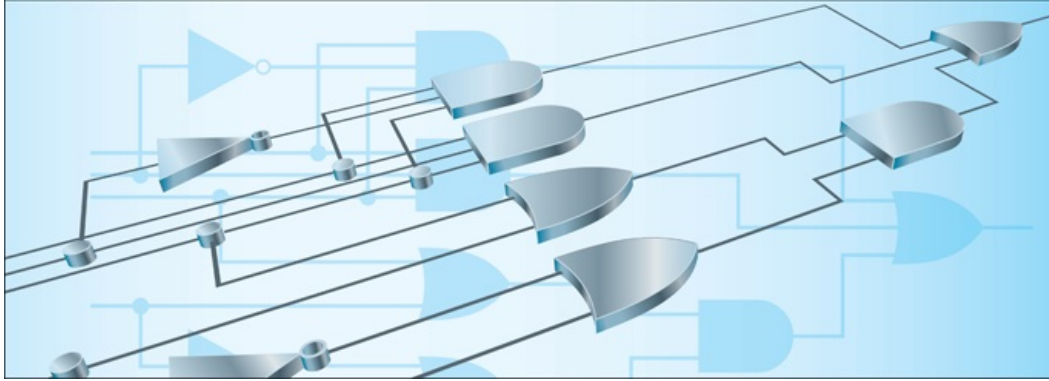


---

## Chapter 7 Logic Circuits

---

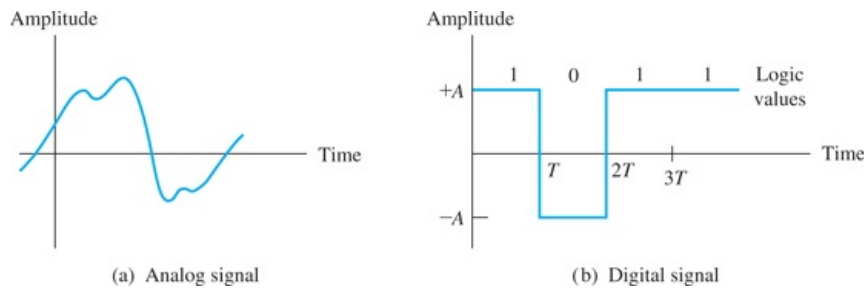


**Study of this chapter will enable you to:**

- State the advantages of digital technology over analog technology.
- Understand the terminology of digital circuits.
- Convert numbers between decimal, binary, and other forms.
- Use the Gray code for position and angular sensors.
- Understand the binary arithmetic operations used in computers and other digital systems.
- Interconnect logic gates of various types to implement a given logic function.
- Use Karnaugh maps to minimize the number of gates needed to implement a logic function.
- Understand how gates are connected together to form flip-flops and registers.

## Introduction to this chapter:

So far, we have considered circuits, such as filters, that process analog signals. For an **analog signal**, each amplitude in a continuous range has a unique significance. For example, a position sensor may produce an analog signal that is proportional to displacement. Each amplitude represents a different position. An analog signal is shown in [Figure 7.1\(a\)](#).




**Figure 7.1**

Analog signals take a continuum of amplitude values. Digital signals take a few discrete amplitudes.

In this chapter, we introduce circuits that process digital signals. For a **digital signal**, only a few restricted ranges of amplitude are allowed, and each amplitude in a given range has the same significance. Most common are **binary** signals that take on amplitudes in only two ranges, and the information associated with the ranges is represented by the **logic values** 1 or 0. An example of a digital signal is shown in [Figure 7.1\(b\)](#). Computers are examples of digital circuits. We will see that digital approaches have some important advantages over analog approaches.

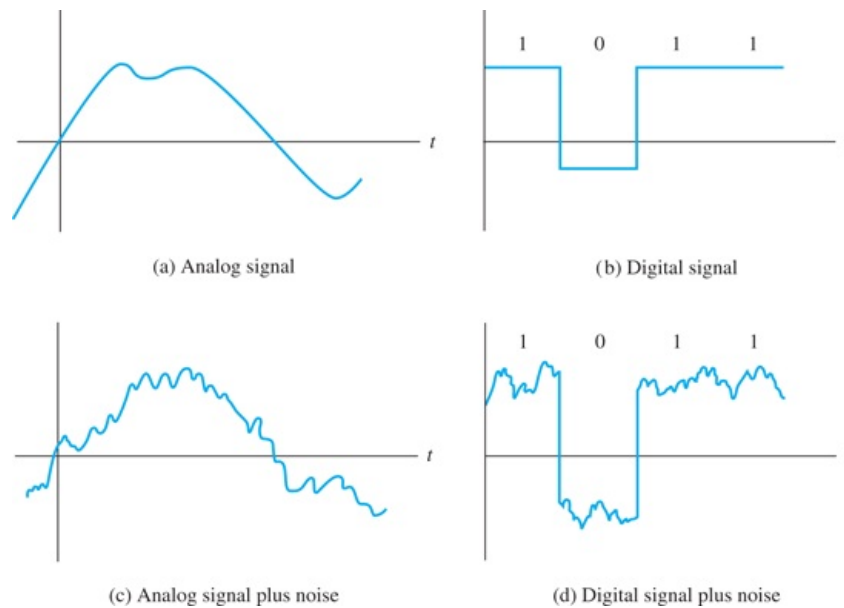
## 7.1 Basic Logic Circuit Concepts

We often encounter analog signals in instrumentation of physical systems. For example, a pressure transducer can yield a voltage that is proportional to pressure versus time in the cylinder of an internal combustion engine. In [Section 6.10](#) , we saw that analog signals can be converted to equivalent digital signals that contain virtually the same information. Then, computers or other digital circuits can be used to process this information. In many applications, we have a choice between digital and analog approaches.

## Advantages of the Digital Approach

Provided that the noise amplitude is not too large, the logic values represented by a digital signal can still be determined after noise is added.

Digital signals have several important advantages over analog signals. After noise is added to an analog signal, it is usually impossible to determine the precise amplitude of the original signal. On the other hand, after noise is added to a digital signal, we can still determine the logic values—provided that the noise amplitude is not too large. This is illustrated in [Figure 7.2](#).



**Figure 7.2**

The information (logic values) represented by a digital signal can still be determined precisely after noise is added. Noise obscures the information contained in an analog signal because the original amplitude cannot be determined exactly after noise is added.

For a given type of logic circuit, one range of voltages represents logic 1, and another range of voltages represents logic 0. For proper operation, a logic circuit only needs to produce a voltage somewhere in the correct range. Thus, component values in digital circuits do not need to be as precise as in analog circuits.

With modern IC technology, it is possible to manufacture exceedingly complex digital circuits economically.

It turns out that with modern integrated-circuit (IC) manufacturing technology, very complex digital logic circuits (containing millions of components) can be produced economically. Analog circuits often call for large capacitances and precise component values that are impossible to manufacture as large-scale ICs. Thus, digital systems have become increasingly important in the past few decades, a trend that will continue.

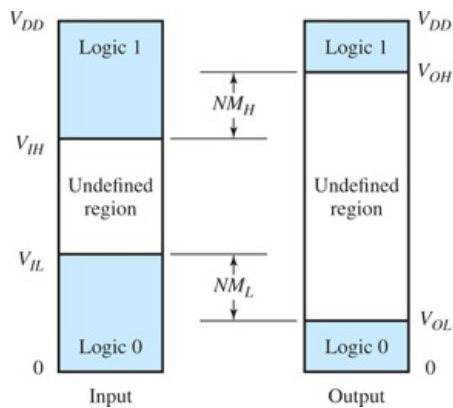
## Positive versus Negative Logic

Usually, the higher amplitude in a binary system represents 1 and the lower-amplitude range represents 0. In this case, we say that we have **positive logic**. On the other hand, it is possible to represent 1 by the lower amplitude and 0 by the higher amplitude, resulting in **negative logic**. Unless stated otherwise, we assume positive logic throughout this book.

The logic value 1 is also called **high**, **true**, or **on**. Logic 0 is also called **low**, **false**, or **off**. Signals in logic systems switch between high and low as the information being represented changes. We denote these signals, or **logic variables**, by uppercase letters such as  $A$ ,  $B$ , and  $C$ .

## Logic Ranges and Noise Margins

Logic circuits are typically designed so that a range of input voltages is accepted as logic 1 and another nonoverlapping range of voltages is accepted as logic 0. The input voltage accepted as logic 0, or low, is denoted as  $V_{IL}$ , and the smallest input voltage accepted as logic 1, or high, is denoted as  $V_{IH}$ . This is illustrated in [Figure 7.3](#). No meaning is assigned to voltages between  $V_{IL}$  and  $V_{IH}$ , which normally occur only during transitions.



**Figure 7.3**

Voltage ranges for logic-circuit inputs and outputs.

For a logic signal, one range of amplitudes represents logic 1, a nonoverlapping range represents logic 0, and no meaning is assigned to the remaining amplitudes, which ordinarily do not occur or occur only during transitions.

Furthermore, the circuits are designed so that the output voltages fall into narrower ranges than the inputs (provided that the inputs are in the acceptable ranges). This is also illustrated in [Figure 7.3](#).  $V_{OL}$  is the highest logic-0 output voltage, and  $V_{OH}$  is the lowest logic-1 output voltage.

Because noise can be added to a logic signal in the interconnections between outputs and inputs, it is important that the outputs have narrower ranges than the acceptable inputs. The differences are called **noise margins** and are given by

$$NM_L = V_{IL} - V_{OL}$$

$$NM_H = V_{OH} - V_{IH}$$

Ideally, noise margins are as large as possible.

## Digital Words

A single binary digit, called a **bit**, represents a very small amount of information. For example, a logic variable  $R$  could be used to represent whether or not it is raining in a particular location (say  $R = 1$  if it is raining, and  $R = 0$  if it is not raining).

To represent more information, we resort to using groups of logic variables called **digital words**. For example, the word  $RWS$  could be formed, in which  $R$  represents rain,  $W$  is 1 if the wind velocity is greater than 15 miles per hour and 0 for less wind, and  $S$  could be 1 for sunny conditions and 0 for cloudy. Then the digital word 110 would tell us that it is rainy, windy, and cloudy. A **byte** is a word consisting of eight bits, and a **nibble** is a four-bit word.

## Transmission of Digital Information

In **parallel transmission**, an  $n$ -bit word is transferred on  $n + 1$  wires, one wire for each bit, plus a common or ground wire. On the other hand, in **serial transmission**, the successive bits of the word are transferred one after the other with a single pair of wires. At the receiving end, the bits are collected and combined into words. Parallel transmission is faster and often used for short distances, such as internal data transfer in a computer. Long-distance digital communication systems are usually serial.

## Examples of Digital Information-Processing Systems

By using a prearranged 100-bit word consisting of logic values and binary numbers, we could give a rather precise report of weather conditions at a given location. Computers, such as those used by the National Weather Bureau, process words received from various weather stations to produce contour maps of temperature, wind velocity, cloud state, precipitation, and so on. These maps are useful in understanding and predicting weather patterns.

Analog signals can be reconstructed from their periodic samples (i.e., measurements of instantaneous amplitude at uniformly spaced points in time), provided that the sampling rate is high enough. Each amplitude value can be represented as a digital word. Thus, an analog signal can be represented by a sequence of digital words. In playback, the digital words are converted to the corresponding analog amplitudes. This is the principle of the compact-disc recording technique.

Thus, electronic circuits can gather, store, transmit, and process information in digital form to produce results that are useful or pleasing.

## 7.2 Representation of Numerical Data in Binary Form

### Binary Numbers

Because digital circuits are (almost always) designed to operate with only two symbols, 0 or 1, it is necessary to represent numerical and other data as words composed of 0s and 1s.

Digital words can represent numerical data. First, consider the decimal (base 10) number 743.2. We interpret this number as

$$7 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1}$$

Similarly, the binary or base-two number 1101.1 is interpreted as

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} = 13.5$$

Hence, the binary number 1101.1 is equivalent to the decimal number 13.5. Where confusion seems likely to occur, we use a subscript to distinguish binary numbers (such as  $1101.1_2$ ) from decimal numbers (such as  $13.5_{10}$ ).

With three bits, we can form  $2^3$  distinct words. These words can represent the decimal integers 0 through 7 as shown:

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

Similarly, a four-bit word has 16 combinations that represent the integers 0 through 15. (Frequently, we include leading zeros in discussing binary numbers in a digital circuit, because circuits are usually designed to operate on fixed-length words and the circuit produces the leading zeros.)

### Conversion of Decimal Numbers to Binary Form

To convert a decimal integer to binary, we repeatedly divide by two until the quotient is zero. Then, the remainders read in reverse order give the binary form of the number.

### Example 7.1 Converting a Decimal Integer to Binary

Convert the decimal integer  $343_{10}$  to binary.

Solution

The operations are shown in [Figure 7.4](#). The decimal number is repeatedly divided by two. When the quotient reaches zero, we stop. Then, the binary equivalent is read as the remainders in reverse order. From the figure, we see that

$$343_{10} = 101010111_2$$

	Quotient	Remainder	
$343/2$	$=$	171	1
$171/2$	$=$	85	1
$85/2$	$=$	42	1
$42/2$	$=$	21	0
$21/2$	$=$	10	1
$10/2$	$=$	5	0
$5/2$	$=$	2	1
$2/2$	$=$	1	0
$1/2$	$=$	0	1

$101010111_2$

Read binary equivalent in reverse order

Stop when quotient equals zero

**Figure 7.4**

Conversion of  $343_{10}$  to binary form.

To convert decimal fractions to binary fractions, we repeatedly multiply the fractional part by two and retain the whole parts of the results as the successive bits of the binary fraction.

### Example 7.2 Converting a Decimal Fraction to Binary

Convert  $0.392_{10}$  to its closest six-bit binary equivalent.

Solution

The conversion is illustrated in [Figure 7.5](#). The fractional part of the number is repeatedly multiplied by two. The whole part of each product is retained as a bit of the binary equivalent. We stop when the desired degree of precision has been reached. Thus, from the figure, we have

$$0.392_{10} > 0.011001_2$$

$2 \times 0.392$	$=$	0	+	0.784
$2 \times 0.784$	$=$	1	+	0.568
$2 \times 0.568$	$=$	1	+	0.136
$2 \times 0.136$	$=$	0	+	0.272
$2 \times 0.272$	$=$	0	+	0.544
$2 \times 0.544$	$=$	1	+	0.088

0  
1  
1  
0  
0  
1  
0.011001<sub>2</sub> (approximate binary equivalent)

**Figure 7.5**

Conversion of  $0.392_{10}$  to binary.

To convert a decimal number having both whole and fractional parts, we convert each part separately and then combine the parts.



### Example 7.3 Converting Decimal Values to Binary

Convert  $343.392_{10}$  to binary.

Convert the whole and fractional parts of the number separately and combine the results.

Solution

From [Examples 7.1](#) and [7.2](#), we have

$$343_{10} = 101010111_2$$

and

$$0.392_{10} \cong 0.011001_2$$

Combining these results, we get

$$343.392_{10} \cong 101010111.011001_2$$

### Example 7.4 Converting Binary Numbers to Decimal

Convert the binary number  $10011.011_2$  to decimal.

Solution

We have

$$\begin{aligned} 10011.011_2 &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} \\ &\quad + 1 \times 2^{-2} + 1 \times 2^{-3} = 19.375_{10} \end{aligned}$$

#### Exercise 7.1

Convert the following numbers to binary form, stopping after you have found six bits (if necessary) for the fractional part:

- a. 23.75;
- b. 17.25;
- c. 4.3.

#### Answer

- a. 10111.11;
- b. 10001.01;
- c. 100.010011.

### Exercise 7.2


Convert the following to decimal equivalents:

- a.  $1101.111_2$ ;
- b.  $100.001_2$ .

### Answer

- a.  $13.875_{10}$ ;
- b.  $4.125_{10}$ .

## Binary Arithmetic

We add binary numbers in much the same way that we add decimal numbers, except that the rules of addition are different (and much simpler). The rules for binary addition are shown in [Figure 7.6](#) .

		Sum	Carry
0 + 0	=	0	0
0 + 1	=	1	0
1 + 1	=	0	1
1 + 1 + 1	=	1	1

**Figure 7.6**

Rules of binary addition.

### **Example 7.5 Adding Binary Numbers**

Add the binary numbers  $1000.111$  and  $1100.011$ .

Solution

See [Figure 7.7](#) .


$$\begin{array}{r} 0001\ 11 \leftarrow \text{Carries} \\ 1000.111 \\ +1100.011 \\ \hline 10101.010 \end{array}$$

**Figure 7.7**

Addition of binary numbers.

## Hexadecimal and Octal Numbers

Binary numbers are inconvenient for humans because it takes many bits to write large numbers (or fractions to a high degree of precision). Hexadecimal (base 16) and octal (base 8) numbers are easily converted to and from binary numbers. Furthermore, they are much more efficient than binary numbers in representing information.

[Table 7.1](#)  shows the symbols used for hexadecimal and octal numbers and their binary equivalents. Notice that we need 16 symbols for the digits of a hexadecimal number. Customarily, the letters A through F are used to represent the digits for 10 through 15.

**Table 7.1. Symbols for Octal and Hexadecimal Numbers and Their Binary Equivalents**

Octal		Hexadecimal	
0	000	0	0000
1	001	1	0001
2	010	2	0010
3	011	3	0011
4	100	4	0100
5	101	5	0101
6	110	6	0110
7	111	7	0111
		8	1000
		9	1001
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

**Example 7.6 Converting Octal Numbers to Decimal**

Convert the octal number  $173.21_8$  to decimal.

Solution

We have

$$173.21_8 = 1 \times 8^2 + 7 \times 8^1 + 3 \times 8^0 + 2 \times 8^{-1} + 1 \times 8^{-2} = 123.265625_{10}$$

**Example 7.7 Converting Hexadecimal Numbers to Decimal**

Convert the hexadecimal number  $1FA.2A_{16}$  to decimal.

Solution

We have

$$\begin{aligned} 1FA.2A_{16} &= 1 \times 16^2 + 15 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 10 \times 16^{-2} \\ &= 506.1640625_{10} \end{aligned}$$

We can convert an octal or hexadecimal number to binary simply by substituting the binary equivalents for each digit.

### Example 7.8 Converting Octal and Hexadecimal Numbers to Binary

Convert the numbers  $317.2_8$  and  $F3A.2_{16}$  to binary.

In converting an octal or hexadecimal number to binary, use [Table 7.1](#) to replace each digit by its binary equivalent.

Solution

We simply use [Table 7.1](#) to replace each digit by its binary equivalent. Thus, we have

$$\begin{aligned} 317.2_8 &= 011\ 001\ 111.010_2 \\ &= 011001111.010_2 \end{aligned}$$

and

$$\begin{aligned} F3A.2_{16} &= 1111\ 0011\ 1010.0010 \\ &= 111100111010.0010_2 \end{aligned}$$

In converting binary numbers to octal, we first arrange the bits in groups of three, starting from the binary point and working outward. If necessary, we insert leading or trailing zeros to complete the groups. Then, we convert each group of three bits to its octal equivalent. Conversion to hexadecimal uses the same approach, except that the binary number is arranged in groups of four bits.

### Example 7.9 Converting Binary Numbers to Octal or Hexadecimal

Convert  $11110110.1_2$  to octal and to hexadecimal.

Working both directions from the binary point, group the bits into three-(octal) or four-(hexadecimal) bit words. Add leading or trailing zeros to complete the groups. Then, use [Table 7.1](#) to replace each binary word by the corresponding symbol.

Solution

For conversion to octal, we first form three-bit groups, working outward from the binary point:

$$11110110.1_2 = 011\ 110\ 110.100$$

Notice that we have appended leading and trailing zeros so that each group contains three bits. Next, we write the octal digit for each group. Thus, we have

$$11110110.1_2 = 011\ 110\ 110.100 = 366.4_8$$

For conversion to hexadecimal, we form four-bit groups appending leading and trailing zeros as needed. Then, we convert each group to its equivalent hexadecimal integer, yielding

$$11110110.1_2 = 1111\ 0110.1000 = F6.8_{16}$$

### Exercise 7.3

Convert the following numbers to binary, octal, and hexadecimal forms:

- a.  $97_{10}$ ;
- b.  $229_{10}$ .

#### Answer

- a.  $97_{10} = 1100001_2 = 141_8 = 61_{16}$ ;
- b.  $229_{10} = 11100101_2 = 345_8 = E5_{16}$ .

### Exercise 7.4

Convert the following numbers to binary form:

- a.  $72_8$ ;
- b.  $FA6_{16}$ .

#### Answer

- a.  $111010_2$ ;
- b.  $111110100110_2$ .

## Binary-Coded Decimal Format

In converting a decimal number to BCD, each digit is replaced by its four-bit equivalent.

Sometimes, decimal numbers are represented in binary form simply by writing the four-bit equivalents for each digit. The resulting numbers are said to be in **binary-coded decimal** (BCD) format. For example, 93.2 becomes

$$93.2 = 1001\ 0011.0010_{\text{BCD}}$$

Code groups 1010, 1011, 1100, 1101, 1110, and 1111 do not occur in BCD (unless an error has occurred). Calculators frequently represent numbers internally in BCD format. As each key is pressed, the BCD code group is stored. The operation

$$9 \times 3 = 27$$

would appear in BCD format as

$$1001 \times 0011 = 0010\ 0111$$

Even though binary code words are used to represent the decimal integers, the operations inside a calculator are partly decimal in nature. On the other hand, calculations are often carried out in true binary fashion in computers.

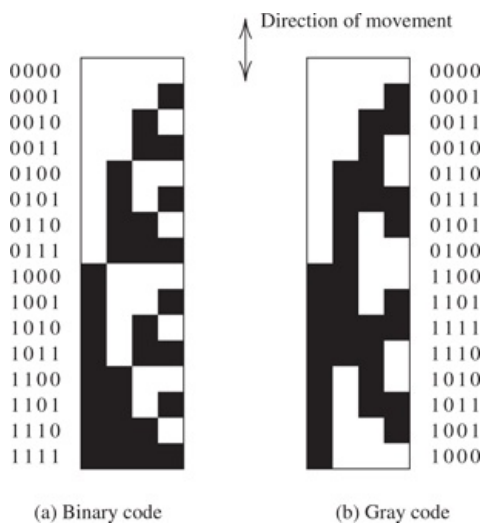
### Exercise 7.5

Express  $197_{10}$  in BCD form.

**Answer**  $197_{10} = 000110010111_{\text{BCD}}$ .

## Gray Code

Consider a transducer for encoding the position of a robot arm in which black-and-white bands are placed on the arm as illustrated in [Figure 7.8](#). In the figure, we assume that the bands are read by light-sensitive diodes in which a black band is converted to logic 1 and a white band is converted to logic 0.



**Figure 7.8**

Black and white bands that can be read by a photodiode array resulting in a digital word representing the position of a robot arm.

A problem occurs if the successive positions are represented by the binary code shown in [Figure 7.8\(a\)](#). For example, when the arm moves from the position represented by 0011 to that of 0100, three bits of the code word must change. Suppose that because the photodiode sensors are not perfectly aligned, 0011 first changes to 0001, then to 0000, and finally to 0100. During this transition, the indicated position is far from the actual position.

In a Gray code, each word differs in only one bit from each of its adjacent words.

A better scheme for coding the positions is to use the **Gray code** shown in [Figure 7.8\(b\)](#). In a Gray code, each code word differs in only one bit from its neighboring code words. Thus, erroneous position indications are avoided during transitions. Gray codes of any desired length can be constructed as shown in [Figure 7.9](#). (Notice that successive code words differ in a single bit.)

One-bit code	Two-bit code	Three-bit code
0	00	0   0   0
1	01	0   0   1
	11	0   1   1
	10	0   1   0
		1   1   0
		1   1   1
		1   0   1
		1   0   0

**Figure 7.9**

A one-bit Gray code simply consists of the two words 0 and 1. To create a Gray code of length  $n$ , repeat the code of length  $n - 1$  in reverse order, place a 0 on the left-hand side of each word in the first half of the list, and place a 1 on the left-hand side of each word in the second half of the list.

Gray codes are also used to encode the angular position of rotating shafts. Then, the last word in the list wraps around and is adjacent to the first word. To represent angular position with a resolution better than 1

degree, we would need a nine-bit Gray code consisting of  $2^9 = 512$  words. Each word would represent an angular sector of  $360/512 = 0.703$  degree in width.

#### Exercise 7.6

We wish to represent the position of a robot arm with a resolution of 0.01 inch or better. The range of motion is 20 inches. How many bits are required for a Gray code that can represent the arm position?

**Answer** 11 bits.

### Complement Arithmetic

The **one's complement** of a binary number is obtained by replacing 1s by 0s, and vice versa. For example, an eight-bit binary number and its one's complement are

$$\begin{array}{r} 01001101 \\ 10110010 \text{ ( one's complement )} \end{array}$$

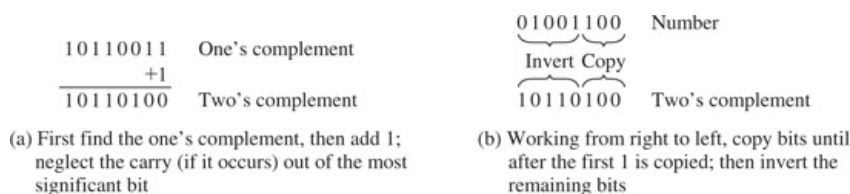
The **two's complement** of a binary number is obtained by adding 1 to the one's complement, neglecting the carry (if any) out of the most significant bit (MSB). For example, to find the two's complement of

$$01001100$$

we first form the one's complement, which is

$$10110011$$

and then add 1. This is illustrated in [Figure 7.10\(a\)](#).



**Figure 7.10**

Two ways to find the two's complement of the binary number 01001100.

Another way to obtain the two's complement of a number is to copy the number—working from right to left—until after the first 1 is copied. Then, the remaining bits are inverted. An example of this process is shown in [Figure 7.10\(b\)](#).

Complements are useful for representing negative numbers and performing subtraction in computers. Furthermore, the use of complement arithmetic simplifies the design of digital computers. Most common is the **signed two's-complement** representation, in which the first bit is taken as the sign bit. If the number is positive, the first bit is 0, whereas if the number is negative, the first bit is 1. Negative numbers are represented as the two's complement of the corresponding positive number. [Figure 7.11](#) shows the signed two's-complement representation using eight bits. In this case, the range of numbers that can be represented runs from  $-128$  to  $+127$ . Of course, if longer words are used, the range is extended.

+127	01111111
	...
+2	00000010
+1	00000001
0	00000000
-1	11111111
-2	11111110
	...
-128	10000000

↖ Sign bit

**Figure 7.11**

Signed two's-complement representation using eight-bit words.

Subtraction is performed by first finding the two's complement of the subtrahend and then adding in binary fashion and ignoring any carry out of the sign bit.

**Example 7.10 Subtraction Using Two's-Complement Arithmetic**

Perform the operation  $29_{10} - 27_{10}$  by using eight-bit signed two's-complement arithmetic.

Solution

First, we convert  $29_{10}$  and  $27_{10}$  to binary form. This yields

$$29_{10} = 00011101$$

and

$$27_{10} = 00011011$$

Next, we find the two's complement of the subtrahend:

$$-27_{10} = 11100101$$

Finally, we add the numbers to find the result:

$$\begin{array}{r}
 00011101 \\
 + 11100101 \\
 \hline
 \text{ignore carry out of sign bit} \rightarrow 00000010
 \end{array}
 \qquad
 \begin{array}{r}
 29 \\
 + (-27) \\
 \hline
 2
 \end{array}$$

Of course, performing addition and subtraction in this manner is tedious for humans. However, computers excel at performing simple operations rapidly and accurately.

In performing two's-complement arithmetic, we must be aware of the possibility of **overflow** in which the result exceeds the maximum value that can be represented by the word length in use. For example, if we use eight-bit words to add

$$97_{10} = 01100001$$

and

$$63_{10} = 00111111$$

we obtain

$$\begin{array}{r}
 01100001 \\
 + 00111111 \\
 \hline
 10100000
 \end{array}$$

The result is the signed two's-complement representation for  $-96$ , rather than the correct answer, which is  $97 + 63 = 160$ . This error occurs because the signed two's-complement representation has a maximum



value of  $+127$  (assuming eight-bit words).

Similarly, **underflow** occurs if the result of an arithmetic operation is less than  $-128$ . Overflow and underflow are not possible if the two numbers to be added have opposite signs. If the two numbers to be added have the same sign and the result has the opposite sign, underflow or overflow has occurred.

If the two numbers to be added have the same sign and the result has the opposite sign, overflow or underflow has occurred.

#### Exercise 7.7

Find the eight-bit signed two's-complement representation of

- a.  $22_{10}$  and
- b.  $-30_{10}$ .

#### Answer

- a. 00010110;
- b. 11100010.


#### Exercise 7.8

Carry out  $19_{10} - 4_{10}$  in eight-bit signed two's-complement form.

#### Answer

$$\begin{array}{rcl} 19 & & 00010011 \\ + \underline{(-4)} & + & \underline{11111100} \\ 15 & & 00001111 \end{array}$$

## 7.3 Combinatorial Logic Circuits

In this section, we consider circuits called **logic gates** that combine several logic-variable inputs to produce a logic-variable output. We focus on the external behavior of logic gates. Later, in [Chapter 11](#) , we will see how gate circuits can be implemented with field-effect transistors.

The circuits that we are about to discuss are said to be **memoryless** because their output values at a given instant depend only on the input values at that instant. Later, we consider logic circuits that are said to possess **memory**, because their present output values depend on previous, as well as present, input values.

## AND Gate

An important logic function is called the AND operation. The AND operation on two logic variables,  $A$  and  $B$ , is represented as  $AB$ , read as “ $A$  and  $B$ .” The AND operation is also called **logical multiplication**.

One way to specify a combinatorial logic system is to list all the possible combinations of the input variables and the corresponding output values. Such a listing is called a **truth table**. The truth table for the AND operation of two variables is shown in [Figure 7.12\(a\)](#). Notice that  $AB$  is 1 if and only if  $A$  and  $B$  are both 1.

$A$	$B$	$C = AB$
0	0	0
0	1	0
1	0	0
1	1	1

(a) Truth table



(b) Symbol for two-input AND gate

**Figure 7.12**

Two-input AND gate.

For the AND operation, we can write the following relations:

$$AA = A \quad (7.1)$$

$$A1 = A \quad (7.2)$$

$$A0 = 0 \quad (7.3)$$

$$AB = BA \quad (7.4)$$

$$A(BC) = (AB)C = ABC \quad (7.5)$$

The circuit symbol for a two-input AND gate (i.e., a circuit that produces an output equal to the AND operation of the inputs) is shown in [Figure 7.12\(b\)](#).

It is possible to have AND gates with more than two inputs. For example, the truth table and circuit symbol for a three-input AND gate are shown in [Figure 7.13](#).

$A$	$B$	$C$	$D = ABC$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a) Truth table



(b) Symbol for three-input AND gate

**Figure 7.13**

Three-input AND gate.

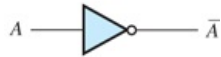
## Logic Inverter

The NOT operation on a logic variable is represented by placing a bar over the symbol for the logic variable. The symbol  $\bar{A}$  is read as “not  $A$ ” or as “ $A$  inverse.” If  $A$  is 0,  $\bar{A}$  is 1, and vice versa.

Circuits that perform the NOT operation are called **inverters**. The truth table and circuit symbol for an inverter are shown in [Figure 7.14](#). The *bubble* placed at the output of the inverter symbol is used to indicate inversion.

$A$	$\bar{A}$
0	1
1	0

(a) Truth table



(b) Symbol for an inverter

**Figure 7.14**

Logical inverter.

We can readily establish the following operations for the NOT operation:

$$\bar{A}A = 0 \quad (7.6)$$

$$\bar{\bar{A}} = A \quad (7.7)$$

## OR Gate

The OR operation of logic variables is written as  $A + B$ , which is read as “A or B.” The truth table and the circuit symbol for a two-input OR gate are shown in [Figure 7.15](#). Notice that  $A + B$  is 1 if A or B (or both) are 1. The OR operation is also called **logical addition**. The truth table and circuit symbol for a three-input OR gate are shown in [Figure 7.16](#). For the OR operation, we can write

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

(a) Truth table



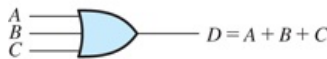
(b) Symbol for two-input OR gate

**Figure 7.15**

Two-input OR gate.

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

(a) Truth table



(b) Circuit symbol

**Figure 7.16**

Three-input OR gate.

$$(A + B) + C = A + (B + C) = A + B + C \quad (7.8)$$

$$A(B + C) = AB + AC \quad (7.9)$$

$$A + 0 = A \quad (7.10)$$

$$A + 1 = 1 \quad (7.11)$$

$$A + \bar{A} = 1 \quad (7.12)$$

$$A + A = A \quad (7.13)$$

## Boolean Algebra

Equation [Equation 7.13](#) illustrates that even though we use the addition sign ( + ) to represent the OR operation, manipulation of logic variables by the AND, OR, and NOT operations is different from ordinary algebra. The mathematical theory of logic variables is called **Boolean algebra**, named for mathematician George Boole.

One way to prove a Boolean algebra identity is to produce a truth table that lists all possible combinations of the variables and to show that both sides of the expression yield the same results.

### Example 7.11 Using a Truth Table to Prove a Boolean Expression

Prove the associative law for the OR operation ([Equation 7.8](#)), which states that

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

Solution

The truth table listing all possible combinations of the variables and the values of both sides of [Equation 7.8](#) is shown in [Table 7.2](#). We can see from the truth table that  $A + (B + C)$  and  $(A + B) + C$  take the same logic values for all combinations of  $A$ ,  $B$ , and  $C$ . Because both expressions yield the same results, the parentheses are not necessary, and we can write

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

**Table 7.2 Truth Table Used to Prove the Associative Law for the OR Operation ([Equation 7.8](#))**

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

### Exercise 7.9

Use truth tables to prove **Equations 7.5** and **7.9**.

**Answer** See [Tables 7.3](#) and [7.4](#).

**Table 7.3 Truth Table Used to Prove That  $A(BC) = (AB)C$  (Equation 7.5)**

$A$	$B$	$C$	$(AB)$	$(BC)$	$(AB)C$	$A(BC)$
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

**Table 7.4 Truth Table Used to Prove That  $A(B + C) = AB + AC$  (Equation 7.9)**

$A$	$B$	$C$	$(B + C)$	$AB$	$AC$	$AB + AC$	$A(B + C)$	
0	0	0	0	0	0	0	0	
0	0	1	1	0	0	0	0	
0	1	0	1	0	0	0	0	
0	1	1	1	0	0	0	0	
1	0	0	0	0	0	0	0	
1	0	1	1	0	1	1	1	
1	1	0	1	1	0	1	1	
1	1	1	1	1	1	1	1	

### Exercise 7.10

Prepare a truth table for the logic expression  $D = AB + C$ .

**Answer** See [Table 7.5](#) .

**Table 7.5** Truth Table for  $D = AB + C$

<i>A</i>	<i>B</i>	<i>C</i>	<i>AB</i>	$D = AB + C$	
0	0	0	0	0	
0	0	1	0	1	
0	1	0	0	0	
0	1	1	0	1	
1	0	0	0	0	
1	0	1	0	1	
1	1	0	1	1	
1	1	1	1	1	

### Implementation of Boolean Expressions

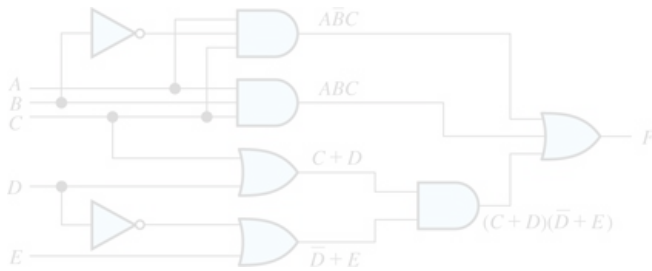
Boolean algebra expressions can be implemented by interconnection of AND gates, OR gates, and inverters.

Boolean algebra expressions can be implemented by interconnection of AND gates, OR gates, and inverters. For example, the logic expression

$$F = \bar{A}BC + ABC + (C + D)(\bar{D} + E) \quad (7.14)$$

can be implemented by the logic circuit shown in [Figure 7.17](#) .





**Figure 7.17**

A circuit that implements the logic expression

$$F = \bar{A}BC + ABC + (C + D)(\bar{D} + E).$$

Sometimes, we can manipulate a logic expression to find an equivalent expression that is simpler. For example, the last term on the right-hand side of [Equation 7.14](#) can be expanded, resulting in

$$F = \bar{A}BC + ABC + \bar{C}\bar{D} + CE + \bar{D}\bar{D} + DE \quad (7.15)$$

But the term  $\bar{D}\bar{D}$  always has the logic value 0, so it can be dropped from the expression. Factoring the first two terms on the right-hand side of [Equation 7.15](#) results in

$$F = AC(\bar{B} + B) + \bar{C}\bar{D} + CE + DE \quad (7.16)$$

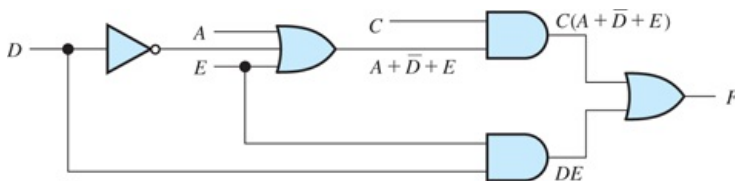
However, the quantity  $\bar{B} + B$  always equals 1, so we can write

$$F = AC + \bar{C}\bar{D} + CE + DE \quad (7.17)$$

Factoring C from the first three terms on the right-hand side, we have

$$F = C(A + \bar{D} + E) + DE \quad (7.18)$$

This can be implemented as shown in [Figure 7.18](#).



**Figure 7.18**

A simpler circuit equivalent to that of [Figure 7.17](#).

We can often find alternative implementations for a given logic function.

Thus, we can often find alternative implementations for a given. Later, we consider methods for finding the implementation using the fewest gates of a given type.

## De Morgan's Laws

Two important results in Boolean algebra are De Morgan's laws, which are given by

$$\overline{AB} = \bar{A} + \bar{B} \quad (7.19)$$

and

$$A + B = \overline{\overline{A} \overline{B}} \quad (7.20)$$

De Morgan's laws can be extended to three variables as follows:

$$\overline{ABC} = \overline{A} + \overline{B} + \overline{C} \quad \text{and} \quad \overline{A + B + C} = \overline{A} \overline{B} \overline{C}$$

Another way to state these laws is as follows: If the variables in a logic expression are replaced by their inverses, the AND operation is replaced by OR, the OR operation is replaced by AND, and the entire expression is inverted, the resulting logic expression yields the same values as before the changes.

Thus, De Morgan's laws can be used to change any (or all) of the AND operations in a logic expression to OR operations and vice versa. By changing various combinations of the operations, a variety of equivalent expressions can be found.

### Example 7.12 Applying De Morgan's Laws

Apply De Morgan's laws, changing all of the ORs to ANDs and all of the ANDs to ORs, to the right-hand side of the logic expression:

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

Solution

First, we replace each variable by its inverse, resulting in the expression

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

Then, we replace the AND operation by OR, and vice versa:

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

Finally, inverting the expression, we can write

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

Therefore, De Morgan's laws give us alternative ways to write logic expressions.

### Exercise 7.11

Use De Morgan's laws to find alternative expressions for

$$D = AB + \overline{B}C$$

and

$$E = [F(\overline{G} + \overline{H}) + F\overline{G}]$$

**Answer**

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

$$E = \overline{(F + \overline{G}H)(F + G)}$$

Any combinatorial logic function can be implemented solely with AND gates and inverters.

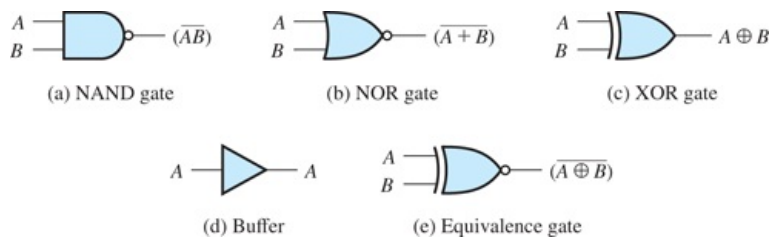
An important implication of De Morgan's laws is that *we can implement any logic function by using AND gates and inverters*. This is true because [Equation 7.20](#) can be employed to replace the OR operation by the AND operation (and logical inversions).

Any combinatorial logic function can be implemented solely with OR gates and inverters.

Similarly, *any logic function can be implemented with OR gates and inverters*, because [Equation 7.19](#) can be used to replace the AND operation with the OR operation (and logical inversions). Consequently, to implement a logic function, we need inverters and either AND gates or OR gates, not both.

### NAND, NOR, and XOR Gates

Some additional logic gates are shown in [Figure 7.19](#). The NAND gate is equivalent to an AND gate followed by an inverter. Notice that the symbol is the same as for an AND gate, with a bubble at the output terminal to indicate that the output has been inverted after the AND operation. Similarly, the NOR gate is equivalent to an OR gate followed by an inverter.



**Figure 7.19**

Additional logic-gate symbols.

The exclusive-OR (XOR) operation for two logic variables  $A$  and  $B$  is represented by  $A \oplus B$  and is defined by

$$0 \oplus 0 = 0$$

$$1 \oplus 0 = 1$$

$$0 \oplus 1 = 1$$

$$1 \oplus 1 = 0$$

Notice that the XOR operation yields 1 if  $A$  is 1 or if  $B$  is 1, but yields 0 if both  $A$  and  $B$  are 1. The XOR operation is also known as **modulo-two addition**.

A **buffer** has a single input and produces an output with the same value as the input. (Buffers are commonly used to provide large currents when a logic signal must be applied to a low-impedance load.)

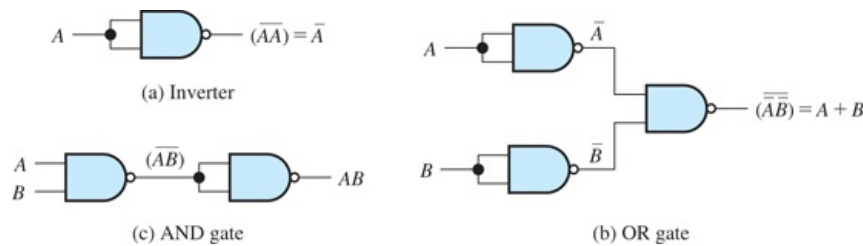
The **equivalence gate** produces a high output only if both inputs have the same value. In effect, it is an XOR followed by an inverter as the symbol of [Figure 7.19\(e\)](#) implies.

## Logical Sufficiency of NAND Gates or of NOR Gates

As we have seen, several combinations of gates often can be found that perform the same function. For example, if the inputs to a NAND are tied together, an inverter results. This is true because

$$\overline{(\overline{A}A)} = A$$

which is illustrated in [Figure 7.20\(a\)](#).



**Figure 7.20**

Basic Boolean operations can be implemented with NAND gates. Therefore, any Boolean function can be implemented by the use of NAND gates alone.

Any combinatorial logic function can be implemented solely with NAND gates.

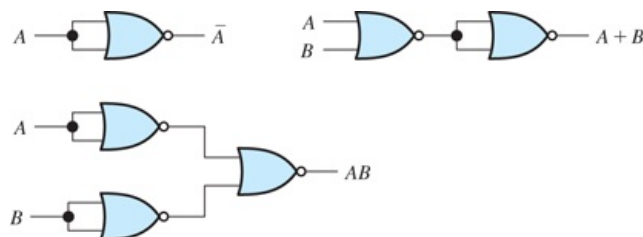
Furthermore, as shown by De Morgan's laws, the OR operation can be realized by inverting the input variables and combining the results in a NAND gate. This is shown in [Figure 7.20\(b\)](#), in which the inverters are formed from NAND gates. Finally, a NAND followed by an inverter results in an AND gate. *Since the basic logic functions (AND, OR, and NOT) can be realized by using only NAND gates, we conclude that NAND gates are sufficient to realize any combinatorial logic function.*

Any combinatorial logic function can be implemented solely with NOR gates.

### Exercise 7.12

Show how to use only NOR gates to realize the AND, OR, and NOT functions.

**Answer** See [Figure 7.21](#).



**Figure 7.21**

The AND, OR, and NOT operations can be implemented with NOR gates. Thus, any combinatorial logic circuit can be designed by using only NOR gates. See [Exercise 7.12](#).

## 7.4 Synthesis of Logic Circuits

In this section, we consider methods to implement logic circuits given the specification for the output in terms of the inputs. Often, the initial specification for a logic circuit is given in natural language. This is translated into a truth table or a Boolean logic expression that can be manipulated to find a practical implementation.

### Sum-of-Products Implementation

Consider the truth table shown in [Table 7.6](#).  $A$ ,  $B$ , and  $C$  are input logic variables, and  $D$  is the desired output. Notice that we have numbered the rows of the truth table with the decimal number corresponding to the binary number formed by  $ABC$ .

**Table 7.6 Truth Table Used to Illustrate SOP and POS Logical Expressions**

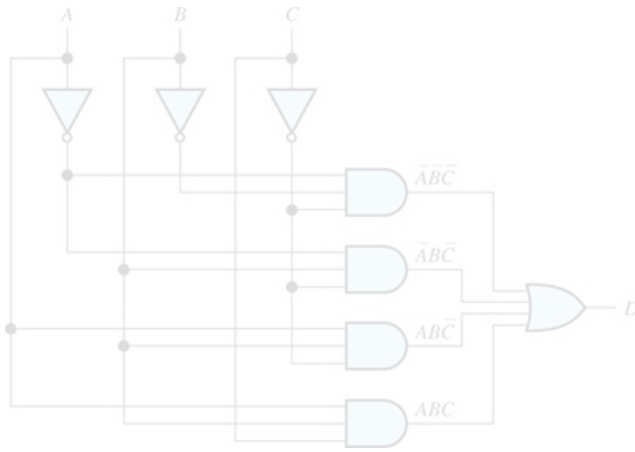
Row	$A$	$B$	$C$	$D$	
0	0	0	0	1	
1	0	0	1	0	
2	0	1	0	1	
3	0	1	1	0	
4	1	0	0	0	
5	1	0	1	0	
6	1	1	0	1	
7	1	1	1	1	

Suppose that we want to find a logic circuit that produces the output variable  $D$ . One way to write a logic expression for  $D$  is to concentrate on the rows of the truth table for which  $D$  is 1. In [Table 7.6](#), these are the rows numbered 0, 2, 6, and 7. Then, we write a logical product of the input logic variables or their inverses that equals 1 for each of these rows. Each input variable or its inverse is included in each product. In writing the product for each row, we invert the logic variables that are 0 in that row. For example, the logical product  $\bar{A}\bar{B}\bar{C}$  equals logic 1 only for row 0. Similarly,  $\bar{A}BC$  equals logic 1 only for row 2,  $AB\bar{C}$  equals logic 1 only for row 6, and  $ABC$  equals 1 only for row 7. Product terms that include all of the input variables (or their inverses) are called **minterms**.

Finally, we write an expression for the output variable as a logical sum of minterms. For [Table 7.6](#), it yields

$$D = \bar{A}\bar{B}\bar{C} + \bar{A}BC + AB\bar{C} + ABC \quad (7.21)$$

This type of expression is called a **sum of products** (SOP). Following this procedure, we can always find an SOP expression for a logic output given the truth table. A logic circuit that implements [Equation 7.21](#) directly is shown in [Figure 7.22](#).



**Figure 7.22**  
Sum-of-products logic circuit for [Table 7.6](#).

In a sum-of-products expression, we form a product of all the input variables (or their inverses) for each row of the truth table for which the result is logic 1. The output is the sum of these products.

A shorthand way to write an SOP is simply to list the row numbers of the truth table for which the output is logic 1. Thus, we can write

$$D = \sum m(0, 2, 6, 7) \quad (7.22)$$

in which  $m$  indicates that we are summing the minterms corresponding to the rows enumerated.

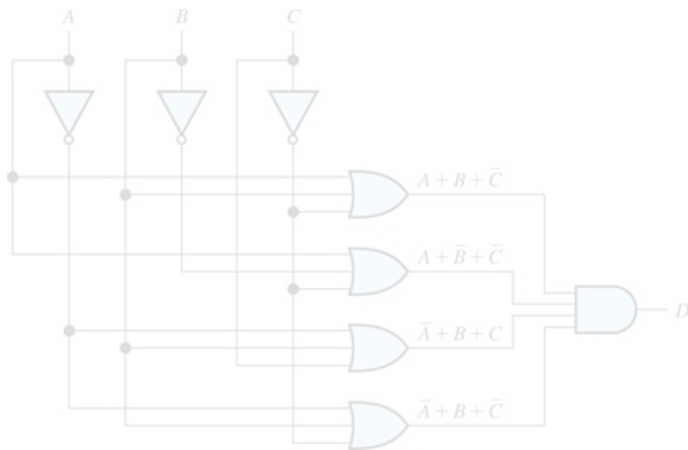
### Product-of-Sums Implementation

Another way to write a logic expression for  $D$  is to concentrate on the rows of the truth table for which  $D$  is 0. For example, in [Table 7.6](#), these are the rows numbered 1, 3, 4, and 5. Then, we write a logical sum that equals 0 for each of these rows. Each input variable or its inverse is included in each sum. In writing the sum for each row, we invert the logic variables that are 1 in that row. For example, the logical sum  $(A + B + \bar{C})$  equals logic 0 only for row 1. Similarly,  $(A + \bar{B} + \bar{C})$  equals logic 0 only for row 3,  $(\bar{A} + B + C)$  equals logic 0 only for row 4, and  $(\bar{A} + \bar{B} + C)$  equals 0 only for row 5. Sum terms that include all of the input variables (or their inverses) are called **maxterms**.

Finally, we write an expression for the output variable as the logical product of maxterms. For [Table 7.6](#), it yields

$$D = (A + B + \bar{C}) (A + \bar{B} + \bar{C}) (\bar{A} + B + C) (\bar{A} + \bar{B} + C) \quad (7.23)$$

This type of expression is called a **product of sums** (POS). We can always find a POS expression for a logic output given the truth table. A circuit that implements [Equation 7.23](#) is shown in [Figure 7.23](#).



**Figure 7.23**

Product-of-sums logic circuit for [Table 7.6](#).

In a product-of-sums expression, we form a sum of all the input variables (or their inverses) for each row of the truth table for which the result is logic 0. The output is the product of these sums.

A shorthand way to write a POS is simply to list the row numbers of the truth table for which the output is logic 1. Thus, we write

$$D = \prod M(1, 3, 4, 5) \quad (7.24)$$

in which  $M$  indicates the maxterms corresponding to the rows enumerated.

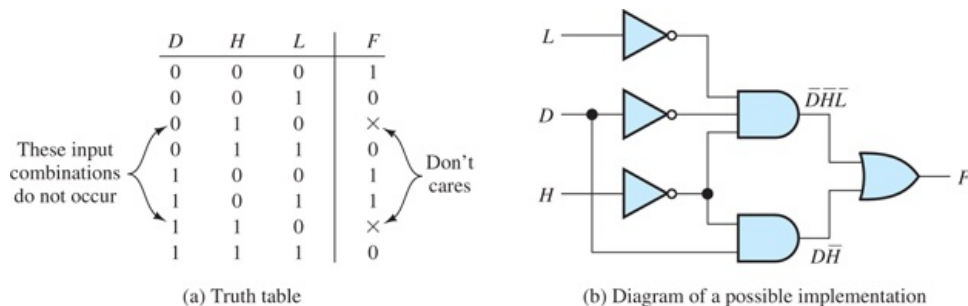
### Example 7.13 Combinatorial Logic Circuit Design

The control logic for a residential heating system is to operate as follows: During the daytime, heating is required only if the temperature falls below 68°F. At night, heating is required only for temperatures below 62°F. Assume that logic signals  $D$ ,  $L$ , and  $H$  are available.  $D$  is high during the daytime and low at night.  $H$  is high only if the temperature is above 68°F.  $L$  is high only if the temperature is above 62°F. Design a logic circuit that produces an output signal  $F$  that is high only when heating is required.

**Solution**

First, we translate the description of the desired operation into a truth table. This is shown in [Figure 7.24\(a\)](#). We have listed all combinations of the inputs. However, two combinations do not occur because temperature cannot be below 62°F ( $L = 0$ ) and also be above 68°F ( $H = 1$ ). The output listed for these combinations is  $\times$ , which is called a **don't care** because we don't care what the output of the logic circuit is for these input combinations.

Some combinations of input variables may not occur; if so, the corresponding outputs are called “don't cares.”



**Figure 7.24**

See [Example 7.10](#).

As we have seen, one way to translate the truth table into a logic expression is to write the SOP in which there is a separate term for each high output. Applying this approach to [Figure 7.24\(a\)](#) yields

$$F = \bar{D}\bar{H}\bar{L} + D\bar{H}\bar{L} + DHL \quad (7.25)$$

The first term on the right-hand side  $\bar{D}\bar{H}\bar{L}$  is high only for row 0 of the truth table. Also, the second term  $D\bar{H}\bar{L}$  is high only for row 4, and the third term  $DHL$  is high only for row 5. Thus, the shorthand way to write the logic expression is

$$F = \sum m(0, 4, 5) \quad (7.26)$$

Notice that in Equations 7.25 and 7.26, the don't cares turn out to be low.

The logic expression of [Equation 7.25](#) can be manipulated into the form

$$F = D\bar{H} + \bar{D}\bar{H}\bar{L}$$

A logic diagram for this is shown in [Figure 7.24\(b\)](#).

An alternative approach is to write a POS with a separate sum term for each row of the truth table having an output value of 0. For the truth table of [Figure 7.24\(a\)](#), we have

$$F = (D + H + \bar{L})(\bar{D} + \bar{H} + \bar{L})(\bar{D} + \bar{H} + \bar{L}) \quad (7.27)$$

The first term in the product  $(D + H + \bar{L})$  is low only for row 1 of the truth table. (Recall that row 1 is actually the second row because we start numbering with 0.) The second term  $(\bar{D} + \bar{H} + \bar{L})$  is low only for row 3, and the last term in the product is low only for the last row of the truth table.

In short form, we can write [Equation 7.27](#) as

$$F = \prod M(1, 3, 7) \quad (7.28)$$

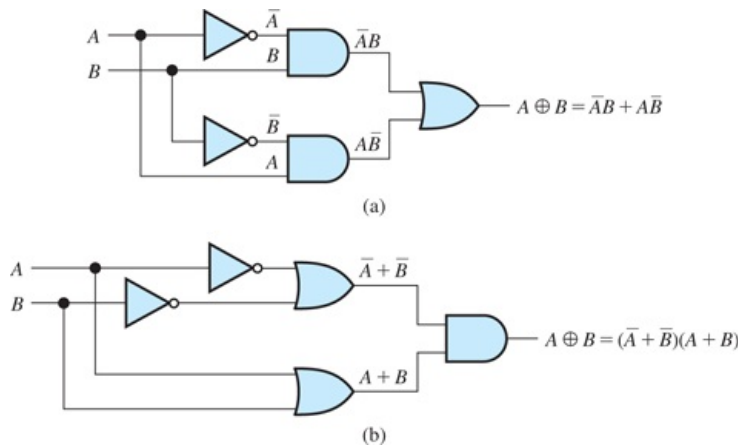
For [Equations 7.27](#) and [7.28](#), the don't cares are high. Because of the different outputs for the don't cares, the expressions we have given for  $F$  in [Equations 7.25](#) and [7.27](#) are not equivalent.



### Exercise 7.13

Show two ways to realize the XOR operation by using AND, OR, and NOT gates.

**Answer** See [Figure 7.25](#).



**Figure 7.25**

Answer for [Exercise 7.13](#).

### Exercise 7.14

A traditional children's riddle concerns a farmer who is traveling with a sack of rye, a goose, and a mischievous dog. The farmer comes to a river that he must cross from east to west. A boat is available, but it only has room for the farmer and one of his possessions. If the farmer is not present, the goose will eat the rye or the dog will eat the goose.

We wish to design a circuit to emulate the conditions of this riddle. A separate switch is provided for the farmer, the rye, the goose, and the dog. Each switch has two positions depending on whether the corresponding object is on the east bank or the west bank of the river. The rules of play stipulate that no more than two switches be moved at a time and that the farmer must move (to row the boat) each time switches are moved. The switch for the farmer provides logic signal  $F$ , which is high if the farmer is on the east bank and low if he is on the west bank. Similar logic signals ( $G$  for the goose,  $D$  for the dog, and  $R$  for the rye) are high if the corresponding object is on the east bank and low if it is on the west bank.

Find a Boolean logic expression based on the sum-of-products approach for a logic signal  $A$  (alarm) that is high anytime the rye or the goose is in danger of being eaten. Repeat for the product-of-sums approach.

**Answer** The truth table is shown in [Table 7.7](#). The Boolean expressions are

$$A = \sum m(3, 6, 7, 8, 9, 12) = FDGR + FDGR + FDGR + FDGR + FDGR + FDGR$$

and



$$A = \prod M(0, 1, 2, 4, 5, 10, 11, 13, 14, 15)$$

Table 7.7 Truth Table for [Exercise 7.14](#) 

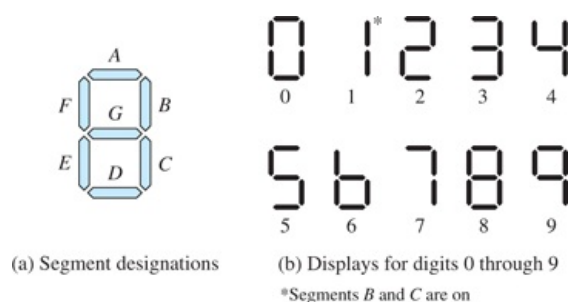
<i>F</i>	<i>D</i>	<i>G</i>	<i>R</i>	<i>A</i>	
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	1	
0	1	1	1	1	
1	0	0	0	1	
1	0	0	1	1	
1	0	1	0	0	
1	0	1	1	0	
1	1	0	0	1	
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	0	

## Decoders, Encoders, and Translators

Many useful combinatorial circuits known as **decoders**, **encoders**, or **translators** are available as ICs. We discuss two examples. In a calculator or watch, we may represent information to be displayed in BCD form. Thus, 0000 is for 0, 0001 is for 1, 0010 is for 2, 0011 is for 3, and so on. Using four-bit words, 16 combinations are possible. However, only 10 combinations are used in BCD. Codes such as 1010 and 1011 do not occur in BCD.

The calculator display typically consists of liquid crystals with seven segments, as illustrated in [Figure 7.26\(a\)](#) . The digits 0 through 9 are displayed by turning on appropriate segments as shown in [Figure 7.26\(b\)](#) . Thus, a decoder is needed to translate the four-bit BCD words into seven-bit words of the form

$ABCDEFG$ , for which  $A$  is high if segment  $A$  of the display is required to be on,  $B$  is high if segment  $B$  is required to be on, and so on. Thus, 0000 is translated to 1111110 because all segments except  $G$  are on to display the symbol for zero. Similarly, 0001 becomes 0110000, and 0010 becomes 1101101. Hence, the BCD-to-seven-segment decoder is a combinatorial circuit having four inputs and seven outputs.

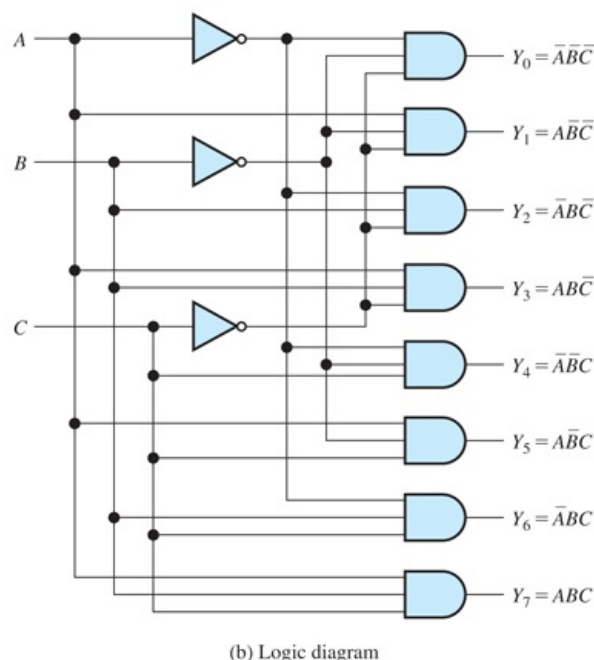


**Figure 7.26**  
Seven-segment display.

Another example is the three-to-eight-line decoder that has a three-bit input and eight output lines. The three-bit input word selects one of the output lines and that output becomes high. The truth table and a circuit implementation are shown in [Figure 7.27](#).

$C$	$B$	$A$	$Y_0$	$Y_1$	$Y_2$	$Y_3$	$Y_4$	$Y_5$	$Y_6$	$Y_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

(a) Truth table



**Figure 7.27**  
Three-line-to-eight-line decoder.

Decoders are available that are used to convert binary numbers to BCD, or vice versa, test whether one number is larger or smaller than another, perform arithmetic operations on binary or BCD numbers, and many similar functions.



## 7.5 Minimization of Logic Circuits

Implementations based on either a sum of minterms or a product of maxterms may not be optimum in minimizing the number of gates needed to realize a logic function.

We have seen that logic functions can be readily expressed either as a logical sum of minterms or as a logical product of maxterms. However, direct implementation of either of these expressions may not yield the best circuit in terms of minimizing the number of gates required. For example, consider the logical expression

$$F = \bar{A}\bar{B}D + \bar{A}BD + BCD + ABC \quad (7.29)$$

Implemented directly, this expression would require two inverters, four AND gates, and one OR gate.

Factoring the first pair of terms, we have

$$F = \bar{A}D(\bar{B} + B) + BCD + ABC$$

However,  $\bar{B} + B = 1$ , so we obtain

$$F = \bar{A}D + BCD + ABC$$

Of course,  $BCD = 1$  only if  $B = 1$ ,  $C = 1$ , and  $D = 1$ . In that case, either  $\bar{A}D = 1$  or  $ABC = 1$ , because we must have either  $\bar{A} = 1$  or  $A = 1$ . Thus, the term  $BCD$  is redundant and can be dropped from the expression. Then, we get

$$F = \bar{A}D + ABC \quad (7.30)$$

Only one inverter, two AND gates, and one OR gate are required to implement this expression.

### Exercise 7.15

Create a truth table to verify that the right-hand sides of [Equations 7.29](#) and [7.30](#) yield the same result.

**Answer** See [Table 7.8](#).

**Table 7.8** Answer for [Exercise 7.15](#)

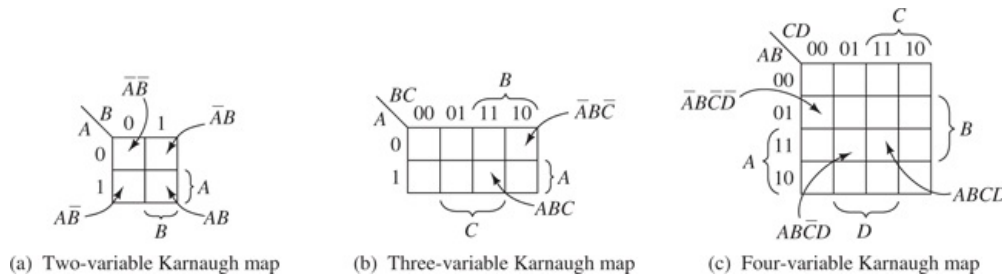
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>F</i>	
0	0	0	0	0	
0	0	0	1	1	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	1	1	
0	1	1	0	0	
0	1	1	1	1	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	0	
1	0	1	1	0	
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	1	
1	1	1	1	1	

### Karnaugh Maps

As we have demonstrated, logic expressions can sometimes be simplified dramatically. However, the algebraic manipulations needed to simplify a given expression are often not readily apparent. By using a

graphical approach known as the **Karnaugh map**, we will find it much easier to minimize the number of terms in a logic expression.

A Karnaugh map is an array of squares. Each square corresponds to one of the minterms of the logic variables or, equivalently, to one of the rows of the truth table. Karnaugh maps for two, three, and four variables are shown in [Figure 7.28](#). The two-variable map consists of four squares, one corresponding to each of the minterms. Similarly, the three-variable map has eight squares, and the four-variable map has 16 squares.



**Figure 7.28**

Karnaugh maps showing the minterms corresponding to some of the squares.

A Karnaugh map is a rectangular array of squares, where each square represents one of the minterms of the logic variables

The minterms corresponding to some of the squares are shown in [Figure 7.28](#). For example, for the three-variable map, the minterm  $A'B'C$  corresponds to the upper right-hand square. Also, the bit combinations corresponding to the rows of the truth table are shown down the left-hand side and across the top of the map. For example, on the four-variable map, the row of the truth table for which the four-bit word  $ABCD$  is 1101 corresponds to the square in the third row (i.e., the row labeled 11) and second column (i.e., the column labeled 01). Thus, we can readily find the square corresponding to any minterm or to any row of the truth table.

In case you are wondering about the order of the bit patterns along the side or top of the four-variable Karnaugh map (i.e., 00 01 11 10), notice that this is a two-bit Gray code. Thus, the patterns for squares with a common side differ in only one bit, so that similar minterms are grouped together. For example, the minterms containing  $A$  (rather than  $\bar{A}$ ) fall in the bottom half of each map. In the four-variable map, the minterms containing  $B$  are in the middle two rows, the minterms containing  $AB$  are in the third row, and so forth. This grouping of similar terms is the key to simplifying logic circuits.

#### Exercise 7.16

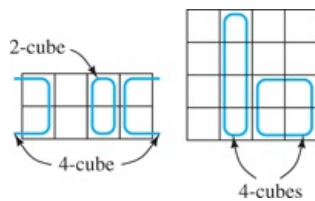
- Write the minterm corresponding to the upper right-hand square in [Figure 7.28\(c\)](#).
- Write the minterm corresponding to the lower left-hand square.

#### Answer

- $\bar{A} \bar{B} C D$ ;
- $A B \bar{C} \bar{D}$ .

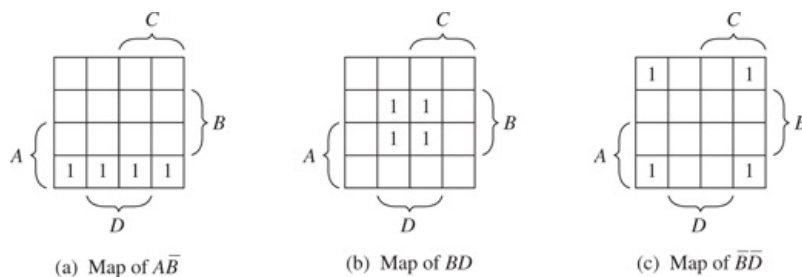
The left and right (as well as top and bottom) edges of the Karnaugh map are considered to be adjacent.

We call two squares that have a common edge a **2-cube**. Similarly, four squares with common edges are called a **4-cube**. In locating cubes, the maps should be considered to fold around from top to bottom and from left to right. Therefore, the squares on the right-hand side are considered to be adjacent to those on the left-hand side, and the top of the map is adjacent to the bottom. Consequently, the four squares in the map corners form a 4-cube. Some cubes are illustrated in [Figure 7.29](#).



**Figure 7.29**  
Karnaugh maps illustrating cubes.

To map a logic function, we place 1s in the squares for which the logic function takes a value of 1. Product terms map 1s into cubes. For example, some product terms are mapped in [Figure 7.30](#).



**Figure 7.30**  
Products of two variables map into 4-cubes on a 4-variable Karnaugh map.

Rectangular arrays (known as cubes) of adjacent squares in a Karnaugh map represent products of logic variables or their inverses.

In a four-variable map consisting of 16 squares, a single logic variable or its inverse covers (maps into) an 8-cube. A product of two variables (such as  $AB$  or  $\bar{A}\bar{B}$ ) covers a 4-cube. A product of three variables maps into a 2-cube.

The Karnaugh map of the logic function

$$F = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + \bar{A}BCD + A\bar{B}\bar{C}D + ABCD \quad (7.31)$$

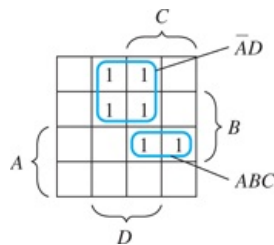
is shown in [Figure 7.31](#). The squares containing 1s form a 4-cube corresponding to the product term  $\bar{A}D$  plus a 2-cube corresponding to  $ABC$ . These are the largest cubes that cover the 1s in the map. Thus, the minimum SOP expression for  $F$  is

$$F = \bar{A}D + ABC \quad (7.32)$$

Because it is relatively easy to spot the set of largest cubes that cover the 1s in a Karnaugh map, we can quickly minimize a logic function.

By finding the fewest and largest (possibly overlapping) cubes for the region in which the logic expression is one, we obtain the minimum SOP for the logic expression.





**Figure 7.31**

Karnaugh map for the logic function of Equation 7.31. From the map, it is evident that  $F = \overline{A}D + ABC$ .

#### Example 7.14 Finding the Minimum SOP Form for a Logic Function

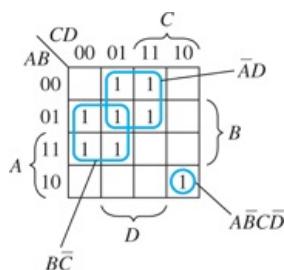
A logic circuit has inputs  $A, B, C$ , and  $D$ . The output of the circuit is given by

$$E = \sum m(1, 3, 4, 5, 7, 10, 12, 13)$$

Find the minimum SOP form for  $E$ .

**Solution**

First, we construct the Karnaugh map. Because there are four input variables, the map contains 16 squares as shown in Figure 7.32. Converting the numbers of the minterms to binary numbers, we obtain 0001, 0011, 0100, 0101, 0111, 1010, 1100, and 1101. Each of these locates a square on the map. For example, 1101 locates the square in the third row and second column, 0011 is the square in the first row and third column, and so forth. Placing a 1 in the square corresponding to each minterm results in the map shown in Figure 7.32.



**Figure 7.32**

Karnaugh map for Example 7.14.

Now we look for the smallest number of the largest size cubes that cover the ones in the map. To cover the ones in this map, we need two 4-cubes and a 1-cube (i.e., a single isolated square) as illustrated in the figure. Finally, the minimum SOP expression is

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

#### Minimum POS Forms

So far, we have concentrated on finding minimum SOP implementations for logic circuits. However, we can easily extend the methods to finding minimal POS circuits by following these steps:

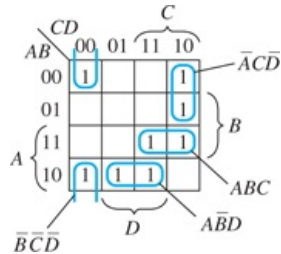
1. Create the Karnaugh map for the desired output.
2. Invert the Karnaugh map, replacing 1s by 0s and vice versa.
3. Look for the least number of cubes of the largest sizes that cover the ones in the inverted map. Then, write the minimum SOP expression for the inverse of desired output.
4. Apply De Morgan's laws to convert the SOP expression to a POS expression. We illustrate with an example.

### Example 7.15 Finding the Minimum POS Form for a Logic Function

Find the minimum POS for the logic variable  $E$  of [Example 7.14](#).

Solution

The Karnaugh map for  $E$  is shown in [Figure 7.32](#). The map for  $\bar{E}$  is obtained by replacing 1s with 0s (blank squares) and vice versa. The result is shown in [Figure 7.33](#).



**Figure 7.33**

Karnaugh map of [Example 7.15](#). (This is the inverse of the map shown in [Figure 7.32](#).)

Now, we look for the smallest number of the largest size cubes that cover the ones in the map. Clearly, there are no 8-cubes or 4-cubes contained in [Figure 7.33](#). A total of eight 1s appear in the map. Thus, the best we can do is to cover the map with four 2-cubes. One option is the grouping shown in the figure, which yields

$$\bar{E} = \bar{A}BC + \bar{A}\bar{B}D + \bar{A}C\bar{D} + \bar{B}\bar{C}\bar{D}$$

Next, we apply De Morgan's laws to obtain a minimum POS form:

$$E = (\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}}) (\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{D}}) (\bar{\bar{A}} + \bar{\bar{C}} + \bar{\bar{D}}) (\bar{\bar{B}} + \bar{\bar{C}} + \bar{\bar{D}})$$

Choosing a different grouping in [Figure 7.33](#) produces another equally good form, which is

$$\bar{E} = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}\bar{D} + \bar{A}C\bar{D} + \bar{B}C\bar{D}$$

Then, applying De Morgan's laws gives another minimum POS form:

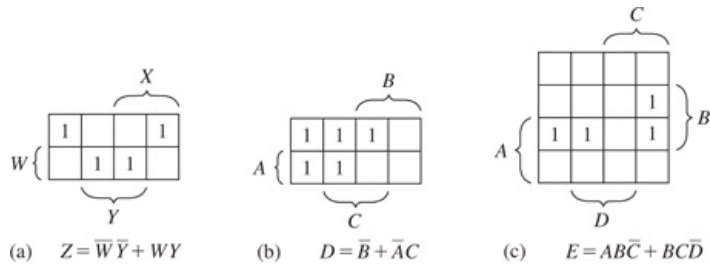
$$E = (\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{C}}) (\bar{\bar{A}} + \bar{\bar{B}} + \bar{\bar{D}}) (\bar{\bar{A}} + \bar{\bar{C}} + \bar{\bar{D}}) (\bar{\bar{B}} + \bar{\bar{C}} + \bar{\bar{D}})$$

### Exercise 7.17

Construct the Karnaugh maps and find the minimum SOP expressions for each of these logic functions:

- $Z = \bar{W}\bar{X}\bar{Y} + \bar{W}X\bar{Y} + \bar{W}X\bar{Y} + WXY$
- $D = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + \bar{A}BC$
- $E = \bar{A}BCD + \bar{A}BC\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D}$

**Answer** See [Figure 7.34](#) .



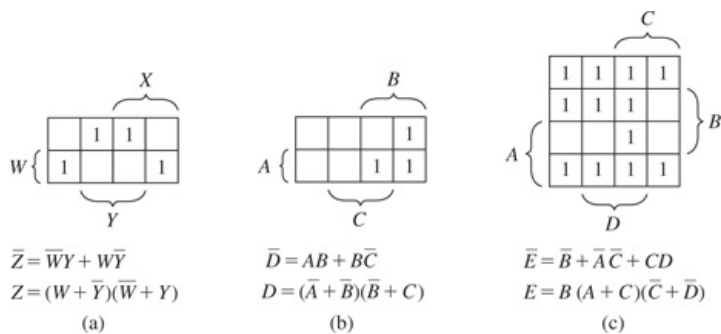
**Figure 7.34**

Answers for [Exercise 7.17](#) .

### Exercise 7.18

Construct the inverse maps and find the minimum POS expressions for each of the logic functions of [Exercise 7.17](#) .

**Answer** See [Figure 7.35](#) .



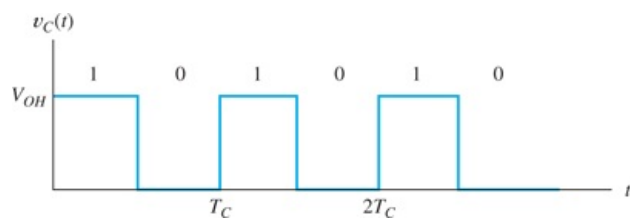
**Figure 7.35**

Answers for [Exercise 7.18](#) .

## 7.6 Sequential Logic Circuits

So far, we have considered combinatorial logic circuits, such as gates, encoders, and decoders, for which the outputs at a given time depend only on the input values at that instant. In this section, we discuss **sequential logic circuits**, for which the outputs depend on past as well as present inputs. We say that such circuits have **memory** because they “remember” past input values.

Often, the operation of a sequential circuit is synchronized by a **clock signal** that consists of periodic logic-1 pulses, as shown in [Figure 7.36](#). The clock signal regulates when the circuits respond to new inputs, so that operations occur in proper sequence. Sequential circuits that are regulated by a clock signal are said to be **synchronous**.



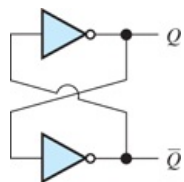
**Figure 7.36**

The clock signal consists of periodic logic-1 pulses.

### Flip-Flops

One of the basic building blocks for sequential circuits is the **flip-flop**. A flip-flop has two stable operating states; therefore, it can store one bit of information. Many useful versions of flip-flops exist, differing in the manner that the clock signal and other input signals control the state of the flip-flop. We discuss several types shortly.

A simple flip-flop can be constructed by using two inverters, with the output of one connected to the input of the other, as shown in [Figure 7.37](#). Two stable states are possible in the circuit. First, the output  $Q$  of the top inverter can be high and then the output of the bottom inverter is low. Thus, the output of the bottom inverter is labeled as  $\bar{Q}$ . Notice that  $Q$  high and  $\bar{Q}$  low are consistent with the logic operation of the inverters, so the circuit can remain in that state. On the other hand,  $Q$  low and  $\bar{Q}$  high are also consistent. The circuit can remain in either state indefinitely.

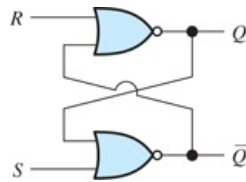


**Figure 7.37**

Simple flip-flop.

### SR Flip-Flop.

The simple two-inverter circuit of [Figure 7.37](#) is not very useful because no provision exists for controlling its state. A more useful circuit is the **set—reset (SR) flip-flop**, consisting of two NOR gates, as shown in [Figure 7.38](#). As long as the S and R inputs are low, the NOR gates act as inverters for the other input signal. Thus, with S and R both low, the SR flip-flop behaves just as the two-inverter circuit of [Figure 7.37](#) does.



**Figure 7.38**

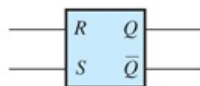
An SR flip-flop can be implemented by cross coupling two NOR gates.

If S is high and R is low,  $\bar{Q}$  is forced low and Q is high (or set). When S returns low, the flip-flop remains in the **set state** (i.e., Q stays high). On the other hand, if R becomes high and S low, Q is forced low. When R returns low, the flip-flop remains in the **reset state** (i.e., Q stays low). In normal operation, R and S are not allowed to be high at the same time. Thus, with R and S low, the SR flip-flop *remembers* which input (R or S) was high most recently.

We use subscripts on logic variables to indicate a sequence of states. For example, the flip-flop output state  $Q_{n-1}$  occurs before  $Q_n$ , which occurs before  $Q_{n+1}$ , and so on. The truth table for the SR flip-flop is shown in [Figure 7.39\(a\)](#). In the first row of the truth table, we see that if both R and S are logic 0, the output remains in the previous state ( $Q_n = Q_{n-1}$ ). The symbol for the SR flip-flop is shown in [Figure 7.39\(b\)](#).

R	S	$Q_n$
0	0	$Q_{n-1}$
0	1	1
1	0	0
1	1	Not allowed

(a) Truth table



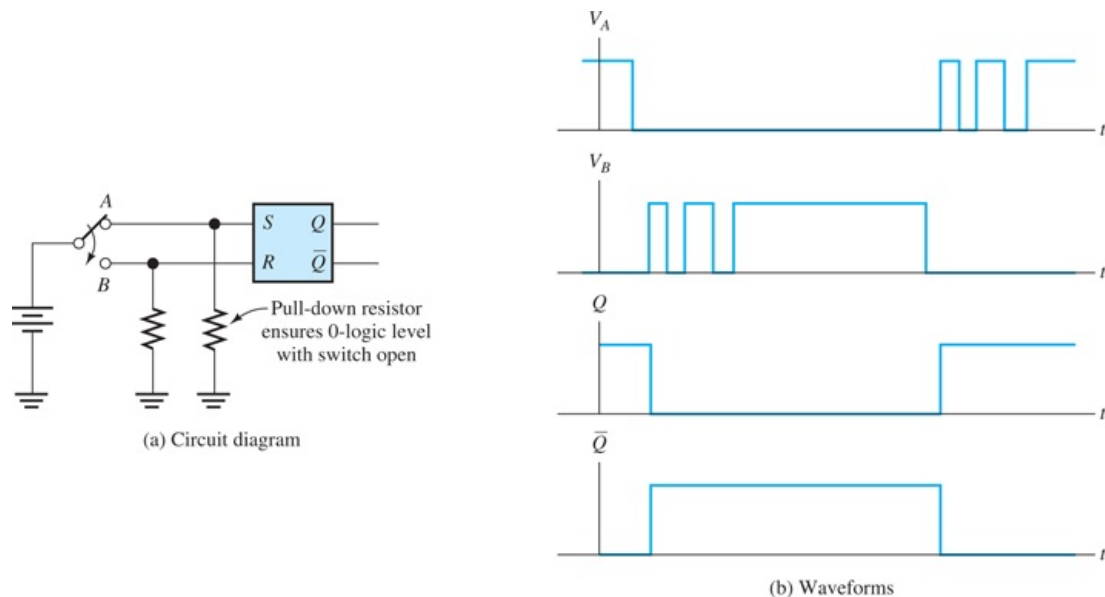
(b) Circuit symbol

**Figure 7.39**

The truth table and symbol for the SR flip-flop.

### Using an SR Flip-Flop to Debounce a Switch.

One application for the SR flip-flop is to *debounce* a switch. Consider the single-pole double-throw switch shown in [Figure 7.40\(a\)](#). When the switch is moved from position A to position B, the waveforms shown in [Figure 7.40\(b\)](#) typically result. At first,  $V_A$  is high because the switch is in position A. Then, the switch breaks contact, and  $V_A$  drops to zero. Next, the switch makes initial contact with B and  $V_B$  goes high. Contact bounce at B again causes  $V_B$  to drop to zero, then back high several times, until finally it ends up high. Later, when the switch is returned to A, contact bounce occurs again.



**Figure 7.40**

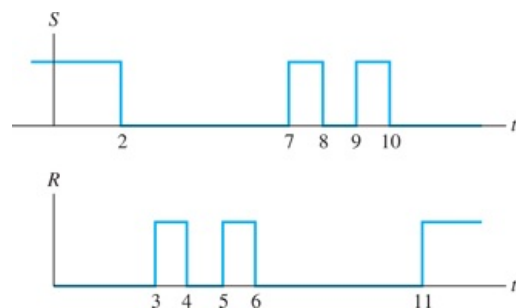
An SR flip-flop can be used to eliminate the effects of switch bounce.

This kind of behavior can be troublesome. For example, a computer keyboard consists of switches that are depressed to select a character. Contact bounce could cause several characters to be accepted by the computer or calculator each time a key is depressed.

An SR flip-flop can eliminate the effects of contact bounce. The switch voltages  $V_A$  and  $V_B$  are connected to the S and R inputs as shown in [Figure 7.40\(a\)](#). At first, when the switch is at position A, the flip-flop is in the set state, and Q is high. When contact is broken with A,  $V_A$  drops to zero, but the flip-flop does not change state until the first time  $V_B$  goes high. As contact bounce occurs, the flip-flop stays in the reset state with Q low. The waveforms for the flip-flop outputs Q and  $\bar{Q}$  are shown in [Figure 7.40\(b\)](#).

#### Exercise 7.19

The waveforms present at the input terminals of an SR flip-flop are shown in [Figure 7.41](#). Sketch the waveforms for Q versus time.



**Figure 7.41**

See [Exercise 7.19](#).

**Answer** See [Figure 7.42](#).

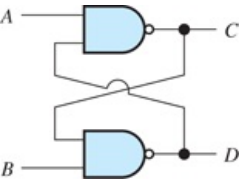


**Figure 7.42**

Answer for [Exercise 7.19](#).

Exercise 7.20

Prepare a truth table similar to that of [Figure 7.39\(a\)](#) for the circuit of [Figure 7.43](#).



**Figure 7.43**

A flip-flop implemented with NAND gates. See [Exercise 7.20](#).

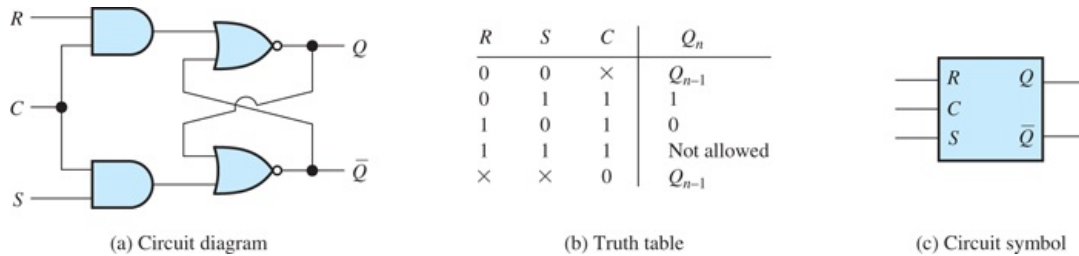
**Answer** See [Table 7.9](#).

**Table 7.9** Truth Table for [Exercise 7.20](#)

<i>A</i>	<i>B</i>	$C_n$	$D_n$	
0	0	1	1	
0	1	1	0	
1	0	0	1	
1	1	$C_{n-1}$	$D_{n-1}$	

### Clocked SR Flip-Flop.

Often, it is advantageous to control the point in time that a flip-flop responds to its inputs. This is accomplished with the **clocked SR flip-flop** shown in Figure 7.44. Two AND gates have been added at the inputs of an SR flip-flop. If the clock signal  $C$  is low, the inputs to the SR flip-flop are both low, and the state cannot change. The clock signal must be high for the  $R$  and  $S$  signals to be transmitted to the input of the SR flip-flop.



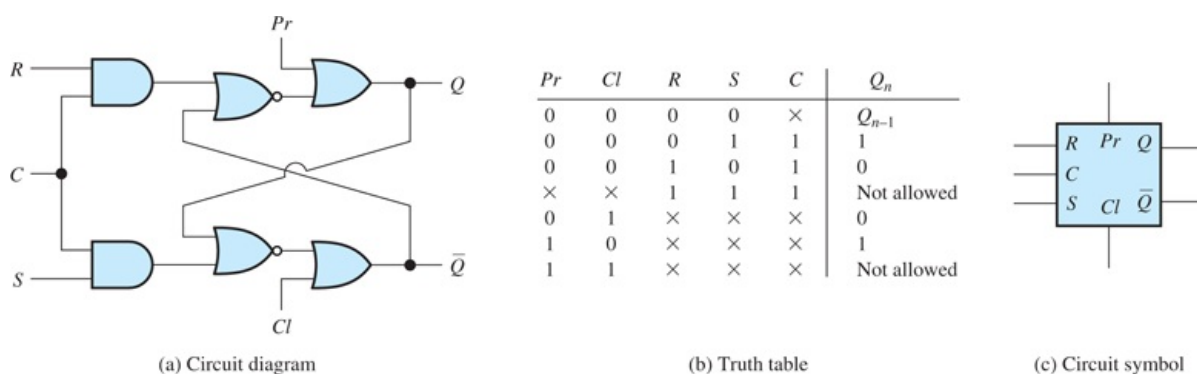
**Figure 7.44**

A clocked SR flip-flop.

The truth table for the clocked SR flip-flop is shown in Figure 7.44(b), and the circuit symbol is shown in Figure 7.44(c). We say that a high clock level **enables** the inputs to the flip-flop. On the other hand, the low clock level **disables** the inputs.

Usually, we design digital systems so that  $R$ ,  $S$ , and  $C$  are not all high at the same time. If all three signals are high and then  $C$  goes low, the state of the flip-flop settles either to  $Q = 1$  or to  $Q = 0$  unpredictably. Usually, systems that behave in an unpredictable manner are not useful.

Sometimes, a clocked SR flip-flop is needed, but it is also necessary to be able to set or clear the flip-flop state independent of the clock. A circuit having this feature is shown in Figure 7.45(a). If the **preset input**  $Pr$  is high,  $Q$  becomes high even if the clock is low. Similarly, the **clear input**  $Cl$  can force  $Q$  low. The  $Pr$  and  $Cl$  inputs are called **asynchronous inputs** because their effect is not synchronized by the clock signal. On the other hand, the  $R$  and  $S$  inputs are recognized only if the clock signal is high, and are therefore called **synchronous inputs**.



**Figure 7.45**

A clocked SR flip-flop with asynchronous preset and clear inputs.

### Edge-Triggered D Flip-Flop.

So far, we have considered circuits for which the level of the clock signal *enables* or *disables* other input signals. On the other hand, **edge-triggered** circuits respond to their inputs only at a transition in the clock signal. If the clock signal is steady, either high or low, the inputs are disabled. At the clock transition, the flip-flop responds to the inputs present just prior to the transition. **Positive-edge-triggered** circuits respond when the clock signal switches from low to high. Conversely, **negative-edge-triggered** circuits respond on the transition from high to low. The positive-going edge of the clock is also called the **leading edge**, and

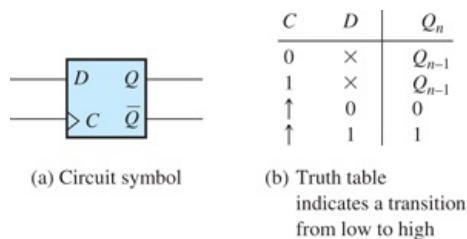


the negative-going edge is called the **trailing edge**. A clock signal illustrating these points is shown in [Figure 7.46](#). Thus, clocked flip-flops can be sensitive either to the level of the clock or to transitions.



**Figure 7.46**  
Clock signal.

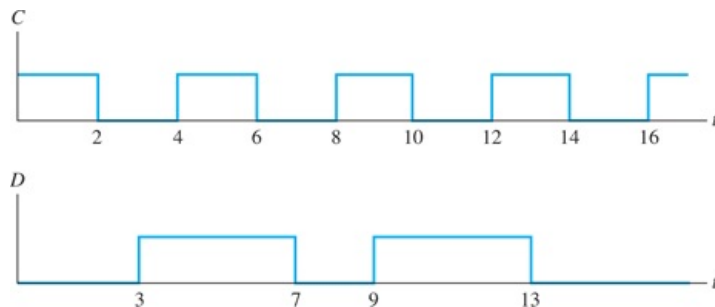
An example of an edge-triggered circuit is the **D flip-flop**, which is also known as the **delay flip-flop**. Its output takes the value of the input that was present just prior to the triggering clock transition. The circuit symbol for the edge-triggered D flip-flop is shown in [Figure 7.47\(a\)](#). The “knife edge” symbol at the C input indicates that the flip-flop is edge triggered. The truth table for a positive-edge-triggered version is shown in [Figure 7.47\(b\)](#). Notice the symbols in the clock column of the truth table, indicating transitions of the clock signal from low to high.



**Figure 7.47**  
A positive-edge-triggered D flip-flop.

#### Exercise 7.21

The input signals to a positive-edge-triggered D flip-flop are shown in [Figure 7.48](#). Sketch the output Q to scale versus time. (Assume that Q is low prior to  $t = 2$ .)



**Figure 7.48**  
See [Exercise 7.21](#).

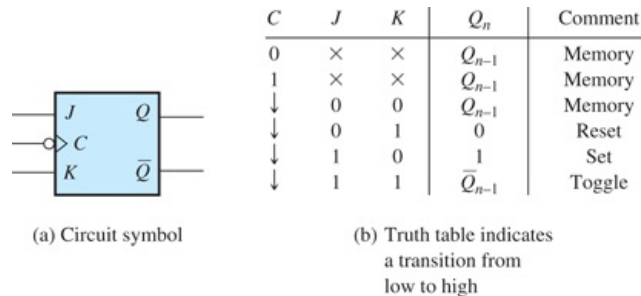
**Answer** See [Figure 7.49](#).



**Figure 7.49**  
Answer for [Exercise 7.21](#).

### JK Flip-Flop.

The circuit symbol and truth table for a negative-edge-triggered **JK flip-flop** are shown in [Figure 7.50](#). Its operation is very similar to that of an *SR* flip-flop except that if both control inputs (*J* and *K*) are high, the state changes on the next negative-going clock edge. Thus when both *J* and *K* are high, the output of the flip-flop **toggles** on each cycle of the clock—switching from high to low on one negative-going clock transition, back to high on the next negative transition, and so on.

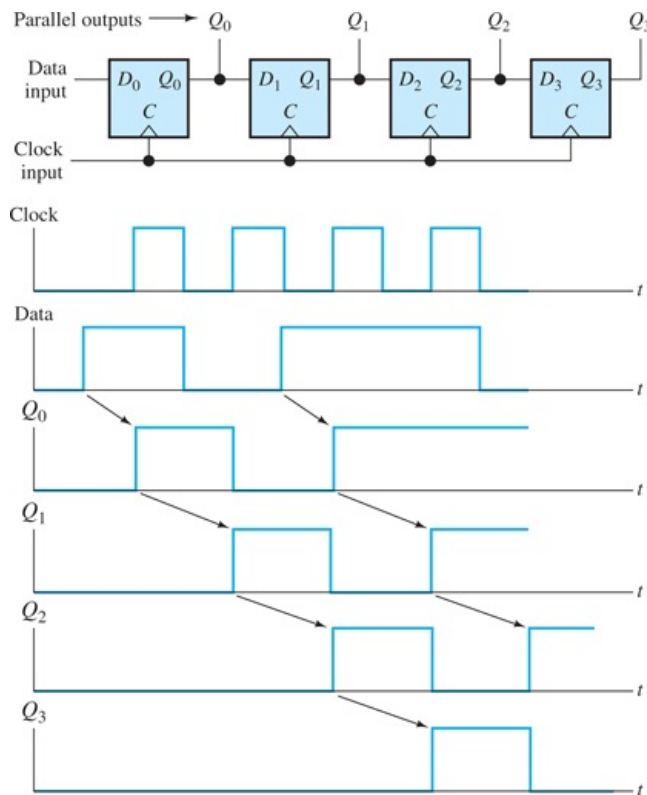


**Figure 7.50**

Negative-edge-triggered *JK* flip-flop.

## Serial-In Parallel-Out Shift Register

A **register** is an array of flip-flops that is used to store or manipulate the bits of a digital word. For example, if we connect several positive-edge-triggered  $D$  flip-flops as shown in [Figure 7.51](#), a **serial-in parallel-out shift register** results. As the name implies, the digital input word is shifted through the register moving one stage for each clock pulse.

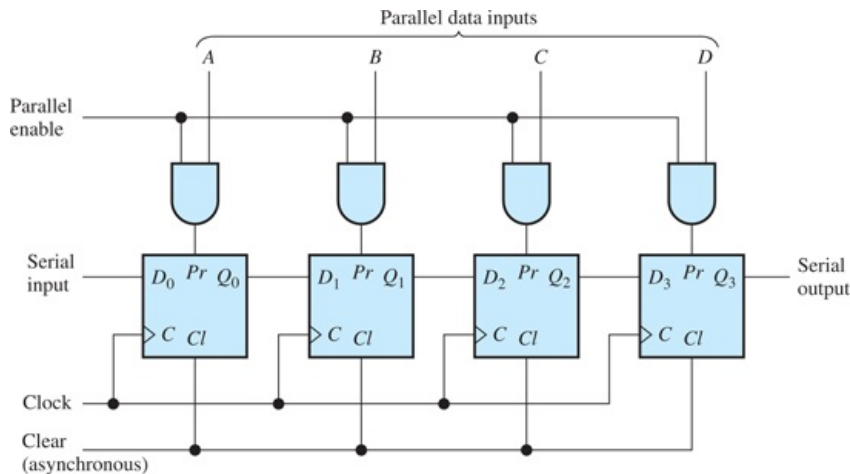


**Figure 7.51**  
Serial-input parallel-output shift register.

The waveforms shown in [Figure 7.51](#) illustrate the operation of the shift register. We assume that the flip-flops are initially (  $t = 0$  ) all in the reset state (  $Q_0 = Q_1 = Q_2 = Q_3 = 0$  ). The input data are applied to the input of the first stage serially (i.e., one bit after another). On the leading edge of the first clock pulse, the first data bit is transferred into the first stage. On the second clock pulse, the first bit is transferred to the second stage, and the second bit is transferred into the first stage. After four clock pulses, four bits of input data have been transferred into the shift register. Thus, serial data applied to the input are converted to parallel form available at the outputs of the stages of the shift register.

## Parallel-In Serial-Out Shift Register

Sometimes, we have parallel data that we wish to transmit serially. Then, the **parallel-in serial-out shift register** shown in [Figure 7.52](#) is useful. This register consists of four positive-edge-triggered  $D$  flip-flops with asynchronous preset and clear inputs. First, the register is cleared by applying a high pulse to the clear input. (The clear input is asynchronous, so a clock pulse is not necessary to clear the register.) Parallel data are applied to the  $A$ ,  $B$ ,  $C$ , and  $D$  inputs. Then, a high pulse is applied to the parallel enable (PE) input. The result is to set each flip-flop for which the corresponding data line is high. Thus, four parallel bits are loaded into the stages of the register. Then, application of clock pulses produces the data in serial form at the output of the last stage.

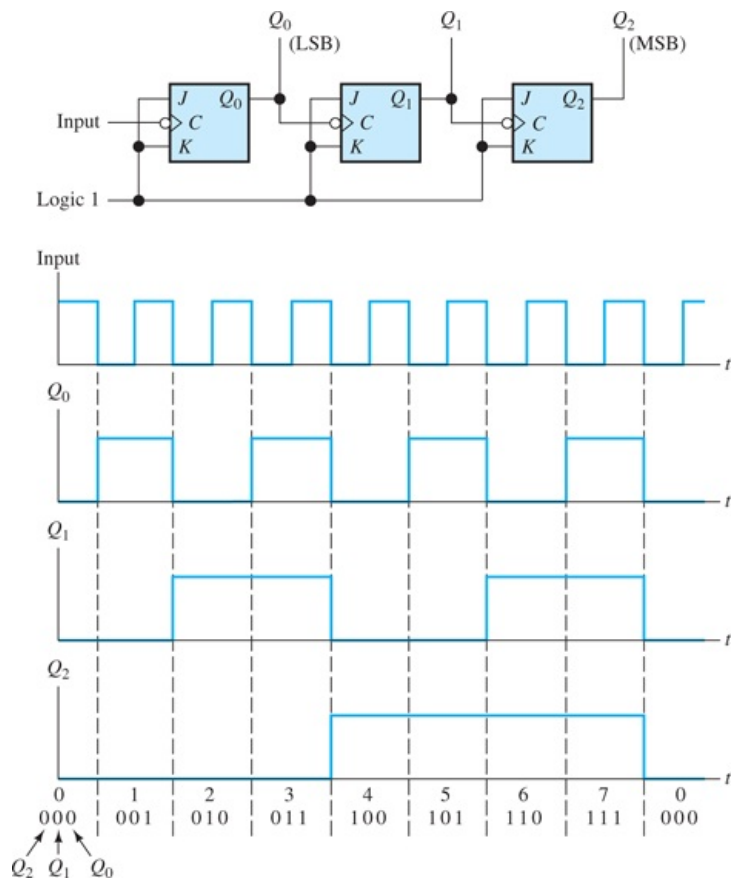


**Figure 7.52**

Parallel-input serial-output shift register.

## Counters

Counters are used to count the pulses of an input signal. An example is the **ripple counter** shown in [Figure 7.53](#). It consists of a cascade of  $JK$  flip-flops. Reference to [Figure 7.50](#) shows that with the  $J$  and  $K$  inputs high, the  $Q$ -output of the flip-flop toggles on each falling edge of the clock input. The input pulses to be counted are connected to the clock input of the first stage, and the output of the first stage is connected to the clock input of the second stage.



**Figure 7.53**  
Ripple counter.

Assume that the flip-flops are initially all in the reset state ( $Q = 0$ ). When the falling edge of the first input pulse occurs,  $Q_0$  changes to logic 1. On the falling edge of the second pulse,  $Q_0$  toggles back to logic 0, and the resulting falling input to the second stage causes  $Q_1$  to become high. As shown by the waveforms in [Figure 7.53](#), after seven pulses, the shift register is in the 111 state. On the eighth pulse, the counter returns to the 000 state. Thus, we say that this is a modulo-8 or mod-8 counter.

## PRACTICAL APPLICATION

### 7.1



### Biomedical Engineering Application of Electronics: Cardiac Pacemaker

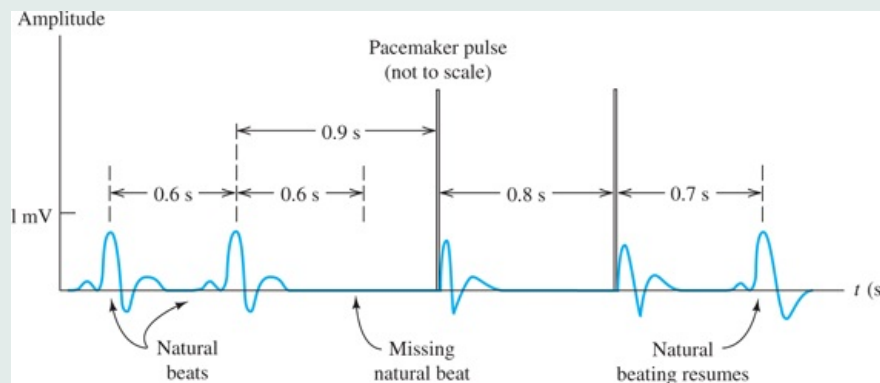
In certain types of heart disease, the biological signals that should stimulate the heart to beat are blocked from reaching the heart muscle. When this blockage occurs, the heart muscle may spontaneously beat at a very low rate, so death does not occur. However, the afflicted person is not able to function at a normal level of activity because of the low heart rate. The application of electrical pacemaker pulses to force beating at a higher rate is dramatically helpful in many of these cases.

Sometimes, the blockage of the natural pacemaking is not complete. In this case, the heart beats normally part of the time but experiences missed beats sporadically. A demand pacemaker can be useful for the patient with partial blockage. The demand pacemaker contains circuits that sense natural heartbeats and apply an electrical pulse to the heart muscle only if a beat does not occur within a predetermined interval. If natural beats are detected, no pulses are applied. This type of circuit is called a *demand* pacemaker because pulses are issued only when needed.

It turns out to be advantageous to the patient for the heart to beat naturally, provided that its natural rate is above some limit. On the other hand, if artificial pulses are required, a slightly higher rate is better. Typical values are a natural limit of 66.7 beats per minute (corresponding to 0.9 s between beats) and 75 beats per minute (corresponding to 0.8 s between beats) for forced pacing. Thus, in a typical situation, the circuit waits 0.9 s after a natural beat before applying a pacing pulse but waits only 0.8 s after an artificial pulse before applying another artificial pulse.

Another feature of the pacemaker is that it should ignore signals from the heart for a short period (about 0.4 s) after detection of a natural beat or after issuing a pacemaker pulse. This is because natural signals occur during the contraction and relaxation of the heart muscle. These signals should not cause the timing functions of the pacemaker to be reset. Thus, when the start of a contraction is sensed (or is stimulated by the circuit), the timing circuits are reset, but cannot be reset again until the contraction and relaxation is over.

The electrical signals present at the terminals of the pacemaker are shown for a typical case in [Figure PA7.1](#). At the left side of the tracing, the signals occurring during natural beating are shown. Then, blockage of the natural beating occurs, and a pacemaker pulse is issued 0.9 s after the last natural beat. The amplitude of these pulses is typically 5 V and their durations are 0.7 ms. After the pacemaker pulse, natural signals occur from the contraction and relaxation of the heart. These are ignored by the circuit. After two forced cycles, the heart again begins natural beating.

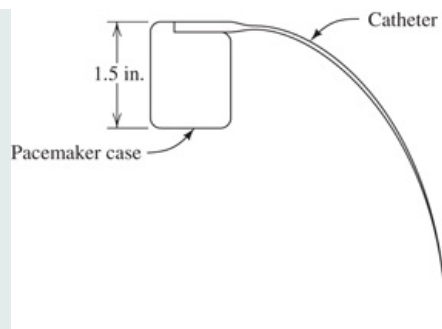


**FIGURE**

**PA7.1**

Typical electrical signals at the terminals of a demand cardiac pacemaker.

The pacemaker circuitry and battery are enclosed in a metal case. This is implanted under the skin on the chest of the patient. A wire (enclosed in an insulating tube known as a catheter) leads from the pacemaker through an artery into the interior of the heart. The electrical terminals of the pacemaker are the metal case and the tip of the catheter. A pacemaker and catheter are shown in [Figure PA7.2](#).

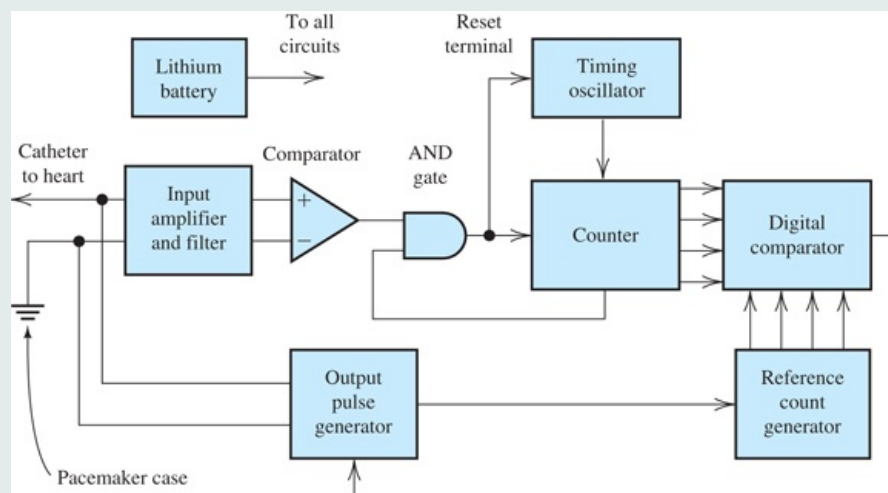


**FIGURE**

**PA7.2**

A cardiac pacemaker and catheter.

The block diagram of a typical demand pacemaker is shown in [Figure PA7.3](#). Notice that the electrical terminals serve both as the input to the amplifier and the output terminals for the pulse generator. The input amplifier increases the amplitude of the natural signals. Amplification is necessary because the natural heart signals have a very small amplitude (on the order of 1 mV), which must be increased before a comparator circuit can be employed to decide on the presence or absence of a natural heartbeat. Filtering to eliminate certain frequency components is employed in the amplifier to enhance the detectability of the heartbeats. Furthermore, proper filtering eliminates the possibility that radio or power-line signals will interfere with the pacemaker. Thus, the important specifications of the amplifier are its gain and frequency response.



**FIGURE**

**PA7.3**

Block diagram of a demand cardiac pacemaker.

Amplifiers and comparators are discussed at length in Part III of this book.


The output of the amplifier is applied to an analog comparator circuit that compares the amplified and filtered signal to a threshold value. If the input signal becomes higher than the threshold, the output of the comparator becomes high. Thus, the comparator output is a digital signal that indicates the detection of either a natural heartbeat or an output-pacing pulse.

This detection decision is passed through an AND gate to the counting and timer circuitry. The second input to the AND gate comes from the counter circuit and is low for 0.4 s after a detected beat. Thus, the AND gate prevents another decision from passing through for 0.4 s after the first. In this manner, the pacemaker ignores input signals for 0.4 s after the start of a natural or forced beat.

The timing functions are accomplished by counting the output cycles of a timing oscillator. The timing oscillator generates a square wave with a period of 0.1 s. When a heartbeat is detected, the timing oscillator is reset to the beginning of a cycle, so the completion of each oscillator cycle occurs exactly at an integer multiple of 0.1 s after the heartbeat. The timing oscillator must maintain a precise period because the proper operation of the circuit depends on accurate timing. Thus, the frequency stability of the timing oscillator is its primary specification.

The counter is a digital circuit that counts the output cycles of the timing oscillator. The counter is also capable of being reset to zero when a heartbeat occurs or when a pacemaker pulse is issued. The digital signals produced by the counter are applied to a digital comparator. Signals from a reference circuit are also applied to the digital comparator. The reference count is nine if the last beat was natural, but the reference count is eight if the last beat was forced. When the counter input to the digital comparator agrees with the reference count, the output of the digital comparator goes to a high level. This causes the pulse generator to issue an output pulse.

The pulse generator must produce output pulses of a specified amplitude and duration. In some designs, the output pulse amplitude is required to be higher than the battery voltage. This can be accomplished by charging capacitors in parallel with the battery and then switching them to series to generate the higher voltage.

What we have described so far is a relatively simple demand pacemaker readily implemented with registers, counters, and gates. By substituting a **microcontroller** and **software** (discussed in [Chapter 8](#) ) for the digital functions of the pacemaker, many additional useful features can be accommodated. For example, an accelerometer can sense the physical activity of the patient and the software can use this information to adjust heart rate. Communicating through magnetic fields linked to a coil in the pacemaker, the physician can instruct the software to alter the operating characteristics appropriately for each patient.

Extremely low power consumption is an important requirement for all pacemaker circuits. This is because the circuit must operate from a small battery for many years. After all, replacement of the battery requires a surgical procedure. When pacing pulses are not needed, a typical circuit can function with a few microamperes from a 2.5-V battery. When pacing pulses are required, the average current drain increases to a few tens of microamperes. This higher current consumption is unavoidable because of the output power required in the form of pacing pulses.

High reliability is very important because malfunction can be life-threatening. A very detailed failure-mode analysis must be performed for every component in the circuit. This is necessary because some failures are much more threatening than others. For example, if the pacemaker fails to issue pacemaking pulses, the person may survive because of the natural (low-rate) pacing of the heart muscle. On the other hand, if the timing generator fails in such a manner that it runs too fast, the unfortunate person's heart will be forced to beat much too fast. This can be quickly fatal, especially for those in a weakened condition from heart disease.

Clearly, circuit design is not the total solution to this problem. Physicians must provide the specifications for the pacemaker. Mechanical and chemical engineers must be involved in selecting the materials and form of the catheter and the case. By working in teams, engineers and physicians have designed electronic pacemakers that provide very dramatic health improvements for many people. Those who have contributed can be most proud of their achievements. Nevertheless, many further improvements are possible and may be achieved by some of the students of this book.



## Conclusions

In this chapter, we have seen that complex combinatorial logic functions can be achieved simply by interconnecting NAND gates (or NOR gates). Furthermore, logic gates can be interconnected to form flip-flops. Interconnections of flip-flops form registers. A complex digital system, such as a computer, consists of many gates, flip-flops, and registers. Thus, logic gates are the basic building blocks for complex digital systems.

## Summary

1. Digital signals are more immune to the effects of noise than analog signals. The logic levels of a digital signal can be determined after noise is added, provided that the noise amplitude is not too high.
2. Component values in digital circuits do not need to be as precise as in analog circuits.
3. Digital circuits are more amenable than analog circuits to implementation as large-scale ICs.
4. In positive logic, the higher voltage represents logic 1.
5. Numerical data can be represented in decimal, binary, octal, hexadecimal, or BCD forms.
6. In the Gray code, each word differs from adjacent words in only a single bit. The Gray code is useful for representing position or angular displacement.
7. In computers, numbers are frequently represented in signed two's-complement form. (See [Figure 7.11](#) on page 366.)
8. Logic variables take two values, logic 1 or logic 0. Logic variables may be combined by the AND, OR, and inversion operations according to the rules of Boolean algebra. A truth table lists all combinations of input variables and the corresponding output.
9. De Morgan's laws state that

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

and


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

10. NAND (or NOR) gates are sufficient to realize any combinatorial logic function.
11. Any combinatorial logic function can be written as a Boolean expression consisting of a logical SOP. Each product is a minterm corresponding to a line of the truth table for which the output variable is logic 1.
12. Any combinatorial logic function can be written as a Boolean expression consisting of a logical POS. Each sum is a maxterm corresponding to a line of the truth table for which the output variable is logic 0.
13. Many useful combinatorial circuits, known as decoders, encoders, or translators, are available as ICs.
14. Karnaugh maps can be used to minimize the number of gates needed to implement a given logic function.
15. Sequential logic circuits are said to have memory because their outputs depend on past as well as present inputs. Synchronous or clocked sequential circuits are regulated by a clock signal.
16. Various types of flip-flops are the *SR* flip-flop, the clocked flip-flop, the *D* flip-flop, and the *JK* flip-flop.
17. Flip-flops can be combined to form registers that are used to store or manipulate digital words.
18. Logic gates can be interconnected to form flip-flops. Interconnections of flip-flops form registers. A complex digital system, such as a computer, consists of many gates, flip-flops, and registers. Thus, logic gates are the basic building blocks for complex digital systems.

# Problems

## Section 7.1: Basic Logic Circuit Concepts

**\*P7.1.** State three advantages of digital technology compared with analog technology.

\* Denotes that answers are contained in the Student Solutions files. See [Appendix E](#)  for more information about accessing the Student Solutions.

**P7.2.** Define these terms: *bit*, *byte*, and *nibble*.

**P7.3.** Explain the difference between positive logic and negative logic.

**P7.4.** What are noise margins? Why are they important?

**P7.5.** How is serial transmission of a digital word different from parallel transmission?

## Section 7.2: Representation of Numerical Data in Binary Form

**P7.6.** Convert the following binary numbers to decimal form:

- a. \* 101.101;
- b. 0111.11;
- c. 1010.01;
- d. 111.111;
- e. 1000.0101;
- f. \* 10101.011.

**P7.7.** Express the following decimal numbers in binary form and in BCD form:

- a. 17;
- b. 8.5;
- c. \* 9.75;
- d. 73.03125;
- e. 67.375.

**P7.8.** How many bits per word are needed to represent the decimal integers 0 through 100? 0 through 1000? 0 through  $10^6$ ?

**P7.9.** Add these pairs of binary numbers:

- a. \* 1101.11 and 101.111;
- b. 1011 and 101;
- c. 10001.111 and 0101.001.

**P7.10.** Find the result (in BCD format) of adding the BCD numbers:

- a. \* 10010011.0101 and 00110111.0001;
- b. 01011000.1000 and 10001001.1001.

**P7.11.** Express the following decimal numbers in binary, octal, and hexadecimal forms:

- a. 173;
- b. 299.5;
- c. 735.75;
- d. \* 313.0625;
- e. 112.25.

**P7.12.** Write each of the following decimal numbers as an eight-bit signed two's-complement number:

- a. 19;
- b.  $-19$ ;
- c. \* 75;
- d. \*  $-87$ ;
- e.  $-95$ ;
- f. 99.

**P7.13.** Express each of the following hexadecimal numbers in binary, octal, and decimal forms:

- a.  $FA.F_{16}$ ;
- b.  $2A.1_{16}$ ;
- c.  $777.7_{16}$ .

**P7.14.** Express each of the following octal numbers in binary, hexadecimal, and decimal forms:

- a.  $777.7_8$ ;
- b.  $123.5_8$ ;
- c.  $24.4_8$ .

**P7.15.** What number follows 777 when counting in

- a. decimal;
- b. octal;
- c. hexadecimal?

**P7.16.** What range of decimal integers can be represented by

- a. three-bit binary numbers;
- b. three-digit octal numbers;
- c. three-digit hexadecimal numbers?

**\*P7.17.** Starting with the three-bit Gray code listed in Figure 7.9, construct a four-bit Gray code. For what applications is a Gray code advantageous? Why?

**P7.18.** Convert the following numbers to decimal form:

- a. \*  $FA5.6_{16}$ ;
- b. \*  $725.3_8$ ;
- c.  $3F4.8_{16}$ ;
- d.  $73.25_8$ ;
- e.  $FF.F0_{16}$ .

**P7.19.** Find the one's and two's complements of the binary numbers:

- a. \* 11101000;
- b. 00000000;
- c. 10101010;
- d. 11111100;
- e. 11000000.

**P7.20.** Perform these operations by using eight-bit signed two's-complement arithmetic:

- a.  $17_{10} + 15_{10}$ ;
- b.  $17_{10} - 15_{10}$ ;
- c. \*  $33_{10} - 37_{10}$ ;
- d.  $15_{10} - 63_{10}$ ;
- e.  $49_{10} - 44_{10}$ .

**P7.21.** Describe how to test whether overflow or underflow has occurred in adding signed two's-complement numbers.

### Section 7.3: Combinatorial Logic Circuits

**P7.22.** What is a truth table?

**\*P7.23.** State De Morgan's laws.

**P7.24.** Draw the circuit symbol and list the truth table for the following: an AND gate, an OR gate, an inverter, a NAND gate, a NOR gate, and an XOR gate. Assume two inputs for each gate (except the inverter).

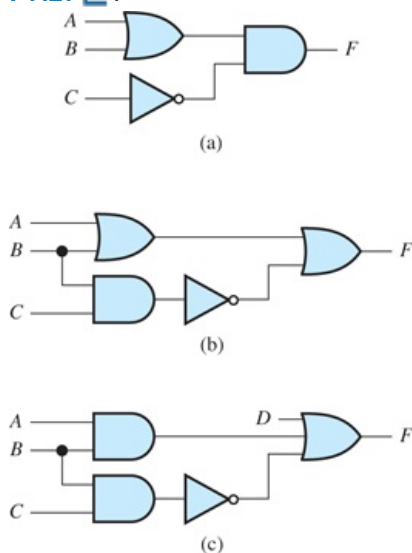
**P7.25.** Describe a method for proving the validity of a Boolean algebra identity.

**P7.26.** Write the truth table for each of these Boolean expressions:

- $D = ABC + \bar{A}\bar{B}$
- $*E = AB + \bar{A}BC + \bar{C}D$
- $Z = WX + (\bar{W} + Y)$
- $D = \bar{A} + \bar{A}B + C$
- $D = (\bar{A} + BC)$

**P7.27.** Write a Boolean expression for the output of each of the logic circuits shown in [Figure](#)

**P7.27** .



**Figure P7.27**

**\*P7.28.** Use a truth table to prove the identity  

$$(\bar{A} + B)(\bar{A} + C) = \bar{A} + BC$$

**P7.29.** Use a truth table to prove the identity  

$$(\bar{A} + B)(\bar{A} + AB) = \bar{A} + B$$

**P7.30.** Use a truth table to prove the identity  

$$A + \bar{A}B = A + B$$

**P7.31.** Use a truth table to prove the identity  

$$ABC + \bar{A}BC + A\bar{B}\bar{C} + \bar{A}\bar{B}C = A + B + C$$

**P7.32.** Draw a circuit to realize each of the following expressions using AND gates, OR gates, and inverters:

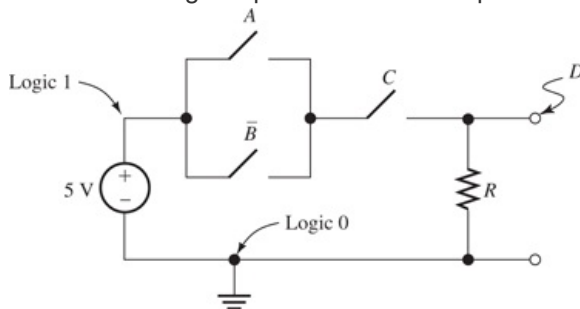
- a.  $F = A + \bar{B}C$   
 b.  $F = \bar{A}\bar{B}C + A\bar{B}C + A\bar{B}\bar{C}$   
 c. \*  $F = (\bar{A} + \bar{B} + C)(A + B + \bar{C})$   
        $(A + \bar{B} + C)$

**P7.33.** Replace the AND operations by ORs and vice versa by applying De Morgan's laws to each of these expressions:

- a.  $F = AB + (\bar{C} + A)\bar{D}$   
 b.  $F = A(\bar{B} + C) + D$   
 c.  $F = \bar{A}\bar{B}C + A(\bar{B} + C)$   
 d. \*  $F = (A + B + C)(A + \bar{B} + C)$   
        $(\bar{A} + B + \bar{C})$   
 e. \*  $F = \bar{A}\bar{B}C + A\bar{B}C + A\bar{B}\bar{C}$

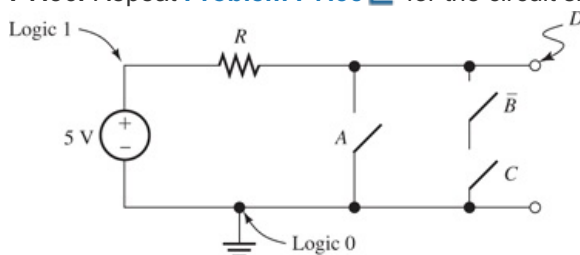
**P7.34.** Why are NAND gates said to be *sufficient* for combinatorial logic? What other type of gate is sufficient?

**P7.35.** Consider the circuit shown in [Figure P7.35](#). The switches are controlled by logic variables such that, if  $A$  is high, switch  $A$  is closed, and if  $A$  is low, switch  $A$  is open. Conversely, if  $B$  is high, the switch labeled  $\bar{B}$  is open, and if  $B$  is low, the switch labeled  $\bar{B}$  is closed. The output variable is high if the output voltage is 5 V, and the output variable is low if the output voltage is zero. Write a logic expression for the output variable. Construct the truth table for the circuit.



**Figure P7.35**

**P7.36.** Repeat [Problem P7.35](#) for the circuit shown in [Figure P7.36](#).



**Figure P7.36**

**P7.37.** Sometimes “bubbles” are used to indicate inverters on the input lines to a gate, as illustrated in [Figure P7.37](#). What are the equivalent gates for those of [Figure P7.37](#)? Justify your answers.

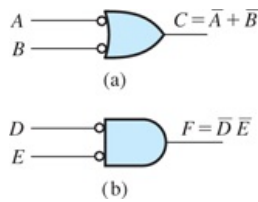


Figure P7.37

## Section 7.4: Synthesis of Logic Circuits

**P7.38.** Using the sum-of-products approach, describe the synthesis of a logic expression from a truth table. Repeat for the product-of-sums approach.

**P7.39.** Give an example of a decoder.

**\*P7.40.** Consider [Table P7.40](#).  $A$ ,  $B$ , and  $C$  represent logic-variable input signals;  $F$  through  $K$  are outputs. Using the product-of-sums approach, write a Boolean expression for  $F$  in terms of the inputs. Repeat by using the sum-of-products approach.

Table P7.40

$A$	$B$	$C$	$F$	$G$	$H$	$I$	$J$	$K$
0	0	0	1	1	1	0	0	1
0	0	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0	0
0	1	1	0	1	0	1	1	0
1	0	0	0	0	1	0	0	0
1	0	1	1	0	1	0	1	0
1	1	0	0	0	1	1	1	1
1	1	1	1	0	1	1	1	1

**P7.41.** Repeat [Problem P7.40](#) for  $G$ .

**P7.42.** Repeat [Problem P7.40](#) for  $H$ .

**P7.43.** Repeat [Problem P7.40](#) for  $I$ .

**P7.44.** Repeat [Problem P7.40](#) for  $J$ .

**P7.45.** Repeat [Problem P7.40](#) for  $K$ .

**P7.46.** Show how to implement the sum-of-products circuit shown in [Figure P7.46](#) by using only NAND gates.

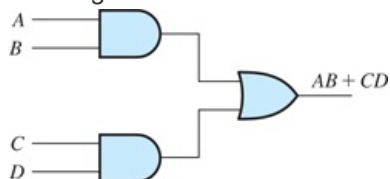
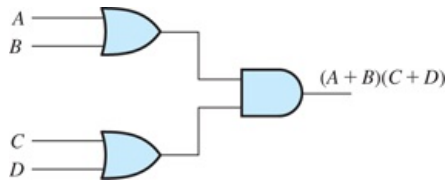


Figure P7.46

**P7.47.** Show how to implement the product-of-sums circuit shown in [Figure P7.47](#) by using only NOR gates.



**Figure P7.47**

**P7.48.** Design a logic circuit to control electrical power to the engine ignition of a speed boat. Logic output  $I$  is to become high if ignition power is to be applied and is to remain low otherwise. Gasoline fumes in the engine compartment present a serious hazard of explosion. A sensor provides a logic input  $F$  that is high if fumes are present. Ignition power should not be applied if fumes are present. To help prevent accidents, ignition power should not be applied while the outdrive is in gear. Logic signal  $G$  is high if the outdrive is in gear and is low otherwise. A blower is provided to clear fumes from the engine compartment and is to be operated for 5 minutes before applying ignition power. Logic signal  $B$  becomes high after the blower has been in operation for 5 minutes. Finally, an emergency override signal  $E$  is provided so that the operator can choose to apply ignition power even if the blower has not operated for 5 minutes and if the outdrive is in gear, but not if gasoline fumes are present.

- Prepare a truth table listing all combinations of the input signals  $B$ ,  $E$ ,  $F$ , and  $G$ . Also, show the desired value of  $I$  for each row in the table.
- Using the sum-of-products approach, write a Boolean expression for  $I$ .
- Using the product-of-sums approach, write a Boolean expression for  $I$ .
- Try to manipulate the expressions of parts (b) and (c) to obtain a logic circuit having the least number of gates and inverters. Use AND gates, OR gates, and inverters.

**\*P7.49.** Use only NAND gates to find a way to implement the XOR function for two inputs,  $A$  and  $B$ . [Hint: The inputs of a two-input NAND can be wired together to obtain an inverter. List the truth table and write the SOP expression. Then, apply De Morgan's laws to convert the OR operation to AND.]

**P7.50.** Use only two-input NOR gates to find a way to implement the XOR function for two inputs,  $A$  and  $B$ . [Hint: The inputs of a two-input NOR can be wired together to obtain an inverter. List the truth table and write the POS expression. Then, apply De Morgan's laws to convert the AND operation to OR.]

**P7.51.** Consider the BCD-to-seven-segment decoder discussed in conjunction with [Figure 7.26](#) on page 380. Suppose that the BCD data are represented by the logic variables  $B_8$ ,  $B_4$ ,  $B_2$ , and  $B_1$ . For example, the decimal number 7 is represented in BCD by the word 0111 in which the leftmost bit is  $B_8 = 0$ , the second bit is  $B_4 = 1$ , and so forth.

- Find a logic circuit based on the product of maxterms having output  $A$  that is high only if segment  $A$  of the display is to be on.
- Repeat for segment  $B$ .

**\*P7.52.** Suppose that two numbers in signed two's-complement form have been added.  $S_1$  is the sign bit of the first number,  $S_2$  is the sign bit of the second number, and  $S_T$  is the sign bit of the total. Suppose that we want a logic circuit with output  $E$  that is high if either overflow or underflow has occurred; otherwise,  $E$  is to remain low.

- Write the truth table.
- Find an SOP expression composed of minterms for  $E$ .
- Draw a circuit that yields  $E$ , using AND, OR, and NOT gates.

## Section 7.5: Minimization of Logic Circuits

**\*P7.53.**



- a. Construct a Karnaugh map for the logic function

$$F = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + \bar{A}BCD + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + A\bar{B}CD + AB\bar{C}\bar{D} + AB\bar{C}D + ABC\bar{D} + ABCD$$

- b. Find the minimum SOP expression. c. Find the minimum POS expression.

**P7.54.** A logic circuit has inputs  $A$ ,  $B$ , and  $C$ . The output of the circuit is given by

$$D = \sum m(0, 3, 4)$$

- a. Construct the Karnaugh map for  $D$ .  
b. Find the minimum SOP expression.  
c. Find two equally good minimum POS expressions.

**P7.55.** A logic circuit has inputs  $A$ ,  $B$ , and  $C$ . The output of the circuit is given by

$$D = \prod M(1, 3, 4, 6)$$

- a. Construct the Karnaugh map for  $D$ .  
b. Find the minimum SOP expression.  
c. Find the minimum POS expression.

**P7.56.**

- a. Construct a Karnaugh map for the logic function

$$D = ABC + ABC + ABC + BC$$

- b. Find the minimum SOP expression and realize the function, using AND, OR, and NOT gates.  
c. Find the minimum POS expression and realize the function, using AND, OR, and NOT gates.

**P7.57.**

- a. Construct a Karnaugh map for the logic function

$$F = ABCD + ABCD + ABCD + ABCD$$

- b. Find the minimum SOP expression.  
c. Realize the minimum SOP function, using AND, OR, and NOT gates.  
d. Find the minimum POS expression.

**\*P7.58.** Consider [Table P7.58](#) in which  $A$ ,  $B$ ,  $C$ , and  $D$  are input variables.  $F$ ,  $G$ ,  $H$ , and  $I$  are the output variables.

- a. Construct a Karnaugh map for the output variable  $F$ .  
b. Find the minimum SOP expression for this logic function.  
c. Use AND, OR, and NOT gates to realize the minimum SOP function.  
d. Find the minimum POS expression.

Table P7.58

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>		<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>
0	0	0	0		0	0	0	1
0	0	0	1		1	0	0	1
0	0	1	0		0	0	0	1
0	0	1	1		0	0	0	0
0	1	0	0		0	0	0	1
0	1	0	1		1	0	0	1
0	1	1	0		0	0	1	1
0	1	1	1		0	1	1	0
1	0	0	0		0	0	0	0
1	0	0	1		1	0	1	0
1	0	1	0		0	0	0	0
1	0	1	1		0	0	1	0
1	1	0	0		1	0	0	0
1	1	0	1		1	0	1	0
1	1	1	0		1	1	1	0
1	1	1	1		1	1	1	0

**P7.59.** Repeat [Problem P7.58](#) for output variable *G*.

**P7.60.** Repeat [Problem P7.58](#) for output variable *H*.

**P7.61.** Repeat [Problem P7.58](#) for output variable *I*.

**P7.62.** We need a logic circuit that gives an output *X* that is high only if a given hexadecimal digit is even (including 0) and less than 7. The inputs to the logic circuit are the bits  $B_8$ ,  $B_4$ ,  $B_2$ , and  $B_1$  of the binary equivalent for the hexadecimal digit. (The MSB is  $B_8$ , and the LSB is  $B_1$ .) Construct a truth table and the Karnaugh map; then, write the minimized SOP expression for *X*.

**P7.63.** We need a logic circuit that gives an output *X* that is high when an error in the form of an unused code occurs in a given BCD codeword. The inputs to the logic circuit are the bits  $B_8$ ,  $B_4$ ,  $B_2$ , and  $B_1$  of the BCD codeword. (The MSB is  $B_8$ , and the LSB is  $B_1$ .) Construct the Karnaugh map and write the minimized SOP and POS expressions for *X*.

**P7.64.** We need a logic circuit that gives a high output if a given hexadecimal digit is 4, 6, C, or E. The inputs to the logic circuit are the bits  $B_8$ ,  $B_4$ ,  $B_2$ , and  $B_1$  of the binary equivalent for the hexadecimal digit. (The MSB is  $B_8$ , and the LSB is  $B_1$ .) Construct the Karnaugh map and write the minimized SOP and POS expressions for *X*.

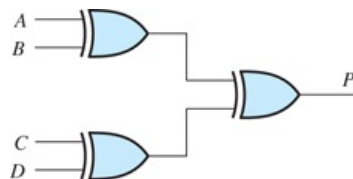
**P7.65.** We need to design a logic circuit for interchanging two logic signals. The system has three inputs  $I_1$ ,  $I_2$ , and *S* as well as two outputs  $O_1$  and  $O_2$ . When *S* is low, we should have  $O_1 = I_1$  and  $O_2 = I_2$ . On the other hand, when *S* is high, we should have  $O_1 = I_2$  and  $O_2 = I_1$ . Thus, *S* acts as the control input for a reversing switch. Use Karnaugh maps to obtain a minimal SOP design. Draw the circuit.

**P7.66.** A city council has three members,  $A$ ,  $B$ , and  $C$ . Each member votes on a proposition (1 for yes, 0 for no). Find a minimized SOP logic expression having inputs  $A$ ,  $B$ , and  $C$  and output  $X$  that is high when the majority vote is yes and low otherwise. Show that the minimized logic circuit checks to see if any pair of the three board members have voted yes. Repeat for a council with five members. [Hint: In this case, the circuit checks to see if any group of three has all voted yes.]

**P7.67.** A city council has four members,  $A$ ,  $B$ ,  $C$ , and  $D$ . Each member votes on a proposition (1 for yes, 0 for no). Find a minimized SOP logic expression having inputs  $A$ ,  $B$ ,  $C$ , and  $D$  and output  $X$  that is high when the vote is tied and low otherwise.

**P7.68.** One way to help ensure that data are communicated correctly is to append a parity bit to each data word such that the number of 1s in the transmitted word is even. Then, if an odd number are found in the received result, we know that at least one error has occurred.

- a. Show that the circuit in [Figure P7.68](#) produces the correct parity bit  $P$  for the nibble (four-bit data word)  $ABCD$ . In other words, show that the transmitted word  $ABCDP$  contains an even number of 1s for all combinations of data.



**Figure P7.68**

- b. Determine the minimum SOP expression for  $P$  in terms of the data bits.
- c. If the received word contains a single bit error, the number of ones in the word will be odd. Draw a circuit using four XOR gates that outputs a 1 if the received word  $ABCDP$  contains an odd number of 1s and outputs a 0 otherwise.

**P7.69.** Suppose we want circuits to convert the binary codes into the three-bit Gray codes shown in [Table P7.69](#). Find the minimum SOP expressions for  $X$ ,  $Y$ , and  $Z$  in terms of  $A$ ,  $B$ , and  $C$ .


**Table P7.69**

Binary Code $ABC$	Gray Code $XYZ$
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

**P7.70.** Find the minimum SOP expressions for  $A$ ,  $B$ , and  $C$  in terms of  $X$ ,  $Y$ , and  $Z$  for the codes of [Table P7.69](#).

**\*P7.71.** We have discussed BCD numbers in which the bits have weights of 8, 4, 2, and 1. Another way to represent decimal integers is the 4221 code in which the weights of the bits are 4, 2, 2, and


1. The decimal integers, the BCD equivalents, and the 4221 equivalents are shown in **Table**

**P7.71** . We want to design logic circuits to convert BCD codewords to 4221 codewords.

- Fill in the Karnaugh map for  $F$ , placing  $x$ 's (don't cares) in the squares for BCD codes that do not occur in the table. Find the minimum SOP expression allowing the various  $x$ 's to be either 1s or 0s to make the expression as simple as possible.
- Repeat (a) for  $G$ .
- Repeat (a) for  $H$ .
- Repeat (a) for  $I$ .

**Table P7.71 BCD, 4221, and excess-3 codewords for the decimal integers.**

Decimal Integer	BCD Codeword <i>ABCD</i>	4221 Codeword <i>FGHI</i>	Excess-3 Codeword <i>WXYZ</i>	
0	0000	0000	0011	
1	0001	0001	0100	
2	0010	0010	0101	
3	0011	0011	0110	
4	0100	1000	0111	
5	0101	0111	1000	
6	0110	1100	1001	
7	0111	1101	1010	
8	1000	1110	1011	
9	1001	1111	1100	

**P7.72.** We want to design logic circuits to convert the 4221 codewords of **Problem P7.71**  to BCD codewords.

- Fill in the Karnaugh map for  $A$ , placing  $x$ 's (don't cares) in the squares for 4221 codes that do not occur in the table. Find the minimum SOP expression allowing the various  $x$ 's to be either 1s or 0s to make the expression as simple as possible.
- Repeat (a) for  $B$ .
- Repeat (a) for  $C$ .
- Repeat (a) for  $D$ .

**P7.73.** Another code that is sometimes used to represent decimal digits is the excess-3 code. To convert a decimal digit to excess-3, we add 3 to the digit and express the sum as a four-bit binary number. For example, to convert the decimal digit 9 to excess-3 code, we have

$$9_{10} + 3_{10} = 12_{10} = 1100_2$$

Thus, 1100 is the excess-3 codeword for 9. The excess-3 codewords for the other decimal digits are shown in Table P7.71.

We want to design logic circuits to convert BCD codewords to excess-3 codewords.

- Fill in the Karnaugh map for  $W$ , placing  $x$ 's (don't cares) in the squares for BCD codes that do not occur in the table. Find the minimum SOP expression allowing the various  $x$ 's to be either 1s or 0s to make the expression as simple as possible.

- b. Repeat (a) for  $X$ .
- c. Repeat (a) for  $Y$ .
- d. Repeat (a) for  $Z$ .

**P7.74.** We want to design logic circuits to convert the excess-3 codewords of [Problem P7.73](#) to BCD codewords.

- a. Fill in the Karnaugh map for  $A$ , placing  $x$ 's (don't cares) in the squares for excess-3 codes that do not occur in the table. Find the minimum SOP expression allowing the various  $x$ 's to be either 1s or 0s to make the expression as simple as possible.
- b. Repeat (a) for  $B$ .
- c. Repeat (a) for  $C$ .
- d. Repeat (a) for  $D$ .

## Section 7.6: Sequential Logic Circuits

**P7.75.** Use NOR gates to draw the diagram of an  $SR$  flip-flop. Repeat using NAND gates.

**P7.76.** Draw the circuit symbol and give the truth table for an  $SR$  flip-flop.

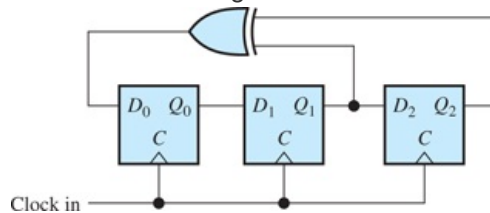
**P7.77.** Draw the circuit symbol and give the truth table for a clocked  $SR$  flip-flop.

**P7.78.** Explain the distinction between synchronous and asynchronous inputs to a flip-flop.

**P7.79.** What is edge triggering?

**P7.80.** Draw the circuit symbol and give the truth table for a positive-edge-triggered  $D$  flip-flop.

**\*P7.81.** Assuming that the initial state of the shift register shown in [Figure P7.81](#) is 100 (i.e.,  $Q_0 = 1$ ,  $Q_1 = 0$ , and  $Q_2 = 0$ ), find the successive states. After how many shifts does the register return to the starting state?

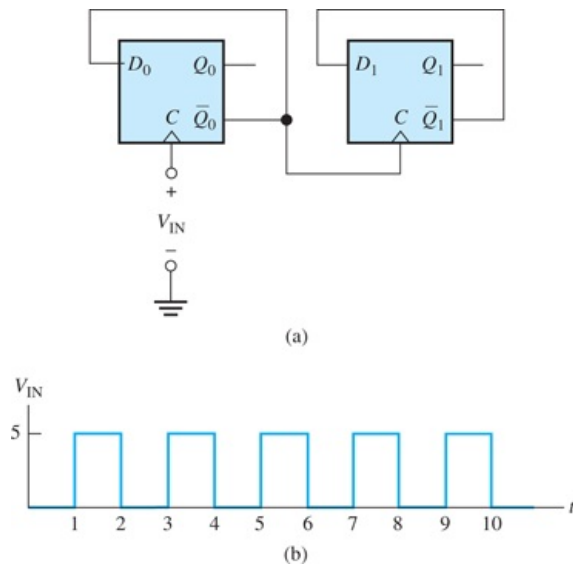


**Figure P7.81**

**P7.82.** Repeat [Problem P7.81](#) if the XOR gate is replaced with

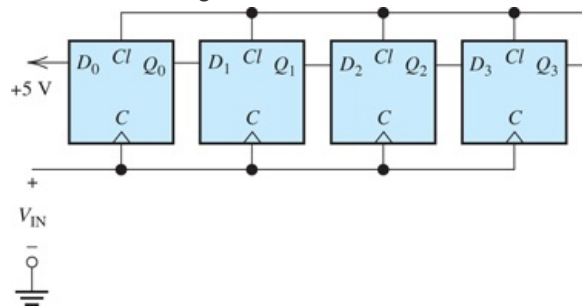
- a. an OR gate;
- b. an AND gate.

**P7.83.** The  $D$  flip-flops of [Figure P7.83](#) are positive-edge triggered. Assuming that prior to  $t = 0$ , the states are  $Q_0 = Q_1 = 0$ , sketch the voltage waveforms at  $Q_0$  and  $Q_1$  versus time. Assume logic levels of 0 V and 5 V.



**Figure P7.83**

**P7.84.** The  $D$  flip-flops of **Figure P7.84** are positive-edge triggered, and the  $Cl$  input is an asynchronous clear. Assume that the states are  $Q_0 = Q_1 = Q_2 = Q_3 = 0$  at  $t = 0$ . The clock input  $V_{IN}$  is shown in **Figure P7.83**. Sketch the voltage waveforms at  $Q_0$ ,  $Q_1$ ,  $Q_2$ , and  $Q_3$  versus time. Assume logic levels of 0 V and 5 V.



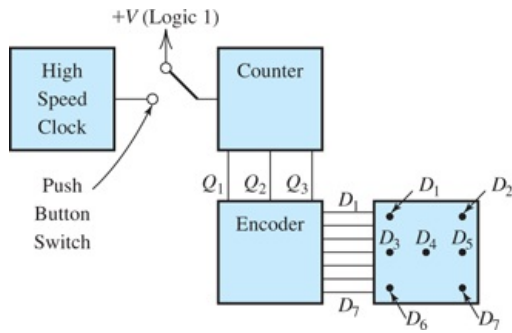
**Figure P7.84**

**\*P7.85.** Use AND gates, OR gates, inverters, and a negative-edge-triggered  $D$  flip-flop to show how to construct the  $JK$  flip-flop of **Figure 7.50** on page 391. Use AND gates, OR gates, inverters, and a negative-edge-triggered  $D$  flip-flop to show how to construct the  $JK$  flip-flop of **Figure 50**.

**P7.86.** Consider the ripple counter of **Figure 7.53** on page 393. Suppose that the flip-flops have asynchronous clear inputs. Show how to add gates so that the count resets to zero immediately when the count reaches six. This results in a modulo-six counter.

**P7.87.** **Figure P7.87** shows the functional diagram of an electronic die that can be used in games of chance. The system contains a high-speed clock, a push-button momentary contact switch that returns to the upper (logic 1) position when released, and a counter that counts through the cycle of states: 001, 010, 011, 100, 101, 110 (i.e., the binary equivalents of the number of spots on the various sides of the die).  $Q_3$  is the MSB, and  $Q_1$  is the LSB. The system has a display consisting of seven light-emitting diodes (LED), each of which lights when logic 1 is applied to it. The encoder is a combinatorial logic circuit that translates the state of the counter into the logic signals needed by the display. Each time the switch is depressed, the counter operates, stopping in a random state when the switch is released.

- Use  $JK$  flip-flops having asynchronous preset and clear inputs to draw the detailed diagram of the counter.
- Design the encoder, using Karnaugh maps to minimize the logic elements needed to produce each of the seven output signals.

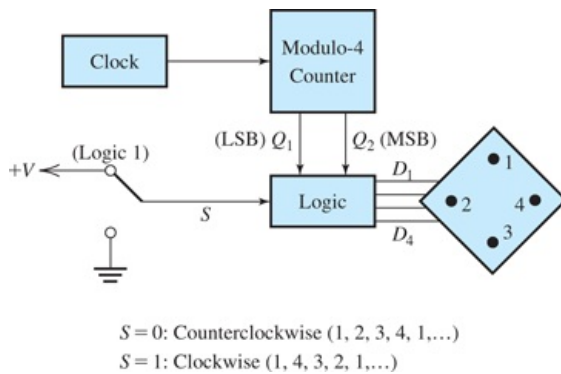


**Figure P7.87**

Electronic die.

**P7.88.** Four LED are arranged at the corners of a diamond, as illustrated in [Figure P7.88](#). When logic 1 is applied to an LED, it lights. Only one diode is to be on at a time. The on state should move from diode to diode either clockwise or counterclockwise, depending on whether  $S$  is high or low, respectively. One complete revolution should be completed in each two-second interval.

- What is the frequency of the clock?
- Draw a suitable logic circuit for the counter.
- Construct the truth table and use Karnaugh maps to determine the minimum SOP expressions for  $D_1$  through  $D_4$  in terms of  $S$ ,  $Q_1$ , and  $Q_2$ .



**Figure P7.88**

## Practice Test

Here is a practice test you can use to check your comprehension of the most important concepts in this chapter. Answers can be found in [Appendix D](#) and complete solutions are included in the Student Solutions files. See [Appendix E](#) for more information about the Student Solutions.

**T7.1.** First, think of one or more correct ways to complete each statement in [Table T7.1\(a\)](#). Then, select the best choice from the list given in [Table T7.1\(b\)](#). [Items in [Table T7.1\(b\)](#) may be used more than once or not at all.]

**Table T7.1**

Item	Best Match
<b>(a)</b>	
<ul style="list-style-type: none"> <li>a. The truth table for a logic expression contains . . .</li> <li>b. De Morgan's laws imply that . . .</li> <li>c. If the higher voltage level represents logic 1 and the lower level represents logic 0, we have . . .</li> <li>d. For a Gray code . . .</li> <li>e. If we invert each bit of a binary number and then add 1 to the result, we have . . .</li> <li>f. If we add two negative signed two's complement numbers, and the left-hand bit of the result is zero, we have had . . .</li> <li>g. Squares on the top and bottom of a Karnaugh map are considered to be . . .</li> <li>h. If the <math>Q</math> output of a positive-edge-triggered D flip-flop is connected to the D input, at each positive clock edge, the state of the flip-flop . . .</li> <li>i. Provided that it is not too large, noise can be completely eliminated from . . .</li> <li>j. Registers are composed of . . .</li> <li>k. An SOP logic circuit is composed of . . .</li> <li>l. Working out from the decimal point and converting four-bit groups of a BCD number to their hexadecimal equivalents produces . . .</li> </ul>	
<b>(b)</b>	
<ul style="list-style-type: none"> <li>1. the decimal equivalent</li> <li>2. old recordings of music</li> <li>3. inverse logic</li> <li>4. congruent</li> <li>5. OR gates</li> <li>6. AND gates and one OR gate</li> <li>7. digital signals</li> <li>8. inverters, AND gates, and one OR gate</li> <li>9. inverses</li> <li>10. flip-flops</li> <li>11. overflow</li> <li>12. a listing of all combinations of inputs and the corresponding outputs</li> <li>13. a table of ones and zeros</li> </ul>	



14. code words appear in numerical order
15. analog signals
16. adjacent
17. each code word is rotated to form the next word
18. if AND operations are changed to OR and vice versa, and the result is inverted, the result is equivalent to the original logic expression
19. NAND gates are sufficient to implement any logic expression
20. positive logic
21. the two's complement
22. negative logic
23. adjacent words differ in a single bit
24. underflow
25. toggles


**T7.2.** Convert the decimal integer,  $353.875_{10}$  to each of these forms:

- a. binary;
- b. octal;
- c. hexadecimal;
- d. BCD.

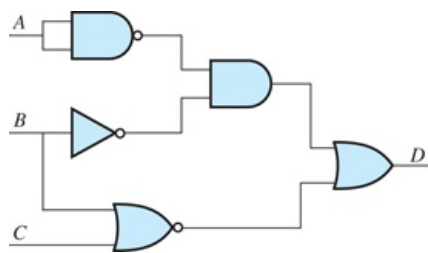
**T7.3.** Find the octal equivalent of  $FA.7_{16}$ .

**T7.4.** Find the decimal equivalent for each of these eight-bit signed two's complement integers:

- a. 01100001;
- b. 10111010.


**T7.5.** For the logic circuit of [Figure T7.5](#) :

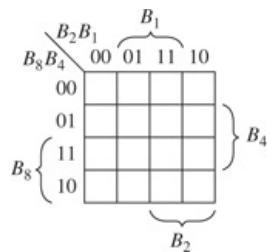
- a. write the logic expression for  $D$  in terms of  $A$ ,  $B$ , and  $C$  directly from the logic diagram;
- b. construct the truth table and the Karnaugh map;
- c. determine the minimum SOP expression for  $D$ ;
- d. determine the minimum POS expression for  $D$ .



**Figure T7.5**

**T7.6.** Suppose we need a logic circuit with a logic output  $G$  that is high only if a certain hexadecimal digit is 1, 5,  $B$ , or  $F$ . The inputs to the logic circuit are the bits  $B_8$ ,  $B_4$ ,  $B_2$ , and  $B_1$  of the binary equivalent for the hexadecimal digit. (The MSB is  $B_8$ , and the LSB is  $B_1$ .)

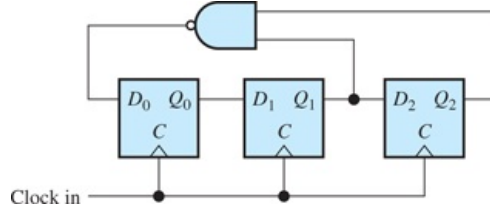
- a. Fill in the Karnaugh map shown in [Figure T7.6](#) .
- b. Determine the minimized SOP expression for  $G$ .
- c. Determine the minimum POS expression for  $G$ .



**Figure T7.6**

Karnaugh map to be filled in for G.

**T7.7.** Consider the shift register shown in [Figure T7.7](#). Assuming that the initial shift-register state is 100 (i.e.,  $Q_0 = 1$ ,  $Q_1 = 0$ , and  $Q_2 = 0$ ), list the next six states. After how many shifts does the register return to its initial state?

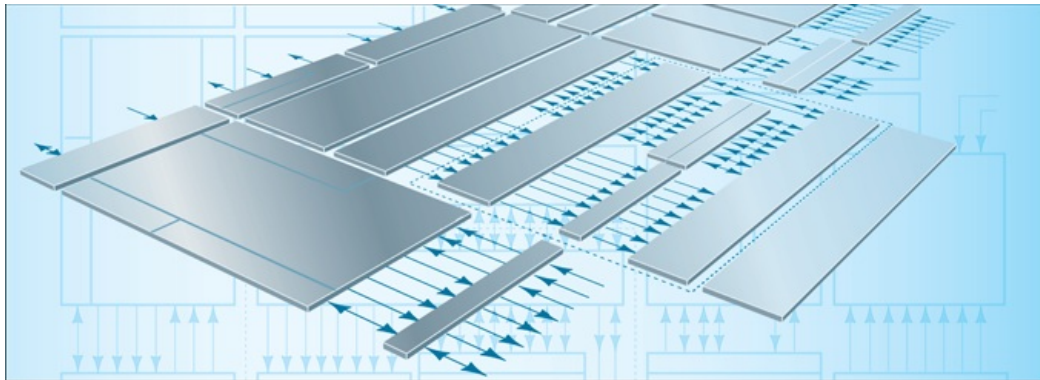


**Figure T7.7**

---

## Chapter 8 Computers, Microcontrollers and Computer-Based Instrumentation Systems

---



**Study of this chapter will enable you to:**

- Identify and describe the functional blocks of a computer.
- Define the terms *microprocessor*, *microcomputer*, and *microcontroller*.
- Select the type of memory needed for a given application.
- Understand how microcontrollers can be applied in your field of specialization.
- Identify the registers in the programmer's model and their functions for the HCS12/9S12 microcontroller family from Freescale Semiconductor, Inc.
- List some of the instructions and addressing modes of HCS12/9S12 microcontrollers.
- Write simple assembly language programs, using the CPU12 instruction set.
- Describe the operation of the elements of a computer-based instrumentation system.
- Identify the types of errors that may be encountered in instrumentation systems.
- Avoid common pitfalls such as ground loops, noise coupling, and loading when using sensors.
- Determine specifications for the elements of computer-based instrumentation systems such as data-acquisition boards.