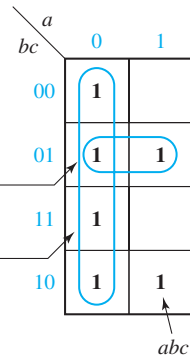
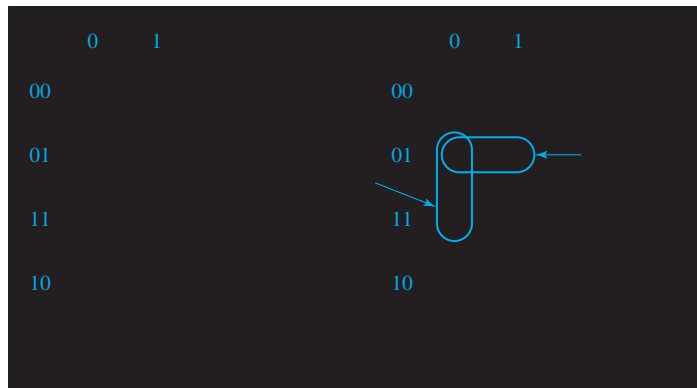


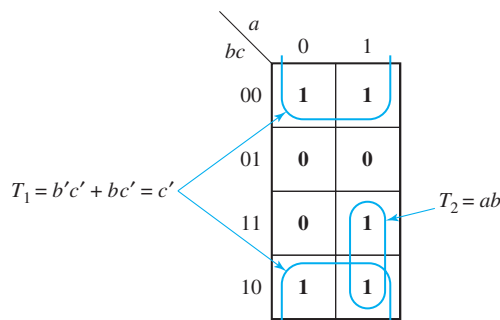
Figure 5-6 illustrates how a simplified expression for a function can be derived using a Karnaugh map. The function to be simplified is first plotted on a Karnaugh map in Figure 5-6(a). Terms in adjacent squares on the map differ in only one variable and can be combined using the theorem $XY' + XY = X$. Thus $a'b'c$ and $a'bc$ combine to form $a'c$, and $a'b'c$ and $ab'c$ combine to form $b'c$, as shown in Figure 5-6(b). A loop around a group of minterms indicates that these terms have been combined. The looped terms can be read directly off the map. Thus, for Figure 5-6(b), term T_1 is in the $a = 0$ (a') column, and it spans the rows where $c = 1$, so $T_1 = a'c$. Note that b has been eliminated because the two minterms in T_1 differ in the variable b . Similarly, the term T_2 is in the $bc = 01$ row so $T_2 = b'c$, and a has been eliminated because T_2 spans the $a = 0$ and $a = 1$ columns. Thus, the minimum sum-of-products form for F is $a'c + b'c$.

FIGURE 5-6
Simplification of a
Three-Variable
Function



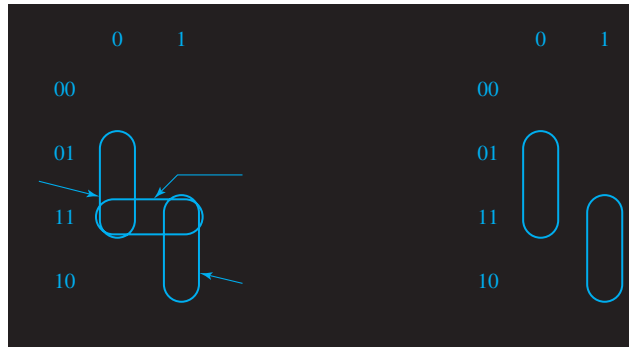
The map for the complement of F (Figure 5-7) is formed by replacing 0's with 1's and 1's with 0's on the map of F . To simplify F' , note that the terms in the top row combine to form $b'c'$, and the terms in the bottom row combine to form bc' . Because $b'c'$ and bc' differ in only one variable, the top and bottom rows can then be combined to form a group of four 1's, thus eliminating two variables and leaving $T_1 = c'$. The remaining 1 combines, as shown, to form $T_2 = ab$, so the minimum sum-of-products form for F' is $c' + ab$.

FIGURE 5-7
Complement of
Map in Figure
5.6(a)



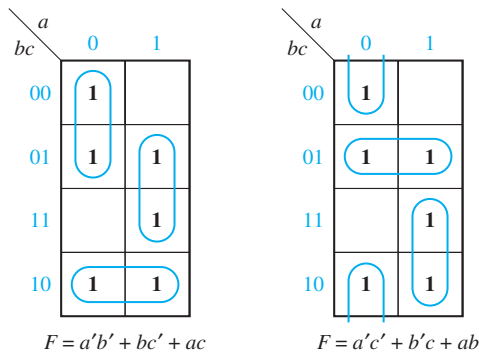
The Karnaugh map can also illustrate the basic theorems of Boolean algebra. Figure 5-8 illustrates the consensus theorem, $XY + X'Z + YZ = XY + X'Z$. Note that the consensus term (YZ) is redundant because its 1's are covered by the other two terms.

FIGURE 5-8
Karnaugh Maps
that Illustrate the
Consensus Theorem



If a function has two or more minimum sum-of-products forms, all of these forms can be determined from a map. Figure 5-9 shows the two minimum solutions for $F = \Sigma m(0, 1, 2, 5, 6, 7)$.

FIGURE 5-9
Function with Two
Minimum Forms



5.3 Four-Variable Karnaugh Maps

Figure 5-10 shows the location of minterms on a four-variable map. Each minterm is located adjacent to the four terms with which it can combine. For example, m_5 (0101) could combine with m_1 (0001), m_4 (0100), m_7 (0111), or m_{13} (1101) because it differs in only one variable from each of the other minterms. The definition of adjacent squares must be extended so that not only are top and bottom rows adjacent as in the three-variable map, but the first and last columns are also adjacent. This requires numbering the columns in the sequence 00, 01, 11, 10 so that minterms 0 and 8, 1 and 9, etc., are in adjacent squares.

FIGURE 5-10
Location
of Minterms on
Four-Variable
Karnaugh Map

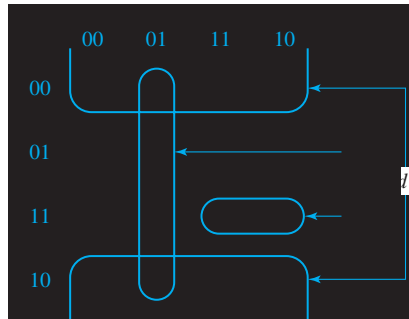
	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

We will now plot the following four-variable expression on a Karnaugh map (Figure 5-11):

$$f(a, b, c, d) = acd + a'b + d'$$

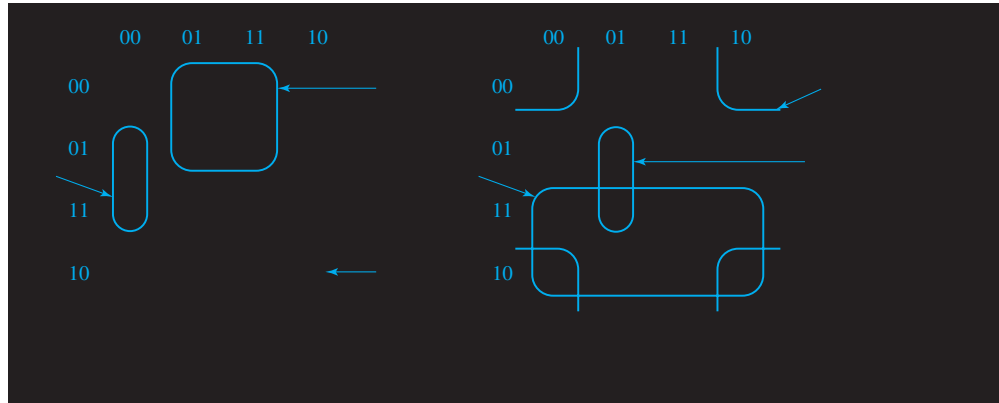
The first term is 1 when $a = c = d = 1$, so we place 1's in the two squares which are in the $a = 1$ column and $cd = 11$ row. The term $a'b$ is 1 when $ab = 01$, so we place four 1's in the $ab = 01$ column. Finally, d' is 1 when $d = 0$, so we place eight 1's in the two rows for which $d = 0$. (Duplicate 1's are not plotted because $1 + 1 = 1$.)

FIGURE 5-11
Plot of
 $acd + a'b + d'$



Next, we will simplify the functions f_1 and f_2 given in Figure 5-12. Because the functions are specified in minterm form, we can determine the locations of the 1's on the map by referring to Figure 5-10. After plotting the maps, we can then combine adjacent groups of 1's. Minterms can be combined in groups of two, four, or eight to eliminate one, two, or three variables, respectively. In Figure 5-12(a), the pair of 1's in the $ab = 00$ column and also in the $d = 1$ rows represents $a'b'd$. The group of four 1's in the $b = 1$ columns and $c = 0$ rows represents bc' .

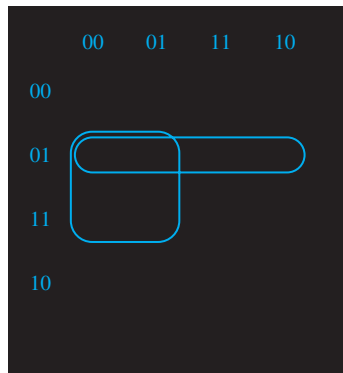
FIGURE 5-12
Simplification of
Four-Variable
Functions



In Figure 5-12(b), note that the four corner 1's span the $b = 0$ columns and $d = 0$ rows and, therefore, can be combined to form the term $b'd'$. The group of eight 1's covers both rows where $c = 1$ and, therefore, represents the term c . The pair of 1's which is looped on the map represents the term $a'bd$ because it is in the $ab = 01$ column and spans the $d = 1$ rows.

The Karnaugh map method is easily extended to functions with don't-care terms. The required minterms are indicated by 1's on the map, and the don't-care minterms are indicated by X's. When choosing terms to form the minimum sum of products, all the 1's must be covered, but the X's are only used if they will simplify the resulting expression. In Figure 5-13, the only don't-care term used in forming the simplified expression is 13.

FIGURE 5-13
Simplification of
an Incompletely
Specified Function



The use of Karnaugh maps to find a minimum sum-of-products form for a function has been illustrated in Figures 5-1, 5-6, and 5-12. A minimum product of sums can also be obtained from the map. Because the 0's of f are 1's of f' , the minimum sum of products for f' can be determined by looping the 0's on a map of f . The complement of the minimum sum of products for f' is then

the minimum product of sums for f . The following example illustrates this procedure for

$$f = x'z' + wyz + w'y'z' + x'y$$

First, the 1's of f are plotted in Figure 5-14. Then, from the 0's,

$$f' = y'z + wxz' + w'xy$$

and the minimum product of sums for f is

$$f = (y + z')(w' + x' + z)(w + x' + y')$$

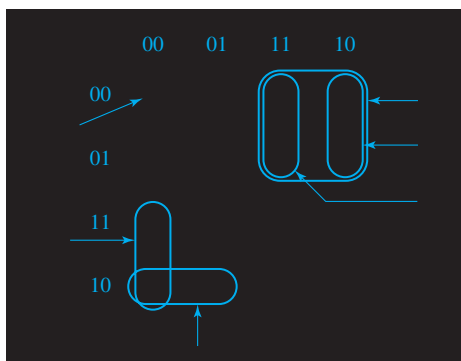
FIGURE 5-14

		wx			
		00	01	11	10
yz	00	1	1	0	1
	01	0	0	0	0
	11	1	0	1	1
	10	1	0	0	1

5.4 Determination of Minimum Expressions Using Essential Prime Implicants

Any single 1 or any group of 1's which can be combined together on a map of the function F represents a product term which is called an *implicant* of F (see Section 6.1 for a formal definition of implicant and prime implicant). Several implicants of F are indicated in Figure 5-15. A product term implicant is called a *prime implicant* if it cannot be combined with another term to eliminate a variable. In Figure 5-15,

FIGURE 5-15

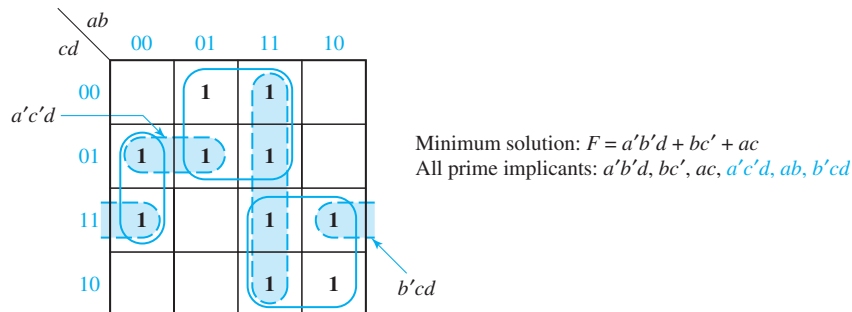


$a'b'c$, $a'cd'$, and ac' are prime implicants because they cannot be combined with other terms to eliminate a variable. On the other hand, $a'b'c'd'$ is not a prime implicant because it can be combined with $a'b'cd'$ or $ab'c'd'$. Neither abc' , nor $ab'c'$ is a prime implicant because these terms can be combined together to form ac' .

All of the prime implicants of a function can be obtained from a Karnaugh map. A single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's on a map form a prime implicant if they are not contained in a group of four 1's; four adjacent 1's form a prime implicant if they are not contained in a group of eight 1's, etc.

The minimum sum-of-products expression for a function consists of some (but not necessarily all) of the prime implicants of a function. In other words, a sum-of-products expression containing a term which is not a prime implicant cannot be minimum. This is true because if a nonprime term were present, the expression could be simplified by combining the nonprime term with additional minterms. In order to find the minimum sum of products from a map, we must find a minimum number of prime implicants which cover all of the 1's on the map. The function plotted in Figure 5-16 has six prime implicants. Three of these prime implicants cover all of the 1's on the map, and the minimum solution is the sum of these three prime implicants. The shaded loops represent prime implicants which are not part of the minimum solution.

FIGURE 5-16
Determination of
All Prime Implicants

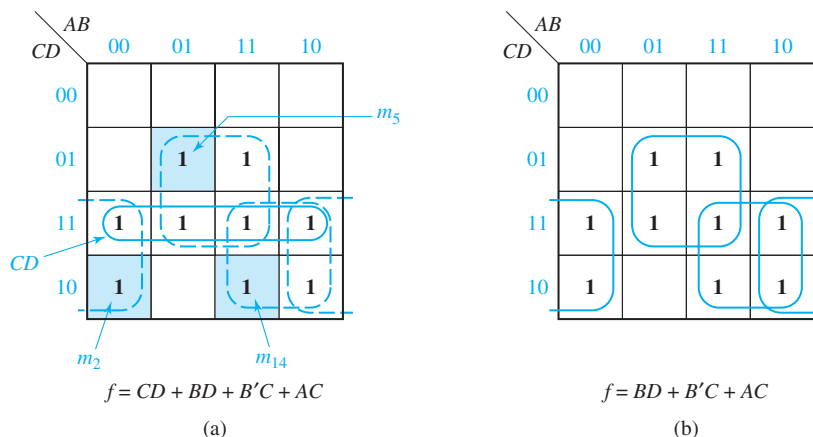


When writing down a list of *all* of the prime implicants from the map, note that there are often prime implicants which are not included in the minimum sum of products. Even though all of the 1's in a term have already been covered by prime implicants, that term may still be a prime implicant provided that it is not included in a larger group of 1's. For example, in Figure 5-16, $a'c'd$ is a prime implicant because it cannot be combined with other 1's to eliminate another variable. However, abd is not a prime implicant because it can be combined with two other 1's to form ab . The term $b'cd$ is also a prime implicant even though both of its 1's are already covered by other prime implicants. In the process of finding prime implicants, don't-cares are treated just like 1's. However, a prime implicant composed entirely of don't-cares can never be part of the minimum solution.

Because all of the prime implicants of a function are generally not needed in forming the minimum sum of products, a systematic procedure for selecting prime

implicants is needed. If prime implicants are selected from the map in the wrong order, a nonminimum solution may result. For example, in Figure 5-17, if CD is chosen first, then BD , $B'C$, and AC are needed to cover the remaining 1's, and the solution contains four terms. However, if the prime implicants indicated in Figure 5-17(b) are chosen first, all 1's are covered and CD is not needed.

FIGURE 5-17

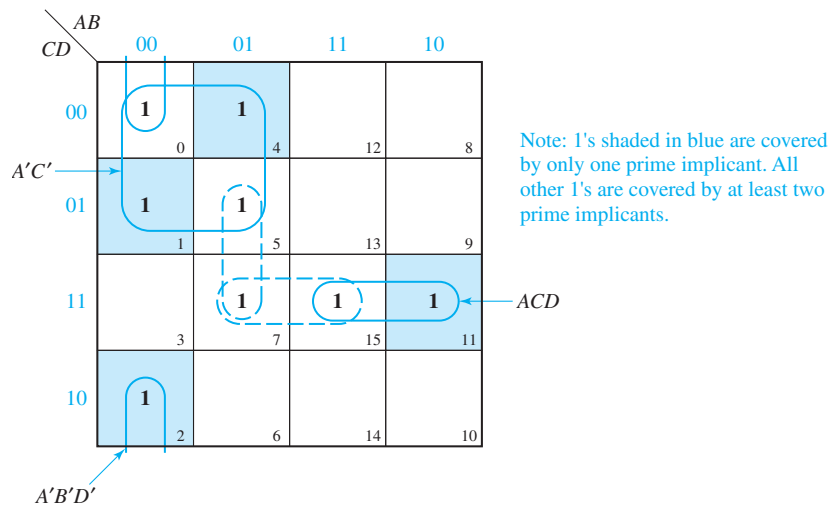


Note that some of the minterms on the map of Figure 5-17(a) can be covered by only a single prime implicant, but other minterms can be covered by two different prime implicants. For example, m_2 is covered only by $B'C$, but m_3 is covered by both $B'C$ and CD . If a minterm is covered by only one prime implicant, that prime implicant is said to be *essential*, and it must be included in the minimum sum of products. Thus, $B'C$ is an essential prime implicant because m_2 is not covered by any other prime implicant. However, CD is *not* essential because each of the 1's in CD can be covered by another prime implicant. The only prime implicant which covers m_5 is BD , so BD is essential. Similarly, AC is essential because no other prime implicant covers m_{14} . In this example, if we choose all of the essential prime implicants, all of the 1's on the map are covered and the nonessential prime implicant CD is not needed.

In general, in order to find a minimum sum of products from a map, we should first loop all of the essential prime implicants. One way of finding essential prime implicants on a map is simply to look at each 1 on the map that has not already been covered, and check to see how many prime implicants cover that 1. If there is only one prime implicant which covers the 1, that prime implicant is essential. If there are two or more prime implicants which cover the 1, we cannot say whether these prime implicants are essential or not without checking the other minterms. For simple problems, we can locate the essential prime implicants in this way by inspection of each 1 on the map. For example, in Figure 5-16, m_4 is covered only by the prime implicant bc' , and m_{10} is covered only by the prime implicant ac . All other 1's on the map are covered by two prime implicants; therefore, the only essential prime implicants are bc' and ac .

For more complicated maps, and especially for maps with five or more variables, we need a more systematic approach for finding the essential prime implicants. When checking a minterm to see if it is covered by only one prime implicant, we must look at all squares adjacent to that minterm. If the given minterm and all of the 1's adjacent to it are covered by a single term, then that term is an *essential* prime implicant.¹ If all of the 1's adjacent to a given minterm are *not* covered by a single term, then there are two or more prime implicants which cover that minterm, and we cannot say whether these prime implicants are essential or not without checking the other minterms. Figure 5-18 illustrates this principle.

FIGURE 5-18

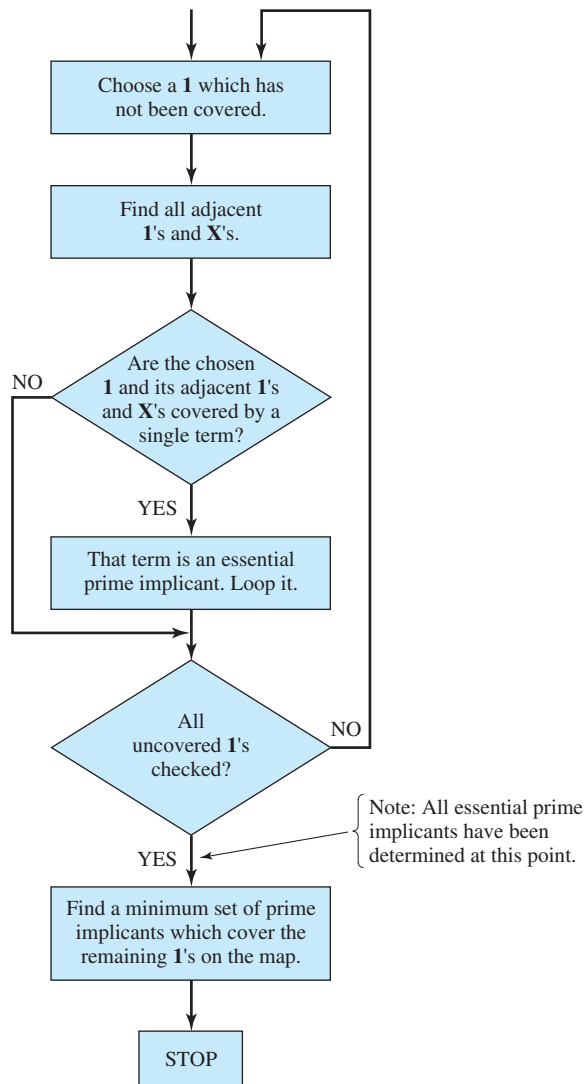


The adjacent 1's for minterm m_0 (l_0) are l_1 , l_2 , and l_4 . Because no single term covers these four 1's, no essential prime implicant is yet apparent. The adjacent 1's for l_1 are l_0 and l_5 , so the term which covers these three 1's ($A'C'$) is an essential prime implicant. Because the only 1 adjacent to l_2 is l_0 , $A'B'D'$ is also essential. Because the 1's adjacent to l_7 (l_5 and l_{15}) are not covered by a single term, neither $A'BD$ nor BCD is essential at this point. However, because the only 1 adjacent to l_{11} is l_{15} , ACD is essential. To complete the minimum solution, one of the nonessential prime implicants is needed. Either $A'BD$ or BCD may be selected. The final solution is

$$A'C' + A'B'D' + ACD + \left\{ \begin{array}{c} A'BD \\ \text{or} \\ BCD \end{array} \right\}$$

¹ This statement is proved in Appendix D.

FIGURE 5-19
Flowchart for
Determining a
Minimum Sum of
Products Using a
Karnaugh Map



If a don't-care minterm is present on the map, we do not have to check it to see if it is covered by one or more prime implicants. However, when checking a 1 for adjacent 1's, we treat the adjacent don't-cares as if they were 1's because don't-cares may be combined with 1's in the process of forming prime implicants. The following procedure can then be used to obtain a minimum sum of products from a Karnaugh map:

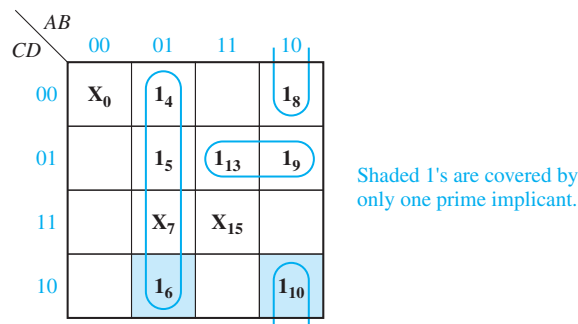
1. Choose a minterm (a 1) which has not yet been covered.
2. Find all 1's and X's adjacent to that minterm. (Check the n adjacent squares on an n -variable map.)
3. If a single term covers the minterm and all of the adjacent 1's and X's, then that term is an essential prime implicant, so select that term. (Note that don't-care terms are treated like 1's in steps 2 and 3 but not in step 1.)

4. Repeat steps 1, 2, and 3 until all essential prime implicants have been chosen.
5. Find a minimum set of prime implicants which cover the remaining 1's on the map. (If there is more than one such set, choose a set with a minimum number of literals.)

Figure 5-19 gives a flowchart for this procedure. The following example (Figure 5-20) illustrates the procedure. Starting with 1_4 , we see that the adjacent 1's and X's (X_0 , 1_5 , and 1_6) are *not* covered by a single term, so no essential prime implicant is apparent. However, 1_6 and its adjacent 1's and X's (1_4 and X_7) are covered by $A'B$, so $A'B$ is an essential prime implicant. Next, looking at 1_{13} , we see that its adjacent 1's and X's (1_5 , 1_9 , and X_{15}) are *not* covered by a single term, so no essential prime implicant is apparent. Similarly, an examination of the terms adjacent to 1_8 and 1_9 reveals no essential prime implicants. However, 1_{10} has only 1_8 adjacent to it, so $AB'D'$ is an essential prime implicant because it covers both 1_{10} and 1_8 . Having first selected the essential prime implicants, we now choose $AC'D$ because it covers both of the remaining 1's on the map.

Judicious selection of the order in which the minterms are selected (step 1) reduces the amount of work required in applying this procedure. As will be seen in the next section, this procedure is especially helpful in obtaining minimum solutions for five- and six-variable problems.

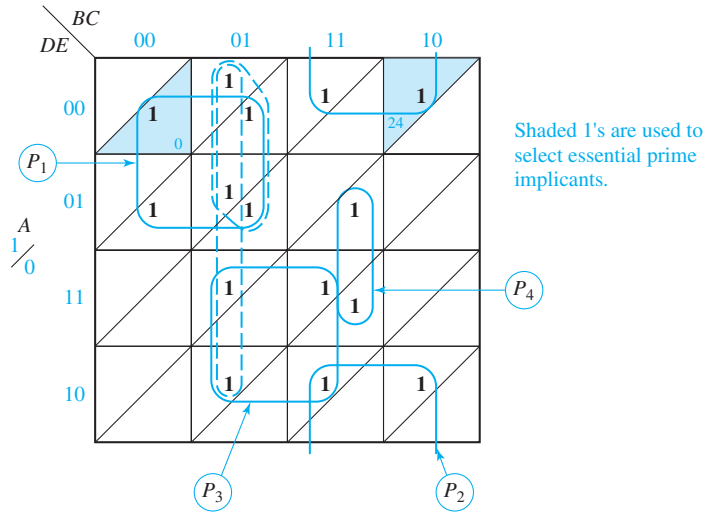
FIGURE 5-20



5.5 Five-Variable Karnaugh Maps

A five-variable map can be constructed in three dimensions by placing one four-variable map on top of a second one. Terms in the bottom layer are numbered 0 through 15 and corresponding terms in the top layer are numbered 16 through 31, so that terms in the bottom layer contain A' and those in the top layer contain A . To represent the map in two dimensions, we will divide each square in a four-variable map by a diagonal line and place terms in the bottom layer below the line and terms in the top layer above the line (Figure 5-21). Terms in the top or bottom layer combine just like terms on a four-variable map. In addition, two terms in the same square which are separated by a diagonal line differ in only one variable and can be combined.

FIGURE 5-23



Prime implicant P_1 is chosen first because all of the 1's adjacent to minterm 0 are covered by P_1 . Prime implicant P_2 is chosen next because all of the 1's adjacent to minterm 24 are covered by P_2 . All of the remaining 1's on the map can be covered by at least two different prime implicants, so we proceed by trial and error. After a few tries, it becomes apparent that the remaining 1's can be covered by three prime implicants. If we choose prime implicants P_3 and P_4 next, the remaining two 1's can be covered by two different groups of four. The resulting minimum solution is

$$F = \underbrace{A'B'D'}_{P_1} + \underbrace{ABE'}_{P_2} + \underbrace{ACD}_{P_3} + \underbrace{A'BCE}_{P_4} + \left\{ \begin{array}{c} AB'C \\ \text{or} \\ B'CD' \end{array} \right\}$$

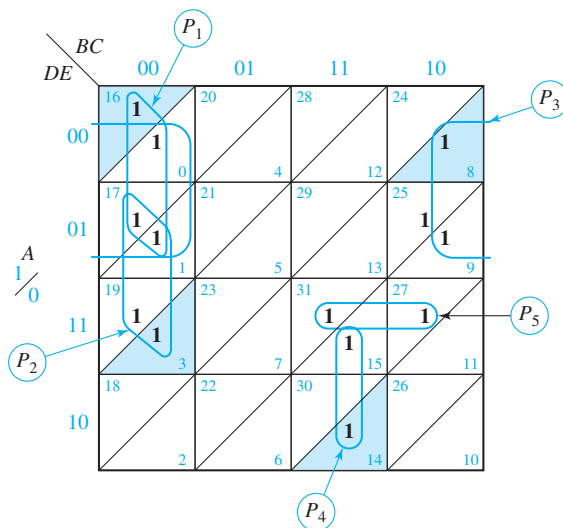
Figure 5-24 is a map of

$$F(A, B, C, D, E) = \Sigma m(0, 1, 3, 8, 9, 14, 15, 16, 17, 19, 25, 27, 31)$$

All 1's adjacent to m_{16} are covered by P_1 , so choose P_1 first. All 1's adjacent to m_3 are covered by P_2 , so P_2 is chosen next. All 1's adjacent to m_8 are covered by P_3 , so P_3 is chosen. Because m_{14} is only adjacent to m_{15} , P_4 is also essential. There are no more essential prime implicants, and the remaining 1's can be covered by two terms, P_5 and (1-9-17-25) or (17-19-25-27). The final solution is

$$F = \underbrace{B'C'D'}_{P_1} + \underbrace{B'C'E}_{P_2} + \underbrace{A'C'D'}_{P_3} + \underbrace{A'BCD}_{P_4} + \underbrace{ABDE}_{P_5} + \left\{ \begin{array}{c} C'D'E \\ \text{or} \\ AC'E \end{array} \right\}$$

FIGURE 5-24



5.6 Other Uses of Karnaugh Maps

Many operations that can be performed using a truth table or algebraically can be done using a Karnaugh map. A map conveys the same information as a truth table—it is just arranged in a different format. If we plot an expression for F on a map, we can read off the minterm and maxterm expansions for F and for F' . From the map of Figure 5-14, the minterm expansion of f is

$$f = \sum m(0, 2, 3, 4, 8, 10, 11, 15)$$

and because each 0 corresponds to a maxterm, the maxterm expansion of f is

$$f = \prod M(1, 5, 6, 7, 9, 12, 13, 14)$$

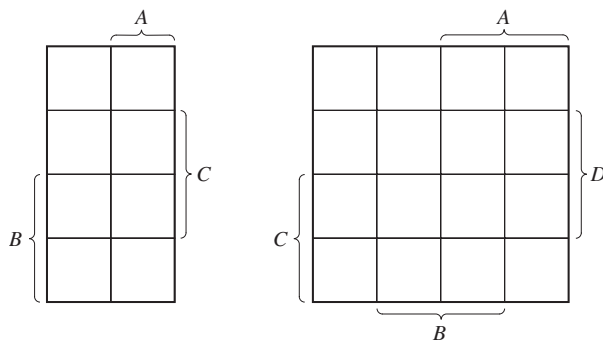
We can prove that two functions are equal by plotting them on maps and showing that they have the same Karnaugh map. We can perform the AND operation (or the OR operation) on two functions by ANDing (or ORing) the 1's and 0's which appear in corresponding positions on their maps. This procedure is valid because it is equivalent to doing the same operations on the truth tables for the functions.

A Karnaugh map can facilitate factoring an expression. Inspection of the map reveals terms which have one or more variables in common. For the map of Figure 5-25, the two terms in the first column have $A'B'$ in common; the two terms in the lower right corner have AC in common.

5.7 Other Forms of Karnaugh Maps

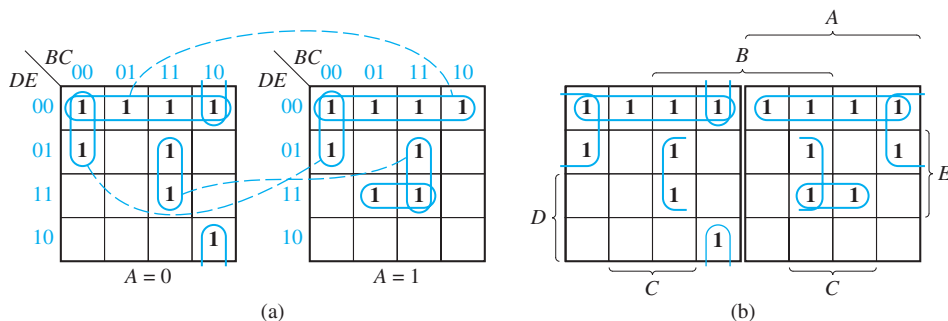
Instead of labeling the sides of a Karnaugh map with 0's and 1's, some people prefer to use the labeling shown in Figure 5-27. For the half of the map labeled A , $A = 1$; and for the other half, $A = 0$. The other variables have a similar interpretation. A map labeled this way is sometimes referred to as a Veitch diagram. It is particularly useful for plotting functions given in algebraic form rather than in minterm or maxterm form. However, when utilizing Karnaugh maps to solve sequential circuit problems (Units 12 through 16), the use of 0's and 1's to label the maps is more convenient.

FIGURE 5-27
Veitch Diagrams



Two alternative forms for five-variable maps are used. One form simply consists of two four-variable maps side-by-side as in Figure 5-28(a). A modification of this uses a *mirror image* map as in Figure 5-28(b). In this map, first and eighth columns are “adjacent” as are second and seventh columns, third and sixth columns, and fourth and fifth columns. The same function is plotted on both these maps.

FIGURE 5-28
Other Forms
of Five-Variable
Karnaugh Maps



$$F = D'E' + B'C'D' + BCE + A'BC'E' + ACDE$$

Programmed Exercise 5.1

Cover the answers to this exercise with a sheet of paper and slide it down as you check your answers. Write your answers in the space provided before looking at the correct answer.

Problem: Determine the minimum sum of products and minimum product of sums for

$$f = b'c'd' + bcd + acd' + a'b'c + a'bc'd$$

First, plot the map for f .

	00	01	11	10
00				
01				
11				
10				

Answer:

	00	01	11	10
00	1			1
01		1		
11	1	1	1	
10	1		1	1

- The minterms adjacent to m_0 on the preceding map are _____ and _____.
- Find an essential prime implicant containing m_0 and loop it.
- The minterms adjacent to m_3 are _____ and _____.
- Is there an essential prime implicant which contains m_3 ?
- Find the remaining essential prime implicant(s) and loop it (them).

Answers:

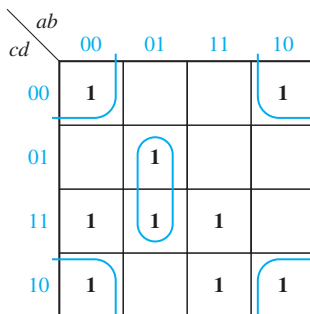
(a) m_2 and m_8

(b)

(c) m_2 and m_7

(e)

(d) No

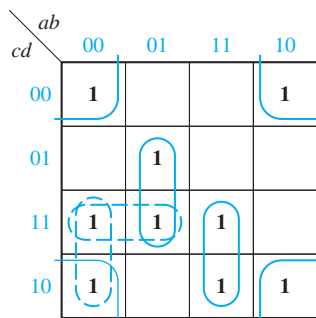


Loop the remaining 1's using a minimum number of loops.
The two possible minimum sum-of-products forms for f are

$f =$ _____ and

$f =$ _____

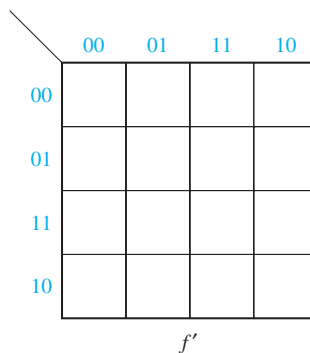
Answer:



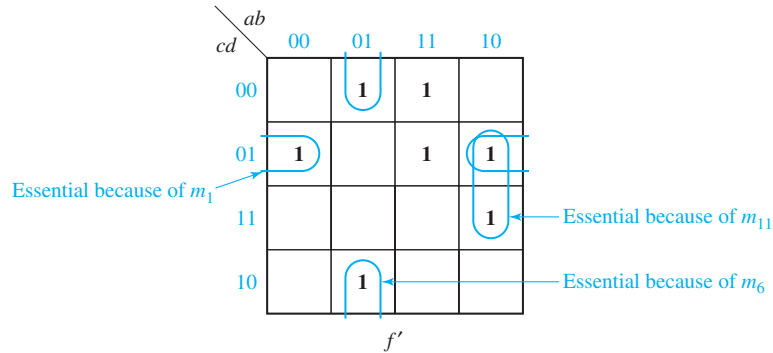
$$f = b'd' + a'bd + abc + \begin{cases} a'cd \\ \text{or} \\ a'b'c \end{cases}$$

Next, we will find the minimum product of sums for f . Start by plotting the map for f' .

Loop all essential prime implicants of f' and indicate which minterm makes each one essential.



Answer:



Loop the remaining 1's and write the minimum sum of products for f' .

$$f' = \underline{\hspace{2cm}}$$

The minimum product of sums for f is therefore

$$f = \underline{\hspace{2cm}}$$

Final Answer:

$$f' = b'c'd + a'bd' + ab'd + abc'$$

$$f = (b + c + d')(a + b' + d)(a' + b + d')(a' + b' + c)$$

Programmed Exercise 5.2

Problem: Determine a minimum sum-of-products expression for

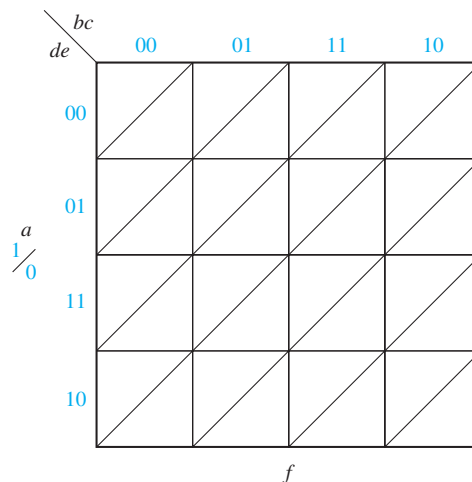
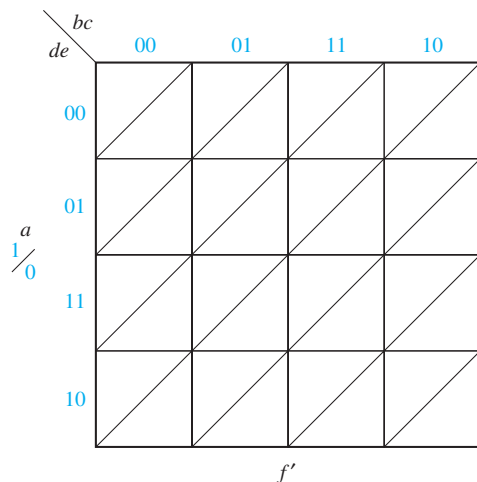
$$f(a, b, c, d, e) = (a' + c + d)(a' + b + e)(a + c' + e')(c + d + e')(b + c + d' + e)(a' + b' + c + e')$$

The first step in the solution is to plot a map for f . Because f is given in product-of-sums form, it is easier to first plot the map for f' and then complement the map. Write f' as a sum of products:

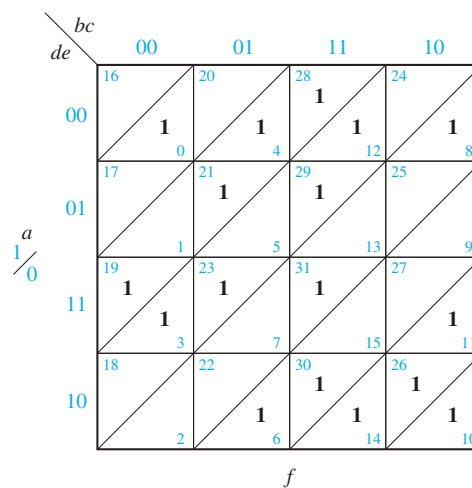
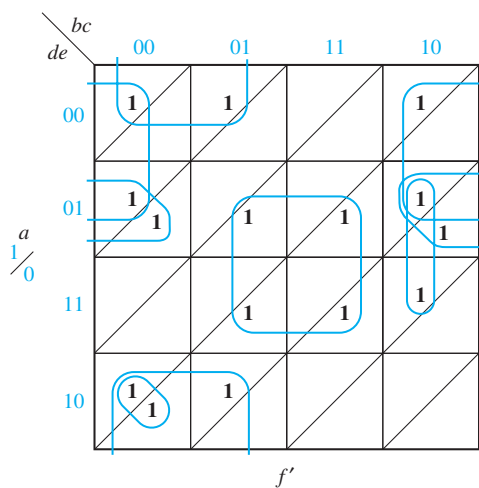
$$f' = \underline{\hspace{2cm}}$$

Now plot the map for f' . (Note that there are three terms in the upper layer, one term in the lower layer, and two terms which span the two layers.)

Next, convert your map for f' to a map for f .



Answer:



The next step is to determine the essential prime implicants of f .

(a) Why is $a'd'e'$ an essential prime implicant?

(b) Which minterms are adjacent to m_3 ? _____ To m_{19} ? _____

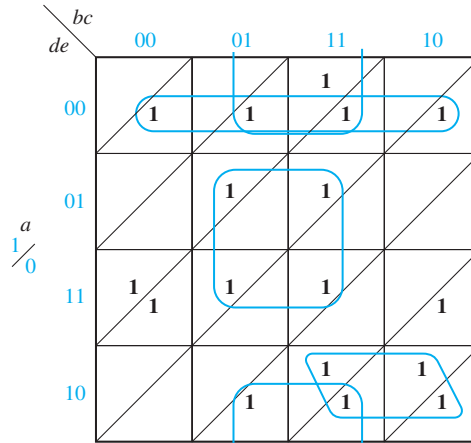
(c) Is there an essential prime implicant which covers m_3 and m_{19} ?

(d) Is there an essential prime implicant which covers m_{21} ?

(e) Loop the essential prime implicants which you have found. Then, find two more essential prime implicants and loop them.

Answers:

- (a) It covers m_0 and both adjacent minterms.
 (b) m_{19} and m_{11} ; m_3 and m_{23}
 (c) No
 (d) Yes
 (e)

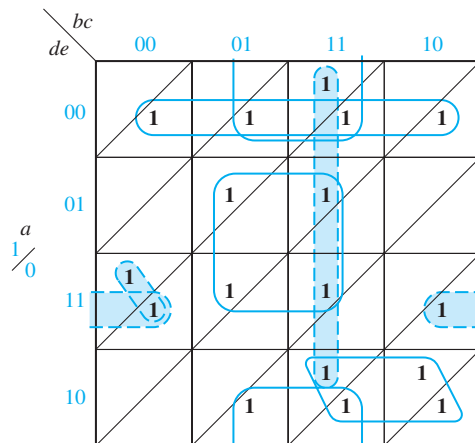


- (a) Why is there no essential prime implicant which covers m_{11} ?
 (b) Why is there no essential prime implicant which covers m_{28} ?

Because there are no more essential prime implicants, loop a minimum number of terms which cover the remaining 1's.

Answers:

- (a) All adjacent 1's of m_{11} (m_3, m_{10}) cannot be covered by one grouping.
 (b) All adjacent 1's of m_{28} (m_{12}, m_{30}, m_{29}) cannot be covered by one grouping.



Note: There are five other possible ways to loop the four remaining 1's.

Write down two different minimum sum-of-products expressions for f .

$f =$ _____
 $f =$ _____

Answer:

$$f = a'd'e' + ace + a'ce' + bde' + \left\{ \begin{array}{c} abc \\ or \\ bce' \end{array} \right\} + \left\{ \begin{array}{c} b'c'de + a'c'de \\ b'c'de + a'bc'd \\ ab'de + a'c'de \end{array} \right\}$$

Problems

- 5.3 Find the minimum sum of products for each function using a Karnaugh map.
- (a) $f_1(a, b, c) = m_0 + m_2 + m_5 + m_6$ (b) $f_2(d, e, f) = \Sigma m(0,1,2,4)$
(c) $f_3(r, s, t) = rt' + r's' + r's$ (d) $f_4(x, y, z) = M_0 \bullet M_5$
- 5.4 (a) Plot the following function on a Karnaugh map. (Do not expand to minterm form before plotting.)
- $$F(A,B,C,D) = BD' + B'CD + ABC + ABC'D + B'D'$$
- (b) Find the minimum sum of products.
(c) Find the minimum product of sums.
- 5.5 A switching circuit has two control inputs (C_1 and C_2), two data inputs (X_1 and X_2), and one output (Z). The circuit performs one of the logic operations AND, OR, EQU (equivalence), or XOR (exclusive OR) on the two data inputs. The function performed depends on the control inputs:

C_1	C_2	Function Performed by Circuit
0	0	OR
0	1	XOR
1	0	AND
1	1	EQU

- (a) Derive a truth table for Z .
(b) Use a Karnaugh map to find a minimum AND-OR gate circuit to realize Z .
- 5.6 Find the minimum sum-of-products expression for each function. Underline the essential prime implicants in your answer and tell which minterm makes each one essential.
- (a) $f(a, b, c, d) = \Sigma m(0, 1, 3, 5, 6, 7, 11, 12, 14)$
(b) $f(a, b, c, d) = \Pi M(1, 9, 11, 12, 14)$
(c) $f(a, b, c, d) = \Pi M(5, 7, 13, 14, 15) \bullet \Pi D(1, 2, 3, 9)$

- 5.7** Find the minimum sum-of-products expression for each function.
- (a) $f(a, b, c, d) = \sum m(0, 2, 3, 4, 7, 8, 14)$
 - (b) $f(a, b, c, d) = \sum m(1, 2, 4, 15) + \sum d(0, 3, 14)$
 - (c) $f(a, b, c, d) = \prod M(1, 2, 3, 4, 9, 15)$
 - (d) $f(a, b, c, d) = \prod M(0, 2, 4, 6, 8) \cdot \prod D(1, 12, 9, 15)$
- 5.8** Find the minimum sum of products and the minimum product of sums for each function:
- (a) $f(a, b, c, d) = \prod M(0, 1, 6, 8, 11, 12) \cdot \prod D(3, 7, 14, 15)$
 - (b) $f(a, b, c, d) = \sum m(1, 3, 4, 11) + \sum d(2, 7, 8, 12, 14, 15)$
- 5.9** Find the minimum sum of products and the minimum product of sums for each function:
- (a) $F(A, B, C, D, E) = \sum m(0, 1, 2, 6, 7, 9, 10, 15, 16, 18, 20, 21, 27, 30) + \sum d(3, 4, 11, 12, 19)$
 - (b) $F(A, B, C, D, E) = \prod M(0, 3, 6, 9, 11, 19, 20, 24, 25, 26, 27, 28, 29, 30) \cdot \prod D(1, 2, 12, 13)$
- 5.10** $F(a, b, c, d, e) = \sum m(0, 3, 4, 5, 6, 7, 8, 12, 13, 14, 16, 21, 23, 24, 29, 31)$
- (a) Find the essential prime implicants using a Karnaugh map, and indicate why each one of the chosen prime implicants is essential (there are four essential prime implicants).
 - (b) Find all of the prime implicants by using the Karnaugh map. (There are nine in all.)
- 5.11** Find a minimum product-of-sums solution for f . Underline the essential prime implicants.
- $$f(a, b, c, d, e) = \sum m(2, 4, 5, 6, 7, 8, 10, 12, 14, 16, 19, 27, 28, 29, 31) + \sum d(1, 30)$$
- 5.12** Given $F = AB'D' + A'B + A'C + CD$.
- (a) Use a Karnaugh map to find the maxterm expression for F (express your answer in both decimal and algebraic notation).
 - (b) Use a Karnaugh map to find the minimum sum-of-products form for F' .
 - (c) Find the minimum product of sums for F .
- 5.13** Find the minimum sum of products for the given expression. Then, make minterm 5 a don't-care term and verify that the minimum sum of products is unchanged. Now, start again with the original expression and find each minterm which could *individually* be made a don't-care without changing the minimum sum of products.
- $$F(A, B, C, D) = A'C' + B'C + ACD' + BC'D$$
- 5.14** Find the minimum sum-of-products expressions for each of these functions.
- (a) $f_1(A, B, C) = m_1 + m_2 + m_5 + m_7$
 - (b) $f_2(d, e, f) = \sum m(1, 5, 6, 7)$
 - (c) $f_3(r, s, t) = rs' + r's' + st'$
 - (d) $f_4(a, b, c) = m_0 + m_2 + m_3 + m_7$
 - (e) $f_5(n, p, q) = \sum m(1, 3, 4, 5)$
 - (f) $f_6(x, y, z) = M_1M_7$

5.15 Find the minimum product-of-sums expression for each of the functions in Problem 5.14.

5.16 Find the minimum sum of products for each of these functions.

- (a) $f_1(A, B, C) = m_1 + m_3 + m_4 + m_6$ (b) $f_2(d, e, f) = \Sigma m(1, 4, 5, 7)$
 (c) $f_3(r, s, t) = r't' + rs' + rs$ (d) $f_1(a, b, c) = m_3 + m_4 + m_6 + m_7$
 (e) $f_2(n, p, q) = \Sigma m(2, 3, 5, 7)$ (f) $f_4(x, y, z) = M_3M_6$

5.17 (a) Plot the following function on a Karnaugh map. (Do not expand to minterm form before plotting.)

$$F(A, B, C, D) = A'B' + CD' + ABC + A'B'CD' + ABCD'$$

- (b) Find the minimum sum of products.
 (c) Find the minimum product of sums.

5.18 Work Problem 5.17 for the following:

$$f(A, B, C, D) = A'B' + A'B'C' + A'BD' + AC'D + A'BD + AB'CD'$$

5.19 A switching circuit has two control inputs (C_1 and C_2), two data inputs (X_1 and X_2), and one output (Z). The circuit performs logic operations on the two data inputs, as shown in this table:

C_1	C_2	Function Performed by Circuit
0	0	X_1X_2
0	1	$X_1 \oplus X_2$
1	0	$X_1' + X_2$
1	1	$X_1 \equiv X_2$

- (a) Derive a truth table for Z .
 (b) Use a Karnaugh map to find a minimum OR-AND gate circuit to realize Z .

5.20 Use Karnaugh maps to find all possible minimum sum-of-products expressions for each function.

- (a) $F(a, b, c) = \Pi M(3, 4)$
 (b) $g(d, e, f) = \Sigma m(1, 4, 6) + \Sigma d(0, 2, 7)$
 (c) $F(p, q, r) = (p + q' + r)(p' + q + r')$
 (d) $F(s, t, u) = \Sigma m(1, 2, 3) + \Sigma d(0, 5, 7)$
 (e) $f(a, b, c) = \Pi M(2, 3, 4)$
 (f) $G(D, E, F) = \Sigma m(1, 6) + \Sigma d(0, 3, 5)$

- 5.21** Simplify the following expression first by using a map and then by using Boolean algebra. Use the map as a guide to determine which theorems to apply to which terms for the algebraic simplification.

$$F = a'b'c' + a'c'd + bcd + abc + ab'$$

- 5.22** Find all prime implicants and all minimum sum-of-products expressions for each of the following functions.

- (a) $f(A, B, C, D) = \Sigma m(4, 11, 12, 13, 14) + \Sigma d(5, 6, 7, 8, 9, 10)$
- (b) $f(A, B, C, D) = \Sigma m(3, 11, 12, 13, 14) + \Sigma d(5, 6, 7, 8, 9, 10)$
- (c) $f(A, B, C, D) = \Sigma m(1, 2, 4, 13, 14) + \Sigma d(5, 6, 7, 8, 9, 10)$
- (d) $f(A, B, C, D) = \Sigma m(4, 15) + \Sigma d(5, 6, 7, 8, 9, 10)$
- (e) $f(A, B, C, D) = \Sigma m(3, 4, 11, 15) + \Sigma d(5, 6, 7, 8, 9, 10)$
- (f) $f(A, B, C, D) = \Sigma m(4) + \Sigma d(5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$
- (g) $f(A, B, C, D) = \Sigma m(4, 15) + \Sigma d(0, 1, 2, 5, 6, 7, 8, 9, 10)$

- 5.23** For each function in Problem 5.22, find all minimum product-of-sums expressions.

- 5.24** Find the minimum sum-of-products expression for

- (a) $\Sigma m(0, 2, 3, 5, 6, 7, 11, 12, 13)$
- (b) $\Sigma m(2, 4, 8) + \Sigma d(0, 3, 7)$
- (c) $\Sigma m(1, 5, 6, 7, 13) + \Sigma d(4, 8)$
- (d) $f(w, x, y, z) = \Sigma m(0, 3, 5, 7, 8, 9, 10, 12, 13) + \Sigma d(1, 6, 11, 14)$
- (e) $\Pi M(0, 1, 2, 5, 7, 9, 11) \cdot \Pi D(4, 10, 13)$

- 5.25** Work Problem 5.24 for the following:

- (a) $f(a, b, c, d) = \Sigma m(1, 3, 4, 5, 7, 9, 13, 15)$
- (b) $f(a, b, c, d) = \Pi M(0, 3, 5, 8, 11)$
- (c) $f(a, b, c, d) = \Sigma m(0, 2, 6, 9, 13, 14) + \Sigma d(3, 8, 10)$
- (d) $f(a, b, c, d) = \Pi M(0, 2, 6, 7, 9, 12, 13) \cdot \Pi D(1, 3, 5)$

- 5.26** Find the minimum product of sums for the following. Underline the essential prime implicants in your answer.

- (a) $\Pi M(0, 2, 4, 5, 6, 9, 14) \cdot \Pi D(10, 11)$
- (b) $\Sigma m(1, 3, 8, 9, 15) + \Sigma d(6, 7, 12)$

- 5.27** Find a minimum sum-of-products and a minimum product-of-sums expression for each function:

- (a) $f(A, B, C, D) = \Pi M(0, 2, 10, 11, 12, 14, 15) \cdot \Pi D(5, 7)$
- (b) $f(w, x, y, z) = \Sigma m(0, 3, 5, 7, 8, 9, 10, 12, 13) + \Sigma d(1, 6, 11, 14)$

- 5.28** A logic circuit realizes the function $F(a, b, c, d) = a'b' + a'cd + ac'd + ab'd'$. Assuming that $a = c$ never occurs when $b = d = 1$, find a simplified expression for F .

- 5.29** Given $F = AB'D' + A'B + A'C + CD$.

- (a) Use a Karnaugh map to find the maxterm expression for F (express your answer in both decimal and algebraic notation).

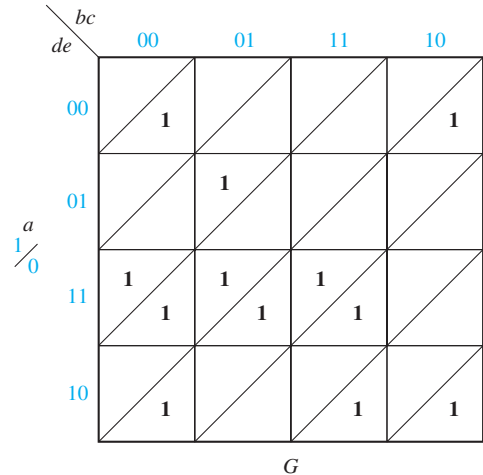
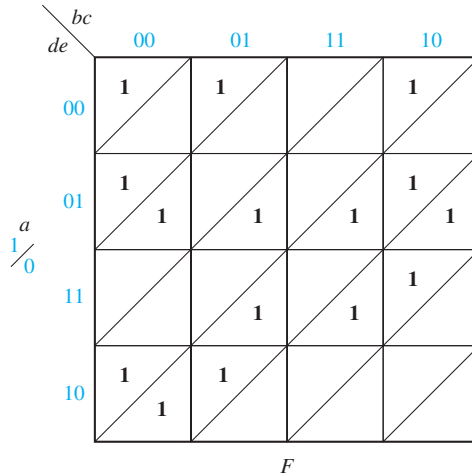
- (b) Use a Karnaugh map to find the minimum sum-of-products form for F' .
 (c) Find the minimum product of sums for F .

5.30 Assuming that the inputs $ABCD = 0101$, $ABCD = 1001$, $ABCD = 1011$ never occur, find a simplified expression for

$$F = A'BC'D + A'B'D + A'CD + ABD + ABC$$

5.31 Find all of the prime implicants for each of the functions plotted on page 150.

5.32 Find all of the prime implicants for each of the plotted functions:



5.33 Given that $f(a, b, c, d, e) = \sum m(6, 7, 9, 11, 12, 13, 16, 17, 18, 20, 21, 23, 25, 28)$, using a Karnaugh map,

- (a) Find the essential prime implicants (three).
 (b) Find the minimum sum of products (7 terms).
 (c) Find all of the prime implicants (twelve).

5.34 A logic circuit realizing the function f has four inputs a, b, c, d . The three inputs a, b , and c are the binary representation of the digits 0 through 7 with a being the most significant bit. The input d is an odd-parity bit; that is, the value of d is such that a, b, c , and d always contains an odd number of 1's. (For example, the digit 1 is represented by $abc = 001$ and $d = 0$, and the digit 3 is represented by $abcd = 0111$.) The function f has value 1 if the input digit is a prime number. (A number is prime if it is divisible only by itself and 1; 1 is considered to be prime, and 0 is not.)

- (a) Draw a Karnaugh map for f .
 (b) Find all prime implicants of f .
 (c) Find all minimum sum of products for f .
 (d) Find all prime implicants of f' .
 (e) Find all minimum product of sums for f .

- 5.35** The decimal digits 0 through 9 are represented using five bits A, B, C, D , and E . The bits A, B, C , and D are the BCD representation of the decimal digit, and bit E is a parity bit that makes the five bits have odd parity. The function $F(A, B, C, D, E)$ has value 1 if the decimal digit represented by A, B, C, D , and E is divisible by either 3 or 4. (Zero is divisible by 3 and 4.)
- Draw a Karnaugh map for f .
 - Find all prime implicants of f . (Prime implicants containing only don't-cares need not be included.)
 - Find all minimum sum of products for f .
 - Find all prime implicants of f' .
 - Find all minimum product of sums for f .
- 5.36** Rework Problem 5.35 assuming the decimal digits are represented in excess-3 rather than BCD.
- 5.37** The function $F(A, B, C, D, E) = \Sigma m(1, 7, 8, 13, 16, 19) + \Sigma d(0, 3, 5, 6, 9, 10, 12, 15, 17, 18, 20, 23, 24, 27, 29, 30)$.
- Draw a Karnaugh map for f .
 - Find all prime implicants of f . (Prime implicants containing only don't-cares need not be included.)
 - Find all minimum sum of products for f .
 - Find all prime implicants of f' .
 - Find all minimum product of sums for f .
- 5.38** $F(a, b, c, d, e) = \Sigma m(0, 1, 4, 5, 9, 10, 11, 12, 14, 18, 20, 21, 22, 25, 26, 28)$
- Find the essential prime implicants using a Karnaugh map, and indicate why each one of the chosen prime implicants is essential (there are four essential prime implicants).
 - Find all of the prime implicants by using the Karnaugh map (there are 13 in all).
- 5.39** Find the minimum sum-of-products expression for F . Underline the essential prime implicants in this expression.
- $f(a, b, c, d, e) = \Sigma m(0, 1, 3, 4, 6, 7, 8, 10, 11, 15, 16, 18, 19, 24, 25, 28, 29, 31) + \Sigma d(5, 9, 30)$
 - $f(a, b, c, d, e) = \Sigma m(1, 3, 5, 8, 9, 15, 16, 20, 21, 23, 27, 28, 31)$
- 5.40** Work Problem 5.39 with
- $$F(A, B, C, D, E) = \Pi M(2, 3, 4, 8, 9, 10, 14, 15, 16, 18, 19, 20, 23, 24, 30, 31)$$
- 5.41** Find the minimum sum-of-products expression for F . Underline the essential prime implicants in your expression.
- $$F(A, B, C, D, E) = \Sigma m(0, 2, 3, 5, 8, 11, 13, 20, 25, 26, 30) + \Sigma d(6, 7, 9, 24)$$
- 5.42** $F(V, W, X, Y, Z) = \Pi M(0, 3, 5, 6, 7, 8, 11, 13, 14, 15, 18, 20, 22, 24) \cdot \Pi D(1, 2, 16, 17)$
- Find a minimum sum-of-products expression for F . Underline the essential prime implicants.

- (b) Find a minimum product-of-sums expression for F . Underline the essential prime implicants.

5.43 Find the minimum product of sums for

(a) $F(a, b, c, d, e) = \Sigma m(1, 2, 3, 4, 5, 6, 25, 26, 27, 28, 29, 30, 31)$

(b) $F(a, b, c, d, e) = \Sigma m(1, 5, 12, 13, 14, 16, 17, 21, 23, 24, 30, 31) + \Sigma d(0, 2, 3, 4)$

5.44 Find a minimum product-of-sums expression for each of the following functions:

(a) $F(v, w, x, y, z) = \Sigma m(4, 5, 8, 9, 12, 13, 18, 20, 21, 22, 25, 28, 30, 31)$

(b) $F(a, b, c, d, e) = \Pi M(2, 4, 5, 6, 8, 10, 12, 13, 16, 17, 18, 22, 23, 24)$

• $\Pi D(0, 11, 30, 31)$

5.45 Find the minimum sum of products for each function. Then, make the specified minterm a don't-care and verify that the minimum sum of products is unchanged. Now, start again with the original expression and find each minterm which could individually be made a don't-care, without changing the minimum sum of products.

(a) $F(A, B, C, D) = A'C' + A'B' + ACD' + BC'D$, minterm 2

(b) $F(A, B, C, D) = A'BD + AC'D + AB' + BCD + A'C'D$, minterm 7

5.46 $F(V, W, X, Y, Z) = \Pi M(0, 3, 6, 9, 11, 19, 20, 24, 25, 26, 27, 28, 29, 30)$

• $\Pi D(1, 2, 12, 13)$

(a) Find two minimum sum-of-products expressions for F .

(b) Underline the essential prime implicants in your answer and tell why each one is essential.

Objectives

1. Represent gates and combinational logic by concurrent VHDL statements.
2. Given a set of concurrent VHDL statements, draw the corresponding combinational logic circuit.
3. Write a VHDL module for a combinational circuit
 - (a) by using concurrent VHDL statements to represent logic equations.
 - (b) by interconnecting VHDL components.
4. Compile and simulate a VHDL module.
5. Use the basic VHDL operators and understand their order of precedence.
6. Use the VHDL types: bit, bit_vector, Boolean, and integer. Define and use an array-type.
7. Use IEEE Standard Logic. Use std_logic_vectors, together with overloaded operators, to perform arithmetic operations.

Study Guide

1. Study Section 10.1, *VHDL Description of Combinational Circuits*.

- (a) Draw a circuit that corresponds to the following VHDL statements:

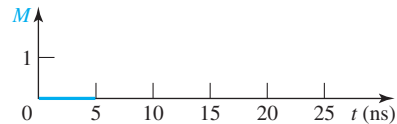
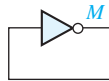
`C <= not A;` `D <= C and B;`

- (b) If A changes at time 5 ns, at what time do each of the following concurrent statements execute? At what times are C and D updated?

`C <= A;`

`D <= A;`

- (c) Write a VHDL statement that corresponds to the following circuit. The inverter has a delay of 5 ns. Draw the waveform for M assuming that M is initially 0.



- (d) Write a VHDL statement to implement $A = B \oplus C$ without using the xor or xnor operator. Do not include gate delays.

- (e) Work Problems 10.1 and 10.2.

2. Study Section 10.2, *VHDL Models for Multiplexers*.

- (a) Implement the following VHDL conditional assignment statement, using a 2-to-1 MUX:

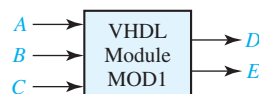
`F <= A when C = '1' else B;`

- (b) Write a VHDL conditional assignment statement that represents the 4-to-1 MUX of Figure 9-2. Assume $I_0 = 1$, $I_1 = 0$, and $I_2 = I_3 = C$.

- (c) Write a VHDL selected signal assignment for the same circuit as in (b).

3. Study Section 10.3, *VHDL Modules*, and Section 10.4, *Signals and Constants*.

- (a) Write an entity for the module MOD1. A , B , C , D , and E are all of type bit.



- (b) Write the architecture for MOD1 if $D = ABC$ and $E = D'$.
- (c) What changes must be made in the code of Figure 10-12 to implement a 5-bit adder?
- (d) Given the concurrent VHDL statements
 $R \leq A$ after 5 ns; -- statement 1
 $S \leq R$ after 10 ns; -- statement 2
 If A changes at time 3 ns, at what time will statement 1 be executed?
 At what time will R be updated?
 At what time will statement 2 be executed?
 At what time will S be updated?
 Answers: 3 ns, 8 ns, 8 ns, and 18 ns
- (e) Write a statement that defines a bit_vector constant C1 equal to 10101011.
- (f) The circuit of Figure 8-5 is implemented as a module without gate delays as follows.
 (In the figure, B is set to 1 and C is set to 0, but here, assume they are inputs.)

```

entity fig8_5 is
  port (A, B, C: in bit; G2: out bit);
end fig8_5;
architecture circuit of fig8_5 is
begin
  G2 <= not(C or (A and B));
end circuit;

```

 Each gate in Figure 8-5 has a delay of 20 ns. Modify the module to include gate delays. (*Hint:* You will need a **signal** declaration to introduce G1 as an internal signal.)
- (g) Work Problems 10.3 and 10.4.
4. Study Section 10.5, *Arrays*.
- (a) Write VHDL statements that define a ROM that is 16 words of 8 bits each. Leave the values stored in the ROM unspecified.
- (b) Work Problem 10.5.

5. Study Section 10.6, *VHDL Operators*.

- (a) For each of the following statements, eliminate one set of parentheses without changing the order of operation.
- (i) **not** ((A & B) **xor** "10")
 - (ii) (**not** (A & B) **xor** "10")
- (b) If A(0 to 7) = "11011011", what will be the result of executing the following concurrent statement?
- ```
B <= A(6 to 7)&A(0 to 5);
```
- What problem will occur when the following concurrent statement is executed?
- ```
A <= A(6 to 7)&A(0 to 5);
```
- (Hint: A concurrent statement executes every time the right-hand side changes.)
- (c) Work Problem 10.6(a).

6. Study Section 10.7, *Packages and Libraries*.

Give the entity and architecture that describes a three-input AND gate with 2-ns delay. Assume that all signals are of type bit.

7. Study Section 10.8, *IEEE Standard Logic*.

- (a) Suppose A, B, C, D, E, and F are of type std_logic. If the following concurrent statements are executed, what are the values of A, B, C, D, E, and F?
- ```
A <= '1'; A <= 'Z';
B <= '0'; B <= A;
C <= '0';
D <= A when C = '0' else 'Z';
D <= C when C = '1' else 'Z';
E <= '0' when A = '1' else C;
E <= A when C = '0' else '1';
F <= '1' when A = '1' and C = '1' else 'Z';
F <= '0' when A = '0' and C = '0' else 'Z';
```
- (b) Given the concurrent statements
- ```
F <= '0';
F <= '1' after 2 ns;
```
- What will happen if F is of type bit? What if F is of type std_logic?
- (c) Suppose in Figure 10-19 that A is 1011, B is 0111, and Cin is 1. What is Addout? Sum? Cout?

- (d) If A is a 6-bit `std_logic_vector` and B is a 4-bit `std_logic_vector`, write concurrent VHDL statements that will add A and B to result in a 6-bit sum and a carry.
- (e) Draw a circuit that implements the following VHDL code:
- ```

signal A, B, C, D: std_logic_vector(1 to 3);
signal E, F, G: std_logic;

D <= A when E = '1' else "ZZZ";
D <= B when F = '1' else "ZZZ";
D <= C when G = '1' else "ZZZ";

```
- (f) Work Problems 10.6(b), 10.7, and 10.8.
8. Before you take the test on Unit 10, pick up a lab assignment sheet and work the assigned lab problems. Turn in your VHDL code and simulation results.



## Introduction to VHDL

As integrated circuit technology has improved to allow more and more components on a chip, digital systems have continued to grow in complexity. As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very tedious and time consuming. For this reason, the use of hardware description languages in the digital design process continues to grow in importance. A hardware description language allows a digital system to be designed and debugged at a higher level before implementation at the gate and flip-flop level. The use of computer-aided design tools to do this conversion is becoming more widespread. This is analogous to writing software programs in a high-level language such as C and then using a compiler to convert the programs to machine language. The two most popular hardware description languages are VHDL and Verilog.

VHDL is a hardware description language that is used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Description Language, and VHSIC in turn stands for Very High Speed Integrated Circuit. However, VHDL is a general-purpose hardware description language which can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. VHDL was originally developed to allow a uniform method for specifying digital systems. The VHDL language became an IEEE standard in 1987, and it is widely used in industry. IEEE published a revised VHDL standard in 1993, and the examples in this text conform to that standard.

VHDL can describe a digital system at several different levels—behavioral, data flow, and structural. For example, a binary adder could be described at the behavioral level in terms of its function of adding two binary numbers, without giving any implementation details. The same adder could be described at the data flow level by giving the logic equations for the adder. Finally, the adder could be described at the structural level by specifying the interconnections of the gates which make up the adder.

VHDL leads naturally to a top-down design methodology in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level, the design can gradually be refined, eventually leading to a structural description which is closely related to the actual hardware implementation. VHDL was designed to be technology independent. If a design is described in VHDL and implemented in today's technology, the same VHDL description could be used as a starting point for a design in some future technology.

In this chapter, we introduce VHDL and illustrate how we can describe simple combinational circuits using VHDL. We will use VHDL in later units to design sequential circuits and more complex digital systems. In Unit 17, we introduce the use of CAD software tools for automatic synthesis from VHDL descriptions. These synthesis tools will derive a hardware implementation from the VHDL code.

---

## 10.1 VHDL Description of Combinational Circuits

We begin by describing a simple gate circuit using VHDL. A VHDL signal is used to describe a signal in a physical system. (Section 10.4 contains a summary of signals, constants, and types. The VHDL language also includes variables similar to variables in programming languages, but to obtain synthesizable code for hardware, signals should be used to represent hardware signals. VHDL variables are not used in this text.) The gate circuit of Figure 10-1 has five signals: A, B, C, D

**FIGURE 10-1**  
Gate Circuit

and E. The symbol “<=” is the signal assignment operator which indicates that the value computed on the right-hand side is assigned to the signal on the left side. A *behavioral* description of the circuit in Figure 10-1 is

```
E <= D or (A and B);
```

Parentheses are used to specify the order of operator execution.

The two assignment statements in Figure 10-1 give a *dataflow* description of the circuit where it is assumed that each gate has a 5-ns propagation delay. When the statements in Figure 10-1 are simulated, the first statement will be evaluated any time A or B changes, and the second statement will be evaluated any time C or D changes. Suppose that initially A = 1, and B = C = D = E = 0. If B changes to 1 at time 0, C will change to 1 at time = 5 ns. Then, E will change to 1 at time = 10 ns.

The circuit of Figure 10-1 can also be described using *structural* VHDL code. To do so requires that a two-input AND-gate component and a two-input OR-gate component be declared and defined. Components may be declared and defined either in a library or within the architecture part of the VHDL code. (VHDL architectures are discussed in Section 10.3, and packages and libraries are discussed in Section 10.7.) Instantiation statements are used to specify how components are connected. Each copy of a component requires a separate instantiation statement to specify how it is connected to other components and to the port inputs and outputs. An instantiation statement is a concurrent statement that executes anytime one of the input signals in its port map changes. The circuit of Figure 10-1 is described by instantiating the AND gate and the OR gate as follows:

```
Gate1: AND2 port map (A, B, D);
Gate2: OR2 port map (C, D, E);
```

The port map for Gate1 connects A and B to the AND-gate inputs, and it connects D to the AND-gate output. Since an instantiation statement is concurrent, whenever A or B changes, these changes go to the Gate1 inputs, and then the component computes a new value of D. Similarly, the second statement passes changes in C or D to the Gate2 inputs, and then the component computes a new value of E. This is exactly how the real hardware works. (The order in which the instantiation statements appear is irrelevant.) Instantiating a component is different than calling a function in a computer program. A function returns a new value whenever it is called, but an instantiated component computes a new output value whenever its input changes.

VHDL signal assignment statements, such as the ones in Figure 10-1, are examples of concurrent statements. The VHDL simulator monitors the right side of each concurrent statement, and any time a signal changes, the expression on the right side is immediately re-evaluated. The new value is assigned to the signal on the left side after an appropriate delay. This is exactly the way the hardware works. Any time a



gate input changes, the gate output is recomputed by the hardware, and the output changes after the gate delay.

When we initially describe a circuit, we may not be concerned about propagation delays. If we write

```
C <= A and B;
E <= C or D;
```

this implies that the propagation delays are 0 ns. In this case, the simulator will assume an infinitesimal delay referred to as  $\Delta$  (delta). Assume that initially  $A = 1$  and  $B = C = D = E = 0$ . If  $B$  is changed to 1 at time = 1 ns, then  $C$  will change at time  $1 + \Delta$  and  $E$  will change at time  $1 + 2\Delta$ .

Unlike a sequential program, the order of the above concurrent statements is unimportant. If we write

```
E <= C or D;
C <= A and B;
```

the simulation results would be exactly the same as before.

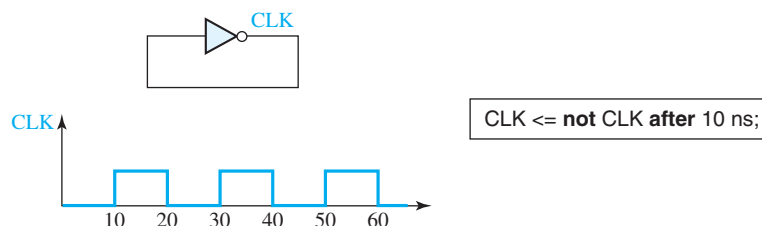
In general, a signal assignment statement has the form

```
signal_name <= expression [after delay];
```

The expression is evaluated when the statement is executed, and the signal on the left side is scheduled to change after delay. The square brackets indicate that *after delay* is optional; they are not part of the statement. If *after delay* is omitted, then the signal is scheduled to be updated after a delta delay. Note that the time at which the statement executes and the time at which the signal is updated are not the same.

Even if a VHDL program has no explicit loops, concurrent statements may execute repeatedly as if they were in a loop. Figure 10-2 shows an inverter with the output connected back to the input. If the output is '0', then this '0' feeds back to the input and the inverter output changes to '1' after the inverter delay, assumed to be 10 ns. Then, the '1' feeds back to the input, and the output changes to '0' after the inverter delay. The signal CLK will continue to oscillate between '0' and '1', as shown in the waveform. The corresponding concurrent VHDL statement will produce the same result. If CLK is initialized to '0', the statement executes and CLK changes to '1' after 10 ns. Because CLK has changed, the statement executes again, and CLK will change back to '0' after another 10 ns. This process will continue indefinitely.

**FIGURE 10-2**  
Inverter with  
Feedback



The statement in Figure 10-2 generates a clock waveform with a half period of 10 ns. On the other hand, the concurrent statement

```
CLK <= not CLK;
```

will cause a run-time error during simulation. Because there is 0 delay, the value of CLK will change at times  $0 + \Delta$ ,  $0 + 2\Delta$ ,  $0 + 3\Delta$ , etc. Because  $\Delta$  is an infinitesimal time, time will never advance to 1 ns.

In general, VHDL is not case sensitive, that is, capital and lower case letters are treated the same by the compiler and the simulator. Thus, the statements

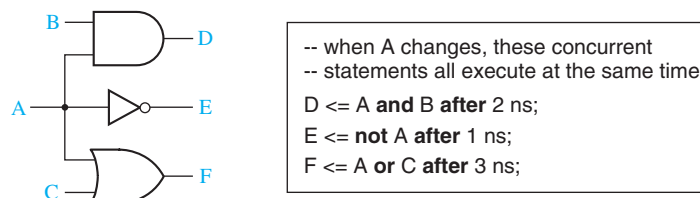
```
Clk <= NOT clk After 10 NS;
and CLK <= not CLK after 10 ns;
```

would be treated exactly the same. Signal names and other VHDL identifiers may contain letters, numbers, and the underscore character (\_). An identifier must start with a letter, and it cannot end with an underscore. Thus, C123 and ab\_23 are legal identifiers, but 1ABC and ABC\_ are not. Every VHDL statement must be terminated with a semicolon. Spaces, tabs, and carriage returns are treated in the same way. This means that a VHDL statement can be continued over several lines, or several statements can be placed on one line. In a line of VHDL code, anything following a double dash (--) is treated as a comment. Words such as **and**, **or**, and **after** are reserved words (or keywords) which have a special meaning to the VHDL compiler. In this text, we will put all reserved words in boldface type.

Figure 10-3 shows three gates that have the signal A as a common input and the corresponding VHDL code. The three concurrent statements execute simultaneously whenever A changes, just as the three gates start processing the signal change at the same time. However, if the gates have different delays, the gate outputs can change at different times. If the gates have delays of 2 ns, 1 ns, and 3 ns, respectively, and A changes at time 5 ns, then the gate outputs D, E, and F can change at times 7 ns, 6 ns, and 8 ns, respectively. The VHDL statements work in the same way. Even though the statements execute simultaneously, the signals D, E, and F are updated at times 7 ns, 6 ns, and 8 ns. However, if no delays were specified, then D, E, and F would all be updated at time  $5 + \Delta$ .

In these examples, every signal is of type bit, which means it can have a value of '0' or '1'. (Bit values in VHDL are enclosed in single quotes to distinguish them from integer values.) In digital design, we often need to perform the same operation on a group of signals. A one-dimensional array of bit signals is referred to as a bit-vector. If a 4-bit vector named B has an index range 0 through 3, then the four elements of the bit-vector are designated B(0), B(1), B(2), and B(3). The statement `B <= "0110"` assigns '0' to B(0), '1' to B(1), '1' to B(2), and '0' to B(3).

**FIGURE 10-3**  
Three Gates with a  
Common Input and  
Different Delays



**FIGURE 10-4**  
Array of AND  
Gates

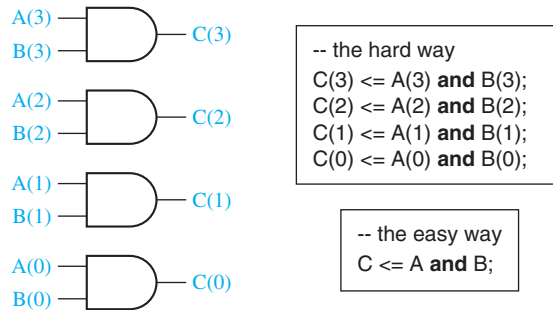


Figure 10-4 shows an array of four AND gates. The inputs are represented by bit-vectors A and B, and the outputs by bit-vector C. Although we can write four VHDL statements to represent the four gates, it is much more efficient to write a single VHDL statement that performs the **and** operation on the bit-vectors A and B. When applied to bit-vectors, the **and** operator performs the **and** operation on corresponding pairs of elements.

The preceding signal assignment statements containing “**after delay**” create what is called an **inertial** delay model. Consider a device with an inertial delay of D time units. If an input change to the device will cause its output to change, then the output changes D time units later. However, this is not what happens if the device receives two input changes within a period of D time units and both input changes should cause the output to change. In this case the device output does not change in response to either input change. As an example, consider the signal assignment

```
C <= A and B after 10 ns;
```

Assume A and B are initially 1, and A changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns. Then C changes to 1 at 10 ns and to 0 at 25 ns, but C does not change in response to the A changes at 30 ns and 35 ns because these two changes occurred less than 10 ns apart. A device with an inertial delay of D time units filters out output changes that would occur in less than or equal to D time units.

VHDL can also model devices with an **ideal (transport)** delay. Output changes caused by input changes to a device exhibiting an ideal (transport) delay of D time units are delayed by D time units, and the output changes occur even if they occur within D time units. The VHDL signal assignment statement that models ideal (transport) delay is

```
signal_name <= transport expression after delay
```

As an example, consider the signal assignment

```
C <= transport A and B after 10 ns;
```

Assume A and B are initially 1 and A changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns. Then C changes to 1 at 10 ns, to 0 at 25 ns, to 1 at 40 ns, and to 0 at 45 ns. Note that the last two changes are separated by just 5 ns.

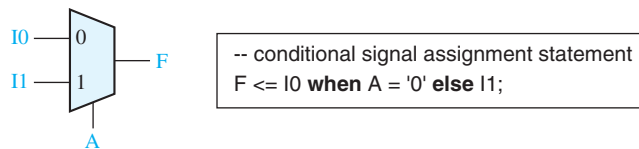
## 10.2 VHDL Models for Multiplexers

Figure 10-5 shows a 2-to-1 multiplexer (MUX) with two data inputs and one control input. The MUX output is  $F = A' \cdot I_0 + A \cdot I_1$ . The corresponding VHDL statement is

```
F <= (not A and I0) or (A and I1);
```

Alternatively, we can represent the MUX by a conditional signal assignment statement, as shown in Figure 10-5. This statement executes whenever A, I0, or I1 changes. The MUX output is I0 when A = '0', and else it is I1. In the conditional statement, I0, I1, and F can either be bits or bit-vectors.

**FIGURE 10-5**  
2-to-1 Multiplexer



The general form of a conditional signal assignment statement is

```
signal_name <= expression1 when condition1
 else expression2 when condition2
 [else expressionN];
```

This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions. If condition1 is true, signal\_name is set equal to the value of expression1, or else if condition2 is true, signal\_name is set equal to the value of expression2, etc. The line in square brackets is optional. Figure 10-6 shows how two cascaded MUXes can be represented by a conditional signal assignment statement. The output MUX selects A when E = '1'; or else it selects the output of the first MUX, which is B when D = '1', or else it is C.

**FIGURE 10-6**  
Cascaded 2-to-1  
MUXes

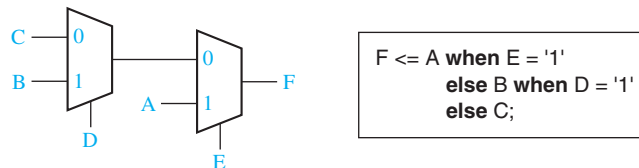


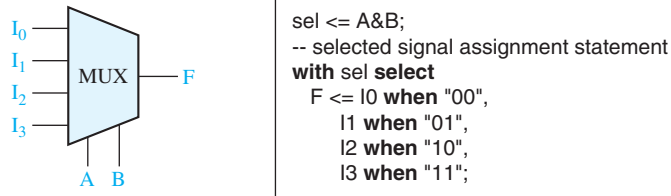
Figure 10-7 shows a 4-to-1 MUX with four data inputs and two control inputs, A and B. The control inputs select which one of the data inputs is transmitted to the output. The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

Thus, one way to model the MUX is with the VHDL statement

```
F <= (not A and not B and I0) or (not A and B and I1) or
 (A and not B and I2) or (A and B and I3);
```

**FIGURE 10-7**  
4-to-1 Multiplexer



Another way to model the 4-to-1 MUX is to use a conditional assignment statement:

```

F <= I0 when A&B = "00"
else I1 when A&B = "01"
else I2 when A&B = "10"
else I3;

```

The expression  $A\&B$  means  $A$  concatenated with  $B$ , that is, the two bits  $A$  and  $B$  are merged together to form a 2-bit vector. This bit vector is tested, and the appropriate MUX input is selected. For example, if  $A = '1'$  and  $B = '0'$ ,  $A\&B = "10"$  and  $I2$  is selected. Instead of concatenating  $A$  and  $B$ , we could use a more complex condition:

```

F <= I0 when A = '0' and B = '0'
else I1 when A = '0' and B = '1'
else I2 when A = '1' and B = '0'
else I3;

```

A third way to model the MUX is to use a selected signal assignment statement, as shown in Figure 10-7.  $A\&B$  cannot be used in this type of statement, so we first set  $Sel$  equal to  $A\&B$ . The value of  $Sel$  then selects the MUX input that is assigned to  $F$ .

The general form of a selected signal assignment statement is

```

with expression_s select
 signal_s <= expression1 [after delay-time] when choice1,
 expression2 [after delay-time] when choice2,
 . . .
 [expression_n [after delay-time] when others];

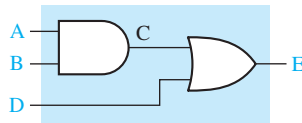
```

This concurrent statement executes whenever a signal changes in any of the expressions. First,  $expression\_s$  is evaluated. If it equals  $choice1$ ,  $signal\_s$  is set equal to  $expression1$ ; if it equals  $choice2$ ,  $signal\_s$  is set equal to  $expression2$ ; etc. If all possible choices for the value of  $expression\_s$  are given, the last line should be omitted; otherwise, the last line is required. When it is present, if  $expression\_s$  is not equal to any of the enumerated choices,  $signal\_s$  is set equal to  $expression\_n$ . The  $signal\_s$  is updated after the specified delay-time, or after  $\Delta$  if the “**after** delay-time” is omitted.

## 10.3 VHDL Modules

To write a complete VHDL module, we must declare all of the input and output signals using an **entity** declaration, and then specify the internal operation of the module using an **architecture** declaration. As an example, consider Figure 10-8. The entity declaration gives the name “two\_gates” to the module. The port declaration specifies the inputs and outputs to the module. A, B, and D are input signals of type bit, and E is an output signal of type bit. The architecture is named “gates”. The signal C is declared within the architecture because it is an internal signal. The two concurrent statements that describe the gates are placed between the keywords **begin** and **end**.

**FIGURE 10-8**  
VHDL Module with  
Two Gates



```
entity two_gates is
 port (A,B,D: in bit; E: out bit);
end two_gates;
architecture gates of two_gates is
 signal C: bit;
begin
 C <= A and B; -- concurrent
 E <= C or D; -- statements
end gates;
```

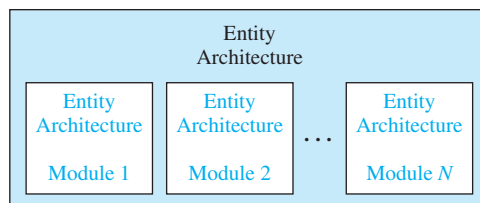
When we describe a system in VHDL, we must specify an entity and an architecture at the top level, and also specify an entity and architecture for each of the component modules that are part of the system (see Figure 10-9). Each entity declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. We will use entity declarations of the form:

```
entity entity-name is
 [port(interface-signal-declaration);]
end [entity] [entity-name];
```

The items enclosed in square brackets are optional. The interface-signal-declaration normally has the following form:

```
list-of-interface-signals: mode type [: = initial-value]
{; list-of-interface-signals: mode type [: = initial-value]};
```

**FIGURE 10-9**  
VHDL Program  
Structure



The curly brackets indicate zero or more repetitions of the enclosed clause. Input signals are of mode **in**, output signals are of mode **out**, and bi-directional signals (see Figure 9-12) are of mode **inout**.

So far, we have only used type `bit` and `bit_vector`; other types are described in Section 10.4. The optional initial-value is used to initialize the signals on the associated list; otherwise, the default initial value is used for the specified type. For example, the port declaration

```
port(A, B: in integer := 2; C, D: out bit);
```

indicates that A and B are input signals of type `integer` that are initially set to 2, and C and D are output signals of type `bit` that are initialized by default to '0'.

Associated with each entity is one or more architecture declarations of the form

```
architecture architecture-name of entity-name is
 [declarations]
begin
 architecture body
end [architecture] [architecture-name];
```

In the declarations section, we can declare signals and components that are used within the architecture. The architecture body contains statements that describe the operation of the module.

Next, we will write the entity and architecture for a full adder module (refer to Section 4.7 for a description of a full adder). The entity specifies the inputs and outputs of the adder module, as shown in Figure 10-10. The port declaration specifies that X, Y and Cin are input signals of type `bit`, and that Cout and Sum are output signals of type `bit`.

**FIGURE 10-10**  
Entity Declaration  
for a Full Adder  
Module



The operation of the full adder is specified by an architecture declaration:

```
architecture Equations of FullAdder is
begin
 -- concurrent assignment statements
 Sum <= X xor Y xor Cin after 10 ns;
 Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

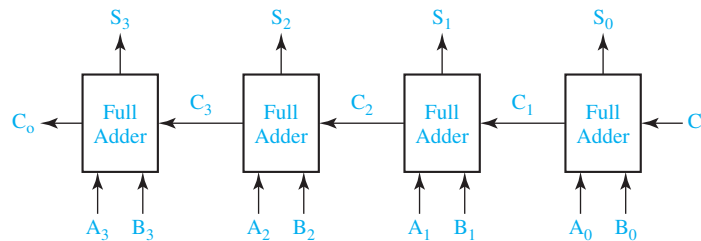
In this example, the architecture name (Equations) is arbitrary, but the entity name (FullAdder) must match the name used in the associated entity declaration.

The VHDL assignment statements for Sum and Cout represent the logic equations for the full adder. Several other architectural descriptions such as a truth table or an interconnection of gates could have been used instead. In the Cout equation, parentheses are required around  $(X \text{ and } Y)$  because VHDL does not specify an order of precedence for the logic operators.

### Four-Bit Full Adder

Next, we will show how to use the FullAdder module defined above as a component in a system which consists of four full adders connected to form a 4-bit binary adder (see Figure 10-11). We first declare the 4-bit adder as an entity (see Figure 10-12). Because the inputs and the sum output are four bits wide, we declare them as bit\_vectors which are dimensioned 3 **downto** 0. (We could have used a range 1 **to** 4 instead.)

**FIGURE 10-11**  
4-Bit Binary Adder



Next, we specify the FullAdder as a component within the architecture of Adder4 (Figure 10-12). The component specification is very similar to the entity declaration for the full adder, and the input and output port signals correspond to those declared for the full adder. Following the component statement, we declare a 3-bit internal carry signal C.

In the body of the architecture, we create several instances of the FullAdder component. (In CAD jargon, we *instantiate* four copies of the FullAdder.) Each copy of FullAdder has a name (such as FA0) and a port map. The signal names following the port map correspond one-to-one with the signals in the component port. Thus, A(0), B(0), and Ci correspond to the inputs X, Y, and Cin, respectively. C(1) and S(0) correspond to the Cout and Sum outputs. Note that the order of the signals in the port map must be the same as the order of the signals in the port of the component declaration.

In preparation for simulation, we can place the entity and architecture for the FullAdder and for Adder4 together in one file and compile. Alternatively, we could compile the FullAdder separately and place the resulting code in a library which is linked in when we compile Adder4.

All of the simulation examples in this text use the ModelSim simulator from Model Tech. Most other VHDL simulators use similar command files and can



**FIGURE 10-12**  
Structural  
Description of  
4-Bit Adder

```
entity Adder4 is
 port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
 S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;
architecture Structure of Adder4 is
 component FullAdder
 port (X, Y, Cin: in bit; -- Inputs
 Cout, Sum: out bit); -- Outputs
 end component;
 signal C: bit_vector(3 downto 1);
 begin -- instantiate four copies of the FullAdder
 FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
 FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
 FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
 FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
 end Structure;
```

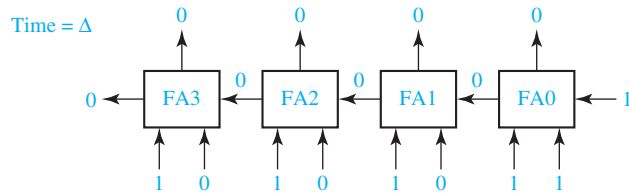
produce output in a similar format. We will use the following simulator commands to test Adder4:

```
add list A B Co C Ci S -- put these signals on the output list
force A 1111 -- set the A inputs to 1111
force B 0001 -- set the B inputs to 0001
force Ci 1 -- set Ci to 1
run 50 ns -- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

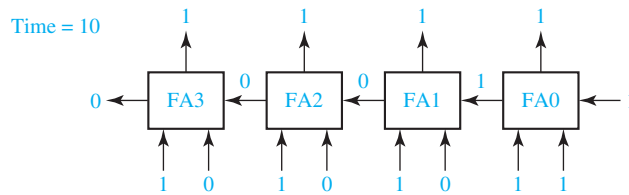
We have chosen to run the simulation for 50 ns because this is more than enough time for the carry to propagate through all of the full adders. The simulation results for the above command list are:

| ns | delta | a    | b    | co | c   | ci | s    |
|----|-------|------|------|----|-----|----|------|
| 0  | +0    | 0000 | 0000 | 0  | 000 | 0  | 0000 |
| 0  | +1    | 1111 | 0001 | 0  | 000 | 1  | 0000 |
| 10 | +0    | 1111 | 0001 | 0  | 001 | 1  | 1111 |
| 20 | +0    | 1111 | 0001 | 0  | 011 | 1  | 1101 |
| 30 | +0    | 1111 | 0001 | 0  | 111 | 1  | 1001 |
| 40 | +0    | 1111 | 0001 | 1  | 111 | 1  | 0001 |
| 50 | +0    | 0101 | 1110 | 1  | 111 | 0  | 0001 |
| 60 | +0    | 0101 | 1110 | 1  | 110 | 0  | 0101 |
| 70 | +0    | 0101 | 1110 | 1  | 100 | 0  | 0111 |
| 80 | +0    | 0101 | 1110 | 1  | 100 | 0  | 0011 |

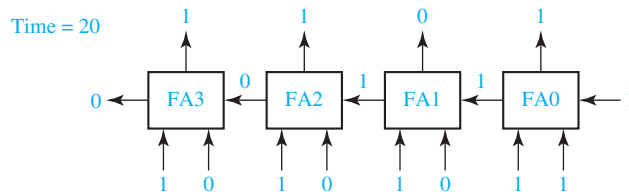
The listing shows how the carry propagates one position every 10 ns. The full adder inputs change at time =  $\Delta$ :



The sum and carry are computed by each FA and appear at the FA outputs 10 ns later:



Because the inputs to FA1 have changed, the outputs change 10 ns later:



The final simulation results are:

1111 + 0001 + 1 = 0001 with a carry of 1 (at time = 40 ns) and  
0101 + 1110 + 0 = 0011 with a carry of 1 (at time = 80 ns).

The simulation stops at 80 ns because no further changes occur after that time. For more details on how the simulator handles  $\Delta$  delays, refer to Section 10.9.

In this section we have shown how to construct a VHDL module using an entity-architecture pair. The 4-bit adder module demonstrates the use of VHDL components to write structural VHDL code. Components used within the architecture are declared at the beginning of the architecture, using a component declaration of the form

```
component component-name
 port (list-of-interface-signals-and-their-types);
end component;
```

The port clause used in the component declaration has the same form as the port clause used in an entity declaration. The connections to each component used in a circuit are specified by using a component instantiation statement of the form

label: component-name **port map** (list-of-actual-signals);

The list of actual signals must correspond one-to-one to the list of interface signals specified in the component declaration.

## 10.4 Signals and Constants

Input and output signals for a module are declared in a port. Signals internal to a module are declared at the start of an architecture, before **begin**, and can be used only within that architecture. Port signals have an associated mode (usually in or out), but internal signals do not. A signal used within an architecture must be declared either in a port or in the declaration section of an architecture, but it cannot be declared in both places. A signal declaration has the form

**signal** list\_of\_signal\_names: type\_name [constraint] [:= initial\_value];

The constraint can be an index range like (0 **to** 5) or (4 **downto** 1), or it can be a range of values such as **range** 0 to 7. Examples:

**signal** A, B, C: bit\_vector(3 **downto** 0):= "1111";

A, B, and C are 4-bit vectors dimensioned 3 **downto** 0 and initialized to 1111.

**signal** E, F: integer **range** 0 to 15;

E and F are integers in the range 0 to 15, initialized by default to 0. The compiler or simulator will flag an error if we attempt to assign a value outside the specified range to E or F.

Constants declared at the start of an architecture can be used anywhere within that architecture. A constant declaration is similar to a signal declaration:

**constant** constant\_name: type\_name [constraint] [:= constant\_value];

A constant named limit of type integer with a value of 17 can be defined as

**constant** limit : integer := 17;

A constant named delay1 of type time with the value of 5 ns can be defined as

**constant** delay1 : time := 5 ns;

This constant could then be used in an assignment statement

A <= B **after** delay1;

Once the value of a constant is defined in a declaration statement, unlike a signal, the value cannot be changed by using an assignment statement.

Signals and constants can have any one of the predefined VHDL types, or they can have a user-defined type. Some of the predefined types are

**Definition**


---

|           |                                                                                                                                                                                   |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| bit       | '0' or '1'                                                                                                                                                                        |
| boolean   | FALSE or TRUE                                                                                                                                                                     |
| integer   | an integer in the range $-(2^{31} - 1)$ to $+(2^{31} - 1)$<br>(some implementations support a wider range)                                                                        |
| positive  | an integer in the range 1 to $2^{31} - 1$ (positive integers)                                                                                                                     |
| natural   | an integer in the range 0 to $2^{31} - 1$ (positive integers and zero)                                                                                                            |
| real      | floating-point number in the range $-1.0\text{E}38$ to $+1.0\text{E}38$                                                                                                           |
| character | any legal VHDL character including upper- and lower case letters, digits, and special characters; each printable character must be enclosed in single quotes, e.g., 'd', '7', '+' |
| time      | an integer with units fs, ps, ns, us, ms, sec, min, or hr                                                                                                                         |

---

Note that the integer range for VHDL is symmetrical even though the range for a 32-bit 2's complement integer is  $-2^{31}$  to  $+(2^{31} - 1)$ .

A common user-defined type is the enumeration type in which all of the values are enumerated. For example, the declarations

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

define a signal called state which can have any one of the values S0, S1, S2, S3, S4, or S5 and which is initialized to S1. If no initialization is given, the default initialization is the left most element in the enumeration list, S0 in this example. If we declare the signal state as shown, the following assignment statement sets state to S3:

```
state <= S3;
```

VHDL is a strongly-typed language so signals of different types generally cannot be mixed in the same assignment statement, and no automatic type conversion is performed. Thus the statement  $A \leq B \text{ or } C$  is only valid if A, B, and C all have the same type or closely related types.

---

## 10.5 Arrays

In order to use an array in VHDL, we must first declare an array type, and then declare an array object. For example, the following declaration defines a one-dimensional array type named SHORT\_WORD:

```
type SHORT_WORD is array (15 downto 0) of bit;
```

An array of this type has an integer index with a range from 15 downto 0, and each element of the array is of type bit.

Next, we will declare array objects of type SHORT\_WORD:

```
signal DATA_WORD: SHORT_WORD;
signal ALT_WORD: SHORT_WORD := "01010101010101";
constant ONE_WORD: SHORT_WORD := (others => '1');
```

DATA\_WORD is a signal array of 16 bits, indexed 15 downto 0, which is initialized (by default) to all '0' bits. ALT\_WORD is a signal array of 16 bits which is initialized to alternating 0's and 1's. ONE\_WORD is a constant array of 16 bits; all bits are set to '1' by (**others** => '1'). Because none of the bits have been set individually,<sup>1</sup> in this case **others** applies to all of the bits.

We can reference individual elements of the array by specifying an index value. For example, ALT\_WORD(0) accesses the far right bit of ALT\_WORD. We can also specify a portion of the array by specifying an index range: ALT\_WORD(5 **downto** 0) accesses the low order six bits of ALT\_WORD, which have an initial value of 010101.

The array type and array object declarations illustrated above have the general forms:

```
type array_type_name is array index_range of element_type;
signal array_name: array_type_name [:= initial_values];
```

In this declaration, **signal** may be replaced with **constant**.

Multidimensional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array signal which is a matrix of integers with four rows and three columns:

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
signal matrixA: matrix4x3 := ((1,2,3),(4,5,6),(7,8,9),(10,11,12));
```

The signal matrixA, will be initialized to

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

The array element matrixA(3,2) references the element in the third row and second column, which has a value of 8. The statement B <= matrixA(2,3) assigns a value of 6 to B.

When an array type is declared, the dimensions of the array may be left undefined. This is referred to as an unconstrained array type. For example,

```
type intvec is array (natural range <>) of integer;
```

declares intvec as an array type which defines a one-dimensional array of integers with an unconstrained index range of natural numbers. The default type for array indices is integer, but another type may be specified. Because the index range is not specified in the unconstrained array type, the range must be specified when the array object is declared. For example,

```
signal intvec5: intvec(1 to 5) := (3,2,6,8,1);
```

defines a signal array named intvec5 with an index range of 1 to 5, which is initialized to 3, 2, 6, 8, 1. The following declaration defines matrix as a two-dimensional array with unconstrained row and column index ranges:

```
type matrix is array (natural range <> , natural range <>) of integer;
```

<sup>1</sup>See Reference [1, p. 86] for information on how to set individual bits.

Predefined unconstrained array types in VHDL include `bit_vector` and `string`, which are defined as follows:

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
```

The characters in a string literal must be enclosed in double quotes. For example, “This is a string.” is a string literal. The following example declares a constant `string1` of type `string`:

```
constant string1: string(1 to 29) := “This string is 29 characters.”
```

A `bit_vector` literal may be written either as a list of bits separated by commas or as a string. For example, `(‘1’,‘0’,‘1’,‘1’,‘0’)` and `“10110”` are equivalent forms. The following declares a constant `A` which is a `bit_vector` with a range 0 to 5.

```
constant A : bit_vector(0 to 5) := “101011”;
```

A truth table can be implemented using a ROM (read-only memory) as illustrated in Figure 9-17. If we represent the ROM outputs by a `bit_vector`, `F(0 to 3)`, we can represent the truth table that is stored in the ROM by an array of `bit_vectors`. The VHDL code for this ROM is given in Figure 10-13. The port declaration (line 4) defines the inputs and outputs for the ROM. The type declaration (line 7) defines an array with 8 rows where each row is 4 bits wide. Line 8 declares `ROM1` to be an array of this type with binary data stored in each row. Line 9 declares an integer called `index`. This index will be used to select one of the 8 rows in the `ROM1` array. In line 11, this index is formed by concatenating the three input bits to form a 3-bit vector, and this vector is converted to an integer. The data is read from the `ROM1` array in line 13. For example, if `A = ‘1’`, `B = ‘0’`, and `C = ‘1’`, `index = 5`, and “0001” is read from the ROM. Lines 1 and 2 allow us to use the `vec2int` function, which is defined in a library named `BITLIB`.

**FIGURE 10-13** VHDL Description of a ROM

---

```
1 library BITLIB;
2 use BITLIB.bit_pack.all;
3 entity ROM9_17 is
4 port (A, B, C: in bit; F: out bit_vector(0 to 3));
5 end entity;
6 architecture ROM of ROM9_17 is
7 type ROM8X4 is array (0 to 7) of bit_vector(0 to 3);
8 constant ROM1: ROM8X4 := (“1010”, “1010”, “0111”, “0101”, “1100”, “0001”, “1111”, “0101”);
9 signal index: Integer range 0 to 7;
10 begin
11 index <= vec2int(A&B&C); -- A&B&C is a 3-bit vector
12 -- vec2int is a function that converts this vector to an integer
13 F <= ROM1 (index);
14 -- this statement reads the output from the ROM
15 end ROM;
```

---

## 10.6 VHDL Operators

Predefined VHDL operators can be grouped into seven classes:

1. binary logical operators: **and or nand nor xor xnor**
2. relational operators: **= /= < <= > >=**
3. shift operators: **sll srl sla sra rol ror**
4. adding operators: **+ - &** (concatenation)
5. unary sign operators: **+ -**
6. multiplying operators: **\* / mod rem**
7. miscellaneous operators: **not abs \*\***

When parentheses are not used, operators in class 7 have highest precedence and are applied first, followed by class 6, then class 5, etc. Class 1 operators have lowest precedence and are applied last. Operators in the same class have the same precedence and are applied from left to right in an expression. The precedence order can be changed by using parentheses. In the following expression, A, B, C, and D are bit\_vectors:

**not A or B and not C & D**

In this expression, **not** is performed first, then **&** (concatenation), then **or**, and finally **and**. The equivalent expression using parentheses is

**((not A) or B) and ((not C) & D)**

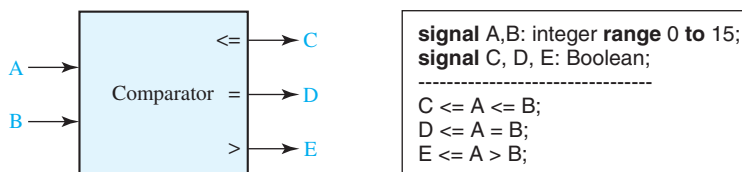
The binary logical operators (class 1) as well as **not** can be applied to bits, booleans, bit\_vectors, and boolean\_vectors. The class 1 operators require two operands of the same type and size, and the result is of that type and size.

Relational operators (class 2) are used to compare two expressions and return a value of FALSE or TRUE. The two expressions must be of the same type and size. Equal (=) and not equal (/=) apply to any type, but the application of the other relational operators is more restricted. Note that “=” is always a relational operator, but “<=” also serves as an assignment operator. Example: If A = 5, B = 4, and C = 3 the expression

(A >= B) and (B <= C) evaluates to FALSE.

Figure 10-14 shows a comparator for two integers with a restricted range. C must be of type Boolean since the condition A <= B evaluates to TRUE or FALSE. If we implement the comparator in hardware, each integer would be represented by a 4-bit signal because the range is restricted to 0 to 15. C, D, and E would each be one bit (0 for FALSE or 1 for TRUE).

**FIGURE 10-14**  
Comparator for  
Integers



The shift operators are used to shift or rotate a `bit_vector`. In the following examples, A is an 8-bit vector equal to “10010101”:

A **sll** 2 is “01010100” (shift left logical, filled with ‘0’)  
 A **srl** 3 is “00010010” (shift right logical, filled with ‘0’)  
 A **sla** 3 is “10101111” (shift left arithmetic, filled with rightmost bit)  
 A **sra** 2 is “11100101” (shift right arithmetic, filled with leftmost bit)  
 A **rol** 3 is “10101100” (rotate left)  
 A **ror** 5 is “10101100” (rotate right)

We will not utilize these shift operators because some software used for synthesis uses different shift operators. Instead, we will do shifting using the concatenation operator. For example, if A in the above listing is dimensioned 7 downto 0, we can implement shift right arithmetic two places as follows:

`A(7)&A(7)&A(7 downto 2) = '1'&'1'&"100101" = "11100101"`

This makes two copies of the sign bit followed by the left 6 bits of A, which gives the same result as A **sra** 2.

The + and – operators can be applied to integer or real numeric operands. The & operator can be used to concatenate two vectors (or an element and a vector, or two elements) to form a longer vector. For example, “010” & “1” is “0101” and “ABC” & “DEF” is “ABCDEF.”

The \* and / operators perform multiplication and division on integer or floating-point operands. The **rem** and **mod** operators calculate the remainder and modulus for integer operands. (We will not use rem and mod; for further discussion of these operators see Reference [1].) The \*\* operator raises an integer or floating-point number to an integer power, and **abs** finds the absolute value of a numeric operand.

## 10.7 Packages and Libraries

Packages and libraries provide a convenient way of referencing frequently used functions and components. A package consists of a package declaration and an optional package body. The package declaration contains a set of declarations which may be shared by several design units. For example, it may contain type, signal, component, function, and procedure declarations. The package body usually contains component descriptions and the function and procedure bodies. The package and its associated compiled VHDL models may be placed in a library, so they can be accessed as required by different VHDL designs. A package declaration has the form:

```
package package-name is
 package declarations
end [package][package-name];
```



```

A package body has the form
package body package-name is
 package body declarations
end [package body][package name];

```

We have developed a package called `bit_pack` which is used in a number of examples in this book. This package contains commonly used components and functions which use signals of type `bit` and `bit_vector`. A complete listing of this package and associated component models is included on the CD-ROM that accompanies this text. Most of the components in this package have a default delay of 10 ns, but this delay can be changed by the use of generics. For an explanation of generics, refer to one of the VHDL references. We have compiled this package and the component models and placed the result in a library called `BITLIB`.

One of the components in the library is a two-input NOR gate named `Nor2`, which has default delay of 10 ns. The package declaration for `bit_pack` includes the component declaration

```

component Nor2
 port (A1, A2: in bit; Z: out bit);
end component;

```

The NOR gate is modeled using a concurrent statement. The entity-architecture pair for this component is

```

-- two-input NOR gate
entity Nor2 is
 port (A1, A2: in bit; Z: out bit);
end Nor2;

architecture concur of Nor2 is
begin
 Z <= not(A1 or A2) after 10 ns;
end concur;

```

To access components and functions within a package requires a **library** statement and a **use** statement. The statement

```
library BITLIB;
```

allows your design to access the `BITLIB`. The statement

```
use BITLIB.bit_pack.all;
```

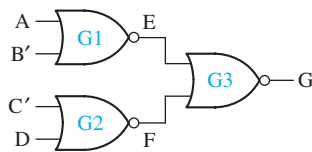
allows your design to use the entire `bit_pack` package. A statement of the form

```
use BITLIB.bit_pack.Nor2;
```

may be used if you want to use a specific component (in this case `Nor2`) or function in the package.

When components from a library package are used, component declarations are not needed. Figure 10-15 shows a NOR-NOR circuit and the corresponding structural VHDL code. This code instantiates three copies of the `Nor2` gate component from the package `bit_pack` and connects the gate inputs and outputs.

**FIGURE 10-15**  
NOR-NOR Circuit  
and Structural  
VHDL Code  
Using Library  
Components



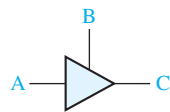
```
library BITLIB;
use BITLIB.bit_pack.all;
entity nor_nor is
 port (A,B,C,D: in bit; G: out bit);
end nor_nor;
architecture structural of nor_nor is
 signal E,F,BN,CN: bit; -- internal signals
begin
 BN <= not B; CN <= not C;
 G1: Nor2 port map (A, BN, E);
 G2: Nor2 port map (CN, D, F);
 G3: Nor2 port map (E, F, G);
end structural;
```

## 10.8 IEEE Standard Logic

Use of two-valued logic (bits and bit vectors) is generally not adequate for simulation of digital systems. In addition to '0' and '1', values of 'Z' (high-impedance or no connection) and 'X' (unknown) are frequently used in digital system simulation. The IEEE Standard 1164 defines a `std_logic` type that actually has nine values ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', and '-'). We will only be concerned with the first five values in this text. 'U' stands for uninitialized. When a logic circuit is first turned on and before it is reset, the signals will be uninitialized. If these signals are represented by `std_logic`, they will have a value of 'U' until they are changed. Just as a group of bits is represented by a `bit_vector`, a group of `std_logic` signals is represented by a `std_logic_vector`.

Figure 10-16 shows how a tri-state buffer can be represented by a concurrent statement. When the buffer is enabled ( $B = '1'$ ), the output is A, or else it is high impedance ('Z'). A and C could be `std_logic_vectors` instead of `std_logic` bits.

**FIGURE 10-16**  
Tri-State Buffer



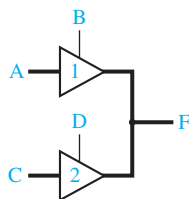
```
signal A,B,C: std_logic;

C <= A when B = '1' else 'Z';
```

Figure 10-17 shows two tri-state buffers with their outputs connected together by a tri-state bus. If buffer 1 has an output of '1' and buffer 2 has a hi-Z output, the bus value is '1'. When both buffers are enabled, if buffer 1 drives '0' onto the bus and buffer 2 drives '1' onto the bus, the result is a bus conflict. In this case, the bus value is unknown, which we represent by an 'X'.

In the VHDL code, A, C, and F are `std_logic_vectors` and F represents the tri-state bus. The signal F is driven from two different sources. If the two concurrent statements

**FIGURE 10-17**  
Tri-State Buffers  
Driving a Bus



```
signal A,C,F: std_logic_vector(3 downto 0);
signal B,D: std_logic;

-- concurrent statements
F <= A when B = '1' else "ZZZZ";
F <= C when D = '1' else "ZZZZ";
```

assign different values to F, VHDL automatically calls a *resolution function* to determine the resulting value. This is similar to the way the hardware works—if the two buffers have different output values, the hardware resolves the values and comes up with an appropriate value on the bus. VHDL uses the table of Figure 10-18 to resolve the bus value when two different `std_logic` signals, S1 and S2, drive the bus. (Only signal values ‘U’, ‘X’, ‘0’, ‘1’, and ‘Z’ are considered here.) This table is similar to Figure 9-10, which is used for four-valued logic simulation, except for the addition of a row and a column corresponding to ‘U’. When an uninitialized signal is connected to any other signal, VHDL considers that the result is uninitialized.

**FIGURE 10-18**  
Resolution Function  
for Two Signals

| S1 | S2 |   |   |   |   |
|----|----|---|---|---|---|
|    | U  | X | 0 | 1 | Z |
| U  | U  | U | U | U | U |
| X  | U  | X | X | X | X |
| 0  | U  | X | 0 | X | 0 |
| 1  | U  | X | X | 1 | 1 |
| Z  | U  | X | 0 | 1 | Z |

If A, B, and F are bits (or `bit_vectors`) and we write the concurrent statements

```
F <= A; F <= not B;
```

the compiler will flag an error because no resolution function exists for signals of type `bit`. If A, B, and F are `std_logic` bits or vectors, the compiler will generate a call to the resolution function and not report an error. If F is assigned conflicting values during simulation, then F will be set to ‘X’ (unknown).

In order to use signals of type `std_logic` and `std_logic_vector` in a VHDL module, the following declarations must be placed before the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

The IEEE `std_logic_1164` package defines `std_logic` and related types, logic operations on these types, and functions for working with these types.

The original IEEE standards for VHDL do not define arithmetic operations on `bit_vectors` or on `std_logic` vectors. Based on these standards, we cannot add, subtract, multiply, or divide `bit_vectors` or `std_logic_vectors` without first converting them to other types. For example, if A and B are `bit_vectors`, the expression `A + B` is not allowed. However, VHDL libraries and packages are available that define arithmetic and comparison operations on `std_logic_vectors`. The operators defined in these packages are referred to as *overloaded* operators. This means that the compiler will automatically use the proper definition of the operator depending on its context. For example, when evaluating the expression `A + B`, if A and B are integers, the compiler will use the integer arithmetic routine to do the addition. On the other hand, if A and B are of type `std_logic_vector`, the compiler will use the addition routine for standard logic vectors. In order to use overloaded operators, the appropriate library and use statements must be included in the VHDL code so that the compiler can locate the definitions of these operators.

In this text, we will use the `std_logic_unsigned` package, originally developed by Synopsis and now widely available. This package treats `std_logic_vectors` as

unsigned numbers. The `std_logic_unsigned` package defines arithmetic operators (+, −, \*) and comparison operators (<, <=, =, /=, >, >=) that operate on `std_logic_vectors`. For +, −, and comparison operators, if the two operands are of different length, the shorter operand is filled on the left end with zeros.

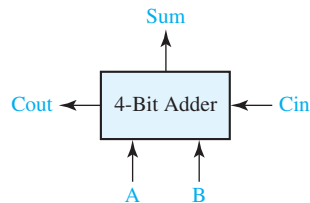
These operations can also be applied when the left operand is a `std_logic_vector` and the right operand is an integer. The arithmetic operations return a `std_logic_vector`, and the comparison operations return a Boolean. For example, if A is “10011”, A + 7 returns a value of “11010”, and A >= 5 returns TRUE. In these examples, + and >= are overloaded operators, and the compiler automatically calls the appropriate routine to add an integer to a `std_logic_vector` or to compare an integer with a `std_logic_vector`.

If A and B are 4-bit `std_logic` vectors, A + B gives their sum as a 4-bit vector, and any carry is lost. If the carry is needed, then A must be extended to five-bits before addition. This is accomplished by concatenating a ‘0’ in front of A. Then ‘0’ & A + B gives a 5-bit sum that can be split into a carry and a 4-bit sum.

Figure 10-19 shows a binary adder and its VHDL representation using the `std_logic_unsigned` package. Addout is a 5-bit sum that is split into Sum and Cout. For example, if A = “1011”, B = “1001”, and Cin = ‘1’, Addout evaluates to “10101”, which is then split into a sum “0101” with a carry out of ‘1’.

Figure 10-20 shows how to implement the bi-directional input-output pin and tri-state buffer of Figure 9-12 using IEEE `std_logic`. The I/O pin declared in the port

**FIGURE 10-19**  
VHDL Code for  
Binary Adder



```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

signal A,B,Sum: std_logic_vector(3 downto 0);
signal Addout: std_logic_vector(4 downto 0);
signal Cin,Cout: std_logic;

Addout <= '0' & A + B + Cin;
Sum <= Addout(3 downto 0);
Cout <= Addout(4);
```

**FIGURE 10-20**  
VHDL Code for  
Bi-Directional  
I/O Pin

```
entity IC_pin is
 port(IO_pin: inout std_logic);
end entity;
architecture bi_dir of IC_pin is
 component IC
 port(input: in std_logic; output: out std_logic);
 end component;
 signal input, output, en: std_logic;
begin
 -- connections to bi-directional I/O pin
 IO_pin <= output when en = '1' else 'Z';
 input <= IO_pin;
 IC1: IC port map (input, output);
end bi_dir;
```

is of mode **inout**. The concurrent statements in the architecture connect the IC output to the pin via a tri-state buffer and also connect the pin to the IC input.

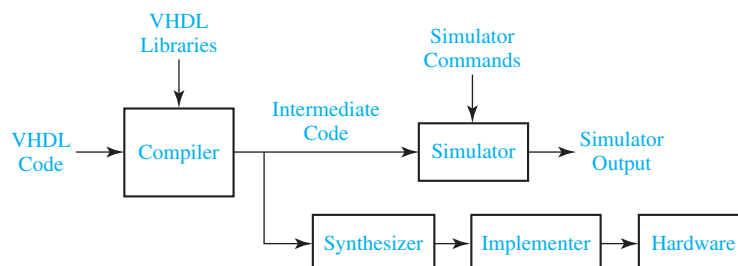
## 10.9 Compilation and Simulation of VHDL Code

After describing a digital system in VHDL, simulation of the VHDL code is important for two reasons. First, we need to verify the VHDL code correctly implements the intended design, and second, we need to verify that the design meets its specifications. Before the VHDL model of a digital system can be simulated, the VHDL code must first be compiled (see Figure 10-21). The VHDL compiler, also called an analyzer, first checks the VHDL source code to see that it conforms to the syntax and semantic rules of VHDL. If there is a syntax error such as a missing semicolon or a semantic error such as trying to add two signals of incompatible types, the compiler will output an error message. The compiler also checks to see that references to libraries are correct. If the VHDL code conforms to all of the rules, the compiler generates intermediate code which can be used by a simulator or by a synthesizer.

In preparation for simulation, the VHDL intermediate code must be converted to a form which can be used by the simulator. This step is referred to as *elaboration*. During elaboration, ports are created for each instance of a component, memory storage is allocated for the required signals, the interconnections among the port signals are specified, and a mechanism is established for executing the VHDL statements in the proper sequence. The resulting data structure represents the digital system being simulated. After an initialization phase, the simulator enters the execution phase. The simulator accepts simulation commands which control the simulation of the digital system and specify the desired simulator output.

Understanding the role of the delta ( $\Delta$ ) time delays is important when interpreting output from a VHDL simulator. Although the delta delays do not show up on waveform outputs from the simulator, they show up on listing outputs. The simulator uses delta delays to make sure that signals are processed in the proper sequence. Basically, the simulator works as follows: Whenever a component input changes, the output is scheduled to change after the specified delay or after  $\Delta$  if no delay is specified. When all input changes have been processed, the simulated time is advanced to the next time at which an output change is specified. When time is advanced by a finite amount (1 ns for example), the  $\Delta$  counter is reset, and simulation resumes. Real time does not advance again until all  $\Delta$  delays associated with the current simulation time have been processed.

**FIGURE 10-21**  
Compilation,  
Simulation, and  
Synthesis of VHDL  
Code

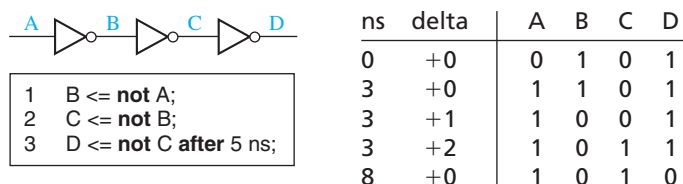


The following example illustrates how the simulator works for the circuit of Figure 10-22. Suppose that A changes at time = 3 ns. Statement 1 executes, and B is scheduled to change at time  $3 + \Delta$ . Then time advances to  $3 + \Delta$ , and statement 2 executes. C is scheduled to change at time  $3 + 2\Delta$ . Time advances to  $3 + 2\Delta$ , and statement 3 executes. D is then scheduled to change at 8 ns. You may think the change should occur at  $(3 + 2\Delta + 5)$  ns. However, when time advances a finite amount (as opposed to  $\Delta$ , which is infinitesimal), the  $\Delta$  counter is reset. For this reason, when events are scheduled a finite time in the future, the  $\Delta$ 's are ignored. Because no further changes are scheduled after 8 ns, the simulator goes into an idle mode and waits for another input change. The table gives the simulator output listing.

After the VHDL code for a digital system has been simulated to verify that it works correctly, the VHDL code can be synthesized to produce a list of required components and their interconnections. The synthesizer output can then be used to implement the digital system using specific hardware such as a CPLD or FPGA. The CAD software used for implementation generates the necessary information to program the CPLD or FPGA hardware. The synthesis and implementation of digital logic from VHDL code is discussed in more detail in Unit 17.

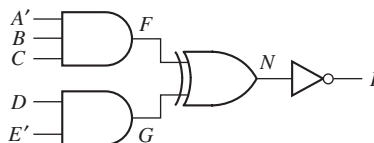
In this chapter, we have covered the basics of VHDL. We have shown how to use VHDL to model combinational logic and how to construct a VHDL module using an entity-architecture pair. Because VHDL is a hardware description language, it differs from an ordinary programming language in several ways. Most importantly, VHDL statements execute concurrently because they must model real hardware in which the components are all in operation at the same time.

**FIGURE 10-22**  
Simulation of  
VHDL Code



## Problems

- 10.1** Write VHDL statements that represent the following circuit:
- Write a statement for each gate.
  - Write one statement for the whole circuit.



- 10.2** Draw the circuit represented by the following VHDL statements:
- ```

F <= E and I;
I <= G or H;
G <= A and B;
H <= not C and D;

```
- 10.3** (a) Implement the following VHDL conditional statement using two 2-to-1 MUXes:
 $F \leq A \text{ when } D = '1' \text{ else } B \text{ when } E = '1' \text{ else } C;$
(b) Implement the same statement using gates.
- 10.4** Write the VHDL code for Figure 9-4 using a conditional signal assignment statement. Use bit_vectors for X, Y, and Z.
- 10.5** Write a VHDL module that implements a full adder using an array of bit_vectors to represent the truth table.
- 10.6** (a) Given that A = "00101101" and B = "10011", determine the value of F:
 $F \leq \text{not } B \& "0111" \text{ or } A \& '1' \text{ and } '1' \& A;$
(b) Given A = "11000", B = "10011", and C = "0111", evaluate the following expression:
 $\text{not } A + C * 2 > B / 4 \& "00"$
- 10.7** Write a VHDL module that finds the average value of four 16-bit unsigned numbers that are represented by std_logic_vectors. Division by four is best accomplished by shifting. Round off your answer to the nearest integer.
- 10.8** Write VHDL code for the system shown in Figure 9-11. Use four concurrent statements to compute the signal on the tri-state bus.
- 10.9** (a) Draw the circuit represented by the following VHDL statements:
- ```

T1 <= not A and not B and I0;
T2 <= not A and B and I1;
T3 <= A and not B and I2;
T4 <= A and B and I3;
F <= T1 or T2 or T3 or T4;

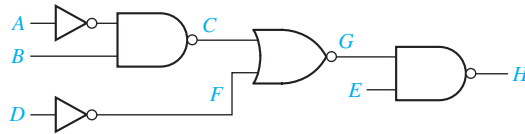
```
- (b) Draw a MUX that implements F. Then write a selected signal assignment statement that describes the MUX.
- 10.10** Assume that the following are concurrent VHDL statements:
- $L \leq P \text{ nand } Q \text{ after } 10 \text{ ns};$
  - $M \leq L \text{ nor } N \text{ after } 5 \text{ ns};$
  - $R \leq \text{not } M;$

Initially at time  $t = 0 \text{ ns}$ ,  $P = 1$ ,  $Q = 1$ , and  $N = 0$ . If Q becomes 0 at time  $t = 4 \text{ ns}$ ,

- At what time will statement (a) execute?
- At what time will L be updated?

- (3) At what time will statement (c) execute?  
 (4) At what time will R be updated?

- 10.11** (a) Write a single concurrent VHDL statement to represent the following circuit. Do not use parentheses in the statement.



- (b) Write individual statements to represent the circuit of part (a). Assume that all NAND gates have a delay of 10 ns, all NOR gates have a delay of 15 ns, and inverters have a delay of 5 ns.
- 10.12** Draw a circuit that implements the following VHDL code.
- ```

V <= T and U;
U <= not R or S and P or not Q or S;
T <= not P or Q or R;
  
```
- 10.13** Suppose L, M, and N are of type std_logic. If the following are concurrent statements, what are the values of L, M, and N? You can use the resolution function given in Figure 10-18.
- ```

L <= '1'; L <= '0';
M <= '1' when L = '0' else 'Z' when L = '1' else '0';
N <= M when L = '0' else not M;
N <= 'Z';

```
- 10.14** (a) Given that D = “011001” and E = “110”, determine the value of F.
- ```

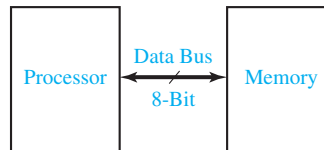
F <= not E & "011" or "000100" and not D;
  
```
- (b) Given A = “101” and B = “011”, evaluate the following expression:
- ```

not (A & B) < (not B & A and not A & A)

```
- 10.15** Write VHDL code to implement the following logic functions using a 16 words × 3 bits ROM.
- ```

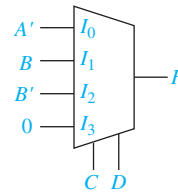
W = A'B'C + C'D + ACD'
X = A'C' + B'D
Y = BD' + B'C'D
  
```
- 10.16** The diagram shows an 8-bit-wide data bus that transfers data between a micro-processor and memory. Data on this bus is determined by the control signals mRead and mWrite. When mRead = ‘1’, the data on the memory’s internal bus ‘membus’ is output to the data bus. When mWrite = ‘1’, the data on the processor’s internal bus

'probus' is output to the data bus. When both control signals are '0', the data bus must be in a high-impedance state.

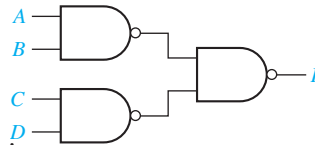


- (a) Write VHDL statements to represent the data bus.
- (b) Normally $mRead = mWrite = '1'$ does not occur. But if it occurs, what value will the data bus take?

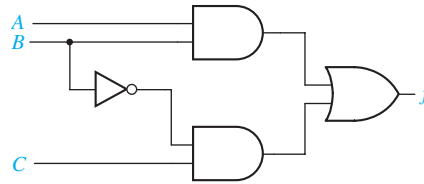
- 10.17** (a) Write a selected signal assignment statement to represent the 4-to-1 MUX shown below. Assume that there is an inherent delay in the MUX that causes the change in output to occur 15 ns after a change in input.
- (b) Repeat (a) using a conditional signal assignment statement.



- 10.18** (a) Write a complete VHDL module for a two-input NAND gate with 4-ns delay.
- (b) Write a complete VHDL module for the following circuit that uses the NAND gate module of Part (a) as a component.

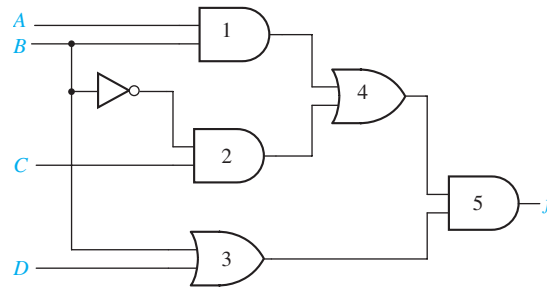


- 10.19** In the following circuit, all gates, including the inverter, have an inertial delay of 10 ns.
- (a) Write VHDL code that gives a dataflow description of the circuit. All delays should be inertial delays.
 - (b) Using the Direct VHDL simulator simulate the circuit. (Use a View Interval of 100 ns.) Initially set $A = 1$, $B = 1$ and $C = 1$, then run the simulator for 40 ns. Change B to 0, and run the simulator for 40 ns. Record the waveform.
 - (c) Change the VHDL code of Part (a) so that the inverter has a delay of 5 ns.
 - (d) Repeat Part (b).
 - (e) Change the VHDL code of Part (c) so that the output OR gate has a transport delay rather than an inertial delay.
 - (f) Repeat Part (b).
 - (g) Explain any differences between the waveforms for Parts (b), (d), and (f).



10.20 In the following circuit, all gates, including the inverter, have an inertial delay of 10 ns except for gate 3, which has delay 40 ns.

- Write VHDL code that gives a dataflow description of the circuit. All delays should be inertial delays.
- Using the Direct VHDL simulator simulate the circuit. (Use a View Interval of 150 ns.) Initially set $A = 1$, $B = 1$, $C = 1$ and $D = 0$, then run the simulator for 60 ns. Change B to 0, and run the simulator for 60 ns. Record the waveform.
- Change the VHDL code of Part (a) so that the inverter has a delay of 5 ns.
- Repeat Part (b).
- Change the VHDL code of Part (c) so that gates 4 and 5 have a transport delay rather than an inertial delay.
- Repeat Part (b).
- Explain any differences between the waveforms for Parts (b), (d), and (f).



10.21 Write VHDL code that gives a behavioral description of a circuit that converts the representation of decimal digits in BCD to the representation using the 2-4-2-1 weighted code, as follows:

Digit	2421 code
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

For the six input combinations that do not represent valid BCD digits, the circuit output should be “XXXX”. Make the inputs and outputs of type `std_logic`.

- (a) Write the code using the **when else** assignment statement.
- (b) Use the VHDL simulator to verify the code of Part (a) for the inputs $x = 0100$, 0101 , 1001 , and 1010 .
- (c) Write the code using the **with select when** assignment statement.
- (d) Use the VHDL simulator to verify the code of Part (c) for the inputs $x = 0100$, 0101 , 1001 , and 1010 .

10.22 Write VHDL code that gives a behavioral description of a circuit that converts the representation of decimal digits in the weighted code with weights 8, 4, -2 and -1 to the representation using the excess-3 code.

- (a) Write the code using the **when else** assignment statement.
- (b) Use the VHDL simulator to verify the code of Part (a) for the inputs $x = 0011$, 0100 , 1001 , and 1010 .
- (c) Write the code using the **with select when** assignment statement.
- (d) Use the VHDL simulator to verify the code of Part (c) for the inputs $x = 0100$, 0101 , 1001 , and 1010 .

Design Problems

10.A (a) Design a 4-to-1 MUX using only three 2-to-1 MUXes. Write an entity-architecture pair to implement a 2-to-1 MUX. Then write an entity-architecture pair to implement a 4-to-1 MUX using three instances of your 2-to-1 MUX.

[Hint: The equation for a 4-to-1 MUX can be rewritten as

$$F = A' (I_0B' + I_1B) + A (I_2B' + I_3B)].$$

Use the following port definitions:

For the 2-to-1 MUX:

```
port (i0, i1: in bit; sel: in bit; z: out bit);
```

For the 4-to-1 MUX:

```
port (i0, i1, i2, i3: in bit; a, b: in bit; f: out bit);
```

- (b) Simulate your code and test it using the following inputs:
 $I_0 = I_2 = 1, I_1 = I_3 = 0, AB = 00, 01, 11, 10$

10.B (a) Show how a BCD to Gray code converter can be designed using a $16 \text{ words} \times 4 \text{ bits}$ ROM. Then write an entity-architecture pair to implement the converter using the ROM. For your code to function correctly, you will need to add the following two lines of code to the top of your program.

```
library BITLIB;
```

```
use BITLIB.bit_pack.all;
```

Use the port definition specified below for the ROM:

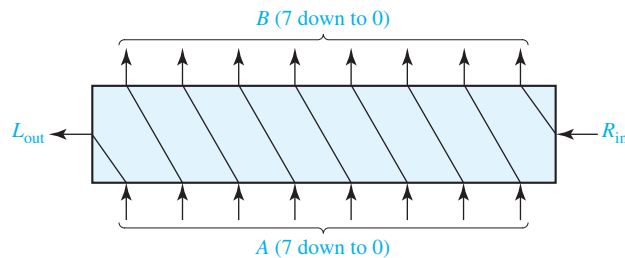
```
port (bcd: in bit_vector (3 downto 0);
```

```
gray: out bit_vector (3 downto 0));
```

- (b) Simulate your code and test it using the following inputs:
BCD = 0010, 0101, 1001
- 10.C** (a) A half adder is a circuit that can add two bits at a time to produce a sum and a carry. Design a half adder using only two gates. Write an entity-architecture pair to implement the half adder. Now write an entity-architecture pair to implement a full adder using two instances of your half adder and an OR gate. Use the port definitions specified below:
For the half adder: **port (a, b: in bit; s, c: out bit);**
For the full adder: **port (a, b, cin: in bit; sum, cout: out bit);**
- (b) Simulate your code and test it using the following inputs:
a b cin = 0 0 1, 0 1 1, 1 1 1, 1 1 0, 1 0 0
- 10.D** (a) Using a 3-to-8 decoder and two four-input OR gates, design a circuit that has three inputs and a 2-bit output. The output of the circuit represents (in binary form) the number of 1's present in the input. For example, when the input is $ABC = 101$, the output will be Count = 10. Write an entity-architecture pair to implement a 3-to-8 decoder. Then write an entity-architecture pair for your circuit, using the decoder as a component. Use the port definitions specified below.
For the 3-to-8 decoder:
port (a, b, c: in bit;
 y0, y1, y2, y3, y4, y5, y6, y7: out bit);
For the main circuit: **port (a, b, c: in bit; count: out bit_vector (1 downto 0));**
- (b) Simulate your code and test it using the following inputs:
a b c = 0 0 0, 0 1 0, 1 1 0, 1 1 1, 0 1 1
- 10.E** (a) Show how a BCD to seven-segment LED code converter can be designed, using a 16 words \times 7 bits ROM. Then write an entity-architecture pair to implement the converter using the ROM. Use the vec2int function in BITLIB for this problem. Use the port definition specified below for the ROM:
port (bcd: in bit_vector (3 downto 0);
 seven: out bit_vector (6 downto 0));
- (b) Simulate your code and test it using the following inputs:
BCD = 0000, 0001, 1000, 1001
- 10.F** (a) Using a 3-to-8 decoder, two three-input OR gates, and one two-input OR gate, design a circuit that has three inputs and a 1-bit output. The output of the circuit is 1 when the input 3-bit number is less than 3 or is greater than 4. Write an entity-architecture pair to implement a 3-to-8 decoder. Then write an entity-architecture pair for your circuit using the decoder as a component. Use the port definitions specified below.
For the 3-to-8 decoder:
port (a, b, c: in bit;
 y0, y1, y2, y3, y4, y5, y6, y7: out bit);
For the main circuit:
port (a, b, c: in bit; output : out bit);

- (b) Simulate your code and test it using the following inputs:
 $a\ b\ c = 0\ 0\ 0, 1\ 0\ 0, 1\ 0\ 1, 0\ 0\ 1, 0\ 1\ 1$

- 10.G** (a) Write the VHDL code for a full subtracter, using logic equations. Assume that the full subtracter has a 5-ns delay.
 (b) Write the VHDL code for a 4-bit subtracter using the module defined in (a) as a component.
 (c) Simulate your code and test it using the following inputs:
 $1100 - 0101, 0110 - 1011$
- 10.H** (a) The diagram shows an 8-bit shifter that shifts its input one place to the left. Write a VHDL module for the shifter.



- (b) Write a VHDL module that multiplies an 8-bit input (C) by 101_2 to give a 11-bit product (D). This can be accomplished by shifting C two places to the left and adding the result to C. Use two of the modules written in (a) as components and an overloaded operator for addition.
- (c) Simulate your code and test it using the following inputs:
 $10100101 \quad 11111111$
- 10.I** (a) Design a 4-to-2 priority encoder using gates [see Unit 9, Study Guide, Part 4(b)]. Write a VHDL module for your encoder. Use the port declaration
Port (y : **in** std_logic_vector(0 to 3);
 a1,b1,c1: **out** std_logic);
- (b) Design an 8-to-3 priority encoder (Figure 9-16), using two instances of the 4-to-2 priority encoder you designed, two 2-to-1 multiplexers, and one OR gate. Write a VHDL module for the 8-to-3 encoder. Use the port declaration
Port (y : **in** std_logic_vector(0 to 7);
 a,b,c,d : **out** std_logic);
 (Hint: In building the 8-to-3 encoder, use one 4-to-2 encoder for the four most significant bits, and the other for the four least significant bits. Outputs b and c of the 8-to-3 encoder should come from the multiplexers.)
- (c) Simulate your code and test it using the following inputs:
 $00000000, 10000000, 11000000, ---, 11111111$
- 10.J** (a) Write a VHDL module for a 4-bit adder, with a carry-in and carry-out, using an overloaded addition operator and std_logic_vector inputs and outputs.

- (b) Design an 8-bit subtracter with a borrow-out, using two of the 4-bit adders you designed in (a), along with any necessary gates or inverters. Write a VHDL module for the subtracter.
- (c) Simulate your code and test it using the following inputs:
 $11011011 - 01110110$, $01110110 - 11011011$
- 10.K** (a) Write a VHDL module for a tri-state buffer, with 6-bit data inputs and outputs and one control input.
- (b) Design a 4-to-1 multiplexer with 6-bit data inputs and outputs and two control inputs. Use four tri-state buffers from part (a) and a 2-to-4 decoder.
- (c) Simulate your code and test it for the following data inputs:
 000111 , 101010 , 111000 , 010101
- 10.L** (a) Write a VHDL module for a ROM with four inputs and three outputs. The 3-bit output should be a binary number equal to the number of 1's in the ROM input.
- (b) Write a VHDL module for a circuit that counts the number of 1's in a 12-bit number. Use three of the modules from (a) along with overloaded addition operators.
- (c) Simulate your code and test it for the following data inputs:
 111111111111 , 010110101101 , 100001011100
- 10.M** (a) Write a VHDL module for a full subtracter using a ROM to implement the truth table.
- (b) Write a VHDL module for a 3-bit subtracter using the module defined in part (a). Your module should have a borrow-in and a borrow-out.
- (c) Simulate your code and test it for the following data:
 $110 - 010$ with a borrow input of 1
 $011 - 101$ with a borrow input of 0
- 10.N** (a) Design a 4-to-2 priority encoder with an enable input, using gates. (See Unit 9, Study Guide Part 4(b)). When enable is 0, all outputs are 0. Write a VHDL module for the encoder. Use the following port declaration:
Port (*y* : **in** std_logic_vector(0 to 3);
enable : **in** std_logic; a1,b1,c1 : **out** std_logic);
- (b) Design an 8-to-3 priority encoder (Figure 9-16) with an enable input, using two of the 4-to-2 priority encoders you designed in (a), three OR gates, an AND gate, and one inverter. Then write a VHDL module for this encoder. Use the port declaration:
Port (*y* : **in** std_logic_vector(0 to 7);
main_enable : **in** std_logic; a,b,c,d : **out** std_logic);
(*Hint*: In building the 8-to-3 encoder, use one 4-to-2 encoder for the four most significant bits, and another for the four least significant bits. Also, outputs b and c of the 8-to-3 encoder should come from OR gates. The enable input to the encoder for the least significant bits depends on the main_enable signal and the c1 output from the encoder for the most significant bits.)
- (c) Simulate your code and test it using the following inputs:
 00000000 , 10000000 , 11000000 , ---, 11111111

Objectives

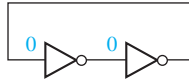
In this unit you will study one of the basic building blocks used in sequential circuits—the flip-flop. Some of the basic analysis techniques used for sequential circuits are introduced here. In particular, you will learn how to construct timing diagrams which show how each signal in the circuit varies as a function of time. Specific objectives are:

1. Explain in words the operation of S-R and gated D latches.
2. Explain in words the operation of D, D-CE, S-R, J-K, and T flip-flops.
3. Make a table and derive the characteristic (next-state) equation for such latches and flip-flops. State any necessary restrictions on the input signals.
4. Draw a timing diagram relating the input and output of such latches and flip-flops.
5. Show how latches and flip-flops can be constructed using gates. Analyze the operation of a flip-flop that is constructed of gates and latches.

Study Guide

1. Review Section 8.3, *Gate Delays and Timing Diagrams*. Then study Section 11.1, *Introduction*.

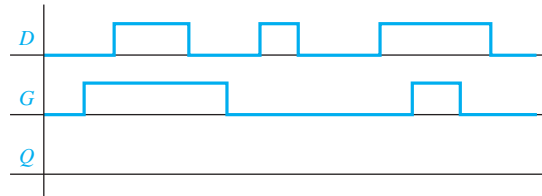
- (a) In the circuit shown, suppose that at some instant of time the inputs to both inverters are 0. Is this a stable condition of the circuit?



Assuming that the output of the left inverter changes before the output of the right inverter, what stable state will the circuit reach? (Indicate 0's and 1's on the inverters' inputs and outputs.)

- (b) Work Problem 11.1.
2. Study Section 11.2, *Set-Reset Latch*.
 - (a) Build an S-R latch in *SimUaid*, using NOR gates as in Figure 11-3. Place switches on the inputs and probes on the outputs. Experiment with it. Describe in words the behavior of your S-R latch.
 - (b) For Figure 11-4(b), what values would P and Q assume if $S = R = 1$?
 - (c) What restriction is necessary on S and R so that the two outputs of the S-R latch are complements?
 - (d) State in words the meaning of the equation $Q^+ = S + R'Q$.
 - (e) Starting with $Q = 0$ and $\bar{S} = \bar{R} = 1$ in Figure 11-10(a), change \bar{S} to 0 and trace signals through the latch until steady-state is reached. Then, change \bar{S} to 1 and \bar{R} to 0 and trace again.
 - (f) Work Problems 11.2 and 11.3.
 3. Study Section 11.3, *Gated D Latch*.
 - (a) Build a gated D latch in *SimUaid*. See Figure 11-11. (Construct the S-R latch as in Study Guide Section 2(a).) Place switches on the inputs and probes on the outputs. Experiment with it. Describe in words the behavior of your gated D latch.
 - (b) State in words the meaning of the equation $Q^+ = G'Q + GD$.

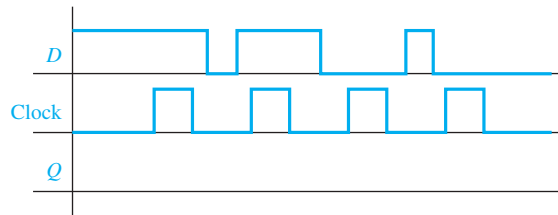
- (c) Given a gated D latch with the following inputs, sketch the waveform for Q .



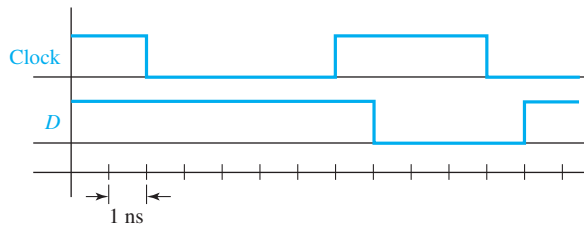
- (d) Work Problem 11.4.

4. Study Section 11.4, *Edge-Triggered D Flip-Flop*.

- (a) Experiment with a D flip-flop in *SimUaid*. Use the D flip-flop on the parts menu. Place switches on the inputs and probes on the outputs. Describe in words the behavior of your D flip-flop.
- (b) Given a rising-edge-triggered D flip-flop with the following inputs, sketch the waveform for Q .



- (c) Work Programmed Exercise 11.29.
- (d) A D flip-flop with a falling-edge trigger is behaving erratically. It has a setup time of 2 ns and a hold time of 2 ns. The figure shows the inputs to the flip-flop over a typical clock cycle. Why might the flip-flop be behaving erratically?



- (e) Suppose that for the circuit of Figure 11-17, new semiconductor technology has allowed us to improve the delays and setup times. The propagation delay of the new inverter is 1.5 ns, and the propagation delay and setup times of the new flip-flop are 3.5 ns and 2 ns, respectively. What is the shortest clock period for the circuit of Figure 11-17(a) which will not violate the timing constraints?
- (f) Work Problem 11.5.

5. Study Section 11.5, *S-R Flip-Flop*.

- (a) Describe in words the behavior of an S-R flip-flop.

- (b) Trace signals through the circuit of Figure 11-19(a) and verify the timing diagram of Figure 11-19(b).
- (c) What is the difference between a master-slave flip-flop and an edge-triggered flip-flop? Assume that Q changes on the rising clock edge in both cases.

(d) Work Problem 11.6.

6. Study Section 11.6, *J-K Flip-Flop*.

- (a) Experiment with a J - K flip-flop in SimUaid. Use the J - K flip-flop in the parts menu. Place switches on the inputs and probes on the outputs. Describe in words the behavior of your J - K flip-flop.

(b) Derive the next-state equation for the J-K flip-flop.

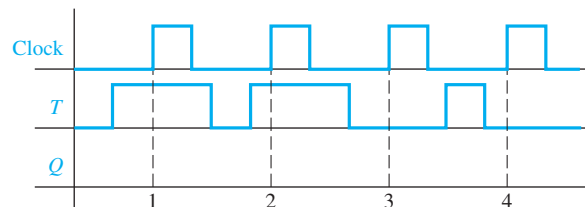
- (c) Examine Figures 11-19(a) and 11-21. Construct a J-K flip-flop, using a master-slave S-R flip-flop and two AND gates. (Do not draw the interior of the S-R flip-flop. Just use the symbol in Figure 11-18.)

(d) Work Problem 11.7.

7. Study Section 11.7, *T Flip-Flop*.

- (a) Construct a T flip-flop in *SimUaid* from a D flip-flop as in Figure 11-24(b). Place switches on the inputs and probes on the outputs. Experiment with it. Describe in words the behavior of the T flip-flop.

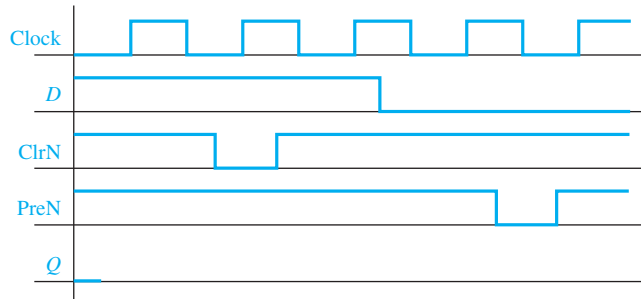
(b) Complete the following timing diagram (assume that $Q = 0$ initially):



8. Study Section 11.8, *Flip-Flops with Additional Inputs*.

- (a) To set the flip-flop of Figure 11-25 to $Q = 1$ without using the clock, the Cln input should be set to _____ and the PreN input to _____. To reset this flip-flop to $Q = 0$ without using the clock, the _____ input should be set to _____ and the _____ input to _____.

- (b) Complete the following timing diagram for a rising-edge-triggered D flip-flop with ClrN and PreN inputs. Assume Q begins at 0.



- (c) In Figure 11-27(a), what would happen if En changed from 1 to 0 while $CLK = 1$?
 What if En changed when $CLK = 0$?
 In order to have Q change synchronization with the clock, what restriction must be placed on the time at which En can change?

Why does this restriction not apply to Figures 11-27(b) and (c)?

- (d) Make a table similar to Figure 11-25(b) that describes the operation of a D flip-flop with a falling-edge clock input, a clock enable input, and an asynchronous active-low clear input (ClrN), but no preset input.

- (e) Work Problems 11.8 and 11.9.

9. Study Section 11.9, *Summary*.

- (a) Given one of the flip-flops in this chapter or a similar flip-flop, you should be able to derive the characteristic equation which gives the next state of the flip-flop in terms of the present state and inputs. You should understand the meaning of each of the characteristic equations given in Section 11.9.
- (b) An S-R flip-flop can be converted to a T flip-flop by adding gates at the S and R inputs. The S and R inputs must be chosen so that the flip-flop will change state whenever $T = 1$ and the clock is pulsed. In order to determine the S and R inputs, ask yourself the question, "Under what conditions must the flip-flop be set to 1, and under what conditions must it be reset?" The flip-flop must be set to 1 if $Q = 0$ and $T = 1$.

Therefore, $S = \underline{\hspace{2cm}}$. In a similar manner, determine the equation for R and draw the circuit which converts an S-R flip-flop to a T flip-flop.

- (c) Work Problem 11.10.
10. When you are satisfied that you can meet the objectives of this unit, take the readiness test.



Latches and Flip-Flops

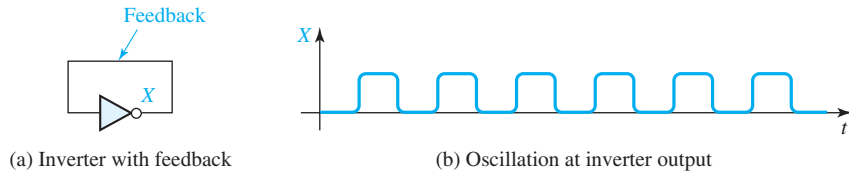
11.1 Introduction

Sequential switching circuits have the property that the output depends not only on the present input but also on the past sequence of inputs. In effect, these circuits must be able to “remember” something about the past history of the inputs in order to produce the present output. Latches and flip-flops are commonly used memory devices in sequential circuits. Basically, latches and flip-flops are memory devices which can assume one of two stable output states and which have one or more inputs that can cause the output state to change. Several common types of latches and flip-flops are described in this unit.

In Units 12 through 16, we will discuss the analysis and design of synchronous digital systems. In such systems, it is common practice to synchronize the operation of all flip-flops by a common clock or pulse generator. Each of the flip-flops has a clock input, and the flip-flops can only change state in response to a clock pulse. The use of a clock to synchronize the operation of several flip-flops is illustrated in Units 12 and 13. A memory element that has no clock input is often called a latch, and we will follow this practice. We will then reserve the term flip-flop to describe a memory device that changes output state in response to a clock input and not in response to a data input.

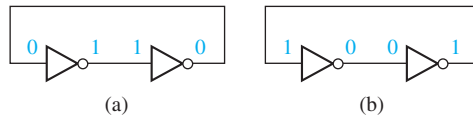
The switching circuits that we have studied so far have not had feedback connections. By feedback we mean that the output of one of the gates is connected back into the input of another gate in the circuit so as to form a closed loop. In order to construct a switching circuit that has memory, such as a latch or flip-flop, we must introduce feedback into the circuit. For example, in the NOR-gate circuit of Figure 11-3(a), the output of the second NOR gate is fed back into the input of the first NOR gate.

FIGURE 11-1



In simple cases, we can analyze circuits with feedback by tracing signals through the circuit. For example, consider the circuit in Figure 11-1(a). If at some instant of time the inverter input is 0, this 0 will propagate through the inverter and cause the output to become 1 after the inverter delay. This 1 is fed back into the input, so after the propagation delay, the inverter output will become 0. When this 0 feeds back into the input, the output will again switch to 1, and so forth. The inverter output will continue to oscillate back and forth between 0 and 1, as shown in Figure 11-1(b), and it will never reach a stable condition. The rate at which the circuit oscillates is determined by the propagation delay in the inverter.

FIGURE 11-2



Next, consider a feedback loop which has two inverters in it, as shown in Figure 11-2(a). In this case, the circuit has two stable conditions, often referred to as stable states. If the input to the first inverter is 0, its output will be 1. Then, the input to the second inverter will be 1, and its output will be 0. This 0 will feed back into the first inverter, but because this input is already 0, no changes will occur. The circuit is then in a stable state. As shown in Figure 11-2(b), a second stable state of the circuit occurs when the input to the first inverter is 1 and the input to the second inverter is 0.

11.2 Set-Reset Latch

We can construct a simple latch by introducing feedback into a NOR-gate circuit, as seen in Figure 11-3(a). As indicated, if the inputs are $S = R = 0$, the circuit can assume a stable state with $Q = 0$ and $P = 1$. Note that this is a stable condition of the circuit because $P = 1$ feeds into the second gate forcing the output to be $Q = 0$, and $Q = 0$ feeds into the first gate allowing its output to be 1. Now if we change S to 1, P will become 0. This is an unstable condition or state of the circuit because both the inputs and output of the second gate are 0; therefore Q will change to 1, leading to the stable state shown in Figure 11-3(b).

FIGURE 11-3

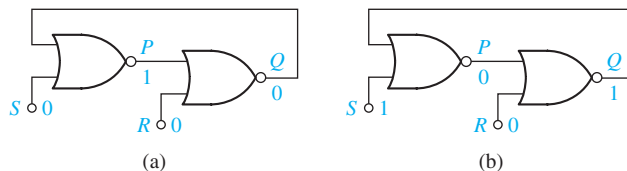
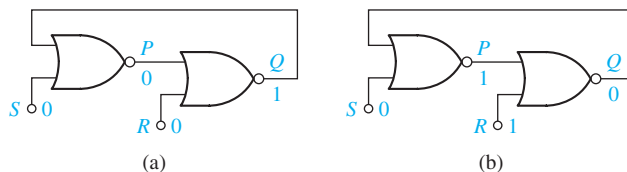


FIGURE 11-4



If S is changed back to 0, the circuit will not change state because $Q = 1$ feeds back into the first gate, causing P to remain 0, as shown in Figure 11-4(a). Note that the inputs are again $S = R = 0$, but the outputs are different than those with which we started. Thus, the circuit has two different stable states for a given set of inputs. If we now change R to 1, Q will become 0 and P will then change back to 1, as seen in Figure 11-4(b). If we then change R back to 0, the circuit remains in this state and we are back where we started.

This circuit is said to have memory because its output depends not only on the present inputs, but also on the past sequence of inputs. If we restrict the inputs so that $R = S = 1$ is not allowed, the stable states of the outputs P and Q are always complements, that is, $P = Q'$. To emphasize the symmetry between the operation of the two gates, the circuit is often drawn in *cross-coupled* form [see Figure 11-5(a)]. As shown in Figures 11-3(b) and 11-4(b), an input $S = 1$ sets the output to $Q = 1$, and an input $R = 1$ resets the output to $Q = 0$. When used with the restriction that R and S cannot be 1 simultaneously, the circuit is commonly referred to as a set-reset (S - R) latch and given the symbol shown in Figure 11-5(b). Note that although Q comes out of the NOR gate with the R input, the standard S - R latch symbol has Q directly above the S input.

If $S = R = 1$, the latch will not operate properly, as shown in Figure 11-6. The notation $1 \rightarrow 0$ means that the input is originally 1 and then changes to 0. Note that when S and R are both 1, P and Q are both 0. Therefore, P is not equal to Q' , and this violates a basic rule of latch operation that requires the latch outputs to be complements. Furthermore, if S and R are simultaneously changed back to 0, P and Q may both change to 1. If $S = R = 0$ and $P = Q = 1$, then after the 1's propagate

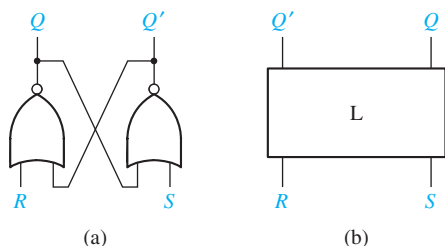
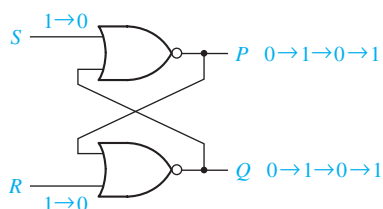
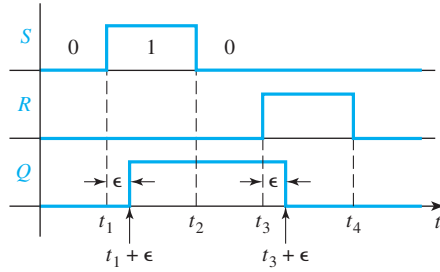
FIGURE 11-5
S-R LatchFIGURE 11-6
Improper S-R Latch
Operation

FIGURE 11-7
Timing Diagram
for S-R Latch



through the gates, P and Q will become 0 again, and the latch may continue to oscillate if the gate delays are equal.

Figure 11-7 shows a timing diagram for the S-R latch. Note that when S changes to 1 at time t_1 , Q changes to 1 a short time (ϵ) later. (ϵ represents the response time or delay time of the latch.) At time t_2 , when S changes back to 0, Q does not change. At time t_3 , R changes to 1, and Q changes back to 0 a short time (ϵ) later. The duration of the S (or R) input pulse must normally be at least as great as ϵ in order for a change in the state of Q to occur. If $S = 1$ for a time less than ϵ , the gate output will not change and the latch will not change state.

When discussing latches and flip-flops, we use the term *present state* to denote the state of the Q output of the latch or flip-flop at the time any input signal changes, and the term *next state* to denote the state of the Q output after the latch or flip-flop has reacted to the input change and stabilized. If we let $Q(t)$ represent the present state and $Q(t + \epsilon)$ represent the next state, an equation for $Q(t + \epsilon)$ can be obtained from the circuit by conceptually breaking the feedback loop at Q and considering $Q(t)$ as an input and $Q(t + \epsilon)$ as the output. Then for the S-R latch of Figure 11-3

$$Q(t + \epsilon) = R(t)'[S(t) + Q(t)] = R(t)'S(t) + R(t)'Q(t) \quad (11-1)$$

and the equation for output P is

$$P(t) = S(t)'Q(t)' \quad (11-2)$$

Normally we write the next-state equation without including time explicitly, using Q to represent the present state of the latch and Q^+ to represent the next state:

$$Q^+ = R'S + R'Q \quad (11-3)$$

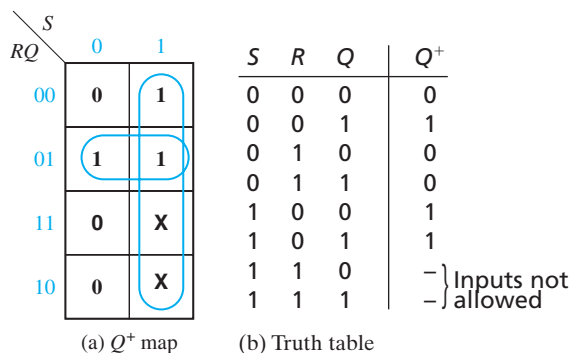
$$P = S'Q' \quad (11-4)$$

These equations are mapped in the next-state and output tables of Table 11-1. The stable states of the latch are circled. Note that for all stable states, $P = Q'$ except when $S = R = 1$. As discussed previously, this is one of the reasons why $S = R = 1$ is

TABLE 11-1
S-R Latch
Next State
and Output

Present State Q	Next State Q^+				Present Output P			
	SR 00	SR 01	SR 11	SR 10	SR 00	SR 01	SR 11	SR 10
0	0	0	0	1	1	1	0	0
1	1	0	0	1	0	0	0	0

FIGURE 11-8
Derivation of Q^+
for an S-R Latch



disallowed as an input combination to the S-R latch. Making $S = R = 1$ a don't-care combination allows simplifying the next-state equation, as shown in Figure 11-8(a). After plotting Equation (11-3) on the map and changing two entries to don't-cares, the next-state equation simplifies to

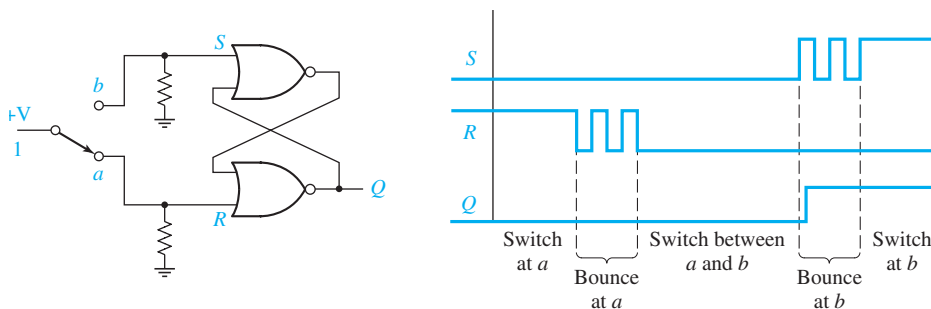
$$Q^+ = S + R'Q \quad (SR = 0) \quad (11-5)$$

In words, this equation tells us that the next state of the latch will be 1 either if it is set to 1 with an S input, or if the present state is 1 and the latch is not reset. The condition $SR = 0$ implies that S and R cannot both be 1 at the same time. An equation that expresses the next state of a latch in terms of its present state and inputs will be referred to as a *next-state equation*, or *characteristic equation*.

Another approach for deriving the characteristic equation for an S-R latch is based on constructing a truth table for the next state of Q . We previously discussed the latch operation by tracing signals through the gates, and the truth table in Figure 11-8(b) is based on this discussion. Plotting Q^+ on a Karnaugh map gives the same result as Figure 11-8(a).

The S-R latch is often used as a component in more complex latches and flip-flops and in asynchronous systems. Another useful application of the S-R latch is for debouncing switches. When a mechanical switch is opened or closed, the switch contacts tend to vibrate or bounce open and closed several times before settling down to their final position. This produces a noisy transition, and this noise can interfere with the proper operation of a logic circuit. The input to the switch in Figure 11-9 is connected to a logic 1 (+V). The pull-down resistors connected to contacts a and b assure that when the switch is between a and b the latch inputs S and R will always be at a logic 0, and the

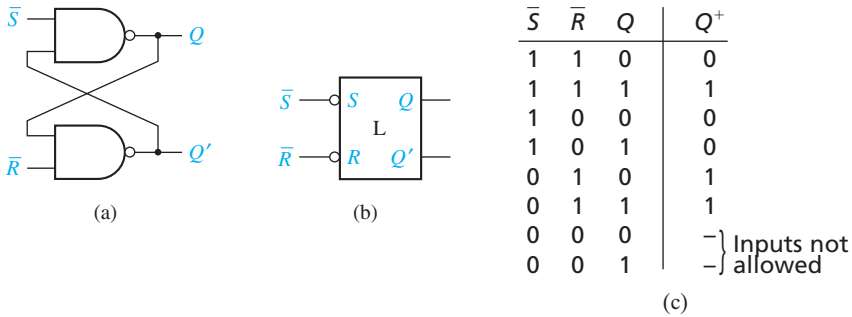
FIGURE 11-9
Switch Debouncing
with an S-R Latch



latch output will not change state. The timing diagram shows what happens when the switch is flipped from a to b . As the switch leaves a , bounces occur at the R input; when the switch reaches b , bounces occur at the S input. After the switch reaches b , the first time S becomes 1, after a short delay the latch switches to the $Q = 1$ state and remains there. Thus Q is free of all bounces even though the switch contacts bounce. This debouncing scheme requires a *double throw* switch that switches between two contacts; it will not work with a *single throw* switch that switches between one contact and open.

An alternative form of the S-R latch uses NAND gates, as shown in Figure 11-10. We will refer to this circuit as an \bar{S} - \bar{R} latch, and the table describes its operation. We have labeled the inputs to this latch \bar{S} and \bar{R} because $\bar{S} = 0$ will set Q to 1 and $\bar{R} = 0$ will reset Q to 0. If \bar{S} and \bar{R} are 0 at the same time, both the Q and Q' outputs are forced to 1. Therefore, for the proper operation of this latch, the condition $\bar{S} = \bar{R} = 0$ is not allowed.

FIGURE 11-10
 \bar{S} - \bar{R} Latch



11.3 Gated D Latch

A gated D latch (Figure 11-11) has two inputs—a data input (D) and a gate input (G). The D latch can be constructed from an S-R latch and gates (Figure 11-11(a)). When $G = 0$, $S = R = 0$, so Q does not change. When $G = 1$ and $D = 1$, $S = 1$ and $R = 0$, so Q is set to 1. When $G = 1$ and $D = 0$, $S = 0$ and $R = 1$, so Q is reset to 0. In other words, when $G = 1$, the Q output follows the D input, and when $G = 0$, the Q output holds the last value of D (no state change). This type of latch is also referred to as a transparent latch because when $G = 1$, the Q output is the same as the D input. From the truth table (Figure 11-12), the characteristic equation for the latch is $Q^+ = G'Q + GD$.

FIGURE 11-11
Gated D Latch

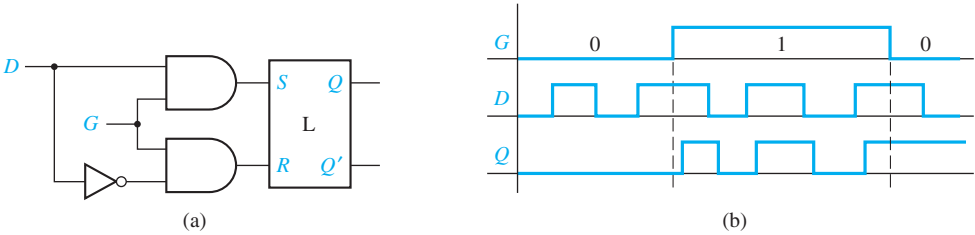
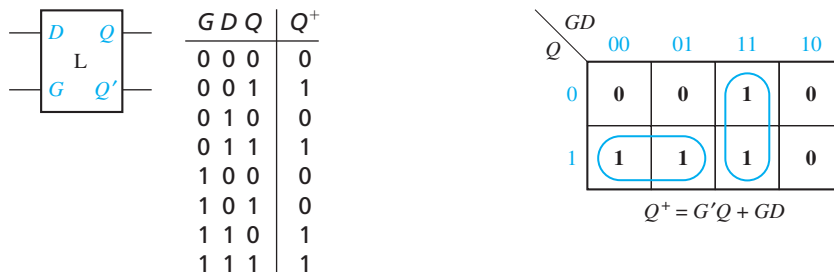


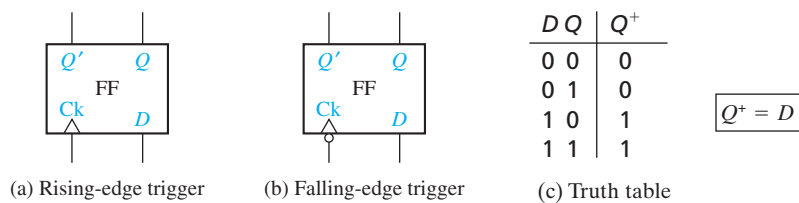
FIGURE 11-12
Symbol and Truth
Table for Gated
Latch



11.4 Edge-Triggered D Flip-Flop

A D flip-flop (Figure 11-13) has two inputs, D (data) and Ck (clock). The small arrowhead on the flip-flop symbol identifies the clock input. Unlike the D latch, the flip-flop output changes only in response to the clock, not to a change in D . If the output can change in response to a 0 to 1 transition on the clock input, we say that the flip-flop is triggered on the *rising edge* (or positive edge) of the clock. If the output can change in response to a 1 to 0 transition on the clock input, we say that the flip-flop is triggered on the *falling edge* (or negative edge) of the clock. An inversion bubble on the clock input indicates a *falling-edge trigger* (Figure 11-13(b)), and no bubble indicates a *rising-edge trigger* [Figure 11-13(a)]. The term *active edge* refers to the clock edge (rising or falling) that triggers the flip-flop state change.

FIGURE 11-13
D Flip-Flops



The state of a D flip-flop after the active clock edge (Q^+) is equal to the input (D) before the active edge. For example, if $D = 1$ before the clock pulse, $Q = 1$ after the active edge, regardless of the previous value of Q . Therefore, the characteristic equation is $Q^+ = D$. If D changes at most once following each clock pulse, the output of the flip-flop is the same as the D input, except that the output changes are delayed until after the active edge of the clock pulse, as illustrated in Figure 11-14.

FIGURE 11-14
Timing for
D Flip-Flop
(Falling-Edge
Trigger)

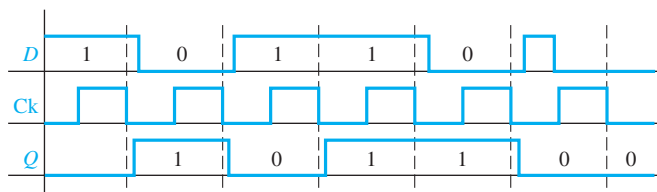
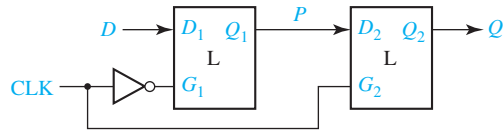
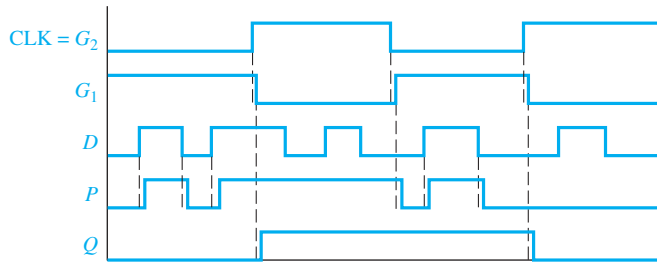


FIGURE 11-15**D Flip-Flop
(Rising-Edge
Trigger)**

(a) Construction from two gated D latches



(b) Timing analysis

A rising-edge-triggered D flip-flop can be constructed from two gated D latches and an inverter, as shown in Figure 11-15(a). The timing diagram is shown in Figure 11-15(b). When $\text{CLK} = 0$, $G_1 = 1$, and the first latch is *transparent* so that the P output follows the D input. Because $G_2 = 0$, the second latch holds the current value of Q . When CLK changes to 1, G_1 changes to 0, and the current value of D is stored in the first latch. Because $G_2 = 1$, the value of P flows through the second latch to the Q output. When CLK changes back to 0, the second latch takes on the value of P and holds it and, then, the first latch starts following the D input again. If the first latch starts following the D input before the second latch takes on the value of P , the flip-flop will not function properly. Therefore, the circuit designers must pay careful attention to timing issues when designing edge-triggered flip-flops. With this circuit, output state changes occur only following the rising edge of the clock. The value of D at the time of the rising edge of the clock determines the value of Q , and any extra changes in D that occur between rising clock edges have no effect on Q .

Because a flip-flop changes state only on the active edge of the clock, the propagation delay of a flip-flop is the time between the active edge of the clock and the resulting change in the output. However, there are also timing issues associated with the D input. To function properly, the D input to an edge-triggered flip-flop must be held at a constant value for a period of time before and after the active edge of the clock. If D changes at the same time as the active edge, the behavior is unpredictable. The amount of time that D must be stable before the active edge is called the setup time (t_{su}), and the amount of time that D must hold the same value after the active edge is the hold time (t_h). The times at which D is allowed to change during the clock cycle are shaded in the timing diagram of Figure 11-16. The propagation delay (t_p) from the time the clock changes until the Q output changes is also indicated. For Figure 11-15(a), the setup time allows a change in D to propagate through the first latch before the rising edge of Clock. The hold time is required so that D gets stored in the first latch before D changes.

Using these timing parameters, we can determine the minimum clock period for a circuit which will not violate the timing constraints. Consider the circuit of

FIGURE 11-16
Setup and Hold
Times for an
Edge-Triggered
D Flip-Flop

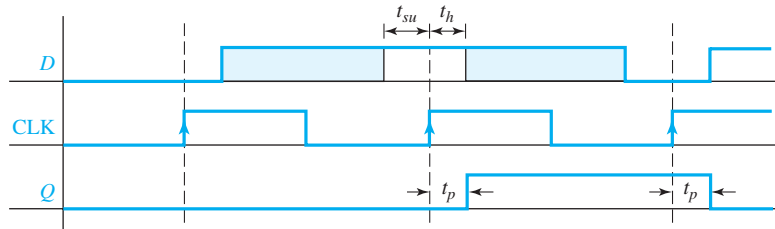
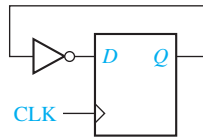


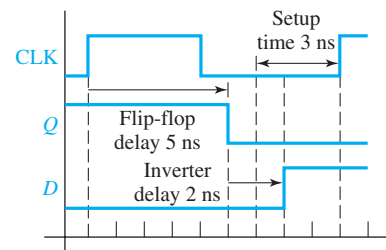
Figure 11-17(a). Suppose the inverter has a propagation delay of 2 ns, and suppose the flip-flop has a propagation delay of 5 ns and a setup time of 3 ns. (The hold time does not affect this calculation.) Suppose, as in Figure 11-17(b), that the clock period is 9 ns, i.e., 9 ns is the time between successive active edges (rising edges for this figure). Then, 5 ns after a clock edge, the flip-flop output will change, and 2 ns after that, the output of the inverter will change. Therefore, the input to the flip-flop will change 7 ns after the rising edge, which is 2 ns before the next rising edge. But the setup time of the flip-flop requires that the input be stable 3 ns before the rising edge; therefore, the flip-flop may not take on the correct value.

Suppose instead that the clock period were 15 ns, as in Figure 11-17(c). Again, the input to the flip-flop will change 7 ns after the rising edge. However, because the clock is slower, this is 8 ns before the next rising edge. Therefore, the flip-flop will work properly. Note in Figure 11-17(c) that there is 5 ns of extra time between the time the D input is correct and the time when it must be correct for the setup time to be satisfied. Therefore, we can use a shorter clock period, and have less extra time, or no extra time. Figure 11-17(d) shows that 10 ns is the minimum clock period which will work for this circuit.

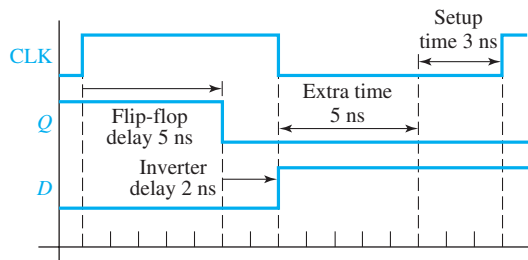
FIGURE 11-17
Determination of
Minimum Clock
Period



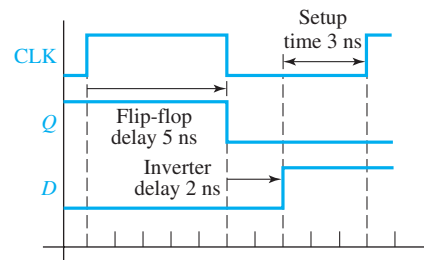
(a) Simple flip-flop circuit



(b) Setup time not satisfied



(c) Setup time satisfied



(d) Minimum clock period

11.5 S-R Flip-Flop

An S-R flip-flop (Figure 11-18) is similar to an S-R latch in that $S = 1$ sets the Q output to 1, and $R = 1$ resets the Q output to 0. The essential difference is that the flip-flop has a clock input, and the Q output can change only after an active clock edge. The truth table and characteristic equation for the flip-flop are the same as for the latch, but the interpretation of Q^+ is different. For the latch, Q^+ is the value of Q after the propagation delay through the latch, while for the flip-flop, Q^+ is the value that Q assumes after the active clock edge.

Figure 11-19(a) shows an S-R flip-flop constructed from two S-R latches and gates. This flip-flop changes state after the rising edge of the clock. The circuit is often referred to as a master-slave flip-flop. When $CLK = 0$, the S and R inputs set the outputs of the master latch to the appropriate value while the slave latch holds the previous value of Q . When the clock changes from 0 to 1, the value of P is held in the master latch and this value is transferred to the slave latch. The master latch holds the value of P while $CLK = 1$, and, hence, Q does not change. When the clock changes from 1 to 0, the Q value is latched in the slave, and the master can process new inputs. Figure 11-19(b) shows the timing diagram. Initially, $S = 1$ and Q changes to 1 at t_1 . Then $R = 1$ and Q changes to 0 at t_3 .

FIGURE 11-18
S-R Flip-Flop

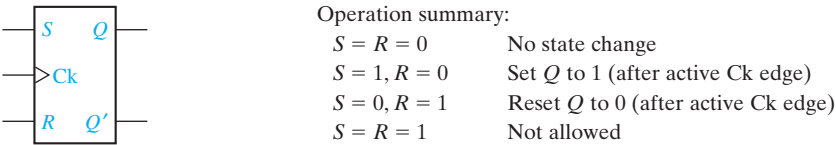
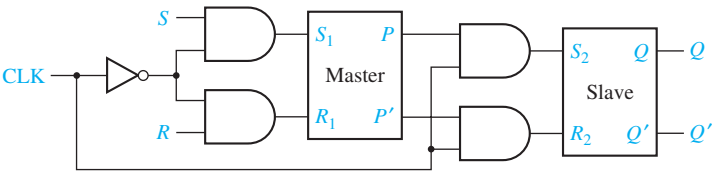
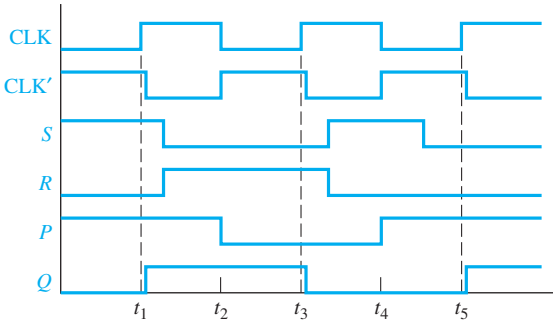


FIGURE 11-19
S-R Flip-Flop
Implementation
and Timing



(a) Implementation with two latches



(b) Timing analysis

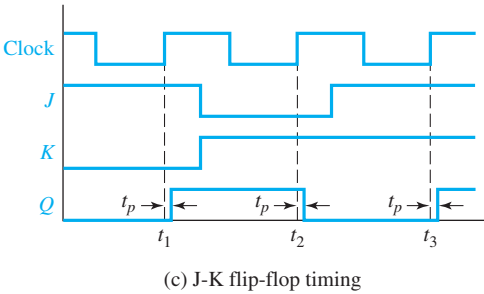
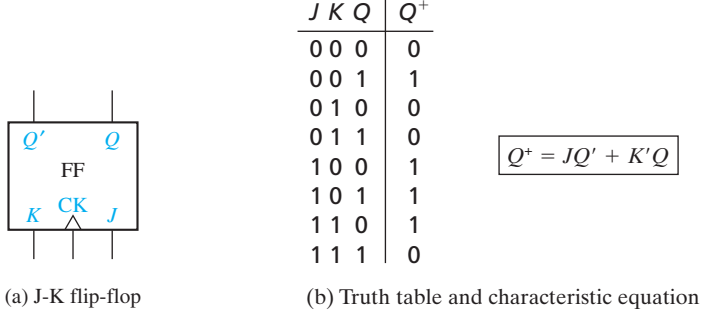
At first glance, this flip-flop appears to operate just like an edge-triggered flip-flop, but there is a subtle difference. For a rising-edge-triggered flip-flop the value of the inputs is sensed at the rising edge of the clock, and the inputs can change while the clock is low. For the master-slave flip-flop, if the inputs change while the clock is low, the flip-flop output may be incorrect. For example, in Figure 11-19(b) at t_4 , $S = 1$ and $R = 0$, so P changes to 1. Then S changes to 0 at t_5 , but P does not change, so at t_5 , Q changes to 1 after the rising edge of CLK. However, at t_5 , $S = R = 0$, so the state of Q should not change. We can solve this problem if we only allow the S and R inputs to change while the clock is high.

11.6 J-K Flip-Flop

The J-K flip-flop (Figure 11-20) is an extended version of the S-R flip-flop. The J-K flip-flop has three inputs— J , K , and the clock (CK). The J input corresponds to S , and K corresponds to R . That is, if $J = 1$ and $K = 0$, the flip-flop output is set to $Q = 1$ after the active clock edge; and if $K = 1$ and $J = 0$, the flip-flop output is reset to $Q = 0$ after the active edge. Unlike the S-R flip-flop, a 1 input may be applied simultaneously to J and K , in which case the flip-flop changes state after the active clock edge. When $J = K = 1$, the active edge will cause Q to change from 0 to 1, or from 1 to 0. The next-state table and characteristic equation for the J-K flip-flop are given in Figure 11-20(b).

Figure 11-20(c) shows the timing for a J-K flip-flop. This flip-flop changes state a short time (t_p) after the rising edge of the clock pulse, provided that J and K have

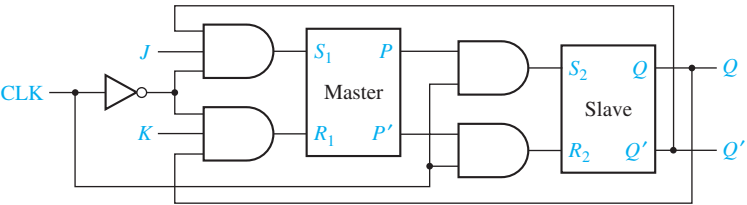
FIGURE 11-20
J-K Flip-Flop
(Q Changes on the Rising Edge)



appropriate values. If $J = 1$ and $K = 0$ when $\text{Clock} = 0$, Q will be set to 1 following the rising edge. If $K = 1$ and $J = 0$ when $\text{Clock} = 0$, Q will be set to 0 after the rising edge. Similarly, if $J = K = 1$, Q will change state after the rising edge. Referring to Figure 11-20(c), because $Q = 0$, $J = 1$, and $K = 0$ before the first rising clock edge, Q changes to 1 at t_1 . Because $Q = 1$, $J = 0$, and $K = 1$ before the second rising clock edge, Q changes to 0 at t_2 . Because $Q = 0$, $J = 1$, and $K = 1$ before the third rising clock edge, Q changes to 1 at t_3 .

One way to realize the J-K flip-flop is with two S-R latches connected in a master-slave arrangement, as shown in Figure 11-21. This is the same circuit as for the S-R master-slave flip-flop, except S and R have been replaced with J and K , and the Q and Q' outputs are feeding back into the input gates. Because $S = J \cdot Q' \cdot \text{Clk}'$ and $R = K \cdot Q \cdot \text{Clk}'$, only one of S and R inputs to the first latch can be 1 at any given time. If $Q = 0$ and $J = 1$, then $S = 1$ and $R = 0$, regardless of the value of K . If $Q = 1$ and $K = 1$, then $S = 0$ and $R = 1$, regardless of the value of J .

FIGURE 11-21
Master-Slave
J-K Flip-Flop
(Q Changes on
Rising Edge)



11.7 T Flip-Flop

The T flip-flop, also called the toggle flip-flop, is frequently used in building counters. Most CPLDs and FPGAs can be programmed to implement T flip-flops. The T flip-flop in Figure 11-22(a) has a T input and a clock input. When $T = 1$ the flip-flop changes state after the active edge of the clock. When $T = 0$, no state change occurs. The next-state table and characteristic equation for the T flip-flop are given in Figure 11-22(b). The characteristic equation states that the next state of the flip-flop (Q^+) will be 1 iff the present state (Q) is 1 and $T = 0$ or the present state is 0 and $T = 1$.

Figure 11-23 shows a timing diagram for the T flip-flop. At times t_2 and t_4 the T input is 1 and the flip-flop state (Q) changes a short time (t_p) after the falling edge of the clock pulse. At times t_1 and t_3 the T input is 0, and the clock edge does not cause a change of state.

FIGURE 11-22
T Flip-Flop

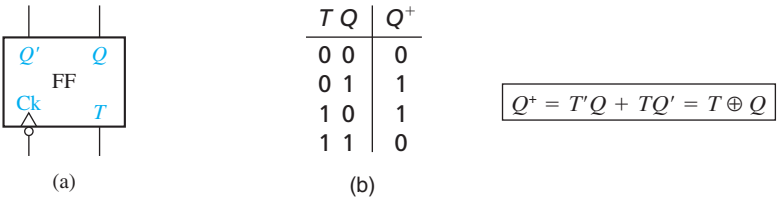


FIGURE 11-23
Timing Diagram
for T Flip-Flop
(Falling-Edge
Trigger)

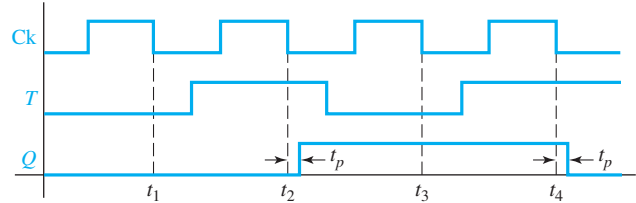
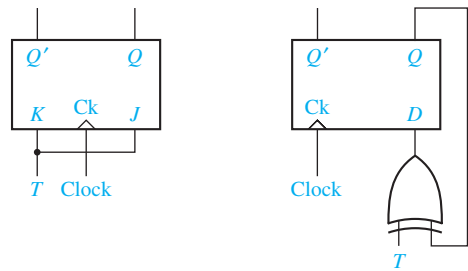


FIGURE 11-24
Implementation
of T Flip-Flops



(a) Conversion of J-K to T (b) Conversion of D to T

One way to implement a T flip-flop is to connect the J and K inputs of a J-K flip-flop together, as shown in Figure 11-24(a). Substituting T for J and K in the J-K characteristic equation gives

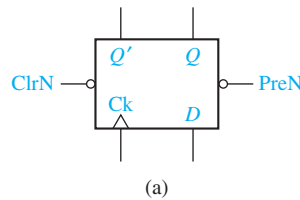
$$Q^+ = JQ' + K'Q = TQ' + T'Q$$

which is the characteristic equation for the T flip-flop. Another way to realize a T flip-flop is with a D flip-flop and an exclusive-OR gate [Figure 11-24(b)]. The D input is $Q \oplus T$, so $Q^+ = Q \oplus T = TQ' + T'Q$, which is the characteristic equation for the T flip-flop.

11.8 Flip-Flops with Additional Inputs

Flip-flops often have additional inputs which can be used to set the flip-flops to an initial state independent of the clock. Figure 11-25 shows a D flip-flop with clear and preset inputs. The small circles (inversion symbols) on these inputs indicate that a logic 0 (rather than a 1) is required to clear or set the flip-flop. This type of input is often referred to as *active-low* because a low voltage or logic 0 will activate the clear

FIGURE 11-25
D Flip-Flop with
Clear and Preset



(a)

Ck	D	PreN	ClrN	Q^+
x	x	0	0	(not allowed)
x	x	0	1	1
x	x	1	0	0
↑	0	1	1	0
↑	1	1	1	1
0,1,↓	x	1	1	Q (no change)

(b)

or preset function. We will use the notation ClrN or PreN to indicate active-low clear and preset inputs. Thus, a logic 0 applied to ClrN will reset the flip-flop to $Q = 0$, and a 0 applied to PreN will set the flip-flop to $Q = 1$. These inputs override the clock and D inputs. That is, a 0 applied to the ClrN will reset the flip-flop regardless of the values of D and the clock. Under normal operating conditions, a 0 should not be applied simultaneously to ClrN and PreN. When ClrN and PreN are both held at logic 1, the D and clock inputs operate in the normal manner. ClrN and PreN are often referred to as asynchronous clear and preset inputs because their operation does not depend on the clock. The table in Figure 11-25(b) summarizes the flip-flop operation. In the table, \uparrow indicates a rising clock edge, and X is a don't-care. The last row of the table indicates that if Ck is held at 0, held at 1, or has a falling edge, Q does not change.

Figure 11-26 illustrates the operation of the clear and preset inputs. At t_1 , ClrN = 0 holds the Q output at 0, so the rising edge of the clock is ignored. At t_2 and t_3 , normal state changes occur because ClrN and PreN are both 1. Then, Q is set to 1 by PreN = 0, but Q is cleared at t_4 by the rising edge of the clock because $D = 0$ at that time.

In synchronous digital systems, the flip-flops are usually driven by a common clock so that all state changes occur at the same time in response to the same clock edge. When designing such systems, we frequently encounter situations where we want some flip-flops to hold existing data even though the data input to the flip-flops may be changing. One way to do this is to gate the clock, as shown in Figure 11-27(a). When $En = 0$, the clock input to the flip-flop is 0, and Q does not change. This method has two potential problems. First, gate delays may cause the clock to arrive at some flip-flops at different times than at other flip-flops, resulting in a loss of synchronization. Second, if En changes at the wrong time, the flip-flop may trigger due to the change in En instead of due to the change in the clock, again resulting in loss of synchronization. Rather than gating the clock, a better way is to use a flip-flop with a clock enable (CE). Such flip-flops are commonly used in CPLDs and FPGAs.

Figure 11-27(b) shows a D flip-flop with a clock enable, which we will call a D-CE flip-flop. When $CE = 0$, the clock is disabled and no state change occurs, so $Q^+ = Q$. When $CE = 1$, the flip-flop acts like a normal D flip-flop, so $Q^+ = D$. Therefore, the characteristic equation is $Q^+ = Q \cdot CE' + D \cdot CE$. The D-CE flip-flop is easily implemented using a D flip-flop and a multiplexer (Figure 11-27(c)). For this circuit, the MUX output is

$$Q^+ = D \cdot CE' + Q \cdot CE$$

Because there is no gate in the clock line, this cannot cause a synchronization problem.

FIGURE 11-26
Timing Diagram
for D Flip-Flop
with Asynchronous
Clear and Preset

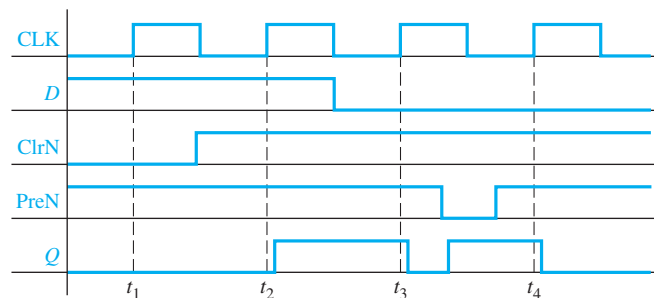
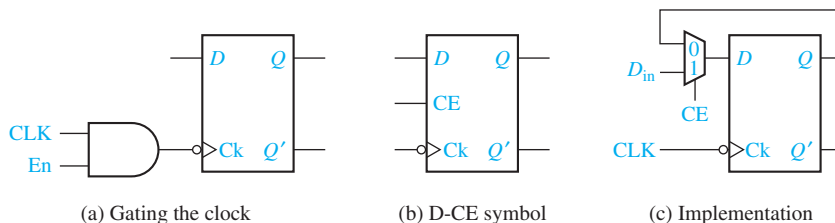


FIGURE 11-27
D Flip-Flop with
Clock Enable



11.9 Summary

In this unit, we have studied several types of latches and flip-flops. Flip-flops have a clock input, and the output changes only in response to a rising or falling edge of the clock. All of these devices have two output states: $Q = 0$ and $Q = 1$. For the S-R latch, $S = 1$ sets Q to 1, and $R = 1$ resets Q to 0. $S = R = 1$ is not allowed. The S-R flip-flop is similar except that Q only changes after the active edge of the clock. The gated D latch transmits D to the Q output when $G = 1$. When G is 0, the current value of D is stored in the latch and Q does not change. For the D flip-flop, Q is set equal to D after the active clock edge. The D-CE flip-flop works the same way, except the clock is only enabled when $CE = 1$. The J-K flip-flop is similar to the S-R flip-flop in that when $J = 1$ the active clock edge sets Q to 1, and when $K = 1$, the active edge resets Q to 0. When $J = K = 1$, the active clock edge causes Q to change state. The T flip-flop changes state on the active clock edge when $T = 1$; otherwise, Q does not change. Flip-flops can have asynchronous clear and preset inputs that cause Q to be cleared to 0 or preset to 1 independently of the clock.

Flip-flops can be constructed using gate circuits with feedback. Analysis of such circuits can be accomplished by tracing signal changes through the gates. Analysis can also be done using flow tables and asynchronous sequential circuit theory, but that is beyond the scope of this text. Timing diagrams are helpful in understanding the time relationships between the input and output signals for a latch or flip-flops. In general, the inputs must be applied a specified time before the active clock edge (the setup time), and they must be held constant a specified time after the active edge (the hold time). The time after the active clock edge before Q changes is the propagation delay.

The characteristic (next-state) equation for a flip-flop can be derived as follows: First, make a truth table that gives the next state (Q^+) as a function of the present state (Q) and the inputs. Any illegal input combinations should be treated as don't-cares. Then, plot a map for Q^+ and read the characteristic equation from the map.

The characteristic equations for the latches and flip-flops discussed in this chapter are:

$$Q^+ = S + R'Q \quad (SR = 0) \quad \text{(S-R latch or flip-flop)} \quad (11-6)$$

$$Q^+ = GD + G'Q \quad \text{(gated D latch)} \quad (11-7)$$

$$Q^+ = D \quad \text{(D flip-flop)} \quad (11-8)$$

$$Q^+ = D \cdot CE + Q \cdot CE' \quad (\text{D-CE flip-flop}) \quad (11-9)$$

$$Q^+ = JQ' + K'Q \quad (\text{J-K flip-flop}) \quad (11-10)$$

$$Q^+ = T \oplus Q = TQ' + T'Q \quad (\text{T flip-flop}) \quad (11-11)$$

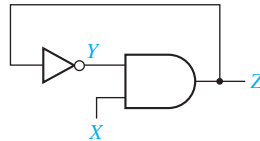
In each case, Q represents an initial or present state of the flip-flop, and Q^+ represents the final or next state. These equations are valid only when the appropriate restrictions on the flip-flop inputs are observed. For the S-R flip-flop, $S = R = 1$ is forbidden. For the master-slave S-R flip-flop, S and R should not change during the half of the clock cycle preceding the active edge. Setup and hold time restrictions must also be satisfied.

The characteristic equations given above apply to both latches and flip-flops, but their interpretation is different for the two cases. For example, for the gated D latch, Q^+ represents the state of the flip-flop a short time after one of the inputs changes. However, for the D flip-flop, Q^+ represents the state of the flip-flop a short time after the active clock edge.

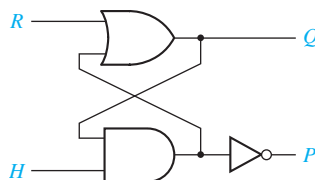
Conversion of one type of flip-flop to another is usually possible by adding external gates. Figure 11-24 shows how a J-K flip-flop and a D flip-flop can be converted to a T flip-flop.

Problems

- 11.1 Assume that the inverter in the given circuit has a propagation delay of 5 ns and the AND gate has a propagation delay of 10 ns. Draw a timing diagram for the circuit showing X , Y , and Z . Assume that X is initially 0, Y is initially 1, after 10 ns X becomes 1 for 80 ns, and then X is 0 again.

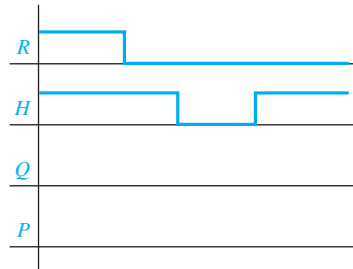


- 11.2 A latch can be constructed from an OR gate, an AND gate, and an inverter connected as follows:

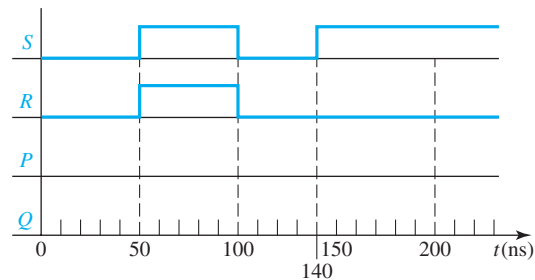


- What restriction must be placed on R and H so that P will always equal Q' (under steady-state conditions)?
- Construct a next-state table and derive the characteristic (next-state) equation for the latch.

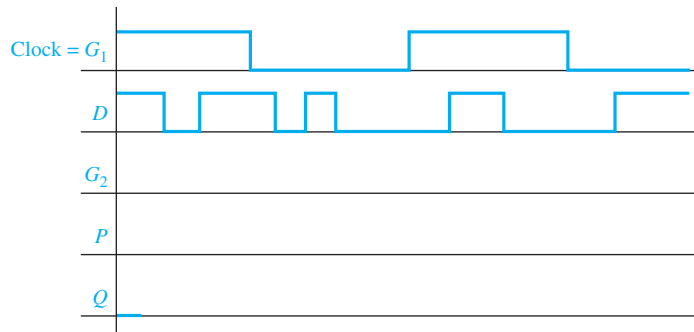
(c) Complete the following timing diagram for the latch.



- 11.3** This problem illustrates the improper operation that can occur if both inputs to an S-R latch are 1 and are then changed back to 0. For Figure 11-6, complete the following timing chart, assuming that each gate has a propagation delay of exactly 10 ns. Assume that initially $P = 1$ and $Q = 0$. Note that when $t = 100$ ns, S and R are both changed to 0. Then, 10 ns later, both P and Q will change to 1. Because these 1's are fed back to the gate inputs, what will happen after another 10 ns?



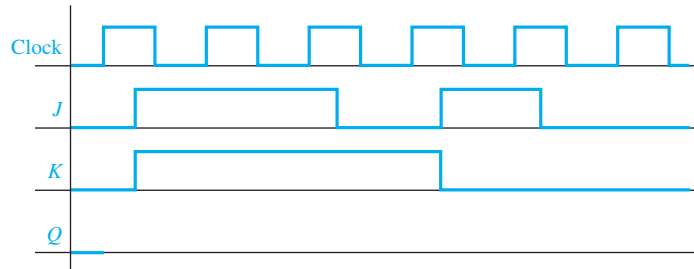
- 11.4** Design a gated D latch using only NAND gates and one inverter.
- 11.5** What change must be made to Figure 11-15(a) to implement a falling-edge-triggered D flip-flop? Complete the following timing diagram for the modified flip-flop.



- 11.6** A reset-dominant flip-flop behaves like an S-R flip-flop, except that the input $S = R = 1$ is allowed, and the flip-flop is reset when $S = R = 1$.
- (a) Derive the characteristic equation for a reset-dominant flip-flop.

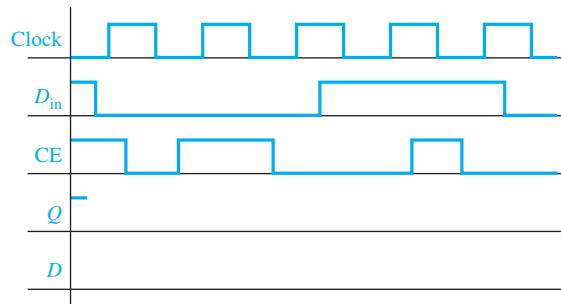
- (b) Show how a reset-dominant flip-flop can be constructed by adding gate(s) to an S-R flip-flop.

11.7 Complete the following timing diagram for the flip-flop of Figure 11-20(a).



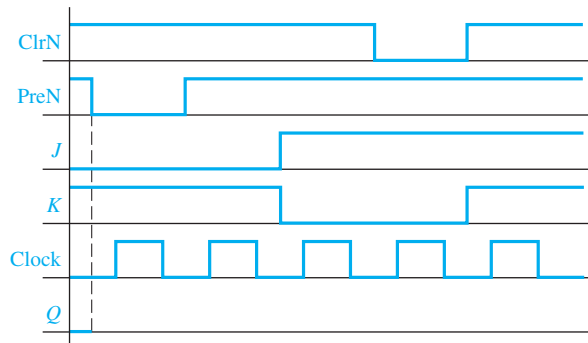
11.8 Complete the following diagrams for the falling-edge-triggered D-CE flip-flop of Figure 11-27(c). Assume Q begins at 1.

- (a) First draw Q based on your understanding of the behavior of a D flip-flop with clock enable.

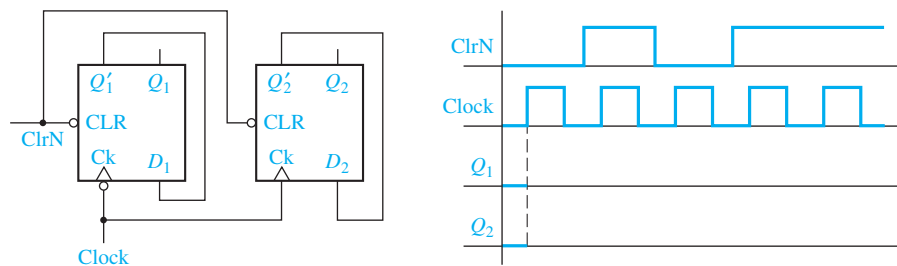


- (b) Now draw in the internal signal D from Figure 11-27(c), and confirm that this gives the same Q as in (a).

11.9 (a) Complete the following timing diagram for a J-K flip-flop with a falling-edge trigger and asynchronous ClrN and PreN inputs.



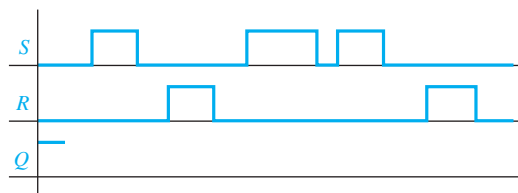
- (b) Complete the timing diagram for the following circuit. Note that the Ck inputs on the two flip-flops are different.



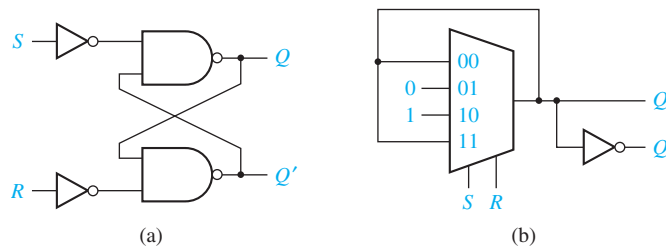
11.10 Convert by adding external gates:

- a D flip-flop to a J-K flip-flop.
- a T flip-flop to a D flip-flop.
- a T flip-flop to a D flip-flop with clock enable.

11.11 Complete the following timing diagram for an S-R latch. Assume Q begins at 1.



11.12 Using a truth table similar to Figure 11-8(b), confirm that each of these circuits is an S-R latch. What happens when $S = R = 1$ for each circuit?

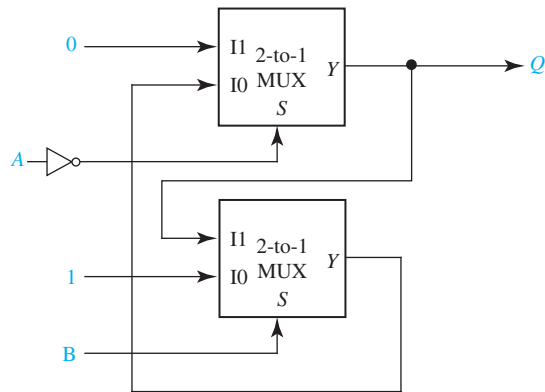


11.13 An AB latch operates as follows: If $A = 0$ and $B = 0$, the latch state is $Q = 0$; if either $A = 1$ or $B = 1$ (but not both), the latch output does not change; and when both $A = 1$ and $B = 1$, the latch state is $Q = 1$.

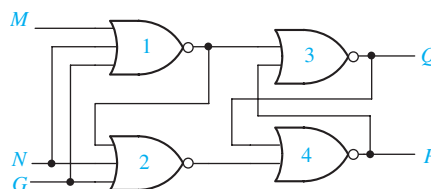
- Construct the state table and derive the characteristic equation for this AB latch.
- Derive a circuit for the AB latch that has four two-input NAND gates and two inverters.
- In your circuit of Part (b), are there any transitions between input combinations that might cause unreliable operation? Verify your answer.

- (d) In your circuit of Part (b), is there a gate output that provides the signal Q' ? Verify your answer.
- (e) Derive a circuit for the AB latch using four two-input NOR gates and two inverters.
- (f) Answer Parts (c) and (d) for your circuit of Part (e).

- 11.14** (a) Construct a state table for this circuit and identify the stable states of the circuit.
- (b) Derive a Boolean algebra equation for the next value of the output Q in terms of Q , A and B .
- (c) Analyze the behavior of the circuit. Is it a useful circuit? If not, explain why not; if yes, explain what it does.

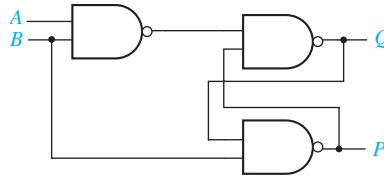


- 11.15** The following circuit is intended to be a gated latch circuit where the signal G is the gate.
- (a) Derive the next-state equation for this circuit using Q as the state variable and P as an output.
- (b) Construct the state table and output table for the circuit. Circle the stable states of the circuit.
- (c) Are there any restrictions on the allowable input combinations on M and N ? Explain your answer.
- (d) Is the output P usable as the complement of Q ? Verify your answer.
- (e) Assume that Gate 1 has a propagation delay of 30 ns and Gates 2, 3, and 4 have propagation delays of 10 ns. Construct a timing diagram for the circuit for the following input change: $M = N = Q = 0$ with G changing from 1 to 0.



11.16 Analyze the latch circuit shown.

- Derive the next-state equation for this circuit using Q as the state variable and P as an output.
- Construct the state table and output table for the circuit. Circle the stable states of the circuit.
- Are there any restrictions on the allowable input combinations on A and B ? Explain your answer.
- Is the output P usable as the complement of Q ? Verify your answer.

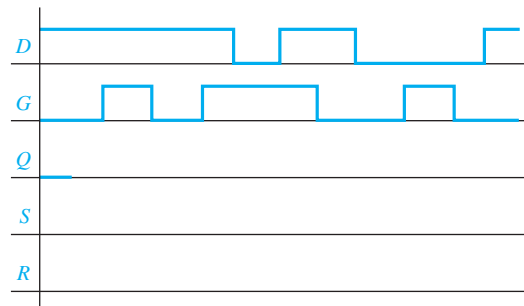


11.17 Derive the characteristic equations for the following latches and flip-flops in product-of-sums form.

- S-R latch or flip-flop
- Gated D latch
- D flip-flop
- D-CE flip-flop
- J-K flip-flop
- T flip-flop

11.18 Complete the following timing diagrams for a gated D latch. Assume Q begins at 0.

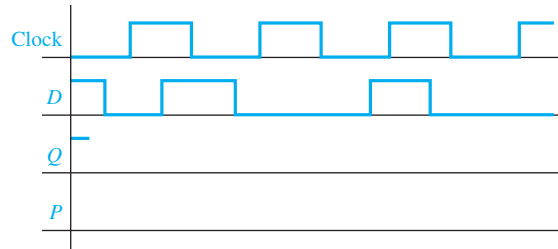
- First draw Q based on your understanding of the behavior of a gated D latch.



- Now draw in the internal signals S and R from Figure 11-11, and confirm that S and R give the same value for Q as in (a).

11.19 Complete the following diagrams for the rising-edge-triggered D flip-flop of Figure 11-15. Assume Q begins at 1.

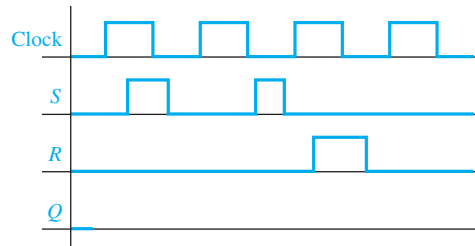
(a) First draw Q based on your understanding of the behavior of a D flip-flop.



(b) Now draw in the internal signal P from Figure 11-15, and confirm that P gives the same Q as in (a).

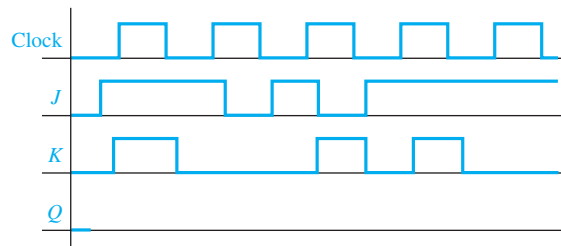
11.20 A set-dominant flip-flop is similar to the reset-dominant flip-flop of Problem 11.6 except that the input combination $S = R = 1$ sets the flip-flop. Repeat Problem 11.6 for a set-dominant flip-flop.

11.21 Fill in the timing diagram for a falling-edge-triggered S-R flip-flop. Assume Q begins at 0.



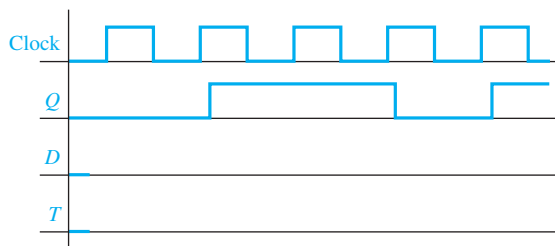
11.22 Fill in the timing diagram for a falling-edge-triggered J-K flip-flop.

(a) Assume Q begins at 0.

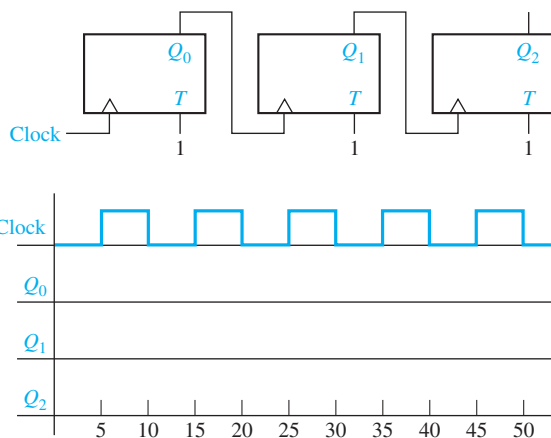


(b) Assume Q begins at 1, but Clock, J , and K are the same.

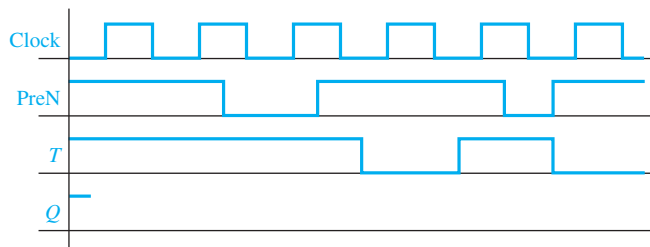
- 11.23** (a) Find the input for a rising-edge-triggered D flip-flop that would produce the output Q as shown. Fill in the timing diagram.
 (b) Repeat for a rising-edge-triggered T flip-flop.



- 11.24** Here is the diagram of a 3-bit ripple counter. Assume $Q_0 = Q_1 = Q_2 = 0$ at $t = 0$, and assume each flip-flop has a delay of 1 ns from the clock input to the Q output. Fill in Q_0 , Q_1 , and Q_2 of the timing diagram. Flip-flop Q_1 , will be triggered when Q_0 changes from 0 to 1.

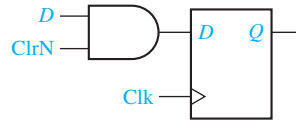


- 11.25** Fill in the following timing diagram for a rising-edge-triggered T flip-flop with an asynchronous active-low PreN input. Assume Q begins at 1.

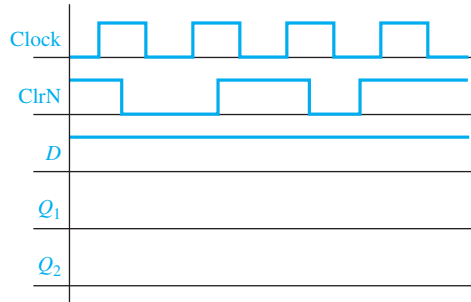


- 11.26** The ClrN and PreN inputs introduced in Section 11.8 are called asynchronous because they operate independently of the clock (i.e., they are not synchronized with the clock). We can also make flip-flops with synchronous clears or preset

inputs. A D-flip-flop with an active-low synchronous ClrN input may be constructed from a regular D flip-flop as follows.



Fill in the timing diagram. For Q_1 , assume a synchronous ClrN as above, and for Q_2 , assume an asynchronous ClrN as in Section 11.8. Assume $Q_1 = Q_2 = 0$ at the beginning.

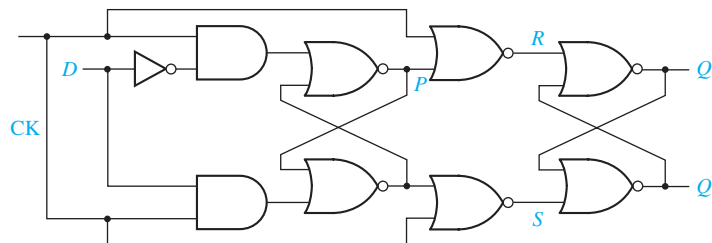


- 11.27** (a) Construct a D flip-flop using an inverter and an S-R flip-flop.
 (b) If the propagation delay and setup time of the S-R flip-flop in (a) are 2.5 ns and 1.5 ns, respectively, and if the inverter has a propagation delay of 1 ns, what are the propagation delay and setup time of the D flip-flop of Part (a)?
- 11.28** Redesign the debouncing circuit of Figure 11-9 using the \bar{S} - \bar{R} latch of Figure 11-10.

Programmed Exercise 11.29

Cover the bottom part of each page with a sheet of paper and slide it down as you check your answers.

The internal logic diagram of a falling-edge-triggered D flip-flop follows. This flip-flop consists of two basic S-R latches with added gates. When the clock input (CK) is 1, the value of D is stored in the first S-R latch (P). When the clock changes from 1 to 0, the value of P is transferred to the output latch (Q). Thus, the operation is similar to that of the master-slave S-R flip-flop shown in Figure 11-19, except for the edges at which the data is stored.



In this exercise you will be asked to analyze the operation of the D flip-flop shown above by filling in a table showing the values of CK, D , P , S , R , and Q after each change of input. It will be helpful if you mark the changes in these values on the circuit diagram as you trace the signals. Initially, assume the following signal values:

CK	D	P	S	R	Q	
0	0	0	0	1	0	(stable)

Verify by tracing signals through the circuit that this is a stable condition of the circuit; that is, no change will occur in P , S , R , or Q . Now assume that CK is changed to 1:

	CK	D	P	S	R	Q	
1.	0	0	0	0	1	0	(stable)
2.	1	0	0	0	1	0	?
3.							

Trace the change in CK through the circuit to see if a change in P , S , or R will occur. If a change does occur, mark row 2 of the preceding table “unstable” and enter the new values in row 3.

Answer:

2.	1	0	0	0	1	0	(unstable)
3.	1	0	0	0	0	0	(stable)
4.	1	1	0	0	0	0	(unstable)
5.	1	1					?

Verify that row 3 is stable; that is, by tracing signals show that no further change in P , S , R , or Q will occur. Next D is changed to 1 as shown in row 4. Verify that row 4 is unstable, fill in the new values in row 5, and indicate if row 5 is stable or unstable.

Answer:

	CK	D	P	S	R	Q	
5.	1	1	1	0	0	0	(stable)
6.	0	1	1	0	0	0	?
7.	0	1					?
8.	0	1					

Then CK is changed to 0 (row 6). If row 6 is unstable, indicate the new value of S in row 7. If row 7 is unstable, indicate the new value of Q in row 8. Then determine whether row 8 is stable or not.

Answer:

	CK	D	P	S	R	Q	
7.	0	1	1	1	0	0	(unstable)
8.	0	1	1	1	0	1	(stable)
9.	0	0					(stable)
10.	1	0					
11.	1	0					

Next, D is changed back to 0 (row 9). Fill in the values in row 9 and verify that it is stable. CK is changed to 1 in row 10. If row 10 is unstable, fill in row 11 and indicate whether it is stable or not.

Answer:

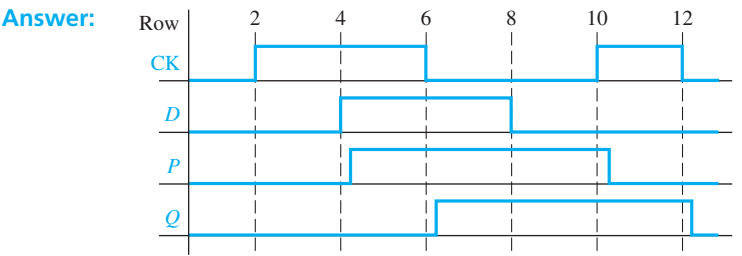
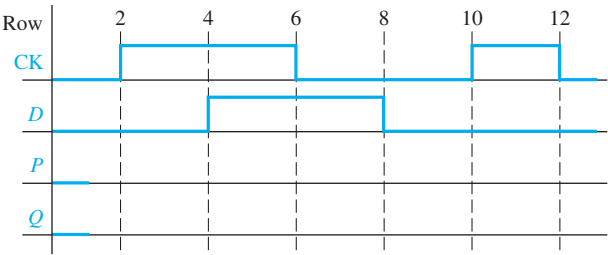
9.	0	0	1	1	0	1	(stable)
10.	1	0	1	1	0	1	(unstable)
11.	1	0	0	0	0	1	(stable)
12.	0	0					
13.	0	0					
14.	0	0					

CK is changed back to 0 in row 12. Complete the rest of the table.

Answer:

12.	0	0	0	0	0	1	(unstable)
13.	0	0	0	0	1	1	(unstable)
14.	0	0	0	0	1	0	(stable)

Using the previous results, plot P and Q on the following timing diagram. Verify that your answer is consistent with the description of the flip-flop operation given in the first paragraph of this exercise.



Chapter 5

Synchronous Sequential Logic

5.1 INTRODUCTION

Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format. The technology enabling and supporting these devices is critically dependent on electronic components that can store information, i.e., have memory. This chapter examines the operation and control of these devices and their use in circuits and enables you to better understand what is happening in these devices when you interact with them. The digital circuits considered thus far have been combinational—their output depends only and immediately on their inputs—they have no memory, i.e., dependence on past values of their inputs. Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed at a later time. Our treatment will distinguish sequential logic from combinational logic.

5.2 SEQUENTIAL CIRCUITS

A block diagram of a sequential circuit is shown in Fig. 5.1. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state

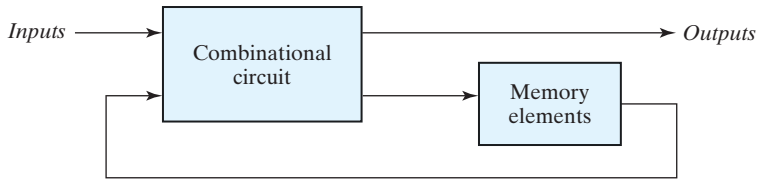


FIGURE 5.1
Block diagram of sequential circuit

in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.** In contrast, the outputs of combinational logic depend only on the present values of the inputs.

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A *synchronous* sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous* sequential circuit depends upon the input signals at any instant of time *and* the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times. The instability problem imposes many difficulties on the designer. These circuits will not be covered in this text.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic train of *clock pulses*. The clock signal is commonly denoted by the identifiers *clock* and *clk*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine *when* computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine *what* changes will take place affecting the storage elements and the outputs. For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits* and are the type most frequently encountered in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of

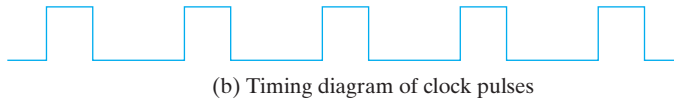
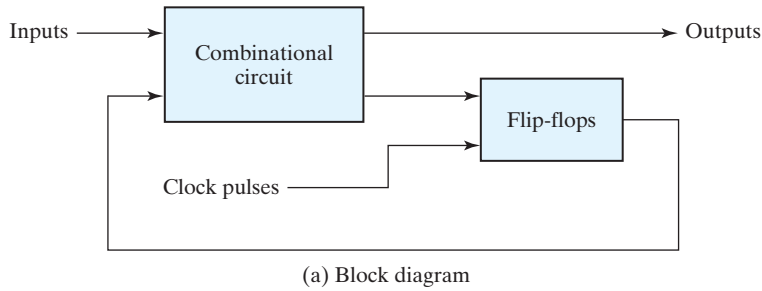


FIGURE 5.2
Synchronous clocked sequential circuit

clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops*. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig. 5.2. The *outputs* are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flip-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig. 5.2, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives. Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

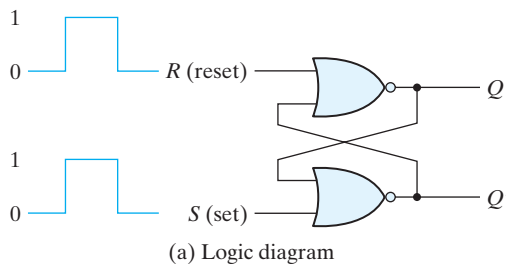
5.3 STORAGE ELEMENTS: LATCHES

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states. The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state. *Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches; those controlled by a clock transition are flip-flops.* Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits. Because they are the building blocks of flip-flops, however, we will consider the fundamental storage mechanism used in latches before considering flip-flops in the next section.

SR Latch

The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset. The *SR* latch constructed with two cross-coupled NOR gates is shown in Fig. 5.3. The latch has two useful states. When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $Q' = 1$, it is in the *reset state*. Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a meta-stable state. Consequently, in practical applications, setting both inputs to 1 is forbidden.

Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the *S* input causes the latch to go to the set state. The *S* input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 5.3(b), two input conditions cause the circuit to be in



S	R	Q	Q'
1	0	1	0
0	0	1	0 (after $S = 1, R = 0$)
0	1	0	1
0	0	0	1 (after $S = 0, R = 1$)
1	1	0	0 (forbidden)

(b) Function table

FIGURE 5.3
SR latch with NOR gates

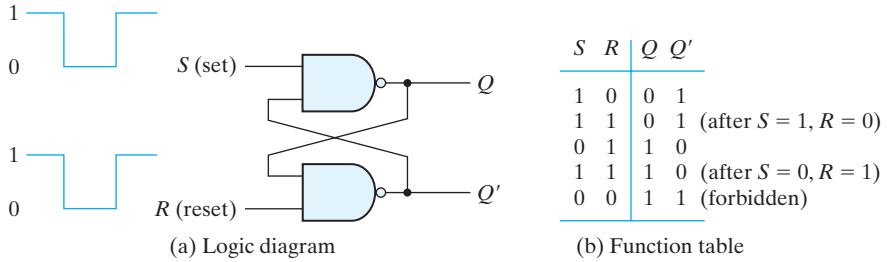


FIGURE 5.4
SR latch with NAND gates

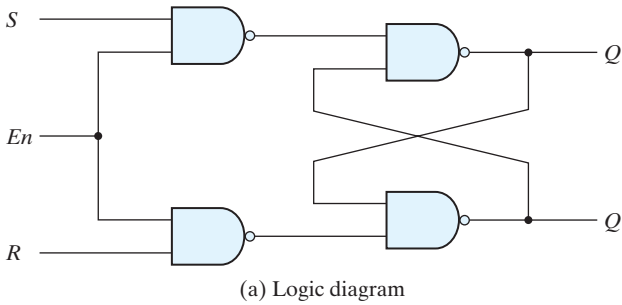
the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentarily applying a 1 to the R input. The 1 can then be removed from R , whereupon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.

If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0. It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.

The SR latch with two cross-coupled NAND gates is shown in Fig. 5.4. It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch. The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit.

The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed by determining whether S and R (or S' and R') can affect the circuit. An SR latch with a control input is shown in Fig. 5.5. It consists of the basic SR latch and two additional NAND gates. The control input En acts as an *enable* signal for the other two inputs. **The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.** This is the quiescent condition for the SR latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with $S = 1, R = 0$, and $En = 1$



En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

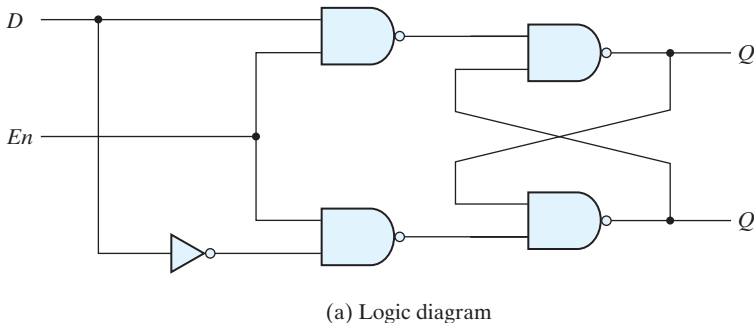
FIGURE 5.5
SR latch with control input

(active-high enabled). To change to the reset state, the inputs must be $S = 0$, $R = 1$, and $En = 1$. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En , so that the state of the output does not change regardless of the values of S and R . Moreover, when $En = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.

An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic SR latch, which puts it in the undefined state. When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the S or R input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice. Nevertheless, the SR latch is an important circuit because other useful latches and flip-flops are constructed from it.

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate 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 Fig. 5.6. This latch has only two inputs: D (data) and



En	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 5.6
D latch

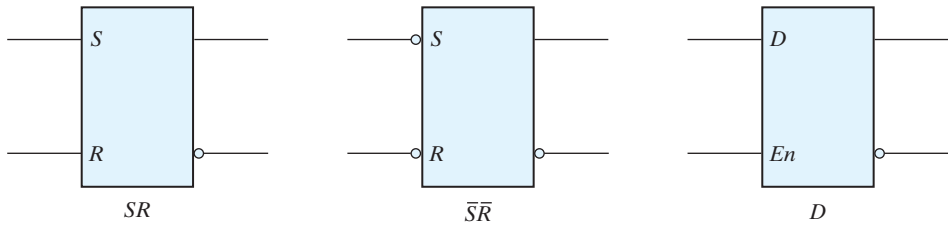


FIGURE 5.7
Graphic symbols for latches

En (enable). The *D* input goes directly to the *S* input, and its complement is applied to the *R* input. As long as the enable input is at 0, the cross-coupled *SR* 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 *En* = 1. If *D* = 1, the *Q* output goes to 1, placing the circuit in the set state. If *D* = 0, output *Q* goes to 0, placing the circuit in the reset state.

The *D* latch receives that designation from its ability to hold *data* in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. The binary information present at the data input of the *D* latch is transferred to the *Q* output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input *D* to the output, and for this reason, the circuit is often called a *transparent* latch. When the enable input signal is de-asserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the *Q* output until the enable input is asserted again. Note that an inverter could be placed at the enable input. Then, depending on the physical circuit, the external enabling signal will be a value of 0 (active low) or 1 (active high).

The graphic symbols for the various latches are shown in Fig. 5.7. A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the *SR* latch has inputs *S* and *R* indicated inside the block. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the *D* latch has inputs *D* and *En* indicated inside the block.

5.4 STORAGE ELEMENTS: FLIP-FLOPS

The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a *trigger*, and the transition it causes is said to trigger the flip-flop. The *D* latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

As seen from the block diagram of Fig. 5.2, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. Consequently, the 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 appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active 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 common clock source.

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 common clock. The problem with the latch is that it responds to a change in the *level* of a clock pulse. As shown in Fig. 5.8(a), a positive level response in the enable input allows changes in the output when the D input changes while the clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal *transition*. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. 5.8, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that

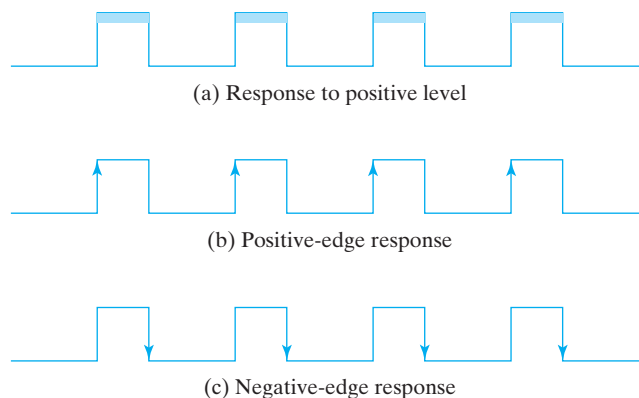


FIGURE 5.8
Clock response in latch and flip-flop

triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse. We will now proceed to show the implementation of both types of flip-flops.

Edge-Triggered *D* Flip-Flop

The construction of a *D* flip-flop with two *D* latches and an inverter is shown in Fig. 5.9. The first latch is called the master and the second the slave. The circuit samples the *D* input and changes its output *Q* only at the negative edge of the synchronizing or controlling clock (designated as *Clk*). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output *Q* is equal to the master output *Y*. The master latch is disabled because *Clk* = 0. When the input pulse changes to the logic-1 level, the data from the external *D* input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its *enable* input is equal to 0. Any change in the input changes the master output at *Y*, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the *D* input. At the same time, the slave is enabled and the value of *Y* is transferred to the output of the flip-flop at *Q*. Thus, *a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0*.

The behavior of the master–slave flip-flop just described dictates that (1) the output may change only once, (2) a change in the output is triggered by the negative edge of the clock, and (3) the change may occur only during the clock’s negative level. The value that is produced at the output of the flip-flop is the value that was *stored in the master stage immediately before the negative edge occurred*. It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the *Clk* terminal and the junction between the other inverter and input *En* of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal.

Another construction of an edge-triggered *D* flip-flop uses three *SR* latches as shown in Fig. 5.10. Two latches respond to the external *D* (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop. The *S* and *R* inputs of the output latch

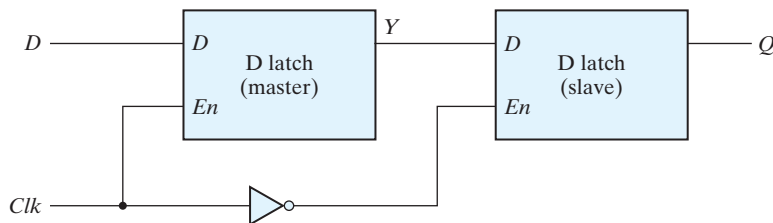


FIGURE 5.9
Master–slave *D* flip-flop

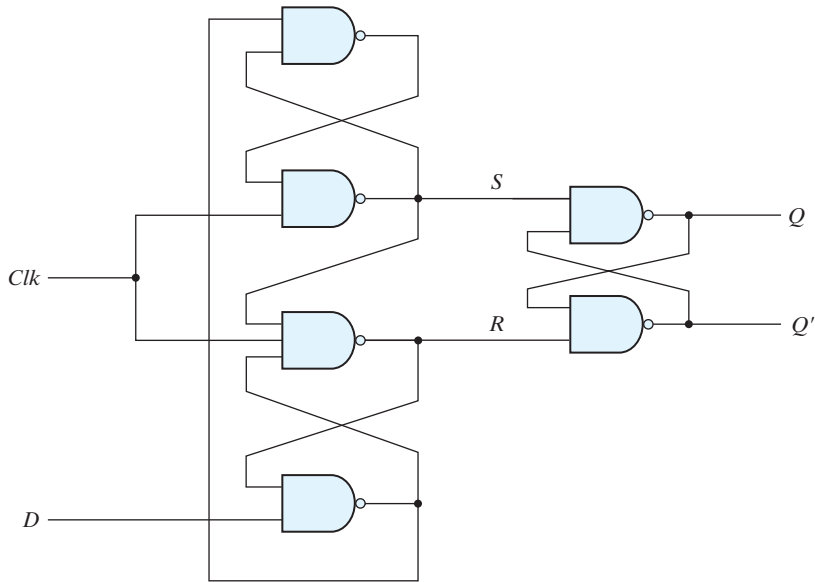


FIGURE 5.10
D-type positive-edge-triggered flip-flop

are maintained at the logic-1 level when $Clk = 0$. This causes the output to remain in its present state. Input D may be equal to 0 or 1. If $D = 0$ when Clk becomes 1, R changes to 0. This causes the flip-flop to go to the reset state, making $Q = 0$. If there is a change in the D input while $Clk = 1$, terminal R remains at 0 because Q is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, R goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if $D = 1$ when Clk goes from 0 to 1, S changes to 0. This causes the circuit to go to the set state, making $Q = 1$. Any change in D while $Clk = 1$ does not affect the output.

In sum, when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of D is transferred to Q . A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in D when Clk is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the *setup time* during which the D input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the *hold time* during which the D input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state. These and other parameters are specified in manufacturers' data books for specific logic families.

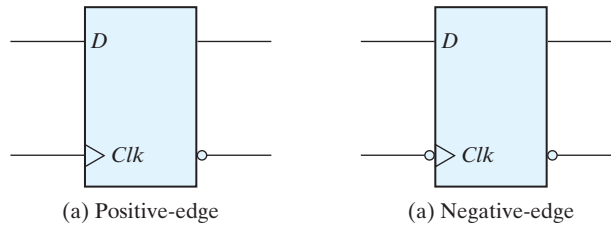


FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

The graphic symbol for the edge-triggered *D* flip-flop is shown in Fig. 5.11. It is similar to the symbol used for the *D* latch, except for the arrowhead-like symbol in front of the letter *Clk*, designating a *dynamic* input. The *dynamic indicator* (>) denotes the fact that the flip-flop responds to the edge transition of the clock. A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit. The absence of a bubble designates a positive-edge response.

Other Flip-Flops

Very large-scale integration circuits contain several thousands of gates within one package. Circuits are constructed by interconnecting the various gates to provide a digital system. Each flip-flop is constructed from an interconnection of gates. The most economical and efficient flip-flop constructed in this manner is the edge-triggered *D* flip-flop, because it requires the smallest number of gates. Other types of flip-flops can be constructed by using the *D* flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the *JK* and *T* flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the *D* flip-flop can set or reset the output, depending on the value of the *D* input immediately before the clock transition. Synchronized by a clock signal, the *JK* flip-flop has two inputs and performs all three operations. The circuit diagram of a *JK* flip-flop constructed with a *D* flip-flop and gates is shown in Fig. 5.12(a). The *J* input sets the flip-flop to 1, the *K* input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the *D* input:

$$D = JQ' + K'Q$$

When $J = 1$ and $K = 0$, $D = Q' + Q = 1$, so the next clock edge sets the output to 1. When $J = 0$ and $K = 1$, $D = 0$, so the next clock edge resets the output to 0. When both $J = K = 1$ and $D = Q'$, the next clock edge complements the output. When both $J = K = 0$ and $D = Q$, the clock edge leaves the output unchanged. The graphic symbol for the *JK* flip-flop is shown in Fig. 5.12(b). It is similar to the graphic symbol of the *D* flip-flop, except that now the inputs are marked *J* and *K*.

The *T* (toggle) flip-flop is a complementing flip-flop and can be obtained from a *JK* flip-flop when inputs *J* and *K* are tied together. This is shown in Fig. 5.13(a). When

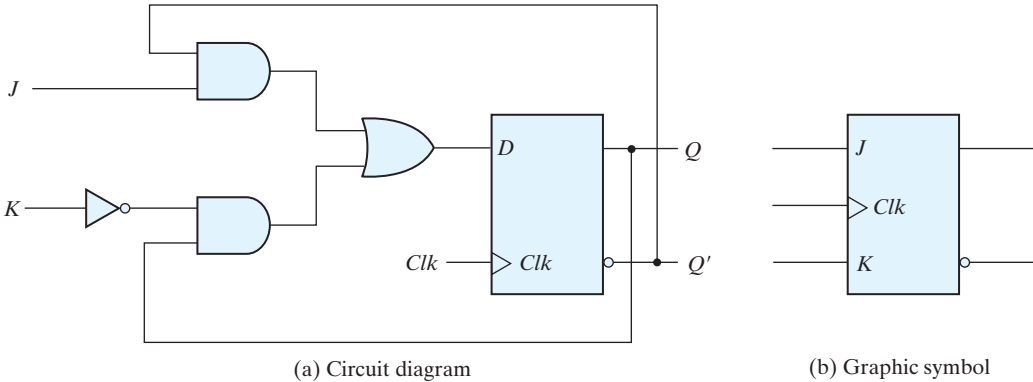


FIGURE 5.12
JK flip-flop

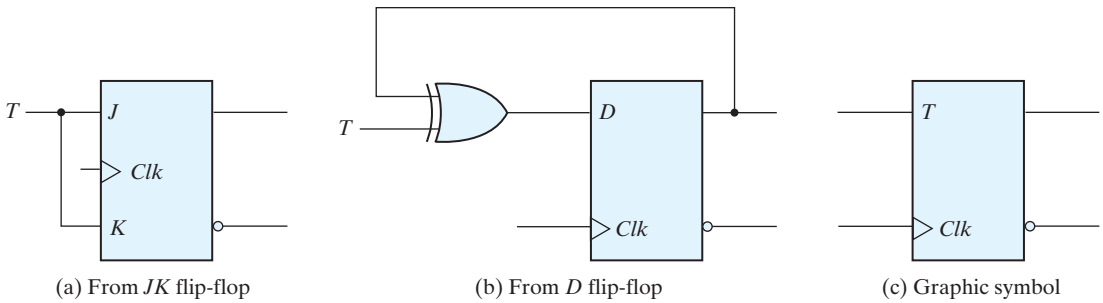


FIGURE 5.13
T flip-flop

$T = 0$ ($J = K = 0$), a clock edge does not change the output. When $T = 1$ ($J = K = 1$), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters.

The T flip-flop can be constructed with a D flip-flop and an exclusive-OR gate as shown in Fig. 5.13(b). The expression for the D input is

$$D = T \oplus Q = TQ' + T'Q$$

When $T = 0$, $D = Q$ and there is no change in the output. When $T = 1$, $D = Q'$ and the output complements. The graphic symbol for this flip-flop has a T symbol in the input.

Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 5.1. They define the next state (i.e., the state that results from a clock transition)

Table 5.1
Flip-Flop Characteristic Tables

<i>JK Flip-Flop</i>			
<i>J</i>	<i>K</i>	<i>Q(t + 1)</i>	
0	0	<i>Q(t)</i>	No change
0	1	0	Reset
1	0	1	Set
1	1	<i>Q'(t)</i>	Complement

<i>D Flip-Flop</i>		
<i>D</i>	<i>Q(t + 1)</i>	
0	0	Reset
1	1	Set

<i>T Flip-Flop</i>		
<i>T</i>	<i>Q(t + 1)</i>	
0	<i>Q(t)</i>	No change
1	<i>Q'(t)</i>	Complement

as a function of the inputs and the present state. $Q(t)$ refers to the present state (i.e., the state present prior to the application of a clock edge). $Q(t + 1)$ is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and $t + 1$. Thus, $Q(t)$ denotes the state of the flip-flop immediately before the clock edge, and $Q(t + 1)$ denotes the state that results from the clock transition.

The characteristic table for the *JK* flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as $Q(t + 1) = Q(t)$, indicating that the clock produces no change of state. When $K = 1$ and $J = 0$, the clock resets the flip-flop and $Q(t + 1) = 0$. With $J = 1$ and $K = 0$, the flip-flop sets and $Q(t + 1) = 1$. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as $Q(t + 1) = Q'(t)$.

The next state of a *D* flip-flop is dependent only on the D input and is independent of the present state. This can be expressed as $Q(t + 1) = D$. It means that the next-state value is equal to the value of D . Note that the *D* flip-flop does not have a “no-change” condition. Such a condition can be accomplished either by disabling the clock or by operating the clock by having the output of the flip-flop connected into the D input. Either method effectively circulates the output of the flip-flop when the state of the flip-flop must remain unchanged.

The characteristic table of the *T* flip-flop has only two conditions: When $T = 0$, the clock edge does not change the state; when $T = 1$, the clock edge complements the state of the flip-flop.

Characteristic Equations

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$Q(t + 1) = D$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 5.12. We obtain

$$Q(t + 1) = JQ' + K'Q$$

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 5.13:

$$Q(t + 1) = T \oplus Q = TQ' + T'Q$$

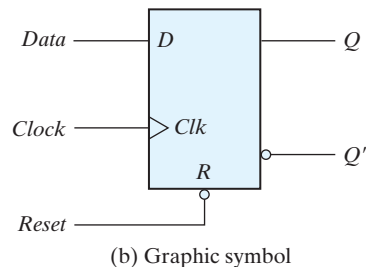
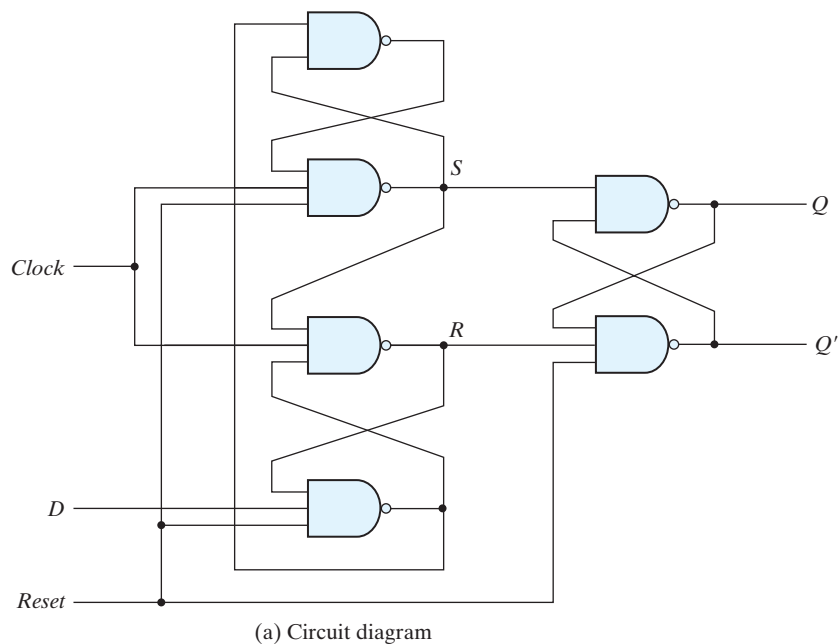
Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called *preset* or *direct set*. The input that clears the flip-flop to 0 is called *clear* or *direct reset*. When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 5.14. The circuit diagram is the same as the one in Fig. 5.10, except for the additional reset input connections to three NAND gates. When the reset input is 0, it forces output Q' to stay at 1, which, in turn, clears output Q to 0, thus resetting the flip-flop. Two other connections from the reset input ensure that the S input of the third SR latch stays at logic 1 while the reset input is at 0, regardless of the values of D and Clk .

The graphic symbol for the D flip-flop with a direct reset has an additional input marked with R . The bubble along the input indicates that the reset is active at the logic-0 level. Flip-flops with a direct set use the symbol S for the asynchronous set input.

The function table specifies the operation of the circuit. When $R = 0$, the output is reset to 0. This state is independent of the values of D or Clk . Normal clock operation can proceed only after the reset input goes to logic 1. The clock at Clk is shown with an upward arrow to indicate that the flip-flop triggers on the positive edge of the clock. The value in D is transferred to Q with every positive-edge clock signal, provided that $R = 1$.



<i>R</i>	<i>Clk</i>	<i>D</i>	<i>Q</i>	<i>Q'</i>
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

FIGURE 5.14
D flip-flop with asynchronous reset

5.5 ANALYSIS OF CLOCKED SEQUENTIAL CIRCUITS

Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. It is also possible

0 is detected in a stream of 1s. It consists of two D flip-flops A and B , an input x and an output y . Since the D input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit:

$$\begin{aligned}A(t + 1) &= A(t)x(t) + B(t)x(t) \\B(t + 1) &= A'(t)x(t)\end{aligned}$$

A state equation is an algebraic expression that specifies the condition for a flip-flop state transition. The left side of the equation, with $(t + 1)$, denotes the next state of the flip-flop one clock edge later. The right side of the equation is a Boolean expression that specifies the present state and input conditions that make the next state equal to 1. Since all the variables in the Boolean expressions are a function of the present state, we can omit the designation (t) after each variable for convenience and can express the state equations in the more compact form

$$\begin{aligned}A(t + 1) &= Ax + Bx \\B(t + 1) &= A'x\end{aligned}$$

The Boolean expressions for the state equations can be derived directly from the gates that form the combinational circuit part of the sequential circuit, since the D values of the combinational circuit determine the next state. Similarly, the present-state value of the output can be expressed algebraically as

$$y(t) = [A(t) + B(t)]x'(t)$$

By removing the symbol (t) for the present state, we obtain the output Boolean equation:

$$y = (A + B)x'$$

State Table

The time sequence of inputs, outputs, and flip-flop states can be enumerated in a *state table* (sometimes called a *transition table*). The state table for the circuit of Fig. 5.15 is shown in Table 5.2. The table consists of four sections labeled *present state*, *input*, *next state*, and *output*. The present-state section shows the states of flip-flops A and B at any given time t . The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock cycle later, at time $t + 1$. The output section gives the value of y at time t for each present state and input condition.

The derivation of a state table requires listing all possible binary combinations of present states and inputs. In this case, we have eight binary combinations from 000 to 111. The next-state values are then determined from the logic diagram or from the state equations. The next state of flip-flop A must satisfy the state equation

$$A(t + 1) = Ax + Bx$$

Table 5.2
State Table for the Circuit of Fig. 5.15

Present State		Input	Next State		Output
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The next-state section in the state table under column A has three 1's where the present state of A and input x are both equal to 1 or the present state of B and input x are both equal to 1. Similarly, the next state of flip-flop B is derived from the state equation

$$B(t + 1) = A'x$$

and is equal to 1 when the present state of A is 0 and input x is equal to 1. The output column is derived from the output equation

$$y = Ax' + Bx'$$

The state table of a sequential circuit with D -type flip-flops is obtained by the same procedure outlined in the previous example. In general, a sequential circuit with m flip-flops and n inputs needs 2^{m+n} rows in the state table. The binary numbers from 0 through $2^{m+n} - 1$ are listed under the present-state and input columns. The next-state section has m columns, one for each flip-flop. The binary values for the next state are derived directly from the state equations. The output section has as many columns as there are output variables. Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

It is sometimes convenient to express the state table in a slightly different form having only three sections: present state, next state, and output. The input conditions are enumerated under the next-state and output sections. The state table of Table 5.2 is repeated in Table 5.3 in this second form. For each present state, there are two possible next states and outputs, depending on the value of the input. One form may be preferable to the other, depending on the application.

State Diagram

The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting

Table 5.3
Second Form of the State Table

Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>y</i>	<i>y</i>
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

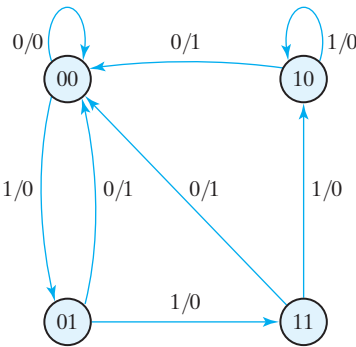


FIGURE 5.16
State diagram of the circuit of Fig. 5.15

the circles. The state diagram of the sequential circuit of Fig. 5.15 is shown in Fig. 5.16. The state diagram provides the same information as the state table and is obtained directly from Table 5.2 or Table 5.3. The binary number inside each circle identifies the state of the flip-flops. The directed lines are labeled with two binary numbers separated by a slash. The input value during the present state is labeled first, and the number after the slash gives the output during the present state with the given input. (It is important to remember that the bit value listed for the output along the directed line occurs during the present state and with the indicated input, and has nothing to do with the transition to the next state.) For example, the directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After the next clock cycle, the circuit goes to the next state, 01. If the input changes to 0, then the output becomes 1, but if the input remains at 1, the output stays at 0. This information is obtained from the state diagram along the two directed lines emanating from the circle with state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

The steps presented in this example are summarized below:

Circuit diagram → Equations – State table → State diagram

This sequence of steps begins with a structural representation of the circuit and proceeds to an abstract representation of its behavior. An HDL model can be in the form of a gate-level description or in the form of a behavioral description. It is important to note that a gate-level approach requires that the designer understands how to select and connect gates and flip-flops to form a circuit having a particular behavior. That understanding comes with experience. On the other hand, an approach based on behavioral modeling does not require the designer to know how to invent a schematic—the designer needs only to know how to describe behavior using the constructs of the HDL, because the circuit is produced automatically by a synthesis tool. Therefore, one does not have to accumulate years of experience in order to become a productive designer of digital circuits; nor does one have to acquire an extensive background in electrical engineering.

There is no difference between a state table and a state diagram, except in the manner of representation. The state table is easier to derive from a given logic diagram and the state equation. The state diagram follows directly from the state table. *The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation.* For example, the state diagram of Fig. 5.16 clearly shows that, starting from state 00, the output is 0 as long as the input stays at 1. The first 0 input after a string of 1's gives an output of 1 and transfers the circuit back to the initial state, 00. The machine represented by this state diagram acts to detect a zero in the bit stream of data. It corresponds to the behavior of the circuit in Fig. 5.15. Other circuits that detect a zero in a stream of data may have a simpler circuit diagram and state diagram.

Flip-Flop Input Equations

The logic diagram of a sequential circuit consists of flip-flops and gates. The interconnections among the gates form a combinational circuit and may be specified algebraically with Boolean expressions. The knowledge of the type of flip-flops and a list of the Boolean expressions of the combinational circuit provide the information needed to draw the logic diagram of the sequential circuit. The part of the combinational circuit that generates external outputs is described algebraically by a set of Boolean functions called *output equations*. The part of the circuit that generates the inputs to flip-flops is described algebraically by a set of Boolean functions called flip-flop *input equations* (or, sometimes, *excitation equations*). We will adopt the convention of using the flip-flop input symbol to denote the input equation variable and a subscript to designate the name of the flip-flop output. For example, the following input equation specifies an OR gate with inputs x and y connected to the D input of a flip-flop whose output is labeled with the symbol Q :

$$D_Q = x + y$$

The sequential circuit of Fig. 5.15 consists of two D flip-flops A and B , an input x , and an output y . The logic diagram of the circuit can be expressed algebraically with two flip-flop input equations and an output equation:

$$D_A = Ax + Bx$$

$$D_B = A'x$$

$$y = (A + B)x'$$

The three equations provide the necessary information for drawing the logic diagram of the sequential circuit. The symbol D_A specifies a D flip-flop labeled A . D_B specifies a second D flip-flop labeled B . The Boolean expressions associated with these two variables and the expression for output y specify the combinational circuit part of the sequential circuit.

The flip-flop input equations constitute a convenient algebraic form for specifying the logic diagram of a sequential circuit. They imply the type of flip-flop from the letter symbol, and they fully specify the combinational circuit that drives the flip-flops. Note that the expression for the input equation for a D flip-flop is identical to the expression for the corresponding state equation. This is because of the characteristic equation that equates the next state to the value of the D input: $Q(t + 1) = D_Q$.

Analysis with D Flip-Flops

We will summarize the procedure for analyzing a clocked sequential circuit with D flip-flops by means of a simple example. The circuit we want to analyze is described by the input equation

$$D_A = A \oplus x \oplus y$$

The D_A symbol implies a D flip-flop with output A . The x and y variables are the inputs to the circuit. No output equations are given, which implies that the output comes from the output of the flip-flop. The logic diagram is obtained from the input equation and is drawn in Fig. 5.17(a).

The state table has one column for the present state of flip-flop A , two columns for the two inputs, and one column for the next state of A . The binary numbers under Axy are listed from 000 through 111 as shown in Fig. 5.17(b). The next-state values are obtained from the state equation

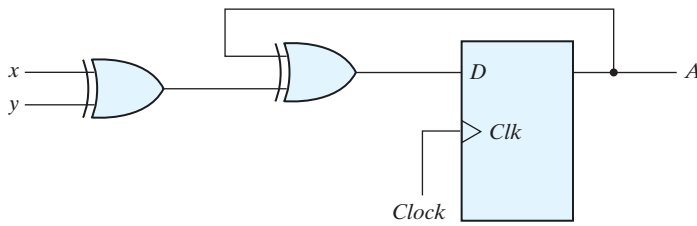
$$A(t + 1) = A \oplus x \oplus y$$

The expression specifies an odd function and is equal to 1 when only one variable is 1 or when all three variables are 1. This is indicated in the column for the next state of A .

The circuit has one flip-flop and two states. The state diagram consists of two circles, one for each state as shown in Fig. 5.17(c). The present state and the output can be either 0 or 1, as indicated by the number inside the circles. A slash on the directed lines is not needed, because there is no output from a combinational circuit. The two inputs can have four possible combinations for each state. Two input combinations during each state transition are separated by a comma to simplify the notation.

Analysis with J/K Flip-Flops

A state table consists of four sections: present state, inputs, next state, and outputs. The first two are obtained by listing all binary combinations. The output section is determined from the output equations. The next-state values are evaluated from the state equations. For a D -type flip-flop, the state equation is the same as the input equation. When a flip-flop other than the D type is used, such as J/K or T , it is necessary to refer



(a) Circuit diagram

Present state	Inputs		Next state
<i>A</i>	<i>x</i>	<i>y</i>	<i>A</i>
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(b) State table



(c) State diagram

FIGURE 5.17
Sequential circuit with *D* flip-flop

to the corresponding characteristic table or characteristic equation to obtain the next-state values. We will illustrate the procedure first by using the characteristic table and again by using the characteristic equation.

The next-state values of a sequential circuit that uses *JK*- or *T*-type flip-flops can be derived as follows:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. List the binary values of each input equation.
3. Use the corresponding flip-flop characteristic table to determine the next-state values in the state table.

As an example, consider the sequential circuit with two *JK* flip-flops *A* and *B* and one input *x*, as shown in Fig. 5.18. The circuit has no outputs; therefore, the state table does not need an output column. (The outputs of the flip-flops may be considered as the outputs in this case.) The circuit can be specified by the flip-flop input equations

$$\begin{aligned} J_A &= B & K_A &= Bx' \\ J_B &= x' & K_B &= A'x + Ax' = A \oplus x \end{aligned}$$

The state table of the sequential circuit is shown in Table 5.4. The present-state and input columns list the eight binary combinations. The binary values listed under the

$K = 0$, the next state is 1. When $J = 0$ and $K = 1$, the next state is 0. When $J = K = 0$, there is no change of state and the next-state value is the same as that of the present state. When $J = K = 1$, the next-state bit is the complement of the present-state bit. Examples of the last two cases occur in the table when the present state AB is 10 and input x is 0. JA and KA are both equal to 0 and the present state of A is 1. Therefore, the next state of A remains the same and is equal to 1. In the same row of the table, JB and KB are both equal to 1. Since the present state of B is 0, the next state of B is complemented and changes to 1.

The next-state values can also be obtained by evaluating the state equations from the characteristic equation. This is done by using the following procedure:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. Substitute the input equations into the flip-flop characteristic equation to obtain the state equations.
3. Use the corresponding state equations to determine the next-state values in the state table.

The input equations for the two JK flip-flops of Fig. 5.18 were listed a couple of paragraphs ago. The characteristic equations for the flip-flops are obtained by substituting A or B for the name of the flip-flop, instead of Q :

$$A(t + 1) = JA' + K'A$$

$$B(t + 1) = JB' + K'B$$

Substituting the values of J_A and K_A from the input equations, we obtain the state equation for A :

$$A(t + 1) = BA' + (Bx')'A = A'B + AB' + Ax$$

The state equation provides the bit values for the column headed “Next State” for A in the state table. Similarly, the state equation for flip-flop B can be derived from the characteristic equation by substituting the values of J_B and K_B :

$$B(t + 1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$$

The state equation provides the bit values for the column headed “Next State” for B in the state table. Note that the columns in Table 5.4 headed “Flip-Flop Inputs” are not needed when state equations are used.

The state diagram of the sequential circuit is shown in Fig. 5.19. Note that since the circuit has no outputs, the directed lines out of the circles are marked with one binary number only, to designate the value of input x .

Analysis with T Flip-Flops

The analysis of a sequential circuit with T flip-flops follows the same procedure outlined for JK flip-flops. The next-state values in the state table can be obtained by using either

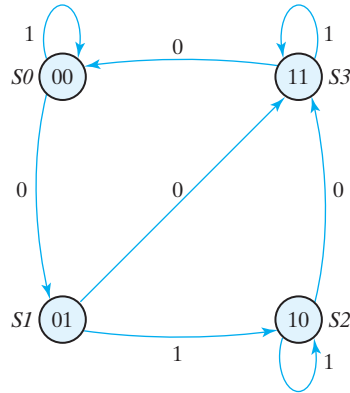


FIGURE 5.19
State diagram of the circuit of Fig. 5.18

the characteristic table listed in Table 5.1 or the characteristic equation

$$Q(t + 1) = T \oplus Q = T'Q + TQ'$$

Now consider the sequential circuit shown in Fig. 5.20. It has two flip-flops A and B , one input x , and one output y and can be described algebraically by two input equations and an output equation:

$$T_A = Bx$$

$$T_B = x$$

$$y = AB$$

The state table for the circuit is listed in Table 5.5. The values for y are obtained from the output equation. The values for the next state can be derived from the state equations by substituting T_A and T_B in the characteristic equations, yielding

$$A(t + 1) = (Bx)'A + (Bx)A' = AB' + Ax' + A'Bx$$

$$B(t + 1) = x \oplus B$$

The next-state values for A and B in the state table are obtained from the expressions of the two state equations.

The state diagram of the circuit is shown in Fig. 5.20(b). As long as input x is equal to 1, the circuit behaves as a binary counter with a sequence of states 00, 01, 10, 11, and back to 00. When $x = 0$, the circuit remains in the same state. Output y is equal to 1 when the present state is 11. Here, the output depends on the present state only and is independent of the input. The two values inside each circle and separated by a slash are for the present state and output.

Mealy and Moore Models of Finite State Machines

The most general model of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two models of sequential circuits: the Mealy model and the Moore model. Both are shown in Fig. 5.21. They differ only in the way the output

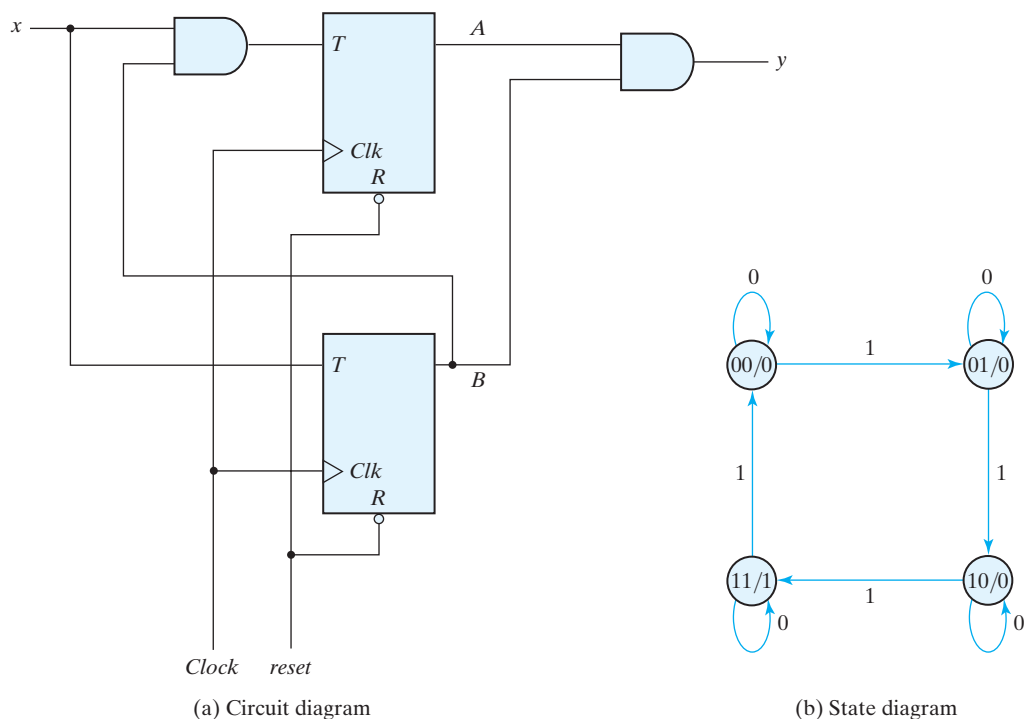


FIGURE 5.20
Sequential circuit with *T* flip-flops (Binary Counter)

Table 5.5
State Table for Sequential Circuit with T Flip-Flops

Present State		Input	Next State		Output
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

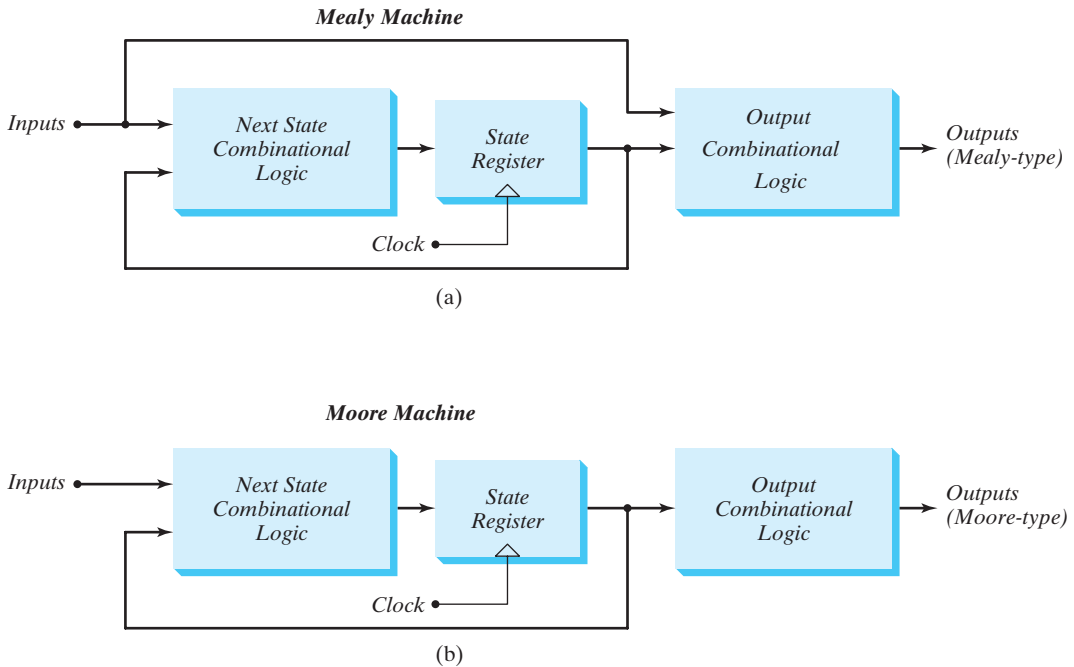


FIGURE 5.21
Block diagrams of Mealy and Moore state machines

is generated. In the Mealy model, the output is a function of both the present state and the input. In the Moore model, the output is a function of only the present state. A circuit may have both types of outputs. The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM. The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine. The Moore model is referred to as a Moore FSM or Moore machine.

The circuit presented previously in Fig. 5.15 is an example of a Mealy machine. Output y is a function of both input x and the present state of A and B . The corresponding state diagram in Fig. 5.16 shows both the input and output values, separated by a slash along the directed lines between the states.

An example of a Moore model is given in Fig. 5.18. Here, the output is a function of the present state only. The corresponding state diagram in Fig. 5.19 has only inputs marked along the directed lines. The outputs are the flip-flop states marked inside the circles. Another example of a Moore model is the sequential circuit of Fig. 5.20. The output depends only on flip-flop values, and that makes it a function of the present state only. The input value in the state diagram is labeled along the directed line, but the output value is indicated inside the circle together with the present state.

In a Moore model, the outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock. In a Mealy model, the outputs may change if the inputs change during the clock

cycle. Moreover, the outputs may have momentary false values because of the delay encountered from the time that the inputs change and the time that the flip-flop outputs change. In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge. The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs. Thus, **the output of the Mealy machine is the value that is present immediately before the active edge of the clock.**

5.6 SYNTHESIZABLE HDL MODELS OF SEQUENTIAL CIRCUITS

The Verilog HDL was introduced in Section 3.9. Combinational circuits were described in Section 4.12, and behavioral modeling with Verilog was introduced in that section as well. Behavioral models are abstract representations of the functionality of digital hardware. That is, they describe how a circuit behaves, but don't specify the internal details of the circuit. Historically, the abstraction has been described by truth tables, state tables, and state diagrams. An HDL describes the functionality differently, by language constructs that represent the operations of registers in a machine. This representation has “added value,” i.e., it is important for you to know how to use, because it can be simulated to produce waveforms demonstrating the behavior of the machine.

Behavioral Modeling

There are two kinds of abstract behaviors in the Verilog HDL. Behavior declared by the keyword **initial** is called *single-pass behavior* and specifies a single statement or a block statement (i.e., a list of statements enclosed by either a **begin . . . end** or a **fork . . . join** keyword pair). A single-pass behavior expires after the associated statement executes. In practice, designers use single-pass behavior primarily to prescribe stimulus signals in a test bench—never to model the behavior of a circuit—because synthesis tools do not accept descriptions that use the **initial** statement. The **always** keyword declares a *cyclic behavior*. Both types of behaviors begin executing when the simulator launches at time $t = 0$. The **initial** behavior expires after its statement executes; the **always** behavior executes and reexecutes indefinitely, until the simulation is stopped. A module may contain an arbitrary number of **initial** or **always** behavioral statements. They execute concurrently with respect to each other, starting at time 0, and may interact through common variables. Here's a word description of how an **always** statement works for a simple model of a D flip-flop: Whenever the rising edge of the clock occurs, if the reset input is asserted, the output q gets 0; otherwise the output Q gets the value of the input D . The execution of statements triggered by the clock is repeated until the simulation ends. We'll see shortly how to write this description in Verilog.

An **initial** behavioral statement executes only once. It begins its execution at the start of simulation and expires after all of its statements have completed execution. As mentioned at the end of Section 4.12, the **initial** statement is useful for generating input signals to simulate a design. In simulating a sequential circuit, it is necessary to generate a clock source for triggering the flip-flops. The following are two possible ways to provide a free-running clock that operates for a specified number of cycles:

<pre> initial begin clock = 1'b0; repeat (30) #10 clock = ~clock; end </pre>	<pre> initial begin clock = 1'b0; end initial 300 \$finish; always #10 clock = ~clock; </pre>
--	---

In the first version, the **initial** block contains two statements enclosed within the **begin** and **end** keywords. The first statement sets *clock* to 0 at time = 0. The second statement specifies a loop that reexecutes 30 times to wait 10 time units and then complement the value of *clock*. This produces 15 clock cycles, each with a cycle time of 20 time units. In the second version, the first **initial** behavior has a single statement that sets *clock* to 0 at time = 0, and it then expires (causes no further simulation activity). The second single-pass behavior declares a stopwatch for the simulation. The system task **finish** causes the simulation to terminate unconditionally after 300 time units have elapsed. Because this behavior has only one statement associated with it, there is no need to write the **begin** . . . **end** keyword pair. After 10 time units, the **always** statement repeatedly complements *clock*, providing a clock generator having a cycle time of 20 time units. The three behavioral statements in the second example can be written in any order.

Here is another way to describe a free-running clock:

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

This version, with two statements in one block statement, initializes the clock and then executes an indefinite loop (**forever**) in which the clock is complemented after a delay of 10 time steps. Note that the single-pass behavior never finishes executing and so does not expire. Another behavior would have to terminate the simulation.

The activity associated with either type of behavioral statement can be controlled by a delay operator that waits for a certain time or by an event control operator that waits for certain conditions to become true or for specified events (changes in signals) to occur. Time delays specified with the # *delay control operator* are commonly used in single-pass behaviors. The delay control operator suspends execution of statements until a specified time has elapsed. We've already seen examples of its use to specify signals in a test bench. Another operator @ is called the *event control operator* and is used to *suspend* activity until an event occurs. An event can be an unconditional change in a signal value (e.g., @ A) or a specified transition of a signal value (e.g., @ (**posedge** clock)). The general form of this type of statement is

```

always @ (event control expression) begin
  // Procedural assignment statements that execute when the condition is met
end

```

The event control expression specifies the condition that must occur to launch execution of the procedural assignment statements. The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such. The right-hand side can be any expression that produces a value using Verilog-defined operators.

The event control expression (also called the sensitivity list) specifies the events that must occur to initiate execution of the procedural statements associated with the **always** block. Statements within the block execute sequentially from top to bottom. After the last statement executes, the behavior waits for the event control expression to be satisfied. Then the statements are executed again. The sensitivity list can specify level-sensitive events, edge-sensitive events, or a combination of the two. In practice, designers do not make use of the third option, because this third form is not one that synthesis tools are able to translate into physical hardware. Level-sensitive events occur in combinational circuits and in latches. For example, the statement

```
always @ (A or B or C)
```

will initiate execution of the procedural statements in the associated **always** block if a change occurs in *A*, *B*, or *C*. In synchronous sequential circuits, changes in flip-flops occur only in response to a transition of a clock pulse. The transition may be either a positive edge or a negative edge of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the expression

```
always @(posedge clock or negedge reset)      // Verilog 1995
```

will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if *reset* goes through a negative transition. The 2001 and 2005 revisions to the Verilog language allow a comma-separated list for the event control expression (or sensitivity list):

```
always @(posedge clock, negedge reset)      // Verilog 2001, 2005
```

A procedural assignment is an assignment of a logic value to a variable within an **initial** or **always** statement. This is in contrast to a continuous assignment discussed in Section 4.12 with dataflow modeling. A continuous assignment has an implicit level-sensitive sensitivity list consisting of all of the variables on the right-hand side of its assignment statement. The updating of a continuous assignment is triggered whenever an event occurs in a variable included on the right-hand side of its expression. In contrast, a procedural assignment is made only when an assignment statement is executed and assigns value to it within a behavioral statement. For example, the clock signal in the preceding example was complemented only when the statement *clock = ~clock* executed; the statement did not execute until 10 time units after the simulation began. It is important to remember that a variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

There are two kinds of procedural assignments: *blocking* and *nonblocking*. The two are distinguished by the symbols that they use. Blocking assignments use the symbol (=) as the assignment operator, and nonblocking assignments use (<=) as the operator.

Blocking assignment statements are executed sequentially in the order they are listed in a block of statements. Nonblocking assignments are executed concurrently by evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. The two types of assignments may be better understood by means of an illustration. Consider these two procedural blocking assignments:

$$\begin{aligned} B &= A \\ C &= B + 1 \end{aligned}$$

The first statement transfers the value of A into B . The second statement increments the value of B and transfers the new value to C . At the completion of the assignments, C contains the value of $A + 1$.

Now consider the two statements as nonblocking assignments:

$$\begin{aligned} B &\leq A \\ C &\leq B + 1 \end{aligned}$$

When the statements are executed, the expressions on the right-hand side are evaluated and stored in a temporary location. The value of A is kept in one storage location and the value of $B + 1$ in another. After all the expressions in the block are evaluated and stored, the assignment to the targets on the left-hand side is made. In this case, C will contain the original value of B , plus 1. A general rule is to **use blocking assignments when sequential ordering is imperative and in cyclic behavior that is level sensitive** (i.e., in combinational logic). **Use nonblocking assignments when modeling concurrent execution** (e.g., edge-sensitive behavior such as synchronous, concurrent register transfers) **and when modeling latched behavior**. Nonblocking assignments are imperative in dealing with register transfer level design, as shown in Chapter 8. They model the concurrent operations of physical hardware synchronized by a common clock. Today's designers are expected to know what features of an HDL are useful in a practical way and how to avoid features that are not. Following these rules for using the assignment operators will prevent conditions that lead synthesis tools astray and create mismatches between the behavior of a model and the behavior of physical hardware that is produced by a synthesis tool.

HDL Models of Flip-Flops and Latches

HDL Examples 5.1 through 5.4 show Verilog descriptions of various flip-flops and a D latch. The D latch is said to be *transparent* because it responds to a change in data input with a change in the output as long as the enable input is asserted—viewing the output is the same as viewing the input. The description of a D latch is shown in HDL Example 5.1. It has two inputs, D and *enable*, and one output, Q . Since Q is assigned value in a behavior, its type must be declared to be **reg**. Hardware latches respond to input signal *levels*, so the two inputs are listed without edge qualifiers in the sensitivity list following the @ symbol in the **always** statement. In this model, there is only one blocking procedural assignment statement, and it specifies the transfer of input D to output Q if enable

is true (logic 1).¹ Note that this statement is executed every time there is a change in *D* if *enable* is 1.

A *D*-type flip-flop is the simplest example of a sequential machine. HDL Example 5.2 describes two positive-edge *D* flip-flops in two modules. The first responds only to the clock; the second includes an asynchronous reset input. Output *Q* must be declared as a **reg** data type in addition to being listed as an output. This is because it is a target output of a procedural assignment statement. The keyword **posedge** ensures that the transfer of input *D* into *Q* is synchronized by the positive-edge transition of *Clk*. A change in *D* at any other time does not change *Q*.

HDL Example 5.1 (*D*-Latch)

```
// Description of D latch (See Fig. 5.6)
module D_latch (Q, D, enable);
    output Q;
    input  D, enable;
    reg    Q;
    always @ (enable or D)
        if (enable) Q <= D;           // Same as: if (enable == 1)
endmodule

// Alternative syntax (Verilog 2001, 2005)
module D_latch (output reg Q, input enable, D);
    always @ (enable, D)
        if (enable) Q <= D;           // No action if enable not asserted
endmodule
```

HDL Example 5.2 (*D*-Type Flip-Flop)

```
// D flip-flop without reset
module D_FF (Q, D, Clk);
    output Q;
    input  D, Clk;
    reg    Q;
    always @ (posedge Clk)
        Q <= D;
endmodule

// D flip-flop with asynchronous reset (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
    always @ (posedge Clk, negedge rst)
        if (!rst) Q <= 1'b0;           // Same as: if (rst == 0)
        else Q <= D;
endmodule
```

¹The statement (single or block) associated with **if**(Boolean expression) executes if the Boolean expression is true.

The second module includes an asynchronous reset input in addition to the synchronous clock. A specific form of an **if** statement is used to describe such a flip-flop so that the model can be synthesized by a software tool. The event expression after the @ symbol in the **always** statement may have any number of edge events, either **posedge** or **negedge**. For modeling hardware, one of the events must be a clock event. The remaining events specify conditions under which asynchronous logic is to be executed. The designer knows which signal is the clock, but *clock* is not an identifier that software tools automatically recognize as the synchronizing signal of a circuit. The tool must be able to infer which signal is the clock, so *you need to write the description in a way that enables the tool to infer the clock correctly*. The rules are simple to follow: (1) Each **if** or **else if** statement in the procedural assignment statements is to correspond to an asynchronous event, (2) the last **else** statement corresponds to the clock event, and (3) the asynchronous events are tested first. There are two edge events in the second module of HDL Example 5.2. The **negedge** *rst* (reset) event is asynchronous, since it matches the **if** (!*rst*) statement. As long as *rst* is 0, *Q* is cleared to 0. If *Clk* has a positive transition, its effect is blocked. Only if *rst* = 1 can the **posedge** clock event synchronously transfer *D* into *Q*.

Hardware always has a reset signal. It is strongly recommended that all models of edge-sensitive behavior include a reset (or preset) input signal; otherwise, the initial state of the flip-flops of the sequential circuit cannot be determined. A sequential circuit cannot be tested with HDL simulation unless an initial state can be assigned with an input signal.

HDL Example 5.3 describes the construction of a *T* or *JK* flip-flop from a *D* flip-flop and gates. The circuit is described with the characteristic equations of the flip-flops:

$$\begin{aligned} Q(t+1) &= Q \oplus T && \text{for a } T \text{ flip-flop} \\ Q(t+1) &= JQ' + K'Q && \text{for a } JK \text{ flip-flop} \end{aligned}$$

The first module, *TFF*, describes a *T* flip-flop by instantiating *DFF*. (Instantiation is explained in Section 4.12.) The declared **wire**, *DT*, is assigned the exclusive-OR of *Q* and *T*, as is required for building a *T* flip-flop with a *D* flip-flop. The instantiation with the value of *DT* replacing *D* in module *DFF* produces the required *T* flip-flop. The *JK* flip-flop is specified in a similar manner by using its characteristic equation to define a replacement for *D* in the instantiated *DFF*.

HDL Example 5.3 (Alternative Flip-Flop Models)

```
// T flip-flop from D flip-flop and gates
module TFF (Q, T, Clk, rst);
  output Q;
  input T, Clk, rst;
  wire DT;
  assign DT = Q ^ T;           // Continuous assignment
  // Instantiate the D flip-flop
  DFF TF1 (Q, DT, Clk, rst);
endmodule
```

```

// JK flip-flop from D flip-flop and gates (V2001, 2005)
module JKFF (output reg Q, input J, K, Clk, rst);
    wire JK;
    assign JK = (J & ~Q) | (~K & Q);
    // Instantiate D flip-flop
    DFF JK1 (Q, JK, Clk, rst);
endmodule

// D flip-flop (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
    always @ (posedge Clk, negedge rst)
        if (!rst) Q <= 1'b0;
        else Q <= D;
endmodule

```

HDL Example 5.4 shows another way to describe a *JK* flip-flop. Here, we describe the flip-flop by using the characteristic table rather than the characteristic equation. The **case** multiway branch condition checks the two-bit number obtained by concatenating the bits of *J* and *K*. The **case** expression (*{J, K}*) is evaluated and compared with the values in the list of statements that follows. The first value that matches the true condition is executed. Since the concatenation of *J* and *K* produces a two-bit number, it can be equal to 00, 01, 10, or 11. The first bit gives the value of *J* and the second the value of *K*. The four possible conditions specify the value of the next state of *Q* after the application of a positive-edge clock.

HDL Example 5.4 (*JK* Flip-Flop)

```

// Functional description of JK flip-flop (V2001, 2005)
module JK_FF (input J, K, Clk, output reg Q, output Q_b);
    assign Q_b = ~ Q ;
    always @ (posedge Clk)
        case ({J,K})
            2'b00: Q <= Q;
            2'b01: Q <= 1'b0;
            2'b10: Q <= 1'b1;
            2'b11: Q <= !Q;
        endcase
endmodule

```

State diagram-Based HDL Models

An HDL model of the operation of a sequential circuit can be based on the format of the circuit's state diagram. A Mealy HDL model is presented in HDL Example 5.5 for the zero-detector machine described by the sequential circuit in Fig. 5.15 and its state diagram shown in Fig. 5.16. The input, output, clock, and reset are declared in the usual manner.

The state of the flip-flops is declared with identifiers *state* and *next_state*. These variables hold the values of the present state and the next value of the sequential circuit. The state's binary assignment is done with a **parameter** statement. (Verilog allows constants to be defined in a module by the keyword **parameter**.) The four states *S0* through *S3* are assigned binary 00 through 11. The notation $S2 = 2'b10$ is preferable to the alternative $S2 = 2$. The former uses only two bits to store the constant, whereas the latter results in a binary number with 32 (or 64) bits because an unsized number is interpreted and sized as an integer.

HDL Example 5.5 (Mealy Machine: Zero Detector)

// Mealy FSM zero detector (See Fig. 5.15 and Fig. 5.16) Verilog 2001, 2005 syntax

```

module Mealy_Zero_Detector (
    output reg y_out,
    input x_in, clock, reset
);
    reg [1: 0] state, next_state;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
    always @ (posedge clock, negedge reset) Verilog 2001, 2005 syntax
        if (reset == 0) state <= S0;
        else state <= next_state;

    always @ (state, x_in) // Form the next state
        case (state)
            S0: if (x_in) next_state = S1; else next_state = S0;
            S1: if (x_in) next_state = S3; else next_state = S0;
            S2: if (~x_in) next_state = S0; else next_state = S2;
            S3: if (x_in) next_state = S2; else next_state = S0;
        endcase

    always @ (state, x_in) // Form the Mealy output
        case (state)
            S0: y_out = 0;
            S1, S2, S3: y_out = ~x_in;
        endcase
endmodule

module t_Mealy_Zero_Detector;
    wire t_y_out;
    reg t_x_in, t_clock, t_reset;

    Mealy_Zero_Detector M0 (t_y_out, t_x_in, t_clock, t_reset);
    initial #200 $finish;
    initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

    initial fork
        t_reset = 0;
        #2 t_reset = 1;
        #87 t_reset = 0;
        #89 t_reset = 1;

```



```

#10 t_x_in = 1;
#30 t_x_in = 0;
#40 t_x_in = 1;
#50 t_x_in = 0;
#52 t_x_in = 1;
#54 t_x_in = 0;
#70 t_x_in = 1;
#80 t_x_in = 1;
#70 t_x_in = 0;
#90 t_x_in = 1;
#100 t_x_in = 0;
#120 t_x_in = 1;
#160 t_x_in = 0;
#170 t_x_in = 1;
join
endmodule

```

The circuit I HDL Example 5.5 detects a 0 following a sequence of 1s in a serial bit stream. Its Verilog model uses three **always** blocks that execute concurrently and interact through common variables. The first **always** statement resets the circuit to the initial state $S0 = 00$ and specifies the synchronous clocked operation. The statement $state \leq next_state$ is synchronized to a positive-edge transition of the clock. This means that any change in the value of $next_state$ in the second **always** block can affect the value of $state$ only as a result of a **posedge** event of $clock$. The second **always** block determines the value of the next state transition as a function of the present state and input. The value assigned to $state$ by the nonblocking assignment is the value of $next_state$ immediately before the rising edge of $clock$. Notice how the multiway branch condition implements the state transitions specified by the annotated edges in the state diagram of Fig. 5.16. The third **always** block specifies the output as a function of the present state and the input. Although this block is listed as a separate behavior for clarity, it could be combined with the second block. Note that the value of output y_out may change if the value of input x_in changes while the circuit is in any given state.

So let's summarize how the model describes the behavior of the machine: At every rising edge of $clock$, if $reset$ is not asserted, the state of the machine is updated by the first **always** block; when $state$ is updated by the first **always** block, the change in $state$ is detected by the sensitivity list mechanism of the second **always** block; then the second **always** block updates the value of $next_state$ (it will be used by the first **always** block at the next tick of the clock); the third **always** block also detects the change in $state$ and updates the value of the output. In addition, the second and third **always** blocks detect changes in x_in and update $next_state$ and y_out accordingly. The test bench provided with *Mealy_Zero_Detector* provides some waveforms to stimulate the model, producing the results shown in Fig. 5.22. Notice how t_y_out responds to changes in both the state and the input, and has a glitch (a transient logic value). We display both to $state[1:0]$ and $next_state[1:0]$ to illustrate how changes in t_x_in influence the value of $next_state$ and t_y_out . The Mealy glitch in t_y_out is due to the (intentional) dynamic behavior of t_x_in . The input, t_x_in , settles

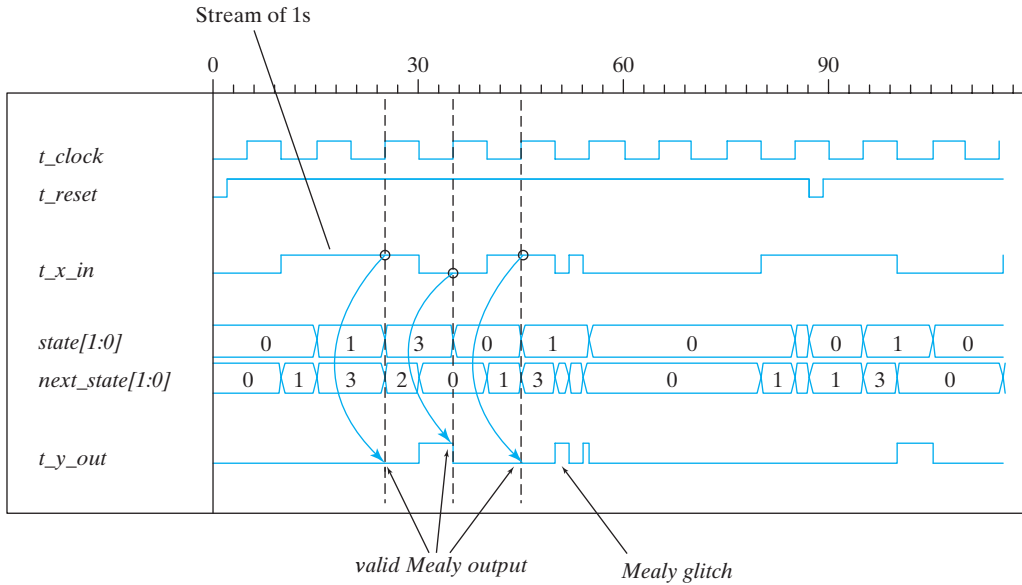


FIGURE 5.22
Simulation output of *Mealy_Zero_Detector*

to a value of 0 immediately before the clock, and at the clock, the state makes a transition from 0 to 1, which is consistent with Fig. 5.16. The output is 1 in state *S1* immediately before the clock, and changes to 0 as the state enters *S0*.

The description of waveforms in the test bench uses the **fork ... join** construct. Statements with the **fork ... join** block execute in parallel, so the time delays are relative to a common reference of $t = 0$, the time at which the block begins execution.² It is usually more convenient to use the **fork ... join** block instead of the **begin ... end** block in describing waveforms. Notice that the waveform of reset is triggered “on the fly” to demonstrate that the machine recovers from an unexpected (asynchronous) reset condition during any state.

How does our Verilog model *Mealy_Zero_Detector* correspond to hardware? The first **always** block corresponds to a *D* flip-flop implementation of the state register in Fig. 5.21; the second **always** block is the combinational logic block describing the next state; the third **always** block describes the output combinational logic of the zero-detecting Mealy machine. The register operation of the state transition uses the nonblocking assignment operator (\leq) because the (edge-sensitive) flip-flops of a sequential machine are updated concurrently by a common clock. The second and third **always** blocks describe combinational logic, which is level sensitive, so they use the blocking ($=$) assignment operator.

²A **fork ... join** block completes execution when the last executing statement within it completes its execution.

Their sensitivity lists include both the state and the input because their logic must respond to a change in either or both of them.

Note: The modeling style illustrated by *Mealy_Zero_Detector* is commonly used by designers because it has a close relationship to the state diagram of the machine that is being described. Notice that the reset signal is associated with the **always** block that synchronizes the state transitions. In this example, it is modeled as an active-low reset. Because the reset condition is included in the description of the state transitions, there is no need to include it in the combinational logic that specifies the next state and the output, and the resulting description is less verbose, simpler, and more readable.

HDL Example 5.6 presents the Verilog behavioral model of the Moore FSM shown in Fig. 5.18 and having the state diagram given in Fig. 5.19. The model illustrates an alternative style in which the state transitions of the machine are described by a single clocked (i.e., edge-sensitive) cyclic behavior, i.e., by one **always** block. The present state of the circuit is identified by the variable *state*, and its transitions are triggered by the rising edge of the clock according to the conditions listed in the **case** statement. The combinational logic that determines the next state is included in the nonblocking assignment to *state*. In this example, the output of the circuits is independent of the input and is taken directly from the outputs of the flip-flops. The two-bit output *y_out* is specified with a continuous assignment statement and is equal to the value of the present state vector. Figure 5.23 shows some simulation results for *Moore_Model_Fig_5_19*. Here are some important observations: (1) the output depends on only the state, (2) reset “on-the-fly” forces the state of the machine back to S0 (00), and (3) the state transitions are consistent with Fig. 5.19.

HDL Example 5.6 (Moore Machine: Zero Detector)

```
// Moore model FSM (see Fig. 5.19)                                Verilog 2001, 2005 syntax
module Moore_Model_Fig_5_19 (
    output [1: 0]      y_out,
    input             x_in, clock, reset
);
    reg [1: 0]        state;
    parameter        S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
    always @ (posedge clock, negedge reset)
        if (reset == 0) state <= S0;                                // Initialize to state S0
        else case (state)
            S0:    if (~x_in) state <= S1; else state <= S0;
            S1:    if (x_in)  state <= S2; else state <= S3;
            S2:    if (~x_in) state <= S3; else state <= S2;
            S3:    if (x_in)  state <= S0; else state <= S1;
        endcase
    assign y_out = state;      // Output of flip-flops
endmodule
```

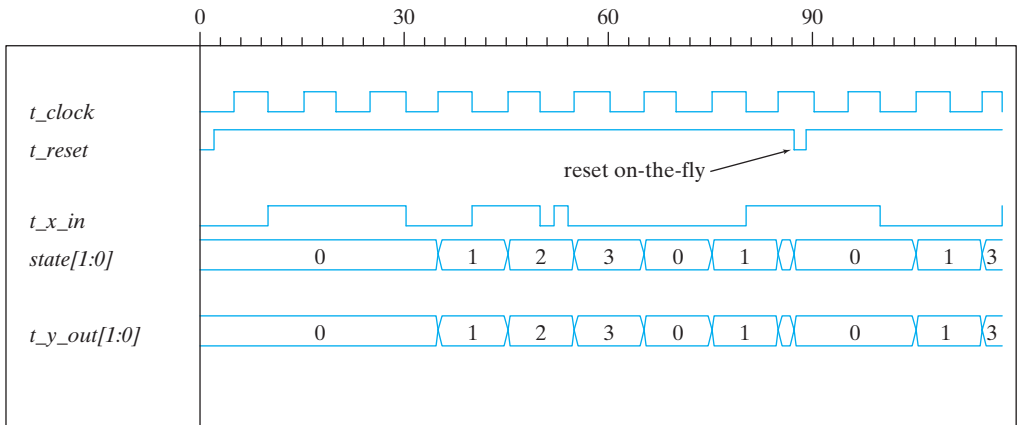


FIGURE 5.23
Simulation output of HDL Example 5.6

Structural Description of Clocked Sequential Circuits

Combinational logic circuits can be described in Verilog by a connection of gates (primitives and UDPs), by dataflow statements (continuous assignments), or by level-sensitive cyclic behaviors (**always** blocks). Sequential circuits are composed of combinational logic and flip-flops, and their HDL models use sequential UDPs and behavioral statements (edge-sensitive cyclic behaviors) to describe the operation of flip-flops. One way to describe a sequential circuit uses a combination of dataflow and behavioral statements. The flip-flops are described with an **always** statement. The combinational part can be described with **assign** statements and Boolean equations. The separate modules can be combined to form a structural model by instantiation within a **module**.

The structural description of a Moore-type zero detector sequential circuit is shown in HDL Example 5.7. We want to encourage the reader to consider alternative ways to model a circuit, so as a point of comparison, we first present *Moore_Model_Fig_5_20*, a Verilog behavioral description of a binary counter having the state diagram examined earlier shown in Fig. 5.20(b). This style of modeling follows directly from the state diagram. An alternative style, used in *Moore_Model_STR_Fig_5_20*, represents the structure shown in Fig. 5.20(a). This style uses two modules. The first describes the circuit of Fig. 5.20(a). The second describes the *T* flip-flop that will be used by the circuit. We also show two ways to model the *T* flip-flop. The first asserts that, at every clock tick, the value of the output of the flip-flop toggles if the toggle input is asserted. The second model describes the behavior of the toggle flip-flop in terms of its characteristic equation. The first style is attractive because it does not require the reader to remember the characteristic equation. Nonetheless, the models are interchangeable and will synthesize to the same hardware circuit. A test bench module provides a stimulus for verifying the functionality of the circuit. The sequential circuit is a two-bit binary counter controlled by input *x_in*. The output, *y_out*, is enabled when the count reaches binary 11. Flip-flops

A and B are included as outputs in order to check their operation. The flip-flop input equations and the output equation are evaluated with continuous assignment (**assign**) statements having the corresponding Boolean expressions. The instantiated T flip-flops use TA and TB as defined by the input equations.

The second module describes the T flip-flop. The *reset* input resets the flip-flop to 0 with an active-low signal. The operation of the flip-flop is specified by its characteristic equation, $Q(t + 1) = Q \oplus T$.

The test bench includes both models of the machine. The stimulus module provides common inputs to the circuits to simultaneously display their output responses. The first **initial** block provides eight clock cycles with a period of 10 ns. The second **initial** block specifies a toggling of input x_{in} that occurs at the negative edge transition of the clock. The result of the simulation is shown in Fig. 5.24. The pair (A, B) goes through the binary sequence 00, 01, 10, 11, and back to 00. The change in the count is triggered by a positive edge of the clock, provided that $x_{in} = 1$. If $x_{in} = 0$, the count does not change. y_{out} is equal to 1 when both A and B are equal to 1. This verifies the main functionality of the circuit, but not a recovery from an unexpected reset event.

HDL Example 5.7 (Binary Counter_Moore Model)

```
// State-diagram-based model (V2001, 2005)
module Moore_Model_Fig_5_20 (
    output y_out,
    input x_in, clock, reset
);
    reg [1: 0] state;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
    always @ (posedge clock, negedge reset)
        if (reset == 0) state <= S0; // Initialize to state S0
        else case (state)
            S0: if (x_in) state <= S1; else state <= S0;
            S1: if (x_in) state <= S2; else state <= S1;
            S2: if (x_in) state <= S3; else state <= S2;
            S3: if (x_in) state <= S0; else state <= S3;
        endcase
    assign y_out = (state == S3); // Output of flip-flops
endmodule

// Structural model
module Moore_Model_STR_Fig_5_20 (
    output y_out, A, B,
    input x_in, clock, reset
);
    wire TA, TB;

    // Flip-flop input equations
    assign TA = x_in & B;
```

```

    assign TB = x_in;
// Output equation
    assign y_out = A & B;

// Instantiate Toggle flip-flops
    Toggle_flip_flop_3 M_A (A, TA, clock, reset);
    Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule

module Toggle_flip_flop (Q, T, CLK, RST_b);
    output    Q;
    input     T, CLK, RST_b;
    reg       Q;

    always @ (posedge CLK, negedge RST_b)
        if (RST_b == 0) Q <= 1'b0;
        else if (T) Q <= ~Q;
endmodule

// Alternative model using characteristic equation
// module Toggle_flip_flop (Q, T, CLK, RST_b);
// output    Q;
// input     T, CLK, RST_b;
// reg       Q;

// always @ (posedge CLK, negedge RST)
// if (RST_b == 0) Q <= 1'b0;
// else Q <= Q ^ T;
// endmodule

module t_Moore_Fig_5_20;
    wire      t_y_out_2, t_y_out_1;
    reg       t_x_in, t_clock, t_reset;

    Moore_Model_Fig_5_20          M1(t_y_out_1, t_x_in, t_clock, t_reset);
    Moore_Model_STR_Fig_5_20      M2(t_y_out_2, A, B, t_x_in, t_clock, t_reset);

    initial #200 $finish;
    initial begin
        t_reset = 0;
        t_clock = 0;
        #5 t_reset = 1;
        repeat (16)
            #5 t_clock = ~t_clock;
    end
    initial begin
        t_x_in = 0;
        #15 t_x_in = 1;
        repeat (8)
            #10 t_x_in = ~t_x_in;
    end
end
endmodule

```

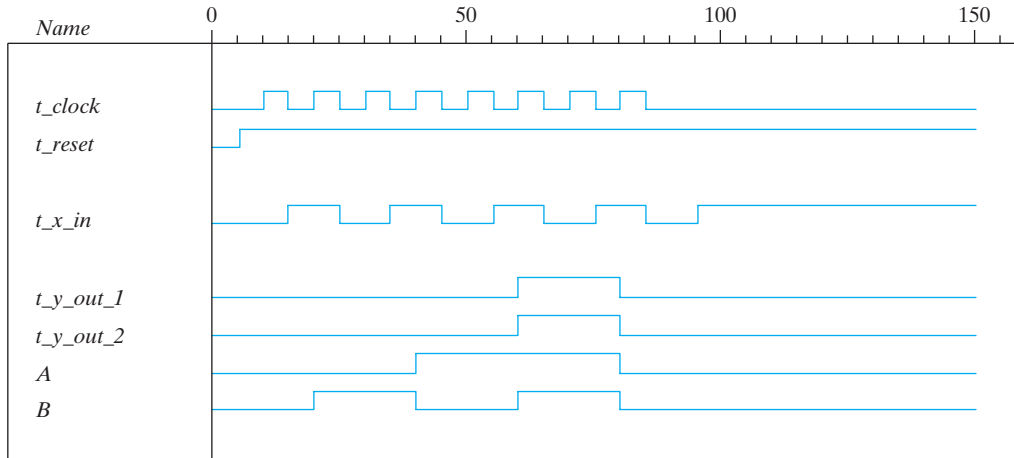


FIGURE 5.24
Simulation output of HDL Example 5.7

5.7 STATE REDUCTION AND ASSIGNMENT

The *analysis* of sequential circuits starts from a circuit diagram and culminates in a state table or diagram. The *design* (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Design procedures are presented in Section 5.8. Two sequential circuits may exhibit the same input–output behavior, but have a different number of internal states in their state diagram. The current section discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flip-flops reduces the cost of a circuit.

State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the *state-reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged. Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates to realize its next state and output logic.

We will illustrate the state-reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram of Fig. 5.25. In our example, only the input–output sequences are important; the internal states are used merely to provide the required sequences. For that reason, the states marked inside the circles are denoted

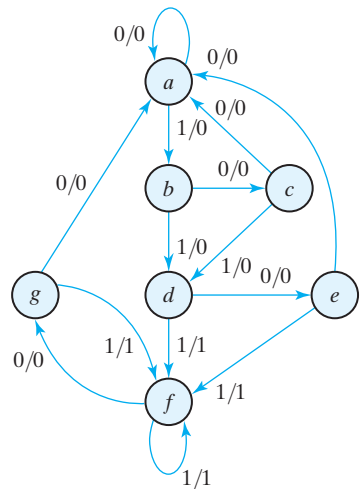


FIGURE 5.25
State diagram

by letter symbols instead of their binary values. This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence 01010110100 starting from the initial state *a*. Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state *a*, an input of 0 produces an output of 0 and the circuit remains in state *a*. With present state *a* and an input of 1, the output is 0 and the next state is *b*. With present state *b* and an input of 0, the output is 0 and the next state is *c*. Continuing this process, we find the complete sequence to be as follows:

state	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>f</i>	<i>g</i>	<i>f</i>	<i>g</i>	<i>a</i>
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

In each column, we have the present state, input value, and output value. The next state is written on top of the next column. It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Now let us assume that we have found a sequential circuit whose state diagram has fewer than seven states, and suppose we wish to compare this circuit with the circuit whose state diagram is given by Fig. 5.25. If identical input sequences are applied to the two circuits and identical outputs occur for all input sequences, then the two circuits are said to be equivalent (as far as the input–output is concerned) and one may be replaced by the other. The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

We now proceed to reduce the number of states for this example. First, we need the state table; it is more convenient to apply procedures for state reduction with the use of a table rather than a diagram. The state table of the circuit is listed in Table 5.6 and is obtained directly from the state diagram.

The following algorithm for the state reduction of a completely specified state table is given here without proof: “Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.” When two states are equivalent, one of them can be removed without altering the input–output relationships.

Now apply this algorithm to Table 5.6. Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations. States *e* and *g* are two such states: They both go to states *a* and *f* and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states *g* and *e* are equivalent, and one of these states can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7. The row with present state *g* is removed, and state *g* is replaced by state *e* each time it occurs in the columns headed “Next State.”

Present state *f* now has next states *e* and *f* and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively. The same next states and outputs appear in the row with present state *d*. Therefore, states *f* and *d* are equivalent, and state *f* can be removed and replaced by *d*. The final reduced table is shown in Table 5.8. The state diagram for the reduced table consists of only five states and is shown in Fig. 5.26. This state diagram satisfies the original input–output specifications and will produce the required output sequence for any given input sequence. The following list derived from the state diagram of Fig. 5.26 is for the input sequence used previously (note that the same output sequence results, although the state sequence is different):

state	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>d</i>	<i>d</i>	<i>e</i>	<i>d</i>	<i>e</i>	<i>a</i>
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Table 5.6
State Table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
<i>a</i>	<i>a</i>	<i>b</i>	0	0
<i>b</i>	<i>c</i>	<i>d</i>	0	0
<i>c</i>	<i>a</i>	<i>d</i>	0	0
<i>d</i>	<i>e</i>	<i>f</i>	0	1
<i>e</i>	<i>a</i>	<i>f</i>	0	1
<i>f</i>	<i>g</i>	<i>f</i>	0	1
<i>g</i>	<i>a</i>	<i>f</i>	0	1

Table 5.7
Reducing the State Table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Table 5.8
Reduced State Table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

In fact, this sequence is exactly the same as that obtained for Fig. 5.25 if we replace g by e and f by d .

Checking each pair of states for equivalency can be done systematically by means of a procedure that employs an implication table, which consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table.

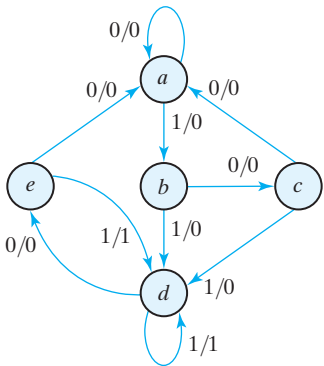


FIGURE 5.26
Reduced state diagram

The sequential circuit of this example was reduced from seven to five states. In general, reducing the number of states in a state table may result in a circuit with less equipment. However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates. In actual practice designers may skip this step because target devices are rich in resources.

State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m states, the codes must contain n bits, where $2^n \geq m$. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111. If the state table of Table 5.6 is used, we must assign binary values to seven states; the remaining state is unused. If the state table of Table 5.8 is used, only five states need binary assignment, and we are left with three unused states. Unused states are treated as don't-care conditions during the design. Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely but not certain that the circuit with five states will require fewer combinational gates than the one with seven states.

The simplest way to code five states is to use the first five integers in binary counting order, as shown in the first assignment of Table 5.9. Another similar assignment is the Gray code shown in assignment 2. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification. Another possible assignment often used in the design of state machines to control data-path units is the one-hot assignment. This configuration uses as many bits as there are states in the circuit. At any given time, only one bit is equal to 1 while all others are kept at 0. This type of assignment uses one flip-flop per state, which is not an issue for register-rich field-programmable gate arrays. (See Chapter 7.) *One-hot encoding usually leads to simpler decoding logic for the next state and output.* One-hot machines can be faster than machines with sequential binary encoding, and the silicon area required by the extra flip-flops can be offset by the area

Table 5.9
Three Possible Binary State Assignments

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
<i>a</i>	000	000	00001
<i>b</i>	001	001	00010
<i>c</i>	010	011	00100
<i>d</i>	011	010	01000
<i>e</i>	100	110	10000

Table 5.10
Reduced State Table with Binary Assignment 1

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1

saved by using simpler decoding logic. This trade-off is not guaranteed, so it must be evaluated for a given design.

Table 5.10 is the reduced state table with binary assignment 1 substituted for the letter symbols of the states. A different assignment will result in a state table with different binary values for the states. The binary form of the state table is used to derive the next-state and output-forming combinational logic part of the sequential circuit. The complexity of the combinational circuit depends on the binary state assignment chosen.

Sometimes, the name *transition table* is used for a state table with a binary assignment. This convention distinguishes it from a state table with symbolic names for the states. In this book, we use the same name for both types of state tables.

5.8 DESIGN PROCEDURE

Design procedures or methodologies specify hardware that will implement a desired behavior. The design effort for small circuits may be manual, but industry relies on automated synthesis tools for designing massive integrated circuits. The sequential building block used by synthesis tools is the *D* flip-flop. Together with additional logic, it can implement the behavior of *JK* and *T* flip-flops. In fact, designers generally do not concern themselves with the type of flip-flop; rather, their focus is on correctly describing the sequential functionality that is to be implemented by the synthesis tool. Here we will illustrate manual methods using *D*, *JK*, and *T* flip-flops.

The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained. In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalent representation, such as a state diagram.³

A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding a combinational

³We will examine later another important representation of a machine's behavior—the algorithmic state machine (ASM) chart.

gate structure that, together with the flip-flops, produces a circuit which fulfills the stated specifications. The number of flip-flops is determined from the number of states needed in the circuit and the choice of state assignment codes. The combinational circuit is derived from the state table by evaluating the flip-flop input equations and output equations. In fact, once the type and number of flip-flops are determined, the design process involves a transformation from a sequential circuit problem into a combinational circuit problem. In this way, the techniques of combinational circuit design can be applied.

The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:

1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be used.
6. Derive the simplified flip-flop input equations and output equations.
7. Draw the logic diagram.

The word specification of the circuit behavior usually assumes that the reader is familiar with digital logic terminology. It is necessary that the designer use intuition and experience to arrive at the correct interpretation of the circuit specifications, because word descriptions may be incomplete and inexact. Once such a specification has been set down and the state diagram obtained, it is possible to use known synthesis procedures to complete the design. Although there are formal procedures for state reduction and assignment (steps 2 and 3), they are seldom used by experienced designers. Steps 4 through 7 in the design can be implemented by exact algorithms and therefore can be automated. The part of the design that follows a well-defined procedure is referred to as *synthesis*. Designers using logic synthesis tools (software) can follow a simplified process that develops an HDL description directly from a state diagram, letting the synthesis tool determine the circuit elements and structure that implement the description.

The first step is a critical part of the process, because succeeding steps depend on it. We will give one simple example to demonstrate how a state diagram is obtained from a word specification.

Suppose we wish to design a circuit that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line (i.e., the input is a *serial bit stream*). The state diagram for this type of circuit is shown in Fig. 5.27. It is derived by starting with state S_0 , the reset state. If the input is 0, the circuit stays in S_0 , but if the input is 1, it goes to state S_1 to indicate that a 1 was detected. If the next input is 1, the change is to state S_2 to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to S_0 . The third consecutive 1 sends the circuit to state S_3 . If more 1's are detected, the circuit stays in S_3 . Any 0 input sends the circuit back to S_0 . In this way, the circuit stays in S_3 as long as there are three or more consecutive 1's received. This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S_3 and is 0 otherwise.

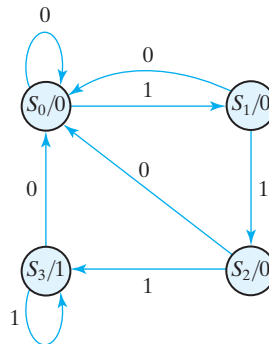


FIGURE 5.27
State diagram for sequence detector

Synthesis Using *D* Flip-Flops

Once the state diagram has been derived, the rest of the design follows a straightforward synthesis procedure. In fact, we can design the circuit by using an HDL description of the state diagram and the proper HDL synthesis tools to obtain a synthesized netlist. (The HDL description of the state diagram will be similar to HDL Example 5.6 in Section 5.6.) To design the circuit by hand, we need to assign binary codes to the states and list the state table. This is done in Table 5.11. The table is derived from the state diagram of Fig. 5.27 with a sequential binary assignment. We choose two *D* flip-flops to represent the four states, and we label their outputs *A* and *B*. There is one input *x* and one output *y*. The characteristic equation of the *D* flip-flop is $Q(t + 1) = D_Q$, which means that the next-state values in the state table specify the *D* input condition for the flip-flop. The flip-flop input equations

Table 5.11
State Table for Sequence Detector

Present State		Input	Next State		Output
<i>A</i>	<i>B</i>		<i>A</i>	<i>B</i>	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

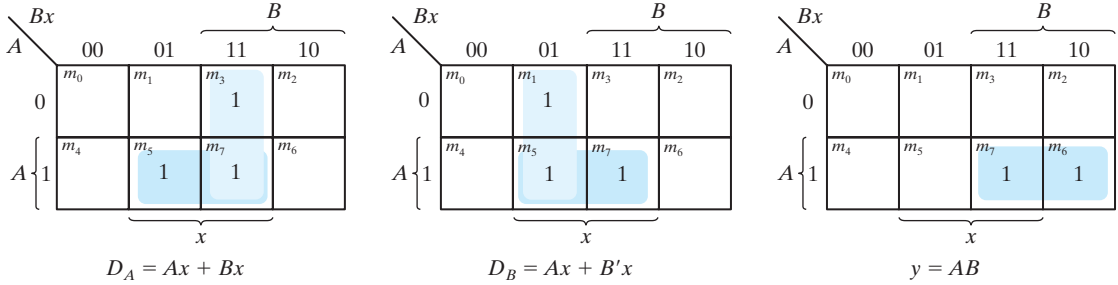


FIGURE 5.28
K-Maps for sequence detector

can be obtained directly from the next-state columns of A and B and expressed in sum-of-minterms form as

$$A(t + 1) = D_A(A, B, x) = \Sigma(3, 5, 7)$$

$$B(t + 1) = D_B(A, B, x) = \Sigma(1, 5, 7)$$

$$y(A, B, x) = \Sigma(6, 7)$$

where A and B are the present-state values of flip-flops A and B , x is the input, and D_A and D_B are the input equations. The minterms for output y are obtained from the output column in the state table.

The Boolean equations are simplified by means of the maps plotted in Fig. 5.28. The simplified equations are

$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

$$y = AB$$

The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table. Software tools automatically infer and select the D -type flip-flop from a properly written HDL model. The schematic of the sequential circuit is drawn in Fig. 5.29.

Excitation Tables

The design of a sequential circuit with flip-flops other than the D type is complicated by the fact that the input equations for the circuit must be derived indirectly from the state table. When D -type flip-flops are employed, the input equations are obtained directly from the next state. This is not the case for the JK and T types of flip-flops. In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations.

The flip-flop characteristic tables presented in Table 5.1 provide the value of the next state when the inputs and the present state are known. These tables are useful

**FIGURE 5.29**

for analyzing sequential circuits and for defining the operation of the flip-flops. During the design process, we usually know the transition from the present state to the next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an *excitation table*.

Table 5.12 shows the excitation tables for the two flip-flops (JK and T). Each table has a column for the present state $Q(t)$, a column for the next state $Q(t + 1)$, and a column for each input to show how the required transition is achieved. There are four possible transitions from the present state to the next state. The required input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol X in the tables represents a don't-care condition, which means that it does not matter whether the input is 1 or 0.

The excitation table for the JK flip-flop is shown in part (a). When both present state and next state are 0, the J input must remain at 0 and the K input can be either 0 or 1. Similarly, when both present state and next state are 1, the K input must remain at 0,

Table 5.12
Flip-Flop Excitation Tables

$Q(t)$	$Q(t = 1)$	J	K	$Q(t)$	$Q(t = 1)$	T
0	0	0	X	0	0	0
0	1	1	X	0	1	1
1	0	X	1	1	0	1
1	1	X	0	1	1	0

(a) *JK* Flip-Flop
(b) *T* Flip-Flop

while the J input can be 0 or 1. If the flip-flop is to have a transition from the 0-state to the 1-state, J must be equal to 1, since the J input sets the flip-flop. However, input K may be either 0 or 1. If $K = 0$, the $J = 1$ condition sets the flip-flop as required; if $K = 1$ and $J = 1$, the flip-flop is complemented and goes from the 0-state to the 1-state as required. Therefore, the K input is marked with a don't-care condition for the 0-to-1 transition. For a transition from the 1-state to the 0-state, we must have $K = 1$, since the K input clears the flip-flop. However, the J input may be either 0 or 1, since $J = 0$ has no effect and $J = 1$ together with $K = 1$ complements the flip-flop with a resultant transition from the 1-state to the 0-state.

The excitation table for the T flip-flop is shown in part (b). From the characteristic table, we find that when input $T = 1$, the state of the flip-flop is complemented, and when $T = 0$, the state of the flip-flop remains unchanged. Therefore, when the state of the flip-flop must remain the same, the requirement is that $T = 0$. When the state of the flip-flop has to be complemented, T must equal 1.

Synthesis Using *JK* Flip-Flops

The manual synthesis procedure for sequential circuits with *JK* flip-flops is the same as with *D* flip-flops, except that the input equations must be evaluated from the present-state to the next-state transition derived from the excitation table. To illustrate the procedure, we will synthesize the sequential circuit specified by Table 5.13. In addition to having columns for the present state, input, and next state, as in a conventional state table, the table shows the flip-flop input conditions from which the input equations are derived. The flip-flop inputs are derived from the state table in conjunction with the excitation table for the *JK* flip-flop. For example, in the first row of Table 5.13, we have a transition for flip-flop A from 0 in the present state to 0 in the next state. In Table 5.12, for the *JK* flip-flop, we find that a transition of states from present state 0 to next state 0 requires that input J be 0 and input K be a don't-care. So 0 and X are entered in the first row under J_A and K_A , respectively. Since the first row also shows a transition for flip-flop B from 0 in the present state to 0 in the next state, 0 and X are inserted into the first row under J_B and K_B , respectively. The second row of the table shows a transition for flip-flop B from 0 in the present state to 1 in the next state. From the excitation table, we find that a transition from 0 to 1 requires that J be 1 and K be a don't-care, so 1 and X are copied into

Table 5.13
State Table and JK Flip-Flop Inputs

Present State		Input	Next State		Flip-Flop Inputs			
A	B		A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1

the second row under J_B and K_B , respectively. The process is continued for each row in the table and for each flip-flop, with the input conditions from the excitation table copied into the proper row of the particular flip-flop being considered.

The flip-flop inputs in Table 5.13 specify the truth table for the input equations as a function of present state A , present state B , and input x . The input equations are simplified in the maps of Fig. 5.30. The next-state values are not used during the simplification,

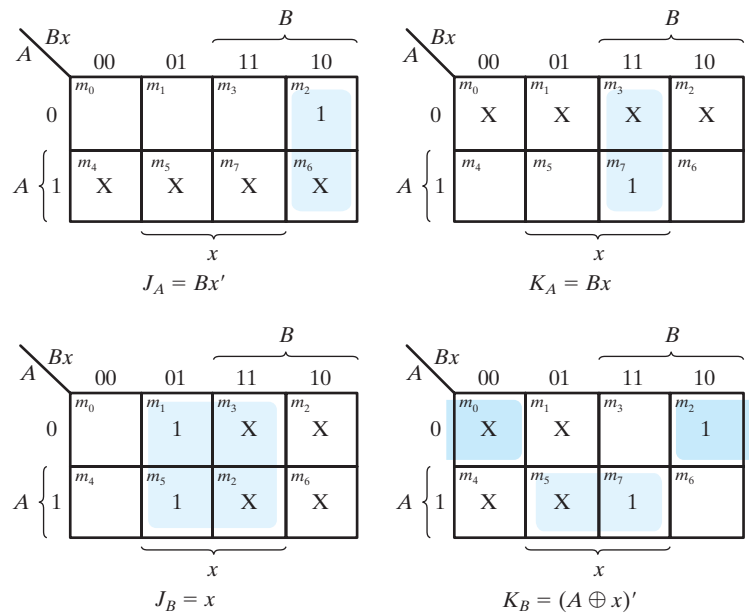


FIGURE 5.30
Maps for J and K input equations

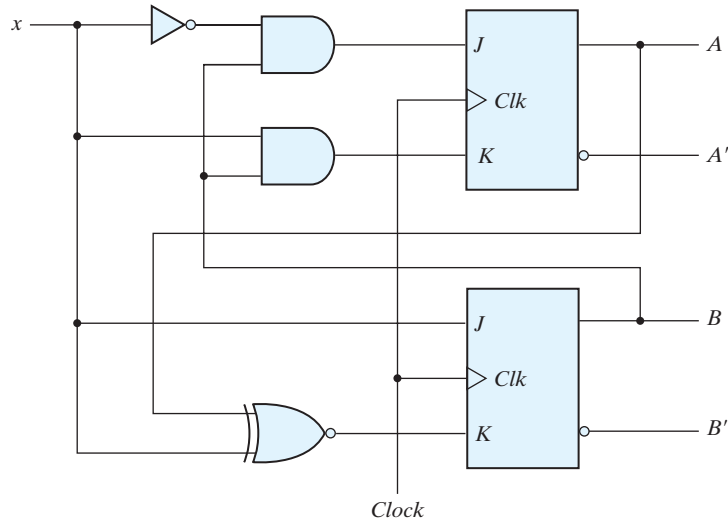


FIGURE 5.31
Logic diagram for sequential circuit with JK flip-flops

since the input equations are a function of the present state and the input only. Note the advantage of using JK -type flip-flops when sequential circuits are designed *manually*. The fact that there are so many don't-care entries indicates that the combinational circuit for the input equations is likely to be simpler, because don't-care minterms usually help in obtaining simpler expressions. If there are unused states in the state table, there will be additional don't-care conditions in the map. Nonetheless, D -type flip-flops are more amenable to an automated design flow.

The four input equations for the pair of JK flip-flops are listed under the maps of Fig. 5.30. The logic diagram (schematic) of the sequential circuit is drawn in Fig. 5.31.

Synthesis Using T Flip-Flops

The procedure for synthesizing circuits using T flip-flops will be demonstrated by designing a binary counter. An n -bit binary counter consists of n flip-flops that can count in binary from 0 to $2^n - 1$. The state diagram of a three-bit counter is shown in Fig. 5.32. As

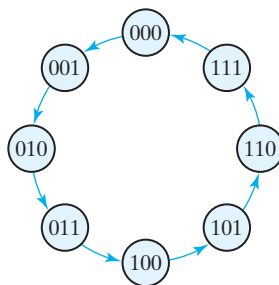


FIGURE 5.32
State diagram of three-bit binary counter

seen from the binary states indicated inside the circles, the flip-flop outputs repeat the binary count sequence with a return to 000 after 111. The directed lines between circles are not marked with input and output values as in other state diagrams. Remember that state transitions in clocked sequential circuits are initiated by a clock edge; the flip-flops remain in their present states if no clock is applied. For that reason, the clock does not appear explicitly as an input variable in a state diagram or state table. From this point of view, the state diagram of a counter does not have to show input and output values along the directed lines. The only input to the circuit is the clock, and the outputs are specified by the present state of the flip-flops. The next state of a counter depends entirely on its present state, and the state transition occurs every time the clock goes through a transition.

Table 5.14 is the state table for the three-bit binary counter. The three flip-flops are symbolized by A_2 , A_1 , and A_0 . Binary counters are constructed most efficiently with T flip-flops because of their complement property. The flip-flop excitation for the T inputs is derived from the excitation table of the T flip-flop and by inspection of the state transition of the present state to the next state. As an illustration, consider the flip-flop input entries for row 001. The present state here is 001 and the next state is 010, which is the next count in the sequence. Comparing these two counts, we note that A_2 goes from 0 to 0, so T_{A_2} is marked with 0 because flip-flop A_2 must not change when a clock occurs. Also, A_1 goes from 0 to 1, so T_{A_1} is marked with a 1 because this flip-flop must be complemented in the next clock edge. Similarly, A_0 goes from 1 to 0, indicating that it must be complemented, so T_{A_0} is marked with a 1. The last row, with present state 111, is compared with the first count 000, which is its next state. Going from all 1's to all 0's requires that all three flip-flops be complemented.

The flip-flop input equations are simplified in the maps of Fig. 5.33. Note that T_{A_0} has 1's in all eight minterms because the least significant bit of the counter is complemented with each count. A Boolean function that includes all minterms defines a constant value of 1. The input equations listed under each map specify the combinational part of the counter. Including these functions with the three flip-flops, we obtain

Table 5.14
State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A_2}	T_{A_1}	T_{A_0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

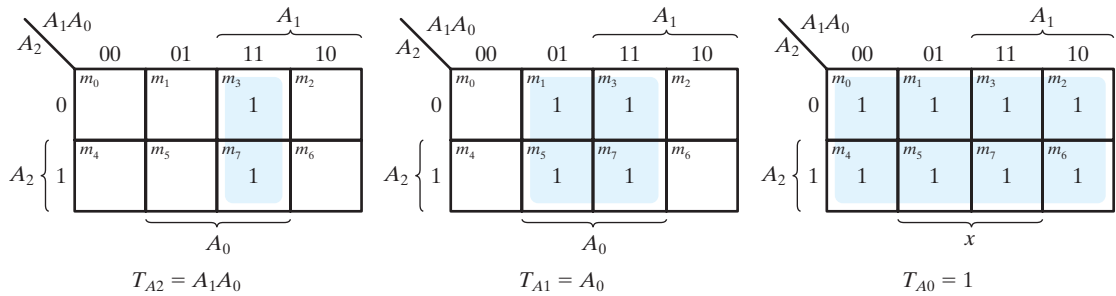


FIGURE 5.33
Maps for three-bit binary counter

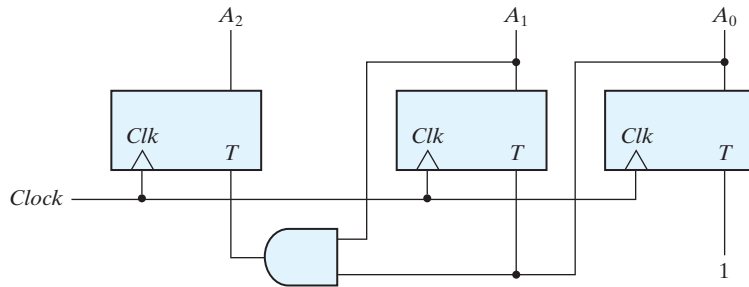


FIGURE 5.34
Logic diagram of three-bit binary counter

the logic diagram of the counter, as shown in Fig. 5.34. For simplicity, the reset signal is not shown, but be aware that every design should include a reset signal.

PROBLEMS

(Answers to problems marked with * appear at the end of the book. Where appropriate, a logic design and its related HDL modeling problem are cross-referenced.)

Note: For each problem that requires writing and verifying an HDL model, a test plan should be written to identify which functional features are to be tested during the simulation and how they will be tested. For example, a reset on the fly could be tested by asserting the reset signal while the simulated machine is in a state other than the reset state. The test plan is to guide development of a test bench that will implement the plan. Simulate the model, using the test bench, and verify that the behavior is correct. If synthesis tools and an ASIC cell library are available, the Verilog descriptions developed for Problems 5.34–5.42 can be assigned as synthesis exercises. The gate-level circuit produced by the synthesis tools should be simulated and compared to the simulation results for the pre-synthesis model. The same exercises can be assigned if an FPGA tool suite is available.

- 5.1** The D latch of Fig. 5.6 is constructed with four NAND gates and an inverter. Consider the following three other ways for obtaining a D latch. In each case, draw the logic diagram and verify the circuit operation.
- Use NOR gates for the SR latch part and AND gates for the other two. An inverter may be needed.
 - Use NOR gates for all four gates. Inverters may be needed.
 - Use four NAND gates only (without an inverter). This can be done by connecting the output of the upper gate in Fig. 5.6 (the gate that goes to the SR latch) to the input of the lower gate (instead of the inverter output).
- 5.2** Construct a JK flip-flop using a D flip-flop, a two-to-one-line multiplexer, and an inverter. (HDL—see Problem 5.34.)
- 5.3** Show that the characteristic equation for the complement output of a JK flip-flop is
- $$Q'(t + 1) = J'Q' + KQ$$
- 5.4** A PN flip-flop has four operations: clear to 0, no change, complement, and set to 1, when inputs P and N are 00, 01, 10, and 11, respectively.
- Tabulate the characteristic table.
 - * Derive the characteristic equation.
 - Tabulate the excitation table.
 - Show how the PN flip-flop can be converted to a D flip-flop.
- 5.5** Explain the differences among a truth table, a state table, a characteristic table, and an excitation table. Also, explain the difference among a Boolean equation, a state equation, a characteristic equation, and a flip-flop input equation.
- 5.6** A sequential circuit with two D flip-flops A and B , two inputs, x and y ; and one output z is specified by the following next-state and output equations (HDL—see Problem 5.35):

$$A(t + 1) = xy' + xB$$

$$B(t + 1) = xA + xB'$$

$$z = A$$

- Draw the logic diagram of the circuit.
 - List the state table for the sequential circuit.
 - Draw the corresponding state diagram.
- 5.7*** A sequential circuit has one flip-flop Q , two inputs x and y , and one output S . It consists of a full-adder circuit connected to a D flip-flop, as shown in Fig. P5.7. Derive the state table and state diagram of the sequential circuit.

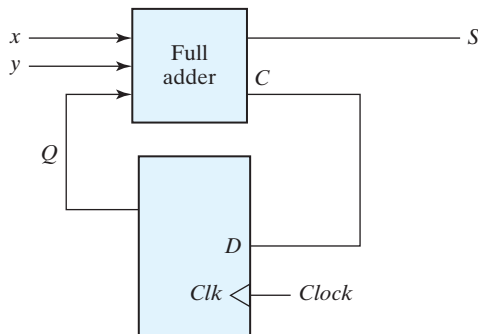


FIGURE P5.7

- 5.8*** Derive the state table and the state diagram of the sequential circuit shown in Fig. P5.8. Explain the function that the circuit performs. (HDL—see Problem 5.36.)

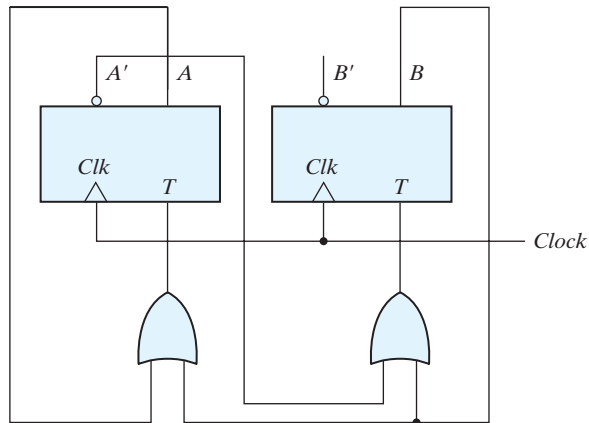


FIGURE P5.8

- 5.9** A sequential circuit has two *JK* flip-flops *A* and *B* and one input *x*. The circuit is described by the following flip-flop input equations:

$$J_A = x \quad K_A = B$$

$$J_B = x \quad K_B = A'$$

- Derive the state equations $A(t+1)$ and $B(t+1)$ by substituting the input equations for the *J* and *K* variables.
- Draw the state diagram of the circuit.

- 5.10** A sequential circuit has two *JK* flip-flops *A* and *B*, two inputs *x* and *y*, and one output *z*. The flip-flop input equations and circuit output equation are

$$J_A = Bx + B'y' \quad K_A = B'xy'$$

$$J_B = A'x \quad K_B = A + xy'$$

$$z = Ax'y' + Bx'y'$$

- Draw the logic diagram of the circuit.
- Tabulate the state table.
- Derive the state equations for *A* and *B*.

- 5.11** For the circuit described by the state diagram of Fig. 5.16,

- * Determine the state transitions and output sequence that will be generated when an input sequence of 01011011101110 is applied to the circuit and it is initially in the state 00.
- Find all of the equivalent states in Fig. 5.16 and draw a simpler, but equivalent, state diagram.
- Using *D* flip-flops, design the equivalent machine (including its logic diagram) described by the state diagram in (b).

5.12 For the following state table

Present State	Next State		Output	
	$x = 0$	$x = 1$	$x = 0$	$x = 1$
<i>a</i>	<i>f</i>	<i>b</i>	0	0
<i>b</i>	<i>d</i>	<i>c</i>	0	0
<i>c</i>	<i>f</i>	<i>e</i>	0	0
<i>d</i>	<i>g</i>	<i>a</i>	1	0
<i>e</i>	<i>d</i>	<i>c</i>	0	0
<i>f</i>	<i>f</i>	<i>b</i>	1	1
<i>g</i>	<i>g</i>	<i>h</i>	0	1
<i>h</i>	<i>g</i>	<i>a</i>	1	0

- (a) Draw the corresponding state diagram.
 - (b)* Tabulate the reduced state table.
 - (c) Draw the state diagram corresponding to the reduced state table.
- 5.13** Starting from state *a*, and the input sequence 01110010011, determine the output sequence for
- (a) The state table of the previous problem.
 - (b) The reduced state table from the previous problem. Show that the same output sequence is obtained for both.
- 5.14** Substitute the one-hot assignment 2 from Table 5.9 to the states in Table 5.8 and obtain the binary state table.
- 5.15** List a state table for the *JK* flip-flop using *Q* as the present and next state and *J* and *K* as inputs. Design the sequential circuit specified by the state table and show that it is equivalent to Fig. 5.12(a).
- 5.16** Design a sequential circuit with two *D* flip-flops *A* and *B*, and one input x_{in} .
- (a)* When $x_{in} = 0$, the state of the circuit remains the same. When $x_{in} = 1$, the circuit goes through the state transitions from 00 to 01, to 11, to 10, back to 00, and repeats.
 - (b) When $x_{in} = 0$, the state of the circuit remains the same. When $x_{in} = 1$, the circuit goes through the state transitions from 00 to 11, to 01, to 10, back to 00, and repeats. (HDL—see Problem 5.38.)
- 5.17** Design a one-input, one-output serial 2's complemer. The circuit accepts a string of bits from the input and generates the 2's complement at the output. The circuit can be reset asynchronously to start and end the operation. (HDL—see Problem 5.39.)
- 5.18*** Design a sequential circuit with two *JK* flip-flops *A* and *B* and two inputs *E* and *F*. If $E = 0$, the circuit remains in the same state regardless of the value of *F*. When $E = 1$ and $F = 1$, the circuit goes through the state transitions from 00 to 01, to 10, to 11, back to 00, and repeats. When $E = 1$ and $F = 0$, the circuit goes through the state transitions from 00 to 11, to 10, to 01, back to 00, and repeats. (HDL—see Problem 5.40.)
- 5.19** A sequential circuit has three flip-flops *A*, *B*, *C*; one input x_{in} ; and one output y_{out} . The state diagram is shown in Fig. P5.19. The circuit is to be designed by treating the unused states as don't-care conditions. Analyze the circuit obtained from the design to determine the effect of the unused states. (HDL—see Problem 5.41.)

- (a)* Use D flip-flops in the design.
 (b) Use JK flip-flops in the design.

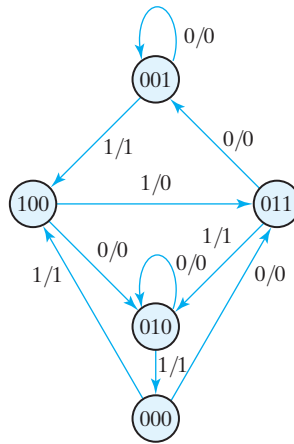


FIGURE P5.19

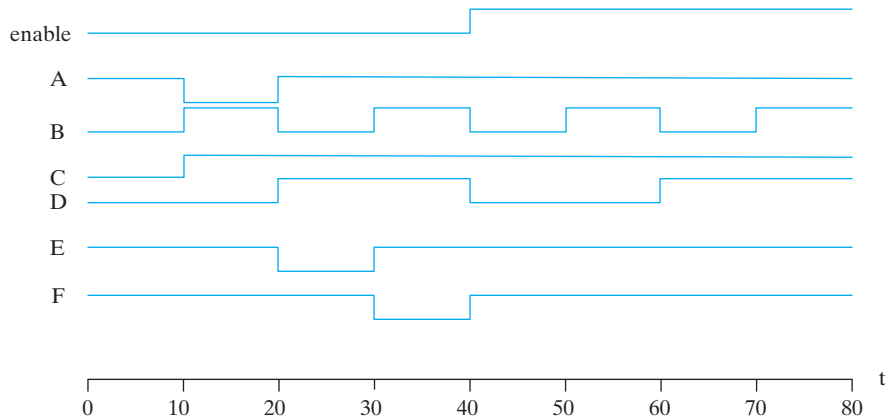
- 5.20** Design the sequential circuit specified by the state diagram of Fig. 5.19, using T flip-flops.
- 5.21** What is the main difference between an **initial** statement and an **always** statement in Verilog HDL?
- 5.22** Draw the waveform generated by the statements below:
- (a) **initial begin**
 $w = 0$; #10 $w = 1$; #40 $w = 0$; #20 $w = 1$; #15 $w = 0$;
end
- (b) **initial fork**
 $w = 0$; #10 $w = 1$; #40 $w = 0$; #20 $w = 1$; #15 $w = 0$;
join
- 5.23*** Consider the following statements assuming that *RegA* contains the value of 50 initially.
- (a) $\text{RegA} = 125$;
 $\text{RegB} = \text{RegA}$;
- (b) $\text{RegA} \leq 125$;
 $\text{RegB} \leq \text{RegA}$;
- What are the values of *RegA* and *RegB* after execution?
- 5.24** Write and verify an HDL behavioral description of a positive-edge-sensitive D flip-flop with asynchronous preset and clear.
- 5.25** A special positive-edge-triggered flip-flop circuit component has four inputs $D1$, $D2$, $D3$, and $D4$, and a two-bit control input that chooses between them. Write and verify an HDL behavioral description of this component.
- 5.26** Write and verify an HDL behavioral description of the JK flip-flop using an if-else statement based on the value of the present state.
- (a)* Obtain the characteristic equation when $Q = 0$ or $Q = 1$.
 (b) Specify how the J and K inputs affect the output of the flip-flop at each clock tick.
- 5.27** Rewrite and verify the description of HDL Example 5.5 by combining the state transitions and output into one **always** block.
- 5.28** Simulate the sequential circuit shown in Fig. 5.17.

- (a) Write the HDL description of the state diagram (i.e., behavioral model).
 - (b) Write the HDL description of the logic (circuit) diagram (i.e., a structural model).
 - (c) Write an HDL stimulus with a sequence of inputs: 00, 01, 11, 10. Verify that the response is the same for both descriptions.
- 5.29** Write a behavioral description of the state machine described by the state diagram shown in Fig. P5.19. Write a test bench and verify the functionality of the description.
- 5.30** Draw the logic diagram for the sequential circuit described by the following HDL module:
- ```

module Seq_Ckt (input A, B, C, E output reg Q, input CLK,);
reg E;

always @ (posedge CLK)
begin
 E <= A || B;
 Q <= E && C;
end
endmodule

```
- 5.31** How should the description in problem 5.30 be written to have the same behavior when the assignments are made with = instead of with <= ?
- 5.32** Using an **initial** statement with a **begin . . . end** block write a Verilog description of the waveforms shown in Fig. P5.32. Repeat using a **fork . . . join** block.

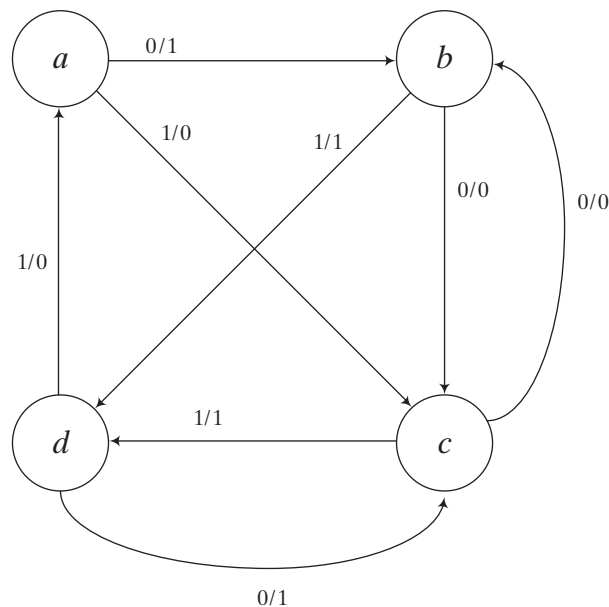


**FIGURE P5.32**  
Waveforms for Problem 5.32

- 5.33** Explain why it is important that the stimulus signals in a test bench be synchronized to the inactive edge of the clock of the sequential circuit that is to be tested.
- 5.34** Write and verify an HDL structural description of the machine having the circuit diagram (schematic) shown in Fig. 5.5.
- 5.35** Write and verify an HDL model of the sequential circuit described in Problem 5.6.
- 5.36** Write and verify an HDL structural description of the machine having the circuit diagram (schematic) shown in Fig. P5.8.
- 5.37** Write and verify HDL behavioral descriptions of the state machines shown in Figs. 5.25

and 5.26. Write a test bench to compare the state sequences and input–output behaviors of the two machines.

- 5.38** Write and verify an HDL behavioral description of the machine described in Problem 5.16.
- 5.39** Write and verify a behavioral description of the machine specified in Problem 5.17.
- 5.40** Write and verify a behavioral description of the machine specified in Problem 5.18.
- 5.41** Write and verify a behavioral description of the machine specified in Problem 5.19. (*Hint:* See the discussion of the **default** case item preceding HDL Example 4.8 in Chapter 4.)
- 5.42** Write and verify an HDL structural description of the circuit shown in Fig. 5.29.
- 5.43** Write and verify an HDL behavioral description of the three-bit binary counter in Fig. 5.34.
- 5.44** Write and verify a Verilog model of a *D* flip-flop having asynchronous reset.
- 5.45** Write and verify an HDL behavioral description of the sequence detector described in Fig. 5.27.
- 5.46** A synchronous finite state machine has an input  $x_{in}$  and an output  $y_{out}$ . When  $x_{in}$  changes from 0 to 1, the output  $y_{out}$  is to assert for three cycles, regardless of the value of  $x_{in}$ , and then de-assert for two cycles before the machine will respond to another assertion of  $x_{in}$ . The machine is to have active-low synchronous reset.
- Draw the state diagram of the machine.
  - Write and verify a Verilog model of the machine.
- 5.47** Write a Verilog model of a synchronous finite state machine whose output is the sequence 0, 2, 4, 6, 8, 10, 12, 14, 0 . . . . The machine is controlled by a single input, *Run*, so that counting occurs while *Run* is asserted, suspends while *Run* is de-asserted, and resumes the count when *Run* is re-asserted. Clearly state any assumptions that you make.
- 5.48** Write a Verilog model of the Mealy FSM described by the state diagram in Fig. P5.48. Develop a test bench and demonstrate that the machine state transitions and output correspond to its state diagram.



**FIGURE P5.48**

- 5.49** Write a Verilog model of the Moore FSM described by the state diagram in Fig. P5.49. Develop a test bench and demonstrate that the machine's state transitions and output correspond to its state diagram.

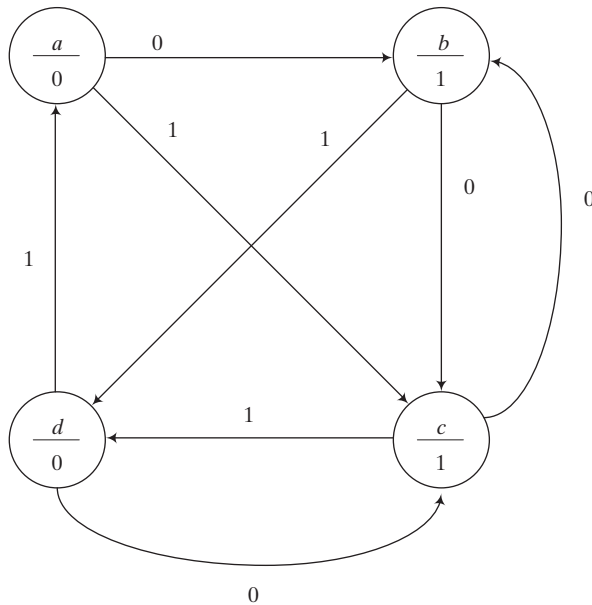


FIGURE P5.49

- 5.50** A synchronous Moore FSM has a single input,  $x_{in}$ , and a single output  $y_{out}$ . The machine is to monitor the input and remain in its initial state until a second sample of  $x_{in}$  is detected to be 1. Upon detecting the second assertion of  $x_{in}$   $y_{out}$  is to asserted and remain asserted until a fourth assertion of  $x_{in}$  is detected. When the fourth assertion of  $x_{in}$  is detected the machine is to return to its initial state and resume monitoring of  $x_{in}$ .

- Draw the state diagram of the machine.
- Write and verify a Verilog model of the machine.

- 5.51** Draw the state diagram of the machine described by the Verilog model given below.

```

module Prob_5_51 (output reg y_out, input x_in, clk, reset);
 parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
 reg [1:0] state, next_state;
 always @ (posedge clk, negedge reset) begin
 if (reset == 1'b0) state <= s0;
 else state <= next_state;
 always @(state, x_in) begin
 y_out = 0;
 next_state = s0;
 case (state)
 s0: if x_in = 1 begin y_out = 0; if (x_in) next_state = s1; else next_state = s0; end
 s1: if x_in = 1 begin y_out = 0; if (x_in) next_state = s2; else next_state = s1; end
 endcase
 end

```

```

s2: if x_in = 1 begin y_out = 1; if (x_in) next_state = s3; else next_state = s2; end
s3: if x_in = 1 begin y_out = 1; if (x_in) next_state = s0; else next_state = s3; end
default: next_state = s0;
endcase
end
endmodule

```

- 5.52** Draw the state diagram of the machine described by the Verilog model given below.

```

module Prob_5_52 (output reg y_out, input x_in, clk, reset);
 parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
 reg [1:0] state, next_state;
 always @ (posedge clk, negedge reset) begin
 if (reset == 1'b0) state <= s0;
 else state <= next_state;
 end
 always @(state, x_in) begin
 y_out = 0;
 next_state = s0;
 case (state)
 s0: if x_in = 1 begin y_out = 0; if (x_in) next_state = s1; else next_state = s0; end
 s1: if x_in = 1 begin y_out = 0; if (x_in) next_state = s2; else next_state = s1; end
 s2: if x_in = 1 if (x_in) begin next_state = s3; y_out = 0;
 else begin next_state = s2; y_out = 1; end
 s3: if x_in = 1 begin y_out = 1; if (x_in) next_state = s0; else next_state = s3; end
 default: next_state = s0;
 endcase
 end
end
endmodule

```

- 5.53** Draw a state diagram and write a Verilog model of a Mealy synchronous state machine having a single input,  $x_{in}$ , and a single output  $y_{out}$ , such that  $y_{out}$  is asserted if the total number of 1's received is a multiple of 3.
- 5.54** A synchronous Moore machine has two inputs,  $x_1$ , and  $x_2$ , and output  $y_{out}$ . If both inputs have the same value the output is asserted for one cycle; otherwise the output is 0. Develop a state diagram and a write a Verilog behavioral model of the machine. Demonstrate that the machine operates correctly.
- 5.55** Develop the state diagram for a Mealy state machine that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line.
- 5.56** Using manual methods, obtain the logic diagram of a three-bit counter that counts in the sequence 0, 2, 4, 6, 0, . . . .
- 5.57** Write and verify a Verilog behavioral model of a three-bit counter that counts in the sequence 0, 2, 4, 6, 0, . . . .
- 5.58** Write and verify a Verilog behavioral model of the counter designed in Problem 5.55.
- 5.59** Write and verify a Verilog structural model of the counter described in Problem 5.56.
- 5.60** Write and verify a Verilog behavioral model of a four-bit counter that counts in the sequence 0, 1, . . . , 9, 0, 1, 2, . . . .