CHAPTER 2

Number Systems, Operations, and Codes

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Review the decimal number system
- Count in the binary number system
- Convert from decimal to binary and from binary to decimal
- Apply arithmetic operations to binary numbers
- Determine the 1's and 2's complements of a binary number
- Express signed binary numbers in sign-magnitude, 1's complement, 2's complement, and floating-point format
- Carry out arithmetic operations with signed binary numbers
- Convert between the binary and hexadecimal number systems
- Add numbers in hexadecimal form
- Convert between the binary and octal number systems
- Express decimal numbers in binary coded decimal (BCD) form
- Add BCD numbers
- Convert between the binary system and the Gray code
- Interpret the American Standard Code for Information Interchange (ASCII)
- Explain how to detect code errors
- Discuss the cyclic redundancy check (CRC)

KEY TERMS

Key terms are in order of appearance in the chapter.

- LSB
- MSB
- Byte
- Floating-point number
- Hexadecimal
- Octal
- BCD
- Alphanumeric
- ASCII
- Parity
- Cyclic redundancy check (CRC)

VISIT THE WEBSITE

Study aids for this chapter are available at http://www.pearsonglobaleditions.com/floyd

INTRODUCTION

The binary number system and digital codes are fundamental to computers and to digital electronics in general. In this chapter, the binary number system and its relationship to other number systems such as decimal, hexadecimal, and octal are presented. Arithmetic operations with binary numbers are covered to provide a basis for understanding how computers and many other types of digital systems work. Also, digital codes such as binary coded decimal (BCD), the Gray code, and the ASCII are covered. The parity method for detecting errors in codes is introduced. The TI-36X calculator is used to illustrate certain operations. The procedures shown may vary on other types.

2–1 Decimal Numbers

You are familiar with the decimal number system because you use decimal numbers every day. Although decimal numbers are commonplace, their weighted structure is often not understood. In this section, the structure of decimal numbers is reviewed. This review will help you more easily understand the structure of the binary number system, which is important in computers and digital electronics.

After completing this section, you should be able to

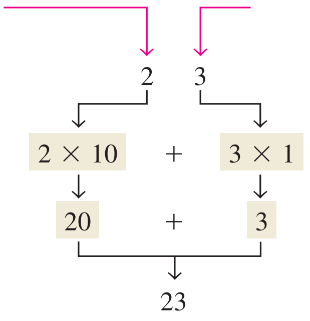
- Explain why the decimal number system is a weighted system
- Explain how powers of ten are used in the decimal system
- Determine the weight of each digit in a decimal number

The decimal number system has ten digits.

In the **decimal** number system each of the ten digits, 0 through 9, represents a certain quantity. As you know, the ten symbols (**digits**) do not limit you to expressing only ten different quantities because you use the various digits in appropriate positions within a number to indicate the magnitude of the quantity. You can express quantities up through nine before running out of digits; if you wish to express a quantity greater than nine, you use two or more digits, and the position of each digit within the number tells you the magnitude it represents. If, for example, you wish to express the quantity twenty-three, you use (by their respective positions in the number) the digit 2 to represent the quantity twenty and the digit 3 to represent the quantity three, as illustrated below.

The digit 2 has a weight of 10 in this position.

The digit 3 has a weight of 1 in this position.



The decimal number system has a base of 10.

The position of each digit in a decimal number indicates the magnitude of the quantity represented and can be assigned a **weight**. The weights for whole numbers are positive powers of ten that increase from right to left, beginning with $10^0 = 1$.

$$\ldots\ 10^5\ 10^4\ 10^3\ 10^2\ 10^1\ 10^0$$

For fractional numbers, the weights are negative powers of ten that decrease from left to right beginning with 10^{-1}.

$$10^2\ 10^1\ 10^0.10^{-1}\ 10^{-2}\ 10^{-3}\ldots$$

Decimal point

The value of a digit is determined by its position in the number.

The value of a decimal number is the sum of the digits after each digit has been multiplied by its weight, as Examples 2–1 and 2–2 illustrate.

EXAMPLE 2–1

Express the decimal number 47 as a sum of the values of each digit.

Solution

The digit 4 has a weight of 10, which is 10^1, as indicated by its position. The digit 7 has a weight of 1, which is 10^0, as indicated by its position.

$$\begin{aligned} 47 &= (4 \times 10^1) + (7 \times 10^0) \\ &= (4 \times 10) + (7 \times 1) = \mathbf{40} + \mathbf{7} \end{aligned}$$

Related Problem*

Determine the value of each digit in 939.

*Answers are at the end of the chapter.

EXAMPLE 2–2

Express the decimal number 568.23 as a sum of the values of each digit.

Solution

The whole number digit 5 has a weight of 100, which is 10^2, the digit 6 has a weight of 10, which is 10^1, the digit 8 has a weight of 1, which is 10^0, the fractional digit 2 has a weight of 0.1, which is 10^{-1}, and the fractional digit 3 has a weight of 0.01, which is 10^{-2}.

$$\begin{aligned} 568.23 &= (5 \times 10^2) + (6 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) + (3 \times 10^{-2}) \\ &= (5 \times 100) + (6 \times 10) + (8 \times 1) + (2 \times 0.1) + (3 \times 0.01) \\ &= \mathbf{500} + \mathbf{60} + \mathbf{8} + \mathbf{0.2} + \mathbf{0.03} \end{aligned}$$

Related Problem

Determine the value of each digit in 67.924.

CALCULATOR SESSION

Powers of Ten

Find the value of 10^3.

TI-36X **Step 1:** 1 0 y^x

Step 2: 3 =

1000

SECTION 2–1 CHECKUP

Answers are at the end of the chapter.

1. What weight does the digit 7 have in each of the following numbers?

   (a) 1370 **(b)** 6725 **(c)** 7051 **(d)** 58.72

2. Express each of the following decimal numbers as a sum of the products obtained by multiplying each digit by its appropriate weight:

   (a) 51 **(b)** 137 **(c)** 1492 **(d)** 106.58

2–2 Binary Numbers

The binary number system is another way to represent quantities. It is less complicated than the decimal system because the binary system has only two digits. The decimal system with its ten digits is a base-ten system; the binary system with its two digits is a base-two system. The two binary digits (bits) are 1 and 0. The position of a 1 or 0 in a binary number indicates its weight, or value within the number, just as the position of a decimal digit determines the value of that digit. The weights in a binary number are based on powers of two.

After completing this section, you should be able to

- Count in binary
- Determine the largest decimal number that can be represented by a given number of bits
- Convert a binary number to a decimal number

Counting in Binary

The binary number system has two digits (bits).

To learn to count in the binary system, first look at how you count in the decimal system. You start at zero and count up to nine before you run out of digits. You then start another digit position (to the left) and continue counting 10 through 99. At this point you have exhausted all two-digit combinations, so a third digit position is needed to count from 100 through 999.

The binary number system has a base of 2.

A comparable situation occurs when you count in binary, except that you have only two digits, called *bits*. Begin counting: 0, 1. At this point you have used both digits, so include another digit position and continue: 10, 11. You have now exhausted all combinations of two digits, so a third position is required. With three digit positions you can continue to count: 100, 101, 110, and 111. Now you need a fourth digit position to continue, and so on. A binary count of zero through fifteen is shown in Table 2–1. Notice the patterns with which the 1s and 0s alternate in each column.

InfoNote

In processor operations, there are many cases where adding or subtracting 1 to a number stored in a counter is necessary. Processors have special instructions that use less time and generate less machine code than the ADD or SUB instructions. For the Intel processors, the INC (increment) instruction adds 1 to a number. For subtraction, the corresponding instruction is DEC (decrement), which subtracts 1 from a number.

TABLE 2–1

Decimal Number	Binary Number			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

The value of a bit is determined by its position in the number.

As you have seen in Table 2–1, four bits are required to count from zero to 15. In general, with n bits you can count up to a number equal to $2^n - 1$.

$$\text{Largest decimal number} = 2^n - 1$$

For example, with five bits ($n = 5$) you can count from zero to thirty-one.

$$2^5 - 1 = 32 - 1 = 31$$

With six bits ($n = 6$) you can count from zero to sixty-three.

$$2^6 - 1 = 64 - 1 = 63$$

CALCULATOR SESSION

Powers of Two

Find the value of 2^5.

TI-36X **Step 1:** 2 y^x

Step 2: 5 =

32

An Application

Learning to count in binary will help you to basically understand how digital circuits can be used to count events. Let's take a simple example of counting tennis balls going into a box from a conveyor belt. Assume that nine balls are to go into each box.

The counter in Figure 2–1 counts the pulses from a sensor that detects the passing of a ball and produces a sequence of logic levels (digital waveforms) on each of its four parallel outputs. Each set of logic levels represents a 4-bit binary number (HIGH = 1 and LOW = 0), as indicated. As the decoder receives these waveforms, it decodes each set of four bits and converts it to the corresponding decimal number in the 7-segment display. When the counter gets to the binary state of 1001, it has counted nine tennis balls, the display shows decimal 9, and a new box is moved under the conveyor belt. Then the counter goes back to its zero state (0000), and the process starts over. (The number 9 was used only in the interest of single-digit simplicity.)

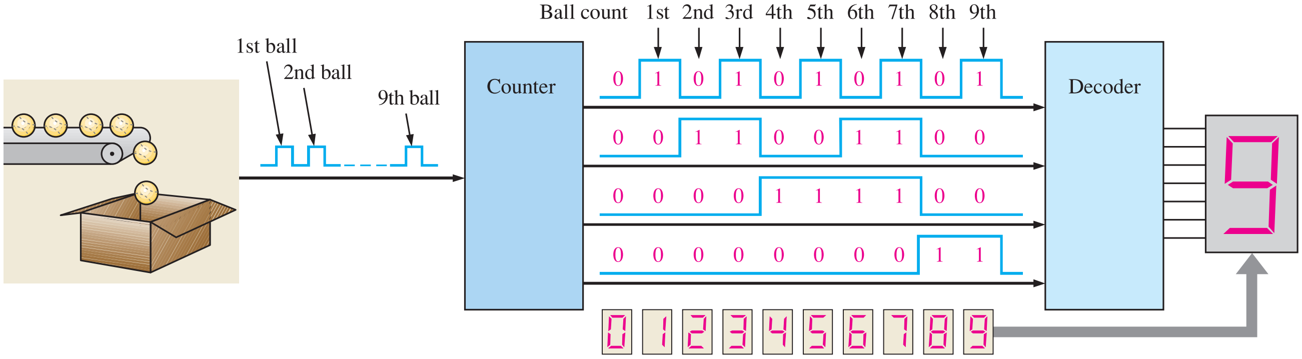


FIGURE 2–1 Illustration of a simple binary counting application.

The Weighting Structure of Binary Numbers

A binary number is a weighted number. The right-most bit is the **LSB** (least significant bit) in a binary whole number and has a weight of $2^0 = 1$. The weights increase from right to left by a power of two for each bit. The left-most bit is the **MSB** (most significant bit); its weight depends on the size of the binary number.

The weight or value of a bit increases from right to left in a binary number.

Fractional numbers can also be represented in binary by placing bits to the right of the binary point, just as fractional decimal digits are placed to the right of the decimal point. The left-most bit is the MSB in a binary fractional number and has a weight of $2^{-1} = 0.5$. The fractional weights decrease from left to right by a negative power of two for each bit.

The weight structure of a binary number is

$$2^{n-1} \ldots 2^3\, 2^2\, 2^1\, 2^0\, .\, 2^{-1}\, 2^{-2} \ldots 2^{-n}$$

Binary point

where n is the number of bits from the binary point. Thus, all the bits to the left of the binary point have weights that are positive powers of two, as previously discussed for whole numbers. All bits to the right of the binary point have weights that are negative powers of two, or fractional weights.

The powers of two and their equivalent decimal weights for an 8-bit binary whole number and a 6-bit binary fractional number are shown in Table 2–2. Notice that the weight doubles for each positive power of two and that the weight is halved for each negative power of two. You can easily extend the table by doubling the weight of the most significant positive power of two and halving the weight of the least significant negative power of two; for example, $2^9 = 512$ and $2^{-7} = 0.0078125$.

InfoNote

Processors use binary numbers to select memory locations. Each location is assigned a unique number called an *address*. Some microprocessors, for example, have 32 address lines which can select 2^{32} (4,294,967,296) unique locations.

TABLE 2–2

Binary weights.

Positive Powers of Two (Whole Numbers)									Negative Powers of Two (Fractional Number)					
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
256	128	64	32	16	8	4	2	1	1/2 0.5	1/4 0.25	1/8 0.125	1/16 0.625	1/32 0.03125	1/64 0.015625

Binary-to-Decimal Conversion

Add the weights of all 1s in a binary number to get the decimal value.

The decimal value of any binary number can be found by adding the weights of all bits that are 1 and discarding the weights of all bits that are 0.

EXAMPLE 2–3

Convert the binary whole number 1101101 to decimal.

Solution

Determine the weight of each bit that is a 1, and then find the sum of the weights to get the decimal number.

$$\begin{array}{rl} \text{Weight:} & 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0 \\ \text{Binary number:} & 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1 \end{array}$$

$$\begin{aligned} 1101101 &= 2^6 + 2^5 + 2^3 + 2^2 + 2^0 \\ &= 64 + 32 + 8 + 4 + 1 = \mathbf{109} \end{aligned}$$

Related Problem

Convert the binary number 10010001 to decimal.

EXAMPLE 2–4

Convert the fractional binary number 0.1011 to decimal.

Solution

Determine the weight of each bit that is a 1, and then sum the weights to get the decimal fraction.

$$\begin{array}{rlllll} \text{Weight:} & & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\ \text{Binary number:} & 0\,. & 1 & 0 & 1 & 1 \end{array}$$

$$\begin{aligned} 0.1011 &= 2^{-1} + 2^{-3} + 2^{-4} \\ &= 0.5 + 0.125 + 0.0625 = \mathbf{0.6875} \end{aligned}$$

Related Problem

Convert the binary number 10.111 to decimal.

SECTION 2–2 CHECKUP

1. What is the largest decimal number that can be represented in binary with eight bits?
2. Determine the weight of the 1 in the binary number 10000.
3. Convert the binary number 10111101.011 to decimal.

2–3 Decimal-to-Binary Conversion

In Section 2–2 you learned how to convert a binary number to the equivalent decimal number. Now you will learn two ways of converting from a decimal number to a binary number.

After completing this section, you should be able to

- Convert a decimal number to binary using the sum-of-weights method
- Convert a decimal whole number to binary using the repeated division-by-2 method
- Convert a decimal fraction to binary using the repeated multiplication-by-2 method

Sum-of-Weights Method

One way to find the binary number that is equivalent to a given decimal number is to determine the set of binary weights whose sum is equal to the decimal number. An easy way to remember binary weights is that the lowest is 1, which is 2^0, and that by doubling any weight, you get the next higher weight; thus, a list of seven binary weights would be 64, 32, 16, 8, 4, 2, 1 as you learned in the last section. The decimal number 9, for example, can be expressed as the sum of binary weights as follows:

To get the binary number for a given decimal number, find the binary weights that add up to the decimal number.

$$9 = 8 + 1 \quad \text{or} \quad 9 = 2^3 + 2^0$$

Placing 1s in the appropriate weight positions, 2^3 and 2^0, and 0s in the 2^2 and 2^1 positions determines the binary number for decimal 9.

2^3	2^2	2^1	2^0	
1	0	0	1	Binary number for decimal 9

EXAMPLE 2–5

Convert the following decimal numbers to binary:

(a) 12 **(b)** 25
(c) 58 **(d)** 82

Solution

(a) $12 = 8 + 4 = 2^3 + 2^2 \longrightarrow$ **1100**

(b) $25 = 16 + 8 + 1 = 2^4 + 2^3 + 2^0 \longrightarrow$ **11001**

(c) $58 = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 \longrightarrow$ **111010**

(d) $82 = 64 + 16 + 2 = 2^6 + 2^4 + 2^1 \longrightarrow$ **1010010**

Related Problem

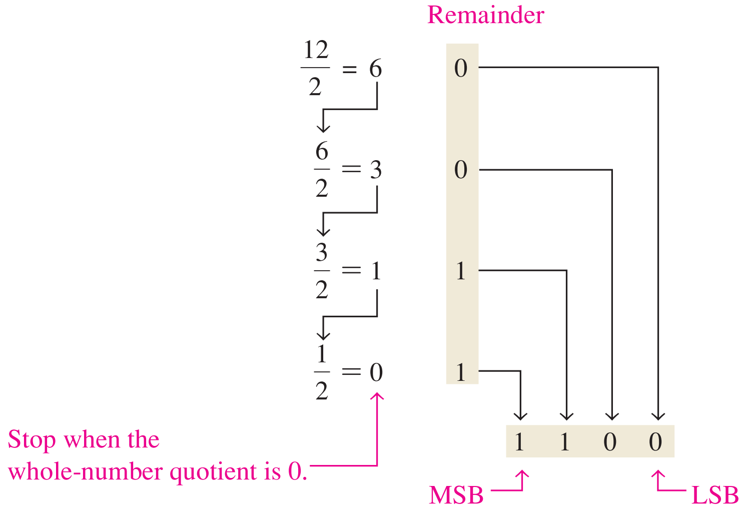
Convert the decimal number 125 to binary.

Repeated Division-by-2 Method

A systematic method of converting whole numbers from decimal to binary is the *repeated division-by-2* process. For example, to convert the decimal number 12 to binary, begin by dividing 12 by 2. Then divide each resulting quotient by 2 until there is a 0 whole-number quotient. The **remainders** generated by each division form the binary number. The first remainder to be produced is the LSB (least significant bit) in the binary number, and the

To get the binary number for a given decimal number, divide the decimal number by 2 until the quotient is 0. Remainders form the binary number.

last remainder to be produced is the MSB (most significant bit). This procedure is illustrated as follows for converting the decimal number 12 to binary.

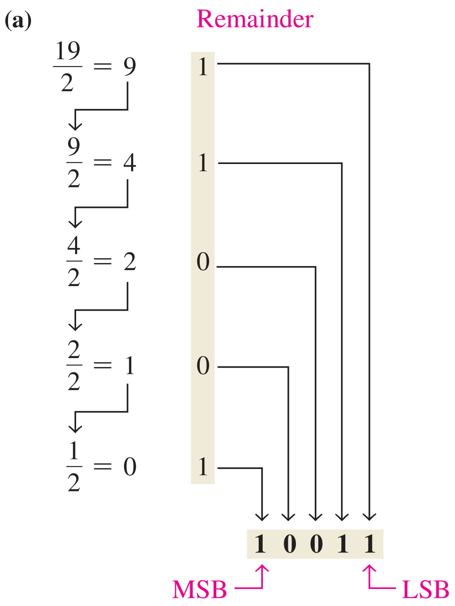


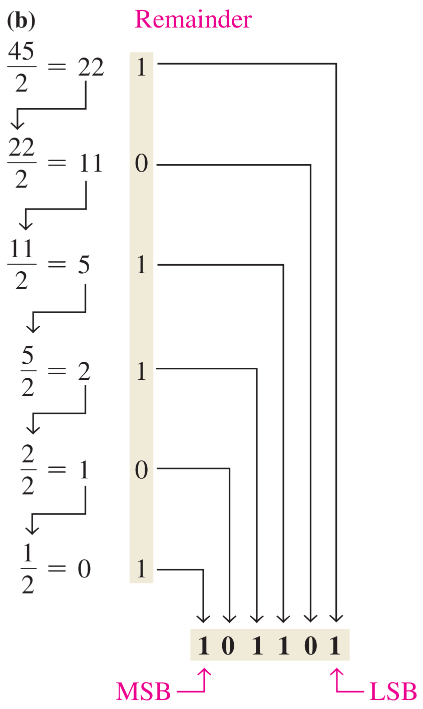
EXAMPLE 2–6

Convert the following decimal numbers to binary:

(a) 19 **(b)** 45

Solution





Related Problem

Convert decimal number 39 to binary.

CALCULATOR SESSION

Conversion of a Decimal Number to a Binary Number

Convert decimal 57 to binary.

DEC

TI-36X **Step 1:** 3rd EE

Step 2: 5 7

BIN

Step 3: 3rd X

111001

Converting Decimal Fractions to Binary

Examples 2–5 and 2–6 demonstrated whole-number conversions. Now let's look at fractional conversions. An easy way to remember fractional binary weights is that the most significant weight is 0.5, which is 2^{-1}, and that by halving any weight, you get the next lower weight; thus a list of four fractional binary weights would be 0.5, 0.25, 0.125, 0.0625.

Sum-of-Weights

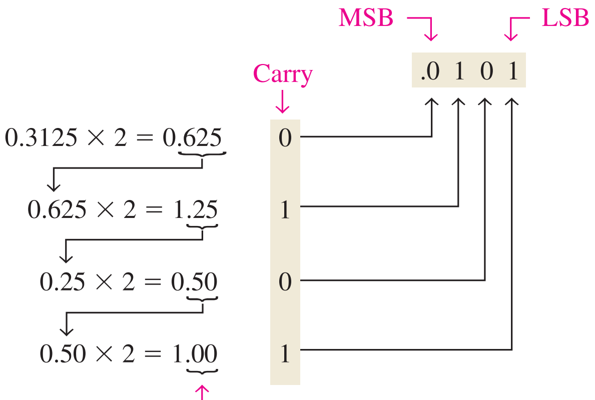
The sum-of-weights method can be applied to fractional decimal numbers, as shown in the following example:

$$0.625 = 0.5 + 0.125 = 2^{-1} + 2^{-3} = 0.101$$

There is a 1 in the 2^{-1} position, a 0 in the 2^{-2} position, and a 1 in the 2^{-3} position.

Repeated Multiplication by 2

As you have seen, decimal whole numbers can be converted to binary by repeated division by 2. Decimal fractions can be converted to binary by repeated multiplication by 2. For example, to convert the decimal fraction 0.3125 to binary, begin by multiplying 0.3125 by 2 and then multiplying each resulting fractional part of the product by 2 until the fractional product is zero or until the desired number of decimal places is reached. The carry digits, or **carries**, generated by the multiplications produce the binary number. The first carry produced is the MSB, and the last carry is the LSB. This procedure is illustrated as follows:



Continue to the desired number of decimal places
or stop when the fractional part is all zeros.

SECTION 2–3 CHECKUP

1. Convert each decimal number to binary by using the sum-of-weights method:

   (a) 23 **(b)** 57 **(c)** 45.5

2. Convert each decimal number to binary by using the repeated division-by-2 method (repeated multiplication-by-2 for fractions):

   (a) 14 **(b)** 21 **(c)** 0.375

2–4 Binary Arithmetic

Binary arithmetic is essential in all digital computers and in many other types of digital systems. To understand digital systems, you must know the basics of binary addition, subtraction, multiplication, and division. This section provides an introduction that will be expanded in later sections.

After completing this section, you should be able to

- Add binary numbers
- Subtract binary numbers
- Multiply binary numbers
- Divide binary numbers

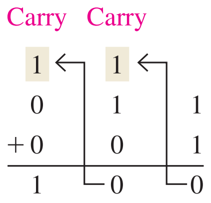
Binary Addition

In binary 1 + 1 = 10, not 2.

The four basic rules for adding binary digits (bits) are as follows:

$0 + 0 = 0$	Sum of 0 with a carry of 0
$0 + 1 = 1$	Sum of 1 with a carry of 0
$1 + 0 = 1$	Sum of 1 with a carry of 0
$1 + 1 = 10$	Sum of 0 with a carry of 1

Notice that the first three rules result in a single bit and in the fourth rule the addition of two 1s yields a binary two (10). When binary numbers are added, the last condition creates a sum of 0 in a given column and a carry of 1 over to the next column to the left, as illustrated in the following addition of 11 + 1:



In the right column, $1 + 1 = 0$ with a carry of 1 to the next column to the left. In the middle column, $1 + 1 + 0 = 0$ with a carry of 1 to the next column to the left. In the left column, $1 + 0 + 0 = 1$.

When there is a carry of 1, you have a situation in which three bits are being added (a bit in each of the two numbers and a carry bit). This situation is illustrated as follows:

Carry bits

1	$+ 0 + 0 = 01$	Sum of 1 with a carry of 0
1	$+ 1 + 0 = 10$	Sum of 0 with a carry of 1
1	$+ 0 + 1 = 10$	Sum of 0 with a carry of 1
1	$+ 1 + 1 = 11$	Sum of 1 with a carry of 1

EXAMPLE 2–7

Add the following binary numbers:

(a) 11 + 11 **(b)** 100 + 10

(c) 111 + 11 **(d)** 110 + 100

Solution

The equivalent decimal addition is also shown for reference.

(a) $$\begin{array}{r} 11 \\ +\,11 \\ \hline \mathbf{110} \end{array} \qquad \begin{array}{r} 3 \\ +\,3 \\ \hline 6 \end{array}$$

(b) $$\begin{array}{r} 100 \\ +\,10 \\ \hline \mathbf{110} \end{array} \qquad \begin{array}{r} 4 \\ +\,2 \\ \hline 6 \end{array}$$

(c) $$\begin{array}{r} 111 \\ +\,11 \\ \hline \mathbf{1010} \end{array} \qquad \begin{array}{r} 7 \\ +\,3 \\ \hline 10 \end{array}$$

(d) $$\begin{array}{r} 110 \\ +\,100 \\ \hline \mathbf{1010} \end{array} \qquad \begin{array}{r} 6 \\ +\,4 \\ \hline 10 \end{array}$$

Related Problem

Add 1111 and 1100.

Binary Subtraction

In binary 10 − 1 = 1, not 9.

The four basic rules for subtracting bits are as follows:

$$\begin{array}{rl} 0 - 0 = 0 & \\ 1 - 1 = 0 & \\ 1 - 0 = 1 & \\ 10 - 1 = 1 & \qquad 0 - 1 \text{ with a borrow of } 1 \end{array}$$

When subtracting numbers, you sometimes have to borrow from the next column to the left. A borrow is required in binary only when you try to subtract a 1 from a 0. In this case, when a 1 is borrowed from the next column to the left, a 10 is created in the column being subtracted, and the last of the four basic rules just listed must be applied. Examples 2–8 and 2–9 illustrate binary subtraction; the equivalent decimal subtractions are also shown.

EXAMPLE 2–8

Perform the following binary subtractions:

(a) 11 − 01 **(b)** 11 − 10

Solution

(a) $$\begin{array}{r} 11 \\ -01 \\ \hline \mathbf{10} \end{array} \qquad \begin{array}{r} 3 \\ -1 \\ \hline 2 \end{array}$$

(b) $$\begin{array}{r} 11 \\ -10 \\ \hline \mathbf{01} \end{array} \qquad \begin{array}{r} 3 \\ -2 \\ \hline 1 \end{array}$$

No borrows were required in this example. The binary number 01 is the same as 1.

Related Problem

Subtract 100 from 111.

EXAMPLE 2–9

Subtract 011 from 101.

Solution

$$\begin{array}{r} 101 \\ -011 \\ \hline \mathbf{010} \end{array} \qquad \begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array}$$

Let’s examine exactly what was done to subtract the two binary numbers since a borrow is required. Begin with the right column.

Left column:
When a 1 is borrowed,
a 0 is left, so 0 − 0 = 0.

Middle column:
Borrow 1 from next column
to the left, making a 10 in
this column, then 10 − 1 = 1.

$$\begin{array}{r} \overset{0}{\cancel{1}}{}^{1}0\,1 \\ -0\,1\,1 \\ \hline 0\,1\,0 \end{array}$$

Right column:
1 − 1 = 0

Related Problem

Subtract 101 from 110.

Binary Multiplication

Binary multiplication of two bits is the same as multiplication of the decimal digits 0 and 1.

The four basic rules for multiplying bits are as follows:

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

Multiplication is performed with binary numbers in the same manner as with decimal numbers. It involves forming partial products, shifting each successive partial product left one place, and then adding all the partial products. Example 2–10 illustrates the procedure; the equivalent decimal multiplications are shown for reference.

EXAMPLE 2–10

Perform the following binary multiplications:

(a) 11 × 11 **(b)** 101 × 111

Solution

(a)

$$\begin{array}{r} 11 \\ \times\ 11 \\ \hline \end{array}\qquad\qquad \begin{array}{r} 3 \\ \times\ 3 \\ \hline 9 \end{array}$$

$$\text{Partial products}\left\{\begin{array}{r} 11 \\ +11\ \ \\ \hline \mathbf{1001} \end{array}\right.$$

(b)

$$\begin{array}{r} 111 \\ \times\ 101 \\ \hline \end{array}\qquad\qquad \begin{array}{r} 7 \\ \times\ 5 \\ \hline 35 \end{array}$$

$$\text{Partial products}\left\{\begin{array}{r} 111 \\ 000\ \\ +111\ \ \ \\ \hline \mathbf{100011} \end{array}\right.$$

Related Problem

Multiply 1101 × 1010.

Binary Division

A calculator can be used to perform arithmetic operations with binary numbers as long as the capacity of the calculator is not exceeded.

Division in binary follows the same procedure as division in decimal, as Example 2–11 illustrates. The equivalent decimal divisions are also given.

EXAMPLE 2–11

Perform the following binary divisions:

(a) 110 ÷ 11 **(b)** 110 ÷ 10

Solution

$$\textbf{(a)}\quad \begin{array}{r} \mathbf{10} \\ 11\overline{)110} \\ \underline{11} \\ 000 \end{array} \qquad \begin{array}{r} 2 \\ 3\overline{)6} \\ \underline{6} \\ 0 \end{array} \qquad\qquad \textbf{(b)}\quad \begin{array}{r} \mathbf{11} \\ 10\overline{)110} \\ \underline{10} \\ 10 \\ \underline{10} \\ 00 \end{array} \qquad \begin{array}{r} 3 \\ 2\overline{)6} \\ \underline{6} \\ 0 \end{array}$$

Related Problem

Divide 1100 by 100.

SECTION 2–4 CHECKUP

1. Perform the following binary additions:

   (a) 1101 + 1010 **(b)** 10111 + 01101

2. Perform the following binary subtractions:

   (a) 1101 − 0100 **(b)** 1001 − 0111

3. Perform the indicated binary operations:

   (a) 110 × 111 **(b)** 1100 ÷ 011

2–5 Complements of Binary Numbers

The 1's complement and the 2's complement of a binary number are important because they permit the representation of negative numbers. The method of 2's complement arithmetic is commonly used in computers to handle negative numbers.

After completing this section, you should be able to

- Convert a binary number to its 1's complement
- Convert a binary number to its 2's complement using either of two methods

Finding the 1's Complement

The 1's **complement** of a binary number is found by changing all 1s to 0s and all 0s to 1s, as illustrated below:

Change each bit in a number to get the 1's complement.

```
1 0 1 1 0 0 1 0    Binary number
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
0 1 0 0 1 1 0 1    1's complement
```

The simplest way to obtain the 1's complement of a binary number with a digital circuit is to use parallel inverters (NOT circuits), as shown in Figure 2–2 for an 8-bit binary number.

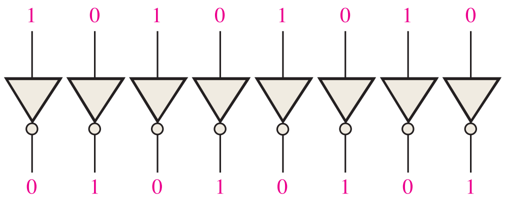


FIGURE 2–2 Example of inverters used to obtain the 1's complement of a binary number.

Finding the 2's Complement

Add 1 to the 1's complement to get the 2's complement.

The 2's complement of a binary number is found by adding 1 to the LSB of the 1's complement.

$$2\text{'s complement} = (1\text{'s complement}) + 1$$

EXAMPLE 2–12

Find the 2's complement of 10110010.

Solution

$$\begin{array}{rl}
10110010 & \text{Binary number} \\
01001101 & \text{1's complement} \\
+ \qquad\quad 1 & \text{Add 1} \\
\hline
\mathbf{01001110} & \text{2's complement}
\end{array}$$

Related Problem

Determine the 2's complement of 11001011.

Change all bits to the left of the least significant 1 to get 2's complement.

An alternative method of finding the 2's complement of a binary number is as follows:

1. Start at the right with the LSB and write the bits as they are up to and including the first 1.
2. Take the 1's complements of the remaining bits.

EXAMPLE 2–13

Find the 2's complement of 10111000 using the alternative method.

Solution

$$\begin{array}{ll}
10111000 & \text{Binary number} \\
\underbrace{\mathbf{0100}}_{\text{1's complements of original bits}}\underbrace{\mathbf{1000}}_{\text{These bits stay the same.}} & \text{2's complement}
\end{array}$$

Related Problem

Find the 2's complement of 11000000.

The 2's complement of a negative binary number can be realized using inverters and an adder, as indicated in Figure 2–3. This illustrates how an 8-bit number can be converted to its 2's complement by first inverting each bit (taking the 1's complement) and then adding 1 to the 1's complement with an adder circuit.

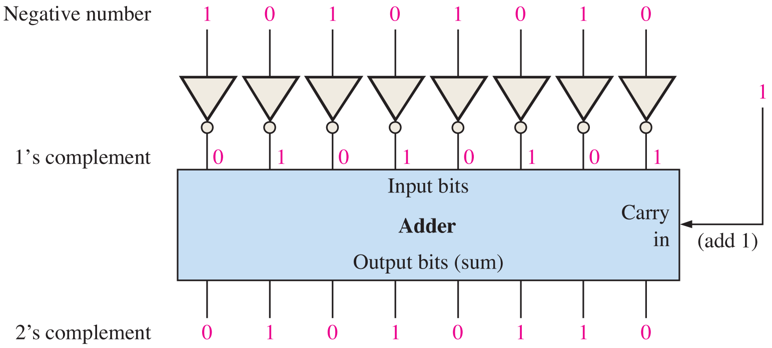


FIGURE 2–3 Example of obtaining the 2's complement of a negative binary number.

To convert from a 1's or 2's complement back to the true (uncomplemented) binary form, use the same two procedures described previously. To go from the 1's complement back to true binary, reverse all the bits. To go from the 2's complement form back to true binary, take the 1's complement of the 2's complement number and add 1 to the least significant bit.

SECTION 2–5 CHECKUP

1. Determine the 1's complement of each binary number:

(a) 00011010 **(b)** 11110111 **(c)** 10001101

2. Determine the 2's complement of each binary number:

(a) 00010110 **(b)** 11111100 **(c)** 10010001

2–6 Signed Numbers

Digital systems, such as the computer, must be able to handle both positive and negative numbers. A signed binary number consists of both sign and magnitude information. The sign indicates whether a number is positive or negative, and the magnitude is the value of the number. There are three forms in which signed integer (whole) numbers can be represented in binary: sign-magnitude, 1's complement, and 2's complement. Of these, the 2's complement is the most important and the sign-magnitude is the least used. Noninteger and very large or small numbers can be expressed in floating-point format.

After completing this section, you should be able to

- Express positive and negative numbers in sign-magnitude
- Express positive and negative numbers in 1's complement
- Express positive and negative numbers in 2's complement
- Determine the decimal value of signed binary numbers
- Express a binary number in floating-point format

The Sign Bit

The left-most bit in a signed binary number is the **sign bit**, which tells you whether the number is positive or negative.

A 0 sign bit indicates a positive number, and a 1 sign bit indicates a negative number.

Sign-Magnitude Form

When a signed binary number is represented in sign-magnitude, the left-most bit is the sign bit and the remaining bits are the magnitude bits. The magnitude bits are in true (uncomplemented) binary for both positive and negative numbers. For example, the decimal number +25 is expressed as an 8-bit signed binary number using the sign-magnitude form as

$$\underset{\text{Sign bit}}{0}\underbrace{0011001}_{\text{Magnitude bits}}$$

The decimal number −25 is expressed as

$$10011001$$

Notice that the only difference between +25 and −25 is the sign bit because the magnitude bits are in true binary for both positive and negative numbers.

In the sign-magnitude form, a negative number has the same magnitude bits as the corresponding positive number but the sign bit is a 1 rather than a zero.

InfoNote

Processors use the 2's complement for negative integer numbers in arithmetic operations. The reason is that subtraction of a number is the same as adding the 2's complement of the number. Processors form the 2's complement by inverting the bits and adding 1, using special instructions that produce the same result as the adder in Figure 2–3.

1's Complement Form

Positive numbers in 1's complement form are represented the same way as the positive sign-magnitude numbers. Negative numbers, however, are the 1's complements of the corresponding positive numbers. For example, using eight bits, the decimal number −25 is expressed as the 1's complement of +25 (00011001) as

11100110

In the 1's complement form, a negative number is the 1's complement of the corresponding positive number.

2's Complement Form

Positive numbers in 2's complement form are represented the same way as in the sign-magnitude and 1's complement forms. Negative numbers are the 2's complements of the corresponding positive numbers. Again, using eight bits, let's take decimal number −25 and express it as the 2's complement of +25 (00011001). Inverting each bit and adding 1, you get

$$-25 = 11100111$$

In the 2's complement form, a negative number is the 2's complement of the corresponding positive number.

EXAMPLE 2–14

Express the decimal number −39 as an 8-bit number in the sign-magnitude, 1's complement, and 2's complement forms.

Solution

First, write the 8-bit number for +39.

00100111

In the *sign-magnitude form,* −39 is produced by changing the sign bit to a 1 and leaving the magnitude bits as they are. The number is

10100111

In the *1's complement form,* −39 is produced by taking the 1's complement of +39 (00100111).

11011000

In the *2's complement form,* −39 is produced by taking the 2's complement of +39 (00100111) as follows:

$$\begin{array}{rl} 11011000 & \text{1's complement} \\ + \quad\quad 1 & \\ \hline \mathbf{11011001} & \text{2's complement} \end{array}$$

Related Problem

Express +19 and −19 as 8-bit numbers in sign-magnitude, 1's complement, and 2's complement.

The Decimal Value of Signed Numbers

Sign-Magnitude

Decimal values of positive and negative numbers in the sign-magnitude form are determined by summing the weights in all the magnitude bit positions where there are 1s and ignoring those positions where there are zeros. The sign is determined by examination of the sign bit.

EXAMPLE 2–15

Determine the decimal value of this signed binary number expressed in sign-magnitude: 10010101.

Solution

The seven magnitude bits and their powers-of-two weights are as follows:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	0	1

Summing the weights where there are 1s,

$$16 + 4 + 1 = 21$$

The sign bit is 1; therefore, the decimal number is **−21**.

Related Problem

Determine the decimal value of the sign-magnitude number 01110111.

1's Complement

Decimal values of positive numbers in the 1's complement form are determined by summing the weights in all bit positions where there are 1s and ignoring those positions where there are zeros. Decimal values of negative numbers are determined by assigning a negative value to the weight of the sign bit, summing all the weights where there are 1s, and adding 1 to the result.

EXAMPLE 2–16

Determine the decimal values of the signed binary numbers expressed in 1's complement:

(a) 00010111 **(b)** 11101000

Solution

(a) The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	1	0	1	1	1

Summing the weights where there are 1s,

$$16 + 4 + 2 + 1 = \mathbf{+23}$$

(b) The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of -2^7 or -128.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	0	0	0

Summing the weights where there are 1s,

$$-128 + 64 + 32 + 8 = -24$$

Adding 1 to the result, the final decimal number is

$$-24 + 1 = \mathbf{-23}$$

Related Problem

Determine the decimal value of the 1's complement number 11101011.

2's Complement

Decimal values of positive and negative numbers in the 2's complement form are determined by summing the weights in all bit positions where there are 1s and ignoring those positions where there are zeros. The weight of the sign bit in a negative number is given a negative value.

EXAMPLE 2–17

Determine the decimal values of the signed binary numbers expressed in 2's complement:

(a) 01010110 **(b)** 10101010

Solution

(a) The bits and their powers-of-two weights for the positive number are as follows:

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	0	1	1	0

Summing the weights where there are 1s,

$$64 + 16 + 4 + 2 = \mathbf{+86}$$

(b) The bits and their powers-of-two weights for the negative number are as follows. Notice that the negative sign bit has a weight of $-2^7 = -128$.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	1	0	1	0

Summing the weights where there are 1s,

$$-128 + 32 + 8 + 2 = \mathbf{-86}$$

Related Problem

Determine the decimal value of the 2's complement number 11010111.

From these examples, you can see why the 2's complement form is preferred for representing signed integer numbers: To convert to decimal, it simply requires a summation of weights regardless of whether the number is positive or negative. The 1's complement system requires adding 1 to the summation of weights for negative numbers but not for positive numbers. Also, the 1's complement form is generally not used because two representations of zero (00000000 or 11111111) are possible.

Range of Signed Integer Numbers

The range of magnitude values represented by binary numbers depends on the number of bits (n).

We have used 8-bit numbers for illustration because the 8-bit grouping is common in most computers and has been given the special name **byte**. With one byte or eight bits, you can represent 256 different numbers. With two bytes or sixteen bits, you can represent 65,536 different numbers. With four bytes or 32 bits, you can represent 4.295×10^9 different numbers. The formula for finding the number of different combinations of n bits is

$$\text{Total combinations} = 2^n$$

For 2's complement signed numbers, the range of values for n-bit numbers is

$$\text{Range} = -(2^{n-1}) \text{ to } +(2^{n-1} - 1)$$

where in each case there is one sign bit and $n - 1$ magnitude bits. For example, with four bits you can represent numbers in 2's complement ranging from $-(2^3) = -8$ to $2^3 - 1 = +7$. Similarly, with eight bits you can go from -128 to $+127$, with sixteen bits you can go from

−32,768 to +32,767, and so on. There is one less positive number than there are negative numbers because zero is represented as a positive number (all zeros).

Floating-Point Numbers

To represent very large **integer** (whole) numbers, many bits are required. There is also a problem when numbers with both integer and fractional parts, such as 23.5618, need to be represented. The floating-point number system, based on scientific notation, is capable of representing very large and very small numbers without an increase in the number of bits and also for representing numbers that have both integer and fractional components.

A **floating-point number** (also known as a *real number*) consists of two parts plus a sign. The **mantissa** is the part of a floating-point number that represents the magnitude of the number and is between 0 and 1. The **exponent** is the part of a floating-point number that represents the number of places that the decimal point (or binary point) is to be moved.

A decimal example will be helpful in understanding the basic concept of floating-point numbers. Let's consider a decimal number which, in integer form, is 241,506,800. The mantissa is .2415068 and the exponent is 9. When the integer is expressed as a floating-point number, it is normalized by moving the decimal point to the left of all the digits so that the mantissa is a fractional number and the exponent is the power of ten. The floating-point number is written as

$$0.2415068 \times 10^9$$

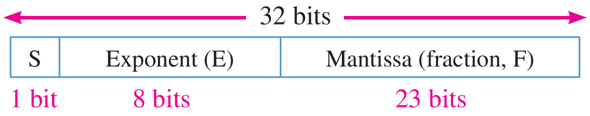
For binary floating-point numbers, the format is defined by ANSI/IEEE Standard 754-1985 in three forms: *single-precision, double-precision,* and *extended-precision.* These all have the same basic formats except for the number of bits. Single-precision floating-point numbers have 32 bits, double-precision numbers have 64 bits, and extended-precision numbers have 80 bits. We will restrict our discussion to the single-precision floating-point format.

InfoNote

In addition to the CPU (central processing unit), computers use *coprocessors* to perform complicated mathematical calculations using floating-point numbers. The purpose is to increase performance by freeing up the CPU for other tasks. The mathematical coprocessor is also known as the floating-point unit (FPU).

Single-Precision Floating-Point Binary Numbers

In the standard format for a single-precision binary number, the sign bit (S) is the left-most bit, the exponent (E) includes the next eight bits, and the mantissa or fractional part (F) includes the remaining 23 bits, as shown next.



In the mantissa or fractional part, the binary point is understood to be to the left of the 23 bits. Effectively, there are 24 bits in the mantissa because in any binary number the left-most (most significant) bit is always a 1. Therefore, this 1 is understood to be there although it does not occupy an actual bit position.

The eight bits in the exponent represent a *biased exponent,* which is obtained by adding 127 to the actual exponent. The purpose of the bias is to allow very large or very small numbers without requiring a separate sign bit for the exponents. The biased exponent allows a range of actual exponent values from −126 to +128.

To illustrate how a binary number is expressed in floating-point format, let's use 1011010010001 as an example. First, it can be expressed as 1 plus a fractional binary number by moving the binary point 12 places to the left and then multiplying by the appropriate power of two.

$$1011010010001 = 1.011010010001 \times 2^{12}$$

Assuming that this is a positive number, the sign bit (S) is 0. The exponent, 12, is expressed as a biased exponent by adding it to 127 (12 + 127 = 139). The biased exponent (E) is expressed as the binary number 10001011. The mantissa is the fractional part (F) of the binary number, .011010010001. Because there is always a 1 to the left of the binary point in the power-of-two expression, it is not included in the mantissa. The complete floating-point number is

S	E	F
0	10001011	01101001000100000000000

Next, let's see how to evaluate a binary number that is already in floating-point format. The general approach to determining the value of a floating-point number is expressed by the following formula:

$$\text{Number} = (-1)^{S}(1 + F)(2^{E-127})$$

To illustrate, let's consider the following floating-point binary number:

S	E	F
1	10010001	10001110001000000000000

The sign bit is 1. The biased exponent is 10010001 $=$ 145. Applying the formula, we get

$$\begin{aligned}\text{Number} &= (-1)^{1}\,(1.10001110001)(2^{145-127})\\ &= (-1)(1.10001110001)(2^{18}) = -1100011100010000000\end{aligned}$$

This floating-point binary number is equivalent to −407,688 in decimal. Since the exponent can be any number between −126 and +128, extremely large and small numbers can be expressed. A 32-bit floating-point number can replace a binary integer number having 129 bits. Because the exponent determines the position of the binary point, numbers containing both integer and fractional parts can be represented.

There are two exceptions to the format for floating-point numbers: The number 0.0 is represented by all 0s, and infinity is represented by all 1s in the exponent and all 0s in the mantissa.

EXAMPLE 2–18

Convert the decimal number 3.248×10^{4} to a single-precision floating-point binary number.

Solution

Convert the decimal number to binary.

$$3.248 \times 10^{4} = 32480 = 111111011100000_2 = 1.11111011100000 \times 2^{14}$$

The MSB will not occupy a bit position because it is always a 1. Therefore, the mantissa is the fractional 23-bit binary number 11111011100000000000000 and the biased exponent is

$$14 + 127 = 141 = 10001101_2$$

The complete floating-point number is

0	**10001101**	**11111011100000000000000**

Related Problem

Determine the binary value of the following floating-point binary number:

0 10011000 10000100010100110000000

SECTION 2–6 CHECKUP

1. Express the decimal number +9 as an 8-bit binary number in the sign-magnitude system.
2. Express the decimal number −33 as an 8-bit binary number in the 1's complement system.
3. Express the decimal number −46 as an 8-bit binary number in the 2's complement system.
4. List the three parts of a signed, floating-point number.

2–7 Arithmetic Operations with Signed Numbers

In the last section, you learned how signed numbers are represented in three different forms. In this section, you will learn how signed numbers are added, subtracted, multiplied, and divided. Because the 2's complement form for representing signed numbers is the most widely used in computers and microprocessor-based systems, the coverage in this section is limited to 2's complement arithmetic. The processes covered can be extended to the other forms if necessary.

After completing this section, you should be able to

- Add signed binary numbers
- Define *overflow*
- Explain how computers add strings of numbers
- Subtract signed binary numbers
- Multiply signed binary numbers using the direct addition method
- Multiply signed binary numbers using the partial products method
- Divide signed binary numbers

Addition

The two numbers in an addition are the **addend** and the **augend**. The result is the **sum**. There are four cases that can occur when two signed binary numbers are added.

1. Both numbers positive
2. Positive number with magnitude larger than negative number
3. Negative number with magnitude larger than positive number
4. Both numbers negative

Let's take one case at a time using 8-bit signed numbers as examples. The equivalent decimal numbers are shown for reference.

Both numbers positive:

$$\begin{array}{rr} 00000111 & 7 \\ +\ 00000100 & +\ 4 \\ \hline 00001011 & 11 \end{array}$$

Addition of two positive numbers yields a positive number.

The sum is positive and is therefore in true (uncomplemented) binary.

Positive number with magnitude larger than negative number:

Addition of a positive number and a smaller negative number yields a positive number.

$$\begin{array}{rrr} & 00001111 & 15 \\ & +\ 11111010 & +\ -6 \\ \hline \text{Discard carry} \longrightarrow 1 & 00001001 & 9 \end{array}$$

The final carry bit is discarded. The sum is positive and therefore in true (uncomplemented) binary.

Negative number with magnitude larger than positive number:

Addition of a positive number and a larger negative number or two negative numbers yields a negative number in 2's complement.

$$\begin{array}{rr} 00010000 & 16 \\ +\ 11101000 & +\ -24 \\ \hline 11111000 & -8 \end{array}$$

The sum is negative and therefore in 2's complement form.

Both numbers negative:

$$\begin{array}{rrr} & 11111011 & -5 \\ & +\ 11110111 & +\ -9 \\ \hline \text{Discard carry} \longrightarrow 1 & 11110010 & -14 \end{array}$$

The final carry bit is discarded. The sum is negative and therefore in 2's complement form.

In a computer, the negative numbers are stored in 2's complement form so, as you can see, the addition process is very simple: *Add the two numbers and discard any final carry bit.*

Overflow Condition

When two numbers are added and the number of bits required to represent the sum exceeds the number of bits in the two numbers, an **overflow** results as indicated by an incorrect sign bit. An overflow can occur only when both numbers are positive or both numbers are negative. If the sign bit of the result is different than the sign bit of the numbers that are added, overflow is indicated. The following 8-bit example will illustrate this condition.

$$\begin{array}{r r} 01111101 & 125 \\ \underline{+\,00111010} & \underline{+\ \ 58} \\ \underset{\uparrow\ \text{Sign incorrect}}{1}\underbrace{0110111}_{\text{Magnitude incorrect}} & 183 \end{array}$$

In this example the sum of 183 requires eight magnitude bits. Since there are seven magnitude bits in the numbers (one bit is the sign), there is a carry into the sign bit which produces the overflow indication.

Numbers Added Two at a Time

Now let's look at the addition of a string of numbers, added two at a time. This can be accomplished by adding the first two numbers, then adding the third number to the sum of the first two, then adding the fourth number to this result, and so on. This is how computers add strings of numbers. The addition of numbers taken two at a time is illustrated in Example 2–19.

EXAMPLE 2–19

Add the signed numbers: 01000100, 00011011, 00001110, and 00010010.

Solution

The equivalent decimal additions are given for reference.

$$\begin{array}{r r l} 68 & 01000100 & \\ \underline{+\ \ 27} & \underline{+\ \ 00011011} & \text{Add 1st two numbers} \\ 95 & 01011111 & \text{1st sum} \\ \underline{+\ \ 14} & \underline{+\ \ 00001110} & \text{Add 3rd number} \\ 109 & 01101101 & \text{2nd sum} \\ \underline{+\ \ 18} & \underline{+\ \ 00010010} & \text{Add 4th number} \\ 127 & \mathbf{01111111} & \text{Final sum} \end{array}$$

Related Problem

Add 00110011, 10111111, and 01100011. These are signed numbers.

Subtraction

Subtraction is addition with the sign of the subtrahend changed.

Subtraction is a special case of addition. For example, subtracting +6 (the **subtrahend**) from +9 (the **minuend**) is equivalent to adding −6 to +9. Basically, *the subtraction operation changes the sign of the subtrahend and adds it to the minuend.* The result of a subtraction is called the **difference**.

The sign of a positive or negative binary number is changed by taking its 2's complement.

For example, when you take the 2's complement of the positive number 00000100 (+4), you get 11111100, which is −4 as the following sum-of-weights evaluation shows:

$$-128 + 64 + 32 + 16 + 8 + 4 = -4$$

As another example, when you take the 2's complement of the negative number 11101101 (−19), you get 00010011, which is +19 as the following sum-of-weights evaluation shows:

$$16 + 2 + 1 = 19$$

Since subtraction is simply an addition with the sign of the subtrahend changed, the process is stated as follows:

To subtract two signed numbers, take the 2's complement of the subtrahend and add. Discard any final carry bit.

Example 2–20 illustrates the subtraction process.

When you subtract two binary numbers with the 2's complement method, it is important that both numbers have the same number of bits.

EXAMPLE 2–20

Perform each of the following subtractions of the signed numbers:

(a) 00001000 − 00000011 **(b)** 00001100 − 11110111

(c) 11100111 − 00010011 **(d)** 10001000 − 11100010

Solution

Like in other examples, the equivalent decimal subtractions are given for reference.

(a) In this case, 8 − 3 = 8 + (−3) = 5.

	00001000	Minuend (+8)
	+ 11111101	2's complement of subtrahend (−3)
Discard carry ⟶	**1 00000101**	Difference (+5)

(b) In this case, 12 − (−9) = 12 + 9 = 21.

00001100	Minuend (+12)
+ 00001001	2's complement of subtrahend (+9)
00010101	Difference (+21)

(c) In this case, −25 − (+19) = −25 + (−19) = −44.

	11100111	Minuend (−25)
	+ 11101101	2's complement of subtrahend (−19)
Discard carry	**1 11010100**	Difference (−44)

(d) In this case, −120 − (−30) = −120 + 30 = −90.

10001000	Minuend (−120)
+ 00011110	2's complement of subtrahend (+30)
10100110	Difference (−90)

Related Problem

Subtract 01000111 from 01011000.

Multiplication

The numbers in a multiplication are the **multiplicand**, the **multiplier**, and the **product**. These are illustrated in the following decimal multiplication:

$$\begin{array}{rl} 8 & \text{Multiplicand} \\ \underline{\times\ 3} & \text{Multiplier} \\ 24 & \text{Product} \end{array}$$

Multiplication is equivalent to adding a number to itself a number of times equal to the multiplier.

The multiplication operation in most computers is accomplished using addition. As you have already seen, subtraction is done with an adder; now let's see how multiplication is done.

Direct addition and *partial products* are two basic methods for performing multiplication using addition. In the direct addition method, you add the multiplicand a number of times equal to the multiplier. In the previous decimal example (8 × 3), three multiplicands are added: $8 + 8 + 8 = 24$. The disadvantage of this approach is that it becomes very lengthy if the multiplier is a large number. For example, to multiply 350 × 75, you must add 350 to itself 75 times. Incidentally, this is why the term *times* is used to mean multiply.

When two binary numbers are multiplied, both numbers must be in true (uncomplemented) form. The direct addition method is illustrated in Example 2–21 adding two binary numbers at a time.

EXAMPLE 2–21

Multiply the signed binary numbers: 01001101 (multiplicand) and 00000100 (multiplier) using the direct addition method.

Solution

Since both numbers are positive, they are in true form, and the product will be positive. The decimal value of the multiplier is 4, so the multiplicand is added to itself four times as follows:

$$\begin{array}{rl} 01001101 & \text{1st time} \\ \underline{+\ 01001101} & \text{2nd time} \\ 10011010 & \text{Partial sum} \\ \underline{+\ 01001101} & \text{3rd time} \\ 11100111 & \text{Partial sum} \\ \underline{+\ 01001101} & \text{4th time} \\ \mathbf{100110100} & \text{Product} \end{array}$$

Since the sign bit of the multiplicand is 0, it has no effect on the outcome. All of the bits in the product are magnitude bits.

Related Problem

Multiply 01100001 by 00000110 using the direct addition method.

The partial products method is perhaps the more common one because it reflects the way you multiply longhand. The multiplicand is multiplied by each multiplier digit beginning with the least significant digit. The result of the multiplication of the multiplicand by a multiplier digit is called a *partial product*. Each successive partial product is moved (shifted) one place to the left and when all the partial products have been produced, they are added to get the final product. Here is a decimal example.

$$\begin{array}{rl} 239 & \text{Multiplicand} \\ \underline{\times\ 123} & \text{Multiplier} \\ 717 & \text{1st partial product } (3 \times 239) \\ 478 & \text{2nd partial product } (2 \times 239) \\ \underline{+\ 239} & \text{3rd partial product } (1 \times 239) \\ 29{,}397 & \text{Final product} \end{array}$$

The sign of the product of a multiplication depends on the signs of the multiplicand and the multiplier according to the following two rules:

- **If the signs are the same, the product is positive.**
- **If the signs are different, the product is negative.**

The basic steps in the partial products method of binary multiplication are as follows:

Step 1: Determine if the signs of the multiplicand and multiplier are the same or different. This determines what the sign of the product will be.

Step 2: Change any negative number to true (uncomplemented) form. Because most computers store negative numbers in 2's complement, a 2's complement operation is required to get the negative number into true form.

Step 3: Starting with the least significant multiplier bit, generate the partial products. When the multiplier bit is 1, the partial product is the same as the multiplicand. When the multiplier bit is 0, the partial product is zero. Shift each successive partial product one bit to the left.

Step 4: Add each successive partial product to the sum of the previous partial products to get the final product.

Step 5: If the sign bit that was determined in step 1 is negative, take the 2's complement of the product. If positive, leave the product in true form. Attach the sign bit to the product.

EXAMPLE 2–22

Multiply the signed binary numbers: 01010011 (multiplicand) and 11000101 (multiplier).

Solution

Step 1: The sign bit of the multiplicand is 0 and the sign bit of the multiplier is 1. The sign bit of the product will be 1 (negative).

Step 2: Take the 2's complement of the multiplier to put it in true form.

$$11000101 \longrightarrow 00111011$$

Step 3 and 4: The multiplication proceeds as follows. Notice that only the magnitude bits are used in these steps.

1010011	Multiplicand
× 0111011	Multiplier
1010011	1st partial product
+ 1010011	2nd partial product
11111001	Sum of 1st and 2nd
+ 0000000	3rd partial product
011111001	Sum
+ 1010011	4th partial product
1110010001	Sum
+ 1010011	5th partial product
100011000001	Sum
+ 1010011	6th partial product
1001100100001	Sum
+ 0000000	7th partial product
1001100100001	Final product

Step 5: Since the sign of the product is a 1 as determined in step 1, take the 2's complement of the product.

$$1001100100001 \longrightarrow 0110011011111$$

Attach the sign bit

1 0110011011111

Related Problem

Verify the multiplication is correct by converting to decimal numbers and performing the multiplication.

Division

The numbers in a division are the **dividend**, the **divisor**, and the **quotient**. These are illustrated in the following standard division format.

$$\frac{\text{dividend}}{\text{divisor}} = \text{quotient}$$

The division operation in computers is accomplished using subtraction. Since subtraction is done with an adder, division can also be accomplished with an adder.

The result of a division is called the *quotient;* the quotient is the number of times that the divisor will go into the dividend. This means that the divisor can be subtracted from the dividend a number of times equal to the quotient, as illustrated by dividing 21 by 7.

21	Dividend
− 7	1st subtraction of divisor
14	1st partial remainder
− 7	2nd subtraction of divisor
7	2nd partial remainder
− 7	3rd subtraction of divisor
0	Zero remainder

In this simple example, the divisor was subtracted from the dividend three times before a remainder of zero was obtained. Therefore, the quotient is 3.

The sign of the quotient depends on the signs of the dividend and the divisor according to the following two rules:

- **If the signs are the same, the quotient is positive.**
- **If the signs are different, the quotient is negative.**

When two binary numbers are divided, both numbers must be in true (uncomplemented) form. The basic steps in a division process are as follows:

Step 1: Determine if the signs of the dividend and divisor are the same or different. This determines what the sign of the quotient will be. Initialize the quotient to zero.

Step 2: Subtract the divisor from the dividend using 2's complement addition to get the first partial remainder and add 1 to the quotient. If this partial remainder is positive, go to step 3. If the partial remainder is zero or negative, the division is complete.

Step 3: Subtract the divisor from the partial remainder and add 1 to the quotient. If the result is positive, repeat for the next partial remainder. If the result is zero or negative, the division is complete.

Continue to subtract the divisor from the dividend and the partial remainders until there is a zero or a negative result. Count the number of times that the divisor is subtracted and you have the quotient. Example 2–23 illustrates these steps using 8-bit signed binary numbers.

EXAMPLE 2–23

Divide 01100100 by 00011001.

Solution

Step 1: The signs of both numbers are positive, so the quotient will be positive. The quotient is initially zero: 00000000.

Step 2: Subtract the divisor from the dividend using 2's complement addition (remember that final carries are discarded).

$$\begin{array}{rl} 01100100 & \text{Dividend} \\ +\ 11100111 & \text{2's complement of divisor} \\ \hline 01001011 & \text{Positive 1st partial remainder} \end{array}$$

Add 1 to quotient: 00000000 + 00000001 = 00000001.

Step 3: Subtract the divisor from the 1st partial remainder using 2's complement addition.

$$\begin{array}{rl} 01001011 & \text{1st partial remainder} \\ +\ 11100111 & \text{2's complement of divisor} \\ \hline 00110010 & \text{Positive 2nd partial remainder} \end{array}$$

Add 1 to quotient: 00000001 + 00000001 = 00000010.

Step 4: Subtract the divisor from the 2nd partial remainder using 2's complement addition.

$$\begin{array}{rl} 00110010 & \text{2nd partial remainder} \\ +\ 11100111 & \text{2's complement of divisor} \\ \hline 00011001 & \text{Positive 3rd partial remainder} \end{array}$$

Add 1 to quotient: 00000010 + 00000001 = 00000011.

Step 5: Subtract the divisor from the 3rd partial remainder using 2's complement addition.

$$\begin{array}{rl} 00011001 & \text{3rd partial remainder} \\ +\ 11100111 & \text{2's complement of divisor} \\ \hline 00000000 & \text{Zero remainder} \end{array}$$

Add 1 to quotient: 00000011 + 00000001 = **00000100** (final quotient). The process is complete.

Related Problem

Verify that the process is correct by converting to decimal numbers and performing the division.

SECTION 2–7 CHECKUP

1. List the four cases when numbers are added.
2. Add the signed numbers 00100001 and 10111100.
3. Subtract the signed numbers 00110010 from 01110111.
4. What is the sign of the product when two negative numbers are multiplied?
5. Multiply 01111111 by 00000101.
6. What is the sign of the quotient when a positive number is divided by a negative number?
7. Divide 00110000 by 00001100.

2–8 Hexadecimal Numbers

The hexadecimal number system has sixteen characters; it is used primarily as a compact way of displaying or writing binary numbers because it is very easy to convert between binary and hexadecimal. As you are probably aware, long binary numbers are difficult to read and write because it is easy to drop or transpose a bit. Since computers and microprocessors understand only 1s and 0s, it is necessary to use these digits when you program in "machine language." Imagine writing a sixteen bit instruction for a microprocessor system in 1s and 0s. It is much more efficient to use hexadecimal or octal; octal numbers are covered in Section 2–9. Hexadecimal is widely used in computer and microprocessor applications.

After completing this section, you should be able to

- List the hexadecimal characters
- Count in hexadecimal
- Convert from binary to hexadecimal
- Convert from hexadecimal to binary
- Convert from hexadecimal to decimal
- Convert from decimal to hexadecimal
- Add hexadecimal numbers
- Determine the 2's complement of a hexadecimal number
- Subtract hexadecimal numbers

The hexadecimal number system consists of digits 0–9 and letters A–F.

The **hexadecimal** number system has a base of sixteen; that is, it is composed of 16 **numeric** and alphabetic **characters**. Most digital systems process binary data in groups that are multiples of four bits, making the hexadecimal number very convenient because each hexadecimal digit represents a 4-bit binary number (as listed in Table 2–3).

TABLE 2–3

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Ten numeric digits and six alphabetic characters make up the hexadecimal number system. The use of letters A, B, C, D, E, and F to represent numbers may seem strange at first, but keep in mind that any number system is only a set of sequential symbols. If you understand what quantities these symbols represent, then the form of the symbols themselves is less important once you get accustomed to using them. We will use the subscript 16 to designate hexadecimal numbers to avoid confusion with decimal numbers. Sometimes you may see an "h" following a hexadecimal number.

InfoNote

With memories in the gigabyte (GB) range, specifying a memory address in binary is quite cumbersome. For example, it takes 32 bits to specify an address in a 4 GB memory. It is much easier to express a 32-bit code using 8 hexadecimal digits.

Counting in Hexadecimal

How do you count in hexadecimal once you get to F? Simply start over with another column and continue as follows:

. . ., E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, 2F, 30, 31, . . .

With two hexadecimal digits, you can count up to FF_{16}, which is decimal 255. To count beyond this, three hexadecimal digits are needed. For instance, 100_{16} is decimal 256, 101_{16} is decimal 257, and so forth. The maximum 3-digit hexadecimal number is FFF_{16}, or decimal 4095. The maximum 4-digit hexadecimal number is $FFFF_{16}$, which is decimal 65,535.

Binary-to-Hexadecimal Conversion

Converting a binary number to hexadecimal is a straightforward procedure. Simply break the binary number into 4-bit groups, starting at the right-most bit and replace each 4-bit group with the equivalent hexadecimal symbol.

EXAMPLE 2–24

Convert the following binary numbers to hexadecimal:

(a) 1100101001010111 **(b)** 111111000101101001

Solution

(a) $$\underbrace{1100}_{\substack{\downarrow\\ \text{C}}}\underbrace{1010}_{\substack{\downarrow\\ \text{A}}}\underbrace{0101}_{\substack{\downarrow\\ 5}}\underbrace{0111}_{\substack{\downarrow\\ 7}} = \mathbf{CA57}_{16}$$

(b) $$\underbrace{0011}_{\substack{\downarrow\\ 3}}\underbrace{1111}_{\substack{\downarrow\\ \text{F}}}\underbrace{0001}_{\substack{\downarrow\\ 1}}\underbrace{0110}_{\substack{\downarrow\\ 6}}\underbrace{1001}_{\substack{\downarrow\\ 9}} = \mathbf{3F169}_{16}$$

Two zeros have been added in part (b) to complete a 4-bit group at the left.

Related Problem

Convert the binary number 1001111011110011100 to hexadecimal.

Hexadecimal-to-Binary Conversion

To convert from a hexadecimal number to a binary number, reverse the process and replace each hexadecimal symbol with the appropriate four bits.

Hexadecimal is a convenient way to represent binary numbers.

EXAMPLE 2–25

Determine the binary numbers for the following hexadecimal numbers:

(a) $10A4_{16}$ **(b)** $CF8E_{16}$ **(c)** 9742_{16}

Solution

(a) $$\overbrace{\mathbf{1}}^{\substack{1\\ \downarrow}}\overbrace{\mathbf{0000}}^{\substack{0\\ \downarrow}}\overbrace{\mathbf{1010}}^{\substack{\text{A}\\ \downarrow}}\overbrace{\mathbf{0100}}^{\substack{4\\ \downarrow}}$$

(b) $$\overbrace{\mathbf{1100}}^{\substack{\text{C}\\ \downarrow}}\overbrace{\mathbf{1111}}^{\substack{\text{F}\\ \downarrow}}\overbrace{\mathbf{1000}}^{\substack{8\\ \downarrow}}\overbrace{\mathbf{1110}}^{\substack{\text{E}\\ \downarrow}}$$

(c) $$\overbrace{\mathbf{1001}}^{\substack{9\\ \downarrow}}\overbrace{\mathbf{0111}}^{\substack{7\\ \downarrow}}\overbrace{\mathbf{0100}}^{\substack{4\\ \downarrow}}\overbrace{\mathbf{0010}}^{\substack{2\\ \downarrow}}$$

In part (a), the MSB is understood to have three zeros preceding it, thus forming a 4-bit group.

Related Problem

Convert the hexadecimal number 6BD3 to binary.

Conversion between hexadecimal and binary is direct and easy.

It should be clear that it is much easier to deal with a hexadecimal number than with the equivalent binary number. Since conversion is so easy, the hexadecimal system is widely used for representing binary numbers in programming, printouts, and displays.

Hexadecimal-to-Decimal Conversion

One way to find the decimal equivalent of a hexadecimal number is to first convert the hexadecimal number to binary and then convert from binary to decimal.

EXAMPLE 2–26

Convert the following hexadecimal numbers to decimal:

(a) $1C_{16}$ **(b)** $A85_{16}$

Solution

Remember, convert the hexadecimal number to binary first, then to decimal.

(a)

$$\overbrace{0001}^{\substack{1\\ \downarrow}}\overbrace{1100}^{\substack{C\\ \downarrow}} = 2^4 + 2^3 + 2^2 = 16 + 8 + 4 = \mathbf{28}_{10}$$

(b)

$$\overbrace{1010}^{\substack{A\\ \downarrow}}\overbrace{1000}^{\substack{8\\ \downarrow}}\overbrace{0101}^{\substack{5\\ \downarrow}} = 2^{11} + 2^9 + 2^7 + 2^2 + 2^0 = 2048 + 512 + 128 + 4 + 1 = \mathbf{2693}_{10}$$

Related Problem

Convert the hexadecimal number 6BD to decimal.

A calculator can be used to perform arithmetic operations with hexadecimal numbers.

Another way to convert a hexadecimal number to its decimal equivalent is to multiply the decimal value of each hexadecimal digit by its weight and then take the sum of these products. The weights of a hexadecimal number are increasing powers of 16 (from right to left). For a 4-digit hexadecimal number, the weights are

16^3	16^2	16^1	16^0
4096	256	16	1

EXAMPLE 2–27

Convert the following hexadecimal numbers to decimal:

(a) $E5_{16}$ **(b)** $B2F8_{16}$

Solution

Recall from Table 2–3 that letters A through F represent decimal numbers 10 through 15, respectively.

(a) $E5_{16} = (E \times 16) + (5 \times 1) = (14 \times 16) + (5 \times 1) = 224 + 5 = \mathbf{229}_{10}$

(b)

$$\begin{aligned} B2F8_{16} &= (B \times 4096) + (2 \times 256) + (F \times 16) + (8 \times 1) \\ &= (11 \times 4096) + (2 \times 256) + (15 \times 16) + (8 \times 1) \\ &= 45{,}056 + 512 + 240 + 8 = \mathbf{45{,}816}_{10} \end{aligned}$$

Related Problem

Convert $60A_{16}$ to decimal.

CALCULATOR SESSION

Conversion of a Hexadecimal Number to a Decimal Number

Convert hexadecimal 28A to decimal.

TI-36X **Step 1:** 3rd ((HEX)

Step 2: 2 8 3rd 1/x (A)

Step 3: 3rd EE (DEC)

650

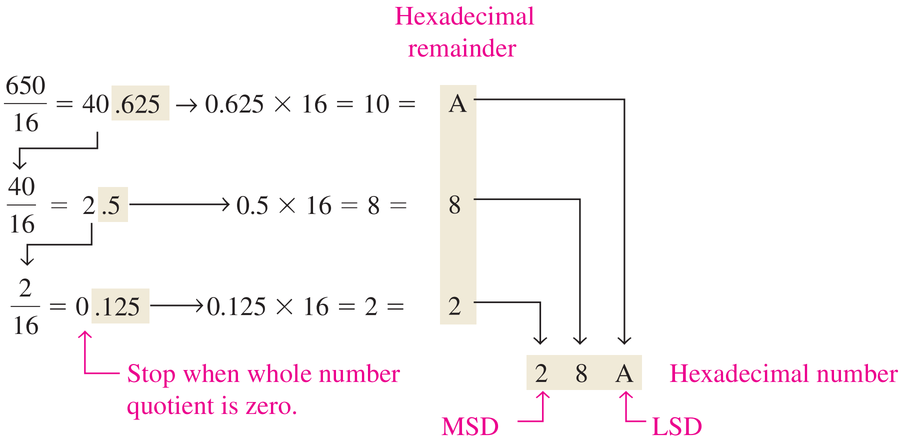
Decimal-to-Hexadecimal Conversion

Repeated division of a decimal number by 16 will produce the equivalent hexadecimal number, formed by the remainders of the divisions. The first remainder produced is the least significant digit (LSD). Each successive division by 16 yields a remainder that becomes a digit in the equivalent hexadecimal number. This procedure is similar to repeated division by 2 for decimal-to-binary conversion that was covered in Section 2–3. Example 2–28 illustrates the procedure. Note that when a quotient has a fractional part, the fractional part is multiplied by the divisor to get the remainder.

EXAMPLE 2–28

Convert the decimal number 650 to hexadecimal by repeated division by 16.

Solution



Related Problem

Convert decimal 2591 to hexadecimal.

CALCULATOR SESSION

Conversion of a Decimal Number to a Hexadecimal Number

Convert decimal 650 to hexadecimal.

TI-36X **Step 1:** 3rd EE (DEC)

Step 2: 6 5 0

Step 3: 3rd ((HEX)

28A

Hexadecimal Addition

Addition can be done directly with hexadecimal numbers by remembering that the hexadecimal digits 0 through 9 are equivalent to decimal digits 0 through 9 and that hexadecimal digits A through F are equivalent to decimal numbers 10 through 15. When adding two hexadecimal numbers, use the following rules. (Decimal numbers are indicated by a subscript 10.)

1. In any given column of an addition problem, think of the two hexadecimal digits in terms of their decimal values. For instance, $5_{16} = 5_{10}$ and $C_{16} = 12_{10}$.
2. If the sum of these two digits is 15_{10} or less, bring down the corresponding hexadecimal digit.
3. If the sum of these two digits is greater than 15_{10}, bring down the amount of the sum that exceeds 16_{10} and carry a 1 to the next column.

EXAMPLE 2–29

Add the following hexadecimal numbers:

(a) $23_{16} + 16_{16}$ **(b)** $58_{16} + 22_{16}$ **(c)** $2B_{16} + 84_{16}$ **(d)** $DF_{16} + AC_{16}$

Solution

(a)
$$\begin{array}{r} 23_{16} \\ +\,16_{16} \\ \hline \mathbf{39_{16}} \end{array}$$

right column: $3_{16} + 6_{16} = 3_{10} + 6_{10} = 9_{10} = 9_{16}$
left column: $2_{16} + 1_{16} = 2_{10} + 1_{10} = 3_{10} = 3_{16}$

(b) $\begin{array}{r} 58_{16} \\ +\ 22_{16} \\ \hline \mathbf{7A_{16}} \end{array}$

right column: $8_{16} + 2_{16} = 8_{10} + 2_{10} = 10_{10} = A_{16}$
left column: $5_{16} + 2_{16} = 5_{10} + 2_{10} = 7_{10} = 7_{16}$

(c) $\begin{array}{r} 2B_{16} \\ +\ 84_{16} \\ \hline \mathbf{AF_{16}} \end{array}$

right column: $B_{16} + 4_{16} = 11_{10} + 4_{10} = 15_{10} = F_{16}$
left column: $2_{16} + 8_{16} = 2_{10} + 8_{10} = 10_{10} = A_{16}$

(d) $\begin{array}{r} DF_{16} \\ +\ AC_{16} \\ \hline \mathbf{18B_{16}} \end{array}$

right column: $F_{16} + C_{16} = 15_{10} + 12_{10} = 27_{10}$
$27_{10} - 16_{10} = 11_{10} = B_{16}$ with a 1 carry
left column: $D_{16} + A_{16} + 1_{16} = 13_{10} + 10_{10} + 1_{10} = 24_{10}$
$24_{10} - 16_{10} = 8_{10} = 8_{16}$ with a 1 carry

Related Problem

Add $4C_{16}$ and $3A_{16}$.

Hexadecimal Subtraction

As you have learned, the 2’s complement allows you to subtract by adding binary numbers. Since a hexadecimal number can be used to represent a binary number, it can also be used to represent the 2’s complement of a binary number.

There are three ways to get the 2’s complement of a hexadecimal number. Method 1 is the most common and easiest to use. Methods 2 and 3 are alternate methods.

Method 1: Convert the hexadecimal number to binary. Take the 2’s complement of the binary number. Convert the result to hexadecimal. This is illustrated in Figure 2–4.

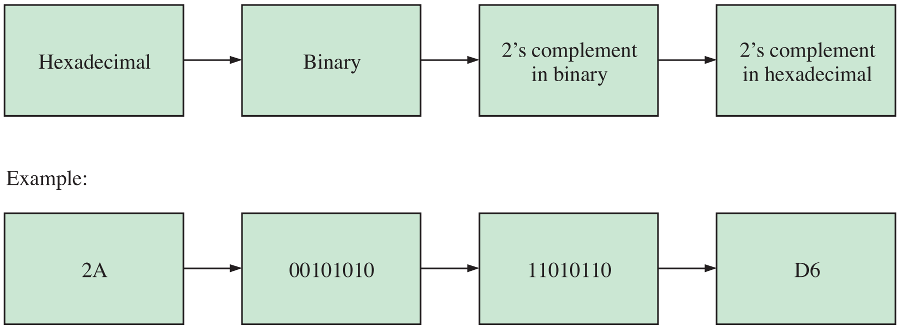


FIGURE 2–4 Getting the 2’s complement of a hexadecimal number, Method 1.

Method 2: Subtract the hexadecimal number from the maximum hexadecimal number and add 1. This is illustrated in Figure 2–5.

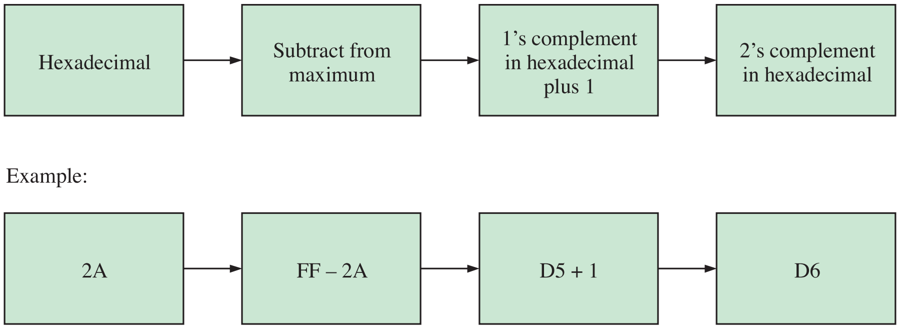

Hexadecimal
Subtract from maximum
1’s complement in hexadecimal plus 1
2’s complement in hexadecimal
Example:
2A
FF – 2A
D5 + 1
D6

FIGURE 2–5 Getting the 2’s complement of a hexadecimal number, Method 2.

Method 3: Write the sequence of single hexadecimal digits. Write the sequence in reverse below the forward sequence. The 1's complement of each hex digit is the digit directly below it. Add 1 to the resulting number to get the 2's complement. This is illustrated in Figure 2–6.

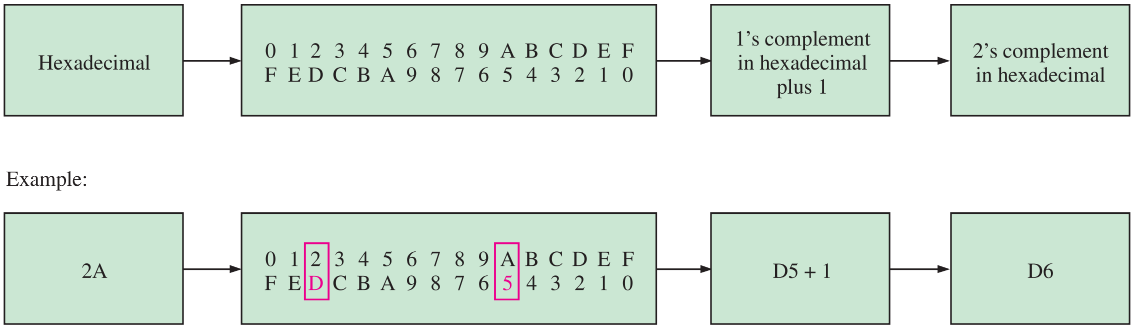

Hexadecimal
0 1 2 3 4 5 6 7 8 9 A B C D E F
F E D C B A 9 8 7 6 5 4 3 2 1 0
1's complement in hexadecimal plus 1
2's complement in hexadecimal
Example:
2A
0 1 2 3 4 5 6 7 8 9 A B C D E F
F E D C B A 9 8 7 6 5 4 3 2 1 0
D5 + 1
D6

FIGURE 2–6 Getting the 2's complement of a hexadecimal number, Method 3.

EXAMPLE 2–30

Subtract the following hexadecimal numbers:

(a) $84_{16} - 2A_{16}$ **(b)** $C3_{16} - 0B_{16}$

Solution

(a) $2A_{16} = 00101010$

2's complement of $2A_{16} = 11010110 = D6_{16}$ (using Method 1)

$$\begin{array}{rl} 84_{16} & \\ \underline{+\ D6_{16}} & \text{Add} \\ \cancel{1}5A_{16} & \text{Drop carry, as in 2's complement addition} \end{array}$$

The difference is $\mathbf{5A_{16}}$.

(b) $0B_{16} = 00001011$

2's complement of $0B_{16} = 11110101 = F5_{16}$ (using Method 1)

$$\begin{array}{rl} C3_{16} & \\ \underline{+\ F5_{16}} & \text{Add} \\ \cancel{1}B8_{16} & \text{Drop carry} \end{array}$$

The difference is $\mathbf{B8}_{16}$.

Related Problem

Subtract 173_{16} from BCD_{16}.

SECTION 2–8 CHECKUP

1. Convert the following binary numbers to hexadecimal:
   (a) 10110011 **(b)** 110011101000
2. Convert the following hexadecimal numbers to binary:
   (a) 57_{16} **(b)** $3A5_{16}$ **(c)** $F80B_{16}$
3. Convert $9B30_{16}$ to decimal.
4. Convert the decimal number 573 to hexadecimal.

5. Add the following hexadecimal numbers directly:

(a) $18_{16} + 34_{16}$ **(b)** $3F_{16} + 2A_{16}$

6. Subtract the following hexadecimal numbers:

(a) $75_{16} - 21_{16}$ **(b)** $94_{16} - 5C_{16}$

2–9 Octal Numbers

Like the hexadecimal number system, the octal number system provides a convenient way to express binary numbers and codes. However, it is used less frequently than hexadecimal in conjunction with computers and microprocessors to express binary quantities for input and output purposes.

After completing this section, you should be able to

- Write the digits of the octal number system
- Convert from octal to decimal
- Convert from decimal to octal
- Convert from octal to binary
- Convert from binary to octal

The **octal** number system is composed of eight digits, which are

0, 1, 2, 3, 4, 5, 6, 7

To count above 7, begin another column and start over:

10, 11, 12, 13, 14, 15, 16, 17, 20, 21, . . .

The octal number system has a base of 8.

Counting in octal is similar to counting in decimal, except that the digits 8 and 9 are not used. To distinguish octal numbers from decimal numbers or hexadecimal numbers, we will use the subscript 8 to indicate an octal number. For instance, 15_8 in octal is equivalent to 13_{10} in decimal and D in hexadecimal. Sometimes you may see an "o" or a "Q" following an octal number.

Octal-to-Decimal Conversion

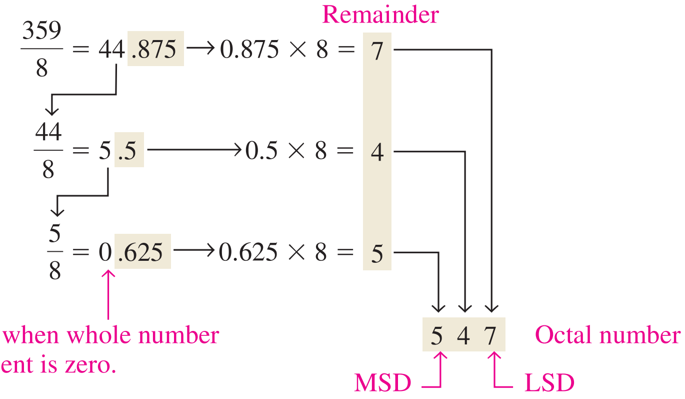
Since the octal number system has a base of eight, each successive digit position is an increasing power of eight, beginning in the right-most column with 8^0. The evaluation of an octal number in terms of its decimal equivalent is accomplished by multiplying each digit by its weight and summing the products, as illustrated here for 2374_8.

$$
\begin{aligned}
\text{Weight:}\quad & 8^3\ 8^2\ 8^1\ 8^0\\
\text{Octal number:}\quad & 2\ \ 3\ \ 7\ \ 4
\end{aligned}
$$

$$
\begin{aligned}
2374_8 &= (2 \times 8^3) + (3 \times 8^2) + (7 \times 8^1) + (4 \times 8^0)\\
&= (2 \times 512) + (3 \times 64) + (7 \times 8) + (4 \times 1)\\
&= 1024 + 192 + 56 + 4 = 1276_{10}
\end{aligned}
$$

Decimal-to-Octal Conversion

A method of converting a decimal number to an octal number is the repeated division-by-8 method, which is similar to the method used in the conversion of decimal numbers to binary or to hexadecimal. To show how it works, let's convert the decimal number 359 to octal. Each successive division by 8 yields a remainder that becomes a digit in the equivalent octal number. The first remainder generated is the least significant digit (LSD).



CALCULATOR SESSION

Conversion of a Decimal Number to an Octal Number

Convert decimal 439 to octal.

DEC

TI-36X **Step 1:** 3rd EE

Step 2: 4 3 9

OCT

Step 3: 3rd)

667

Octal-to-Binary Conversion

Because each octal digit can be represented by a 3-bit binary number, it is very easy to convert from octal to binary. Each octal digit is represented by three bits as shown in Table 2–4.

Octal is a convenient way to represent binary numbers, but it is not as commonly used as hexadecimal.

TABLE 2–4

Octal/binary conversion.

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

To convert an octal number to a binary number, simply replace each octal digit with the appropriate three bits.

EXAMPLE 2–31

Convert each of the following octal numbers to binary:

(a) 13_8 **(b)** 25_8 **(c)** 140_8 **(d)** 7526_8

Solution

(a) $\begin{array}{cc} 1 & 3 \\ \downarrow & \downarrow \\ \overbrace{001} & \overbrace{011} \end{array}$ **(b)** $\begin{array}{cc} 2 & 5 \\ \downarrow & \downarrow \\ \overbrace{010} & \overbrace{101} \end{array}$ **(c)** $\begin{array}{ccc} 1 & 4 & 0 \\ \downarrow & \downarrow & \downarrow \\ \overbrace{001} & \overbrace{100} & \overbrace{000} \end{array}$ **(d)** $\begin{array}{cccc} 7 & 5 & 2 & 6 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \overbrace{111} & \overbrace{101} & \overbrace{010} & \overbrace{110} \end{array}$

Related Problem

Convert each of the binary numbers to decimal and verify that each value agrees with the decimal value of the corresponding octal number.

Binary-to-Octal Conversion

Conversion of a binary number to an octal number is the reverse of the octal-to-binary conversion. The procedure is as follows: Start with the right-most group of three bits and, moving from right to left, convert each 3-bit group to the equivalent octal digit. If there are not three bits available for the left-most group, add either one or two zeros to make a complete group. These leading zeros do not affect the value of the binary number.

EXAMPLE 2–32

Convert each of the following binary numbers to octal:

(a) 110101 **(b)** 101111001 **(c)** 100110011010 **(d)** 11010000100

Solution

(a) $\underbrace{110}_{\substack{\downarrow\\6}}\underbrace{101}_{\substack{\downarrow\\5}} = \mathbf{65_8}$

(b) $\underbrace{101}_{\substack{\downarrow\\5}}\underbrace{111}_{\substack{\downarrow\\7}}\underbrace{001}_{\substack{\downarrow\\1}} = \mathbf{571_8}$

(c) $\underbrace{100}_{\substack{\downarrow\\4}}\underbrace{110}_{\substack{\downarrow\\6}}\underbrace{011}_{\substack{\downarrow\\3}}\underbrace{010}_{\substack{\downarrow\\2}} = \mathbf{4632_8}$

(d) $\underbrace{011}_{\substack{\downarrow\\3}}\underbrace{010}_{\substack{\downarrow\\2}}\underbrace{000}_{\substack{\downarrow\\0}}\underbrace{100}_{\substack{\downarrow\\4}} = \mathbf{3204_8}$

Related Problem

Convert the binary number 1010101000111110010 to octal.

SECTION 2–9 CHECKUP

1. Convert the following octal numbers to decimal:

   (a) 73_8 **(b)** 125_8

2. Convert the following decimal numbers to octal:

   (a) 98_{10} **(b)** 163_{10}

3. Convert the following octal numbers to binary:

   (a) 46_8 **(b)** 723_8 **(c)** 5624_8

4. Convert the following binary numbers to octal:

   (a) 110101111 **(b)** 1001100010 **(c)** 10111111001

2–10 Binary Coded Decimal (BCD)

Binary coded decimal (BCD) is a way to express each of the decimal digits with a binary code. There are only ten code groups in the BCD system, so it is very easy to convert between decimal and BCD. Because we like to read and write in decimal, the BCD code provides an excellent interface to binary systems. Examples of such interfaces are keypad inputs and digital readouts.

After completing this section, you should be able to

- Convert each decimal digit to BCD
- Express decimal numbers in BCD
- Convert from BCD to decimal
- Add BCD numbers

The 8421 BCD Code

In BCD, 4 bits represent each decimal digit.

The 8421 code is a type of **BCD** (binary coded decimal) code. Binary coded decimal means that each decimal digit, 0 through 9, is represented by a binary code of four bits. The designation 8421 indicates the binary weights of the four bits (2^3, 2^2, 2^1, 2^0). The ease of conversion between 8421 code numbers and the familiar decimal numbers is the main advantage of this code. All you have to remember are the ten binary combinations that represent the ten decimal digits as shown in Table 2–5. The 8421 code is the predominant BCD code, and when we refer to BCD, we always mean the 8421 code unless otherwise stated.

TABLE 2–5

Decimal/BCD conversion.

Decimal Digit	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Invalid Codes

You should realize that, with four bits, sixteen numbers (0000 through 1111) can be represented but that, in the 8421 code, only ten of these are used. The six code combinations that are not used—1010, 1011, 1100, 1101, 1110, and 1111—are invalid in the 8421 BCD code.

To express any decimal number in BCD, simply replace each decimal digit with the appropriate 4-bit code, as shown by Example 2–33.

EXAMPLE 2–33

Convert each of the following decimal numbers to BCD:

(a) 35 **(b)** 98 **(c)** 170 **(d)** 2469

Solution

(a) $\overbrace{\mathbf{0011}}^{\substack{3\\ \downarrow}}\overbrace{\mathbf{0101}}^{\substack{5\\ \downarrow}}$ **(b)** $\overbrace{\mathbf{1001}}^{\substack{9\\ \downarrow}}\overbrace{\mathbf{1000}}^{\substack{8\\ \downarrow}}$

(c) $\overbrace{\mathbf{0001}}^{\substack{1\\ \downarrow}}\overbrace{\mathbf{0111}}^{\substack{7\\ \downarrow}}\overbrace{\mathbf{0000}}^{\substack{0\\ \downarrow}}$ **(d)** $\overbrace{\mathbf{0010}}^{\substack{2\\ \downarrow}}\overbrace{\mathbf{0100}}^{\substack{4\\ \downarrow}}\overbrace{\mathbf{0110}}^{\substack{6\\ \downarrow}}\overbrace{\mathbf{1001}}^{\substack{9\\ \downarrow}}$

Related Problem

Convert the decimal number 9673 to BCD.

It is equally easy to determine a decimal number from a BCD number. Start at the right-most bit and break the code into groups of four bits. Then write the decimal digit represented by each 4-bit group.

EXAMPLE 2–34

Convert each of the following BCD codes to decimal:

(a) 10000110 **(b)** 001101010001 **(c)** 1001010001110000

Solution

(a) $\underbrace{1000}_{\substack{\downarrow\\ \mathbf{8}}}\underbrace{0110}_{\substack{\downarrow\\ \mathbf{6}}}$ **(b)** $\underbrace{0011}_{\substack{\downarrow\\ \mathbf{3}}}\underbrace{0101}_{\substack{\downarrow\\ \mathbf{5}}}\underbrace{0001}_{\substack{\downarrow\\ \mathbf{1}}}$ **(c)** $\underbrace{1001}_{\substack{\downarrow\\ \mathbf{9}}}\underbrace{0100}_{\substack{\downarrow\\ \mathbf{4}}}\underbrace{0111}_{\substack{\downarrow\\ \mathbf{7}}}\underbrace{0000}_{\substack{\downarrow\\ \mathbf{0}}}$

Related Problem

Convert the BCD code 10000010001001110110 to decimal.

InfoNote

BCD is sometimes used for arithmetic operations in processors. To represent BCD numbers in a processor, they usually are “packed,” so that eight bits have two BCD digits. Normally, a processor will add numbers as if they were straight binary. Special instructions are available for computer programmers to correct the results when BCD numbers are added or subtracted. For example, in Assembly Language, the programmer will include a DAA (Decimal Adjust for Addition) instruction to automatically correct the answer to BCD following an addition.

Applications

Digital clocks, digital thermometers, digital meters, and other devices with seven-segment displays typically use BCD code to simplify the displaying of decimal numbers. BCD is not as efficient as straight binary for calculations, but it is particularly useful if only limited processing is required, such as in a digital thermometer.

BCD Addition

BCD is a numerical code and can be used in arithmetic operations. Addition is the most important operation because the other three operations (subtraction, multiplication, and division) can be accomplished by the use of addition. Here is how to add two BCD numbers:

Step 1: Add the two BCD numbers, using the rules for binary addition in Section 2–4.

Step 2: If a 4-bit sum is equal to or less than 9, it is a valid BCD number.

Step 3: If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6 (0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added, simply add the carry to the next 4-bit group.

Example 2–35 illustrates BCD additions in which the sum in each 4-bit column is equal to or less than 9, and the 4-bit sums are therefore valid BCD numbers. Example 2–36 illustrates the procedure in the case of invalid sums (greater than 9 or a carry).

An alternative method to add BCD numbers is to convert them to decimal, perform the addition, and then convert the answer back to BCD.

EXAMPLE 2–35

Add the following BCD numbers:

(a) 0011 + 0100 **(b)** 00100011 + 00010101

(c) 10000110 + 00010011 **(d)** 010001010000 + 010000010111

Solution

The decimal number additions are shown for comparison.

(a)
$$\begin{array}{r} 0011 \\ +\,0100 \\ \hline \mathbf{0111} \end{array} \qquad \begin{array}{r} 3 \\ +\,4 \\ \hline 7 \end{array}$$

(b)
$$\begin{array}{rr} 0010 & 0011 \\ +\,0001 & 0101 \\ \hline \mathbf{0011} & \mathbf{1000} \end{array} \qquad \begin{array}{r} 23 \\ +\,15 \\ \hline 38 \end{array}$$

(c)
$$\begin{array}{rr} 1000 & 0110 \\ +\,0001 & 0011 \\ \hline \mathbf{1001} & \mathbf{1001} \end{array} \qquad \begin{array}{r} 86 \\ +\,13 \\ \hline \mathbf{99} \end{array}$$

(d)
$$\begin{array}{rrr} 0100 & 0101 & 0000 \\ +\,0100 & 0001 & 0111 \\ \hline \mathbf{1000} & \mathbf{0110} & \mathbf{0111} \end{array} \qquad \begin{array}{r} 450 \\ +\,417 \\ \hline \mathbf{867} \end{array}$$

Note that in each case the sum in any 4-bit column does not exceed 9, and the results are valid BCD numbers.

Related Problem

Add the BCD numbers: 1001000001000011 + 0000100100100101.

EXAMPLE 2–36

Add the following BCD numbers:

(a) 1001 + 0100 **(b)** 1001 + 1001

(c) 00010110 + 00010101 **(d)** 01100111 + 01010011

Solution

The decimal number additions are shown for comparison.

(a)

$$\begin{array}{rrlr}
 & 1001 & & 9\\
 & \underline{+\,0100} & & \underline{+4}\\
 & 1101 & \text{Invalid BCD number } (>9) & 13\\
 & +\,0110 & \text{Add 6} & \\
\hline
\underbrace{\mathbf{0001}}_{\substack{\downarrow\\ 1}} & \underbrace{\mathbf{0011}}_{\substack{\downarrow\\ 3}} & \text{Valid BCD number} &
\end{array}$$

(b)

$$\begin{array}{rrlr}
 & 1001 & & 9\\
 & \underline{+\,1001} & & \underline{+\,9}\\
1 & 0010 & \text{Invalid because of carry} & 18\\
 & +\,0110 & \text{Add 6} & \\
\hline
\underbrace{\mathbf{0001}}_{\substack{\downarrow\\ 1}} & \underbrace{\mathbf{1000}}_{\substack{\downarrow\\ 8}} & \text{Valid BCD number} &
\end{array}$$

(c)

$$\begin{array}{rrlr}
0001 & 0110 & & 16\\
\underline{+\,0001} & \underline{0101} & & \underline{+\,15}\\
0010 & 1011 & \text{Right group is invalid } (>9), & 31\\
 & & \quad\text{left group is valid.} & \\
 & +\,0110 & \text{Add 6 to invalid code. Add} & \\
 & & \quad\text{carry, 0001, to next group.} & \\
\hline
\underbrace{\mathbf{0011}}_{\substack{\downarrow\\ 3}} & \underbrace{\mathbf{0001}}_{\substack{\downarrow\\ 1}} & \text{Valid BCD number} &
\end{array}$$

(d)

$$\begin{array}{rrrlr}
 & 0110 & 0111 & & 67\\
 & \underline{+\,0101} & \underline{0011} & & \underline{+\,53}\\
 & 1011 & 1010 & \text{Both groups are invalid } (>9) & 120\\
 & +\,0110 & +\,0110 & \text{Add 6 to both groups} & \\
\hline
\underbrace{\mathbf{0001}}_{\substack{\downarrow\\ 1}} & \underbrace{\mathbf{0010}}_{\substack{\downarrow\\ 2}} & \underbrace{\mathbf{0000}}_{\substack{\downarrow\\ 0}} & \text{Valid BCD number} &
\end{array}$$

Related Problem

Add the BCD numbers: 01001000 + 00110100.

SECTION 2–10 CHECKUP

1. What is the binary weight of each 1 in the following BCD numbers?

   (a) 0010 **(b)** 1000 **(c)** 0001 **(d)** 0100

2. Convert the following decimal numbers to BCD:

   (a) 6 **(b)** 15 **(c)** 273 **(d)** 849

3. What decimal numbers are represented by each BCD code?

   (a) 10001001 **(b)** 001001111000 **(c)** 000101010111

4. In BCD addition, when is a 4-bit sum invalid?

2–11 Digital Codes

Many specialized codes are used in digital systems. You have just learned about the BCD code; now let's look at a few others. Some codes are strictly numeric, like BCD, and others are alphanumeric; that is, they are used to represent numbers, letters, symbols, and instructions. The codes introduced in this section are the Gray code, the ASCII code, and the Unicode.

After completing this section, you should be able to

- Explain the advantage of the Gray code
- Convert between Gray code and binary
- Use the ASCII code
- Discuss the Unicode

The Gray Code

The single bit change characteristic of the Gray code minimizes the chance for error.

The **Gray code** is unweighted and is not an arithmetic code; that is, there are no specific weights assigned to the bit positions. The important feature of the Gray code is that *it exhibits only a single bit change from one code word to the next in sequence.* This property is important in many applications, such as shaft position encoders, where error susceptibility increases with the number of bit changes between adjacent numbers in a sequence.

Table 2–6 is a listing of the 4-bit Gray code for decimal numbers 0 through 15. Binary numbers are shown in the table for reference. Like binary numbers, *the Gray code can have any number of bits.* Notice the single-bit change between successive Gray code words. For instance, in going from decimal 3 to decimal 4, the Gray code changes from 0010 to 0110, while the binary code changes from 0011 to 0100, a change of three bits. The only bit change in the Gray code is in the third bit from the right: the other bits remain the same.

TABLE 2–6

Four-bit Gray code.

Decimal	Binary	Gray Code	Decimal	Binary	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Binary-to-Gray Code Conversion

Conversion between binary code and Gray code is sometimes useful. The following rules explain how to convert from a binary number to a Gray code word:

1. The most significant bit (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

For example, the conversion of the binary number 10110 to Gray code is as follows:

$$\begin{tikzcd}
1 \arrow[r, "+"] \arrow[d] & 0 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 0 \arrow[d] & \text{Binary} \\
1 & 1 & 1 & 0 & 1 & \text{Gray}
\end{tikzcd}$$

The Gray code is 11101.

Gray-to-Binary Code Conversion

To convert from Gray code to binary, use a similar method; however, there are some differences. The following rules apply:

1. The most significant bit (left-most) in the binary code is the same as the corresponding bit in the Gray code.
2. Add each binary code bit generated to the Gray code bit in the next adjacent position. Discard carries.

For example, the conversion of the Gray code word 11011 to binary is as follows:

$$\begin{tikzcd}
1 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & \text{Gray} \\
1 \arrow[ur, "+"] & 0 \arrow[ur, "+"] & 0 \arrow[ur, "+"] & 1 \arrow[ur, "+"] & 0 & \text{Binary}
\end{tikzcd}$$

The binary number is 10010.

EXAMPLE 2–37

(a) Convert the binary number 11000110 to Gray code.

(b) Convert the Gray code 10101111 to binary.

Solution

(a) Binary to Gray code:

$$\begin{tikzcd}
1 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 0 \arrow[r, "+"] \arrow[d] & 0 \arrow[r, "+"] \arrow[d] & 0 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 1 \arrow[r, "+"] \arrow[d] & 0 \arrow[d] \\
\mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1}
\end{tikzcd}$$

(b) Gray code to binary:

$$\begin{tikzcd}
1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] & 1 \arrow[d] \\
\mathbf{1} \arrow[ur, "+"] & \mathbf{1} \arrow[ur, "+"] & \mathbf{0} \arrow[ur, "+"] & \mathbf{0} \arrow[ur, "+"] & \mathbf{1} \arrow[ur, "+"] & \mathbf{0} \arrow[ur, "+"] & \mathbf{1} \arrow[ur, "+"] & \mathbf{0}
\end{tikzcd}$$

Related Problem

(a) Convert binary 101101 to Gray code.

(b) Convert Gray code 100111 to binary.

An Application

The concept of a 3-bit shaft position encoder is shown in Figure 2–7. Basically, there are three concentric rings that are segmented into eight sectors. The more sectors there are, the more accurately the position can be represented, but we are using only eight to illustrate. Each sector of each ring is either reflective or nonreflective. As the rings rotate with the shaft, they come under an IR emitter that produces three separate IR beams. A 1 is indicated where there is a reflected beam, and a 0 is indicated where there is no reflected beam. The IR detector senses the presence or absence of reflected

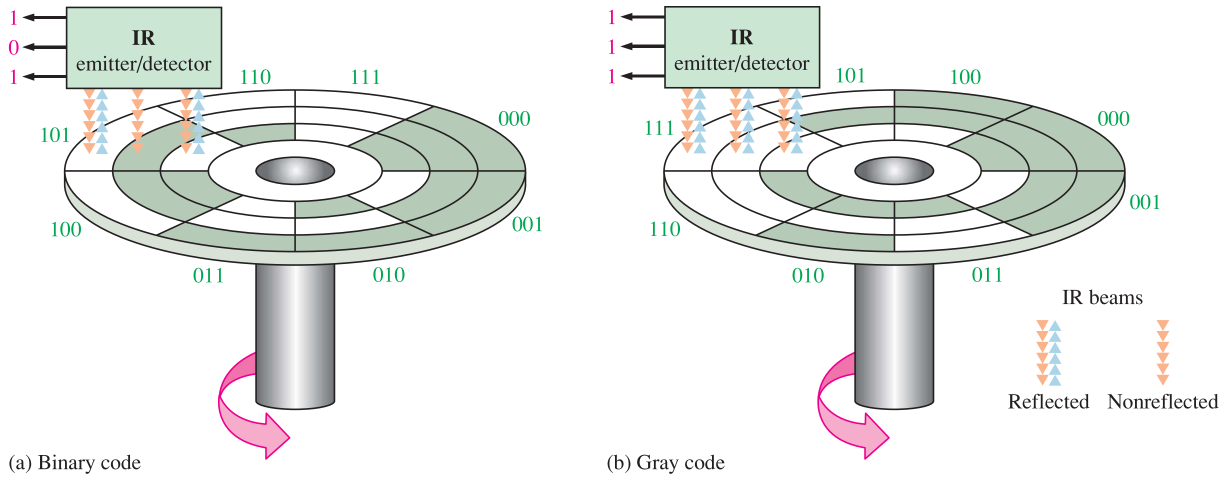


FIGURE 2–7 A simplified illustration of how the Gray code solves the error problem in shaft position encoders. Three bits are shown to illustrate the concept, although most shaft encoders use more than 10 bits to achieve a higher resolution.

beams and produces a corresponding 3-bit code. The IR emitter/detector is in a fixed position. As the shaft rotates counterclockwise through 360°, the eight sectors move under the three beams. Each beam is either reflected or absorbed by the sector surface to represent a binary or Gray code number that indicates the shaft position.

In Figure 2–7(a), the sectors are arranged in a straight binary pattern, so that the detector output goes from 000 to 001 to 010 to 011 and so on. When a beam is aligned over a reflective sector, the output is 1; when a beam is aligned over a nonreflective sector, the output is 0. If one beam is slightly ahead of the others during the transition from one sector to the next, an erroneous output can occur. Consider what happens when the beams are on the 111 sector and about to enter the 000 sector. If the MSB beam is slightly ahead, the position would be incorrectly indicated by a transitional 011 instead of a 111 or a 000. In this type of application, it is virtually impossible to maintain precise mechanical alignment of the IR emitter/detector beams; therefore, some error will usually occur at many of the transitions between sectors.

The Gray code is used to eliminate the error problem which is inherent in the binary code. As shown in Figure 2–7(b), the Gray code assures that only one bit will change between adjacent sectors. This means that even though the beams may not be in precise alignment, there will never be a transitional error. For example, let's again consider what happens when the beams are on the 111 sector and about to move into the next sector, 101. The only two possible outputs during the transition are 111 and 101, no matter how the beams are aligned. A similar situation occurs at the transitions between each of the other sectors.

Alphanumeric Codes

In order to communicate, you need not only numbers, but also letters and other symbols. In the strictest sense, **alphanumeric** codes are codes that represent numbers and alphabetic characters (letters). Most such codes, however, also represent other characters such as symbols and various instructions necessary for conveying information.

At a minimum, an alphanumeric code must represent 10 decimal digits and 26 letters of the alphabet, for a total of 36 items. This number requires six bits in each code combination because five bits are insufficient ($2^5 = 32$). There are 64 total combinations of six bits, so there are 28 unused code combinations. Obviously, in many applications, symbols other than just numbers and letters are necessary to communicate completely. You need spaces, periods, colons, semicolons, question marks, etc. You also need instructions to tell the receiving system what to do with the information. With codes that are six bits long, you can handle decimal numbers, the alphabet, and 28 other symbols. This should give you an idea of the requirements for a basic alphanumeric code. The ASCII is a common alphanumeric code and is covered next.

ASCII

ASCII is the abbreviation for American Standard Code for Information Interchange. Pronounced "askee," ASCII is a universally accepted alphanumeric code used in most computers and other electronic equipment. Most computer keyboards are standardized with the ASCII. When you enter a letter, a number, or control command, the corresponding ASCII code goes into the computer.

ASCII has 128 characters and symbols represented by a 7-bit binary code. Actually, ASCII can be considered an 8-bit code with the MSB always 0. This 8-bit code is 00 through 7F in hexadecimal. The first thirty-two ASCII characters are nongraphic commands that are never printed or displayed and are used only for control purposes. Examples of the control characters are "null," "line feed," "start of text," and "escape." The other characters are graphic symbols that can be printed or displayed and include the letters of the alphabet (lowercase and uppercase), the ten decimal digits, punctuation signs, and other commonly used symbols.

Table 2–7 is a listing of the ASCII code showing the decimal, hexadecimal, and binary representations for each character and symbol. The left section of the table lists the names of the 32 control characters (00 through 1F hexadecimal). The graphic symbols are listed in the rest of the table (20 through 7F hexadecimal).

InfoNote

A computer keyboard has a dedicated microprocessor that constantly scans keyboard circuits to detect when a key has been pressed and released. A unique scan code is produced by computer software representing that particular key. The scan code is then converted to an alphanumeric code (ASCII) for use by the computer.

EXAMPLE 2–38

Use Table 2–7 to determine the binary ASCII codes that are entered from the computer's keyboard when the following C language program statement is typed in. Also express each code in hexadecimal.

if (x > 5)

Solution

The ASCII code for each symbol is found in Table 2–7.

Symbol	Binary	Hexadecimal
i	1101001	69_{16}
f	1100110	66_{16}
Space	0100000	20_{16}
(	0101000	28_{16}
x	1111000	78_{16}
>	0111110	$3E_{16}$
5	0110101	35_{16}
)	0101001	29_{16}

Related Problem

Use Table 2–7 to determine the sequence of ASCII codes required for the following C program statement and express each code in hexadecimal:

if (y < 8)

The ASCII Control Characters

The first thirty-two codes in the ASCII table (Table 2–7) represent the control characters. These are used to allow devices such as a computer and printer to communicate with each other when passing information and data. The control key function allows a control character to be entered directly from an ASCII keyboard by pressing the control key (CTRL) and the corresponding symbol.

TABLE 2–7

American Standard Code for Information Interchange (ASCII).

Control Characters				Graphic Symbols											
Name	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**	**Symbol**	**Dec**	**Binary**	**Hex**
NUL	0	0000000	00	space	32	0100000	20	@	64	1000000	40	′	96	1100000	60
SOH	1	0000001	01	!	33	0100001	21	A	65	1000001	41	a	97	1100001	61
STX	2	0000010	02	”	34	0100010	22	B	66	1000010	42	b	98	1100010	62
ETX	3	0000011	03	#	35	0100011	23	C	67	1000011	43	c	99	1100011	63
EOT	4	0000100	04	$	36	0100100	24	D	68	1000100	44	d	100	1100100	64
ENQ	5	0000101	05	%	37	0100101	25	E	69	1000101	45	e	101	1100101	65
ACK	6	0000110	06	&	38	0100110	26	F	70	1000110	46	f	102	1100110	66
BEL	7	0000111	07	’	39	0100111	27	G	71	1000111	47	g	103	1100111	67
BS	8	0001000	08	(	40	0101000	28	H	72	1001000	48	h	104	1101000	68
HT	9	0001001	09	)	41	0101001	29	I	73	1001001	49	i	105	1101001	69
LF	10	0001010	0A	*	42	0101010	2A	J	74	1001010	4A	j	106	1101010	6A
VT	11	0001011	0B	+	43	0101011	2B	K	75	1001011	4B	k	107	1101011	6B
FF	12	0001100	0C	,	44	0101100	2C	L	76	1001100	4C	l	108	1101100	6C
CR	13	0001101	0D	−	45	0101101	2D	M	77	1001101	4D	m	109	1101101	6D
SO	14	0001110	0E	.	46	0101110	2E	N	78	1001110	4E	n	110	1101110	6E
SI	15	0001111	0F	/	47	0101111	2F	O	79	1001111	4F	o	111	1101111	6F
DLE	16	0010000	10	0	48	0110000	30	P	80	1010000	50	p	112	1110000	70
DC1	17	0010001	11	1	49	0110001	31	Q	81	1010001	51	q	113	1110001	71
DC2	18	0010010	12	2	50	0110010	32	R	82	1010010	52	r	114	1110010	72
DC3	19	0010011	13	3	51	0110011	33	S	83	1010011	53	s	115	1110011	73
DC4	20	0010100	14	4	52	0110100	34	T	84	1010100	54	t	116	1110100	74
NAK	21	0010101	15	5	53	0110101	35	U	85	1010101	55	u	117	1110101	75
SYN	22	0010110	16	6	54	0110110	36	V	86	1010110	56	v	118	1110110	76
ETB	23	0010111	17	7	55	0110111	37	W	87	1010111	57	w	119	1110111	77
CAN	24	0011000	18	8	56	0111000	38	X	88	1011000	58	x	120	1111000	78
EM	25	0011001	19	9	57	0111001	39	Y	89	1011001	59	y	121	1111001	79
SUB	26	0011010	1A	:	58	0111010	3A	Z	90	1011010	5A	z	122	1111010	7A
ESC	27	0011011	1B	;	59	0111011	3B	[	91	1011011	5B	{	123	1111011	7B
FS	28	0011100	1C	<	60	0111100	3C	\	92	1011100	5C	|	124	1111100	7C
GS	29	0011101	1D	=	61	0111101	3D	]	93	1011101	5D	}	125	1111101	7D
RS	30	0011110	1E	>	62	0111110	3E	^	94	1011110	5E	~	126	1111110	7E
US	31	0011111	1F	?	63	0111111	3F	_	95	1011111	5F	Del	127	1111111	7F

Extended ASCII Characters

In addition to the 128 standard ASCII characters, there are an additional 128 characters that were adopted by IBM for use in their PCs (personal computers). Because of the popularity of the PC, these particular extended ASCII characters are also used in applications other than PCs and have become essentially an unofficial standard.

The extended ASCII characters are represented by an 8-bit code series from hexadecimal 80 to hexadecimal FF and can be grouped into the following general categories: foreign (non-English) alphabetic characters, foreign currency symbols, Greek letters, mathematical symbols, drawing characters, bar graphing characters, and shading characters.

Unicode

Unicode provides the ability to encode all of the characters used for the written languages of the world by assigning each character a unique numeric value and name utilizing the universal character set (UCS). It is applicable in computer applications dealing with multilingual text, mathematical symbols, or other technical characters.

Unicode has a wide array of characters, and their various encoding forms are used in many environments. While ASCII basically uses 7-bit codes, Unicode uses relatively abstract "code points"—non-negative integer numbers—that map sequences of one or more bytes, using different encoding forms and schemes. To permit compatibility, Unicode assigns the first 128 code points to the same characters as ASCII. One can, therefore, think of ASCII as a 7-bit encoding scheme for a very small subset of Unicode and of the UCS.

Unicode consists of about 100,000 characters, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, and an enumeration of character properties such as uppercase and lowercase. It also consists of a number of related items, such as character properties, rules for text normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic or Hebrew, and left-to-right scripts).

SECTION 2–11 CHECKUP

1. Convert the following binary numbers to the Gray code:

   (a) 1100 **(b)** 1010 **(c)** 11010

2. Convert the following Gray codes to binary:

   (a) 1000 **(b)** 1010 **(c)** 11101

3. What is the ASCII representation for each of the following characters? Express each as a bit pattern and in hexadecimal notation.

   (a) K **(b)** r **(c)** $ **(d)** +

2–12 Error Codes

In this section, three methods for adding bits to codes to detect a single-bit error are discussed. The parity method of error detection is introduced, and the cyclic redundancy check is discussed. Also, the Hamming code for error detection and correction is presented.

After completing this section, you should be able to

- Determine if there is an error in a code based on the parity bit
- Assign the proper parity bit to a code
- Explain the cyclic redundancy (CRC) check
- Describe the Hamming code

Parity Method for Error Detection

A parity bit tells if the number of 1s is odd or even.

Many systems use a parity bit as a means for bit **error detection**. Any group of bits contain either an even or an odd number of 1s. A parity bit is attached to a group of bits to make the total number of 1s in a group always even or always odd. An even parity bit makes the total number of 1s even, and an odd parity bit makes the total odd.

A given system operates with even or odd **parity**, but not both. For instance, if a system operates with even parity, a check is made on each group of bits received to make sure the total number of 1s in that group is even. If there is an odd number of 1s, an error has occurred.

As an illustration of how parity bits are attached to a code, Table 2–8 lists the parity bits for each BCD number for both even and odd parity. The parity bit for each BCD number is in the *P* column.

TABLE 2–8

The BCD code with parity bits.

Even Parity		Odd Parity	
P	**BCD**	***P***	**BCD**
0	0000	1	0000
1	0001	0	0001
1	0010	0	0010
0	0011	1	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

The parity bit can be attached to the code at either the beginning or the end, depending on system design. Notice that the total number of 1s, including the parity bit, is always even for even parity and always odd for odd parity.

Detecting an Error

A parity bit provides for the detection of a single bit error (or any odd number of errors, which is very unlikely) but cannot check for two errors in one group. For instance, let's assume that we wish to transmit the BCD code 0101. (Parity can be used with any number of bits; we are using four for illustration.) The total code transmitted, including the even parity bit, is

$$\overset{\substack{\text{Even parity bit}\\\downarrow}}{0}\underbrace{0101}_{\substack{\uparrow\\\text{BCD code}}}$$

Now let's assume that an error occurs in the third bit from the left (the 1 becomes a 0).

$$\overset{\substack{\text{Even parity bit}\\\downarrow}}{0}0\underset{\substack{\uparrow\\\text{Bit error}}}{0}01$$

When this code is received, the parity check circuitry determines that there is only a single 1 (odd number), when there should be an even number of 1s. Because an even number of 1s does not appear in the code when it is received, an error is indicated.

An odd parity bit also provides in a similar manner for the detection of a single error in a given group of bits.

EXAMPLE 2–39

Assign the proper even parity bit to the following code groups:

(a) 1010 **(b)** 111000 **(c)** 101101
(d) 1000111001001 **(e)** 101101011111

Solution

Make the parity bit either 1 or 0 as necessary to make the total number of 1s even. The parity bit will be the left-most bit (color).

(a) **0**1010 **(b)** **1**111000 **(c)** **0**101101
(d) **0**100011100101 **(e)** **1**101101011111

Related Problem

Add an even parity bit to the 7-bit ASCII code for the letter K.

EXAMPLE 2–40

An odd parity system receives the following code groups: 10110, 11010, 110011, 110101110100, and 1100010101010. Determine which groups, if any, are in error.

Solution

Since odd parity is required, any group with an even number of 1s is incorrect. The following groups are in error: **110011** and **1100010101010**.

Related Problem

The following ASCII character is received by an odd parity system: 00110111. Is it correct?

Cyclic Redundancy Check

The **cyclic redundancy check (CRC)** is a widely used code used for detecting one- and two-bit transmission errors when digital data are transferred on a communication link. The communication link can be between two computers that are connected to a network or between a digital storage device (such as a CD, DVD, or a hard drive) and a PC. If it is properly designed, the CRC can also detect multiple errors for a number of bits in sequence (burst errors). In CRC, a certain number of check bits, sometimes called a *checksum*, are appended to the data bits (added to end) that are being transmitted. The transmitted data are tested by the receiver for errors using the CRC. Not every possible error can be identified, but the CRC is much more efficient than just a simple parity check.

CRC is often described mathematically as the division of two polynomials to generate a remainder. A polynomial is a mathematical expression that is a sum of terms with positive exponents. When the coefficients are limited to 1s and 0s, it is called a *univariate polynomial*. An example of a univariate polynomial is $1x^3 + 0x^2 + 1x^1 + 1x^0$ or simply $x^3 + x^1 + x^0$, which can be fully described by the 4-bit binary number 1011. Most cyclic redundancy checks use a 16-bit or larger polynomial, but for simplicity the process is illustrated here with four bits.

Modulo-2 Operations

Simply put, CRC is based on the division of two binary numbers; and, as you know, division is just a series of subtractions and shifts. To do subtraction, a method called *modulo-2* addition can be used. Modulo-2 addition (or subtraction) is the same as binary addition with the carries discarded, as shown in the truth table in Table 2–9. **Truth tables** are widely used to describe the operation of logic circuits, as you will learn in Chapter 3. With two bits, there is a total of four possible combinations, as shown in the table. This particular table describes the modulo-2 operation also known as *exclusive-OR* and can be implemented with a logic

TABLE 2–9

Modulo-2 operation.

Input Bits	Output Bit
0 0	0
0 1	1
1 0	1
1 1	0

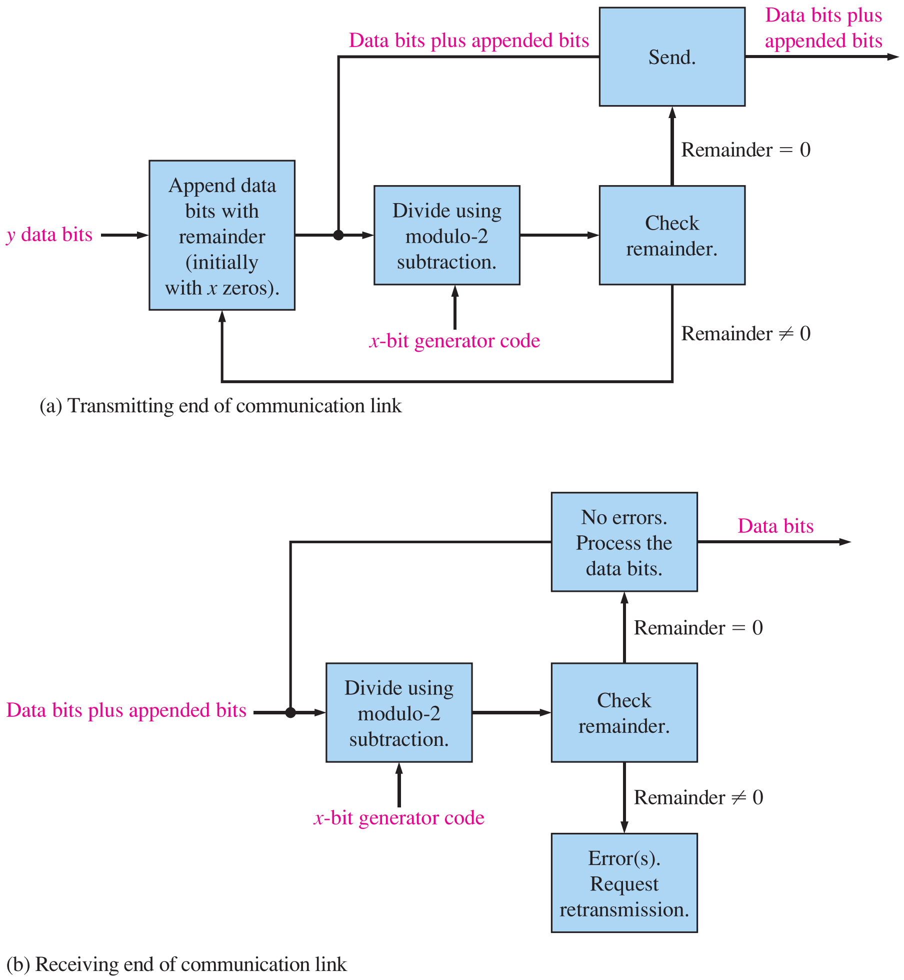
gate that will be introduced in Chapter 3. A simple rule for modulo-2 is that the output is 1 if the inputs are different; otherwise, it is 0.

CRC Process

The process is as follows:

1. Select a fixed generator code; it can have fewer bits than the data bits to be checked. This code is understood in advance by both the sending and receiving devices and must be the same for both.
2. Append a number of 0s equal to the number of bits in the generator code to the data bits.
3. Divide the data bits including the appended bits by the generator code bits using modulo-2.
4. If the remainder is 0, the data and appended bits are sent as is.
5. If the remainder is not 0, the appended bits are made equal to the remainder bits in order to get a 0 remainder before data are sent.
6. At the receiving end, the receiver divides the incoming appended data bit code by the same generator code as used by the sender.
7. If the remainder is 0, there is no error detected (it is possible in rare cases for multiple errors to cancel). If the remainder is not 0, an error has been detected in the transmission and a retransmission is requested by the receiver.

Figure 2–8 illustrates the CRC process.


Data bits plus appended bits
Send.
Data bits plus appended bits
Remainder = 0
Append data bits with remainder (initially with x zeros).
y data bits
Divide using modulo-2 subtraction.
Check remainder.
x-bit generator code
Remainder ≠ 0
No errors. Process the data bits.
Data bits
Remainder = 0
Data bits plus appended bits
Divide using modulo-2 subtraction.
Check remainder.
x-bit generator code
Remainder ≠ 0
Error(s). Request retransmission.

(a) Transmitting end of communication link

(b) Receiving end of communication link

FIGURE 2–8 The CRC process.

EXAMPLE 2–41

Determine the transmitted CRC for the following byte of data (D) and generator code (G). Verify that the remainder is 0.

D: 11010011
G: 1010

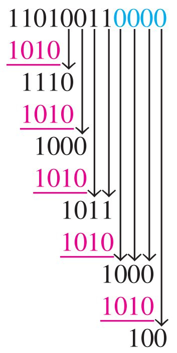
Solution

Since the generator code has four data bits, add four 0s (blue) to the data byte. The appended data (D′) is

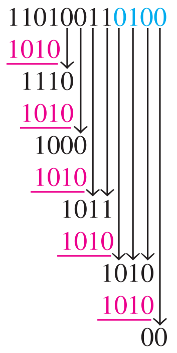
$$D' = 110100110000$$

Divide the appended data by the generator code (red) using the modulo-2 operation until all bits have been used.

$$\frac{D'}{G} = \frac{110100110000}{1010}$$



Remainder = 0100. Since the remainder is not 0, append the data with the four remainder bits (blue). Then divide by the generator code (red). The transmitted CRC is **110100110100**.



Remainder = 0

Related Problem

Change the generator code to 1100 and verify that a 0 remainder results when the CRC process is applied to the data byte (11010011).

EXAMPLE 2–42

During transmission, an error occurs in the second bit from the left in the appended data byte generated in Example 2–41. The received data is

$$D' = 100100110100$$

Apply the CRC process to the received data to detect the error using the same generator code (1010).

Solution

```
100100110100
1010
  1100
  1010
   1101
   1010
    1111
    1010
     1010
     1010
        0100
```

Remainder = 0100. Since it is not zero, an error is indicated.

Related Problem

Assume two errors in the data byte as follows: 10011011. Apply the CRC process to check for the errors using the same received data and the same generator code.

Hamming Code

The **Hamming code** is used to detect and correct a single-bit error in a transmitted code. To accomplish this, four redundancy bits are introduced in a 7-bit group of data bits. These redundancy bits are interspersed at bit positions 2^n ($n = 0, 1, 2, 3$) within the original data bits. At the end of the transmission, the redundancy bits have to be removed from the data bits. A recent version of the Hamming code places all the redundancy bits at the end of the data bits, making their removal easier than that of the interspersed bits. *A coverage of the classic Hamming code is available on the website.*

SECTION 2–12 CHECKUP

1. Which odd-parity code is in error?

   (a) 1011 **(b)** 1110 **(c)** 0101 **(d)** 1000

2. Which even-parity code is in error?

   (a) 11000110 **(b)** 00101000 **(c)** 10101010 **(d)** 11111011

3. Add an even parity bit to the end of each of the following codes.

   (a) 1010100 **(b)** 0100000 **(c)** 1110111 **(d)** 1000110

4. What does CRC stand for?

5. Apply modulo-2 operations to determine the following:

   (a) $1 + 1$ **(b)** $1 - 1$ **(c)** $1 - 0$ **(d)** $0 + 1$

SUMMARY

- A binary number is a weighted number in which the weight of each whole number digit is a positive power of two and the weight of each fractional digit is a negative power of two. The whole number weights increase from right to left—from least significant digit to most significant.
- A binary number can be converted to a decimal number by summing the decimal values of the weights of all the 1s in the binary number.
- A decimal whole number can be converted to binary by using the sum-of-weights or the repeated division-by-2 method.
- A decimal fraction can be converted to binary by using the sum-of-weights or the repeated multiplication-by-2 method.
- The basic rules for binary addition are as follows:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 10 \end{aligned}$$

- The basic rules for binary subtraction are as follows:

$$\begin{aligned} 0 - 0 &= 0 \\ 1 - 1 &= 0 \\ 1 - 0 &= 1 \\ 10 - 1 &= 1 \end{aligned}$$

- The 1's complement of a binary number is derived by changing 1s to 0s and 0s to 1s.
- The 2's complement of a binary number can be derived by adding 1 to the 1's complement.
- Binary subtraction can be accomplished with addition by using the 1's or 2's complement method.
- A positive binary number is represented by a 0 sign bit.
- A negative binary number is represented by a 1 sign bit.
- For arithmetic operations, negative binary numbers are represented in 1's complement or 2's complement form.
- In an addition operation, an overflow is possible when both numbers are positive or when both numbers are negative. An incorrect sign bit in the sum indicates the occurrence of an overflow.
- The hexadecimal number system consists of 16 digits and characters, 0 through 9 followed by A through F.
- One hexadecimal digit represents a 4-bit binary number, and its primary usefulness is in simplifying bit patterns and making them easier to read.
- A decimal number can be converted to hexadecimal by the repeated division-by-16 method.
- The octal number system consists of eight digits, 0 through 7.
- A decimal number can be converted to octal by using the repeated division-by-8 method.
- Octal-to-binary conversion is accomplished by simply replacing each octal digit with its 3-bit binary equivalent. The process is reversed for binary-to-octal conversion.
- A decimal number is converted to BCD by replacing each decimal digit with the appropriate 4-bit binary code.
- The ASCII is a 7-bit alphanumeric code that is used in computer systems for input and output of information.
- A parity bit is used to detect an error in a code.
- The CRC (cyclic redundancy check) is based on polynomial division using modulo-2 operations.

KEY TERMS

Key terms and other bold terms in the chapter are defined in the end-of-book glossary.

Alphanumeric Consisting of numerals, letters, and other characters.

ASCII American Standard Code for Information Interchange; the most widely used alphanumeric code.

BCD Binary coded decimal; a digital code in which each of the decimal digits, 0 through 9, is represented by a group of four bits.

Byte A group of eight bits.

Cyclic redundancy check (CRC) A type of error detection code.

Floating-point number A number representation based on scientific notation in which the number consists of an exponent and a mantissa.

Hexadecimal Describes a number system with a base of 16.

LSB Least significant bit; the right-most bit in a binary whole number or code.

MSB Most significant bit; the left-most bit in a binary whole number or code.

Octal Describes a number system with a base of eight.

Parity In relation to binary codes, the condition of evenness or oddness of the number of 1s in a code group.

TRUE/FALSE QUIZ

Answers are at the end of the chapter.

1. The octal number system is a weighted system with eight digits.

2. The binary number system is a weighted system with two digits.

3. MSB stands for most significant bit.

4. In hexadecimal, $9 + 1 = 10$.

5. The 1's complement of the binary number 1010 is 0101.

6. The 2's complement of the binary number 1111 is 0000.

7. The right-most bit in a signed binary number is the sign bit.

8. The hexadecimal number system has 16 characters, six of which are alphabetic characters.

9. BCD stands for binary coded decimal.

10. An error in a given code can be detected by verifying the parity bit.

11. CRC stands for cyclic redundancy check.

12. The modulo-2 sum of 11 and 10 is 100.

SELF-TEST

Answers are at the end of the chapter.

1. $3 \times 10^1 + 4 \times 10^0$ is
(a) 0.34 **(b)** 3.4 **(c)** 34 **(d)** 340

2. The decimal equivalent of 1000 is
(a) 2 **(b)** 4 **(c)** 6 **(d)** 8

3. The binary number 11011101 is equal to the decimal number
(a) 121 **(b)** 221 **(c)** 441 **(d)** 256

4. The decimal number 21 is equivalent to the binary number
(a) 10101 **(b)** 10001 **(c)** 10000 **(d)** 11111

5. The decimal number 250 is equivalent to the binary number
(a) 11111010 **(b)** 11110110 **(c)** 11111000 **(d)** 11111011

6. The sum of $1111 + 1111$ in binary equals
(a) 0000 **(b)** 2222 **(c)** 11110 **(d)** 11111

7. The difference of 1000 − 100 equals
(a) 100 **(b)** 101 **(c)** 110 **(d)** 111

8. The 1's complement of 11110000 is
(a) 11111111 **(b)** 11111110 **(c)** 00001111 **(d)** 10000001

9. The 2's complement of 11001100 is
(a) 00110011 **(b)** 00110100 **(c)** 00110101 **(d)** 00110110

10. The decimal number +122 is expressed in the 2's complement form as
(a) 01111010 **(b)** 11111010 **(c)** 01000101 **(d)** 10000101

11. The decimal number −34 is expressed in the 2's complement form as
(a) 01011110 **(b)** 10100010 **(c)** 11011110 **(d)** 01011101

12. A single-precision floating-point binary number has a total of
(a) 8 bits **(b)** 16 bits **(c)** 24 bits **(d)** 32 bits

13. In the 2's complement form, the binary number 10010011 is equal to the decimal number
(a) −19 **(b)** +109 **(c)** +91 **(d)** −109

14. The binary number 101100111001010100001 can be written in octal as
(a) 5471230_8 **(b)** 5471241_8 **(c)** 2634521_8 **(d)** 23162501_8

15. The binary number 10001101010001101111 can be written in hexadecimal as
(a) $AD467_{16}$ **(b)** $8C46F_{16}$ **(c)** $8D46F_{16}$ **(d)** $AE46F_{16}$

16. The binary number for $F7A9_{16}$ is
(a) 1111011110101001 **(b)** 1110111110101001
(c) 1111111010110001 **(d)** 1111011010101001

17. The BCD number for decimal 473 is
(a) 111011010 **(b)** 110001110011 **(c)** 010001110011 **(d)** 010011110011

18. Refer to Table 2–7. The command STOP in ASCII is
(a) 1010011101010010011111010000 **(b)** 1010010100110010011101010000
(c) 1001010110110110011101010001 **(d)** 1010011101010010011101100100

19. The code that has an even-parity error is
(a) 1010011 **(b)** 1101000 **(c)** 1001000 **(d)** 1110111

20. In the cyclic redundancy check, the absence of errors is indicated by
(a) Remainder = generator code **(b)** Remainder = 0
(c) Remainder = 1 **(d)** Quotient = 0

PROBLEMS

Answers to odd-numbered problems are at the end of the book.

Section 2–1 Decimal Numbers

1. What is the weight of 7 in each of the following decimal numbers?
(a) 1947 **(b)** 1799 **(c)** 1979

2. Express each of the following decimal numbers as a power of ten:
(a) 1000 **(b)** 10000000 **(c)** 1000000000

3. Give the value of each digit in the following decimal numbers:
(a) 263 **(b)** 5436 **(c)** 234543

4. How high can you count with six decimal digits?

Section 2–2 Binary Numbers

5. Convert the following binary numbers to decimal:
(a) 001 **(b)** 010 **(c)** 101 **(d)** 110
(e) 1010 **(f)** 1011 **(g)** 1110 **(h)** 1111

6. Convert the following binary numbers into decimal:
(a) 100001 **(b)** 100111 **(c)** 101010 **(d)** 111001
(e) 1100000 **(f)** 11111101 **(g)** 11110010 **(h)** 11111111

7. Convert each binary number to decimal:

(a) 110011.11 **(b)** 101010.01 **(c)** 1000001.111
(d) 1111000.101 **(e)** 1011100.10101 **(f)** 1110001.0001
(g) 1011010.1010 **(h)** 1111111.11111

8. What is the highest decimal number that can be represented by each of the following numbers of binary digits (bits)?

(a) two **(b)** three **(c)** four **(d)** five **(e)** six
(f) seven **(g)** eight **(h)** nine **(i)** ten **(j)** eleven

9. How many bits are required to represent the following decimal numbers?

(a) 5 **(b)** 10 **(c)** 15 **(d)** 20
(e) 100 **(f)** 120 **(g)** 140 **(h)** 160

10. Generate the binary sequence for each decimal sequence:

(a) 0 through 7 **(b)** 8 through 15 **(c)** 16 through 31
(d) 32 through 63 **(e)** 64 through 75

Section 2–3 Decimal-to-Binary Conversion

11. Convert each decimal number to binary by using the sum-of-weights method:

(a) 12 **(b)** 15 **(c)** 25 **(d)** 50
(e) 65 **(f)** 97 **(g)** 127 **(h)** 198

12. Convert each decimal fraction to binary using the sum-of-weights method:

(a) 0.26 **(b)** 0.762 **(c)** 0.0975

13. Convert each decimal number to binary using repeated division by 2:

(a) 13 **(b)** 17 **(c)** 23 **(d)** 30
(e) 35 **(f)** 40 **(g)** 49 **(h)** 60

14. Convert each decimal fraction to binary using repeated multiplication by 2:

(a) 0.76 **(b)** 0.456 **(c)** 0.8732

Section 2–4 Binary Arithmetic

15. Add the binary numbers:

(a) 10 + 10 **(b)** 10 + 11 **(c)** 100 + 11
(d) 111 + 101 **(e)** 1111 + 111 **(f)** 1111 + 1111

16. Use direct subtraction on the following binary numbers:

(a) 10 − 1 **(b)** 100 − 11 **(c)** 110 − 100
(d) 1111 − 11 **(e)** 1101 − 101 **(f)** 110000 − 1111

17. Perform the following binary multiplications:

(a) 11 × 10 **(b)** 101 × 11 **(c)** 111 × 110
(d) 1100 × 101 **(e)** 1110 × 1110 **(f)** 1111 × 1100

18. Divide the binary numbers as indicated:

(a) 110 ÷ 11 **(b)** 1010 ÷ 10 **(c)** 1111 ÷ 101

Section 2–5 Complements of Binary Numbers

19. What are two ways of representing zero in 1's complement form?

20. How is zero represented in 2's complement form?

21. Determine the 1's complement of each binary number:

(a) 100 **(b)** 111 **(c)** 1100
(d) 10111011 **(e)** 1001010 **(f)** 10101010

22. Determine the 2's complement of each binary number using either method:

(a) 11 **(b)** 110 **(c)** 1010 **(d)** 1001
(e) 101010 **(f)** 11001 **(g)** 11001100 **(h)** 11000111

Section 2–6 Signed Numbers

23. Express each decimal number in binary as an 8-bit sign-magnitude number:

(a) +29 **(b)** −85 **(c)** +100 **(d)** −123

24. Express each decimal number as an 8-bit number in the 1's complement form:

(a) −34 **(b)** +57 **(c)** −99 **(d)** +115

25. Express each decimal number as an 8-bit number in the 2's complement form:

(a) +12 **(b)** −68 **(c)** +101 **(d)** −125

26. Determine the decimal value of each signed binary number in the sign-magnitude form:

(a) 10011001 **(b)** 01110100 **(c)** 10111111

27. Determine the decimal value of each signed binary number in the 1's complement form:

(a) 10011001 **(b)** 01110100 **(c)** 10111111

28. Determine the decimal value of each signed binary number in the 2's complement form:

(a) 10011001 **(b)** 01110100 **(c)** 10111111

29. Express each of the following sign-magnitude binary numbers in single-precision floating-point format:

(a) 0111110000101011 **(b)** 100110000011000

30. Determine the values of the following single-precision floating-point numbers:

(a) 1 10000001 01001001110001000000000
(b) 0 11001100 10000111110100100000000

Section 2–7 Arithmetic Operations with Signed Numbers

31. Convert each pair of decimal numbers to binary and add using the 2's complement form:

(a) 33 and 15 **(b)** 56 and −27 **(c)** −46 and 25 **(d)** −110 and −84

32. Perform each addition in the 2's complement form:

(a) 00010110 + 00110011 **(b)** 01110000 + 10101111

33. Perform each addition in the 2's complement form:

(a) 10001100 + 00111001 **(b)** 11011001 + 11100111

34. Perform each subtraction in the 2's complement form:

(a) 00110011 − 00010000 **(b)** 01100101 − 11101000

35. Multiply 01101010 by 11110001 in the 2's complement form.

36. Divide 10001000 by 00100010 in the 2's complement form.

Section 2–8 Hexadecimal Numbers

37. Convert each hexadecimal number to binary:

(a) 46_{16} **(b)** 54_{16} **(c)** $B4_{16}$ **(d)** $1A3_{16}$
(e) FA_{16} **(f)** ABC_{16} **(g)** $ABCD_{16}$

38. Convert each binary number to hexadecimal:

(a) 1111 **(b)** 1011 **(c)** 11111
(d) 10101010 **(e)** 10101100 **(f)** 10111011

39. Convert each hexadecimal number to decimal:

(a) 42_{16} **(b)** 64_{16} **(c)** $2B_{16}$ **(d)** $4D_{16}$
(e) FF_{16} **(f)** BC_{16} **(g)** $6F1_{16}$ **(h)** ABC_{16}

40. Convert each decimal number to hexadecimal:

(a) 10 **(b)** 15 **(c)** 32 **(d)** 54
(e) 365 **(f)** 3652 **(g)** 7825 **(h)** 8925

41. Perform the following additions:

(a) $25_{16} + 33_{16}$ **(b)** $43_{16} + 62_{16}$ **(c)** $A4_{16} + F5_{16}$ **(d)** $FC_{16} + AE_{16}$

42. Perform the following subtractions:

(a) $60_{16} - 39_{16}$ **(b)** $A5_{16} - 98_{16}$ **(c)** $F1_{16} - A6_{16}$ **(d)** $AC_{16} - 10_{16}$

Section 2–9 Octal Numbers

43. Convert each octal number to decimal:

(a) 14_8 **(b)** 53_8 **(c)** 67_8 **(d)** 174_8
(e) 635_8 **(f)** 254_8 **(g)** 2673_8 **(h)** 7777_8

44. Convert each decimal number to octal by repeated division by 8:

(a) 23 **(b)** 45 **(c)** 65 **(d)** 84
(e) 124 **(f)** 156 **(g)** 654 **(h)** 9999

45. Convert each octal number into binary:

(a) 17_8 **(b)** 26_8 **(c)** 145_8 **(d)** 456_8
(e) 653_8 **(f)** 777_8

46. Convert each binary number to octal:

(a) 100 **(b)** 110 **(c)** 1100
(d) 1111 **(e)** 11001 **(f)** 11110
(g) 110011 **(h)** 101010 **(i)** 10101111

Section 2–10 Binary Coded Decimal (BCD)

47. Convert each of the following decimal numbers to 8421 BCD:

(a) 10 **(b)** 13 **(c)** 18 **(d)** 21 **(e)** 25 **(f)** 36
(g) 44 **(h)** 57 **(i)** 69 **(j)** 98 **(k)** 125 **(l)** 156

48. Convert each of the decimal numbers in Problem 47 to straight binary, and compare the number of bits required with that required for BCD.

49. Convert the following decimal numbers to BCD:

(a) 104 **(b)** 128 **(c)** 132 **(d)** 150 **(e)** 186
(f) 210 **(g)** 359 **(h)** 547 **(i)** 1051

50. Convert each of the BCD numbers to decimal:

(a) 0001 **(b)** 0110 **(c)** 1001
(d) 00011000 **(e)** 00011001 **(f)** 00110010
(g) 01000101 **(h)** 10011000 **(i)** 100001110000

51. Convert each of the BCD numbers to decimal:

(a) 10000000 **(b)** 001000110111
(c) 001101000110 **(d)** 010000100001
(e) 011101010100 **(f)** 100000000000
(g) 100101111000 **(h)** 0001011010000011
(i) 1001000000011000 **(j)** 0110011001100111

52. Add the following BCD numbers:

(a) 0010 + 0001 **(b)** 0101 + 0011
(c) 0111 + 0010 **(d)** 1000 + 0001
(e) 00011000 + 00010001 **(f)** 01100100 + 00110011
(g) 01000000 + 01000111 **(h)** 10000101 + 00010011

53. Add the following BCD numbers:

(a) 1000 + 0110 **(b)** 0111 + 0101
(c) 1001 + 1000 **(d)** 1001 + 0111
(e) 00100101 + 00100111 **(f)** 01010001 + 01011000
(g) 10011000 + 10010111 **(h)** 010101100001 + 011100001000

54. Convert each pair of decimal numbers to BCD, and add as indicated:

(a) 4 + 3 **(b)** 5 + 2 **(c)** 6 + 4 **(d)** 17 + 12
(e) 28 + 23 **(f)** 65 + 58 **(g)** 113 + 101 **(h)** 295 + 157

Section 2–11 Digital Codes

55. In a certain application a 4-bit binary sequence cycles from 1111 to 0000 periodically. There are four bit changes, and because of circuit delays, these changes may not occur at the same instant. For example, if the LSB changes first, the number will appear as 1110 during the transition from 1111 to 0000 and may be misinterpreted by the system. Illustrate how the Gray code avoids this problem.

56. Convert each binary number to Gray code:

(a) 11011 **(b)** 1001010 **(c)** 1111011101110

57. Convert each Gray code to binary:

(a) 1010 **(b)** 00010 **(c)** 11000010001

58. Convert each of the following decimal numbers to ASCII. Refer to Table 2–7.

(a) 1 **(b)** 3 **(c)** 6 **(d)** 10 **(e)** 18
(f) 29 **(g)** 56 **(h)** 75 **(i)** 107

59. Determine each ASCII character. Refer to Table 2–7.

(a) 0011000 **(b)** 1001010 **(c)** 0111101

(d) 0100011 **(e)** 0111110 **(f)** 1000010

60. Decode the following ASCII coded message:

1001000 1100101 1101100 1101100 1101111 0101110
0100000 1001000 1101111 1110111 0100000 1100001
1110010 1100101 0100000 1111001 1101111 1110101
0111111

61. Write the message in Problem 60 in hexadecimal.

62. Convert the following statement to ASCII:

30 INPUT A, B

Section 2–12 Error Codes

63. Determine which of the following even parity codes are in error:

(a) 100110010 **(b)** 011101010 **(c)** 10111111010001010

64. Determine which of the following odd parity codes are in error:

(a) 11110110 **(b)** 00110001 **(c)** 01010101010101010

65. Attach the proper even parity bit to each of the following bytes of data:

(a) 10100100 **(b)** 00001001 **(c)** 11111110

66. Apply modulo-2 to the following:

(a) 1100 + 1011 **(b)** 1111 + 0100 **(c)** 10011001 + 100011100

67. Verify that modulo-2 subtraction is the same as modulo-2 addition by adding the result of each operation in problem 66 to either of the original numbers to get the other number. This will show that the result is the same as the difference of the two numbers.

68. Apply CRC to the data bits 10110010 using the generator code 1010 to produce the transmitted CRC code.

69. Assume that the code produced in problem 68 incurs an error in the most significant bit during transmission. Apply CRC to detect the error.

ANSWERS

SECTION CHECKUPS

Section 2–1 Decimal Numbers

1. **(a)** 1370: 10 **(b)** 6725: 100 **(c)** 7051: 1000 **(d)** 58.72: 0.1

2. **(a)** 51 = (5 × 10) + (1 × 1)

(b) 137 = (1 × 100) + (3 × 10) + (7 × 1)

(c) 1492 = (1 × 1000) + (4 × 100) + (9 × 10) + (2 × 1)

(d) 106.58 = (1 × 100) + (0 × 10) + (6 × 1) + (5 × 0.1) + (8 × 0.01)

Section 2–2 Binary Numbers

1. $2^8 - 1 = 255$

2. Weight is 16.

3. $10111101.011 = 189.375$

Section 2–3 Decimal-to-Binary Conversion

1. **(a)** $23 = 10111$ **(b)** $57 = 111001$ **(c)** $45.5 = 101101.1$

2. **(a)** $14 = 1110$ **(b)** $21 = 10101$ **(c)** $0.375 = 0.011$

Section 2–4 Binary Arithmetic

1. **(a)** $1101 + 1010 = 10111$ **(b)** $10111 + 01101 = 100100$

2. **(a)** $1101 - 0100 = 1001$ **(b)** $1001 - 0111 = 0010$

3. **(a)** $110 \times 111 = 101010$ **(b)** $1100 \div 011 = 100$

Section 2–5 Complements of Binary Numbers

1. **(a)** 1's comp of 00011010 = 11100101 **(b)** 1's comp of 11110111 = 00001000

(c) 1's comp of 10001101 = 01110010

2. **(a)** 2's comp of 00010110 = 11101010 **(b)** 2's comp of 11111100 = 00000100

(c) 2's comp of 10010001 = 01101111

Section 2–6 Signed Numbers

1. Sign-magnitude: $+9 = 00001001$

2. 1's comp: $-33 = 11011110$

3. 2's comp: $-46 = 11010010$

4. Sign bit, exponent, and mantissa

Section 2–7 Arithmetic Operations with Signed Numbers

1. Cases of addition: positive number is larger, negative number is larger, both are positive, both are negative

2. $00100001 + 10111100 = 11011101$

3. $01110111 - 00110010 = 01000101$

4. Sign of product is positive.

5. $00000101 \times 01111111 = 01001111011$

6. Sign of quotient is negative.

7. $00110000 \div 00001100 = 00000100$

Section 2–8 Hexadecimal Numbers

1. **(a)** $10110011 = \text{B3}_{16}$ **(b)** $110011101000 = \text{CE8}_{16}$

2. **(a)** $57_{16} = 01010111$ **(b)** $\text{3A5}_{16} = 001110100101$

(c) $\text{F8OB}_{16} = 1111100000001011$

3. $\text{9B30}_{16} = 39{,}728_{10}$

4. $573_{10} = \text{23D}_{16}$

5. **(a)** $18_{16} + 34_{16} = \text{4C}_{16}$ **(b)** $\text{3F}_{16} + \text{2A}_{16} = 69_{16}$

6. **(a)** $75_{16} - 21_{16} = 54_{16}$ **(b)** $94_{16} - \text{5C}_{16} = 38_{16}$

Section 2–9 Octal Numbers

1. **(a)** $73_8 = 59_{10}$ **(b)** $125_8 = 85_{10}$

2. **(a)** $98_{10} = 142_8$ **(b)** $163_{10} = 243_8$

3. **(a)** $46_8 = 100110$ **(b)** $723_8 = 111010011$ **(c)** $5624_8 = 101110010100$

4. **(a)** $110101111 = 657_8$ **(b)** $1001100010 = 1142_8$ **(c)** $10111111001 = 2771_8$

Section 2–10 Binary Coded Decimal (BCD)

1. **(a)** 0010: 2 **(b)** 1000: 8 **(c)** 0001: 1 **(d)** 0100: 4

2. **(a)** $6_{10} = 0110$ **(b)** $15_{10} = 00010101$ **(c)** $273_{10} = 001001110011$
(d) $849_{10} = 100001001001$

3. **(a)** $10001001 = 89_{10}$ **(b)** $001001111000 = 278_{10}$ **(c)** $000101010111 = 157_{10}$

4. A 4-bit sum is invalid when it is greater than 9_{10}.

Section 2–11 Digital Codes

1. **(a)** $1100_2 = 1010$ Gray **(b)** $1010_2 = 1111$ Gray **(c)** $11010_2 = 10111$ Gray

2. **(a)** 1000 Gray $= 1111_2$ **(b)** 1010 Gray $= 1100_2$ **(c)** 11101 Gray $= 10110_2$

3. **(a)** K: $1001011 \rightarrow 4B_{16}$ **(b)** r: $1110010 \rightarrow 72_{16}$
(c) \$: $0100100 \rightarrow 24_{16}$ **(d)** +: $0101011 \rightarrow 2B_{16}$

Section 2–12 Error Codes

1. **(c)** 0101 has an error.

2. **(d)** 11111011 has an error.

3. **(a)** 10101001 **(b)** 01000001 **(c)** 11101110 **(d)** 10001101

4. Cyclic redundancy check

5. **(a)** 0 **(b)** 0 **(c)** 1 **(d)** 1

RELATED PROBLEMS FOR EXAMPLES

2–1 9 has a value of 900, 3 has a value of 30, 9 has a value of 9.

2–2 6 has a value of 60, 7 has a value of 7, 9 has a value of 9/10 (0.9), 2 has a value of 2/100 (0.02), 4 has a value of 4/1000 (0.004).

2–3 $10010001 = 128 + 16 + 1 = 145$

2–4 $10.111 = 2 + 0.5 + 0.25 + 0.125 = 2.875$

2–5 $125 = 64 + 32 + 16 + 8 + 4 + 1 = 1111101$

2–6 $39 = 100111$

2–7 $1111 + 1100 = 11011$

2–8 $111 - 100 = 011$

2–9 $110 - 101 = 001$

2–10 $1101 \times 1010 = 10000010$

2–11 $1100 \div 100 = 11$

2–12 00110101

2–13 01000000

2–14 See Table 2–10.

TABLE 2–10

	Sign-Magnitude	1's Comp	2's Comp
+19	00010011	00010011	00010011
−19	10010011	11101100	11101101

2–15 $01110111 = +119_{10}$

2–16 $11101011 = -20_{10}$

2–17 $11010111 = -41_{10}$

2–18 11000010001010011000000000

2–19 01010101

2–20 00010001

2–21 1001000110

2–22 (83)(−59) = −4897 (10110011011111 in 2's comp)

2–23 100 ÷ 25 = 4 (0100)

2–24 $4F79C_{16}$

2–25 0110101111010011_2

2–26 $6BD_{16} = 011010111101 = 2^{10} + 2^9 + 2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$
$= 1024 + 512 + 128 + 32 + 16 + 8 + 4 + 1 = 1725_{10}$

2–27 $60A_{16} = (6 \times 256) + (0 \times 16) + (10 \times 1) = 1546_{10}$

2–28 $2591_{10} = A1F_{16}$

2–29 $4C_{16} + 3A_{16} = 86_{16}$

2–30 $BCD_{16} - 173_{16} = A5A_{16}$

2–31 **(a)** $001011_2 = 11_{10} = 13_8$ **(b)** $010101_2 = 21_{10} = 25_8$
(c) $001100000_2 = 96_{10} = 140_8$ **(d)** $111101010110_2 = 3926_{10} = 7526_8$

2–32 1250762_8

2–33 1001011001110011

2–34 $82{,}276_{10}$

2–35 1001100101101000

2–36 10000010

2–37 **(a)** 111011 (Gray) **(b)** 111010_2

2–38 The sequence of codes for if (y < 8) is $69_{16}66_{16}20_{16}28_{16}79_{16}3C_{16}38_{16}29_{16}$

2–39 01001011

2–40 Yes

2–41 A 0 remainder results

2–42 Errors are indicated.

TRUE/FALSE QUIZ

1. T **2.** T **3.** T **4.** F **5.** T **6.** F **7.** F **8.** T **9.** T **10.** T
11. T **12.** F

SELF-TEST

1. (c) **2.** (d) **3.** (b) **4.** (a) **5.** (a) **6.** (c) **7.** (a) **8.** (c)
9. (b) **10.** (a) **11.** (c) **12.** (d) **13.** (d) **14.** (b) **15.** (c) **16.** (a)
17. (c) **18.** (a) **19.** (b) **20.** (b)

CHAPTER 3

Logic Gates

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Describe the operation of the inverter, the AND gate, and the OR gate
- Describe the operation of the NAND gate and the NOR gate
- Express the operation of NOT, AND, OR, NAND, and NOR gates with Boolean algebra
- Describe the operation of the exclusive-OR and exclusive-NOR gates
- Use logic gates in simple applications
- Recognize and use both the distinctive shape logic gate symbols and the rectangular outline logic gate symbols of ANSI/IEEE Standard 91-1984/Std. 91a-1991
- Construct timing diagrams showing the proper time relationships of inputs and outputs for the various logic gates
- Discuss the basic concepts of programmable logic
- Make basic comparisons between the major IC technologies—CMOS and bipolar (TTL)
- Explain how the different series within the CMOS and bipolar (TTL) families differ from each other
- Define *propagation delay time, power dissipation, speed-power product,* and *fan-out* in relation to logic gates
- List specific fixed-function integrated circuit devices that contain the various logic gates
- Troubleshoot logic gates for opens and shorts by using the oscilloscope

KEY TERMS

Key terms are in order of appearance in the chapter.

- Inverter
- Truth table
- Boolean algebra
- Complement
- AND gate
- OR gate
- NAND gate
- NOR gate
- Exclusive-OR gate
- Exclusive-NOR gate
- AND array
- Fuse
- Antifuse
- EPROM
- EEPROM
- Flash
- SRAM
- Target device
- JTAG
- VHDL
- CMOS
- Bipolar
- Propagation delay time
- Fan-out
- Unit load

VISIT THE WEBSITE

Study aids for this chapter are available at http://www.pearsonglobaleditions.com/floyd

INTRODUCTION

The emphasis in this chapter is on the operation, application, and troubleshooting of logic gates. The relationship of input and output waveforms of a gate using timing diagrams is thoroughly covered.

Logic symbols used to represent the logic gates are in accordance with ANSI/IEEE Standard 91-1984/Std. 91a-1991. This standard has been adopted by private industry and the military for use in internal documentation as well as published literature.

Both fixed-function logic and programmable logic are discussed in this chapter. Because integrated circuits (ICs) are used in all applications, the logic function of a device is generally of greater importance to the technician or technologist than the details of the component-level circuit operation within the IC package. Therefore, detailed coverage of the devices at the component level can be treated as an optional topic. Digital integrated circuit technologies are discussed in Chapter 15 on the website, all or parts of which may be introduced at appropriate points throughout the text.

Suggestion: Review Section 1–3 before you start this chapter.

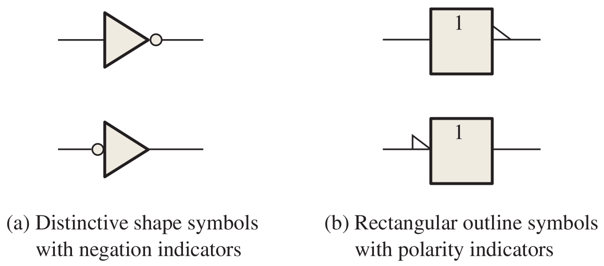
3–1 The Inverter

The inverter (NOT circuit) performs the operation called *inversion* or *complementation.* The inverter changes one logic level to the opposite level. In terms of bits, it changes a 1 to a 0 and a 0 to a 1.

After completing this section, you should be able to

- Identify negation and polarity indicators
- Identify an inverter by either its distinctive shape symbol or its rectangular outline symbol
- Produce the truth table for an inverter
- Describe the logical operation of an inverter

Standard logic symbols for the **inverter** are shown in Figure 3–1. Part (a) shows the *distinctive shape* symbols, and part (b) shows the *rectangular outline* symbols. In this textbook, distinctive shape symbols are generally used; however, the rectangular outline symbols are found in many industry publications, and you should become familiar with them as well. (Logic symbols are in accordance with **ANSI/IEEE** Standard 91-1984 and its supplement Standard 91a-1991.)



(a) Distinctive shape symbols with negation indicators

(b) Rectangular outline symbols with polarity indicators

FIGURE 3–1 Standard logic symbols for the inverter (ANSI/IEEE Std. 91-1984/ Std. 91a-1991).

The Negation and Polarity Indicators

The negation indicator is a "bubble" (○) that indicates **inversion** or *complementation* when it appears on the input or output of any logic element, as shown in Figure 3–1(a) for the inverter. Generally, inputs are on the left of a logic symbol and the output is on the right. When appearing on the input, the bubble means that a 0 is the active or *asserted* input state, and the input is called an active-LOW input. When appearing on the output, the bubble means that a 0 is the active or asserted output state, and the output is called an active-LOW output. The absence of a bubble on the input or output means that a 1 is the active or asserted state, and in this case, the input or output is called active-HIGH.

The polarity or level indicator is a "triangle" (◺) that indicates inversion when it appears on the input or output of a logic element, as shown in Figure 3–1(b). When appearing on the input, it means that a LOW level is the active or asserted input state. When appearing on the output, it means that a LOW level is the active or asserted output state.

Either indicator (bubble or triangle) can be used both on distinctive shape symbols and on rectangular outline symbols. Figure 3–1(a) indicates the principal inverter symbols used in this text. Note that a change in the placement of the negation or polarity indicator does not imply a change in the way an inverter operates.

Inverter Truth Table

When a HIGH level is applied to an inverter input, a LOW level will appear on its output. When a LOW level is applied to its input, a HIGH will appear on its output. This operation is summarized in Table 3–1, which shows the output for each possible input in terms of levels and corresponding bits. A table such as this is called a **truth table**.

TABLE 3–1

Inverter truth table.

Input	**Output**
LOW (0)	HIGH (1)
HIGH (1)	LOW (0)

Inverter Operation

Figure 3–2 shows the output of an inverter for a pulse input, where t_1 and t_2 indicate the corresponding points on the input and output pulse waveforms.

> **When the input is LOW, the output is HIGH; when the input is HIGH, the output is LOW, thereby producing an inverted output pulse.**

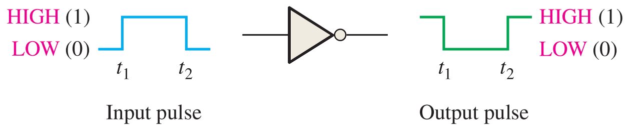


FIGURE 3–2 Inverter operation with a pulse input. Open file F03-02 to verify inverter operation. *A Multisim tutorial is available on the website.*

MultiSim

Timing Diagrams

Recall from Chapter 1 that a *timing diagram* is basically a graph that accurately displays the relationship of two or more waveforms with respect to each other on a time basis. For example, the time relationship of the output pulse to the input pulse in Figure 3–2 can be shown with a simple timing diagram by aligning the two pulses so that the occurrences of the pulse edges appear in the proper time relationship. The rising edge of the input pulse and the falling edge of the output pulse occur at the same time (ideally). Similarly, the falling edge of the input pulse and the rising edge of the output pulse occur at the same time (ideally). This timing relationship is shown in Figure 3–3. In practice, there is a very small delay from the input transition until the corresponding output transition. Timing diagrams are especially useful for illustrating the time relationship of digital waveforms with multiple pulses.

A timing diagram shows how two or more waveforms relate in time.

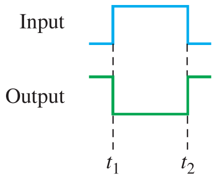


FIGURE 3–3 Timing diagram for the case in Figure 3–2.

EXAMPLE 3–1

A waveform is applied to an inverter in Figure 3–4. Determine the output waveform corresponding to the input and show the timing diagram. According to the placement of the bubble, what is the active output state?

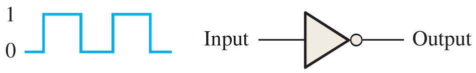


FIGURE 3–4

Solution

The output waveform is exactly opposite to the input (inverted), as shown in Figure 3–5, which is the basic timing diagram. The active or asserted output state is **0**.

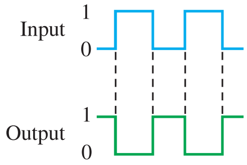


FIGURE 3–5

Related Problem*

If the inverter is shown with the negative indicator (bubble) on the input instead of the output, how is the timing diagram affected?

*Answers are at the end of the chapter.

Logic Expression for an Inverter

Boolean algebra uses variables and operators to describe a logic circuit.

In **Boolean algebra**, which is the mathematics of logic circuits and will be covered thoroughly in Chapter 4, a variable is generally designated by one or two letters although there can be more. Letters near the beginning of the alphabet usually designate inputs, while letters near the end of the alphabet usually designate outputs. The **complement** of a variable is designated by a bar over the letter. A variable can take on a value of either 1 or 0. If a given variable is 1, its complement is 0 and vice versa.

The operation of an inverter (NOT circuit) can be expressed as follows: If the input variable is called A and the output variable is called X, then

$$X = \overline{A}$$

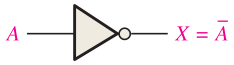


FIGURE 3–6 The inverter complements an input variable.

This expression states that the output is the complement of the input, so if $A = 0$, then $X = 1$, and if $A = 1$, then $X = 0$. Figure 3–6 illustrates this. The complemented variable $\overline{A}$ can be read as "A bar" or "not A."

An Application

Figure 3–7 shows a circuit for producing the 1's complement of an 8-bit binary number. The bits of the binary number are applied to the inverter inputs and the 1's complement of the number appears on the outputs.

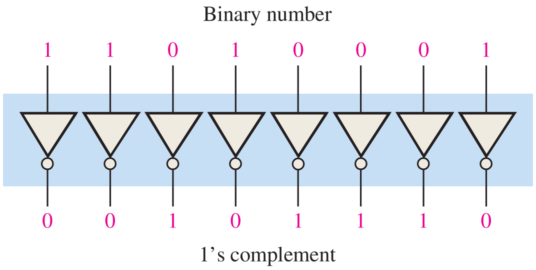


FIGURE 3–7 Example of a 1's complement circuit using inverters.

SECTION 3–1 CHECKUP

Answers are at the end of the chapter.

1. When a 1 is on the input of an inverter, what is the output?
2. An active-HIGH pulse (HIGH level when asserted, LOW level when not) is required on an inverter input.
   - **(a)** Draw the appropriate logic symbol, using the distinctive shape and the negation indicator, for the inverter in this application.
   - **(b)** Describe the output when a positive-going pulse is applied to the input of an inverter.

3–2 The AND Gate

The AND gate is one of the basic gates that can be combined to form any logic function. An AND gate can have two or more inputs and performs what is known as logical multiplication.

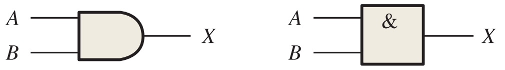
After completing this section, you should be able to

- Identify an AND gate by its distinctive shape symbol or by its rectangular outline symbol
- Describe the operation of an AND gate
- Generate the truth table for an AND gate with any number of inputs
- Produce a timing diagram for an AND gate with any specified input waveforms
- Write the logic expression for an AND gate with any number of inputs
- Discuss examples of AND gate applications

The term *gate* was introduced in Chapter 1 and is used to describe a circuit that performs a basic logic operation. The AND gate is composed of two or more inputs and a single output, as indicated by the standard logic symbols shown in Figure 3–8. Inputs are on the left, and the output is on the right in each symbol. Gates with two inputs are shown; however, an AND gate can have any number of inputs greater than one. Although examples of both distinctive shape symbols and rectangular outline symbols are shown, the distinctive shape symbol, shown in part (a), is used predominantly in this book.

InfoNote

Logic gates are one of the fundamental building blocks of digital systems. Most of the functions in a computer, with the exception of certain types of memory, are implemented with logic gates used on a very large scale. For example, a microprocessor, which is the main part of a computer, is made up of hundreds of thousands or even millions of logic gates.



(a) Distinctive shape

(b) Rectangular outline with the AND (&) qualifying symbol

FIGURE 3–8 Standard logic symbols for the AND gate showing two inputs (ANSI/IEEE Std. 91-1984/Std. 91a-1991).

Operation of an AND Gate

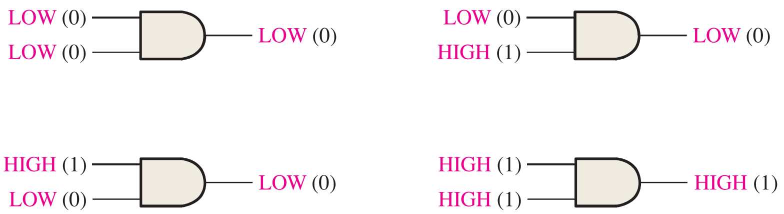
An **AND gate** produces a HIGH output *only* when *all* of the inputs are HIGH. When any of the inputs is LOW, the output is LOW. Therefore, the basic purpose of an AND gate is to determine when certain conditions are simultaneously true, as indicated by HIGH levels on all of its inputs, and to produce a HIGH on its output to indicate that all these conditions are

An AND gate can have more than two inputs.

true. The inputs of the 2-input AND gate in Figure 3–8 are labeled A and B, and the output is labeled X. The gate operation can be stated as follows:

For a 2-input AND gate, output X is HIGH only when inputs A and B are HIGH; X is LOW when either A or B is LOW, or when both A and B are LOW.

Figure 3–9 illustrates a 2-input AND gate with all four possibilities of input combinations and the resulting output for each.



MultiSim

FIGURE 3–9 All possible logic levels for a 2-input AND gate. Open file F03-09 to verify AND gate operation.

AND Gate Truth Table

For an AND gate, all HIGH inputs produce a HIGH output.

The logical operation of a gate can be expressed with a truth table that lists all input combinations with the corresponding outputs, as illustrated in Table 3–2 for a 2-input AND gate. The truth table can be expanded to any number of inputs. Although the terms HIGH and LOW tend to give a “physical” sense to the input and output states, the truth table is shown with 1s and 0s; a HIGH is equivalent to a 1 and a LOW is equivalent to a 0 in positive logic. For any AND gate, regardless of the number of inputs, the output is HIGH *only* when *all* inputs are HIGH.

TABLE 3–2

Truth table for a 2-input AND gate.

Inputs		Output
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

1 = HIGH, 0 = LOW

The total number of possible combinations of binary inputs to a gate is determined by the following formula:

$$N = 2^n \quad \text{Equation 3–1}$$

where N is the number of possible input combinations and n is the number of input variables. To illustrate,

For two input variables: $N = 2^2 = 4$ combinations

For three input variables: $N = 2^3 = 8$ combinations

For four input variables: $N = 2^4 = 16$ combinations

You can determine the number of input bit combinations for gates with any number of inputs by using Equation 3–1.

EXAMPLE 3–2

(a) Develop the truth table for a 3-input AND gate.

(b) Determine the total number of possible input combinations for a 4-input AND gate.

TABLE 3–3

Inputs			Output
A	B	C	X
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

Solution

(a) There are eight possible input combinations ($2^3 = 8$) for a 3-input AND gate. The input side of the truth table (Table 3–3) shows all eight combinations of three bits. The output side is all 0s except when all three input bits are 1s.

(b) $N = 2^4 = \mathbf{16}$. There are 16 possible combinations of input bits for a 4-input AND gate.

Related Problem

Develop the truth table for a 4-input AND gate.

AND Gate Operation with Waveform Inputs

In most applications, the inputs to a gate are not stationary levels but are voltage waveforms that change frequently between HIGH and LOW logic levels. Now let's look at the operation of AND gates with pulse waveform inputs, keeping in mind that an AND gate obeys the truth table operation regardless of whether its inputs are constant levels or levels that change back and forth.

Let's examine the waveform operation of an AND gate by looking at the inputs with respect to each other in order to determine the output level at any given time. In Figure 3–10, inputs A and B are both HIGH (1) during the time interval, t_1, making output X HIGH (1) during this interval. During time interval t_2, input A is LOW (0) and input B is HIGH (1), so the output is LOW (0). During time interval t_3, both inputs are HIGH (1) again, and therefore the output is HIGH (1). During time interval t_4, input A is HIGH (1) and input B is LOW (0), resulting in a LOW (0) output. Finally, during time interval t_5, input A is LOW (0), input B is LOW (0), and the output is therefore LOW (0). As you know, a diagram of input and output waveforms showing time relationships is called a *timing diagram.*

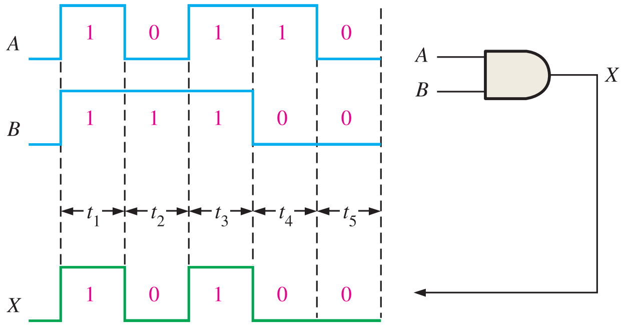


FIGURE 3–10 Example of AND gate operation with a timing diagram showing input and output relationships.

EXAMPLE 3–3

If two waveforms, A and B, are applied to the AND gate inputs as in Figure 3–11, what is the resulting output waveform?

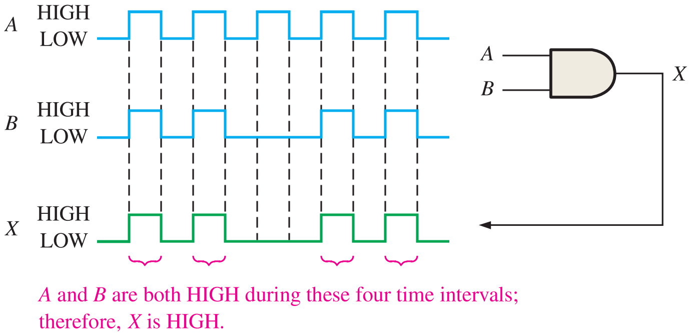


FIGURE 3–11

Solution

The output waveform X is HIGH only when both A and B waveforms are HIGH as shown in the timing diagram in Figure 3–11.

Related Problem

Determine the output waveform and show a timing diagram if the second and fourth pulses in waveform A of Figure 3–11 are replaced by LOW levels.

Remember, when analyzing the waveform operation of logic gates, it is important to pay careful attention to the time relationships of all the inputs with respect to each other and to the output.

EXAMPLE 3–4

For the two input waveforms, *A* and *B*, in Figure 3–12, show the output waveform with its proper relation to the inputs.

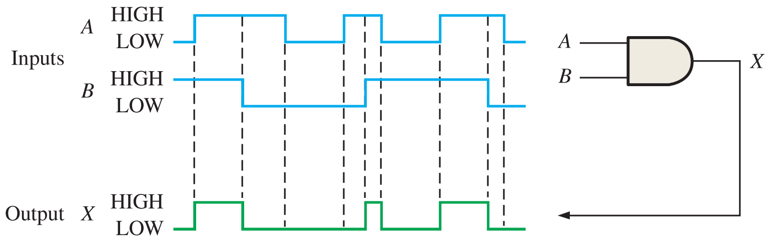


FIGURE 3–12

Solution

The output waveform is HIGH only when both of the input waveforms are HIGH as shown in the timing diagram.

Related Problem

Show the output waveform if the *B* input to the AND gate in Figure 3–12 is always HIGH.

EXAMPLE 3–5

For the 3-input AND gate in Figure 3–13, determine the output waveform in relation to the inputs.

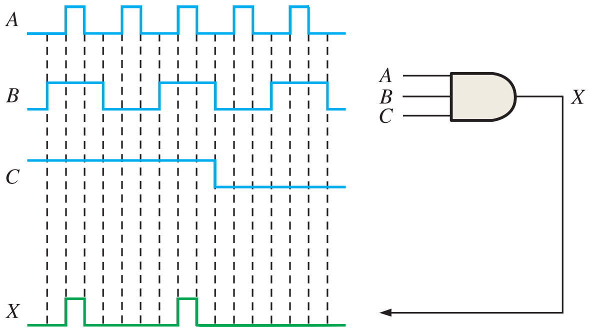


FIGURE 3–13

Solution

The output waveform *X* of the 3-input AND gate is HIGH only when all three input waveforms *A*, *B*, and *C* are HIGH.

Related Problem

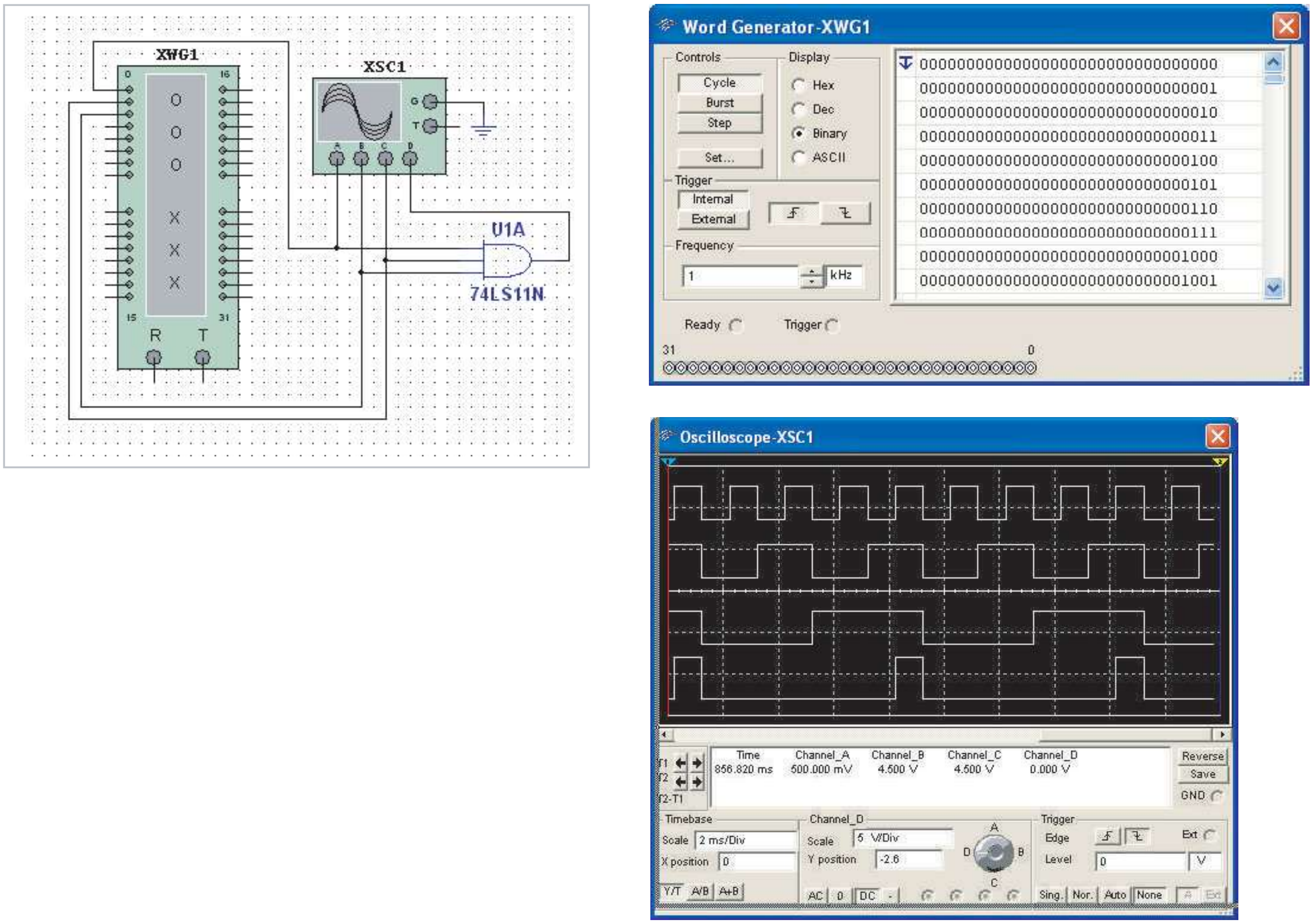
What is the output waveform of the AND gate in Figure 3–13 if the *C* input is always HIGH?

EXAMPLE 3–6

Use Multisim to simulate a 3-input AND gate with input waveforms that cycle through binary numbers 0 through 9.

Solution

Use the Multisim word generator in the up counter mode to provide the combination of waveforms representing the binary sequence, as shown in Figure 3–14. The first three waveforms on the oscilloscope display are the inputs, and the bottom waveform is the output.



Word Generator-XWG1: Display ● Binary; Frequency 1 kHz

```
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001000
00000000000000000000000000001001
```

Oscilloscope-XSC1:

Time	Channel_A	Channel_B	Channel_C	Channel_D
856.820 ms	500.000 mV	4.500 V	4.500 V	0.000 V

Timebase: Scale 2 ms/Div; X position 0

Channel_D: Scale 5 V/Div; Y position -2.6

Trigger: Level 0 V

FIGURE 3–14

Related Problem

Use Multisim software to create the setup and simulate the 3-input AND gate as illustrated in this example.

Logic Expressions for an AND Gate

The logical AND function of two variables is represented mathematically either by placing a dot between the two variables, as $A \cdot B$, or by simply writing the adjacent letters without the dot, as AB. We will normally use the latter notation.

InfoNote

Processors can utilize all of the basic logic operations when it is necessary to selectively manipulate certain bits in one or more bytes of data. Selective bit manipulations are done with a *mask.* For example, to clear (make all 0s) the right four bits in a data byte but keep the left four bits, ANDing the data byte with 11110000 will do the job. Notice that any bit ANDed with zero will be 0 and any bit ANDed with 1 will remain the same. If 10101010 is ANDed with the mask 11110000, the result is 10100000.

Boolean multiplication follows the same basic rules governing binary multiplication, which were discussed in Chapter 2 and are as follows:

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

Boolean multiplication is the same as the AND function.

The operation of a 2-input AND gate can be expressed in equation form as follows: If one input variable is A, if the other input variable is B, and if the output variable is X, then the Boolean expression is

$$X = AB$$

Figure 3–15(a) shows the AND gate logic symbol with two input variables and the output variable indicated.

When variables are shown together like *ABC*, they are ANDed.

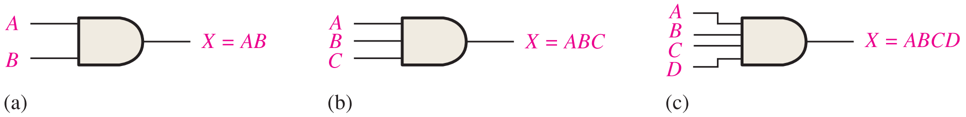


FIGURE 3–15 Boolean expressions for AND gates with two, three, and four inputs.

To extend the AND expression to more than two input variables, simply use a new letter for each input variable. The function of a 3-input AND gate, for example, can be expressed as $X = ABC$, where A, B, and C are the input variables. The expression for a 4-input AND gate can be $X = ABCD$, and so on. Parts (b) and (c) of Figure 3–15 show AND gates with three and four input variables, respectively.

You can evaluate an AND gate operation by using the Boolean expressions for the output. For example, each variable on the inputs can be either a 1 or a 0; so for the 2-input AND gate, make substitutions in the equation for the output, $X = AB$, as shown in Table 3–4. This evaluation shows that the output X of an AND gate is a 1 (HIGH) only when both inputs are 1s (HIGHs). A similar analysis can be made for any number of input variables.

TABLE 3–4

A	B	$AB = X$
0	0	$0 \cdot 0 = 0$
0	1	$0 \cdot 1 = 0$
1	0	$1 \cdot 0 = 0$
1	1	$1 \cdot 1 = 1$

Applications

The AND Gate as an Enable/Inhibit Device

A common application of the AND gate is to **enable** (that is, to allow) the passage of a signal (pulse waveform) from one point to another at certain times and to **inhibit** (prevent) the passage at other times.

A simple example of this particular use of an AND gate is shown in Figure 3–16, where the AND gate controls the passage of a signal (waveform A) to a digital counter. The purpose of this circuit is to measure the frequency of waveform A. The enable pulse has a width of precisely 1 ms. When the enable pulse is HIGH, waveform A passes through the gate to the counter; and when the enable pulse is LOW, the signal is prevented from passing through the gate (inhibited).

During the 1 millisecond (1 ms) interval of the enable pulse, pulses in waveform A pass through the AND gate to the counter. The number of pulses passing through during the 1 ms interval is equal to the frequency of waveform A. For example, Figure 3–16 shows six pulses in one millisecond, which is a frequency of 6 kHz. If 1000 pulses pass through the gate in the 1 ms interval of the enable pulse, there are 1000 pulses/ms, or a frequency of 1 MHz.

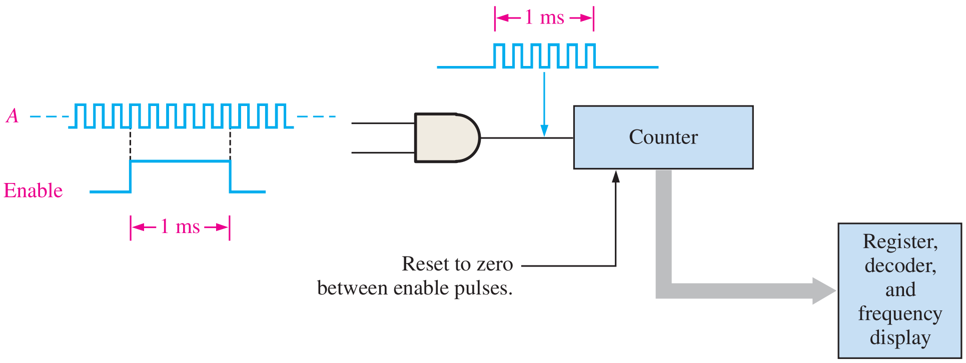

1 ms
A
Enable
1 ms
Counter
Reset to zero between enable pulses.
Register, decoder, and frequency display

FIGURE 3–16 An AND gate performing an enable/inhibit function for a frequency counter.

The counter counts the number of pulses per second and produces a binary output that goes to a decoding and display circuit to produce a readout of the frequency. The enable pulse repeats at certain intervals and a new updated count is made so that if the frequency changes, the new value will be displayed. Between enable pulses, the counter is reset so that it starts at zero each time an enable pulse occurs. The current frequency count is stored in a register so that the display is unaffected by the resetting of the counter.

A Seat Belt Alarm System

In Figure 3–17, an AND gate is used in a simple automobile seat belt alarm system to detect when the ignition switch is on *and* the seat belt is unbuckled. If the ignition switch is on, a HIGH is produced on input *A* of the AND gate. If the seat belt is not properly buckled, a HIGH is produced on input *B* of the AND gate. Also, when the ignition switch is turned on, a timer is started that produces a HIGH on input *C* for 30 s. If all three conditions exist—that is, if the ignition is on *and* the seat belt is unbuckled *and* the timer is running—the output of the AND gate is HIGH, and an audible alarm is energized to remind the driver.

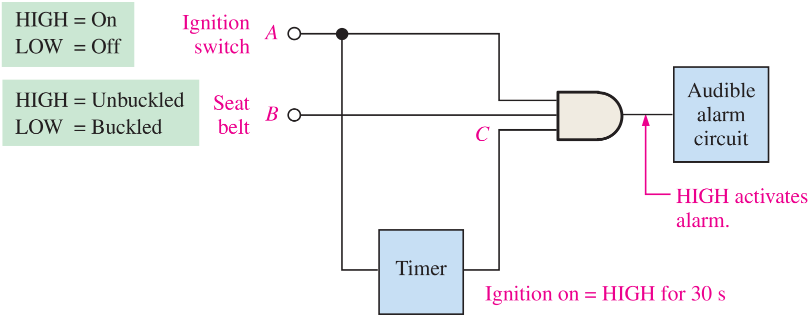


FIGURE 3–17 A simple seat belt alarm circuit using an AND gate.

SECTION 3–2 CHECKUP

1. When is the output of an AND gate HIGH?
2. When is the output of an AND gate LOW?
3. Describe the truth table for a 5-input AND gate.

3–3 The OR Gate

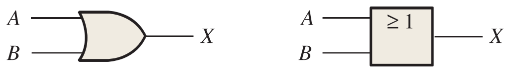
The OR gate is another of the basic gates from which all logic functions are constructed. An OR gate can have two or more inputs and performs what is known as logical addition.

After completing this section, you should be able to

- Identify an OR gate by its distinctive shape symbol or by its rectangular outline symbol
- Describe the operation of an OR gate
- Generate the truth table for an OR gate with any number of inputs
- Produce a timing diagram for an OR gate with any specified input waveforms
- Write the logic expression for an OR gate with any number of inputs
- Discuss an OR gate application

An OR gate can have more than two inputs.

An **OR gate** has two or more inputs and one output, as indicated by the standard logic symbols in Figure 3–18, where OR gates with two inputs are illustrated. An OR gate can have any number of inputs greater than one. Although both distinctive shape and rectangular outline symbols are shown, the distinctive shape OR gate symbol is used in this textbook.



(a) Distinctive shape

(b) Rectangular outline with the OR (≥ 1) qualifying symbol

FIGURE 3–18 Standard logic symbols for the OR gate showing two inputs (ANSI/IEEE Std. 91-1984/Std. 91a-1991).

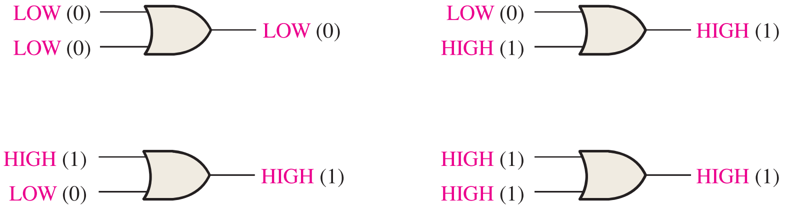
Operation of an OR Gate

For an OR gate, at least one HIGH input produces a HIGH output.

An OR gate produces a HIGH on the output when *any* of the inputs is HIGH. The output is LOW only when all of the inputs are LOW. Therefore, an OR gate determines when one or more of its inputs are HIGH and produces a HIGH on its output to indicate this condition. The inputs of the 2-input OR gate in Figure 3–18 are labeled *A* and *B*, and the output is labeled *X*. The operation of the gate can be stated as follows:

For a 2-input OR gate, output *X* is HIGH when either input *A* or input *B* is HIGH, or when both *A* and *B* are HIGH; *X* is LOW only when both *A* and *B* are LOW.

The HIGH level is the active or asserted output level for the OR gate. Figure 3–19 illustrates the operation for a 2-input OR gate for all four possible input combinations.



MultiSim

FIGURE 3–19 All possible logic levels for a 2-input OR gate. Open file F03-19 to verify OR gate operation.

OR Gate Truth Table

The operation of a 2-input OR gate is described in Table 3–5. This truth table can be expanded for any number of inputs; but regardless of the number of inputs, the output is HIGH when one or more of the inputs are HIGH.

TABLE 3–5

Truth table for a 2-input OR gate.

Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	1

1 = HIGH, 0 = LOW

OR Gate Operation with Waveform Inputs

Now let's look at the operation of an OR gate with pulse waveform inputs, keeping in mind its logical operation. Again, the important thing in the analysis of gate operation with pulse waveforms is the time relationship of all the waveforms involved. For example, in Figure 3–20, inputs *A* and *B* are both HIGH (1) during time interval t_1, making output *X* HIGH (1). During time interval t_2, input *A* is LOW (0), but because input *B* is HIGH (1), the output is HIGH (1). Both inputs are LOW (0) during time interval t_3, so there is a LOW (0) output during this time. During time interval t_4, the output is HIGH (1) because input *A* is HIGH (1).

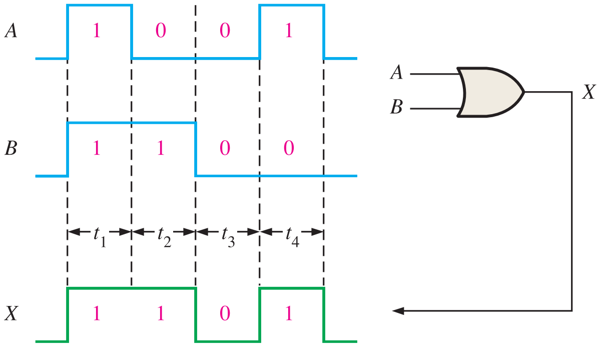
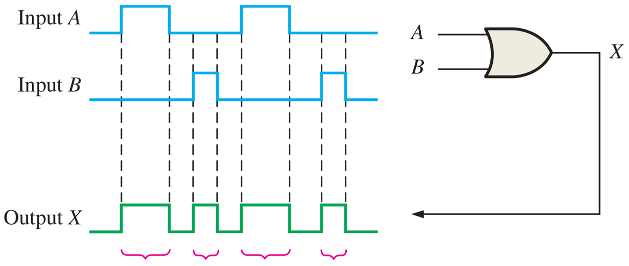


FIGURE 3–20 Example of OR gate operation with a timing diagram showing input and output time relationships.

In this illustration, we have applied the truth table operation of the OR gate to each of the time intervals during which the levels are nonchanging. Examples 3–7 through 3–9 further illustrate OR gate operation with waveforms on the inputs.

EXAMPLE 3–7

If the two input waveforms, *A* and *B*, in Figure 3–21 are applied to the OR gate, what is the resulting output waveform?



When either input or both inputs are HIGH, the output is HIGH.

FIGURE 3–21

Solution

The output waveform *X* of a 2-input OR gate is HIGH when either or both input waveforms are HIGH as shown in the timing diagram. In this case, both input waveforms are never HIGH at the same time.

Related Problem

Determine the output waveform and show the timing diagram if input *A* is changed such that it is HIGH from the beginning of the existing first pulse to the end of the existing second pulse.

EXAMPLE 3–8

For the two input waveforms, *A* and *B*, in Figure 3–22, show the output waveform with its proper relation to the inputs.

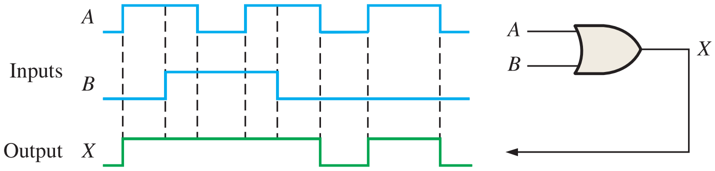


FIGURE 3–22

Solution

When either or both input waveforms are HIGH, the output is HIGH as shown by the output waveform *X* in the timing diagram.

Related Problem

Determine the output waveform and show the timing diagram if the middle pulse of input *A* is replaced by a LOW level.

EXAMPLE 3–9

For the 3-input OR gate in Figure 3–23, determine the output waveform in proper time relation to the inputs.

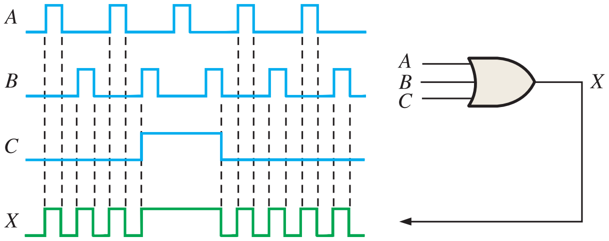


FIGURE 3–23

Solution

The output is HIGH when one or more of the input waveforms are HIGH as indicated by the output waveform *X* in the timing diagram.

Related Problem

Determine the output waveform and show the timing diagram if input *C* is always LOW.

Logic Expressions for an OR Gate

The logical OR function of two variables is represented mathematically by a $+$ between the two variables, for example, $A + B$. The plus sign is read as “OR.”

When variables are separated by $+$, they are ORed.

Addition in Boolean algebra involves variables whose values are either binary 1 or binary 0. The basic rules for **Boolean addition** are as follows:

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

Boolean addition is the same as the OR function.

Notice that Boolean addition differs from binary addition in the case where two 1s are added. There is no carry in Boolean addition.

The operation of a 2-input OR gate can be expressed as follows: If one input variable is A, if the other input variable is B, and if the output variable is X, then the Boolean expression is

$$X = A + B$$

Figure 3–24(a) shows the OR gate logic symbol with two input variables and the output variable labeled.

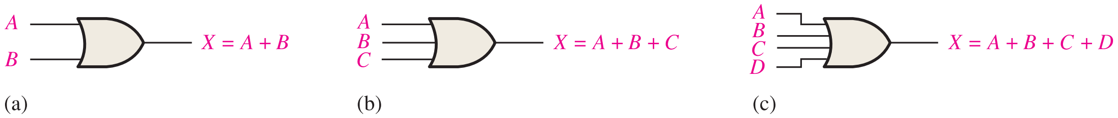


FIGURE 3–24 Boolean expressions for OR gates with two, three, and four inputs.

To extend the OR expression to more than two input variables, a new letter is used for each additional variable. For instance, the function of a 3-input OR gate can be expressed as $X = A + B + C$. The expression for a 4-input OR gate can be written as $X = A + B + C + D$, and so on. Parts (b) and (c) of Figure 3–24 show OR gates with three and four input variables, respectively.

OR gate operation can be evaluated by using the Boolean expressions for the output X by substituting all possible combinations of 1 and 0 values for the input variables, as shown in Table 3–6 for a 2-input OR gate. This evaluation shows that the output X of an OR gate is a 1 (HIGH) when any one or more of the inputs are 1 (HIGH). A similar analysis can be extended to OR gates with any number of input variables.

InfoNote

A mask operation that is used in computer programming to selectively make certain bits in a data byte equal to 1 (called setting) while not affecting any other bit is done with the OR operation. A mask is used that contains a 1 in any position where a data bit is to be set. For example, if you want to force the sign bit in an 8-bit signed number to equal 1, but leave all other bits unchanged, you can OR the data byte with the mask 10000000.

TABLE 3–6

A	B	$A + B = X$
0	0	$0 + 0 = 0$
0	1	$0 + 1 = 1$
1	0	$1 + 0 = 1$
1	1	$1 + 1 = 1$

An Application

A simplified portion of an intrusion detection and alarm system is shown in Figure 3–25. This system could be used for one room in a home—a room with two windows and a door. The sensors are magnetic switches that produce a HIGH output when open and a LOW output when closed. As long as the windows and the door are secured, the switches are closed and all three of the OR gate inputs are LOW. When one of the windows or the door is opened, a HIGH is produced on that input to the OR gate and the gate output goes HIGH. It then activates and latches an alarm circuit to warn of the intrusion.

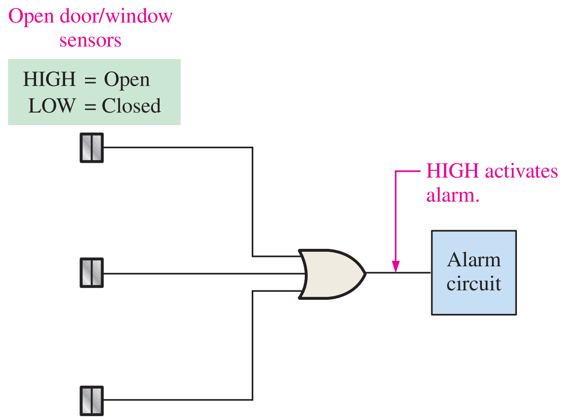


FIGURE 3–25 A simplified intrusion detection system using an OR gate.

SECTION 3–3 CHECKUP

1. When is the output of an OR gate HIGH?
2. When is the output of an OR gate LOW?
3. Describe the truth table for a 3-input OR gate.

3–4 The NAND Gate

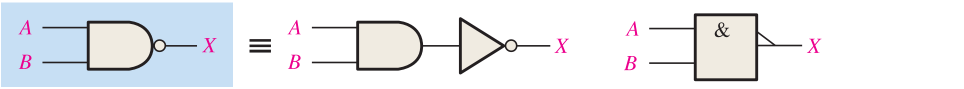
The NAND gate is a popular logic element because it can be used as a universal gate; that is, NAND gates can be used in combination to perform the AND, OR, and inverter operations. The universal property of the NAND gate will be examined thoroughly in Chapter 5.

After completing this section, you should be able to

- Identify a NAND gate by its distinctive shape symbol or by its rectangular outline symbol
- Describe the operation of a NAND gate
- Develop the truth table for a NAND gate with any number of inputs
- Produce a timing diagram for a NAND gate with any specified input waveforms
- Write the logic expression for a NAND gate with any number of inputs
- Describe NAND gate operation in terms of its negative-OR equivalent
- Discuss examples of NAND gate applications

The NAND gate is the same as the AND gate except the output is inverted.

The term *NAND* is a contraction of NOT-AND and implies an AND function with a complemented (inverted) output. The standard logic symbol for a 2-input NAND gate and its equivalency to an AND gate followed by an inverter are shown in Figure 3–26(a), where the symbol ≡ means equivalent to. A rectangular outline symbol is shown in part (b).



(a) Distinctive shape, 2-input NAND gate and its NOT/AND equivalent

(b) Rectangular outline, 2-input NAND gate with polarity indicator

FIGURE 3–26 Standard NAND gate logic symbols (ANSI/IEEE Std. 91-1984/Std. 91a-1991).

Operation of a NAND Gate

A **NAND gate** produces a LOW output only when all the inputs are HIGH. When any of the inputs is LOW, the output will be HIGH. For the specific case of a 2-input NAND gate, as shown in Figure 3–26 with the inputs labeled *A* and *B* and the output labeled *X*, the operation can be stated as follows:

> **For a 2-input NAND gate, output *X* is LOW only when inputs *A* and *B* are HIGH; *X* is HIGH when either *A* or *B* is LOW, or when both *A* and *B* are LOW.**

This operation is opposite that of the AND in terms of the output level. In a NAND gate, the LOW level (0) is the active or asserted output level, as indicated by the bubble on the output. Figure 3–27 illustrates the operation of a 2-input NAND gate for all four input combinations, and Table 3–7 is the truth table summarizing the logical operation of the 2-input NAND gate.

TABLE 3–7

Truth table for a 2-input NAND gate.

Inputs		Output
A	*B*	*X*
0	0	1
0	1	1
1	0	1
1	1	0

1 = HIGH, 0 = LOW.

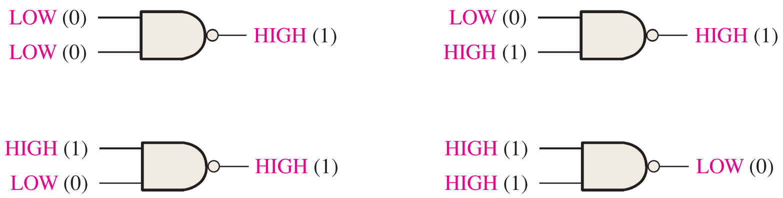


FIGURE 3–27 Operation of a 2-input NAND gate. Open file F03-27 to verify NAND gate operation.

MultiSim

NAND Gate Operation with Waveform Inputs

Now let's look at the pulse waveform operation of a NAND gate. Remember from the truth table that the only time a LOW output occurs is when all of the inputs are HIGH.

EXAMPLE 3–10

If the two waveforms *A* and *B* shown in Figure 3–28 are applied to the NAND gate inputs, determine the resulting output waveform.

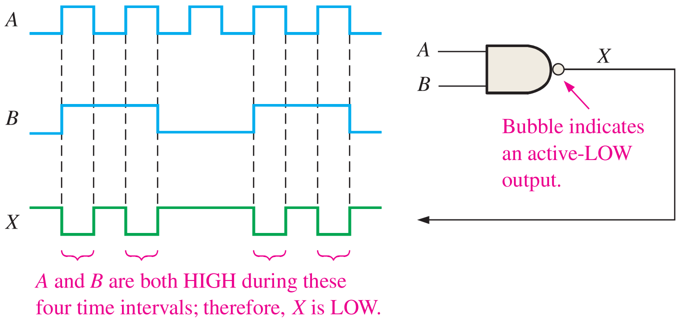

A
B
X
A
B
X
Bubble indicates an active-LOW output.
A and B are both HIGH during these four time intervals; therefore, X is LOW.

FIGURE 3–28

Solution

Output waveform *X* is LOW only during the four time intervals when both input waveforms *A* and *B* are HIGH as shown in the timing diagram.

Related Problem

Determine the output waveform and show the timing diagram if input waveform *B* is inverted.

EXAMPLE 3–11

Show the output waveform for the 3-input NAND gate in Figure 3–29 with its proper time relationship to the inputs.

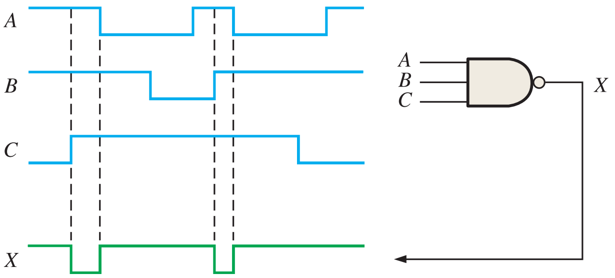


FIGURE 3–29

Solution

The output waveform X is LOW only when all three input waveforms are HIGH as shown in the timing diagram.

Related Problem

Determine the output waveform and show the timing diagram if input waveform A is inverted.

Negative-OR Equivalent Operation of a NAND Gate

Inherent in a NAND gate's operation is the fact that one or more LOW inputs produce a HIGH output. Table 3–7 shows that output X is HIGH (1) when any of the inputs, A and B, is LOW (0). From this viewpoint, a NAND gate can be used for an OR operation that requires one or more LOW inputs to produce a HIGH output. This aspect of NAND operation is referred to as **negative-OR**. The term *negative* in this context means that the inputs are defined to be in the active or asserted state when LOW.

For a 2-input NAND gate performing a negative-OR operation, output X is HIGH when either input A or input B is LOW, or when both A and B are LOW.

When a NAND gate is used to detect one or more LOWs on its inputs rather than all HIGHs, it is performing the negative-OR operation and is represented by the standard logic symbol shown in Figure 3–30. Although the two symbols in Figure 3–30 represent the same physical gate, they serve to define its role or mode of operation in a particular application, as illustrated by Examples 3–12 and 3–13.

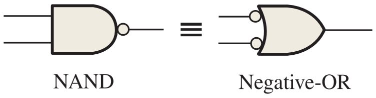


FIGURE 3–30 ANSI/IEEE standard symbols representing the two equivalent operations of a NAND gate.

EXAMPLE 3–12

Two tanks store certain liquid chemicals that are required in a manufacturing process. Each tank has a sensor that detects when the chemical level drops to 25% of full. The sensors produce a HIGH level of 5 V when the tanks are more than one-quarter full. When the volume of chemical in a tank drops to one-quarter full, the sensor puts out a LOW level of 0 V.

It is required that a single green light-emitting diode (LED) on an indicator panel show when both tanks are more than one-quarter full. Show how a NAND gate can be used to implement this function.

Solution

Figure 3–31 shows a NAND gate with its two inputs connected to the tank level sensors and its output connected to the indicator panel. The operation can be stated as follows: If tank A *and* tank B are above one-quarter full, the LED is on.

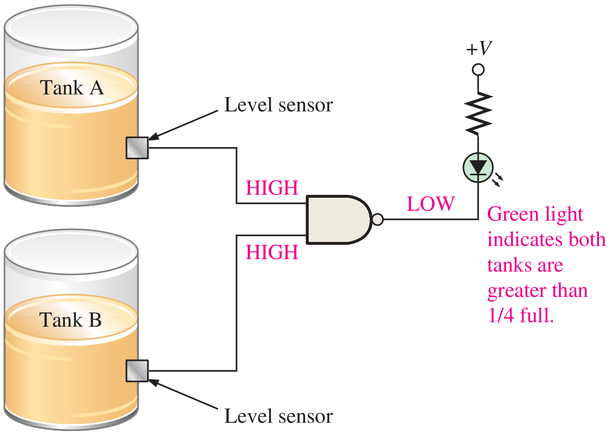


FIGURE 3–31

As long as both sensor outputs are HIGH (5 V), indicating that both tanks are more than one-quarter full, the NAND gate output is LOW (0 V). The green LED circuit is connected so that a LOW voltage turns it on. The resistor limits the LED current.

Related Problem

How can the circuit of Figure 3–31 be modified to monitor the levels in three tanks rather than two?

EXAMPLE 3–13

For the process described in Example 3–12 it has been decided to have a red LED display come on when at least one of the tanks falls to the quarter-full level rather than have the green LED display indicate when both are above one quarter. Show how this requirement can be implemented.

Solution

Figure 3–32 shows a NAND gate operating as a negative-OR gate to detect the occurrence of at least one LOW on its inputs. A sensor puts out a LOW voltage if the volume in its tank goes to one-quarter full or less. When this happens, the gate output goes HIGH. The red LED circuit in the panel is connected so that a HIGH voltage turns it on. The operation can be stated as follows: If tank *A or* tank *B or* both are below one-quarter full, the LED is on.

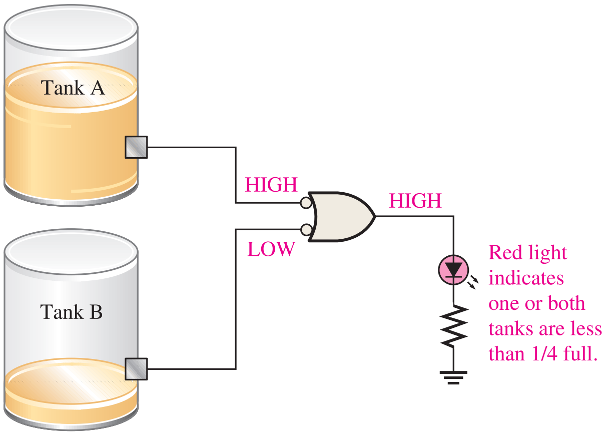

Tank A
HIGH
HIGH
LOW
Red light indicates one or both tanks are less than 1/4 full.
Tank B

FIGURE 3–32

Notice that, in this example and in Example 3–12, the same 2-input NAND gate is used, but in this example it is operating as a negative-OR gate and a different gate symbol is used in the schematic. This illustrates the different way in which the NAND and equivalent negative-OR operations are used.

Related Problem

How can the circuit in Figure 3–32 be modified to monitor four tanks rather than two?

EXAMPLE 3–14

For the 4-input NAND gate in Figure 3–33, operating as a negative-OR gate, determine the output with respect to the inputs.

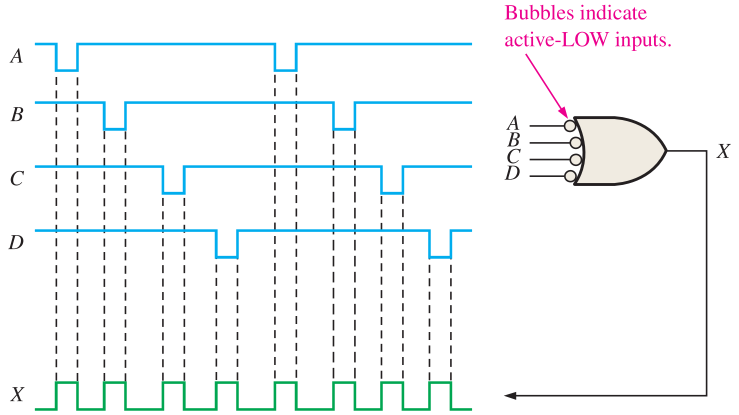


FIGURE 3–33

Solution

The output waveform X is HIGH any time an input waveform is LOW as shown in the timing diagram.

Related Problem

Determine the output waveform if input waveform A is inverted before it is applied to the gate.

Logic Expressions for a NAND Gate

A bar over a variable or variables indicates an inversion.

The Boolean expression for the output of a 2-input NAND gate is

$$X = \overline{AB}$$

This expression says that the two input variables, A and B, are first ANDed and then complemented, as indicated by the bar over the AND expression. This is a description in equation form of the operation of a NAND gate with two inputs. Evaluating this expression for all possible values of the two input variables, you get the results shown in Table 3–8.

TABLE 3–8

A	B	$\overline{AB} = X$
0	0	$\overline{0 \cdot 0} = \overline{0} = 1$
0	1	$\overline{0 \cdot 1} = \overline{0} = 1$
1	0	$\overline{1 \cdot 0} = \overline{0} = 1$
1	1	$\overline{1 \cdot 1} = \overline{1} = 0$

Once an expression is determined for a given logic function, that function can be evaluated for all possible values of the variables. The evaluation tells you exactly what the output of the logic circuit is for each of the input conditions, and it therefore gives you a complete description of the circuit's logic operation. The NAND expression can be extended to more than two input variables by including additional letters to represent the other variables.

SECTION 3–4 CHECKUP

1. When is the output of a NAND gate LOW?
2. When is the output of a NAND gate HIGH?
3. Describe the functional differences between a NAND gate and a negative-OR gate. Do they both have the same truth table?
4. Write the output expression for a NAND gate with inputs *A*, *B*, and *C*.

3–5 The NOR Gate

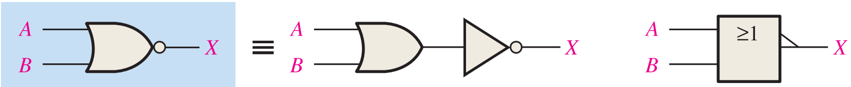
The NOR gate, like the NAND gate, is a useful logic element because it can also be used as a universal gate; that is, NOR gates can be used in combination to perform the AND, OR, and inverter operations. The universal property of the NOR gate will be examined thoroughly in Chapter 5.

After completing this section, you should be able to

- Identify a NOR gate by its distinctive shape symbol or by its rectangular outline symbol
- Describe the operation of a NOR gate
- Develop the truth table for a NOR gate with any number of inputs
- Produce a timing diagram for a NOR gate with any specified input waveforms
- Write the logic expression for a NOR gate with any number of inputs
- Describe NOR gate operation in terms of its negative-AND equivalent
- Discuss examples of NOR gate applications

The term *NOR* is a contraction of NOT-OR and implies an OR function with an inverted (complemented) output. The standard logic symbol for a 2-input NOR gate and its equivalent OR gate followed by an inverter are shown in Figure 3–34(a). A rectangular outline symbol is shown in part (b).

The NOR is the same as the OR except the output is inverted.



(a) Distinctive shape, 2-input NOR gate and its NOT/OR equivalent

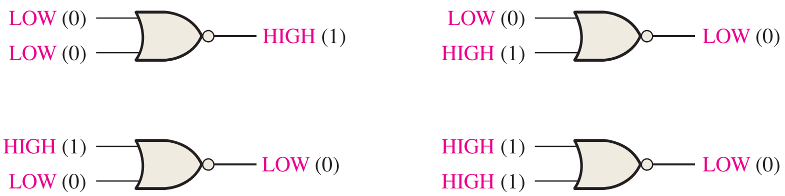
(b) Rectangular outline, 2-input NOR gate with polarity indicator

FIGURE 3–34 Standard NOR gate logic symbols (ANSI/IEEE Std. 91-1984/Std. 91a-1991).

Operation of a NOR Gate

A **NOR gate** produces a LOW output when *any* of its inputs is HIGH. Only when all of its inputs are LOW is the output HIGH. For the specific case of a 2-input NOR gate, as shown in Figure 3–34 with the inputs labeled *A* and *B* and the output labeled *X*, the operation can be stated as follows:

> **For a 2-input NOR gate, output *X* is LOW when either input *A* or input *B* is HIGH, or when both *A* and *B* are HIGH; *X* is HIGH only when both *A* and *B* are LOW.**



MultiSim **FIGURE 3–35** Operation of a 2-input NOR gate. Open file F03-35 to verify NOR gate operation.

TABLE 3–9

Truth table for a 2-input NOR gate.

Inputs		Output
A	*B*	*X*
0	0	1
0	1	0
1	0	0
1	1	0

1 = HIGH, 0 = LOW.

This operation results in an output level opposite that of the OR gate. In a NOR gate, the LOW output is the active or asserted output level as indicated by the bubble on the output. Figure 3–35 illustrates the operation of a 2-input NOR gate for all four possible input combinations, and Table 3–9 is the truth table for a 2-input NOR gate.

NOR Gate Operation with Waveform Inputs

The next two examples illustrate the operation of a NOR gate with pulse waveform inputs. Again, as with the other types of gates, we will simply follow the truth table operation to determine the output waveforms in the proper time relationship to the inputs.

EXAMPLE 3–15

If the two waveforms shown in Figure 3–36 are applied to a NOR gate, what is the resulting output waveform?

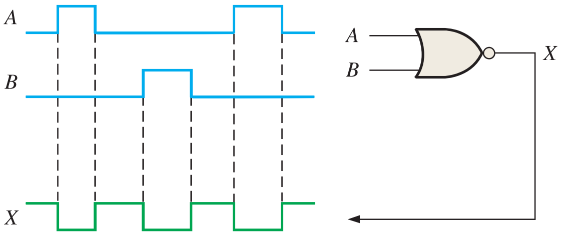


FIGURE 3–36

Solution

Whenever any input of the NOR gate is HIGH, the output is LOW as shown by the output waveform *X* in the timing diagram.

Related Problem

Invert input *B* and determine the output waveform in relation to the inputs.

EXAMPLE 3–16

Show the output waveform for the 3-input NOR gate in Figure 3–37 with the proper time relation to the inputs.

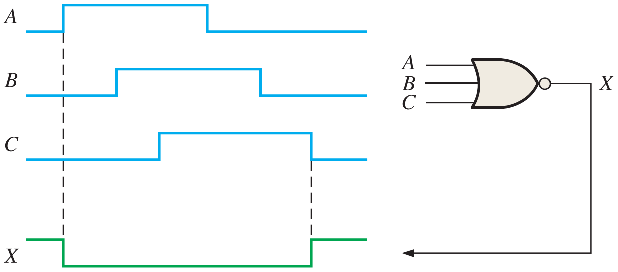


FIGURE 3–37

Solution

The output *X* is LOW when any input is HIGH as shown by the output waveform *X* in the timing diagram.

Related Problem

With the *B* and *C* inputs inverted, determine the output and show the timing diagram.

Negative-AND Equivalent Operation of the NOR Gate

A NOR gate, like the NAND, has another aspect of its operation that is inherent in the way it logically functions. Table 3–9 shows that a HIGH is produced on the gate output only when all of the inputs are LOW. From this viewpoint, a NOR gate can be used for an AND operation that requires all LOW inputs to produce a HIGH output. This aspect of NOR operation is called **negative-AND**. The term *negative* in this context means that the inputs are defined to be in the active or asserted state when LOW.

For a 2-input NOR gate performing a negative-AND operation, output *X* is HIGH only when both inputs *A* and *B* are LOW.

When a NOR gate is used to detect all LOWs on its inputs rather than one or more HIGHs, it is performing the negative-AND operation and is represented by the standard symbol in Figure 3–38. Remember that the two symbols in Figure 3–38 represent the same physical gate and serve only to distinguish between the two modes of its operation. The following three examples illustrate this.

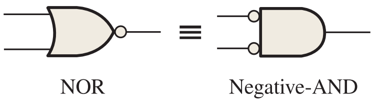


FIGURE 3–38 Standard symbols representing the two equivalent operations of a NOR gate.

EXAMPLE 3–17

A device is needed to indicate when two LOW levels occur simultaneously on its inputs and to produce a HIGH output as an indication. Specify the device.

Solution

A 2-input NOR gate operating as a negative-AND gate is required to produce a HIGH output when both inputs are LOW, as shown in Figure 3–39.

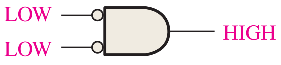


FIGURE 3–39

Related Problem

A device is needed to indicate when one or two HIGH levels occur on its inputs and to produce a LOW output as an indication. Specify the device.

EXAMPLE 3–18

As part of an aircraft's functional monitoring system, a circuit is required to indicate the status of the landing gears prior to landing. A green LED display turns on if all three gears are properly extended when the "gear down" switch has been activated in preparation for landing. A red LED display turns on if any of the gears fail to extend properly prior to landing. When a landing gear is extended, its sensor produces a LOW voltage. When a landing gear is retracted, its sensor produces a HIGH voltage. Implement a circuit to meet this requirement.

Solution

Power is applied to the circuit only when the "gear down" switch is activated. Use a NOR gate for each of the two requirements as shown in Figure 3–40. One NOR gate operates as a negative-AND to detect a LOW from each of the three landing gear sensors. When all three of the gate inputs are LOW, the three landing gears are properly extended and the resulting HIGH output from the negative-AND gate turns on the green LED display. The other NOR gate operates as a NOR to detect if one or more of the landing gears remain retracted when the "gear down" switch is activated. When one or more of the landing gears remain retracted, the resulting HIGH from the sensor is detected by the NOR gate, which produces a LOW output to turn on the red LED warning display.

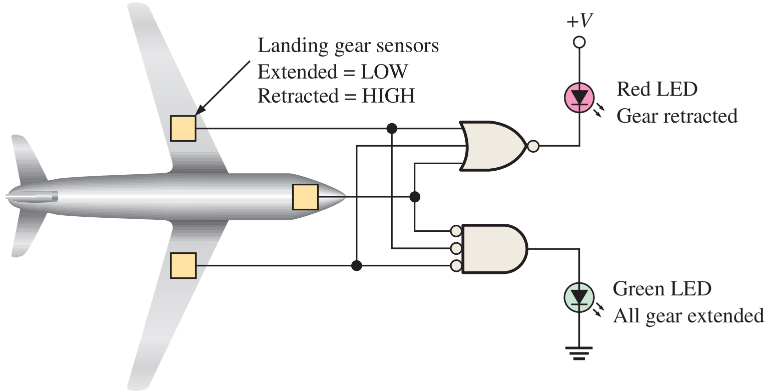


FIGURE 3–40

Related Problem

What type of gate should be used to detect if all three landing gears are retracted after takeoff, assuming a LOW output is required to activate an LED display?

When driving a load such as an LED with a logic gate, consult the manufacturer's data sheet for maximum drive capabilities (output current). A regular IC logic gate may not be capable of handling the current required by certain loads such as some LEDs. Logic gates with a buffered output, such as an open-collector (OC) or open-drain (OD) output, are available in many types of IC logic gate configurations. The output current capability of typical IC logic gates is limited to the μA or relatively low mA range. For example, standard TTL can handle output currents up to 16 mA but only when the output is LOW. Most LEDs require currents in the range of about 10 mA to 50 mA.

EXAMPLE 3–19

For the 4-input NOR gate operating as a negative-AND in Figure 3–41, determine the output relative to the inputs.

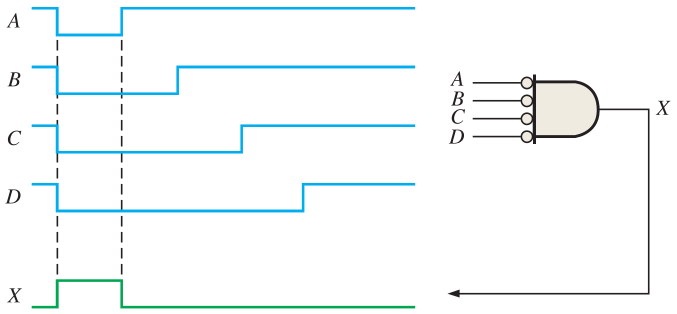


FIGURE 3–41

Solution

Any time all of the input waveforms are LOW, the output is HIGH as shown by output waveform X in the timing diagram.

Related Problem

Determine the output with input D inverted and show the timing diagram.

Logic Expressions for a NOR Gate

The Boolean expression for the output of a 2-input NOR gate can be written as

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

This equation says that the two input variables are first ORed and then complemented, as indicated by the bar over the OR expression. Evaluating this expression, you get the results shown in Table 3–10. The NOR expression can be extended to more than two input variables by including additional letters to represent the other variables.

TABLE 3–10

A	B	$\overline{A + B} = X$
0	0	$\overline{0 + 0} = \overline{0} = 1$
0	1	$\overline{0 + 1} = \overline{1} = 0$
1	0	$\overline{1 + 0} = \overline{1} = 0$
1	1	$\overline{1 + 1} = \overline{1} = 0$

SECTION 3–5 CHECKUP

1. When is the output of a NOR gate HIGH?
2. When is the output of a NOR gate LOW?
3. Describe the functional difference between a NOR gate and a negative-AND gate. Do they both have the same truth table?
4. Write the output expression for a 3-input NOR with input variables A, B, and C.

3–6 The Exclusive-OR and Exclusive-NOR Gates

Exclusive-OR and exclusive-NOR gates are formed by a combination of other gates already discussed, as you will see in Chapter 5. However, because of their fundamental importance in many applications, these gates are often treated as basic logic elements with their own unique symbols.

After completing this section, you should be able to

- Identify the exclusive-OR and exclusive-NOR gates by their distinctive shape symbols or by their rectangular outline symbols
- Describe the operations of exclusive-OR and exclusive-NOR gates
- Show the truth tables for exclusive-OR and exclusive-NOR gates
- Produce a timing diagram for an exclusive-OR or exclusive-NOR gate with any specified input waveforms
- Discuss examples of exclusive-OR and exclusive-NOR gate applications

The Exclusive-OR Gate

Standard symbols for an exclusive-OR (XOR for short) gate are shown in Figure 3–42. The XOR gate has only two inputs. The **exclusive-OR gate** performs modulo-2 addition (introduced in Chapter 2). The output of an exclusive-OR gate is HIGH *only* when the two

InfoNote

Exclusive-OR gates connected to form an adder circuit allow a processor to perform addition, subtraction, multiplication, and division in its Arithmetic Logic Unit (ALU). An exclusive-OR gate combines basic AND, OR, and NOT logic.

For an exclusive-OR gate, opposite inputs make the output HIGH.

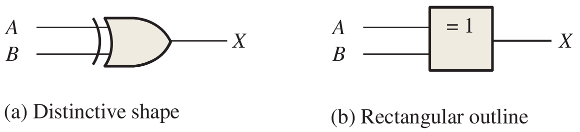


FIGURE 3–42 Standard logic symbols for the exclusive-OR gate.

TABLE 3–11

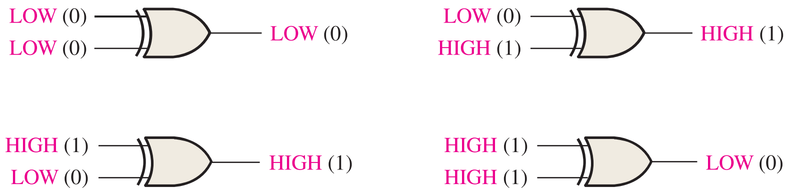
Truth table for an exclusive-OR gate.

Inputs		Output
A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

inputs are at opposite logic levels. This operation can be stated as follows with reference to inputs *A* and *B* and output *X*:

For an exclusive-OR gate, output *X* is HIGH when input *A* is LOW and input *B* is HIGH, or when input *A* is HIGH and input *B* is LOW; *X* is LOW when *A* and *B* are both HIGH or both LOW.

The four possible input combinations and the resulting outputs for an XOR gate are illustrated in Figure 3–43. The HIGH level is the active or asserted output level and occurs only when the inputs are at opposite levels. The operation of an XOR gate is summarized in the truth table shown in Table 3–11.



MultiSim

FIGURE 3–43 All possible logic levels for an exclusive-OR gate. Open file F03-43 to verify XOR gate operation.

EXAMPLE 3–20

A certain system contains two identical circuits operating in parallel. As long as both are operating properly, the outputs of both circuits are always the same. If one of the circuits fails, the outputs will be at opposite levels at some time. Devise a way to monitor and detect that a failure has occurred in one of the circuits.

Solution

The outputs of the circuits are connected to the inputs of an XOR gate as shown in Figure 3–44. A failure in either one of the circuits produces differing outputs, which cause the XOR inputs to be at opposite levels. This condition produces a HIGH on the output of the XOR gate, indicating a failure in one of the circuits.

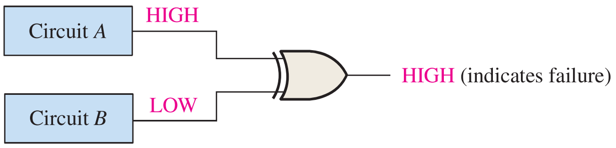


FIGURE 3–44

Related Problem

Will the exclusive-OR gate always detect simultaneous failures in both circuits of Figure 3–44? If not, under what condition?

The Exclusive-NOR Gate

Standard symbols for an **exclusive-NOR** (XNOR) **gate** are shown in Figure 3–45. Like the XOR gate, an XNOR has only two inputs. The bubble on the output of the XNOR symbol indicates that its output is opposite that of the XOR gate. When the two input logic levels are opposite, the output of the exclusive-NOR gate is LOW. The operation can be stated as follows (*A* and *B* are inputs, *X* is the output):

> **For an exclusive-NOR gate, output *X* is LOW when input *A* is LOW and input *B* is HIGH, or when *A* is HIGH and *B* is LOW; *X* is HIGH when *A* and *B* are both HIGH or both LOW.**

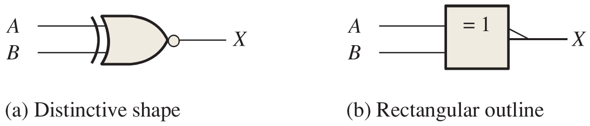


FIGURE 3–45 Standard logic symbols for the exclusive-NOR gate.

TABLE 3–12

Truth table for an exclusive-NOR gate.

Inputs		Output
A	*B*	*X*
0	0	1
0	1	0
1	0	0
1	1	1

The four possible input combinations and the resulting outputs for an XNOR gate are shown in Figure 3–46. The operation of an XNOR gate is summarized in Table 3–12. Notice that the output is HIGH when the same level is on both inputs.

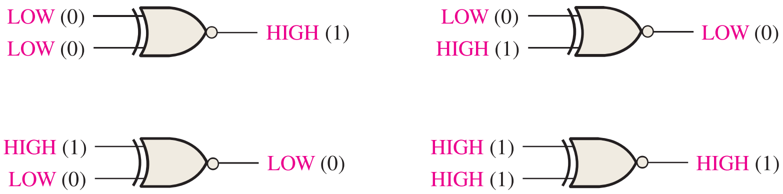


FIGURE 3–46 All possible logic levels for an exclusive-NOR gate. Open file F03-46 to verify XNOR gate operation.

MultiSim

Operation with Waveform Inputs

As we have done with the other gates, let's examine the operation of XOR and XNOR gates with pulse waveform inputs. As before, we apply the truth table operation during each distinct time interval of the pulse waveform inputs, as illustrated in Figure 3–47 for an XOR gate. You can see that the input waveforms *A* and *B* are at opposite levels during time intervals t_2 and t_4. Therefore, the output *X* is HIGH during these two times. Since both inputs are at the same level, either both HIGH or both LOW, during time intervals t_1 and t_3, the output is LOW during those times as shown in the timing diagram.

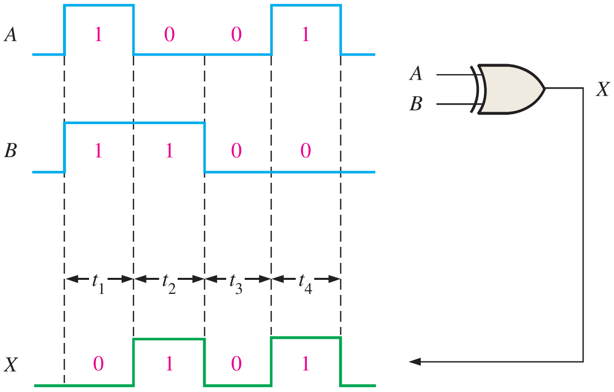


FIGURE 3–47 Example of exclusive-OR gate operation with pulse waveform inputs.

EXAMPLE 3–21

Determine the output waveforms for the XOR gate and for the XNOR gate, given the input waveforms, *A* and *B*, in Figure 3–48.

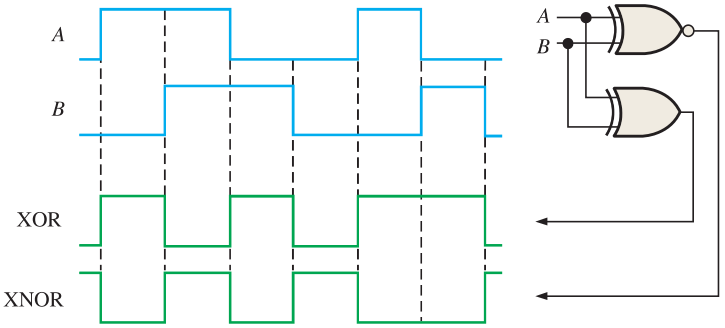


FIGURE 3–48

Solution

The output waveforms are shown in Figure 3–48. Notice that the XOR output is HIGH only when both inputs are at opposite levels. Notice that the XNOR output is HIGH only when both inputs are the same.

Related Problem

Determine the output waveforms if the two input waveforms, *A* and *B*, are inverted.

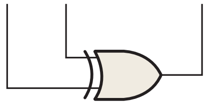
An Application

An exclusive-OR gate can be used as a two-bit modulo-2 adder. Recall from Chapter 2 that the basic rules for binary addition are as follows: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 10$. An examination of the truth table for an XOR gate shows that its output is the binary sum of the two input bits. In the case where the inputs are both 1s, the output is the sum 0, but you lose the carry of 1. In Chapter 6 you will see how XOR gates are combined to make complete adding circuits. Table 3–13 illustrates an XOR gate used as a modulo-2 adder. It is used in CRC systems to implement the division process that was described in Chapter 2.

TABLE 3–13

An XOR gate used to add two bits.

Input Bits		Output (Sum)	
A	*B*	Σ	
0	0	0	
0	1	1	
1	0	1	
1	1	0	(without the 1 carry bit)



SECTION 3–6 CHECKUP

1. When is the output of an XOR gate HIGH?
2. When is the output of an XNOR gate HIGH?
3. How can you use an XOR gate to detect when two bits are different?

3–7 Programmable Logic

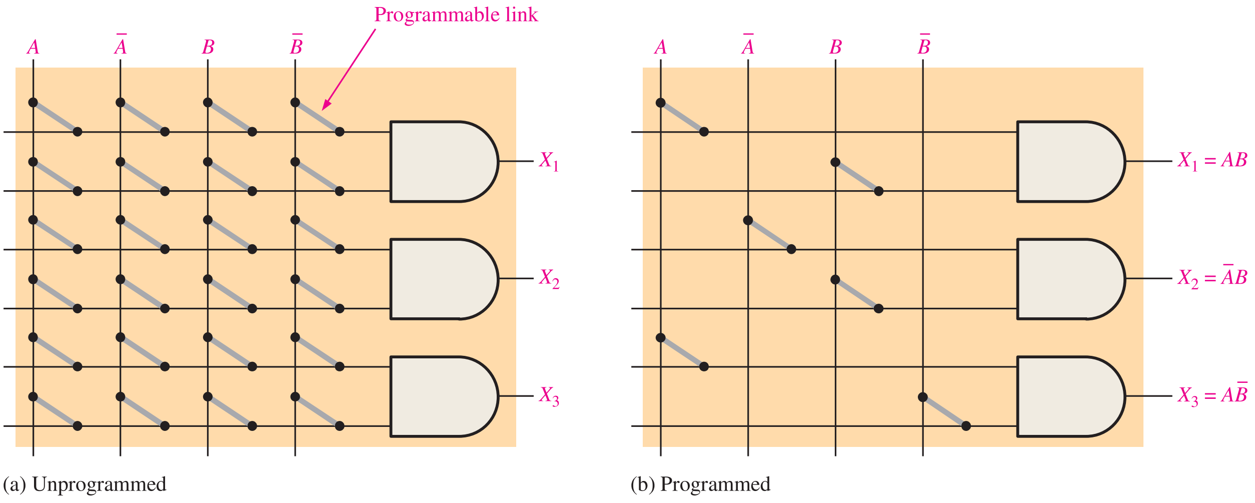
Programmable logic was introduced in Chapter 1. In this section, the basic concept of the programmable AND array, which forms the basis for most programmable logic, is discussed, and the major process technologies are covered. A programmable logic device (PLD) is one that does not initially have a fixed-logic function but that can be programmed to implement just about any logic design. As you have learned, two types of PLD are the SPLD and CPLD. In addition to the PLD, the other major category of programmable logic is the FPGA. Also, basic VHDL programming is introduced.

After completing this section, you should be able to

- Describe the concept of a programmable AND array
- Discuss various process technologies for programming a PLD
- Discuss downloading a design to a programmable logic device
- Discuss text entry and graphic entry as two methods for programmable logic design
- Explain in-system programming
- Write VHDL descriptions of logic gates

The AND Array

Most types of PLDs use some form of **AND array**. Basically, this array consists of AND gates and a matrix of interconnections with a programmable link at each cross point, as shown in Figure 3–49(a). Programmable links allow a connection between a row line and a column line in the interconnection matrix to be opened or left intact. For each input to an AND gate, only one programmable link is left intact in order to connect the desired variable to the gate input. Figure 3–49(b) illustrates an array after it has been programmed.



(a) Unprogrammed

(b) Programmed

FIGURE 3–49 Concept of a programmable AND array.

EXAMPLE 3–22

Show the AND array in Figure 3–49(a) programmed for the following outputs: $X_1 = A\overline{B}$, $X_2 = \overline{A}B$, and $X_3 = \overline{A}\,\overline{B}$

Solution

See Figure 3–50.

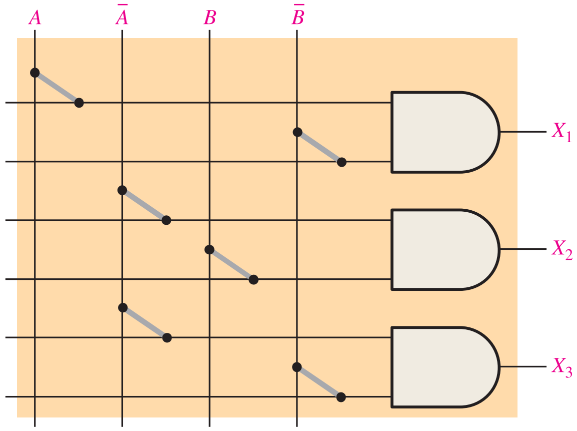


FIGURE 3–50

Related Problem

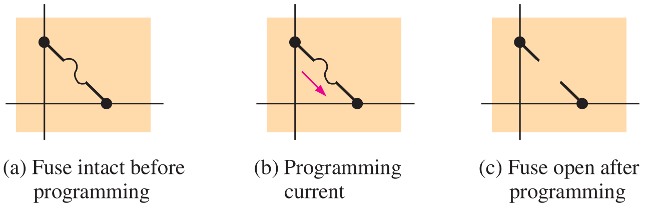
How many rows, columns, and AND gate inputs are required for three input variables in a 3-AND gate array?

Programmable Link Process Technologies

A process technology is the physical method by which a link is made. Several different process technologies are used for programmable links in PLDs.

Fuse Technology

This was the original programmable link technology. It is still used in some SPLDs. The **fuse** is a metal link that connects a row and a column in the interconnection matrix. Before programming, there is a fused connection at each intersection. To program a device, the selected fuses are opened by passing a current through them sufficient to "blow" the fuse and break the connection. The intact fuses remain and provide a connection between the rows and columns. The fuse link is illustrated in Figure 3–51. Programmable logic devices that use fuse technology are one-time programmable (**OTP**).


(a) Fuse intact before programming

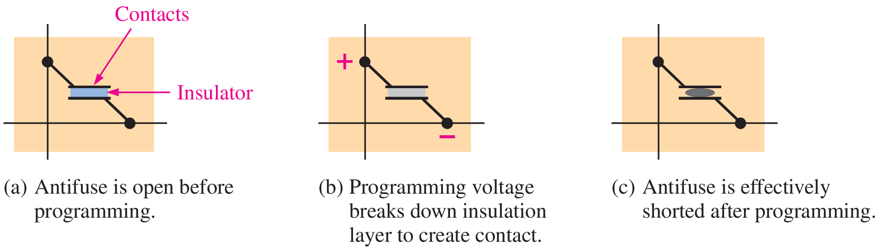
(b) Programming current

(c) Fuse open after programming

FIGURE 3–51 The programmable fuse link.

Antifuse Technology

An **antifuse** programmable link is the opposite of a fuse link. Instead of breaking the connection, a connection is made during programming. An antifuse starts out as an open circuit whereas the fuse starts out as a short circuit. Before programming, there are no connections between the rows and columns in the interconnection matrix. An antifuse is basically two conductors separated by an insulator. To program a device with antifuse technology, a programmer tool applies a sufficient voltage across selected antifuses to break down the insulation between the two conductive materials, causing the insulator to become a low-resistance link. The antifuse link is illustrated in Figure 3–52. An antifuse device is also a one-time programmable (OTP) device.



(a) Antifuse is open before programming.

(b) Programming voltage breaks down insulation layer to create contact.

(c) Antifuse is effectively shorted after programming.

FIGURE 3–52 The programmable antifuse link.

EPROM Technology

In certain programmable logic devices, the programmable links are similar to the memory cells in **EPROMs** (electrically programmable read-only memories). This type of PLD is programmed using a special tool known as a device programmer. The device is inserted into the programmer, which is connected to a computer running the programming software. Most EPROM-based PLDs are one-time programmable (OTP). However, those with windowed packages can be erased with UV (ultraviolet) light and reprogrammed using a standard PLD programming fixture. EPROM process technology uses a special type of MOS transistor, known as a floating-gate transistor, as the programmable link. The floating-gate device utilizes a process called Fowler-Nordheim tunneling to place electrons in the floating-gate structure.

In a programmable AND array, the floating-gate transistor acts as a switch to connect the row line to either a HIGH or a LOW, depending on the input variable. For input variables that are not used, the transistor is programmed to be permanently *off* (open). Figure 3–53 shows one AND gate in a simple array. Variable A controls the state of the transistor in the first column, and variable B controls the transistor in the third column. When a transistor is *off,* like an open switch, the input line to the AND gate is at $+V$ (HIGH). When a transistor is *on,* like a closed switch, the input line is connected to ground (LOW). When variable A

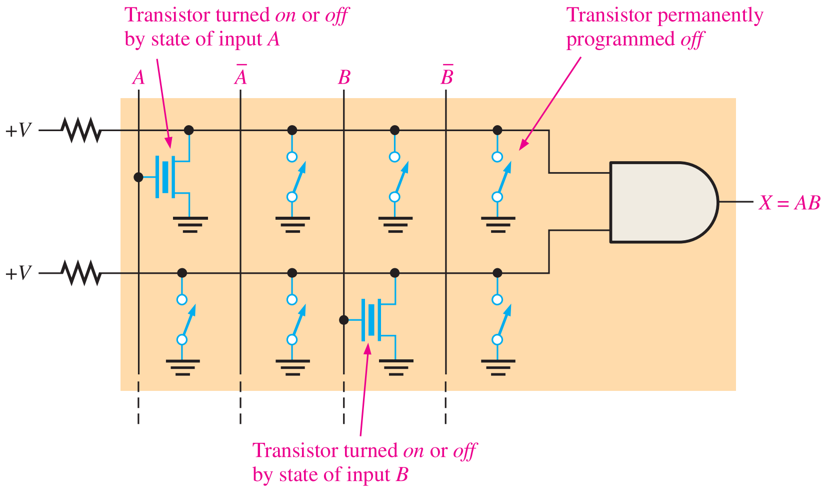

Transistor turned *on* or *off* by state of input A
Transistor permanently programmed *off*
A
$\bar{A}$
B
$\bar{B}$
$+V$
$+V$
$X = AB$
Transistor turned *on* or *off* by state of input B

FIGURE 3–53 A simple AND array with EPROM technology. Only one gate in the array is shown for simplicity.

or *B* is 0 (LOW), the transistor is *on,* keeping the input line to the AND gate LOW. When *A* or *B* is 1 (HIGH), the transistor is *off,* keeping the input line to the AND gate HIGH.

EEPROM Technology

Electrically erasable programmable read-only memory technology is similar to EPROM because it also uses a type of floating-gate transistor in E^2CMOS cells. The difference is that **EEPROM** can be erased and reprogrammed electrically without the need for UV light or special fixtures. An E^2CMOS device can be programmed after being installed on a printed circuit board (PCB), and many can be reprogrammed while operating in a system. This is called **in-system programming (ISP)**. Figure 3–53 can also be used as an example to represent an AND array with EEPROM technology.

InfoNote

Most system-level designs incorporate a variety of devices such as RAMs, ROMs, controllers, and processors that are interconnected by a large quantity of general-purpose logic devices often referred to as “glue” logic. PLDs have come to replace many of the SSI and MSI “glue” devices. The use of PLDs provides a reduction in package count.

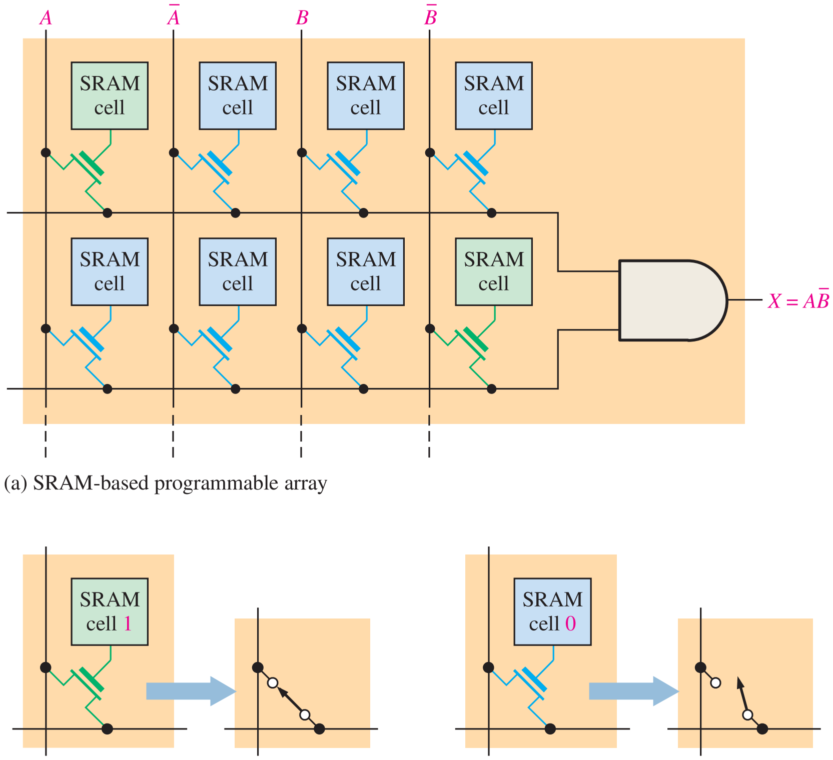
For example, in memory systems, PLDs can be used for memory address decoding and to generate memory write signals as well as other functions.

Flash Technology

Flash technology is based on a single transistor link and is both nonvolatile and reprogrammable. Flash elements are a type of EEPROM but are faster and result in higher density devices than the standard EEPROM link. A detailed discussion of the flash memory element can be found in Chapter 11.

SRAM Technology

Many FPGAs and some CPLDs use a process technology similar to that used in **SRAMs** (static random-access memories). The basic concept of SRAM-based programmable logic arrays is illustrated in Figure 3–54(a). A SRAM-type memory cell is used to turn a transistor *on* or *off* to connect or disconnect rows and columns. For example, when the memory cell contains a 1 (green), the transistor is *on* and connects the associated row and column lines, as shown in part (b). When the memory cell contains a 0 (blue), the transistor is *off* so there is no connection between the lines, as shown in part (c).



(a) SRAM-based programmable array

(b) Transistor *on*

(c) Transistor *off*

FIGURE 3–54 Concept of an AND array with SRAM technology.

SRAM technology is different from the other process technologies discussed because it is a volatile technology. This means that a SRAM cell does not retain data when power is turned *off*. The programming data must be loaded into a memory; and when power is turned *on,* the data from the memory reprograms the SRAM-based PLD.

The fuse, antifuse, EPROM, EEPROM, and flash process technologies are nonvolatile, so they retain their programming when the power is *off*. A fuse is permanently open, an antifuse is permanently closed, and floating-gate transistors used in EPROM and EEPROM-based arrays can retain their *on* or *off* state indefinitely.

Device Programming

The general concept of programming was introduced in Chapter 1, and you have seen how interconnections can be made in a simple array by opening or closing the programmable links. SPLDs, CPLDs, and FPGAs are programmed in essentially the same way. The devices with OTP (one-time programmable) process technologies (fuse, antifuse, or EPROM) must be programmed with a special hardware fixture called a *programmer*. The programmer is connected to a computer by a standard interface cable. Development software is installed on the computer, and the device is inserted into the programmer socket. Most programmers have adapters that allow different types of packages to be plugged in.

EEPROM, flash, and SRAM-based programmable logic devices are reprogrammable and can be reconfigured multiple times. Although a device programmer can be used for this type of device, it is generally programmed initially on a PLD development board, as shown in Figure 3–55. A logic design can be developed using this approach because any necessary changes during the design process can be readily accomplished by simply reprogramming the PLD. A PLD to which a software logic design can be downloaded is called a **target device**. In addition to the target device, development boards typically provide other circuitry and connectors for interfacing to the computer and other peripheral circuits. Also, test points and display devices for observing the operation of the programmed device are included on the development board.

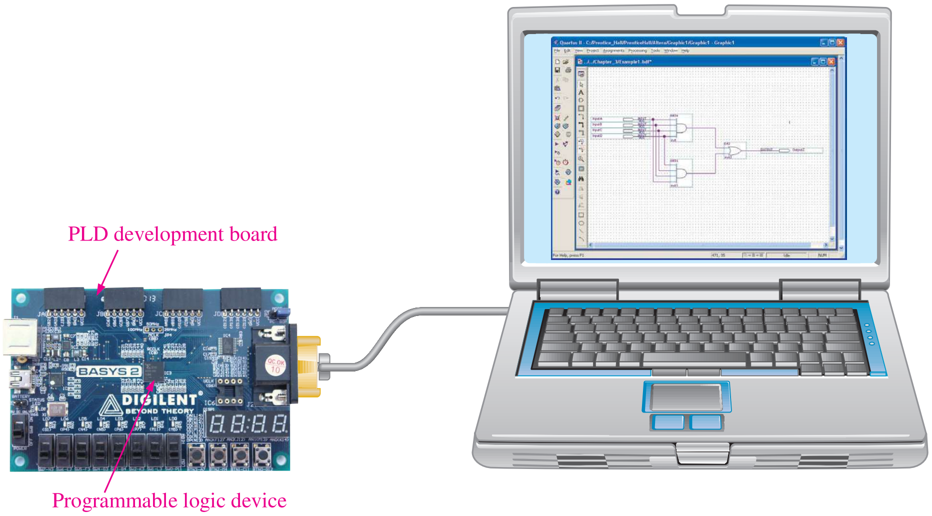


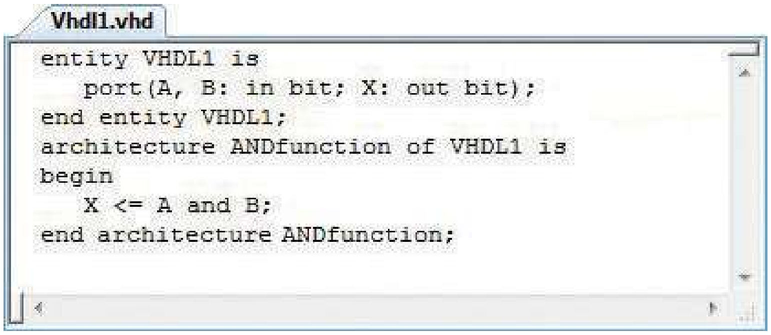
FIGURE 3–55 Programming setup for reprogrammable logic devices. (Photo courtesy of Digilent, Inc.)

Design Entry

As you learned in Chapter 1, design entry is where the logic design is programmed into the development software. The two main ways to enter a design are by text entry or graphic (schematic) entry, and manufacturers of programmable logic provide software packages to support their devices that allow for both methods.

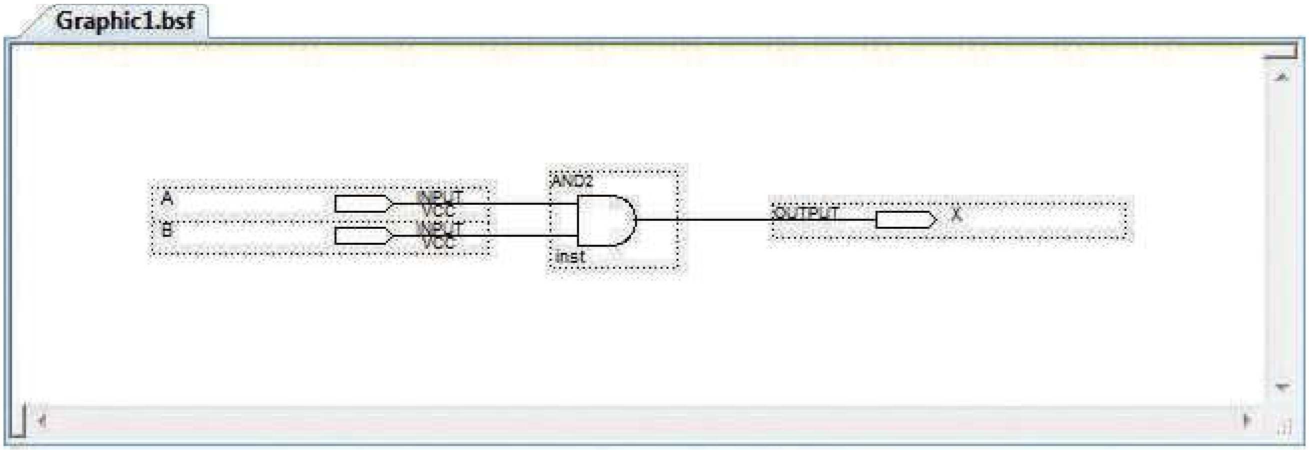
Text entry in most development software, regardless of the manufacturer, supports two or more hardware development languages (HDLs). For example, all software packages support both IEEE standard HDLs, VHDL, and Verilog. Some software packages also support certain proprietary languages such as AHDL.

In **graphic (schematic) entry**, logic symbols such as AND gates and OR gates are placed on the screen and interconnected to form the desired circuit. In this method you use the familiar logic symbols, but the software actually converts each symbol and interconnections to a text file for the computer to use; you do not see this process. A simple example of both a text entry screen and a graphic entry screen for an AND gate is shown in Figure 3–56. As a general rule, graphic entry is used for less-complex logic circuits and text entry, although it can also be used for very simple logic, is used for larger, more complex implementation.



```
entity VHDL1 is
   port(A, B: in bit; X: out bit);
end entity VHDL1;
architecture ANDfunction of VHDL1 is
begin
   X <= A and B;
end architecture ANDfunction;
```

(a) VHDL text entry



(b) Equivalent graphic (schematic) entry

FIGURE 3–56 Examples of design entry of an AND gate.

In-System Programming (ISP)

Certain CPLDs and FPGAs can be programmed after they have been installed on a system printed circuit board (PCB). After a logic design has been developed and fully tested on a development board, it can then be programmed into a "blank" device that is already soldered onto a system board in which it will be operating. Also, if a design change is required, the device on the system board can be reconfigured to incorporate the design modifications.

In a production situation, programming a device on the system board minimizes handling and eliminates the need for keeping stocks of preprogrammed devices. It also rules out the possibility of wrong parts being placed in a product. Unprogrammed (blank) devices can be kept in the warehouse and programmed on-board as needed. This minimizes the capital a business needs for inventories and enhances the quality of its products.

JTAG

The standard established by the Joint Test Action Group is the commonly used name for IEEE Std. 1149.1. The **JTAG** standard was developed to provide a simple method, called boundary scan, for testing programmable devices for functionality as well as testing circuit boards for bad connections—shorted pins, open pins, bad traces, and the like. Also, JTAG has been used as a convenient way of configuring programmable devices in-system. As the demand for field-upgradable products increases, the use of JTAG as a convenient way of reprogramming CPLDs and FPGAs increases.

JTAG-compliant devices have internal dedicated hardware that interprets instructions and data provided by four dedicated signals. These signals are defined by the JTAG standard to be TDI (Test Data In), TDO (Test Data Out), TMS (Test Mode Select), and TCK (Test Clock). The dedicated JTAG hardware interprets instructions and data on the TDI and TMS signals, and drives data out on the TDO signal. The TCK signal is used to clock the process. A JTAG-compliant PLD is represented in Figure 3–57.

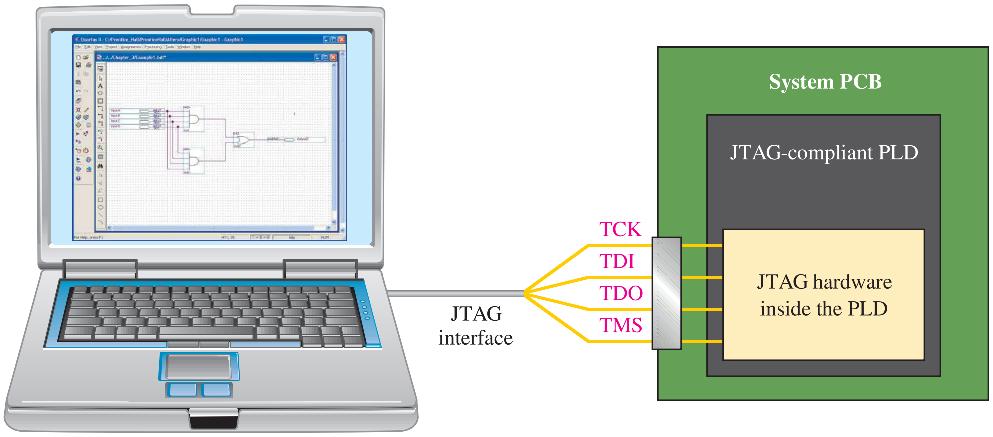


FIGURE 3–57 Simplified illustration of in-system programming via a JTAG interface.

Embedded Processor

Another approach to in-system programming is the use of an embedded microprocessor and memory. The processor is embedded within the system along with the CPLD or FPGA and other circuitry, and it is dedicated to the purpose of in-system configuration of the programmable device.

As you have learned, SRAM-based devices are volatile and lose their programmed data when the power is turned *off.* It is necessary to store the programming data in a PROM (programmable read-only memory), which is nonvolatile. When power is turned *on,* the embedded processor takes control of transferring the stored data from the PROM to the CPLD or FPGA.

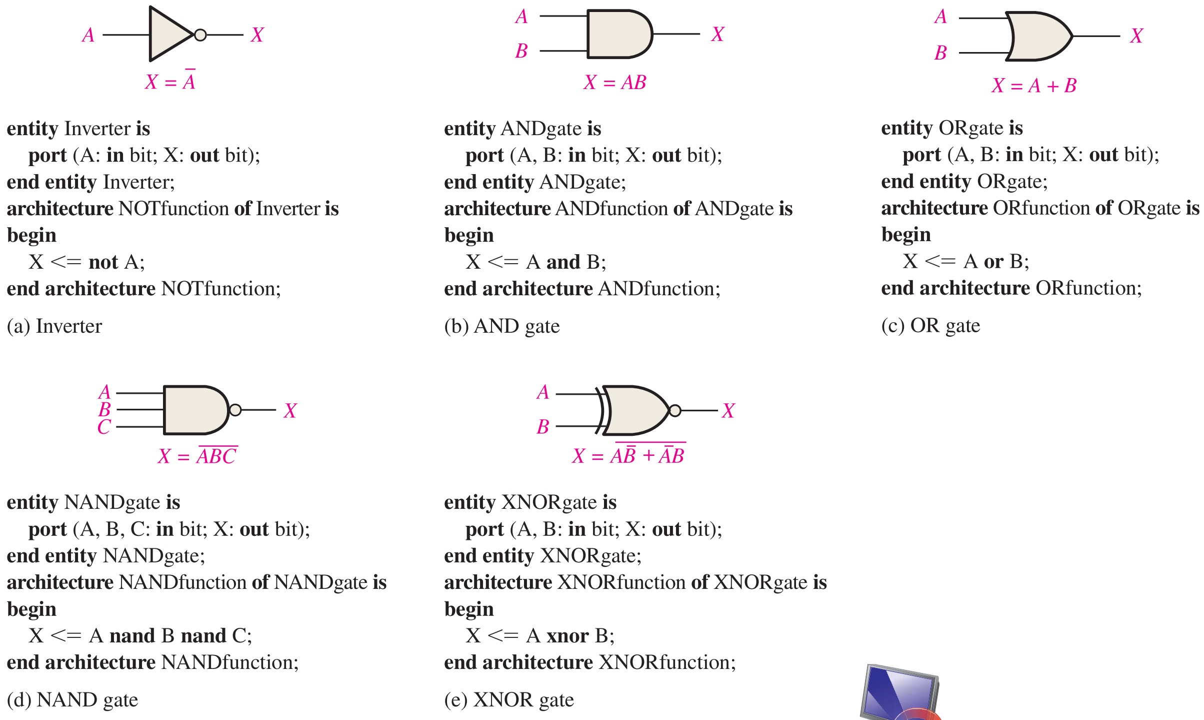
Also, an embedded processor is sometimes used for reconfiguration of a programmable device while the system is running. In this case, design changes are done with software, and the new data are then loaded into a PROM without disturbing the operation of the system. The processor controls the transfer of the data to the device "on-the-fly" at an appropriate time.

VHDL Descriptions of Logic Gates

Hardware description languages (HDLs) differ from software programming languages because HDLs include ways of describing logic connections and characteristics. An HDL implements a logic design in hardware (PLD), whereas a software programming language, such as C or BASIC, instructs existing hardware what to do. The two standard HDLs used for programming

PLDs are VHDL and Verilog. Both of these HDLs have their advocates, but VHDL will be used in this textbook. *A VHDL tutorial is available on the website*.

Figure 3–58 shows **VHDL** programs for gates described in this chapter. Two gates are left as Checkup exercises. VHDL has an *entity/architecture* structure. The **entity** defines the logic element and its inputs/outputs or ports; the **architecture** describes the logic operation. Keywords that are part of the VHDL syntax are shown bold for clarity.



```
entity Inverter is
   port (A: in bit; X: out bit);
end entity Inverter;
architecture NOTfunction of Inverter is
begin
   X <= not A;
end architecture NOTfunction;
```

(a) Inverter

```
entity ANDgate is
   port (A, B: in bit; X: out bit);
end entity ANDgate;
architecture ANDfunction of ANDgate is
begin
   X <= A and B;
end architecture ANDfunction;
```

(b) AND gate

```
entity ORgate is
   port (A, B: in bit; X: out bit);
end entity ORgate;
architecture ORfunction of ORgate is
begin
   X <= A or B;
end architecture ORfunction;
```

(c) OR gate

```
entity NANDgate is
   port (A, B, C: in bit; X: out bit);
end entity NANDgate;
architecture NANDfunction of NANDgate is
begin
   X <= A nand B nand C;
end architecture NANDfunction;
```

(d) NAND gate

```
entity XNORgate is
   port (A, B: in bit; X: out bit);
end entity XNORgate;
architecture XNORfunction of XNORgate is
begin
   X <= A xnor B;
end architecture XNORfunction;
```

(e) XNOR gate

FIGURE 3–58 Logic gates described with VHDL.

SECTION 3–7 CHECKUP

1. List six process technologies used for programmable links in programmable logic.
2. What does the term *volatile* mean in relation to PLDs and which process technology is volatile?
3. What are two design entry methods for programming PLDs and FPGAs?
4. Define JTAG.
5. Write a VHDL description of a 3-input NOR gate.
6. Write a VHDL description of an XOR gate.

3–8 Fixed-Function Logic Gates

Fixed-function logic integrated circuits have been around for a long time and are available in a variety of logic functions. Unlike a PLD, a fixed-function IC comes with logic functions that cannot be programmed in and cannot be altered. The fixed-function logic is on a much smaller scale than the amount of logic that can be programmed into a PLD. Although the trend in technology is definitely toward programmable logic, fixed-function logic is used in specialized applications where PLDs are not the optimum choice. Fixed-function logic devices are sometimes called "glue logic" because of their usefulness in tying together larger units of logic such as PLDs in a system.

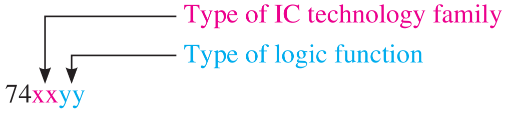
After completing this section, you should be able to

- List common 74 series gate logic functions
- List the major integrated circuit technologies and name some integrated circuit families
- Obtain data sheet information
- Define *propagation delay time*
- Define *power dissipation*
- Define *unit load* and *fan-out*
- Define *speed-power product*

All of the various fixed-function logic devices currently available are implemented in two major categories of circuit technology: **CMOS** (complementary metal-oxide semiconductor) and **bipolar** (also known as **TTL**, transistor-transistor logic). A type of bipolar technology that is available in very limited devices is ECL (emitter-coupled logic). BiCMOS is another integrated circuit technology that combines both bipolar and CMOS. CMOS is the most dominant circuit technology.

74 Series Logic Gate Functions

The 74 series is the standard fixed-function logic devices. The device label format includes one or more letters that indentify the type of logic circuit technology family in the IC package and two or more digits that identify the type of logic function. For example, 74HC04 is a fixed-function IC that has six inverters in a package as indicated by 04. The letters, HC, following the prefix 74 identify the circuit technology family as a type of CMOS logic.

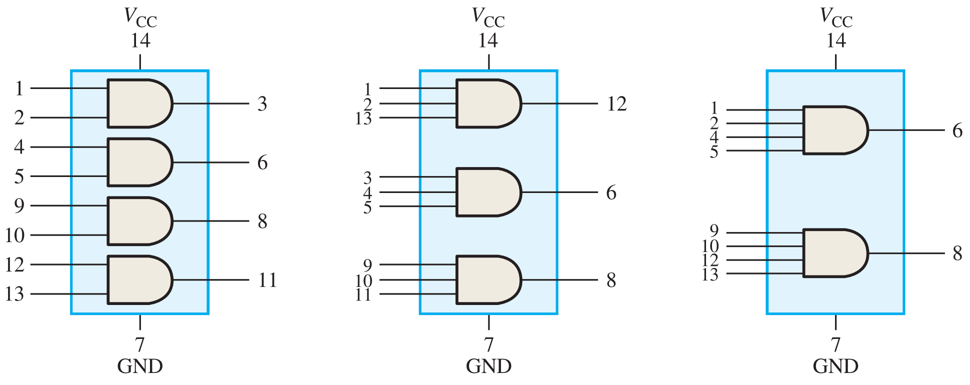

Type of IC technology family

Type of logic function

74xxyy

AND Gate

Figure 3–59 shows three configurations of fixed-function AND gates in the 74 series. The 74xx08 is a quad 2-input AND gate device, the 74xx11 is a triple 3-input AND gate device,



(a) 74xx08

(b) 74xx11

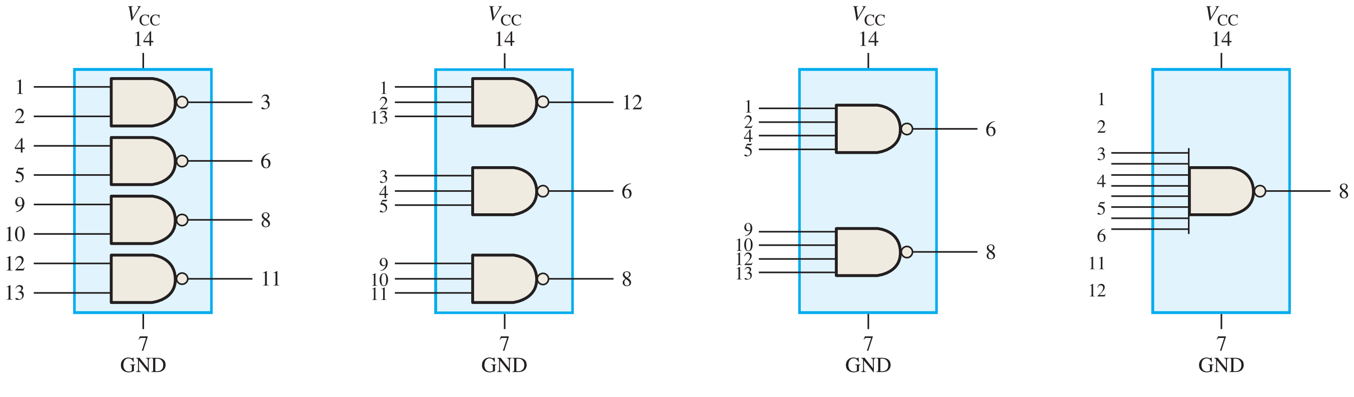
(c) 74xx21

FIGURE 3–59 74 series AND gate devices with pin numbers.

and the 74xx21 is a dual 4-input AND gate device. The label xx can represent any of the integrated circuit technology families such as HC or LS. The numbers on the inputs and outputs are the IC package pin numbers.

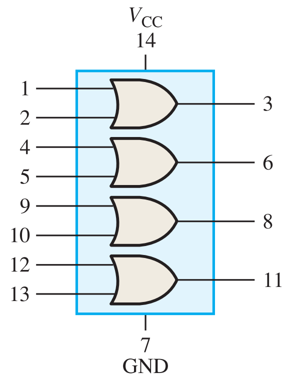
NAND Gate

Figure 3–60 shows four configurations of fixed-function NAND gates in the 74 series. The 74xx00 is a quad 2-input NAND gate device, the 74xx10 is a triple 3-input NAND gate device, the 74xx20 is a dual 4-input NAND gate device, and the 74xx30 is a single 8-input NAND gate device.



(a) 74xx00 (b) 74xx10 (c) 74xx20 (d) 74xx30

FIGURE 3–60 74 series NAND gate devices with package pin numbers.



74xx32

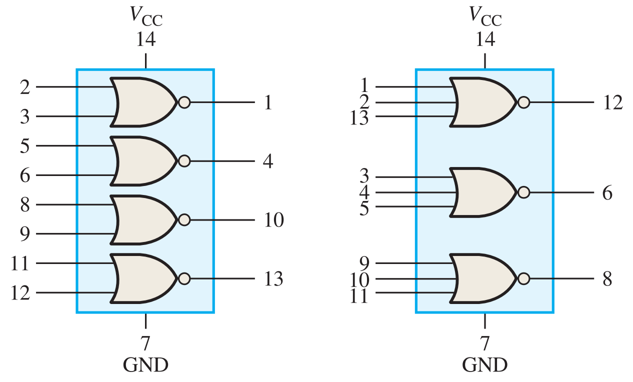
FIGURE 3–61 74 series OR gate device.

OR Gate

Figure 3–61 shows a fixed-function OR gate in the 74 series. The 74xx32 is a quad 2-input OR gate device.

NOR Gate

Figure 3–62 shows two configurations of fixed-function NOR gates in the 74 series. The 74xx02 is a quad 2-input NOR gate device, and the 74xx27 is a triple 3-input NOR gate device.



(a) 74xx02 (b) 74xx27

FIGURE 3–62 74 series NOR gate devices.

XOR Gate

Figure 3–63 shows a fixed-function XOR (exclusive-OR) gate in the 74 series. The 74xx86 is a quad 2-input XOR gate.

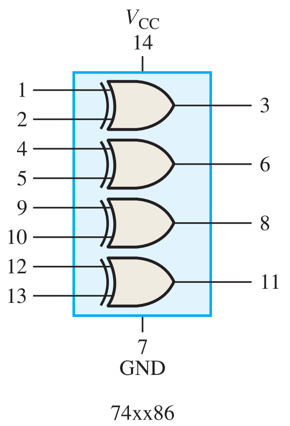
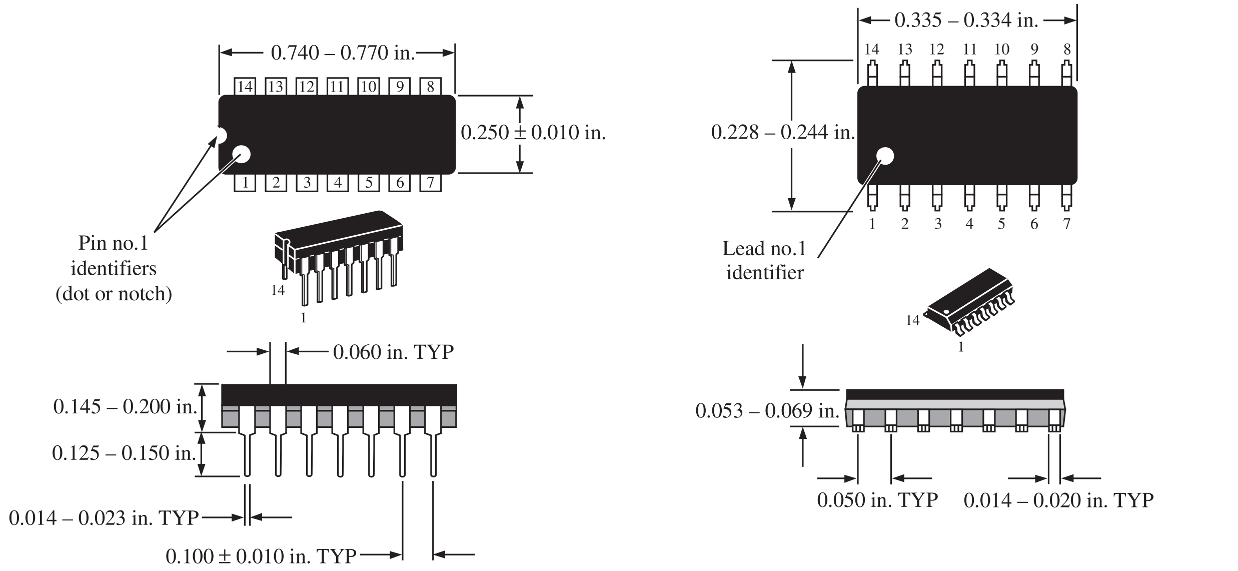


FIGURE 3–63 74 series XOR gate.

IC Packages

All of the 74 series CMOS are pin-compatible with the same types of devices in bipolar. This means that a CMOS digital IC such as the 74HC00 (quad 2-input NAND), which contains four 2-input NAND gates in one IC package, has the identical package pin numbers for each input and output as does the corresponding bipolar device. Typical IC gate packages, the dual in-line package (DIP) for plug-in or feedthrough mounting and the small-outline integrated circuit (SOIC) package for surface mounting, are shown in Figure 3–64. In some cases, other types of packages are also available. The SOIC package is significantly smaller than the DIP. Packages with a single gate are known as *little logic*. Most logic gate functions are available and are implemented in a CMOS circuit technology. Typically, the gates have only two inputs and have a different designation than multigate devices. For example, the 74xx1G00 is a single 2-input NAND gate.



(a) 14-pin dual in-line package (DIP) for feedthrough mounting

(b) 14-pin small outline package (SOIC) for surface mounting

FIGURE 3–64 Typical dual in-line (DIP) and small-outline (SOIC) packages showing pin numbers and basic dimensions.

HandsOnTip

Handling Precautions for CMOS

CMOS logic is very sensitive to static charge and can be damaged by ESD (electrostatic discharge) if not handled properly as follows:

1. Store and ship in conductive foam.
2. Connect instruments to earth ground.
3. Connect wrist to earth ground through a large series resistor.
4. Do not remove devices from circuit with power on.
5. Do not apply signal voltage when power is off.

74 Series Logic Circuit Families

Although many logic circuit families have become obsolete and some are rapidly on the decline, others are still very active and available. CMOS is the most available and most popular type of logic circuit technology, and the HC (high-speed CMOS) family is the most recommended for new projects. For bipolar, the LS (low-power schottky) family is the most widely used. The HCT, which a variation of the HC family, is compatible with bipolar devices such as LS.

Table 3–14 lists many logic circuit technology families. Because the active status of any given logic family is always in flux, check with a manufacturer, such as Texas Instruments, for information on active/nonactive status and availability for a logic function in a given circuit technology.

TABLE 3–14

74 series logic families based on circuit technology.

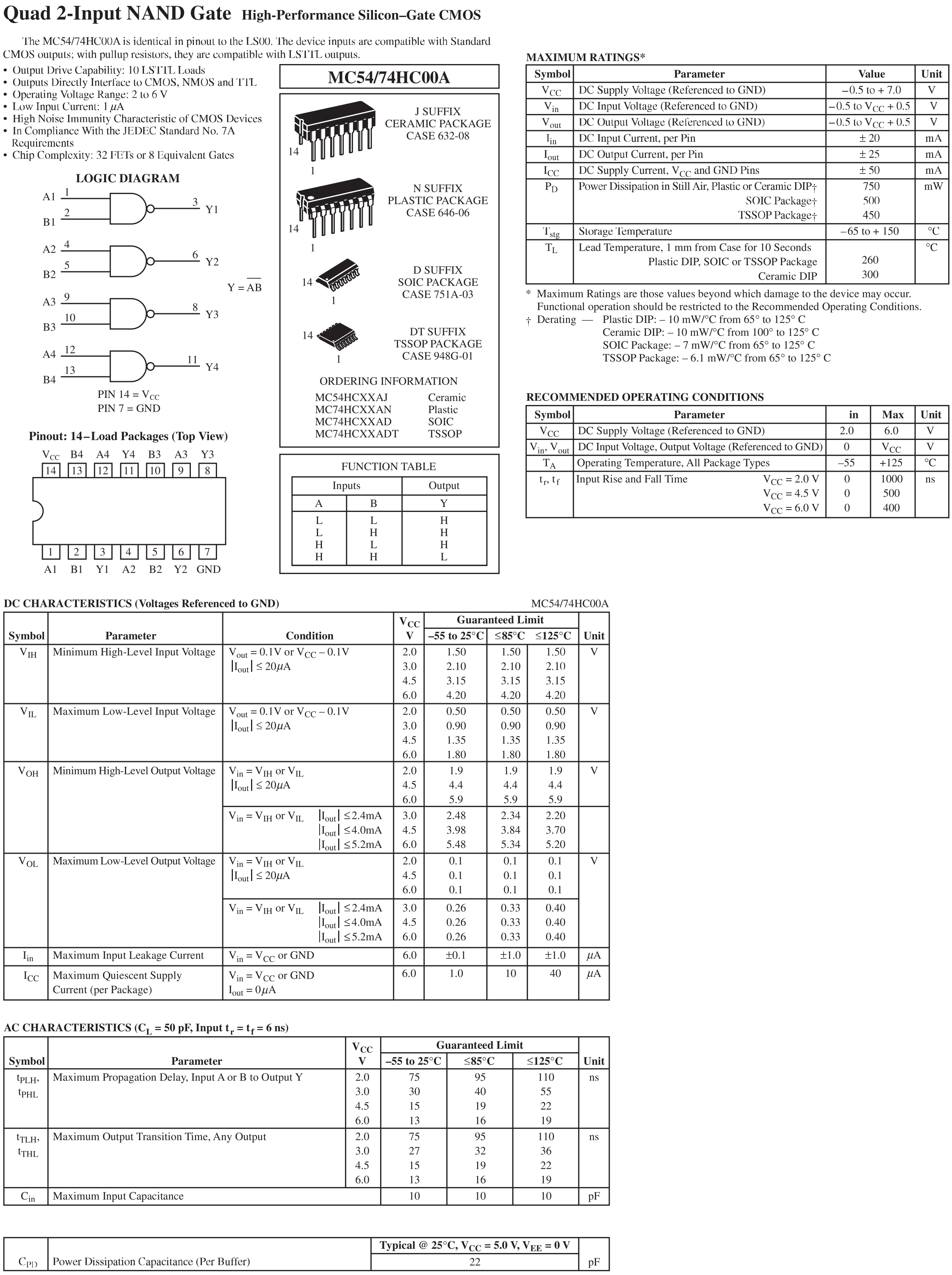
Circuit Type	Description	Circuit Technology
ABT	Advanced BiCMOS	BiCMOS
AC	Advanced CMOS	CMOS
ACT	Bipolar compatible AC	CMOS
AHC	Advanced high-speed CMOS	CMOS
AHCT	Bipolar compatible AHC	CMOS
ALB	Advanced low-voltage BiCMOS	BiCMOS
ALS	Advanced low-power Schottky	Bipolar
ALVC	Advanced low-voltage CMOS	CMOS
AUC	Advanced ultra-low-voltage CMOS	CMOS
AUP	Advanced ultra-low-power CMOS	CMOS
AS	Advanced Schottky	Bipolar
AVC	Advanced very-low-power CMOS	CMOS
BCT	Standard BiCMOS	BiCMOS
F	Fast	Bipolar
FCT	Fast CMOS technology	CMOS
HC	High-speed CMOS	CMOS
HCT	Bipolar compatible HC	CMOS
LS	Low-power Schottky	Bipolar
LV-A	Low-voltage CMOS	CMOS
LV-AT	Bipolar compatible LV-A	CMOS
LVC	Low-voltage CMOS	CMOS
LVT	Low-voltage biCMOS	BiCMOS
S	Schottky	Bipolar

The type of integrated circuit technology has nothing to do with the logic function itself. For example, the 74HC00, 74HCT00, and 74LS00 are all quad 2-input NAND gates with identical package pin configurations. The differences among these three logic devices are in the electrical and performance characteristics such as power consumption, dc supply voltage, switching speed, and input/output voltage levels. CMOS and bipolar circuits are implemented with two different types of transistors. Figures 3–65 and 3–66 show partial data sheets for the 74HC00A quad 2-input NAND gate in CMOS and in bipolar technologies, respectively.

Performance Characteristics and Parameters

High-speed logic has a short propagation delay time.

Several things define the performance of a logic circuit. These performance characteristics are the switching speed measured in terms of the propagation delay time, the power



Quad 2-Input NAND Gate **High-Performance Silicon–Gate CMOS**

The MC54/74HC00A is identical in pinout to the LS00. The device inputs are compatible with Standard CMOS outputs; with pullup resistors, they are compatible with LSTTL outputs.

- Output Drive Capability: 10 LSTTL Loads
- Outputs Directly Interface to CMOS, NMOS and TTL
- Operating Voltage Range: 2 to 6 V
- Low Input Current: 1 μA
- High Noise Immunity Characteristic of CMOS Devices
- In Compliance With the JEDEC Standard No. 7A Requirements
- Chip Complexity: 32 FETs or 8 Equivalent Gates

LOGIC DIAGRAM

$Y = \overline{AB}$

PIN 14 = V_{CC}
PIN 7 = GND

Pinout: 14–Load Packages (Top View)

MC54/74HC00A

J SUFFIX
CERAMIC PACKAGE
CASE 632-08

N SUFFIX
PLASTIC PACKAGE
CASE 646-06

D SUFFIX
SOIC PACKAGE
CASE 751A-03

DT SUFFIX
TSSOP PACKAGE
CASE 948G-01

ORDERING INFORMATION

MC54HCXXAJ	Ceramic
MC74HCXXAN	Plastic
MC74HCXXAD	SOIC
MC74HCXXADT	TSSOP

FUNCTION TABLE

Inputs		Output
A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

MAXIMUM RATINGS*

Symbol	Parameter	Value	Unit
V_{CC}	DC Supply Voltage (Referenced to GND)	–0.5 to + 7.0	V
V_{in}	DC Input Voltage (Referenced to GND)	–0.5 to V_{CC} + 0.5	V
V_{out}	DC Output Voltage (Referenced to GND)	–0.5 to V_{CC} + 0.5	V
I_{in}	DC Input Current, per Pin	± 20	mA
I_{out}	DC Output Current, per Pin	± 25	mA
I_{CC}	DC Supply Current, V_{CC} and GND Pins	± 50	mA
P_D	Power Dissipation in Still Air, Plastic or Ceramic DIP†	750	mW
	SOIC Package†	500	
	TSSOP Package†	450	
T_{stg}	Storage Temperature	–65 to + 150	°C
T_L	Lead Temperature, 1 mm from Case for 10 Seconds		°C
	Plastic DIP, SOIC or TSSOP Package	260	
	Ceramic DIP	300	

* Maximum Ratings are those values beyond which damage to the device may occur. Functional operation should be restricted to the Recommended Operating Conditions.

† Derating — Plastic DIP: – 10 mW/°C from 65° to 125° C
Ceramic DIP: – 10 mW/°C from 100° to 125° C
SOIC Package: – 7 mW/°C from 65° to 125° C
TSSOP Package: – 6.1 mW/°C from 65° to 125° C

RECOMMENDED OPERATING CONDITIONS

Symbol	Parameter		in	Max	Unit
V_{CC}	DC Supply Voltage (Referenced to GND)		2.0	6.0	V
V_{in}, V_{out}	DC Input Voltage, Output Voltage (Referenced to GND)		0	V_{CC}	V
T_A	Operating Temperature, All Package Types		–55	+125	°C
t_r, t_f	Input Rise and Fall Time	V_{CC} = 2.0 V	0	1000	ns
		V_{CC} = 4.5 V	0	500	
		V_{CC} = 6.0 V	0	400	

DC CHARACTERISTICS (Voltages Referenced to GND) MC54/74HC00A

Symbol	Parameter	Condition	V_{CC} V	Guaranteed Limit –55 to 25°C	≤85°C	≤125°C	Unit
V_{IH}	Minimum High-Level Input Voltage	V_{out} = 0.1V or V_{CC} – 0.1V $\lvert I_{out}\rvert \le 20\mu A$	2.0	1.50	1.50	1.50	V
			3.0	2.10	2.10	2.10	
			4.5	3.15	3.15	3.15	
			6.0	4.20	4.20	4.20	
V_{IL}	Maximum Low-Level Input Voltage	V_{out} = 0.1V or V_{CC} – 0.1V $\lvert I_{out}\rvert \le 20\mu A$	2.0	0.50	0.50	0.50	V
			3.0	0.90	0.90	0.90	
			4.5	1.35	1.35	1.35	
			6.0	1.80	1.80	1.80	
V_{OH}	Minimum High-Level Output Voltage	$V_{in} = V_{IH}$ or V_{IL} $\lvert I_{out}\rvert \le 20\mu A$	2.0	1.9	1.9	1.9	V
			4.5	4.4	4.4	4.4	
			6.0	5.9	5.9	5.9	
		$V_{in} = V_{IH}$ or V_{IL} $\lvert I_{out}\rvert \le 2.4mA$	3.0	2.48	2.34	2.20	
		$\lvert I_{out}\rvert \le 4.0mA$	4.5	3.98	3.84	3.70	
		$\lvert I_{out}\rvert \le 5.2mA$	6.0	5.48	5.34	5.20	
V_{OL}	Maximum Low-Level Output Voltage	$V_{in} = V_{IH}$ or V_{IL} $\lvert I_{out}\rvert \le 20\mu A$	2.0	0.1	0.1	0.1	V
			4.5	0.1	0.1	0.1	
			6.0	0.1	0.1	0.1	
		$V_{in} = V_{IH}$ or V_{IL} $\lvert I_{out}\rvert \le 2.4mA$	3.0	0.26	0.33	0.40	
		$\lvert I_{out}\rvert \le 4.0mA$	4.5	0.26	0.33	0.40	
		$\lvert I_{out}\rvert \le 5.2mA$	6.0	0.26	0.33	0.40	
I_{in}	Maximum Input Leakage Current	$V_{in} = V_{CC}$ or GND	6.0	±0.1	±1.0	±1.0	μA
I_{CC}	Maximum Quiescent Supply Current (per Package)	$V_{in} = V_{CC}$ or GND $I_{out} = 0\mu A$	6.0	1.0	10	40	μA

AC CHARACTERISTICS (C_L = 50 pF, Input $t_r = t_f$ = 6 ns)

Symbol	Parameter	V_{CC} V	Guaranteed Limit –55 to 25°C	≤85°C	≤125°C	Unit
t_{PLH}, t_{PHL}	Maximum Propagation Delay, Input A or B to Output Y	2.0	75	95	110	ns
		3.0	30	40	55	
		4.5	15	19	22	
		6.0	13	16	19	
t_{TLH}, t_{THL}	Maximum Output Transition Time, Any Output	2.0	75	95	110	ns
		3.0	27	32	36	
		4.5	15	19	22	
		6.0	13	16	19	
C_{in}	Maximum Input Capacitance		10	10	10	pF

		Typical @ 25°C, V_{CC} = 5.0 V, V_{EE} = 0 V	
C_{PD}	Power Dissipation Capacitance (Per Buffer)	22	pF

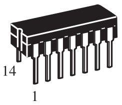
FIGURE 3–65 CMOS logic. Partial data sheet for a 54/74HC00A quad 2-input NAND gate. The 54 prefix indicates military grade and the 74 prefix indicates commercial grade.

QUAD 2-INPUT NAND GATE

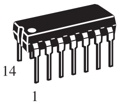
• ESD > 3500 Volts

SN54/74LS00

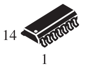
QUAD 2-INPUT NAND GATE
LOW POWER SCHOTTKY



J SUFFIX
CERAMIC
CASE 632-08



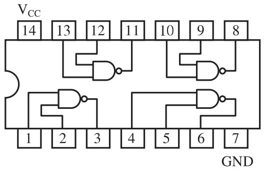
N SUFFIX
PLASTIC
CASE 646-06



D SUFFIX
SOIC
CASE 751A-02

ORDERING INFORMATION

SN54LSXXJ	Ceramic
SN74LSXXN	Plastic
SN74LSXXD	SOIC



SN54/74LS00

DC CHARACTERISTICS OVER OPERATING TEMPERATURE RANGE (unless otherwise specified)

			Limits				
Symbol	**Parameter**		**Min**	**Typ**	**Max**	**Unit**	**Test Conditions**
V_{IH}	Input HIGH Voltage		2.0			V	Guaranteed Input HIGH Voltage for All Inputs
V_{IL}	Input LOW Voltage	54			0.7	V	Guaranteed Input LOW Voltage for All Inputs
		74			0.8		
V_{IK}	Input Clamp Diode Voltage			–0.65	–1.5	V	V_{CC} = MIN, I_{IN} = –18 mA
V_{OH}	Ouput HIGH Voltage	54	2.5	3.5		V	V_{CC} = MIN, I_{OH} = MAX, V_{IN} = V_{IH} or V_{IL} per Truth Table
		74	2.7	3.5		V	
V_{OL}	Ouput LOW Voltage	54, 74		0.25	0.4	V	I_{OL} = 4.0 mA V_{CC} = V_{CC} MIN, V_{IN} = V_{IL} or V_{IH} per Truth Table
		74		0.35	0.5	V	I_{OL} = 8.0 mA
I_{IH}	Input HIGH Current				20	μA	V_{CC} = MAX, V_{IN} = 2.7 V
					0.1	mA	V_{CC} = MAX, V_{IN} = 7.0 V
I_{IL}	Input LOW Current				–0.4	mA	V_{CC} = MAX, I_N = 0.4 V
I_{OS}	Short Circuit Current (Note 1)		–20		–100	mA	V_{CC} = MAX
I_{CC}	Power Supply Current Total, Output HIGH				1.6	mA	V_{CC} = MAX
	Total, Output LOW				4.4		

NOTE 1: Not more than one output should be shorted at a time, nor for more than 1 second.

AC CHARACTERISTICS (T_A = 25°C)

		Limits				
Symbol	**Parameter**	**Min**	**Typ**	**Max**	**Unit**	**Test Conditions**
t_{PLH}	Turn-Off Delay, Input to Output		9.0	15	ns	V_{CC} = 5.0 V C_L = 15 pF
t_{PHL}	Turn-On Delay, Input to Output		10	15	ns	

GUARANTEED OPERATING RANGES

Symbol	Parameter		Min	Typ	Max	Unit
V_{CC}	Supply Voltage	54 74	4.5 4.75	5.0 5.0	5.5 5.25	V
T_A	Operating Ambient Temperature Range	54 74	–55 0	25 25	125 70	°C
I_{OH}	Output Current — High	54, 74			–0.4	mA
I_{OL}	Output Current — Low	54 74			4.0 8.0	mA

FIGURE 3–66 Bipolar logic. Partial data sheet for a 54/74LS00 quad 2-input NAND gate.

dissipation, the fan-out or drive capability, the speed-power product, the dc supply voltage, and the input/output logic levels.

Propagation Delay Time

This parameter is a result of the limitation on switching speed or frequency at which a logic circuit can operate. The terms *low speed* and *high speed,* applied to logic circuits, refer to the propagation delay time. The shorter the propagation delay, the higher the switching speed of the circuit and thus the higher the frequency at which it can operate.

Propagation delay time, t_P, of a logic gate is the time interval between the transition of an input pulse and the occurrence of the resulting transition of the output pulse. There are two different measurements of propagation delay time associated with a logic gate that apply to all the types of basic gates:

- t_{PHL}: The time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse, with the output changing from the HIGH level to the LOW level (HL).
- t_{PLH}: The time between a specified reference point on the input pulse and a corresponding reference point on the resulting output pulse, with the output changing from the LOW level to the HIGH level (LH).

For the HCT family CMOS, the propagation delay is 7 ns, for the AC family it is 5 ns, and for the ALVC family it is 3 ns. For standard-family bipolar (TTL) gates, the typical propagation delay is 11 ns and for F family gates it is 3.3 ns. All specified values are dependent on certain operating conditions as stated on a data sheet.

EXAMPLE 3–23

Show the propagation delay times of an inverter.

Solution

An input/output pulse of an inverter is shown in Figure 3–67, and the propagation delay times, t_{PHL} and t_{PLH}, are indicated. In this case, the delays are measured between the 50% points of the corresponding edges of the input and output pulses. The values of t_{PHL} and t_{PLH} are not necessarily equal but in many cases they are the same.

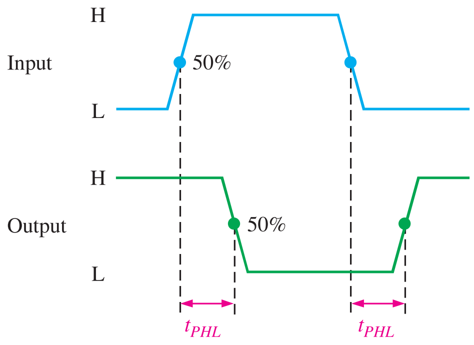


FIGURE 3–67

Related Problem

One type of logic gate has a specified maximum t_{PLH} and t_{PHL} of 10 ns. For another type of gate the value is 4 ns. Which gate can operate at the highest frequency?

DC Supply Voltage (V_{CC})

The typical dc supply voltage for CMOS logic is either 5 V, 3.3 V, 2.5 V, or 1.8 V, depending on the category. An advantage of CMOS is that the supply voltages can vary over a wider range than for bipolar logic. The 5 V CMOS can tolerate supply variations from 2 V to 6 V and still operate properly although propagation delay time and power dissipation are significantly affected. The 3.3 V CMOS can operate with supply voltages from 2 V to 3.6 V. The typical dc supply voltage for bipolar logic is 5.0 V with a minimum of 4.5 V and a maximum of 5.5 V.

Power Dissipation

The **power dissipation**, P_D, of a logic gate is the product of the dc supply voltage and the average supply current. Normally, the supply current when the gate output is LOW is greater than when the gate output is HIGH. The manufacturer's data sheet usually designates the supply current for the LOW output state as I_{CCL} and for the HIGH state as I_{CCH}. The average supply current is determined based on a 50% duty cycle (output LOW half the time and HIGH half the time), so the average power dissipation of a logic gate is

A lower power dissipation means less current from the dc supply.

$$P_D = V_{CC}\left(\frac{I_{CCH} + I_{CCL}}{2}\right) \qquad \text{Equation 3–2}$$

CMOS gates have very low power dissipations compared to the bipolar family. However, the power dissipation of CMOS is dependent on the frequency of operation. At zero frequency the quiescent power is typically in the microwatt/gate range, and at the maximum operating frequency it can be in the low milliwatt range; therefore, power is sometimes specified at a given frequency. The HC family, for example, has a power of 2.75 μW/gate at 0 Hz (quiescent) and 600 μW/gate at 1 MHz.

Power dissipation for bipolar gates is independent of frequency. For example, the ALS family uses 1.4 mW/gate regardless of the frequency and the F family uses 6 mW/gate.

Input and Output Logic Levels

V_{IL} is the LOW level input voltage for a logic gate, and V_{IH} is the HIGH level input voltage. The 5 V CMOS accepts a maximum voltage of 1.5 V as V_{IL} and a minimum voltage of 3.5 V as V_{IH}. Bipolar logic accepts a maximum voltage of 0.8 V as V_{IL} and a minimum voltage of 2 V as V_{IH}.

V_{OL} is the LOW level output voltage and V_{OH} is the HIGH level output voltage. For 5 V CMOS, the maximum V_{OL} is 0.33 V and the minimum V_{OH} is 4.4 V. For bipolar logic, the maximum V_{OL} is 0.4 V and the minimum V_{OH} is 2.4 V. All values depend on operating conditions as specified on the data sheet.

Speed-Power Product (SPP)

This parameter (**speed-power product**) can be used as a measure of the performance of a logic circuit taking into account the propagation delay time and the power dissipation. It is especially useful for comparing the various logic gate series within the CMOS and bipolar technology families or for comparing a CMOS gate to a TTL gate.

The SPP of a logic circuit is the product of the propagation delay time and the power dissipation and is expressed in joules (J), which is the unit of energy. The formula is

$$\boldsymbol{SPP = t_p P_D} \qquad \text{Equation 3–3}$$

EXAMPLE 3–24

A certain gate has a propagation delay of 5 ns and $I_{CCH} = 1$ mA and $I_{CCL} = 2.5$ mA with a dc supply voltage of 5 V. Determine the speed-power product.

Solution

$$P_D = V_{CC}\left(\frac{I_{CCH} + I_{CCL}}{2}\right) = 5\text{ V}\left(\frac{1\text{ mA} + 2.5\text{ mA}}{2}\right) = 5\text{ V}(1.75\text{ mA}) = 8.75\text{ mW}$$

$$SPP = (5\text{ ns})(8.75\text{ mW}) = \mathbf{43.75\ pJ}$$

Related Problem

If the propagation delay of a gate is 15 ns and its SPP is 150 pJ, what is its average power dissipation?

Fan-Out and Loading

The **fan-out** of a logic gate is the maximum number of inputs of the same series in an IC family that can be connected to a gate's output and still maintain the output voltage levels within specified limits. Fan-out is a significant parameter only for bipolar logic because of the type of circuit technology. Since very high impedances are associated with CMOS circuits, the fan-out is very high but depends on frequency because of capacitive effects.

A higher fan-out means that a gate output can be connected to more gate inputs.

Fan-out is specified in terms of **unit loads**. A unit load for a logic gate equals one input to a like circuit. For example, a unit load for a 74LS00 NAND gate equals *one* input to another logic gate in the 74LS family (not necessarily a NAND gate). Because the current from a LOW input (I_{IL}) of a 74LS00 gate is 0.4 mA and the current that a LOW output (I_{OL}) can accept is 8.0 mA, the number of unit loads that a 74LS00 gate can drive in the LOW state is

$$\text{Unit loads} = \frac{I_{OL}}{I_{IL}} = \frac{8.0\text{ mA}}{0.4\text{ mA}} = 20$$

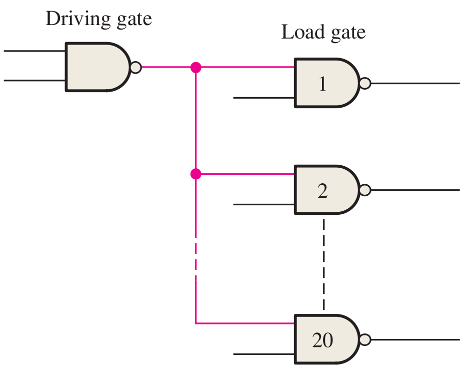
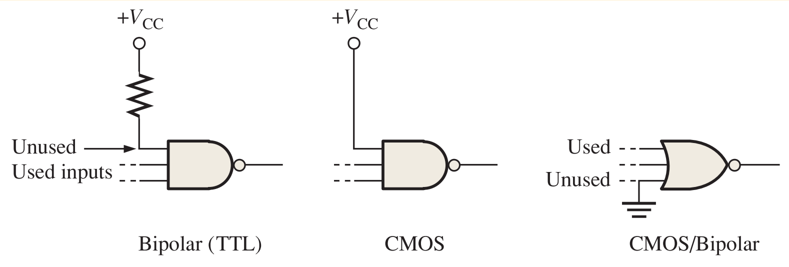


FIGURE 3–68 The LS family NAND gate output fans out to a maximum of 20 LS family gate inputs.

Figure 3–68 shows LS logic gates driving a number of other gates of the same circuit technology, where the number of gates depends on the particular circuit technology. For example, as you have seen, the maximum number of gate inputs (unit loads) that a 74LS family bipolar gate can drive is 20.

HandsOnTip

Unused gate inputs for bipolar (TTL) and CMOS should be connected to the appropriate logic level (HIGH or LOW). For AND/NAND, it is recommended that unused inputs be connected to V_{CC} (through a 1.0 kΩ resistor with bipolar) and for OR/NOR, unused inputs should be connected to ground.

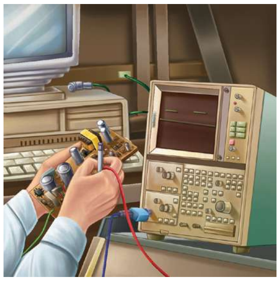


SECTION 3–8 CHECKUP

1. How is fixed-function logic different than PLD logic?
2. List the two types of IC technologies that are the most widely used.
3. Identify the following IC logic designators:

   (a) LS **(b)** HC **(c)** HCT
4. Which IC technology generally has the lowest power dissipation?
5. What does the term *hex inverter* mean? What does *quad 2-input NAND* mean?
6. A positive pulse is applied to an inverter input. The time from the leading edge of the input to the leading edge of the output is 10 ns. The time from the trailing edge of the input to the trailing edge of the output is 8 ns. What are the values of t_{PLH} and t_{PHL}?
7. A certain gate has a propagation delay time of 6 ns and a power dissipation of 3 mW. Determine the speed-power product?
8. Define I_{CCL} and I_{CCH}.
9. Define V_{IL} and V_{IH}.
10. Define V_{OL} and V_{OH}.

3–9 Troubleshooting



Troubleshooting is the process of recognizing, isolating, and correcting a fault or failure in a circuit or system. To be an effective troubleshooter, you must understand how the circuit or system is supposed to work and be able to recognize incorrect performance. For example, to determine whether or not a certain logic gate is faulty, you must know what the output should be for given inputs.

After completing this section, you should be able to

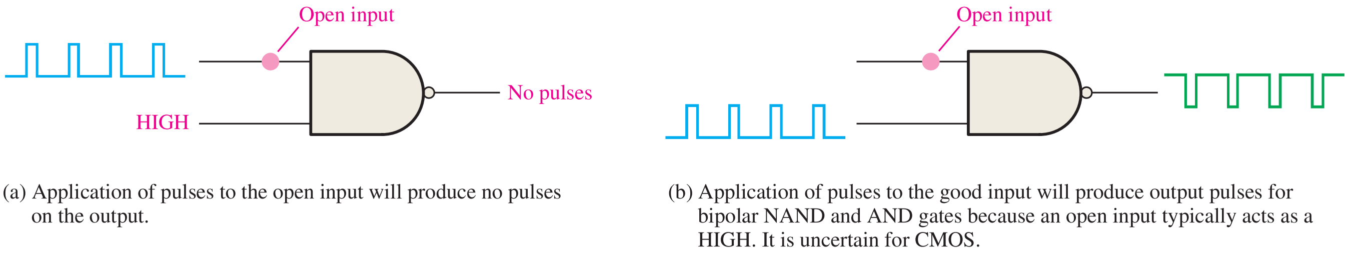
- Test for internally open inputs and outputs in IC gates
- Recognize the effects of a shorted IC input or output
- Test for external faults on a PCB board
- Troubleshoot a simple frequency counter using an oscillosope

Internal Failures of IC Logic Gates

Opens and shorts are the most common types of internal gate failures. These can occur on the inputs or on the output of a gate inside the IC package. *Before attempting any troubleshooting, check for proper dc supply voltage and ground.*

Effects of an Internally Open Input

An internal open is the result of an open component on the chip or a break in the tiny wire connecting the IC chip to the package pin. An open input prevents a signal on that input from getting to the output of the gate, as illustrated in Figure 3–69(a) for the case of a 2-input NAND gate. An open TTL (bipolar) input acts effectively as a HIGH level, so pulses applied to the good input get through to the NAND gate output as shown in Figure 3–69(b).



(a) Application of pulses to the open input will produce no pulses on the output.

(b) Application of pulses to the good input will produce output pulses for bipolar NAND and AND gates because an open input typically acts as a HIGH. It is uncertain for CMOS.

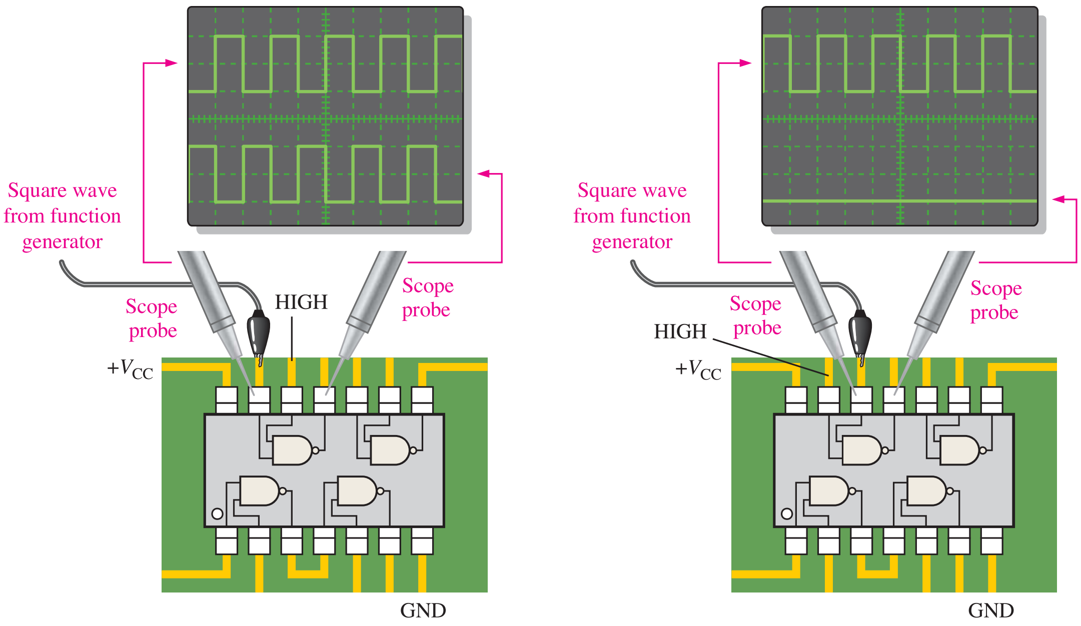
FIGURE 3–69 The effect of an open input on a NAND gate.

Conditions for Testing Gates

When testing a NAND gate or an AND gate, always make sure that the inputs that are not being pulsed are HIGH to enable the gate. When checking a NOR gate or an OR gate, always make sure that the inputs that are not being pulsed are LOW. When checking an XOR or XNOR gate, the level of the nonpulsed input does not matter because the pulses on the other input will force the inputs to alternate between the same level and opposite levels.

Troubleshooting an Open Input

Troubleshooting this type of failure is easily accomplished with an oscilloscope and function generator, as demonstrated in Figure 3–70 for the case of a quad 2-input NAND gate package. When measuring digital signals with a scope, always use dc coupling.



(a) Pin 13 input and pin 11 output OK

(b) Pin 12 input is open.

FIGURE 3–70 Troubleshooting a NAND gate for an open input.

The first step in troubleshooting an IC that is suspected of being faulty is to make sure that the dc supply voltage (V_{CC}) and ground are at the appropriate pins of the IC. Next, apply continuous pulses to one of the inputs to the gate, making sure that the other input is HIGH (in the case of a NAND gate). In Figure 3–70(a), start by applying a pulse waveform to pin 13, which is one of the inputs to the suspected gate. If a pulse waveform is indicated on the output (pin 11 in this case), then the pin 13 input is not open. By the way, this also proves that the output is not open. Next, apply the pulse waveform to the other gate input (pin 12), making sure the other input is HIGH. There is no pulse waveform on the output at pin 11 and the output is LOW, indicating that the pin 12 input is open, as shown in Figure 3–70(b). The input not being pulsed must be HIGH for the case of a NAND gate or AND gate. If this were a NOR gate, the input not being pulsed would have to be LOW.

Effects of an Internally Open Output

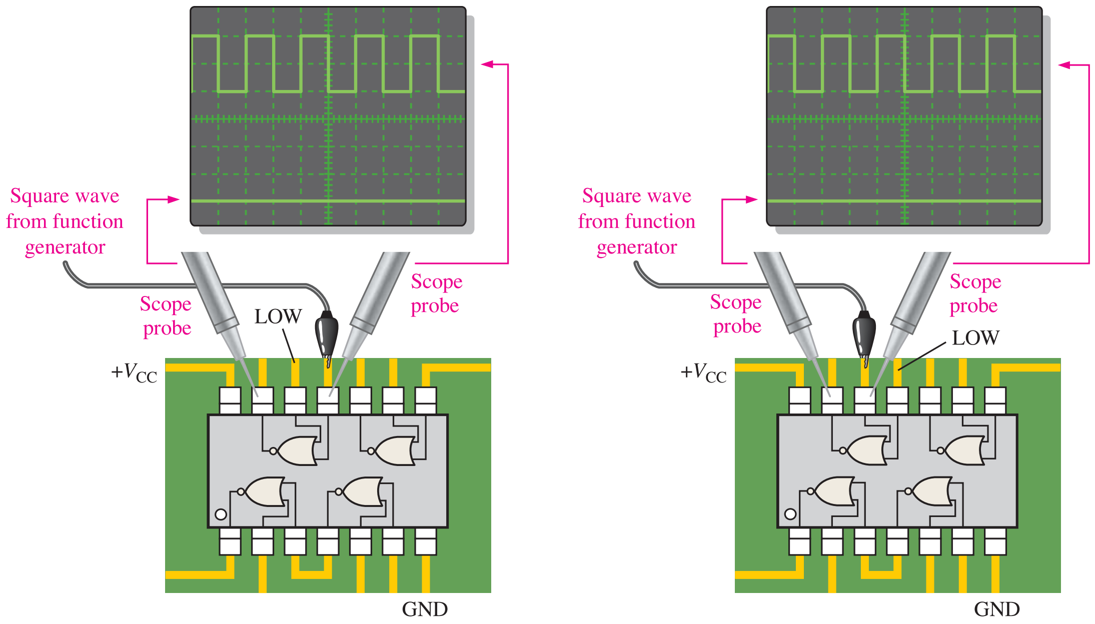
An internally open gate output prevents a signal on any of the inputs from getting to the output. Therefore, no matter what the input conditions are, the output is unaffected. The level at the output pin of the IC will depend upon what it is externally connected to. It could be either HIGH, LOW, or floating (not fixed to any reference). In any case, there will be no signal on the output pin.

Troubleshooting an Open Output

Figure 3–71 illustrates troubleshooting an open NOR gate output. In part (a), one of the inputs of the suspected gate (pin 11 in this case) is pulsed, and the output (pin 13) has no pulse waveform. In part (b), the other input (pin 12) is pulsed and again there is no pulse waveform on the output. Under the condition that the input that is not being pulsed is at a LOW level, this test shows that the output is internally open.

Shorted Input or Output

Although not as common as an open, an internal short to the dc supply voltage, ground, another input, or an output can occur. When an input or output is shorted to the supply voltage, it will be stuck in the HIGH state. If an input or output is shorted to ground, it will be



(a) Pulse input on pin 11. No pulse output.

(b) Pulse input on pin 12. No pulse output.

FIGURE 3–71 Troubleshooting a NOR gate for an open output.

stuck in the LOW state (0 V). If two inputs or an input and an output are shorted together, they will always be at the same level.

External Opens and Shorts

Many failures involving digital ICs are due to faults that are external to the IC package. These include bad solder connections, solder splashes, wire clippings, improperly etched printed circuit boards (PCBs), and cracks or breaks in wires or printed circuit interconnections. These open or shorted conditions have the same effect on the logic gate as the internal faults, and troubleshooting is done in basically the same ways. A visual inspection of any circuit that is suspected of being faulty is the first thing a technician should do.

EXAMPLE 3–25

You are checking a 74LS10 triple 3-input NAND gate IC that is one of many ICs located on a PCB. You have checked pins 1 and 2 and they are both HIGH. Now you apply a pulse waveform to pin 13, and place your scope probe first on pin 12 and then on the connecting PCB trace, as indicated in Figure 3–72. Based on your observation of the scope screen, what is the most likely problem?

Solution

The waveform with the probe in position 1 shows that there is pulse activity on the gate output at pin 12, but there are no pulses on the PCB trace as indicated by the probe in position 2. The gate is working properly, but the signal is not getting from pin 12 of the IC to the PCB trace.

Most likely there is a bad solder connection between pin 12 of the IC and the PCB, which is creating an open. You should resolder that point and check it again.

Related Problem

If there are no pulses at either probe position 1 or 2 in Figure 3–72, what fault(s) does this indicate?

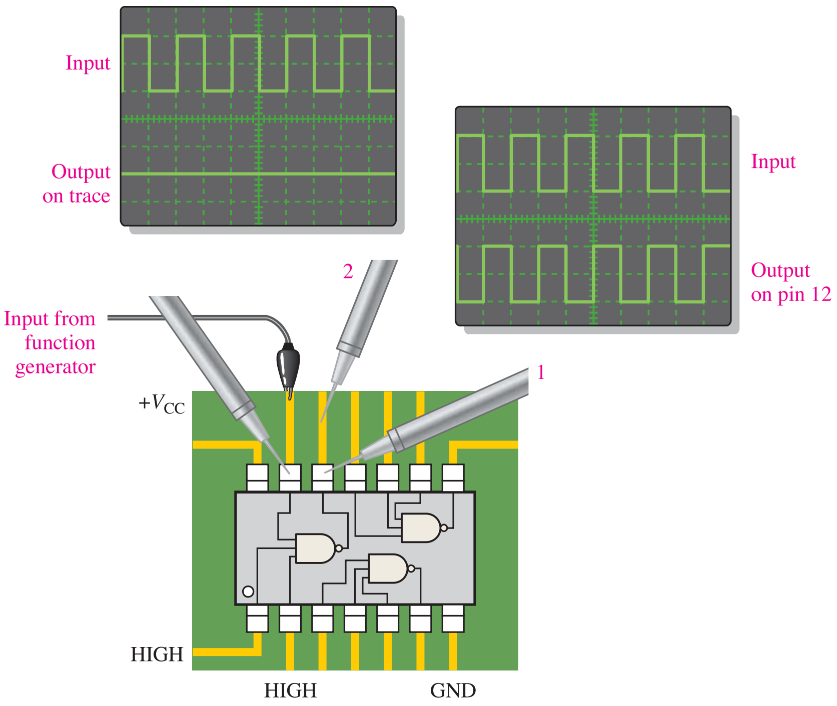


FIGURE 3–72

In most cases, you will be troubleshooting ICs that are mounted on PCBs or prototype assemblies and interconnected with other ICs. As you progress through this book, you will learn how different types of digital ICs are used together to perform system functions. At this point, however, we are concentrating on individual IC gates. This limitation does not prevent us from looking at the system concept at a very basic and simplified level.

To continue the emphasis on systems, Examples 3–26 and 3–27 deal with troubleshooting the frequency counter that was introduced in Section 3–2.

EXAMPLE 3–26

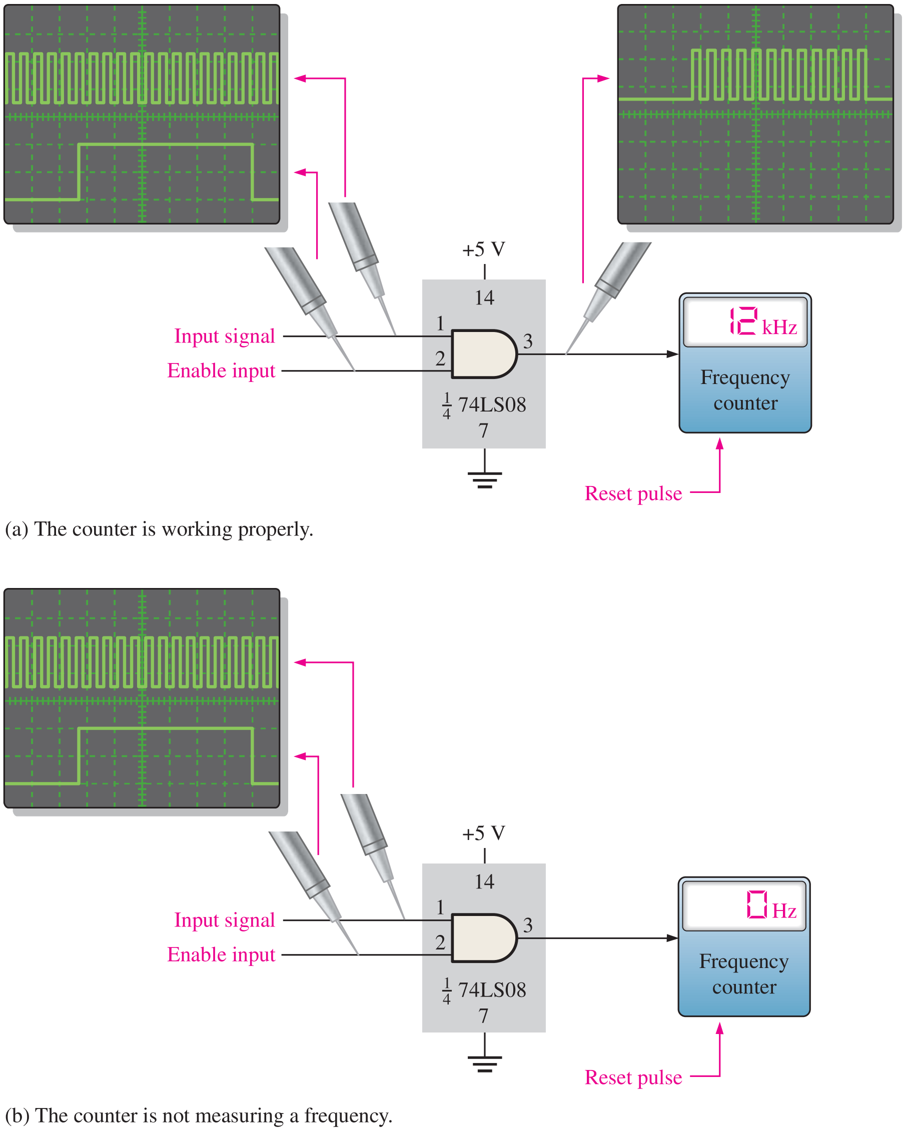
After trying to operate the frequency counter shown in Figure 3–73, you find that it constantly reads out all 0s on its display, regardless of the input frequency. Determine the cause of this malfunction. The enable pulse has a width of 1 ms.

Figure 3–73(a) gives an example of how the frequency counter should be working with a 12 kHz pulse waveform on the input to the AND gate. Part (b) shows that the display is improperly indicating 0 Hz.

Solution

Three possible causes are

1. A constant active or asserted level on the counter reset input, which keeps the counter at zero.
2. No pulse signal on the input to the counter because of an internal open or short in the counter. This problem would keep the counter from advancing after being reset to zero.



(a) The counter is working properly.

(b) The counter is not measuring a frequency.

FIGURE 3–73

3. No pulse signal on the input to the counter because of an open AND gate output or the absence of input signals, again keeping the counter from advancing from zero.

The first step is to make sure that V_{CC} and ground are connected to all the right places; assume that they are found to be okay. Next, check for pulses on both inputs to the AND gate. The scope indicates that there are proper pulses on both of these inputs. A check of the counter reset shows a LOW level which is known to be the unasserted level and, therefore, this is not the problem. The next check on pin 3 of the 74LS08 shows that there are no pulses on the output of the AND gate, indicating that the gate output is open. Replace the 74LS08 IC and check the operation again.

Related Problem

If pin 2 of the 74LS08 AND gate is open, what indication should you see on the frequency display?

EXAMPLE 3–27

The frequency counter shown in Figure 3–74 appears to measure the frequency of input signals incorrectly. It is found that when a signal with a precisely known frequency is applied to pin 1 of the AND gate, the oscilloscope display indicates a higher frequency. Determine what is wrong. The readings on the screen indicate time per division.

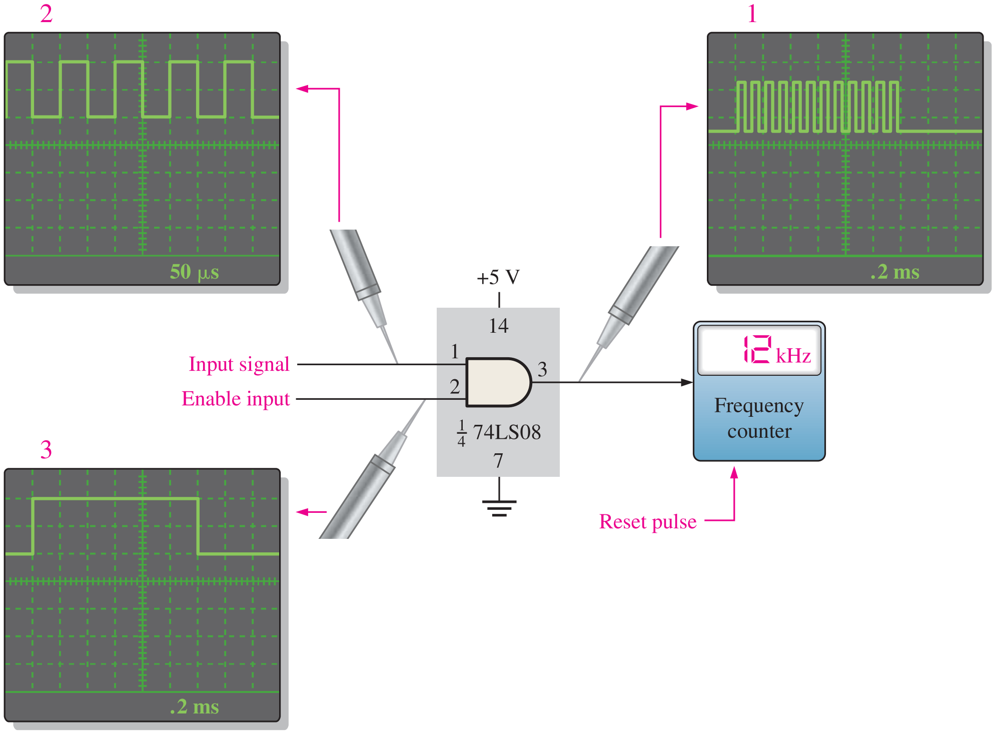


FIGURE 3–74

Solution

Recall from Section 3–2 that the input pulses were allowed to pass through the AND gate for exactly 1 ms. The number of pulses counted in 1 ms is equal to the frequency in hertz. Therefore, the 1 ms interval, which is produced by the enable pulse on pin 2 of the AND gate, is very critical to an accurate frequency measurement. The enable pulses are produced internally by a precision oscillator circuit. The pulse must be exactly 1 ms in width and in this case it occurs every 3 ms to update the count. Just prior to each enable pulse, the counter is reset to zero so that it starts a new count each time.

Since the counter appears to be counting more pulses than it should to produce a frequency readout that is too high, the enable pulse is the primary suspect. Exact time-interval measurements must be made on the oscilloscope.

An input pulse waveform of exactly 10 kHz is applied to pin 1 of the AND gate and the frequency counter incorrectly shows 12 kHz. The first scope measurement, on the output of the AND gate, shows that there are 12 pulses for each enable pulse. In the second scope measurement, the input frequency is verified to be precisely 10 kHz (period $= 100\ \mu s$). In the third scope measurement, the width of the enable pulse is found to be 1.2 ms rather than 1 ms.

The conclusion is that the enable pulse is out of calibration for some reason.

Related Problem

What would you suspect if the readout were indicating a frequency less than it should be?

HandsOnTip

Proper grounding is very important when setting up to take measurements or work on a circuit. Properly grounding the oscilloscope protects you from shock and grounding yourself protects your circuits from damage. Grounding the oscilloscope means to connect it to earth ground by plugging the three-prong power cord into a grounded outlet. Grounding yourself means using a wrist-type grounding strap, particularly when you are working with CMOS logic. The wrist strap must have a high-value resistor between the strap and ground for protection against accidental contact with a voltage source.

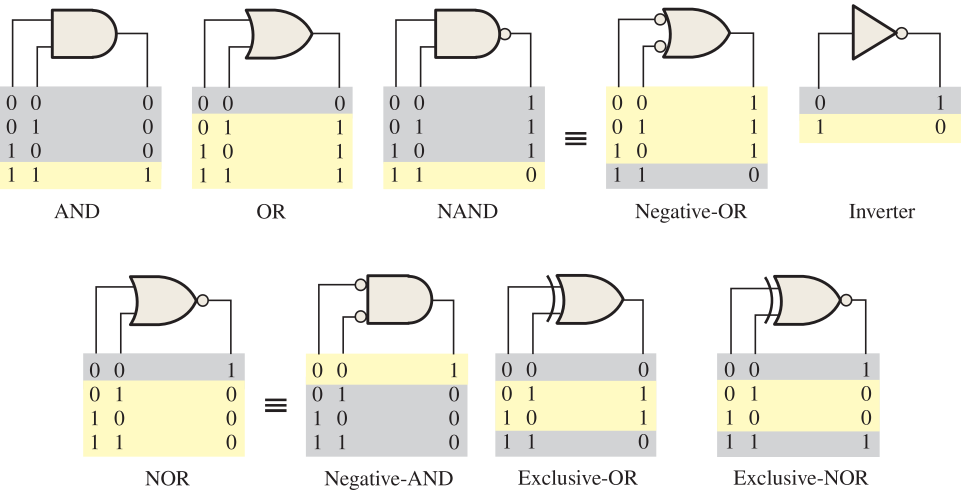
Also, for accurate measurements, make sure that the ground in the circuit you are testing is the same as the scope ground. This can be done by connecting the ground lead on the scope probe to a known ground point in the circuit, such as the metal chassis or a ground point on the PCB. You can also connect the circuit ground to the GND jack on the front panel of the scope.

SECTION 3–9 CHECKUP

1. What are the most common types of failures in ICs?
2. If two different input waveforms are applied to a 2-input bipolar NAND gate and the output waveform is just like one of the inputs, but inverted, what is the most likely problem?
3. Name two characteristics of pulse waveforms that can be measured on the oscilloscope.

SUMMARY

- The inverter output is the complement of the input.
- The AND gate output is HIGH only when all the inputs are HIGH.
- The OR gate output is HIGH when any of the inputs is HIGH.
- The NAND gate output is LOW only when all the inputs are HIGH.
- The NAND can be viewed as a negative-OR whose output is HIGH when any input is LOW.
- The NOR gate output is LOW when any of the inputs is HIGH.
- The NOR can be viewed as a negative-AND whose output is HIGH only when all the inputs are LOW.
- The exclusive-OR gate output is HIGH when the inputs are not the same.
- The exclusive-NOR gate output is LOW when the inputs are not the same.
- Distinctive shape symbols and truth tables for various logic gates (limited to 2 inputs) are shown in Figure 3–75.



AND

0	0	0
0	1	0
1	0	0
1	1	1

OR

0	0	0
0	1	1
1	0	1
1	1	1

NAND

0	0	1
0	1	1
1	0	1
1	1	0

≡

Negative-OR

0	0	1
0	1	1
1	0	1
1	1	0

Inverter

0	1
1	0

NOR

0	0	1
0	1	0
1	0	0
1	1	0

≡

Negative-AND

0	0	1
0	1	0
1	0	0
1	1	0

Exclusive-OR

0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR

0	0	1
0	1	0
1	0	0
1	1	1

Note: Active states are shown in yellow.

FIGURE 3–75

- Most programmable logic devices (PLDs) are based on some form of AND array.
- Programmable link technologies are fuse, antifuse, EPROM, EEPROM, flash, and SRAM.
- A PLD can be programmed in a hardware fixture called a programmer or mounted on a development printed circuit board.
- PLDs have an associated software development package for programming.
- Two methods of design entry using programming software are text entry (HDL) and graphic (schematic) entry.
- ISP PLDs can be programmed after they are installed in a system, and they can be reprogrammed at any time.
- JTAG stands for Joint Test Action Group and is an interface standard (IEEE Std. 1149.1) used for programming and testing PLDs.
- An embedded processor is used to facilitate in-system programming of PLDs.
- In PLDs, the circuit is programmed in and can be changed by reprogramming.
- The average power dissipation of a logic gate is

$$P_D = V_{CC}\left(\frac{I_{CCH} + I_{CCL}}{2}\right)$$

- The speed-power product of a logic gate is

$$SPP = t_p P_D$$

- As a rule, CMOS has a lower power consumption than bipolar.
- In fixed-function logic, the circuit cannot be altered.

KEY TERMS

Key terms and other bold terms in the chapter are defined in the end-of-book glossary.

AND array An array of AND gates consisting of a matrix of programmable interconnections.

AND gate A logic gate that produces a HIGH output only when all of the inputs are HIGH.

Antifuse A type of PLD nonvolatile programmable link that can be left open or can be shorted once as directed by the program.

Bipolar A class of integrated logic circuits implemented with bipolar transistors; also known as TTL.

Boolean algebra The mathematics of logic circuits.

CMOS Complementary metal-oxide semiconductor; a class of integrated logic circuits that is implemented with a type of field-effect transistor.

Complement The inverse or opposite of a number. LOW is the complement of HIGH, and 0 is the complement of 1.

EEPROM A type of nonvolatile PLD reprogrammable link based on electrically erasable programmable read-only memory cells and can be turned on or off repeatedly by programming.

EPROM A type of PLD nonvolatile programmable link based on electrically programmable read-only memory cells and can be turned either on or off once with programming.

Exclusive-NOR (XNOR) gate A logic gate that produces a LOW only when the two inputs are at opposite levels.

Exclusive-OR (XOR) gate A logic gate that produces a HIGH output only when its two inputs are at opposite levels.

Fan-out The number of equivalent gate inputs of the same family series that a logic gate can drive.

Flash A type of PLD nonvolatile reprogrammable link technology based on a single transistor cell.

Fuse A type of PLD nonvolatile programmable link that can be left shorted or can be opened once as directed by the program.

Inverter A logic circuit that inverts or complements its input.

JTAG Joint Test Action Group; an interface standard designated IEEE Std. 1149.1.

NAND gate A logic gate that produces a LOW output only when all the inputs are HIGH.

NOR gate A logic gate in which the output is LOW when one or more of the inputs are HIGH.

OR gate A logic gate that produces a HIGH output when one or more inputs are HIGH.

Propagation delay time The time interval between the occurrence of an input transition and the occurrence of the corresponding output transition in a logic circuit.

SRAM A type of PLD volatile reprogrammable link based on static random-access memory cells and can be turned on or off repeatedly with programming.

Target device A PLD mounted on a programming fixture or development board into which a software logic design is to be downloaded.

Truth table A table showing the inputs and corresponding output(s) of a logic circuit.

Unit load A measure of fan-out. One gate input represents one unit load to the output of a gate within the same IC family.

VHDL A standard hardware description language that describes a function with an entity/architecture structure.

TRUE/FALSE QUIZ

Answers are at the end of the chapter.

1. An inverter performs a NOT operation.
2. A NOT gate cannot have more than one input.
3. If any input to an OR gate is zero, the output is zero.
4. If all inputs to an AND gate are 1, the output is 0.
5. A NAND gate can be considered as an AND gate followed by a NOT gate.
6. A NOR gate can be considered as an OR gate followed by an inverter.
7. The output of an exclusive-OR is 0 if the inputs are opposite.
8. Two types of fixed-function logic integrated circuits are bipolar and NMOS.
9. Once programmed, PLD logic can be changed.
10. Fan-out is the number of similar gates that a given gate can drive.

SELF-TEST

Answers are at the end of the chapter.

1. When the input to an inverter is LOW (0), the output is
   (a) HIGH or 0 **(b)** LOW or 0 **(c)** HIGH or 1 **(d)** LOW or 1
2. An inverter performs an operation known as
   (a) complementation **(b)** assertion **(c)** inversion **(d)** both answers (a) and (c)
3. The output of an AND gate with inputs A, B and C is 0 (LOW) when
   (a) $A = 0, B = 0, C = 0$ **(b)** $A = 0, B = 1, C = 1$ **(c)** both answers (a) and (b)
4. The output of an OR gate with inputs A, B and C is 0 (LOW) when
   (a) $A = 0, B = 0, C = 0$ **(b)** $A = 0, B = 1, C = 1$ **(c)** both answers (a) and (b)
5. A pulse is applied to each input of a 2-input NAND gate. One pulse goes HIGH at $t = 0$ and goes back LOW at $t = 1$ ms. The other pulse goes HIGH at $t = 0.8$ ms and goes back LOW at $t = 3$ ms. The output pulse can be described as follows:
   (a) It goes LOW at $t = 0$ and back HIGH at $t = 3$ ms.
   (b) It goes LOW at $t = 0.8$ ms and back HIGH at $t = 3$ ms.
   (c) It goes LOW at $t = 0.8$ ms and back HIGH at $t = 1$ ms.
   (d) It goes LOW at $t = 0.8$ ms and back LOW at $t = 1$ ms.
6. A pulse is applied to each input of a 2-input NOR gate. One pulse goes HIGH at $t = 0$ and goes back LOW at $t = 1$ ms. The other pulse goes HIGH at $t = 0.8$ ms and goes back LOW at $t = 3$ ms. The output pulse can be described as follows:
   (a) It goes LOW at $t = 0$ and back HIGH at $t = 3$ ms.
   (b) It goes LOW at $t = 0.8$ ms and back HIGH at $t = 3$ ms.
   (c) It goes LOW at $t = 0.8$ ms and back HIGH at $t = 1$ ms.
   (d) It goes HIGH at $t = 0.8$ ms and back LOW at $t = 1$ ms.

7. A pulse is applied to each input of an exclusive-OR gate. One pulse goes HIGH at $t = 0$ and goes back LOW at $t = 1$ ms. The other pulse goes HIGH at $t = 0.8$ ms and goes back LOW at $t = 3$ ms. The output pulse can be described as follows:
(a) It goes HIGH at $t = 0$ and back LOW at $t = 3$ ms.
(b) It goes HIGH at $t = 0$ and back LOW at $t = 0.8$ ms.
(c) It goes HIGH at $t = 1$ ms and back LOW at $t = 3$ ms.
(d) both answers (b) and (c)

8. A positive-going pulse is applied to an inverter. The time interval from the leading edge of the input to the leading edge of the output is 7 ns. This parameter is
(a) speed-power product **(b)** propagation delay, t_{PHL}
(c) propagation delay, t_{PLH} **(d)** pulse width

9. Most PLDs utilize an array of
(a) NOT gates
(b) NOR gates
(c) OR gates
(d) AND gates

10. The rows and columns of the interconnection matrix in an SPLD are connected using
(a) fuses **(b)** switches
(c) gates **(d)** transistors

11. An antifuse is formed using
(a) two insulators separated by a conductor **(b)** two conductors separated by an insulator
(c) an insulator packed beside a conductor **(d)** two conductors connected in a series

12. An EPROM can be programmed using
(a) transistors **(b)** diodes
(c) a multiprogrammer **(d)** a device programmer

13. Two ways to enter a logic design using PLD development software are
(a) text and numeric **(b)** text and graphic
(c) graphic and coded **(d)** compile and sort

14. JTAG stands for
(a) Joint Test Action Group **(b)** Java Top Array Group
(c) Joint Test Array Group **(d)** Joint Time Analysis Group

15. In-system programming of a PLD typically utilizes
(a) an embedded clock generator **(b)** an embedded processor
(c) an embedded PROM **(d)** both (a) and (b)
(e) both (b) and (c)

16. To measure the period of a pulse waveform, you must use
(a) a DMM **(b)** a logic probe
(c) an oscilloscope **(d)** a logic pulser

17. Once you measure the period of a pulse waveform, the frequency is found by
(a) using another setting **(b)** measuring the duty cycle
(c) finding the reciprocal of the period **(d)** using another type of instrument

PROBLEMS

Answers to odd-numbered problems are at the end of the book.

Section 3–1 The Inverter

1. The input waveform shown in Figure 3–76 is applied to a system of two inverters connected in a series. Draw the output waveform across each inverter in proper relation to the input.

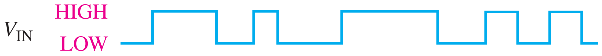


FIGURE 3–76

2. A combination of inverters is shown in Figure 3–77. If a LOW is applied to point *A*, determine the net output at points *E* and *F*.

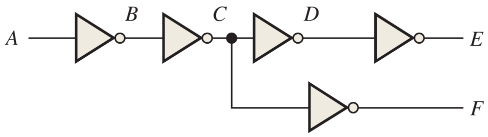


FIGURE 3–77

3. If the waveform in Figure 3–76 is applied to point *A* in Figure 3–77, determine the waveforms at points *B* through *F*.

Section 3–2 The AND Gate

4. Draw the rectangular outline symbol for a 3-input AND gate.

5. Determine the output, *X*, for a 2-input AND gate with the input waveforms shown in Figure 3–78. Show the proper relationship of output to inputs with a timing diagram.

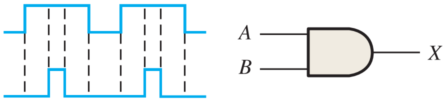


FIGURE 3–78

6. The waveforms in Figure 3–79 are applied to points *A* and *B* of a 2-input AND gate followed by an inverter. Draw the output waveform.

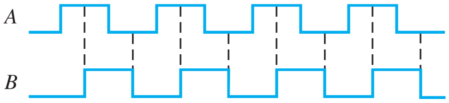


FIGURE 3–79

7. The input waveforms applied to a 3-input AND gate are as indicated in Figure 3–80. Show the output waveform in proper relation to the inputs with a timing diagram.

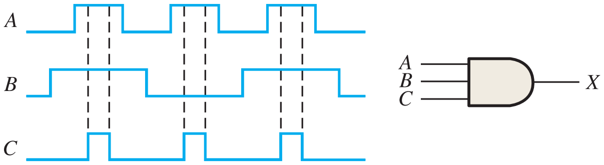


FIGURE 3–80

8. The input waveforms applied to a 4-input AND gate are as indicated in Figure 3–81. The output of the AND gate is fed to an inverter. Draw the net output waveform of this system.

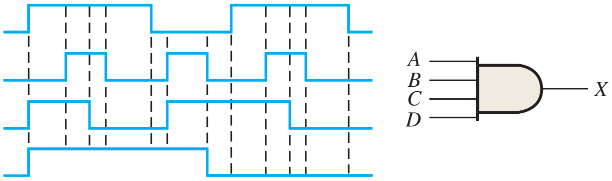


FIGURE 3–81

Section 3–3 The OR Gate

9. Draw the rectangular outline symbol for a 3-input OR gate.

10. Write the expression for a 4-input OR gate with inputs *A*, *B*, *C*, *D*, and output *X*.

11. Determine the output for a 2-input OR gate when the input waveforms are as in Figure 3–79 and draw a timing diagram.

12. Repeat Problem 7 for a 3-input OR gate.

13. Repeat Problem 8 for a 4-input OR gate.

14. For the waveforms given in Figure 3–82, *A* and *B* are ANDed with output *F*, *D* and *E* are ANDed with output *G*, and *C*, *F*, and *G* are ORed. Draw the net output waveform.

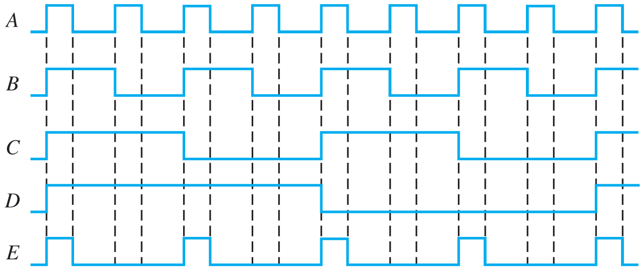


FIGURE 3–82

15. Draw the rectangular outline symbol for a 4-input OR gate.

16. Show the truth table for a system of a 3-input OR gate followed by an inverter.

Section 3–4 The NAND Gate

17. For the set of input waveforms in Figure 3–83, determine the output for the gate shown and draw the timing diagram.

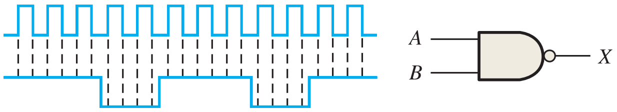


FIGURE 3–83

18. Determine the gate output for the input waveforms in Figure 3–84 and draw the timing diagram.

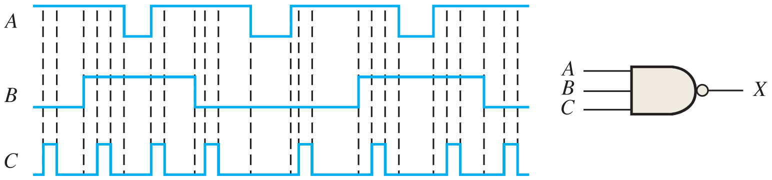


FIGURE 3–84

19. Determine the output waveform in Figure 3–85.

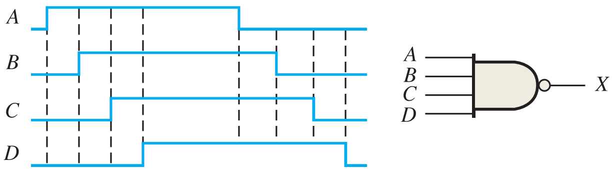


FIGURE 3–85

20. As you have learned, the two logic symbols shown in Figure 3–86 represent equivalent operations. The difference between the two is strictly from a functional viewpoint. For the NAND symbol, look for two HIGHs on the inputs to give a LOW output. For the negative-OR, look for at least one LOW on the inputs to give a HIGH on the output. Using these two functional viewpoints, show that each gate will produce the same output for the given inputs.

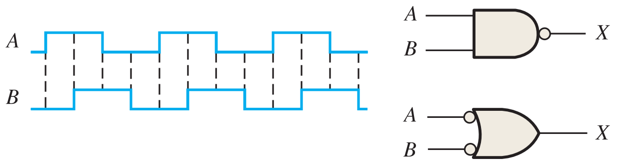


FIGURE 3–86

Section 3–5 The NOR Gate

21. Repeat Problem 17 for a 2-input NOR gate.

22. Determine the output waveform in Figure 3–87 and draw the timing diagram.

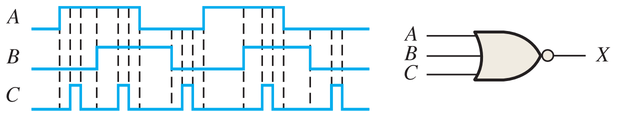


FIGURE 3–87

23. Repeat Problem 19 for a 4-input NOR gate.

24. The NAND and the negative-OR symbols represent equivalent operations, but they are functionally different. For the NOR symbol, look for at least one HIGH on the inputs to give a LOW on the output. For the negative-AND, look for two LOWs on the inputs to give a HIGH output. Using these two functional points of view, show that both gates in Figure 3–88 will produce the same output for the given inputs.

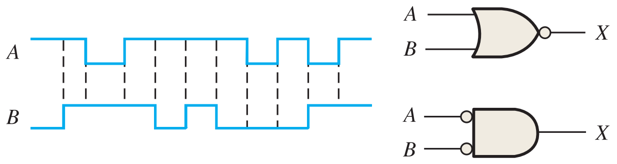


FIGURE 3–88

Section 3–6 The Exclusive-OR and Exclusive-NOR Gates

25. How does an exclusive-OR gate differ from an OR gate in its logical operation?

26. Repeat Problem 17 for an exclusive-OR gate.

27. Repeat Problem 17 for an exclusive-NOR gate.

28. Determine the output of an exclusive-NOR gate for the inputs shown in Figure 3–79 and draw a timing diagram.

Section 3–7 Programmable Logic

29. In the simple programmed AND array with programmable links in Figure 3–89, determine the Boolean output expressions.

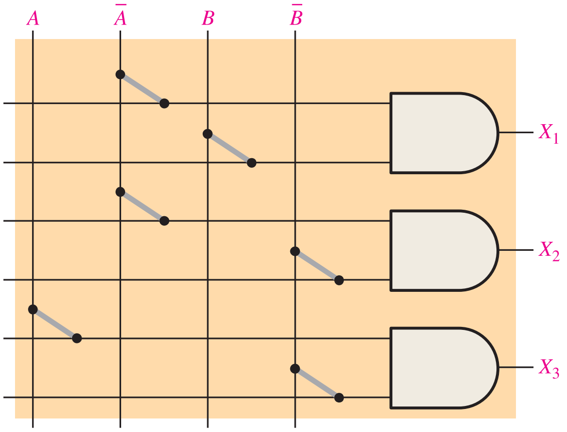


FIGURE 3–89

30. Determine by row and column number which fusible links must be blown in the programmable AND array of Figure 3–90 to implement each of the following product terms: $X_1 = \overline{A}BC$, $X_2 = AB\overline{C}$, $X_3 = \overline{A}B\overline{C}$.

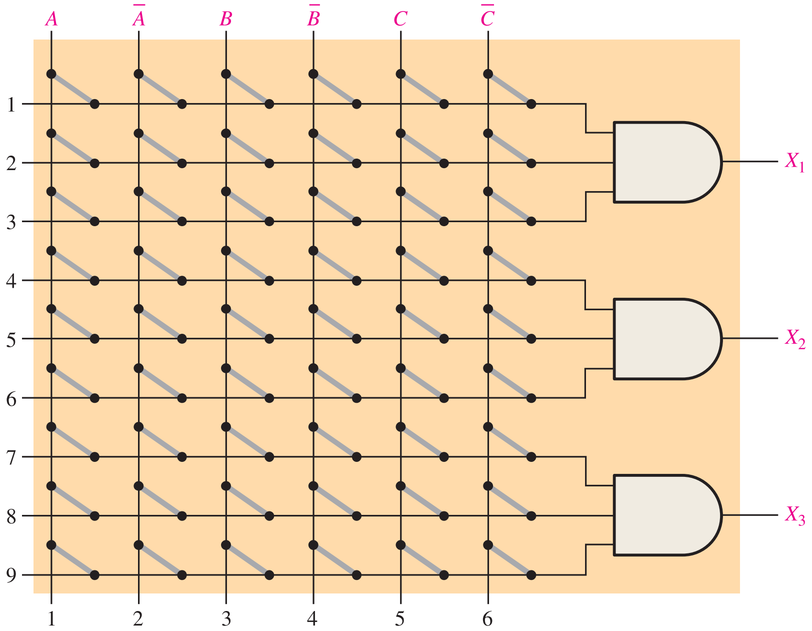


FIGURE 3–90

31. Describe a 4-input AND gate using VHDL.

32. Describe a 5-input NOR gate using VHDL.

Section 3–8 Fixed-Function Logic Gates

33. In the comparison of certain logic devices, it is noted that the power dissipation for one particular type increases as the frequency increases. Is the device bipolar or CMOS?

34. Using the data sheets in Figures 3–65 and 3–66, determine the following:

(a) 74LS00 power dissipation at maximum supply voltage and a 50% duty cycle
(b) Minimum HIGH level output voltage for a 74LS00
(c) Maximum propagation delay for a 74LS00
(d) Maximum LOW level output voltage for a 74HC00A
(e) Maximum propagation delay for a 74HC00A

35. Determine t_{PLH} and t_{PHL} from the oscilloscope display in Figure 3–91. The readings indicate volts/div and sec/div for each channel.

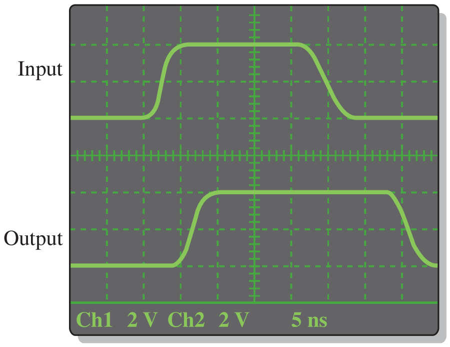


FIGURE 3–91

36. Gate A has $t_{PLH} = t_{PHL} = 6$ ns. Gate B has $t_{PLH} = t_{PHL} = 10$ ns. Which gate can be operated at a higher frequency?

37. If a logic gate operates on a dc supply voltage of $+5$ V and draws an average current of 4 mA, what is its power dissipation?

38. The variable I_{CCH} represents the dc supply current from V_{CC} when all outputs of an IC are HIGH. The variable I_{CCL} represents the dc supply current when all outputs are LOW. For a 74LS00 IC, determine the typical power dissipation when all four gate outputs are HIGH. (See data sheet in Figure 3–66.)

Section 3–9 Troubleshooting

39. Examine the conditions indicated in Figure 3–92, and identify the faulty gates.

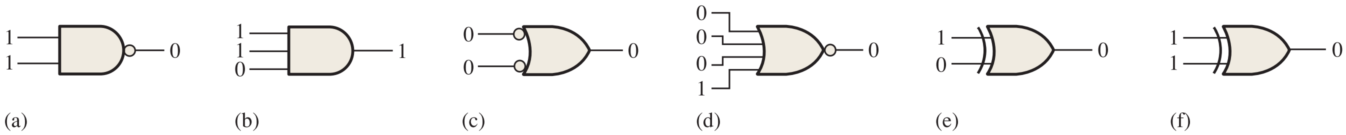


FIGURE 3–92

40. Determine the faulty gates in Figure 3–93 by analyzing the timing diagrams.

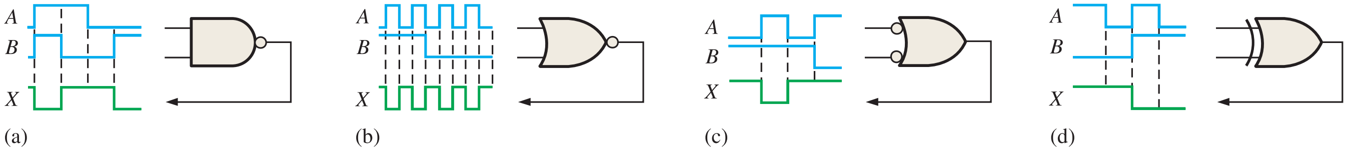
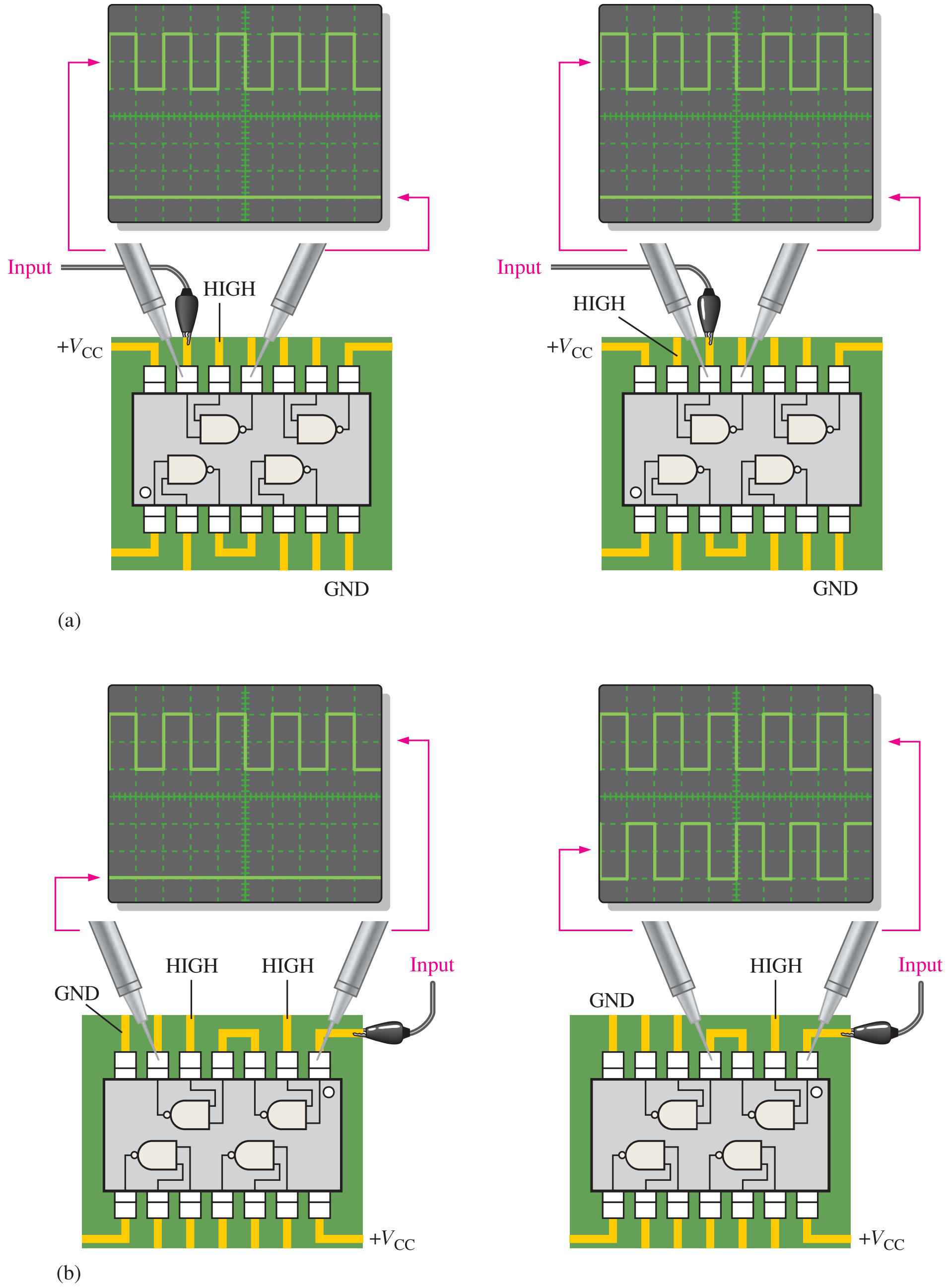


FIGURE 3–93

41. Using an oscilloscope, you make the observations indicated in Figure 3–94. For each observation determine the most likely gate failure.



(a)

(b)

FIGURE 3–94

42. The seat belt alarm circuit in Figure 3–17 has malfunctioned. You find that when the ignition switch is turned on and the seat belt is unbuckled, the alarm comes on and will not go off. What is the most likely problem? How do you troubleshoot it?

43. Every time the ignition switch is turned on in the circuit of Figure 3–17, the alarm comes on for thirty seconds, even when the seat belt is buckled. What is the most probable cause of this malfunction?

44. What failure(s) would you suspect if the output of a 3-input NAND gate stays HIGH no matter what the inputs are?

Special Design Problems

45. Modify the frequency counter in Figure 3–16 to operate with an enable pulse that is active-LOW rather than HIGH during the 1 ms interval.

46. Assume that the enable signal in Figure 3–16 has the waveform shown in Figure 3–95. Assume that waveform *B* is also available. Devise a circuit that will produce an active-HIGH reset pulse to the counter only during the time that the enable signal is LOW.

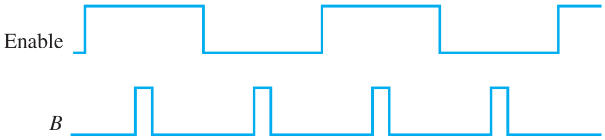


FIGURE 3–95

47. Design a circuit to fit in the beige block of Figure 3–96 that will cause the headlights of an automobile to be turned off automatically 15 s after the ignition switch is turned off, if the light switch is left on. Assume that a LOW is required to turn the lights off.

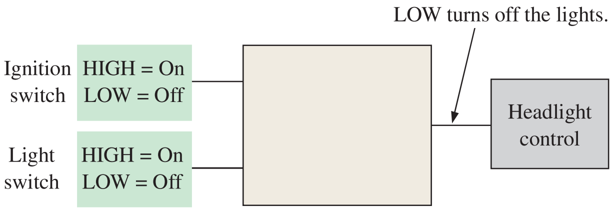


FIGURE 3–96

48. Modify the logic circuit for the intrusion alarm in Figure 3–25 so that two additional rooms, each with two windows and one door, can be protected.

49. Further modify the logic circuit from Problem 48 for a change in the input sensors where Open = LOW and Closed = HIGH.

50. Sensors are used to monitor the pressure and the temperature of a chemical solution stored in a vat. The circuitry for each sensor produces a HIGH voltage when a specified maximum value is exceeded. An alarm requiring a LOW voltage input must be activated when either the pressure or the temperature is excessive. Design a circuit for this application.

51. In a certain automated manufacturing process, electrical components are automatically inserted in a PCB. Before the insertion tool is activated, the PCB must be properly positioned, and the component to be inserted must be in the chamber. Each of these prerequisite conditions is indicated by a HIGH voltage. The insertion tool requires a LOW voltage to activate it. Design a circuit to implement this process.

MultiSim

Multisim Troubleshooting Practice

52. Open file P03-52. For the specified fault, predict the effect on the circuit. Then introduce the fault and verify whether your prediction is correct.

53. Open file P03-53. For the specified fault, predict the effect on the circuit. Then introduce the fault and verify whether your prediction is correct.

54. Open file P03-54. For the observed behavior indicated, predict the fault in the circuit. Then introduce the suspected fault and verify whether your prediction is correct.

55. Open file P03-55. For the observed behavior indicated, predict the fault in the circuit. Then introduce the suspected fault and verify whether your prediction is correct.

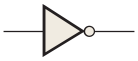
ANSWERS

SECTION CHECKUPS

Section 3–1 The Inverter

1. When the inverter input is 1, the output is 0.

2. **(a)**



(b) A negative-going pulse is on the output (HIGH to LOW and back HIGH).

Section 3–2 The AND Gate

1. An AND gate output is HIGH only when all inputs are HIGH.
2. An AND gate output is LOW when one or more inputs are LOW.
3. Five-input AND: $X = 1$ when $ABCDE = 11111$, and $X = 0$ for all other combinations of $ABCDE$.

Section 3–3 The OR Gate

1. An OR gate output is HIGH when one or more inputs are HIGH.
2. An OR gate output is LOW only when all inputs are LOW.
3. Three-input OR: $X = 0$ when $ABC = 000$, and $X = 1$ for all other combinations of ABC.

Section 3–4 The NAND Gate

1. A NAND gate output is LOW only when all inputs are HIGH.
2. A NAND gate output is HIGH when one or more inputs are LOW.
3. NAND: active-LOW output for all HIGH inputs; negative-OR: active-HIGH output for one or more LOW inputs. They have the same truth tables.
4. $X = \overline{ABC}$

Section 3–5 The NOR Gate

1. A NOR gate output is HIGH only when all inputs are LOW.
2. A NOR gate output is LOW when one or more inputs are HIGH.
3. NOR: active-LOW output for one or more HIGH inputs; negative-AND: active-HIGH output for all LOW inputs. They have the same truth tables.
4. $X = \overline{A + B + C}$

Section 3–6 The Exclusive-OR and Exclusive-NOR Gates

1. An XOR gate output is HIGH when the inputs are at opposite levels.
2. An XNOR gate output is HIGH when the inputs are at the same levels.
3. Apply the bits to the XOR gate inputs; when the output is HIGH, the bits are different.

Section 3–7 Programmable Logic

1. Fuse, antifuse, EPROM, EEPROM, flash, and SRAM
2. Volatile means that all the data are lost when power is off and the PLD must be reprogrammed; SRAM-based
3. Text entry and graphic entry
4. JTAG is Joint Test Action Group; the IEEE Std. 1149.1 for programming and test interfacing.
5.
```
entity NORgate is
   port (A, B, C: in bit; X: out bit);
end entity NORgate;
architecture NORfunction of NORgate is
begin
   X <= A nor B nor C;
end architecture NORfunction;
```
6.
```
entity XORgate is
   port (A, B: in bit; X: out bit);
end entity XORgate;
architecture XORfunction of XORgate is
begin
   X <= A xor B;
end architecture XORfunction;
```

Section 3–8 Fixed-Function Logic Gates

1. Fixed-function logic cannot be changed. PLDs can be programmed for any logic function.
2. CMOS and bipolar (TTL)

3. **(a)** LS—Low-power Schottky
   (b) HC—High-speed CMOS
   (c) HCT—HC CMOS TTL compatible
4. Lowest power—CMOS
5. Six inverters in a package; four 2-input NAND gates in a package
6. $t_{PLH} = 10$ ns; $t_{PHL} = 8$ ns
7. 18 pJ
8. I_{CCL}—dc supply current for LOW output state; I_{CCH}—dc supply current for HIGH output state
9. V_{IL}—LOW input voltage; V_{IH}—HIGH input voltage
10. V_{OL}—LOW output voltage; V_{OH}—HIGH output voltage

Section 3–9 Troubleshooting

1. Opens and shorts are the most common failures.
2. An open input which effectively makes input HIGH
3. Amplitude and period

RELATED PROBLEMS FOR EXAMPLES

3–1 The timing diagram is not affected.

3–2 See Table 3–15.

TABLE 3–15

Inputs *ABCD*	Output *X*	Inputs *ABCD*	Output *X*
0000	0	1000	0
0001	0	1001	0
0010	0	1010	0
0011	0	1011	0
0100	0	1100	0
0101	0	1101	0
0110	0	1110	0
0111	0	1111	1

3–3 See Figure 3–97.

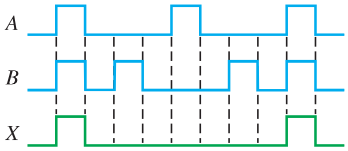


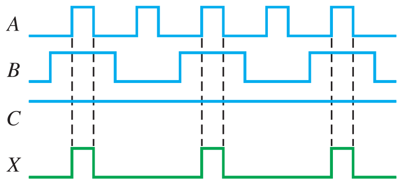
FIGURE 3–97

3–4 The output waveform is the same as input *A*.

3–5 See Figure 3–98.

3–6 Results are the same as example.

3–7 See Figure 3–99.



C = HIGH

FIGURE 3–98

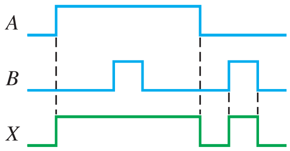


FIGURE 3–99

3–8 See Figure 3–100.

3–9 See Figure 3–101.

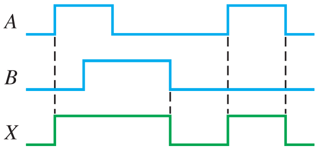


FIGURE 3–100

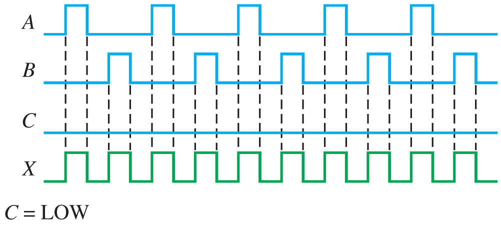


FIGURE 3–101

3–10 See Figure 3–102.

3–11 See Figure 3–103.

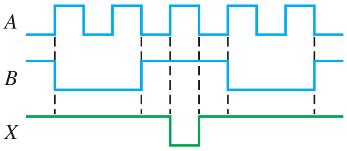


FIGURE 3–102

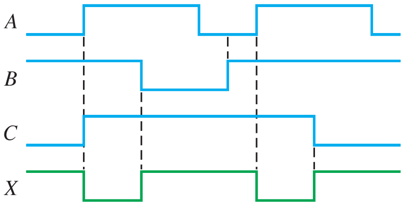


FIGURE 3–103

3–12 Use a 3-input NAND gate.

3–13 Use a 4-input NAND gate operating as a negative-OR gate.

3–14 See Figure 3–104.

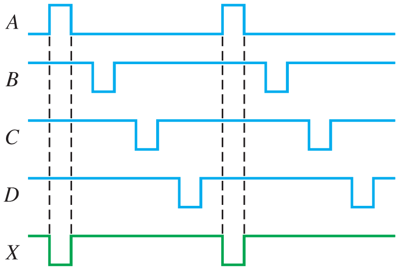


FIGURE 3–104

3–15 See Figure 3–105.

3–16 See Figure 3–106.

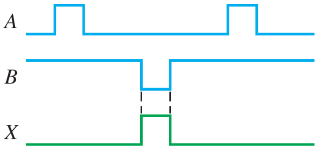


FIGURE 3–105

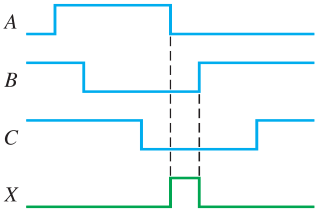


FIGURE 3–106

3–17 Use a 2-input NOR gate.

3–18 A 3-input NAND gate.

3–19 The output is always LOW. The output is a straight line.

3–20 The exclusive-OR gate will not detect simultaneous failures if both circuits produce the same outputs.

3–21 The outputs are unaffected.

3–22 6 columns, 9 rows, and 3 AND gates with three inputs each

3–23 The gate with 4 ns t_{PLH} and t_{PHL} can operate at the highest frequency.

3–24 10 mW

3–25 The gate output or pin 13 input is internally open.

3–26 The display will show an erratic readout because the counter continues until reset.

3–27 The enable pulse is too short or the counter is reset too soon.

TRUE/FALSE QUIZ

1. T **2.** T **3.** F **4.** F **5.** T

6. T **7.** F **8.** F **9.** T **10.** T

SELF-TEST

1. (c) **2.** (d) **3.** (c) **4.** (a) **5.** (c) **6.** (a) **7.** (d) **8.** (b) **9.** (d)

10. (a) **11.** (b) **12.** (d) **13.** (b) **14.** (a) **15.** (d) **16.** (c) **17.** (c)

CHAPTER 4

Boolean Algebra and Logic Simplification

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Apply the basic laws and rules of Boolean algebra
- Apply DeMorgan's theorems to Boolean expressions
- Describe gate combinations with Boolean expressions
- Evaluate Boolean expressions
- Simplify expressions by using the laws and rules of Boolean algebra
- Convert any Boolean expression into a sum-of-products (SOP) form
- Convert any Boolean expression into a product of-sums (POS) form
- Relate a Boolean expression to a truth table
- Use a Karnaugh map to simplify Boolean expressions
- Use a Karnaugh map to simplify truth table functions
- Utilize "don't care" conditions to simplify logic functions
- Use the Quine-McCluskey method to simplify Boolean expressions
- Write a VHDL program for simple logic
- Apply Boolean algebra and the Karnaugh map method in an application

KEY TERMS

Key terms are in order of appearance in the chapter.

- Variable
- Complement
- Sum term
- Product term
- Sum-of-products (SOP)
- Product-of-sums (POS)
- Karnaugh map
- Minimization
- "Don't care"

VISIT THE WEBSITE

Study aids for this chapter are available at http://www.pearsonglobaleditions.com/floyd

INTRODUCTION

In 1854, George Boole published a work titled *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities.* It was in this publication that a "logical algebra," known today as Boolean algebra, was formulated. Boolean algebra is a convenient and systematic way of expressing and analyzing the operation of logic circuits. Claude Shannon was the first to apply Boole's work to the analysis and design of logic circuits. In 1938, Shannon wrote a thesis at MIT titled *A Symbolic Analysis of Relay and Switching Circuits*.

This chapter covers the laws, rules, and theorems of Boolean algebra and their application to digital circuits. You will learn how to define a given circuit with a Boolean expression and then evaluate its operation. You will also learn how to simplify logic circuits using the methods of Boolean algebra, Karnaugh maps, and the Quine-McCluskey method.

Boolean expressions using the hardware description language VHDL are also covered.

4–1 Boolean Operations and Expressions

Boolean algebra is the mathematics of digital logic. A basic knowledge of Boolean algebra is indispensable to the study and analysis of logic circuits. In the last chapter, Boolean operations and expressions in terms of their relationship to NOT, AND, OR, NAND, and NOR gates were introduced.

After completing this section, you should be able to

- Define *variable*
- Define *literal*
- Identify a sum term
- Evaluate a sum term
- Identify a product term
- Evaluate a product term
- Explain Boolean addition
- Explain Boolean multiplication

InfoNote

In a microprocessor, the arithmetic logic unit (ALU) performs arithmetic and Boolean logic operations on digital data as directed by program instructions. Logical operations are equivalent to the basic gate operations that you are familiar with but deal with a minimum of 8 bits at a time. Examples of Boolean logic instructions are AND, OR, NOT, and XOR, which are called *mnemonics*. An assembly language program uses the mnemonics to specify an operation. Another program called an *assembler* translates the mnemonics into a binary code that can be understood by the microprocessor.

Variable, complement, and *literal* are terms used in Boolean algebra. A **variable** is a symbol (usually an italic uppercase letter or word) used to represent an action, a condition, or data. Any single variable can have only a 1 or a 0 value. The **complement** is the inverse of a variable and is indicated by a bar over the variable (overbar). For example, the complement of the variable A is $\overline{A}$. If $A = 1$, then $\overline{A} = 0$. If $A = 0$, then $\overline{A} = 1$. The complement of the variable A is read as "not A" or "A bar." Sometimes a prime symbol rather than an overbar is used to denote the complement of a variable; for example, B' indicates the complement of B. In this book, only the overbar is used. A **literal** is a variable or the complement of a variable.

Boolean Addition

Recall from Chapter 3 that **Boolean addition** is equivalent to the OR operation. The basic rules are illustrated with their relation to the OR gate in Figure 4–1.

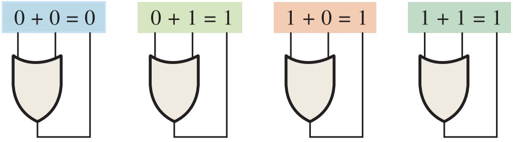


FIGURE 4–1

In Boolean algebra, a **sum term** is a sum of literals. In logic circuits, a sum term is produced by an OR operation with no AND operations involved. Some examples of sum terms are $A + B$, $A + \overline{B}$, $A + B + \overline{C}$, and $\overline{A} + B + C + \overline{D}$.

The OR operation is the Boolean equivalent of addition.

A sum term is equal to 1 when one or more of the literals in the term are 1. A sum term is equal to 0 only if each of the literals is 0.

EXAMPLE 4–1

Determine the values of A, B, C, and D that make the sum term $A + \overline{B} + C + \overline{D}$ equal to 0.

Solution

For the sum term to be 0, each of the literals in the term must be 0. Therefore, $A = \mathbf{0}$, $B = \mathbf{1}$ so that $\overline{B} = 0$, $C = \mathbf{0}$, and $D = \mathbf{1}$ so that $\overline{D} = 0$.

$$A + \overline{B} + C + \overline{D} = 0 + \overline{1} + 0 + \overline{1} = 0 + 0 + 0 + 0 = 0$$

Related Problem*

Determine the values of A and B that make the sum term $\overline{A} + B$ equal to 0.

*Answers are at the end of the chapter.

Boolean Multiplication

Also recall from Chapter 3 that **Boolean multiplication** is equivalent to the AND operation. The basic rules are illustrated with their relation to the AND gate in Figure 4–2.

The AND operation is the Boolean equivalent of multiplication.

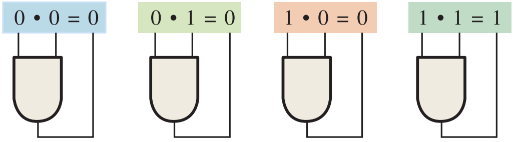


FIGURE 4–2

In Boolean algebra, a **product term** is the product of literals. In logic circuits, a product term is produced by an AND operation with no OR operations involved. Some examples of product terms are AB, $A\overline{B}$, ABC, and $A\overline{B}C\overline{D}$.

A product term is equal to 1 only if each of the literals in the term is 1. A product term is equal to 0 when one or more of the literals are 0.

EXAMPLE 4–2

Determine the values of A, B, C, and D that make the product term $A\overline{B}C\overline{D}$ equal to 1.

Solution

For the product term to be 1, each of the literals in the term must be 1. Therefore, $A = \mathbf{1}$, $B = \mathbf{0}$ so that $\overline{B} = 1$, $C = \mathbf{1}$, and $D = \mathbf{0}$ so that $\overline{D} = 1$.

$$A\overline{B}C\overline{D} = 1 \cdot \overline{0} \cdot 1 \cdot \overline{0} = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

Related Problem

Determine the values of A and B that make the product term $\overline{A}\,\overline{B}$ equal to 1.

SECTION 4–1 CHECKUP

Answers are at the end of the chapter.

1. If $A = 0$, what does $\overline{A}$ equal?
2. Determine the values of A, B, and C that make the sum term $\overline{A} + \overline{B} + C$ equal to 0.
3. Determine the values of A, B, and C that make the product term $A\overline{B}C$ equal to 1.

4–2 Laws and Rules of Boolean Algebra

As in other areas of mathematics, there are certain well-developed rules and laws that must be followed in order to properly apply Boolean algebra. The most important of these are presented in this section.

After completing this section, you should be able to

- Apply the commutative laws of addition and multiplication
- Apply the associative laws of addition and multiplication
- Apply the distributive law
- Apply twelve basic rules of Boolean algebra

Laws of Boolean Algebra

The basic laws of Boolean algebra—the **commutative laws** for addition and multiplication, the **associative laws** for addition and multiplication, and the **distributive law**—are the same as in ordinary algebra. Each of the laws is illustrated with two or three variables, but the number of variables is not limited to this.

Commutative Laws

The *commutative law of addition* for two variables is written as

$$A + B = B + A \qquad \text{Equation 4–1}$$

This law states that the order in which the variables are ORed makes no difference. Remember, in Boolean algebra as applied to logic circuits, addition and the OR operation are the same. Figure 4–3 illustrates the commutative law as applied to the OR gate and shows that it doesn't matter to which input each variable is applied. (The symbol $\equiv$ means "equivalent to.")

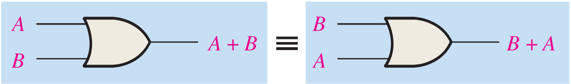


FIGURE 4–3 Application of commutative law of addition.

The *commutative law of multiplication* for two variables is

$$AB = BA \qquad \text{Equation 4–2}$$

This law states that the order in which the variables are ANDed makes no difference. Figure 4–4 illustrates this law as applied to the AND gate. Remember, in Boolean algebra as applied to logic circuits, multiplication and the AND function are the same.

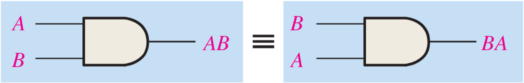


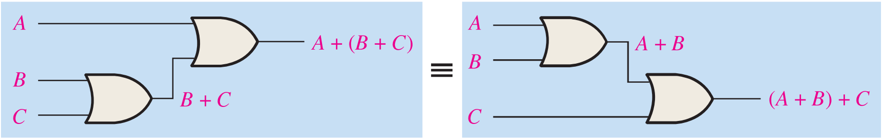
FIGURE 4–4 Application of commutative law of multiplication.

Associative Laws

The *associative law of addition* is written as follows for three variables:

$$A + (B + C) = (A + B) + C \qquad \text{Equation 4–3}$$

This law states that when ORing more than two variables, the result is the same regardless of the grouping of the variables. Figure 4–5 illustrates this law as applied to 2-input OR gates.



MultiSim

FIGURE 4–5 Application of associative law of addition. Open file F04-05 to verify. *A Multisim tutorial is available on the website.*

The *associative law of multiplication* is written as follows for three variables:

$$A(BC) = (AB)C \qquad \text{Equation 4–4}$$

This law states that it makes no difference in what order the variables are grouped when ANDing more than two variables. Figure 4–6 illustrates this law as applied to 2-input AND gates.

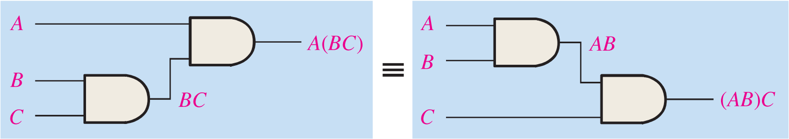


FIGURE 4–6 Application of associative law of multiplication. Open file F04-06 to verify.

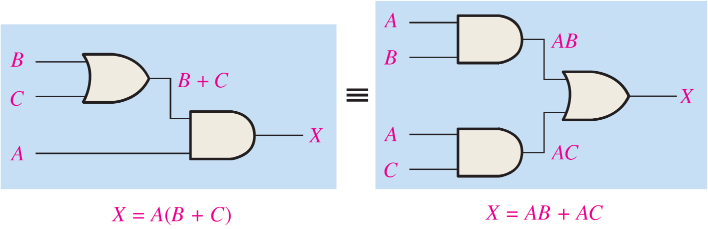
MultiSim

Distributive Law

The distributive law is written for three variables as follows:

$$A(B + C) = AB + AC \qquad \text{Equation 4–5}$$

This law states that ORing two or more variables and then ANDing the result with a single variable is equivalent to ANDing the single variable with each of the two or more variables and then ORing the products. The distributive law also expresses the process of *factoring* in which the common variable A is factored out of the product terms, for example, $AB + AC = A(B + C)$. Figure 4–7 illustrates the distributive law in terms of gate implementation.



$X = A(B + C)$

$X = AB + AC$

FIGURE 4–7 Application of distributive law. Open file F04-07 to verify.

MultiSim

Rules of Boolean Algebra

Table 4–1 lists 12 basic rules that are useful in manipulating and simplifying **Boolean expressions**. Rules 1 through 9 will be viewed in terms of their application to logic gates. Rules 10 through 12 will be derived in terms of the simpler rules and the laws previously discussed.

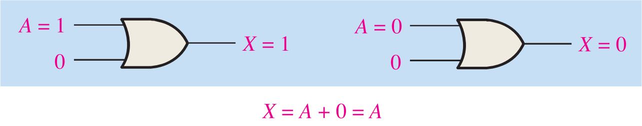
TABLE 4–1

Basic rules of Boolean algebra.

1. $A + 0 = A$
2. $A + 1 = 1$
3. $A \cdot 0 = 0$
4. $A \cdot 1 = A$
5. $A + A = A$
6. $A + \overline{A} = 1$
7. $A \cdot A = A$
8. $A \cdot \overline{A} = 0$
9. $\overline{\overline{A}} = A$
10. $A + AB = A$
11. $A + \overline{A}B = A + B$
12. $(A + B)(A + C) = A + BC$

A, B, or C can represent a single variable or a combination of variables.

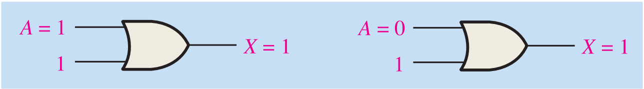
Rule 1: $A + 0 = A$ A variable ORed with 0 is always equal to the variable. If the input variable A is 1, the output variable X is 1, which is equal to A. If A is 0, the output is 0, which is also equal to A. This rule is illustrated in Figure 4–8, where the lower input is fixed at 0.



$X = A + 0 = A$

FIGURE 4–8

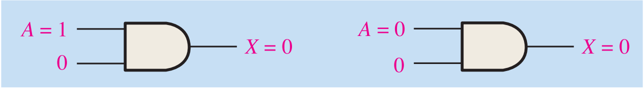
Rule 2: $A + 1 = 1$ A variable ORed with 1 is always equal to 1. A 1 on an input to an OR gate produces a 1 on the output, regardless of the value of the variable on the other input. This rule is illustrated in Figure 4–9, where the lower input is fixed at 1.



$X = A + 1 = 1$

FIGURE 4–9

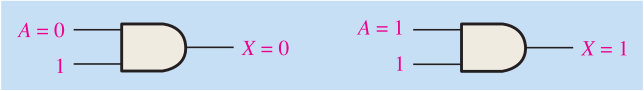
Rule 3: $A \cdot 0 = 0$ A variable ANDed with 0 is always equal to 0. Any time one input to an AND gate is 0, the output is 0, regardless of the value of the variable on the other input. This rule is illustrated in Figure 4–10, where the lower input is fixed at 0.



$X = A \bullet 0 = 0$

FIGURE 4–10

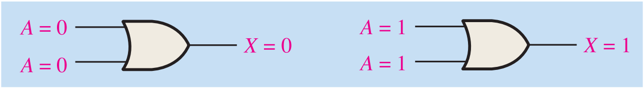
Rule 4: $A \cdot 1 = A$ A variable ANDed with 1 is always equal to the variable. If A is 0, the output of the AND gate is 0. If A is 1, the output of the AND gate is 1 because both inputs are now 1s. This rule is shown in Figure 4–11, where the lower input is fixed at 1.



$X = A \bullet 1 = A$

FIGURE 4–11

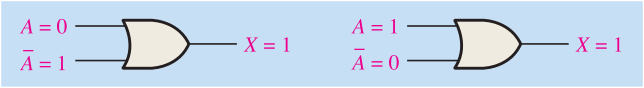
Rule 5: $A + A = A$ A variable ORed with itself is always equal to the variable. If A is 0, then $0 + 0 = 0$; and if A is 1, then $1 + 1 = 1$. This is shown in Figure 4–12, where both inputs are the same variable.



$X = A + A = A$

FIGURE 4–12

Rule 6: $A + \overline{A} = 1$ A variable ORed with its complement is always equal to 1. If A is 0, then $0 + \overline{0} = 0 + 1 = 1$. If A is 1, then $1 + \overline{1} = 1 + 0 = 1$. See Figure 4–13, where one input is the complement of the other.



$X = A + \overline{A} = 1$

FIGURE 4–13

Rule 7: $A \cdot A = A$ A variable ANDed with itself is always equal to the variable. If $A = 0$, then $0 \cdot 0 = 0$; and if $A = 1$, then $1 \cdot 1 = 1$. Figure 4–14 illustrates this rule.

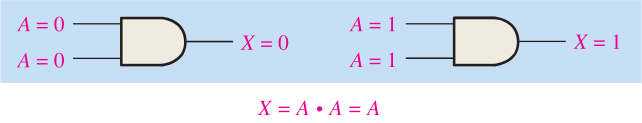


FIGURE 4–14

Rule 8: $A \cdot \overline{A} = 0$ A variable ANDed with its complement is always equal to 0. Either A or $\overline{A}$ will always be 0; and when a 0 is applied to the input of an AND gate, the output will be 0 also. Figure 4–15 illustrates this rule.

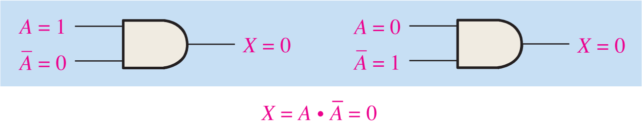


FIGURE 4–15

Rule 9: $\overline{\overline{A}} = A$ The double complement of a variable is always equal to the variable. If you start with the variable A and complement (invert) it once, you get $\overline{A}$. If you then take $\overline{A}$ and complement (invert) it, you get A, which is the original variable. This rule is shown in Figure 4–16 using inverters.

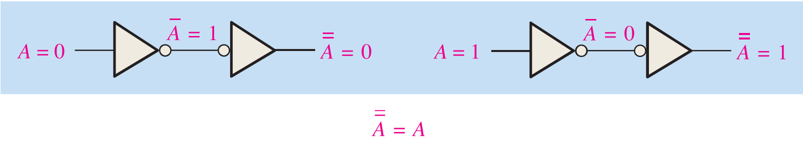

$A = 0$
$\overline{A} = 1$
$\overline{\overline{A}} = 0$
$A = 1$
$\overline{A} = 0$
$\overline{\overline{A}} = 1$
$\overline{\overline{A}} = A$

FIGURE 4–16

Rule 10: $A + AB = A$ This rule can be proved by applying the distributive law, rule 2, and rule 4 as follows:

$$
\begin{aligned}
A + AB &= A \cdot 1 + AB = A(1 + B) && \text{Factoring (distributive law)}\\
&= A \cdot 1 && \text{Rule 2: } (1 + B) = 1\\
&= A && \text{Rule 4: } A \cdot 1 = A
\end{aligned}
$$

The proof is shown in Table 4–2, which shows the truth table and the resulting logic circuit simplification.

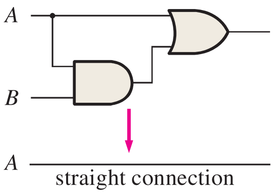
TABLE 4–2

Rule 10: $A + AB = A$. Open file T04-02 to verify.

MultiSim

A	B	AB	$A + AB$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

equal



Rule 11: $A + \overline{A}B = A + B$ This rule can be proved as follows:

$$\begin{aligned}
A + \overline{A}B &= (A + AB) + \overline{A}B && \text{Rule 10: } A = A + AB\\
&= (AA + AB) + \overline{A}B && \text{Rule 7: } A = AA\\
&= AA + AB + A\overline{A} + \overline{A}B && \text{Rule 8: adding } A\overline{A} = 0\\
&= (A + \overline{A})(A + B) && \text{Factoring}\\
&= 1 \cdot (A + B) && \text{Rule 6: } A + \overline{A} = 1\\
&= A + B && \text{Rule 4: drop the 1}
\end{aligned}$$

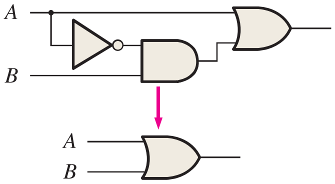
The proof is shown in Table 4–3, which shows the truth table and the resulting logic circuit simplification.

TABLE 4–3

Rule 11: $A + \overline{A}B = A + B$. Open file T04-03 to verify.

A	B	$\overline{A}B$	$A + \overline{A}B$	$A + B$
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

equal



Rule 12: $(A + B)(A + C) = A + BC$ This rule can be proved as follows:

$$\begin{aligned}
(A + B)(A + C) &= AA + AC + AB + BC && \text{Distributive law}\\
&= A + AC + AB + BC && \text{Rule 7: } AA = A\\
&= A(1 + C) + AB + BC && \text{Factoring (distributive law)}\\
&= A \cdot 1 + AB + BC && \text{Rule 2: } 1 + C = 1\\
&= A(1 + B) + BC && \text{Factoring (distributive law)}\\
&= A \cdot 1 + BC && \text{Rule 2: } 1 + B = 1\\
&= A + BC && \text{Rule 4: } A \cdot 1 = A
\end{aligned}$$

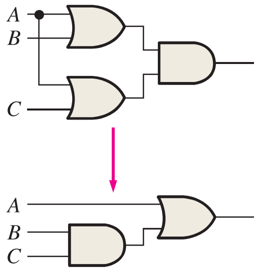
The proof is shown in Table 4–4, which shows the truth table and the resulting logic circuit simplification.

TABLE 4–4

Rule 12: $(A + B)(A + C) = A + BC$. Open file T04-04 to verify.

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

equal



SECTION 4–2 CHECKUP

1. Apply the associative law of addition to the expression $A + (B + C + D)$.
2. Apply the distributive law to the expression $A(B + C + D)$.

4–3 DeMorgan's Theorems

DeMorgan, a mathematician who knew Boole, proposed two theorems that are an important part of Boolean algebra. In practical terms, DeMorgan's theorems provide mathematical verification of the equivalency of the NAND and negative-OR gates and the equivalency of the NOR and negative-AND gates, which were discussed in Chapter 3.

After completing this section, you should be able to

- State DeMorgan's theorems
- Relate DeMorgan's theorems to the equivalency of the NAND and negative-OR gates and to the equivalency of the NOR and negative-AND gates
- Apply DeMorgan's theorems to the simplification of Boolean expressions

DeMorgan's first theorem is stated as follows:

To apply DeMorgan's theorem, break the bar over the product of variables and change the sign from AND to OR.

The complement of a product of variables is equal to the sum of the complements of the variables.

Stated another way,

The complement of two or more ANDed variables is equivalent to the OR of the complements of the individual variables.

The formula for expressing this theorem for two variables is

$$\overline{XY} = \overline{X} + \overline{Y} \qquad \text{Equation 4–6}$$

DeMorgan's second theorem is stated as follows:

The complement of a sum of variables is equal to the product of the complements of the variables.

Stated another way,

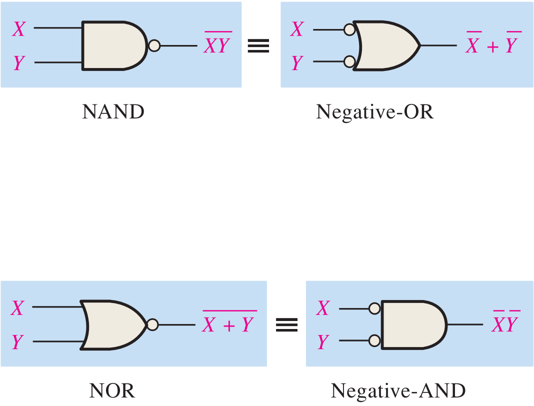
The complement of two or more ORed variables is equivalent to the AND of the complements of the individual variables.

The formula for expressing this theorem for two variables is

$$\overline{X + Y} = \overline{X}\,\overline{Y} \qquad \text{Equation 4–7}$$

Figure 4–17 shows the gate equivalencies and truth tables for Equations 4–6 and 4–7.

As stated, DeMorgan's theorems also apply to expressions in which there are more than two variables. The following examples illustrate the application of DeMorgan's theorems to 3-variable and 4-variable expressions.



Inputs		Output	
X	Y	$\overline{XY}$	$\overline{X}+\overline{Y}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Inputs		Output	
X	Y	$\overline{X+Y}$	$\overline{X}\,\overline{Y}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

FIGURE 4–17 Gate equivalencies and the corresponding truth tables that illustrate DeMorgan's theorems. Notice the equality of the two output columns in each table. This shows that the equivalent gates perform the same logic function.

EXAMPLE 4–3

Apply DeMorgan's theorems to the expressions $\overline{XYZ}$ and $\overline{X+Y+Z}$.

Solution

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

Related Problem

Apply DeMorgan's theorem to the expression $\overline{\overline{X}+\overline{Y}+\overline{Z}}$.

EXAMPLE 4–4

Apply DeMorgan's theorems to the expressions $\overline{WXYZ}$ and $\overline{W+X+Y+Z}$.

Solution

$$\overline{WXYZ} = \overline{W}+\overline{X}+\overline{Y}+\overline{Z}$$
$$\overline{W+X+Y+Z} = \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z}$$

Related Problem

Apply DeMorgan's theorem to the expression $\overline{\overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z}}$.

Each variable in DeMorgan's theorems as stated in Equations 4–6 and 4–7 can also represent a combination of other variables. For example, X can be equal to the term $AB + C$, and Y can be equal to the term $A + BC$. So if you can apply DeMorgan's theorem for two variables as stated by $\overline{XY} = \overline{X} + \overline{Y}$ to the expression $\overline{(AB + C)(A + BC)}$, you get the following result:

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

Notice that in the preceding result you have two terms, $\overline{AB + C}$ and $\overline{A + BC}$, to each of which you can again apply DeMorgan's theorem $\overline{X + Y} = \overline{X}\,\overline{Y}$ individually, as follows:

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

Notice that you still have two terms in the expression to which DeMorgan's theorem can again be applied. These terms are $\overline{AB}$ and $\overline{BC}$. A final application of DeMorgan's theorem gives the following result:

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

Although this result can be simplified further by the use of Boolean rules and laws, DeMorgan's theorems cannot be used any more.

Applying DeMorgan's Theorems

The following procedure illustrates the application of DeMorgan's theorems and Boolean algebra to the specific expression

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

Step 1: Identify the terms to which you can apply DeMorgan's theorems, and think of each term as a single variable. Let $\overline{A + B\overline{C}} = X$ and $D(\overline{E + \overline{F}}) = Y$.

Step 2: Since $\overline{X + Y} = \overline{X}\,\overline{Y}$,

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

Step 3: Use rule 9 ($\overline{\overline{A}} = A$) to cancel the double bars over the left term (this is not part of DeMorgan's theorem).

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

Step 4: Apply DeMorgan's theorem to the second term.

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

Step 5: Use rule 9 ($\overline{\overline{A}} = A$) to cancel the double bars over the $E + \overline{F}$ part of the term.

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

The following three examples will further illustrate how to use DeMorgan's theorems.

EXAMPLE 4–5

Apply DeMorgan's theorems to each of the following expressions:

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

(b) $\overline{ABC + DEF}$

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

Solution

(a) Let $A + B + C = X$ and $D = Y$. The expression $\overline{(A + B + C)D}$ is of the form $\overline{XY} = \overline{X} + \overline{Y}$ and can be rewritten as

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

Next, apply DeMorgan's theorem to the term $\overline{A + B + C}$.

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

(b) Let $ABC = X$ and $DEF = Y$. The expression $\overline{ABC + DEF}$ is of the form $\overline{X + Y} = \overline{X}\,\overline{Y}$ and can be rewritten as

$$\overline{ABC + DEF} = (\overline{ABC})(\overline{DEF})$$

Next, apply DeMorgan's theorem to each of the terms $\overline{ABC}$ and $\overline{DEF}$.

$$(\overline{ABC})(\overline{DEF}) = (\overline{A} + \overline{B} + \overline{C})(\overline{D} + \overline{E} + \overline{F})$$

(c) Let $A\overline{B} = X$, $\overline{C}D = Y$, and $EF = Z$. The expression $\overline{A\overline{B} + \overline{C}D + EF}$ is of the form $\overline{X + Y + Z} = \overline{X}\,\overline{Y}\,\overline{Z}$ and can be rewritten as

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

Next, apply DeMorgan's theorem to each of the terms $\overline{A\overline{B}}$, $\overline{\overline{C}D}$, and $\overline{EF}$.

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

Related Problem

Apply DeMorgan's theorems to the expression $\overline{\overline{ABC} + D + E}$.

EXAMPLE 4–6

Apply DeMorgan's theorems to each expression:

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

(b) $\overline{(\overline{A} + B) + CD}$

(c) $\overline{(A + B)\overline{C}\,\overline{D} + E + \overline{F}}$

Solution

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

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

(c) $\overline{(A + B)\overline{C}\,\overline{D} + E + \overline{F}} = \overline{((A + B)\overline{C}\,\overline{D})}(\overline{E + \overline{F}}) = (\overline{A}\,\overline{B} + C + D)\overline{E}F$

Related Problem

Apply DeMorgan's theorems to the expression $\overline{\overline{A}B(C + \overline{D}) + E}$.

EXAMPLE 4–7

The Boolean expression for an exclusive-OR gate is $A\overline{B} + \overline{A}B$. With this as a starting point, use DeMorgan's theorems and any other rules or laws that are applicable to develop an expression for the exclusive-NOR gate.

Solution

Start by complementing the exclusive-OR expression and then applying DeMorgan's theorems as follows:

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

Next, apply the distributive law and rule 8 ($A \cdot \overline{A} = 0$).

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

The final expression for the XNOR is $\overline{A}\,\overline{B} + AB$. Note that this expression equals 1 any time both variables are 0s or both variables are 1s.

Related Problem

Starting with the expression for a 4-input NAND gate, use DeMorgan's theorems to develop an expression for a 4-input negative-OR gate.

SECTION 4–3 CHECKUP

1. Apply DeMorgan's theorems to the following expressions:
(a) $\overline{ABC} + (\overline{\overline{D} + E})$ (b) $\overline{(A + B)C}$ (c) $\overline{A + B + C} + \overline{\overline{D}E}$

4–4 Boolean Analysis of Logic Circuits

Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.

After completing this section, you should be able to

- Determine the Boolean expression for a combination of gates
- Evaluate the logic operation of a circuit from the Boolean expression
- Construct a truth table

Boolean Expression for a Logic Circuit

A combinational logic circuit can be described by a Boolean equation.

To derive the Boolean expression for a given combinational logic circuit, begin at the left-most inputs and work toward the final output, writing the expression for each gate. For the example circuit in Figure 4–18, the Boolean expression is determined in the following three steps:

1. The expression for the left-most AND gate with inputs C and D is CD.
2. The output of the left-most AND gate is one of the inputs to the OR gate and B is the other input. Therefore, the expression for the OR gate is $B + CD$.
3. The output of the OR gate is one of the inputs to the right-most AND gate and A is the other input. Therefore, the expression for this AND gate is $A(B + CD)$, which is the final output expression for the entire circuit.

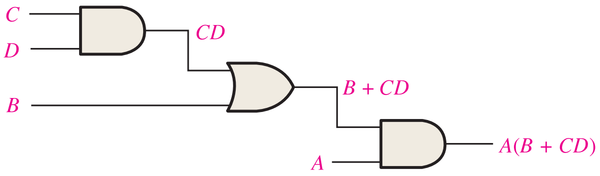


FIGURE 4–18 A combinational logic circuit showing the development of the Boolean expression for the output.

Constructing a Truth Table for a Logic Circuit

A combinational logic circuit can be described by a truth table.

Once the Boolean expression for a given logic circuit has been determined, a truth table that shows the output for all possible values of the input variables can be developed. The procedure requires that you evaluate the Boolean expression for all possible combinations of values for the input variables. In the case of the circuit in Figure 4–18, there are four input variables (A, B, C, and D) and therefore sixteen ($2^4 = 16$) combinations of values are possible.

Evaluating the Expression

To evaluate the expression $A(B + CD)$, first find the values of the variables that make the expression equal to 1, using the rules for Boolean addition and multiplication. In this case, the expression equals 1 only if $A = 1$ and $B + CD = 1$ because

$$A(B + CD) = 1 \cdot 1 = 1$$

Now determine when the $B + CD$ term equals 1. The term $B + CD = 1$ if either $B = 1$ or $CD = 1$ or if both B and CD equal 1 because

$$B + CD = 1 + 0 = 1$$
$$B + CD = 0 + 1 = 1$$
$$B + CD = 1 + 1 = 1$$

The term $CD = 1$ only if $C = 1$ and $D = 1$.

To summarize, the expression $A(B + CD) = 1$ when $A = 1$ and $B = 1$ regardless of the values of C and D or when $A = 1$ and $C = 1$ and $D = 1$ regardless of the value of B. The expression $A(B + CD) = 0$ for all other value combinations of the variables.

Putting the Results in Truth Table Format

The first step is to list the sixteen input variable combinations of 1s and 0s in a binary sequence as shown in Table 4–5. Next, place a 1 in the output column for each combination of input variables that was determined in the evaluation. Finally, place a 0 in the output column for all other combinations of input variables. These results are shown in the truth table in Table 4–5.

TABLE 4–5

Truth table for the logic circuit in Figure 4–18.

Inputs				Output
A	*B*	*C*	*D*	$A(B + CD)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

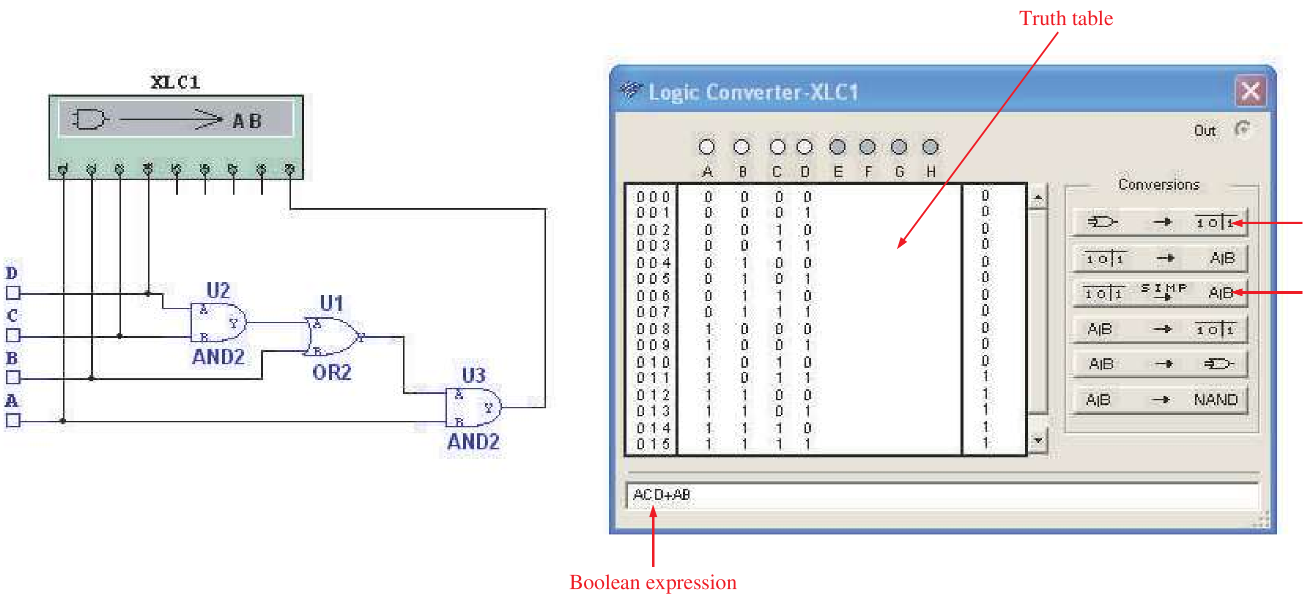
EXAMPLE 4–8

Use Multisim to generate the truth table for the logic circuit in Figure 4–18.

Solution

Construct the circuit in Multisim and connect the Multisim Logic Converter to the inputs and output, as shown in Figure 4–19. Click on the ⊐▷ → 1 0|1 conversion bar, and the truth table appears in the display as shown.

You can also generate the simplified Boolean expression from the truth table by clicking on 1 0|1 SIMP→ A|B.



Truth table

	A	B	C	D	E	F	G	H	Out
000	0	0	0	0					0
001	0	0	0	1					0
002	0	0	1	0					0
003	0	0	1	1					0
004	0	1	0	0					0
005	0	1	0	1					0
006	0	1	1	0					0
007	0	1	1	1					0
008	1	0	0	0					0
009	1	0	0	1					0
010	1	0	1	0					0
011	1	0	1	1					1
012	1	1	0	0					1
013	1	1	0	1					1
014	1	1	1	0					1
015	1	1	1	1					1

ACD+AB

Boolean expression

FIGURE 4–19

Related Problem

Open Multisim. Create the setup and do the conversions shown in this example.

MultiSim

SECTION 4–4 CHECKUP

1. Replace the AND gates with OR gates and the OR gate with an AND gate in Figure 4–18. Determine the Boolean expression for the output.
2. Construct a truth table for the circuit in Question 1.

4–5 Logic Simplification Using Boolean Algebra

A logic expression can be reduced to its simplest form or changed to a more convenient form to implement the expression most efficiently using Boolean algebra. The approach taken in this section is to use the basic laws, rules, and theorems of Boolean algebra to manipulate and simplify an expression. This method depends on a thorough knowledge of Boolean algebra and considerable practice in its application, not to mention a little ingenuity and cleverness.

After completing this section, you should be able to

- Apply the laws, rules, and theorems of Boolean algebra to simplify general expressions

A simplified Boolean expression uses the fewest gates possible to implement a given expression. Examples 4–9 through 4–12 illustrate Boolean simplification.

EXAMPLE 4–9

Using Boolean algebra techniques, simplify this expression:

$$AB + A(B + C) + B(B + C)$$

Solution

The following is not necessarily the only approach.

Step 1: Apply the distributive law to the second and third terms in the expression, as follows:

$$AB + AB + AC + BB + BC$$

Step 2: Apply rule 7 ($BB = B$) to the fourth term.

$$AB + AB + AC + B + BC$$

Step 3: Apply rule 5 ($AB + AB = AB$) to the first two terms.

$$AB + AC + B + BC$$

Step 4: Apply rule 10 ($B + BC = B$) to the last two terms.

$$AB + AC + B$$

Step 5: Apply rule 10 ($AB + B = B$) to the first and third terms.

$$B + AC$$

At this point the expression is simplified as much as possible. Once you gain experience in applying Boolean algebra, you can often combine many individual steps.

Related Problem

Simplify the Boolean expression $A\overline{B} + A(\overline{B + C}) + B(\overline{B + C})$.

Simplification means fewer gates for the same function.

Figure 4–20 shows that the simplification process in Example 4–9 has significantly reduced the number of logic gates required to implement the expression. Part (a) shows that five gates are required to implement the expression in its original form; however, only two gates are needed for the simplified expression, shown in part (b). It is important to realize that these two gate circuits are equivalent. That is, for any combination of levels on the A, B, and C inputs, you get the same output from either circuit.

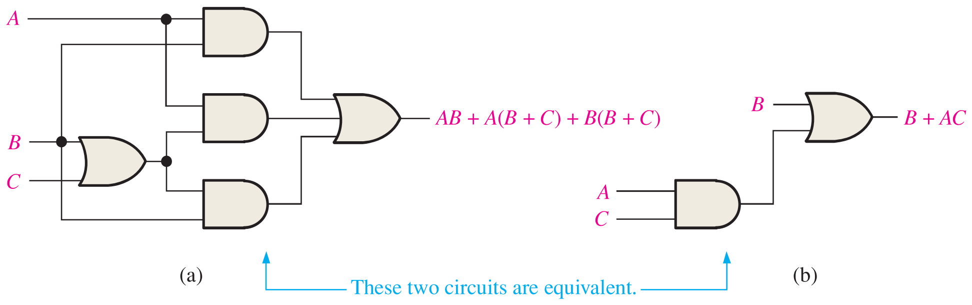


FIGURE 4–20 Gate circuits for Example 4–9. Open file F04-20 to verify equivalency.

EXAMPLE 4–10

Simplify the following Boolean expression:

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

Note that brackets and parentheses mean the same thing: the term inside is multiplied (ANDed) with the term outside.

Solution

Step 1: Apply the distributive law to the terms within the brackets.

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

Step 2: Apply rule 8 ($\overline{B}B = 0$) to the second term within the parentheses.

$$(A\overline{B}C + A \cdot 0 \cdot D + \overline{A}\,\overline{B})C$$

Step 3: Apply rule 3 ($A \cdot 0 \cdot D = 0$) to the second term within the parentheses.

$$(A\overline{B}C + 0 + \overline{A}\,\overline{B})C$$

Step 4: Apply rule 1 (drop the 0) within the parentheses.

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

Step 5: Apply the distributive law.

$$A\overline{B}CC + \overline{A}\,\overline{B}C$$

Step 6: Apply rule 7 ($CC = C$) to the first term.

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

Step 7: Factor out $\overline{B}C$.

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

Step 8: Apply rule 6 ($A + \overline{A} = 1$).

$$\overline{B}C \cdot 1$$

Step 9: Apply rule 4 (drop the 1).

$$\overline{B}C$$

Related Problem

Simplify the Boolean expression $[AB(C + \overline{BD}) + \overline{AB}]CD$.

EXAMPLE 4–11

Simplify the following Boolean expression:

$$\overline{A}BC + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}C + ABC$$

Solution

Step 1: Factor BC out of the first and last terms.

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

Step 2: Apply rule 6 ($\overline{A} + A = 1$) to the term in parentheses, and factor $A\overline{B}$ from the second and last terms.

$$BC \cdot 1 + A\overline{B}(\overline{C} + C) + \overline{A}\,\overline{B}\,\overline{C}$$

Step 3: Apply rule 4 (drop the 1) to the first term and rule 6 ($\overline{C} + C = 1$) to the term in parentheses.

$$BC + A\overline{B} \cdot 1 + \overline{A}\,\overline{B}\,\overline{C}$$

Step 4: Apply rule 4 (drop the 1) to the second term.

$$BC + A\overline{B} + \overline{A}\,\overline{B}\,\overline{C}$$

Step 5: Factor $\overline{B}$ from the second and third terms.

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

Step 6: Apply rule 11 ($A + \overline{A}\,\overline{C} = A + \overline{C}$) to the term in parentheses.

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

Step 7: Use the distributive and commutative laws to get the following expression:

$$BC + A\overline{B} + \overline{B}\,\overline{C}$$

Related Problem

Simplify the Boolean expression $AB\overline{C} + \overline{A}\,\overline{B}C + \overline{A}BC + \overline{A}\,\overline{B}\,\overline{C}$.

EXAMPLE 4–12

Simplify the following Boolean expression:

$$\overline{AB + AC} + \overline{A}\,\overline{B}C$$

Solution

Step 1: Apply DeMorgan's theorem to the first term.

$$(\overline{AB})(\overline{AC}) + \overline{A}\,\overline{B}C$$

Step 2: Apply DeMorgan's theorem to each term in parentheses.

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

Step 3: Apply the distributive law to the two terms in parentheses.

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

Step 4: Apply rule 7 ($\overline{A}\,\overline{A} = \overline{A}$) to the first term, and apply rule 10
[$\overline{A}\,\overline{B} + \overline{A}\,\overline{B}C = \overline{A}\,\overline{B}(1 + C) = \overline{A}\,\overline{B}$] to the third and last terms.

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

Step 5: Apply rule 10 [$\overline{A} + \overline{A}\,\overline{C} = \overline{A}(1 + \overline{C}) = \overline{A}$] to the first and second terms.

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

Step 6: Apply rule 10 [$\overline{A} + \overline{A}\,\overline{B} = \overline{A}(1 + \overline{B}) = \overline{A}$] to the first and second terms.

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

Related Problem

Simplify the Boolean expression $\overline{AB} + \overline{AC} + \overline{A}\,\overline{B}\,\overline{C}$.

EXAMPLE 4–13

Use Multisim to perform the logic simplification shown in Figure 4–20.

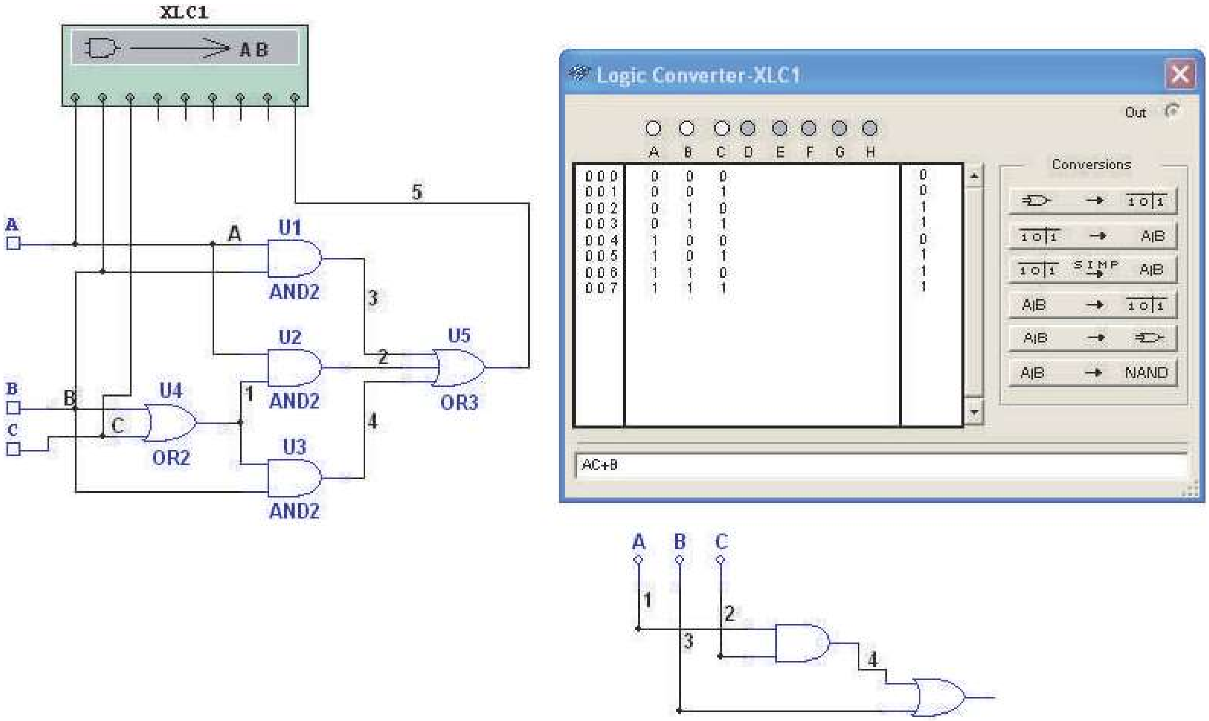
Solution

Step 1: Connect the Multisim Logic Converter to the circuit as shown in Figure 4–21.

Step 2: Generate the truth table by clicking on ⊐▷ → 1 0|1.

Step 3: Generate the simplified Boolean expression by clicking on 1 0|1 SIMP→ A|B.

Step 4: Generate the simplified logic circuit by clicking on A|B → ⊐▷.



	A	B	C	
000	0	0	0	0
001	0	0	1	0
002	0	1	0	1
003	0	1	1	1
004	1	0	0	0
005	1	0	1	1
006	1	1	0	1
007	1	1	1	1

AC+B

FIGURE 4–21

Related Problem

Open Multisim. Create the setup and perform the logic simplification illustrated in this example.

MultiSim

SECTION 4–5 CHECKUP

1. Simplify the following Boolean expressions:

   (a) $A + AB + A\overline{B}C$ **(b)** $(\overline{A} + B)C + ABC$ **(c)** $A\overline{B}C(BD + CDE) + A\overline{C}$

2. Implement each expression in Question 1 as originally stated with the appropriate logic gates. Then implement the simplified expression, and compare the number of gates.

4–6 Standard Forms of Boolean Expressions

All Boolean expressions, regardless of their form, can be converted into either of two standard forms: the sum-of-products form or the product-of-sums form. Standardization makes the evaluation, simplification, and implementation of Boolean expressions much more systematic and easier.

After completing this section, you should be able to

- Identify a sum-of-products expression
- Determine the domain of a Boolean expression
- Convert any sum-of-products expression to a standard form
- Evaluate a standard sum-of-products expression in terms of binary values
- Identify a product-of-sums expression

- Convert any product-of-sums expression to a standard form
- Evaluate a standard product-of-sums expression in terms of binary values
- Convert from one standard form to the other

The Sum-of-Products (SOP) Form

An SOP expression can be implemented with one OR gate and two or more AND gates.

A product term was defined in Section 4–1 as a term consisting of the product (Boolean multiplication) of literals (variables or their complements). When two or more product terms are summed by Boolean addition, the resulting expression is a **sum-of-products (SOP)**. Some examples are

$$AB + ABC$$
$$ABC + CDE + \overline{B}C\overline{D}$$
$$\overline{A}B + \overline{A}B\overline{C} + AC$$

Also, an SOP expression can contain a single-variable term, as in $A + \overline{A}\,\overline{B}C + BC\overline{D}$. Refer to the simplification examples in the last section, and you will see that each of the final expressions was either a single product term or in SOP form. In an SOP expression, a single overbar cannot extend over more than one variable; however, more than one variable in a term can have an overbar. For example, an SOP expression can have the term $\overline{A}\,\overline{B}\,\overline{C}$ but not $\overline{ABC}$.

Domain of a Boolean Expression

The **domain** of a general Boolean expression is the set of variables contained in the expression in either complemented or uncomplemented form. For example, the domain of the expression $\overline{A}B + A\overline{B}C$ is the set of variables A, B, C and the domain of the expression $AB\overline{C} + C\overline{D}E + \overline{B}C\overline{D}$ is the set of variables A, B, C, D, E.

AND/OR Implementation of an SOP Expression

Implementing an SOP expression simply requires ORing the outputs of two or more AND gates. A product term is produced by an AND operation, and the sum (addition) of two or more product terms is produced by an OR operation. Therefore, an SOP expression can be implemented by AND-OR logic in which the outputs of a number (equal to the number of product terms in the expression) of AND gates connect to the inputs of an OR gate, as shown in Figure 4–22 for the expression $AB + BCD + AC$. The output X of the OR gate equals the SOP expression.

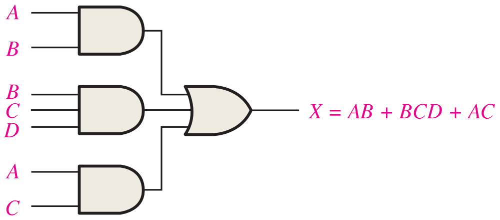


FIGURE 4–22 Implementation of the SOP expression $AB + BCD + AC$.

NAND/NAND Implementation of an SOP Expression

NAND gates can be used to implement an SOP expression. By using only NAND gates, an AND/OR function can be accomplished, as illustrated in Figure 4–23. The first level of NAND gates feed into a NAND gate that acts as a negative-OR gate. The NAND and negative-OR inversions cancel and the result is effectively an AND/OR circuit.

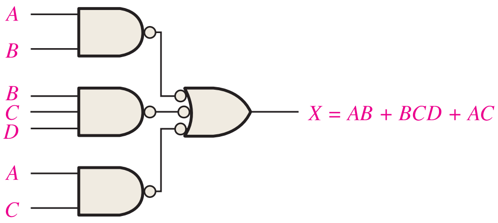


FIGURE 4–23 This NAND/NAND implementation is equivalent to the AND/OR in Figure 4–22.

Conversion of a General Expression to SOP Form

Any logic expression can be changed into SOP form by applying Boolean algebra techniques. For example, the expression $A(B + CD)$ can be converted to SOP form by applying the distributive law:

$$A(B + CD) = AB + ACD$$

EXAMPLE 4–14

Convert each of the following Boolean expressions to SOP form:

(a) $AB + B(CD + EF)$ **(b)** $(A + B)(B + C + D)$ **(c)** $\overline{(\overline{A + B}) + C}$

Solution

(a) $AB + B(CD + EF) = AB + BCD + BEF$

(b) $(A + B)(B + C + D) = AB + AC + AD + BB + BC + BD$

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

Related Problem

Convert $\overline{A}B\overline{C} + (A + \overline{B})(B + \overline{C} + A\overline{B})$ to SOP form.

The Standard SOP Form

So far, you have seen SOP expressions in which some of the product terms do not contain all of the variables in the domain of the expression. For example, the expression $\overline{A}B\overline{C} + A\overline{B}D + \overline{A}B\overline{C}D$ has a domain made up of the variables A, B, C, and D. However, notice that the complete set of variables in the domain is not represented in the first two terms of the expression; that is, D or $\overline{D}$ is missing from the first term and C or $\overline{C}$ is missing from the second term.

A *standard SOP expression* is one in which *all* the variables in the domain appear in each product term in the expression. For example, $A\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} + AB\overline{C}\,\overline{D}$ is a standard SOP expression. Standard SOP expressions are important in constructing truth tables, covered in Section 4–7, and in the Karnaugh map simplification method, which is covered in Section 4–8. Any nonstandard SOP expression (referred to simply as SOP) can be converted to the standard form using Boolean algebra.

Converting Product Terms to Standard SOP

Each product term in an SOP expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements. As stated in the following steps, a nonstandard SOP expression is converted into standard form using Boolean algebra rule 6 ($A + \overline{A} = 1$) from Table 4–1: A variable added to its complement equals 1.

Step 1: Multiply each nonstandard product term by a term made up of the sum of a missing variable and its complement. This results in two product terms. As you know, you can multiply anything by 1 without changing its value.

Step 2: Repeat Step 1 until all resulting product terms contain all variables in the domain in either complemented or uncomplemented form. In converting a product term to standard form, the number of product terms is doubled for each missing variable, as Example 4–15 shows.

EXAMPLE 4–15

Convert the following Boolean expression into standard SOP form:

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

Solution

The domain of this SOP expression is A, B, C, D. Take one term at a time. The first term, $A\overline{B}C$, is missing variable D or $\overline{D}$, so multiply the first term by $D + \overline{D}$ as follows:

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

In this case, two standard product terms are the result.

The second term, $\overline{A}\,\overline{B}$, is missing variables C or $\overline{C}$ and D or $\overline{D}$, so first multiply the second term by $C + \overline{C}$ as follows:

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

The two resulting terms are missing variable D or $\overline{D}$, so multiply both terms by $D + \overline{D}$ as follows:

$$\begin{aligned}\overline{A}\,\overline{B} &= \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C} = \overline{A}\,\overline{B}C(D + \overline{D}) + \overline{A}\,\overline{B}\,\overline{C}(D + \overline{D})\\ &= \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}\end{aligned}$$

In this case, four standard product terms are the result.

The third term, $AB\overline{C}D$, is already in standard form. The complete standard SOP form of the original expression is as follows:

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

Related Problem

Convert the expression $W\overline{X}Y + \overline{X}Y\overline{Z} + WX\overline{Y}$ to standard SOP form.

Binary Representation of a Standard Product Term

A standard product term is equal to 1 for only one combination of variable values. For example, the product term $A\overline{B}C\overline{D}$ is equal to 1 when $A = 1$, $B = 0$, $C = 1$, $D = 0$, as shown below, and is 0 for all other combinations of values for the variables.

$$A\overline{B}C\overline{D} = 1 \cdot \overline{0} \cdot 1 \cdot \overline{0} = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

In this case, the product term has a binary value of 1010 (decimal ten).

Remember, a product term is implemented with an AND gate whose output is 1 only if each of its inputs is 1. Inverters are used to produce the complements of the variables as required.

An SOP expression is equal to 1 only if one or more of the product terms in the expression is equal to 1.

EXAMPLE 4–16

Determine the binary values for which the following standard SOP expression is equal to 1:

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

Solution

The term $ABCD$ is equal to 1 when $A = 1$, $B = 1$, $C = 1$, and $D = 1$.

$$ABCD = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

The term $A\overline{B}\,\overline{C}D$ is equal to 1 when $A = 1$, $B = 0$, $C = 0$, and $D = 1$.

$$A\overline{B}\,\overline{C}D = 1 \cdot \overline{0} \cdot \overline{0} \cdot 1 = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

The term $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ is equal to 1 when $A = 0$, $B = 0$, $C = 0$, and $D = 0$.

$$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} = \overline{0} \cdot \overline{0} \cdot \overline{0} \cdot \overline{0} = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

The SOP expression equals 1 when any or all of the three product terms is 1.

Related Problem

Determine the binary values for which the following SOP expression is equal to 1:

$$\overline{X}YZ + X\overline{Y}Z + XY\overline{Z} + \overline{X}Y\overline{Z} + XYZ$$

Is this a standard SOP expression?

The Product-of-Sums (POS) Form

A sum term was defined in Section 4–1 as a term consisting of the sum (Boolean addition) of literals (variables or their complements). When two or more sum terms are multiplied, the resulting expression is a **product-of-sums (POS)**. Some examples are

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

A POS expression can contain a single-variable term, as in $\overline{A}(A + \overline{B} + C)(\overline{B} + \overline{C} + D)$. In a POS expression, a single overbar cannot extend over more than one variable; however, more than one variable in a term can have an overbar. For example, a POS expression can have the term $\overline{A} + \overline{B} + \overline{C}$ but not $\overline{A + B + C}$.

Implementation of a POS Expression

Implementing a POS expression simply requires ANDing the outputs of two or more OR gates. A sum term is produced by an OR operation, and the product of two or more sum terms is produced by an AND operation. Therefore, a POS expression can be implemented by logic in which the outputs of a number (equal to the number of sum terms in the expression) of OR gates connect to the inputs of an AND gate, as Figure 4–24 shows for the expression $(A + B)(B + C + D)(A + C)$. The output X of the AND gate equals the POS expression.

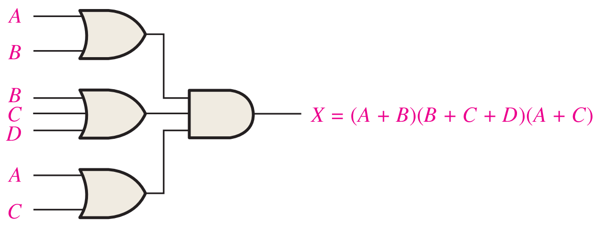


FIGURE 4–24 Implementation of the POS expression $(A + B)(B + C + D)(A + C)$.

The Standard POS Form

So far, you have seen POS expressions in which some of the sum terms do not contain all of the variables in the domain of the expression. For example, the expression

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

has a domain made up of the variables A, B, C, and D. Notice that the complete set of variables in the domain is not represented in the first two terms of the expression; that is, D or $\overline{D}$ is missing from the first term and C or $\overline{C}$ is missing from the second term.

A *standard POS expression* is one in which *all* the variables in the domain appear in each sum term in the expression. For example,

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

is a standard POS expression. Any nonstandard POS expression (referred to simply as POS) can be converted to the standard form using Boolean algebra.

Converting a Sum Term to Standard POS

Each sum term in a POS expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements. As stated in the following steps, a nonstandard POS expression is converted into standard form using Boolean algebra rule 8 ($A \cdot \overline{A} = 0$) from Table 4–1: A variable multiplied by its complement equals 0.

Step 1: Add to each nonstandard product term a term made up of the product of the missing variable and its complement. This results in two sum terms. As you know, you can add 0 to anything without changing its value.

Step 2: Apply rule 12 from Table 4–1: $A + BC = (A + B)(A + C)$

Step 3: Repeat Step 1 until all resulting sum terms contain all variables in the domain in either complemented or uncomplemented form.

EXAMPLE 4–17

Convert the following Boolean expression into standard POS form:

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

Solution

The domain of this POS expression is A, B, C, D. Take one term at a time. The first term, $A + \overline{B} + C$, is missing variable D or $\overline{D}$, so add $D\overline{D}$ and apply rule 12 as follows:

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

The second term, $\overline{B} + C + \overline{D}$, is missing variable A or $\overline{A}$, so add $A\overline{A}$ and apply rule 12 as follows:

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

The third term, $A + \overline{B} + \overline{C} + D$, is already in standard form. The standard POS form of the original expression is as follows:

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

Related Problem

Convert the expression $(A + \overline{B})(B + C)$ to standard POS form.

Binary Representation of a Standard Sum Term

A standard sum term is equal to 0 for only one combination of variable values. For example, the sum term $A + \overline{B} + C + \overline{D}$ is 0 when $A = 0$, $B = 1$, $C = 0$, and $D = 1$, as shown below, and is 1 for all other combinations of values for the variables.

$$A + \overline{B} + C + \overline{D} = 0 + \overline{1} + 0 + \overline{1} = 0 + 0 + 0 + 0 = 0$$

In this case, the sum term has a binary value of 0101 (decimal 5). Remember, a sum term is implemented with an OR gate whose output is 0 only if each of its inputs is 0. Inverters are used to produce the complements of the variables as required.

A POS expression is equal to 0 only if one or more of the sum terms in the expression is equal to 0.

EXAMPLE 4–18

Determine the binary values of the variables for which the following standard POS expression is equal to 0:

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

Solution

The term $A + B + C + D$ is equal to 0 when $A = 0$, $B = 0$, $C = 0$, and $D = 0$.

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

The term $A + \overline{B} + \overline{C} + D$ is equal to 0 when $A = 0$, $B = 1$, $C = 1$, and $D = 0$.

$$A + \overline{B} + \overline{C} + D = 0 + \overline{1} + \overline{1} + 0 = 0 + 0 + 0 + 0 = 0$$

The term $\overline{A} + \overline{B} + \overline{C} + \overline{D}$ is equal to 0 when $A = 1$, $B = 1$, $C = 1$, and $D = 1$.

$$\overline{A} + \overline{B} + \overline{C} + \overline{D} = \overline{1} + \overline{1} + \overline{1} + \overline{1} = 0 + 0 + 0 + 0 = 0$$

The POS expression equals 0 when any of the three sum terms equals 0.

Related Problem

Determine the binary values for which the following POS expression is equal to 0:

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

Is this a standard POS expression?

Converting Standard SOP to Standard POS

The binary values of the product terms in a given standard SOP expression are not present in the equivalent standard POS expression. Also, the binary values that are not represented in the SOP expression are present in the equivalent POS expression. Therefore, to convert from standard SOP to standard POS, the following steps are taken:

Step 1: Evaluate each product term in the SOP expression. That is, determine the binary numbers that represent the product terms.

Step 2: Determine all of the binary numbers not included in the evaluation in Step 1.

Step 3: Write the equivalent sum term for each binary number from Step 2 and express in POS form.

Using a similar procedure, you can go from POS to SOP.

EXAMPLE 4–19

Convert the following SOP expression to an equivalent POS expression:

$$\overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + ABC$$

Solution

The evaluation is as follows:

$$000 + 010 + 011 + 101 + 111$$

Since there are three variables in the domain of this expression, there are a total of eight (2^3) possible combinations. The SOP expression contains five of these combinations, so the POS must contain the other three which are 001, 100, and 110. Remember, these are the binary values that make the sum term 0. The equivalent POS expression is

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

Related Problem

Verify that the SOP and POS expressions in this example are equivalent by substituting binary values into each.

SECTION 4–6 CHECKUP

1. Identify each of the following expressions as SOP, standard SOP, POS, or standard POS:

   (a) $AB + \overline{A}BD + \overline{A}C\overline{D}$ **(b)** $(A + \overline{B} + C)(A + B + \overline{C})$

   (c) $\overline{A}BC + AB\overline{C}$ **(d)** $(A + \overline{C})(A + B)$

2. Convert each SOP expression in Question 1 to standard form.
3. Convert each POS expression in Question 1 to standard form.

4–7 Boolean Expressions and Truth Tables

All standard Boolean expressions can be easily converted into truth table format using binary values for each term in the expression. The truth table is a common way of presenting, in a concise format, the logical operation of a circuit. Also, standard SOP or POS expressions can be determined from a truth table. You will find truth tables in data sheets and other literature related to the operation of digital circuits.

After completing this section, you should be able to

- Convert a standard SOP expression into truth table format
- Convert a standard POS expression into truth table format
- Derive a standard expression from a truth table
- Properly interpret truth table data

Converting SOP Expressions to Truth Table Format

Recall from Section 4–6 that an SOP expression is equal to 1 only if at least one of the product terms is equal to 1. A truth table is simply a list of the possible combinations of input variable values and the corresponding output values (1 or 0). For an expression with a domain of two variables, there are four different combinations of those variables ($2^2 = 4$). For an expression with a domain of three variables, there are eight different combinations of those variables ($2^3 = 8$). For an expression with a domain of four variables, there are sixteen different combinations of those variables ($2^4 = 16$), and so on.

The first step in constructing a truth table is to list all possible combinations of binary values of the variables in the expression. Next, convert the SOP expression to standard form if it is not already. Finally, place a 1 in the output column (X) for each binary value that makes the standard SOP expression a 1 and place a 0 for all the remaining binary values. This procedure is illustrated in Example 4–20.

EXAMPLE 4–20

Develop a truth table for the standard SOP expression $\overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + ABC$.

Solution

There are three variables in the domain, so there are eight possible combinations of binary values of the variables as listed in the left three columns of Table 4–6. The binary values that make the product terms in the expressions equal to 1 are

TABLE 4–6

Inputs			Output	
A	B	C	X	Product Term
0	0	0	0	
0	0	1	1	$\overline{A}\,\overline{B}C$
0	1	0	0	
0	1	1	0	
1	0	0	1	$A\overline{B}\,\overline{C}$
1	0	1	0	
1	1	0	0	
1	1	1	1	ABC

$\overline{A}\,\overline{B}C$: 001; $A\overline{B}\,\overline{C}$: 100; and ABC: 111. For each of these binary values, place a 1 in the output column as shown in the table. For each of the remaining binary combinations, place a 0 in the output column.

Related Problem

Create a truth table for the standard SOP expression $\overline{A}B\overline{C} + A\overline{B}C$.

Converting POS Expressions to Truth Table Format

Recall that a POS expression is equal to 0 only if at least one of the sum terms is equal to 0. To construct a truth table from a POS expression, list all the possible combinations of binary values of the variables just as was done for the SOP expression. Next, convert the POS expression to standard form if it is not already. Finally, place a 0 in the output column (X) for each binary value that makes the expression a 0 and place a 1 for all the remaining binary values. This procedure is illustrated in Example 4–21.

EXAMPLE 4–21

Determine the truth table for the following standard POS expression:

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

Solution

There are three variables in the domain and the eight possible binary values are listed in the left three columns of Table 4–7. The binary values that make the sum terms in the expression equal to 0 are $A + B + C$: 000; $A + \overline{B} + C$: 010; $A + \overline{B} + \overline{C}$: 011; $\overline{A} + B + \overline{C}$: 101; and $\overline{A} + \overline{B} + C$: 110. For each of these binary values, place a 0 in the output column as shown in the table. For each of the remaining binary combinations, place a 1 in the output column.

TABLE 4–7

Inputs			Output	
A	B	C	X	Sum Term
0	0	0	0	$(A + B + C)$
0	0	1	1	
0	1	0	0	$(A + \overline{B} + C)$
0	1	1	0	$(A + \overline{B} + \overline{C})$
1	0	0	1	
1	0	1	0	$(\overline{A} + B + \overline{C})$
1	1	0	0	$(\overline{A} + \overline{B} + C)$
1	1	1	1	

Notice that the truth table in this example is the same as the one in Example 4–20. This means that the SOP expression in the previous example and the POS expression in this example are equivalent.

Related Problem

Develop a truth table for the following standard POS expression:

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

Determining Standard Expressions from a Truth Table

To determine the standard SOP expression represented by a truth table, list the binary values of the input variables for which the output is 1. Convert each binary value to the corresponding product term by replacing each 1 with the corresponding variable and each 0 with the corresponding variable complement. For example, the binary value 1010 is converted to a product term as follows:

$$1010 \longrightarrow A\overline{B}C\overline{D}$$

If you substitute, you can see that the product term is 1:

$$A\overline{B}C\overline{D} = 1 \cdot \overline{0} \cdot 1 \cdot \overline{0} = 1 \cdot 1 \cdot 1 \cdot 1 = 1$$

To determine the standard POS expression represented by a truth table, list the binary values for which the output is 0. Convert each binary value to the corresponding sum term by replacing each 1 with the corresponding variable complement and each 0 with the corresponding variable. For example, the binary value 1001 is converted to a sum term as follows:

$$1001 \longrightarrow \overline{A} + B + C + \overline{D}$$

If you substitute, you can see that the sum term is 0:

$$\overline{A} + B + C + \overline{D} = \overline{1} + 0 + 0 + \overline{1} = 0 + 0 + 0 + 0 = 0$$

EXAMPLE 4–22

From the truth table in Table 4–8, determine the standard SOP expression and the equivalent standard POS expression.

TABLE 4–8

	Inputs		Output
A	***B***	***C***	***X***
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Solution

There are four 1s in the output column and the corresponding binary values are 011, 100, 110, and 111. Convert these binary values to product terms as follows:

$$
\begin{aligned}
011 &\longrightarrow \overline{A}BC \\
100 &\longrightarrow A\overline{B}\,\overline{C} \\
110 &\longrightarrow AB\overline{C} \\
111 &\longrightarrow ABC
\end{aligned}
$$

The resulting standard SOP expression for the output X is

$$X = \overline{A}BC + A\overline{B}\,\overline{C} + AB\overline{C} + ABC$$

For the POS expression, the output is 0 for binary values 000, 001, 010, and 101. Convert these binary values to sum terms as follows:

$$
\begin{aligned}
000 &\longrightarrow A + B + C \\
001 &\longrightarrow A + B + \overline{C} \\
010 &\longrightarrow A + \overline{B} + C \\
101 &\longrightarrow \overline{A} + B + \overline{C}
\end{aligned}
$$

The resulting standard POS expression for the output X is

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

Related Problem

By substitution of binary values, show that the SOP and the POS expressions derived in this example are equivalent; that is, for any binary value each SOP and POS term should either both be 1 or both be 0, depending on the binary value.

SECTION 4–7 CHECKUP

1. If a certain Boolean expression has a domain of five variables, how many binary values will be in its truth table?
2. In a certain truth table, the output is a 1 for the binary value 0110. Convert this binary value to the corresponding product term using variables W, X, Y, and Z.
3. In a certain truth table, the output is a 0 for the binary value 1100. Convert this binary value to the corresponding sum term using variables W, X, Y, and Z.

4–8 The Karnaugh Map

A Karnaugh map provides a systematic method for simplifying Boolean expressions and, if properly used, will produce the simplest SOP or POS expression possible, known as the minimum expression. As you have seen, the effectiveness of algebraic simplification depends on your familiarity with all the laws, rules, and theorems of Boolean algebra and on your ability to apply them. The Karnaugh map, on the other hand, provides a "cookbook" method for simplification. Other simplification techniques include the Quine-McCluskey method and the Espresso algorithm.

After completing this section, you should be able to

- Construct a Karnaugh map for three or four variables
- Determine the binary value of each cell in a Karnaugh map
- Determine the standard product term represented by each cell in a Karnaugh map
- Explain cell adjacency and identify adjacent cells

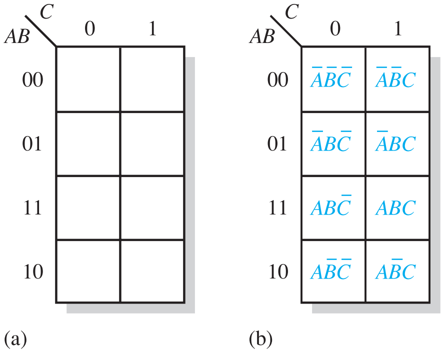
The purpose of a Karnaugh map is to simplify a Boolean expression.

A **Karnaugh map** is similar to a truth table because it presents all of the possible values of input variables and the resulting output for each value. Instead of being organized into columns and rows like a truth table, the Karnaugh map is an array of **cells** in which each cell represents a binary value of the input variables. The cells are arranged in a way so that simplification of a given expression is simply a matter of properly grouping the cells. Karnaugh maps can be used for expressions with two, three, four, and five variables, but we will discuss only 3-variable and 4-variable situations to illustrate the principles. *A discussion of 5-variable Karnaugh maps is available on the website*.

The number of cells in a Karnaugh map, as well as the number of rows in a truth table, is equal to the total number of possible input variable combinations. For three variables, the number of cells is $2^3 = 8$. For four variables, the number of cells is $2^4 = 16$.

The 3-Variable Karnaugh Map

The 3-variable Karnaugh map is an array of eight cells, as shown in Figure 4–25(a). In this case, A, B, and C are used for the variables although other letters could be used. Binary values of A and B are along the left side (notice the sequence) and the values of C are across the top. The value of a given cell is the binary values of A and B at the left in the same row combined with the value of C at the top in the same column. For example, the cell in the upper left corner has a binary value of 000 and the cell in the lower right corner has a binary value of 101. Figure 4–25(b) shows the standard product terms that are represented by each cell in the Karnaugh map.



$AB \backslash C$	0	1
00	$\overline{A}\overline{B}\overline{C}$	$\overline{A}\overline{B}C$
01	$\overline{A}B\overline{C}$	$\overline{A}BC$
11	$AB\overline{C}$	ABC
10	$A\overline{B}\overline{C}$	$A\overline{B}C$

(b)

FIGURE 4–25 A 3-variable Karnaugh map showing Boolean product terms for each cell.

The 4-Variable Karnaugh Map

The 4-variable Karnaugh map is an array of sixteen cells, as shown in Figure 4–26(a). Binary values of A and B are along the left side and the values of C and D are across the top. The value of a given cell is the binary values of A and B at the left in the same row combined with the binary values of C and D at the top in the same column. For example, the cell in the upper right corner has a binary value of 0010 and the cell in the lower right corner has a binary value of 1010. Figure 4–26(b) shows the standard product terms that are represented by each cell in the 4-variable Karnaugh map.

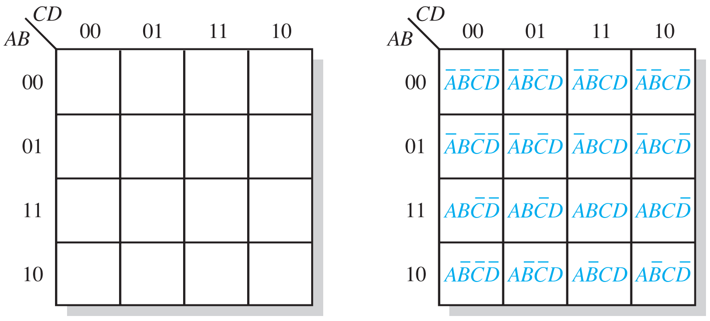
Cell Adjacency

Cells that differ by only one variable are adjacent.

Cells with values that differ by more than one variable are not adjacent.

The cells in a Karnaugh map are arranged so that there is only a single-variable change between adjacent cells. **Adjacency** is defined by a single-variable change. In the 3-variable map the 010 cell is adjacent to the 000 cell, the 011 cell, and the 110 cell. The 010 cell is not adjacent to the 001 cell, the 111 cell, the 100 cell, or the 101 cell.

Physically, each cell is adjacent to the cells that are immediately next to it on any of its four sides. A cell is not adjacent to the cells that diagonally touch any of its corners. Also, the cells in the top row are adjacent to the corresponding cells in the bottom row and



AB \ CD	00	01	11	10
00	$\overline{A}\overline{B}\overline{C}\overline{D}$	$\overline{A}\overline{B}\overline{C}D$	$\overline{A}\overline{B}CD$	$\overline{A}\overline{B}C\overline{D}$
01	$\overline{A}B\overline{C}\overline{D}$	$\overline{A}B\overline{C}D$	$\overline{A}BCD$	$\overline{A}BC\overline{D}$
11	$AB\overline{C}\overline{D}$	$AB\overline{C}D$	$ABCD$	$ABC\overline{D}$
10	$A\overline{B}\overline{C}\overline{D}$	$A\overline{B}\overline{C}D$	$A\overline{B}CD$	$A\overline{B}C\overline{D}$

(a) (b)

FIGURE 4–26 A 4-variable Karnaugh map.

the cells in the outer left column are adjacent to the corresponding cells in the outer right column. This is called "wrap-around" adjacency because you can think of the map as wrapping around from top to bottom to form a cylinder or from left to right to form a cylinder. Figure 4–27 illustrates the cell adjacencies with a 4-variable map, although the same rules for adjacency apply to Karnaugh maps with any number of cells.

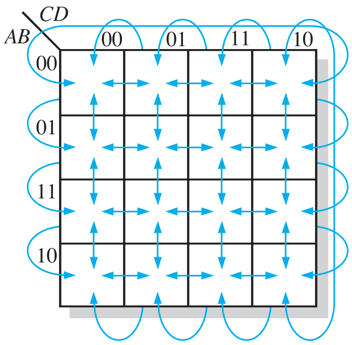


FIGURE 4–27 Adjacent cells on a Karnaugh map are those that differ by only one variable. Arrows point between adjacent cells.

The Quine-McCluskey Method

Minimizing Boolean functions using Karnaugh maps is practical only for up to four or five variables. Also, the Karnaugh map method does not lend itself to be automated in the form of a computer program.

The Quine-McCluskey method is more practical for logic simplification of functions with more than four or five variables. It also has the advantage of being easily implemented with a computer or programmable calculator.

The Quine-McCluskey method is functionally similar to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a way to check that the minimal form of a Boolean function has been reached. This method is sometimes referred to as the *tabulation method*. An introduction to the Quine-McCluskey method is provided in Section 4–11.

Espresso Algorithm

Although the Quine-McCluskey method is well suited to be implemented in a computer program and can handle more variables than the Karnaugh map method, the result is still far from efficient in terms of processing time and memory usage. Adding a variable to the function will roughly double both of these parameters because the truth table length increases exponentially with the number of variables. Functions with a large number of variables have to be minimized with other methods such as the Espresso logic minimizer, which has become the de facto world standard. *An Espresso algorithm tutorial is available on the website*.

Compared to the other methods, Espresso is essentially more efficient in terms of reducing memory usage and computation time by several orders of magnitude. There is essentially no restrictions to the number of variables, output functions, and product terms of a combinational logic function. In general, tens of variables with tens of output functions can be handled by Espresso.

The Espresso algorithm has been incorporated as a standard logic function minimization step in most logic synthesis tools for programmable logic devices. For implementing a function in multilevel logic, the minimization result is optimized by factorization and mapped onto the available basic logic cells in the target device, such as an FPGA (Field-Programmable Gate Array).

SECTION 4–8 CHECKUP

1. In a 3-variable Karnaugh map, what is the binary value for the cell in each of the following locations:

(a) upper left corner **(b)** lower right corner

(c) lower left corner **(d)** upper right corner

2. What is the standard product term for each cell in Question 1 for variables X, Y, and Z?

3. Repeat Question 1 for a 4-variable map.

4. Repeat Question 2 for a 4-variable map using variables W, X, Y, and Z.

4–9 Karnaugh Map SOP Minimization

As stated in the last section, the Karnaugh map is used for simplifying Boolean expressions to their minimum form. A minimized SOP expression contains the fewest possible terms with the fewest possible variables per term. Generally, a minimum SOP expression can be implemented with fewer logic gates than a standard expression. In this section, Karnaugh maps with up to four variables are covered.

After completing this section, you should be able to

- Map a standard SOP expression on a Karnaugh map
- Combine the 1s on the map into maximum groups
- Determine the minimum product term for each group on the map
- Combine the minimum product terms to form a minimum SOP expression
- Convert a truth table into a Karnaugh map for simplification of the represented expression
- Use “don’t care” conditions on a Karnaugh map

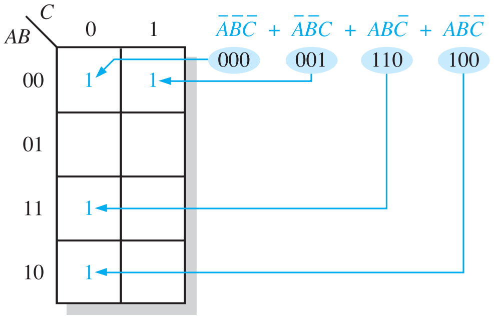
Mapping a Standard SOP Expression

For an SOP expression in standard form, a 1 is placed on the Karnaugh map for each product term in the expression. Each 1 is placed in a cell corresponding to the value of a product term. For example, for the product term $A\overline{B}C$, a 1 goes in the 101 cell on a 3-variable map.

When an SOP expression is completely mapped, there will be a number of 1s on the Karnaugh map equal to the number of product terms in the standard SOP expression. The cells that do not have a 1 are the cells for which the expression is 0. Usually, when working with SOP expressions, the 0s are left off the map. The following steps and the illustration in Figure 4–28 show the mapping process.

Step 1: Determine the binary value of each product term in the standard SOP expression. After some practice, you can usually do the evaluation of terms mentally.

Step 2: As each product term is evaluated, place a 1 on the Karnaugh map in the cell having the same value as the product term.



C
AB
0
1
$\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + AB\overline{C} + A\overline{B}\,\overline{C}$
000
001
110
100
00
01
11
10
1
1
1
1

FIGURE 4–28 Example of mapping a standard SOP expression.

EXAMPLE 4–23

Map the following standard SOP expression on a Karnaugh map:

$$\overline{A}\,\overline{B}C + \overline{A}B\overline{C} + AB\overline{C} + ABC$$

Solution

Evaluate the expression as shown below. Place a 1 on the 3-variable Karnaugh map in Figure 4–29 for each standard product term in the expression.

$$\begin{array}{cccc} \overline{A}\,\overline{B}C & +\ \overline{A}B\overline{C} & +\ AB\overline{C} & +\ ABC \\ 0\,0\,1 & 0\,1\,0 & 1\,1\,0 & 1\,1\,1 \end{array}$$

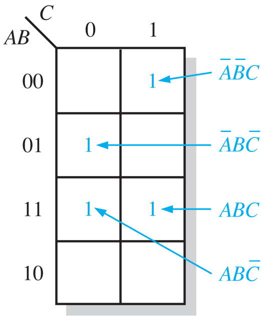


FIGURE 4–29

Related Problem

Map the standard SOP expression $\overline{A}BC + A\overline{B}C + A\overline{B}\,\overline{C}$ on a Karnaugh map.

EXAMPLE 4–24

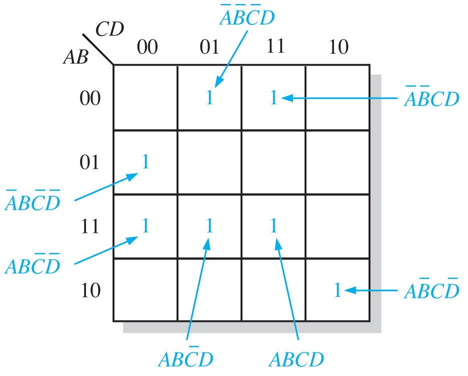
Map the following standard SOP expression on a Karnaugh map:

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

Solution

Evaluate the expression as shown below. Place a 1 on the 4-variable Karnaugh map in Figure 4–30 for each standard product term in the expression.

$$\underset{0011}{\overline{A}\,\overline{B}CD} + \underset{0100}{\overline{A}B\overline{C}\,\overline{D}} + \underset{1101}{AB\overline{C}D} + \underset{1111}{ABCD} + \underset{1100}{AB\overline{C}\,\overline{D}} + \underset{0001}{\overline{A}\,\overline{B}\,\overline{C}D} + \underset{1010}{A\overline{B}C\overline{D}}$$



$\overline{A}\,\overline{B}\,\overline{C}D$
CD
AB
00
01
11
10
00
01
11
10
1
1
1
1
1
1
1
$\overline{A}\,\overline{B}CD$
$\overline{A}B\overline{C}\,\overline{D}$
$AB\overline{C}\,\overline{D}$
$A\overline{B}C\overline{D}$
$AB\overline{C}D$
$ABCD$

FIGURE 4–30

Related Problem

Map the following standard SOP expression on a Karnaugh map:

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

Mapping a Nonstandard SOP Expression

A Boolean expression must first be in standard form before you use a Karnaugh map. If an expression is not in standard form, then it must be converted to standard form by the procedure covered in Section 4–6 or by numerical expansion. Since an expression should be evaluated before mapping anyway, numerical expansion is probably the most efficient approach.

Numerical Expansion of a Nonstandard Product Term

Recall that a nonstandard product term has one or more missing variables. For example, assume that one of the product terms in a certain 3-variable SOP expression is $A\overline{B}$. This term can be expanded numerically to standard form as follows. First, write the binary value of the two variables and attach a 0 for the missing variable $\overline{C}$: 100. Next, write the binary value of the two variables and attach a 1 for the missing variable C: 101. The two resulting binary numbers are the values of the standard SOP terms $A\overline{B}\,\overline{C}$ and $A\overline{B}C$.

As another example, assume that one of the product terms in a 3-variable expression is B (remember that a single variable counts as a product term in an SOP expression). This term can be expanded numerically to standard form as follows. Write the binary value of the variable; then attach all possible values for the missing variables A and C as follows:

B
010
011
110
111

The four resulting binary numbers are the values of the standard SOP terms $\overline{A}B\overline{C}$, $\overline{A}BC$, $AB\overline{C}$, and ABC.

EXAMPLE 4–25

Map the following SOP expression on a Karnaugh map: $\overline{A} + A\overline{B} + AB\overline{C}$.

Solution

The SOP expression is obviously not in standard form because each product term does not have three variables. The first term is missing two variables, the second term is missing one variable, and the third term is standard. First expand the terms numerically as follows:

$$\begin{array}{lll} \overline{A} & + A\overline{B} & + AB\overline{C} \\ 000 & 100 & 110 \\ 001 & 101 & \\ 010 & & \\ 011 & & \end{array}$$

Map each of the resulting binary values by placing a 1 in the appropriate cell of the 3-variable Karnaugh map in Figure 4–31.

AB \ C	0	1
00	1	1
01	1	1
11	1	
10	1	1

FIGURE 4–31

Related Problem

Map the SOP expression $BC + \overline{A}\,\overline{C}$ on a Karnaugh map.

EXAMPLE 4–26

Map the following SOP expression on a Karnaugh map:

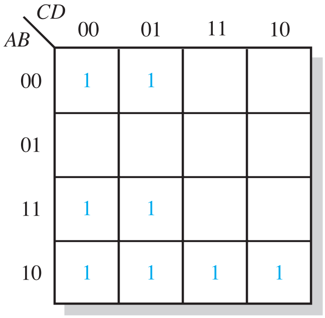
$$\overline{B}\,\overline{C} + A\overline{B} + AB\overline{C} + A\overline{B}C\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + A\overline{B}CD$$

Solution

The SOP expression is obviously not in standard form because each product term does not have four variables. The first and second terms are both missing two variables, the third term is missing one variable, and the rest of the terms are standard. First expand the terms by including all combinations of the missing variables numerically as follows:

$$\begin{array}{cccccccccccc} \overline{B}\,\overline{C} & + & A\overline{B} & + & AB\overline{C} & + & A\overline{B}C\overline{D} & + & \overline{A}\,\overline{B}\,\overline{C}D & + & A\overline{B}CD \\ 0000 & & 1000 & & 1100 & & 1010 & & 0001 & & 1011 \\ 0001 & & 1001 & & 1101 & & & & & & \\ 1000 & & 1010 & & & & & & & & \\ 1001 & & 1011 & & & & & & & & \end{array}$$

Map each of the resulting binary values by placing a 1 in the appropriate cell of the 4-variable Karnaugh map in Figure 4–32. Notice that some of the values in the expanded expression are redundant.



AB \ CD	00	01	11	10
00	1	1		
01				
11	1	1		
10	1	1	1	1

FIGURE 4–32

Related Problem

Map the expression $A + \overline{C}D + AC\overline{D} + \overline{A}BC\overline{D}$ on a Karnaugh map.

Karnaugh Map Simplification of SOP Expressions

The process that results in an expression containing the fewest possible terms with the fewest possible variables is called **minimization**. After an SOP expression has been mapped, a minimum SOP expression is obtained by grouping the 1s and determining the minimum SOP expression from the map.

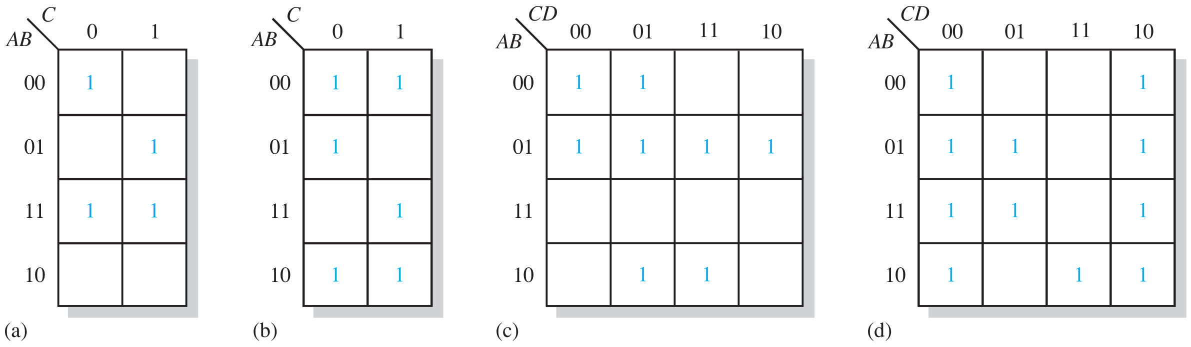
Grouping the 1s

You can group 1s on the Karnaugh map according to the following rules by enclosing those adjacent cells containing 1s. The goal is to maximize the size of the groups and to minimize the number of groups.

1. A group must contain either 1, 2, 4, 8, or 16 cells, which are all powers of two. In the case of a 3-variable map, $2^3 = 8$ cells is the maximum group.
2. Each cell in a group must be adjacent to one or more cells in that same group, but all cells in the group do not have to be adjacent to each other.
3. Always include the largest possible number of 1s in a group in accordance with rule 1.
4. Each 1 on the map must be included in at least one group. The 1s already in a group can be included in another group as long as the overlapping groups include noncommon 1s.

EXAMPLE 4–27

Group the 1s in each of the Karnaugh maps in Figure 4–33.



(a)

AB \ C	0	1
00	1	
01		1
11	1	1
10		

(b)

AB \ C	0	1
00	1	1
01	1	
11		1
10	1	1

(c)

AB \ CD	00	01	11	10
00	1	1		
01	1	1	1	1
11				
10		1	1	

(d)

AB \ CD	00	01	11	10
00	1			1
01	1	1		1
11	1	1		1
10	1		1	1

FIGURE 4–33

Solution

The groupings are shown in Figure 4–34. In some cases, there may be more than one way to group the 1s to form maximum groupings.

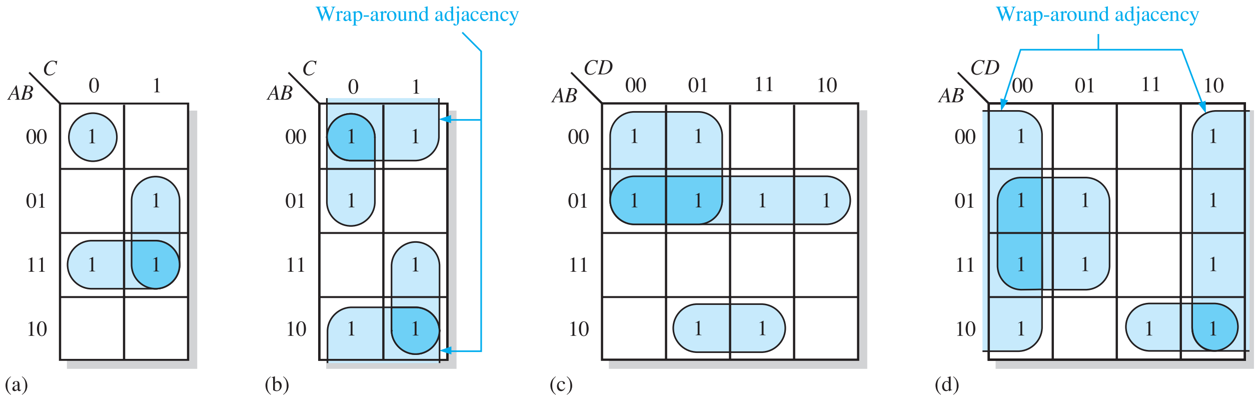


FIGURE 4–34

Related Problem

Determine if there are other ways to group the 1s in Figure 4–34 to obtain a minimum number of maximum groupings.

Determining the Minimum SOP Expression from the Map

When all the 1s representing the standard product terms in an expression are properly mapped and grouped, the process of determining the resulting minimum SOP expression begins. The following rules are applied to find the minimum product terms and the minimum SOP expression:

1. Group the cells that have 1s. Each group of cells containing 1s creates one product term composed of all variables that occur in only one form (either uncomplemented or complemented) within the group. Variables that occur both uncomplemented and complemented within the group are eliminated. These are called *contradictory variables.*

2. Determine the minimum product term for each group.
 (a) For a 3-variable map:
  (1) A 1-cell group yields a 3-variable product term
  (2) A 2-cell group yields a 2-variable product term
  (3) A 4-cell group yields a 1-variable term
  (4) An 8-cell group yields a value of 1 for the expression
 (b) For a 4-variable map:
  (1) A 1-cell group yields a 4-variable product term
  (2) A 2-cell group yields a 3-variable product term
  (3) A 4-cell group yields a 2-variable product term
  (4) An 8-cell group yields a 1-variable term
  (5) A 16-cell group yields a value of 1 for the expression

3. When all the minimum product terms are derived from the Karnaugh map, they are summed to form the minimum SOP expression.

EXAMPLE 4–28

Determine the product terms for the Karnaugh map in Figure 4–35 and write the resulting minimum SOP expression.

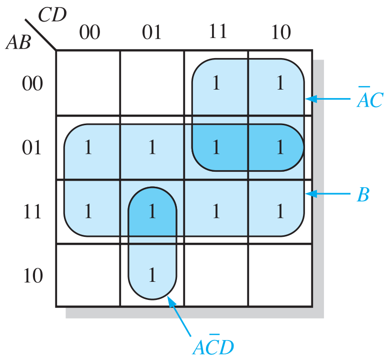


FIGURE 4–35

Solution

Eliminate variables that are in a grouping in both complemented and uncomplemented forms. In Figure 4–35, the product term for the 8-cell group is B because the cells within that group contain both A and $\overline{A}$, C and $\overline{C}$, and D and $\overline{D}$, which are eliminated. The 4-cell group contains B, $\overline{B}$, D, and $\overline{D}$, leaving the variables $\overline{A}$ and C, which form the product term $\overline{A}C$. The 2-cell group contains B and $\overline{B}$, leaving variables A, $\overline{C}$, and D which form the product term $A\overline{C}D$. Notice how overlapping is used to maximize the size of the groups. The resulting minimum SOP expression is the sum of these product terms:

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

Related Problem

For the Karnaugh map in Figure 4–35, add a 1 in the lower right cell (1010) and determine the resulting SOP expression.

EXAMPLE 4–29

Determine the product terms for each of the Karnaugh maps in Figure 4–36 and write the resulting minimum SOP expression.

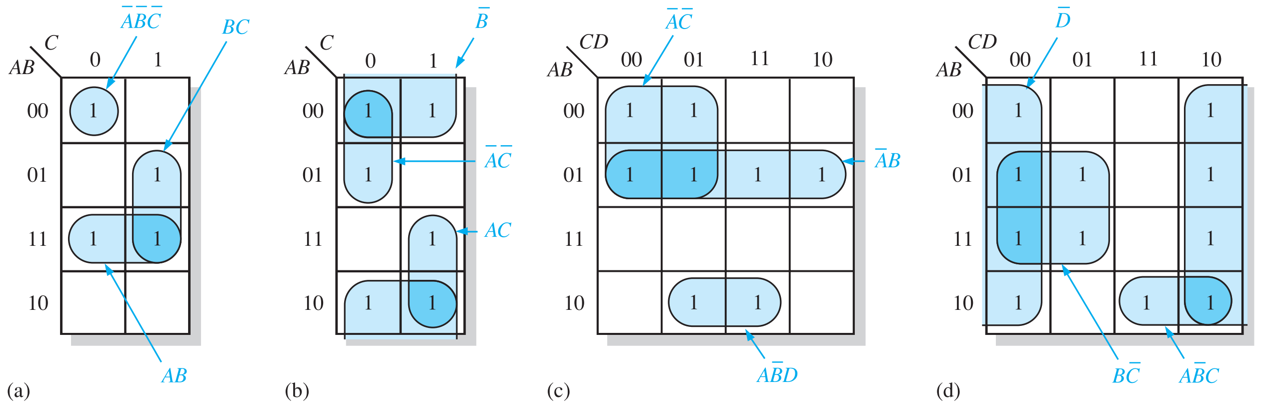


FIGURE 4–36

Solution

The resulting minimum product term for each group is shown in Figure 4–36. The minimum SOP expressions for each of the Karnaugh maps in the figure are

(a) $AB + BC + \overline{A}\,\overline{B}\,\overline{C}$

(b) $\overline{B} + \overline{A}\,\overline{C} + AC$

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

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

Related Problem

For the Karnaugh map in Figure 4–36(d), add a 1 in the 0111 cell and determine the resulting SOP expression.

EXAMPLE 4–30

Use a Karnaugh map to minimize the following standard SOP expression:

$$A\overline{B}C + \overline{A}BC + \overline{A}\,\overline{B}C + \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C}$$

Solution

The binary values of the expression are

$$101 + 011 + 001 + 000 + 100$$

Map the standard SOP expression and group the cells as shown in Figure 4–37.

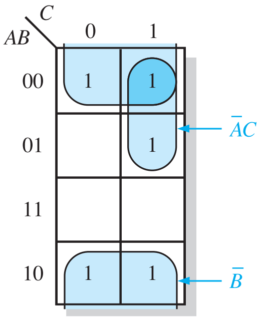


FIGURE 4–37

Notice the "wrap around" 4-cell group that includes the top row and the bottom row of 1s. The remaining 1 is absorbed in an overlapping group of two cells. The group of four 1s produces a single variable term, $\overline{B}$. This is determined by observing that within the group, $\overline{B}$ is the only variable that does not change from cell to cell. The group of two 1s produces a 2-variable term $\overline{A}C$. This is determined by observing that within the group, $\overline{A}$ and C do not change from one cell to the next. The product term for each group is shown. The resulting minimum SOP expression is

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

Keep in mind that this minimum expression is equivalent to the original standard expression.

Related Problem

Use a Karnaugh map to simplify the following standard SOP expression:

$$X\overline{Y}Z + XY\overline{Z} + \overline{X}YZ + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$$

EXAMPLE 4–31

Use a Karnaugh map to minimize the following SOP expression:

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

Solution

The first term $\overline{B}\,\overline{C}\,\overline{D}$ must be expanded into $A\overline{B}\,\overline{C}\,\overline{D}$ and $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ to get the standard SOP expression, which is then mapped; the cells are grouped as shown in Figure 4–38.

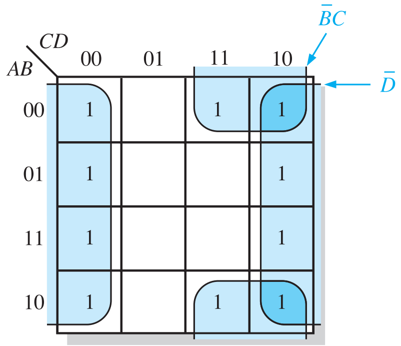


FIGURE 4–38

Notice that both groups exhibit "wrap around" adjacency. The group of eight is formed because the cells in the outer columns are adjacent. The group of four is formed to pick up the remaining two 1s because the top and bottom cells are adjacent. The product term for each group is shown. The resulting minimum SOP expression is

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

Keep in mind that this minimum expression is equivalent to the original standard expression.

Related Problem

Use a Karnaugh map to simplify the following SOP expression:

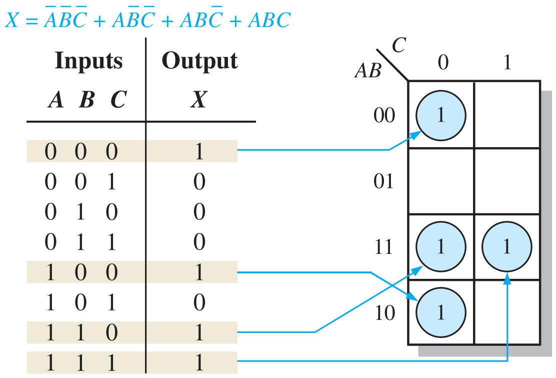
$$\overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}YZ + W\overline{X}\,\overline{Y}Z + \overline{W}YZ + W\overline{X}\,\overline{Y}\,\overline{Z}$$

Mapping Directly from a Truth Table

You have seen how to map a Boolean expression; now you will learn how to go directly from a truth table to a Karnaugh map. Recall that a truth table gives the output of a Boolean expression for all possible input variable combinations. An example of a Boolean expression and its truth table representation is shown in Figure 4–39. Notice in the truth table that the output X is 1 for four different input variable combinations. The 1s in the output column of the truth table are mapped directly onto a Karnaugh map into the cells corresponding to the values of the associated input variable combinations, as shown in Figure 4–39. In the figure you can see that the Boolean expression, the truth table, and the Karnaugh map are simply different ways to represent a logic function.

"Don't Care" Conditions

Sometimes a situation arises in which some input variable combinations are not allowed. For example, recall that in the BCD code covered in Chapter 2, there are six invalid combinations: 1010, 1011, 1100, 1101, 1110, and 1111. Since these unallowed states



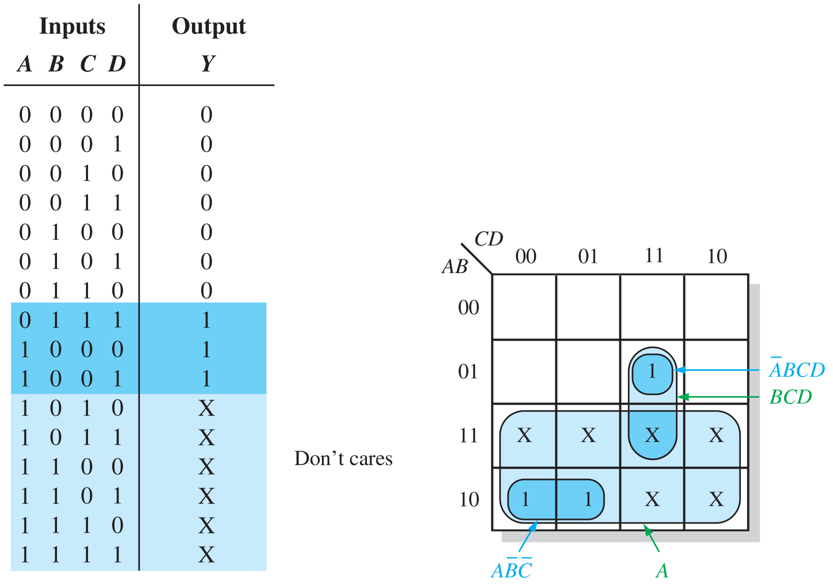
$X = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C} + ABC$

Inputs			Output
A	*B*	*C*	*X*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

FIGURE 4–39 Example of mapping directly from a truth table to a Karnaugh map.

will never occur in an application involving the BCD code, they can be treated as **"don't care"** terms with respect to their effect on the output. That is, for these "don't care" terms either a 1 or a 0 may be assigned to the output; it really does not matter since they will never occur.

The "don't care" terms can be used to advantage on the Karnaugh map. Figure 4–40 shows that for each "don't care" term, an X is placed in the cell. When grouping the 1s, the Xs can be treated as 1s to make a larger grouping or as 0s if they cannot be used to advantage. The larger a group, the simpler the resulting term will be.



Inputs				Output
A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

(a) Truth table

(b) Without "don't cares" $Y = A\overline{B}\,\overline{C} + \overline{A}BCD$
With "don't cares" $Y = A + BCD$

FIGURE 4–40 Example of the use of "don't care" conditions to simplify an expression.

The truth table in Figure 4–40(a) describes a logic function that has a 1 output only when the BCD code for 7, 8, or 9 is present on the inputs. If the "don't cares" are used as 1s, the resulting expression for the function is $A + BCD$, as indicated in part (b). If the "don't cares" are not used as 1s, the resulting expression is $A\overline{B}\,\overline{C} + \overline{A}BCD$; so you can see the advantage of using "don't care" terms to get the simplest expression.

EXAMPLE 4–32

In a 7-segment display, each of the seven segments is activated for various digits. For example, segment *a* is activated for the digits 0, 2, 3, 5, 6, 7, 8, and 9, as illustrated in Figure 4–41. Since each digit can be represented by a BCD code, derive an SOP expression for segment *a* using the variables *ABCD* and then minimize the expression using a Karnaugh map.

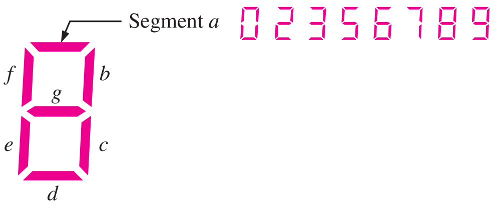


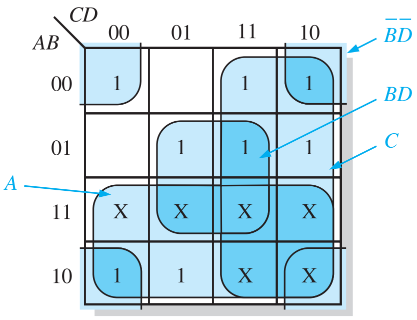
FIGURE 4–41 7-segment display.

Solution

The expression for segment *a* is

$$a = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}C\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D$$

Each term in the expression represents one of the digits in which segment *a* is used. The Karnaugh map minimization is shown in Figure 4–42. X's (don't cares) are entered for those states that do not occur in the BCD code.



AB \ CD	00	01	11	10
00	1		1	1
01		1	1	1
11	X	X	X	X
10	1	1	X	X

FIGURE 4–42

From the Karnaugh map, the minimized expression for segment *a* is

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

Related Problem

Draw the logic diagram for the segment-*a* logic.

SECTION 4–9 CHECKUP

1. Lay out Karnaugh maps for three and four variables.
2. Group the 1s and write the simplified SOP expression for the Karnaugh map in Figure 4–29.
3. Write the original standard SOP expressions for each of the Karnaugh maps in Figure 4–36.

4–10 Karnaugh Map POS Minimization

In the last section, you studied the minimization of an SOP expression using a Karnaugh map. In this section, we focus on POS expressions. The approaches are much the same except that with POS expressions, 0s representing the standard sum terms are placed on the Karnaugh map instead of 1s.

After completing this section, you should be able to

- Map a standard POS expression on a Karnaugh map
- Combine the 0s on the map into maximum groups
- Determine the minimum sum term for each group on the map
- Combine the minimum sum terms to form a minimum POS expression
- Use the Karnaugh map to convert between POS and SOP

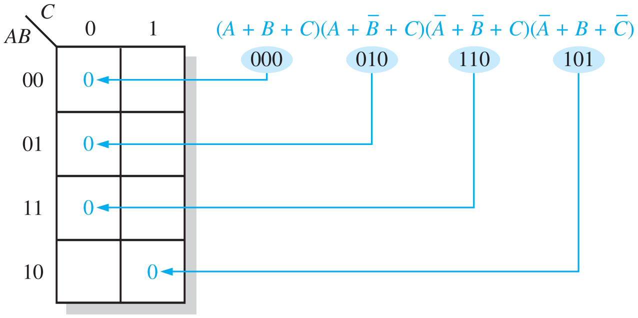
Mapping a Standard POS Expression

For a POS expression in standard form, a 0 is placed on the Karnaugh map for each sum term in the expression. Each 0 is placed in a cell corresponding to the value of a sum term. For example, for the sum term $A + \overline{B} + C$, a 0 goes in the 010 cell on a 3-variable map.

When a POS expression is completely mapped, there will be a number of 0s on the Karnaugh map equal to the number of sum terms in the standard POS expression. The cells that do not have a 0 are the cells for which the expression is 1. Usually, when working with POS expressions, the 1s are left off. The following steps and the illustration in Figure 4–43 show the mapping process.

Step 1: Determine the binary value of each sum term in the standard POS expression. This is the binary value that makes the term equal to 0.

Step 2: As each sum term is evaluated, place a 0 on the Karnaugh map in the corresponding cell.



C
AB
0
1
00
01
11
10
0
0
0
0
$(A + B + C)(A + \overline{B} + C)(\overline{A} + \overline{B} + C)(\overline{A} + B + \overline{C})$
000
010
110
101

FIGURE 4–43 Example of mapping a standard POS expression.

EXAMPLE 4–33

Map the following standard POS expression on a Karnaugh map:

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

Solution

Evaluate the expression as shown below and place a 0 on the 4-variable Karnaugh map in Figure 4–44 for each standard sum term in the expression.

$$\underset{1100}{(\overline{A} + \overline{B} + C + D)}\underset{1011}{(\overline{A} + B + \overline{C} + \overline{D})}\underset{0010}{(A + B + \overline{C} + D)}\underset{1111}{(\overline{A} + \overline{B} + \overline{C} + \overline{D})}\underset{0011}{(A + B + \overline{C} + \overline{D})}$$

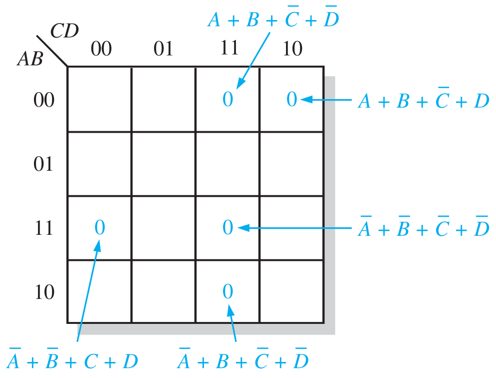

$A + B + \overline{C} + \overline{D}$
CD
AB
00
01
11
10
00
01
11
10
0
0
0
0
0
$A + B + \overline{C} + D$
$\overline{A} + \overline{B} + \overline{C} + \overline{D}$
$\overline{A} + \overline{B} + C + D$
$\overline{A} + B + \overline{C} + \overline{D}$

FIGURE 4–44

Related Problem

Map the following standard POS expression on a Karnaugh map:

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

Karnaugh Map Simplification of POS Expressions

The process for minimizing a POS expression is basically the same as for an SOP expression except that you group 0s to produce minimum sum terms instead of grouping 1s to produce minimum product terms. The rules for grouping the 0s are the same as those for grouping the 1s that you learned in Section 4–9.

EXAMPLE 4–34

Use a Karnaugh map to minimize the following standard POS expression:

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

Also, derive the equivalent SOP expression.

Solution

The combinations of binary values of the expression are

$$(0 + 0 + 0)(0 + 0 + 1)(0 + 1 + 0)(0 + 1 + 1)(1 + 1 + 0)$$

Map the standard POS expression and group the cells as shown in Figure 4–45.

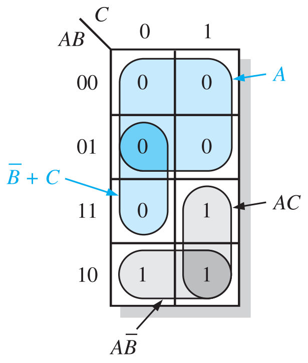


FIGURE 4–45

Notice how the 0 in the 110 cell is included into a 2-cell group by utilizing the 0 in the 4-cell group. The sum term for each blue group is shown in the figure and the resulting minimum POS expression is

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

Keep in mind that this minimum POS expression is equivalent to the original standard POS expression.

Grouping the 1s as shown by the gray areas yields an SOP expression that is equivalent to grouping the 0s.

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

Related Problem

Use a Karnaugh map to simplify the following standard POS expression:

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

EXAMPLE 4–35

Use a Karnaugh map to minimize the following POS expression:

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

Solution

The first term must be expanded into $\overline{A} + B + C + D$ and $A + B + C + D$ to get a standard POS expression, which is then mapped; and the cells are grouped as shown in Figure 4–46. The sum term for each group is shown and the resulting minimum POS expression is

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

Keep in mind that this minimum POS expression is equivalent to the original standard POS expression.

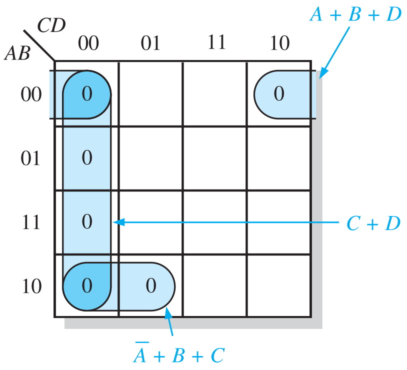


FIGURE 4–46

Related Problem

Use a Karnaugh map to simplify the following POS expression:

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

Converting Between POS and SOP Using the Karnaugh Map

When a POS expression is mapped, it can easily be converted to the equivalent SOP form directly from the Karnaugh map. Also, given a mapped SOP expression, an equivalent POS expression can be derived directly from the map. This provides a good way to compare both minimum forms of an expression to determine if one of them can be implemented with fewer gates than the other.

For a POS expression, all the cells that do not contain 0s contain 1s, from which the SOP expression is derived. Likewise, for an SOP expression, all the cells that do not contain 1s contain 0s, from which the POS expression is derived. Example 4–36 illustrates this conversion.

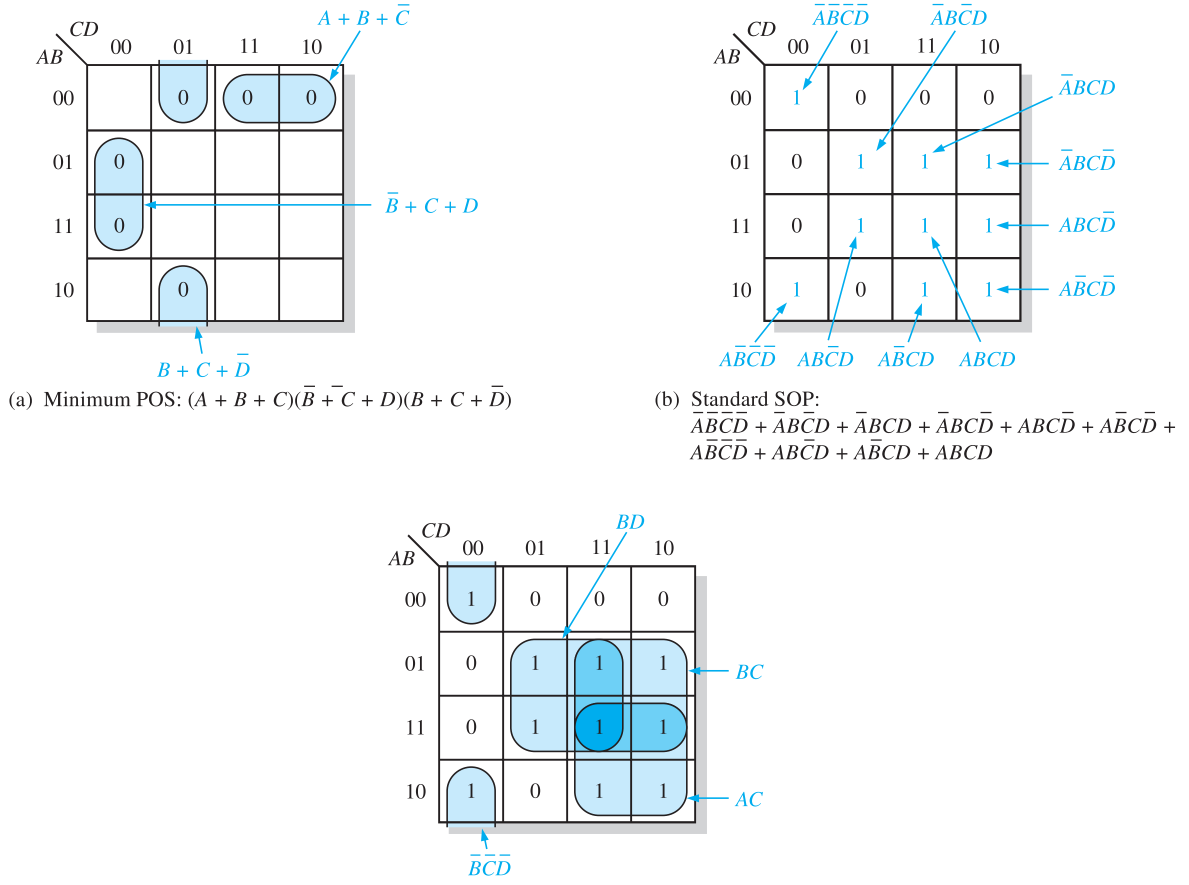
EXAMPLE 4–36

Using a Karnaugh map, convert the following standard POS expression into a minimum POS expression, a standard SOP expression, and a minimum SOP expression.

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

Solution

The 0s for the standard POS expression are mapped and grouped to obtain the minimum POS expression in Figure 4–47(a). In Figure 4–47(b), 1s are added to the cells that do not contain 0s. From each cell containing a 1, a standard product term is obtained as indicated. These product terms form the standard SOP expression. In Figure 4–47(c), the 1s are grouped and a minimum SOP expression is obtained.


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

AB \ CD	00	01	11	10
00		0	0	0
01	0			
11	0			
10		0		

(a) Minimum POS: $(A + B + C)(\overline{B} + \overline{}C + D)(B + C + \overline{D})$

AB \ CD	00	01	11	10
00	1	0	0	0
01	0	1	1	1
11	0	1	1	1
10	1	0	1	1

(b) Standard SOP:
$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD + \overline{A}BC\overline{D} + ABC\overline{D} + A\overline{B}C\overline{D} +$
$A\overline{B}\,\overline{C}\,\overline{D} + AB\overline{C}D + A\overline{B}CD + ABCD$

AB \ CD	00	01	11	10
00	1	0	0	0
01	0	1	1	1
11	0	1	1	1
10	1	0	1	1

(c) Minimum SOP: $AC + BC + BD + \overline{B}\,\overline{C}\,\overline{D}$

FIGURE 4–47

Related Problem

Use a Karnaugh map to convert the following expression to minimum SOP form:

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

SECTION 4–10 CHECKUP

1. What is the difference in mapping a POS expression and an SOP expression?
2. What is the standard sum term for a 0 in cell 1011?
3. What is the standard product term for a 1 in cell 0010?

4–11 The Quine-McCluskey Method

For Boolean functions up to four variables, the Karnaugh map method is a powerful minimization method. When there are five variables, the Karnaugh map method is difficult to apply and completely impractical beyond five. The Quine-McCluskey method is a formal tabular method for applying the Boolean distributive law to various terms to find the minimum sum of products by eliminating literals that appear in two terms as complements. (For example, $ABCD + ABC\overline{D} = ABC$). *A Quine-McCluskey method tutorial is available on the website.*

After completing this section, you should be able to

- Describe the Quine-McCluskey method
- Reduce a Boolean expression using the Quine-McCluskey method

Unlike the Karnaugh mapping method, Quine-McCluskey lends itself to the computerized reduction of Boolean expressions, which is its principal use. For simple expressions, with up to four or perhaps even five variables, the Karnaugh map is easier for most people because it is a graphic method.

To apply the Quine-McCluskey method, first write the function in standard **minterm** (SOP) form. To illustrate, we will use the expression

$$X = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + A\overline{B}C\overline{D} + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD$$

and represent it as binary numbers on the truth table shown in Table 4–9. The minterms that appear in the function are listed in the right column.

TABLE 4–9

ABCD	*X*	Minterm
0000	0	
0001	1	m_1
0010	0	
0011	1	m_3
0100	1	m_4
0101	1	m_5
0110	0	
0111	0	
1000	0	
1001	0	
1010	1	m_{10}
1011	0	
1100	1	m_{12}
1101	1	m_{13}
1110	0	
1111	1	m_{15}

The second step in applying the Quine-McCluskey method is to arrange the minterms in the original expression in groups according to the number of 1s in each minterm, as shown in Table 4–10. In this example, there are four groups of minterms. (Note that if m_0 had been in the original expression, there would be five groups.)

TABLE 4–10

Number of 1s	Minterm	*ABCD*
1	m_1	0001
	m_4	0100
2	m_3	0011
	m_5	0101
	m_{10}	1010
	m_{12}	1100
3	m_{13}	1101
4	m_{15}	1111

Third, compare adjacent groups, looking to see if any minterms are the same in every position *except one.* If they are, place a check mark by those two minterms, as shown in Table 4–11. You should check each minterm against all others in the following group, but it is not necessary to check any groups that are not adjacent. In the column labeled *First Level,* you will have a list of the minterm names and the binary equivalent with an x as the placeholder for the literal that differs. In the example, minterm m_1 in Group 1 (0001) is identical to m_3 in Group 2 (0011) except for the *C* position, so place a check mark by these two minterms and enter 00x1 in the column labeled *First Level*. Minterm m_4 (0100) is identical to m_5 (0101) except for the *D* position, so check these two minterms and enter 010x in the last column. If a given term can be used more than once, it should be. In this case, notice that m_1 can be used again with m_5 in the second row with the x now placed in the *B* position.

TABLE 4–11

Number of 1s in Minterm	Minterm	ABCD	First Level
1	m_1	0001 ✓	(m_1, m_3) 00x1
	m_4	0100 ✓	(m_1, m_5) 0x01
2	m_3	0011 ✓	(m_4, m_5) 010x
	m_5	0101 ✓	(m_4, m_{12}) x100
	m_{10}	1010	(m_5, m_{13}) x101
	m_{12}	1100 ✓	(m_{12}, m_{13}) 110x
3	m_{13}	1101 ✓	(m_{13}, m_{15}) 11x1
4	m_{15}	1111 ✓	

In Table 4–11, minterm m_4 and minterm m_{12} are identical except for the *A* position. Both minterms are checked and x100 is entered in the *First Level* column . Follow this procedure for groups 2 and 3. In these groups, m_5 and m_{13} are combined and so are m_{12} and m_{13} (notice that m_{12} was previously used with m_4 and is used again). For groups 3 and 4, both m_{13} and m_{15} are added to the list in the *First Level* column .

In this example, minterm m_{10} does not have a check mark because no other minterm meets the requirement of being identical except for one position. This term is called an *essential prime implicant,* and it must be included in our final reduced expression.

The terms listed in the *First Level* have been used to form a reduced table (Table 4–12) with one less group than before. The number of 1s remaining in the *First Level* are counted and used to form three new groups.

Terms in the new groups are compared against terms in the adjacent group down. You need to compare these terms only if the x is in the same relative position in adjacent groups; otherwise go on. If the two expressions differ by exactly one position, a check mark is

TABLE 4–12

First Level	Number of 1s in First Level	Second Level
(m_1, m_3) 00x1	1	$(m_4, m_5, m_{12}, m_{13})$ x10x
(m_1, m_5) 0x01		$(m_4, m_5, m_{12}, m_{13})$ x10x
(m_4, m_5) 010x ✓		
(m_4, m_{12}) x100 ✓		
(m_5, m_{13}) x101 ✓	2	
(m_{12}, m_{13}) 110x ✓		
(m_{13}, m_{15}) 11x1	3	

placed next to both terms as before and all of the minterms are listed in the Second Level list. As before, the one position that has changed is entered as an x in the *Second Level.*

For our example, notice that the third term in Group 1 and the second term in Group 2 meet this requirement, differing only with the *A* literal. The fourth term in Group 1 also can be combined with the first term in Group 2, forming a redundant set of minterms. One of these can be crossed off the list and will not be used in the final expression.

With complicated expressions, the process described can be continued. For our example, we can read the *Second Level* expression as $B\overline{C}$. The terms that are unchecked will form other terms in the final reduced expression. The first unchecked term is read as $\overline{A}\,\overline{B}D$. The next one is read as $\overline{A}\,\overline{C}D$. The last unchecked term is *ABD*. Recall that m_{10} was an essential prime implicant, so is picked up in the final expression. The reduced expression using the unchecked terms is:

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

Although this expression is correct, it may not be the minimum possible expression. There is a final check that can eliminate any unnecessary terms. The terms for the expression are written into a prime implicant table, with minterms for each prime implicant checked, as shown in Table 4–13.

TABLE 4–13

	Minterms							
Prime Implicants	m_1	m_3	m_4	m_5	m_{10}	m_{12}	m_{13}	m_{15}
$B\overline{C}$ $(m_4, m_5, m_{12}, m_{13})$			✓	✓		✓	✓	
$\overline{A}\,\overline{B}\,D$ (m_1, m_3)	✓	✓						
$\overline{A}\,\overline{C}D$ (m_1, m_5)	✓			✓				
ABD (m_{13}, m_{15})							✓	✓
$A\overline{B}C\overline{D}$ (m_{10})					✓			

If a minterm has a single check mark, then the prime implicant is essential and must be included in the final expression. The term *ABD* must be included because m_{15} is only covered by it. Likewise m_{10} is only covered by $A\overline{B}C\overline{D}$, so it must be in the final expression. Notice that the two minterms in $\overline{A}\,\overline{C}D$ are covered by the prime implicants in the first two rows, so this term is unnecessary. The final reduced expression is, therefore,

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

SECTION 4–11 CHECKUP

1. What is a minterm?
2. What is an essential prime implicant?

4–12 Boolean Expressions with VHDL

The ability to create simple and compact code is important in a VHDL program. By simplifying a Boolean expression for a given logic function, it is easier to write and debug the VHDL code; in addition, the result is a clearer and more concise program. Many VHDL development software packages contain tools that automatically optimize a program when it is compiled and converted to a downloadable file. However, this does not relieve you from creating program code that is clear and concise. You should not only be concerned with the number of lines of code, but you should also be concerned with the complexity of each line of code. In this section, you will see the difference in VHDL code when simplification methods are applied. Also, three levels of abstraction used in the description of a logic function are examined. *A VHDL tutorial is available on the website.*

After completing this section, you should be able to

- Write VHDL code to represent a simplified logic expression and compare it to the code for the original expression
- Relate the advantages of optimized Boolean expressions as applied to a target device
- Understand how a logic function can be described at three levels of abstraction
- Relate VHDL approaches to the description of a logic function to the three levels of abstraction

Boolean Algebra in VHDL Programming

The basic rules of Boolean algebra that you have learned in this chapter should be applied to any applicable VHDL code. Eliminating unnecessary gate logic allows you to create compact code that is easier to understand, especially when someone has to go back later and update or modify the program.

In Example 4–37, DeMorgan's theorems are used to simplify a Boolean expression, and VHDL programs for both the original expression and the simplified expression are compared.

EXAMPLE 4–37

First, write a VHDL program for the logic described by the following Boolean expression. Next, apply DeMorgan's theorems and Boolean rules to simplify the expression. Then write a program to reflect the simplified expression.

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

Solution

The VHDL program for the logic represented by the original expression is

```
entity OriginalLogic is
  port (A, B, C, D: in bit; X: out bit);
end entity OriginalLogic;
architecture Expression1 of OriginalLogic is
begin
  X <= not((A and C) or not(B and not C) or D)  or not(not(B and C));
end architecture Expression1;
```

Four inputs and one output are described.

The original logic contains four inputs, 3 AND gates, 2 OR gates, and 3 inverters.

By selectively applying DeMorgan's theorem and the laws of Boolean algebra, you can reduce the Boolean expression to its simplest form.

$$\begin{aligned}
\overline{(AC + \overline{B\overline{C}} + D)} + \overline{\overline{BC}} &= (\overline{AC})(\overline{\overline{B\overline{C}}})\overline{D} + \overline{\overline{BC}} && \text{Apply DeMorgan}\\
&= (\overline{AC})(B\overline{C})\overline{D} + BC && \text{Cancel double complements}\\
&= (\overline{A} + \overline{C})B\overline{C}\,\overline{D} + BC && \text{Apply DeMorgan and factor}\\
&= \overline{A}B\overline{C}\,\overline{D} + B\overline{C}\,\overline{D} + BC && \text{Distributive law}\\
&= B\overline{C}\,\overline{D}(1 + \overline{A}) + BC && \text{Factor}\\
&= B\overline{C}\,\overline{D} + BC && \text{Rule: } 1 + A = 1
\end{aligned}$$

The VHDL program for the logic represented by the reduced expression is

```
entity ReducedLogic is
   port (B, C, D: in bit; X: out bit);
end entity ReducedLogic;
architecture Expression2 of ReducedLogic is
begin
   X <= (B and not C and not D) or ( B and C);
end architecture Expression2;
```

3 inputs and 1 output are described.

The simplified logic contains three inputs, 3 AND gates, 1 OR gate, and 2 inverters.

As you can see, Boolean simplification is applicable to even simple VHDL programs.

Related Problem

Write the VHDL architecture statement for the expression $X = (\overline{A} + B + C)D$ as stated. Apply any applicable Boolean rules and rewrite the VHDL statement.

Example 4–38 demonstrates a more significant reduction in VHDL code complexity, using a Karnaugh map to reduce an expression.

EXAMPLE 4–38

(a) Write a VHDL program to describe the following SOP expression.

(b) Minimize the expression and show how much the VHDL program is simplified.

$$\begin{aligned}
X = {} & \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}B\overline{C}\,\overline{D} + \overline{A}BC\overline{D} + \overline{A}\,\overline{B}C\overline{D} + A\overline{B}\,\overline{C}\,\overline{D}\\
& + A\overline{B}C\overline{D} + ABC\overline{D} + AB\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}D + \overline{A}B\overline{C}D + AB\overline{C}D
\end{aligned}$$

Solution

(a) The VHDL program for the SOP expression without minimization is large and hard to follow as you can see in the following VHDL code. Code such as this is subject to error. The VHDL program for the original SOP expression is as follows:

```
entity OriginalSOP is
   port (A, B, C, D: in bit; X: out bit);
end entity OriginalSOP;
architecture Equation1 of OriginalSOP is
begin
   X <= (not A and not B and not C and not D) or
        (not A and not B and not C and D) or
        (not A and B and not C and not D) or
        (not A and B and C and not D) or
        (not A and not B and C and not D) or
        (A and not B and not C and not D) or
        (A and not B and C and not D) or
        (A and B and C and not D) or
        (A and B and not C and not D) or
```

```
            (A and not B and not C and D) or
            (not A and B and not C and D) or
            (A and B and not C and D);
      end architecture Equation1;
```

(b) Now, use a four-variable Karnaugh map to reduce the original SOP expression to a minimum form. The original SOP expression is mapped in Figure 4–48.

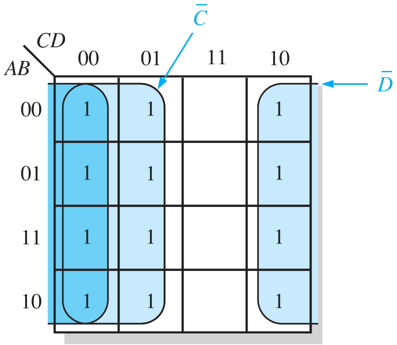


FIGURE 4–48

The original SOP Boolean expression that is plotted on the Karnaugh map in Figure 4–48 contains twelve 4-variable terms as indicated by the twelve 1s on the map. Recall that only the variables that do not change within a group remain in the expression for that group. The simplified expression taken from the map is developed next.

Combining the terms from the Karnaugh map, you get the following simplified expression, which is equivalent to the original SOP expression.

$$X = \overline{C} + \overline{D}$$

Using the simplified expression, the VHDL code can be rewitten with fewer terms, making the code more readable and easier to modify. Also, the logic implemented in a target device by the reduced code consumes much less space in the PLD. The VHDL program for the simplified SOP expression is as follows:

```
entity SimplifiedSOP is
  port (A, B, C, D: in bit; X: out bit);
end entity SimplifiedSOP;
architecture Equation2 of SimplifiedSOP is
begin
  X <= not C or not D
end architecture Equation2;
```

Related Problem

Write a VHDL architecture statement to describe the logic for the expression

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

As you have seen, the simplification of Boolean logic is important in the design of any logic function described in VHDL. Target devices have finite capacity and therefore require the creation of compact and efficient program code. Throughout this chapter, you have learned that the simplification of complex Boolean logic can lead to the elimination of unnecessary logic as well as the simplification of VHDL code.

Levels of Abstraction

A given logic function can be described at three different levels. It can be described by a truth table or a state diagram, by a Boolean expression, or by its logic diagram (schematic).

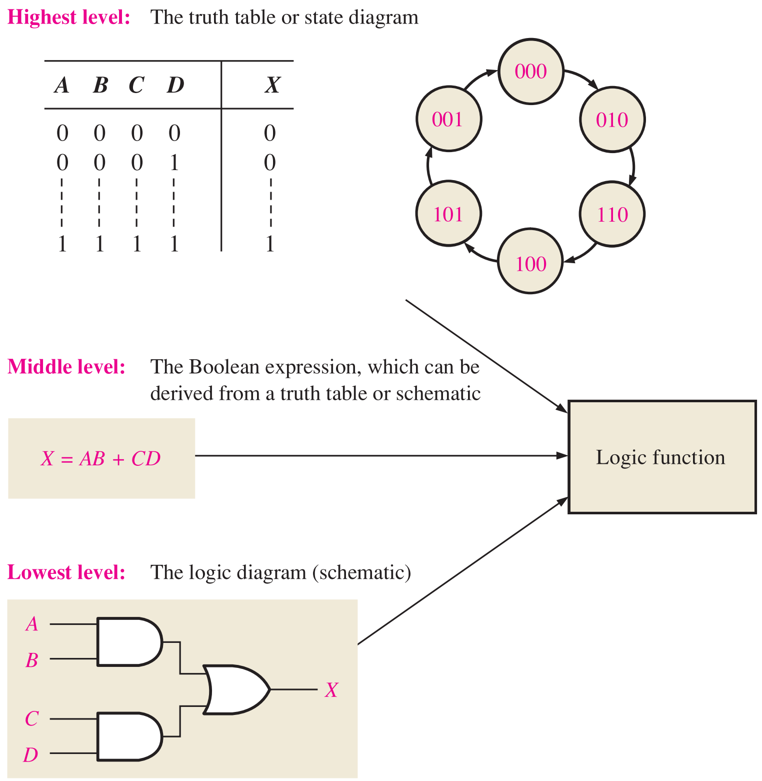

Highest level: The truth table or state diagram
A B C D X
0 0 0 0 0
0 0 0 1 0
1 1 1 1 1
000
010
110
100
101
001
Middle level: The Boolean expression, which can be derived from a truth table or schematic
X = AB + CD
Logic function
Lowest level: The logic diagram (schematic)
A
B
C
D
X

FIGURE 4–49 Illustration of the three levels of abstraction for describing a logic function.

The truth table and state diagram are the most abstract ways to describe a logic function. A Boolean expression is the next level of abstraction, and a schematic is the lowest level of abstraction. This concept is illustrated in Figure 4–49 for a simple logic circuit. VHDL provides three approaches for describing functions that correspond to the three levels of abstraction.

- The data flow approach is analogous to describing a logic function with a Boolean expression. The data flow approach specifies each of the logic gates and how the data flows through them. This approach was applied in Examples 4–37 and 4–38.
- The structural approach is analogous to using a logic diagram or schematic to describe a logic function. It specifies the gates and how they are connected, rather than how signals (data) flow through them. The structural approach is used to develop VHDL code for describing logic circuits in Chapter 5.
- The behavioral approach is analogous to describing a logic function using a state diagram or truth table. However, this approach is the most complex; it is usually restricted to logic functions whose operations are time dependent and normally require some type of memory.

SECTION 4–12 CHECKUP

1. What are the advantages of Boolean logic simplification in terms of writing a VHDL program?
2. How does Boolean logic simplification benefit a VHDL program in terms of the target device?
3. Name the three levels of abstraction for a combinational logic function and state the corresponding VHDL approaches for describing a logic function.

Applied Logic

Seven-Segment Display

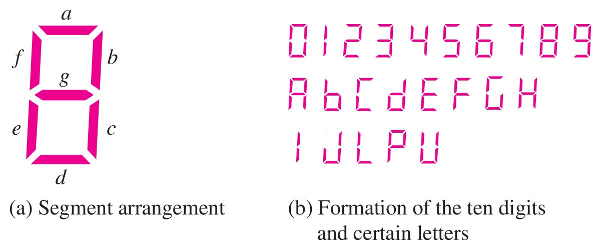
Seven-segment displays are used in many types of products that you see every day. A 7-segment display was used in the tablet-bottling system that was introduced in Chapter 1. The display in the bottling system is driven by logic circuits that decode a binary coded decimal (BCD) number and activate the appropriate digits on the display. BCD-to-7-segment decoder/drivers are readily available as single IC packages for activating the ten decimal digits.

In addition to the numbers from 0 to 9, the 7-segment display can show certain letters. For the tablet-bottling system, a requirement has been added to display the letters A, b, C, d, and E on a separate common-anode 7-segment display that uses a hexadecimal keypad for both the numerical inputs and the letters. These letters will be used to identify the type of vitamin tablet that is being bottled at any given time. In this application, the decoding logic for displaying the five letters is developed.

The 7-Segment Display

Two types of 7-segment displays are the LED and the LCD. Each of the seven segments in an LED display uses a light-emitting diode to produce a colored light when there is current through it and can be seen in the dark. An LCD or liquid-crystal display operates by polarizing light so that when a segment is not activated by a voltage, it reflects incident light and appears invisible against its background; however, when a segment is activated, it does not reflect light and appears black. LCD displays cannot be seen in the dark.

The seven segments in both LED and LCD displays are arranged as shown in Figure 4–50 and labeled *a*, *b*, *c*, *d*, *e*, *f*, and *g* as indicated in part (a). Selected segments are activated to create each of the ten decimal digits as well as certain letters of the alphabet, as shown in part (b). The letter *b* is shown as lowercase because a capital B would be the same as the digit 8. Similarly, for *d*, a capital letter would appear as a 0.



(a) Segment arrangement

(b) Formation of the ten digits and certain letters

FIGURE 4–50 Seven-segment display.

Exercise

1. List the segments used to form the digit 2.
2. List the segments used to form the digit 5.
3. List the segments used to form the letter A.
4. List the segments used to form the letter E.
5. Is there any one segment that is common to all digits?
6. Is there any one segment that is common to all letters?

Display Logic

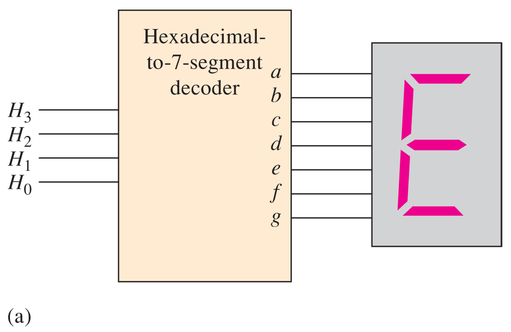
The segments in a 7-segment display can be used in the formation of various letters as shown in Figure 4–50(b). Each segment must be activated by its own decoding circuit that detects the code for any of the letters in which that segment is used. Because a common-anode display is used, the segments are turned *on* with a LOW (0) logic level and turned *off* with a HIGH (1) logic level. The active segments are shown for each of the letters required for the tablet-bottling system in Table 4–14. Even though the active level is LOW (lighting the LED), the logic expressions are developed exactly the same way as discussed in this chapter, by mapping the desired output (1, 0, or X) for every possible input, grouping the 1s on the map, and reading the SOP expression from the map. In effect, the reduced logic expression is the logic for keeping a given segment OFF. At first, this may sound confusing, but it is simple in practice and it avoids an output current capability issue with bipolar (TTL) logic (discussed in Chapter 15 on the website).

TABLE 4–14

Active segments for each of the five letters used in the system display.

Letter	Segments Activated
A	*a, b, c, e, f, g*
b	*c, d, e, f, g*
C	*a, d, e, f*
d	*b, c, d, e, g*
E	*a, d, e, f, g*

A block diagram of a 7-segment logic and display for generating the five letters is shown in Figure 4–51(a), and the truth table is shown in part (b). The logic has four hexadecimal inputs and seven outputs, one for each segment. Because the letter F is not used as an input, we will show it on the truth table with all outputs set to 1 (OFF).



Letter	Hexadecimal Inputs				Segment Ouputs						
	H_3	H_2	H_1	H_0	*a*	*b*	*c*	*d*	*e*	*f*	*g*
A	1	0	1	0	0	0	0	1	0	0	0
b	1	0	1	1	1	1	0	0	0	0	0
C	1	1	0	0	0	1	1	0	0	0	1
d	1	1	0	1	1	0	0	0	0	1	0
E	1	1	1	0	0	1	1	0	0	0	0
F	1	1	1	1	1	1	1	1	1	1	1

(b)

FIGURE 4–51 Hexadecimal-to-7-segment decoder for letters *A* through *E*, used in the system.

Karnaugh Maps and the Invalid BCD Code Detector

To develop the simplified logic for each segment, the truth table information in Figure 4–51 is mapped onto Karnaugh maps. Recall that the BCD numbers will not be shown on the letter display. For this reason, an entry that represents a BCD number will be entered as an "X" ("don't care") on the K-maps. This makes the logic much simpler but would put some strange outputs on the display unless steps are taken to eliminate that possibility. Because all of the letters are *invalid* BCD characters, the display is activated only when an invalid BCD code is entered into the keypad, thus allowing only letters to be displayed.

Expressions for the Segment Logic

Using the table in 4–51(b), a standard SOP expression can be written for each segment and then minimized using a K-map. The desired outputs from the truth table are entered in the appropriate cells representing the hex inputs. To obtain the minimum SOP expressions for the display logic, the 1s and Xs are grouped.

Segment a Segment *a* is used for the letters A, C, and E. For the letter A, the hexadecimal code is 1010 or, in terms of variables, $H_3\overline{H}_2H_1\overline{H}_0$. For the letter C, the hexadecimal code is 1100 or $H_3H_2\overline{H}_1\overline{H}_0$. For the letter E, the code is 1110 or $H_3H_2H_1\overline{H}_0$. The complete standard SOP expression for segment *a* is

$$a = H_3\overline{H}_2H_1\overline{H}_0 + H_3H_2\overline{H}_1\overline{H}_0 + H_3H_2H_1\overline{H}_0$$

Because a LOW is the active output state for each segment logic circuit, a 0 is entered on the Karnaugh map in each cell that represents the code for the letters in which the segment is *on*. The simplification of the expression for segment *a* is shown in Figure 4–52(a) after grouping the 1s and Xs.

Segment b Segment *b* is used for the letters A and d. The complete standard SOP expression for segment *b* is

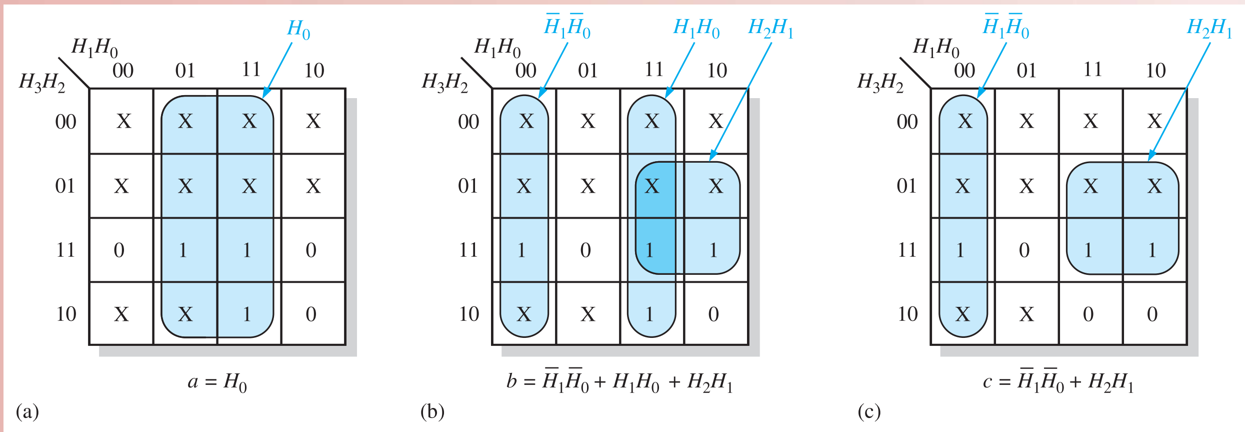
$$b = H_3\overline{H}_2H_1\overline{H}_0 + H_3H_2\overline{H}_1H_0$$

The simplification of the expression for segment *b* is shown in Figure 4–52(b).

Segment c Segment *c* is used for the letters A, b, and d. The complete standard SOP expression for segment *c* is

$$c = H_3\overline{H}_2H_1\overline{H}_0 + H_3\overline{H}_2H_1H_0 + H_3H_2\overline{H}_1H_0$$

The simplification of the expression for segment *c* is shown in Figure 4–52(c).


H_0
$\overline{H}_1\overline{H}_0$
H_1H_0
H_2H_1
$\overline{H}_1\overline{H}_0$
H_2H_1

H_3H_2 \ H_1H_0	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	0	1	1	0
10	X	X	1	0

$a = H_0$

(a)

H_3H_2 \ H_1H_0	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	1	0	1	1
10	X	X	1	0

$b = \overline{H}_1\overline{H}_0 + H_1H_0 + H_2H_1$

(b)

H_3H_2 \ H_1H_0	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	1	0	1	1
10	X	X	0	0

$c = \overline{H}_1\overline{H}_0 + H_2H_1$

(c)

FIGURE 4–52 Minimization of the expressions for segments *a*, *b*, and *c*.

Exercise

7. Develop the minimum expression for segment *d*.
8. Develop the minimum expression for segment *e*.
9. Develop the minimum expression for segment *f*.
10. Develop the minimum expression for segment *g*.

The Logic Circuits

From the minimum expressions, the logic circuits for each segment can be implemented. For segment *a*, connect the H_0 input directly (no gate) to the *a* segment on the display. The segment *b* and segment *c* logic are shown in Figure 4–53 using AND or OR gates. Notice that two of the terms (H_2H_1 and $\overline{H}_1\overline{H}_0$) appear in the expressions for both *b* and *c* logic so two of the AND gates can be used in both, as indicated.

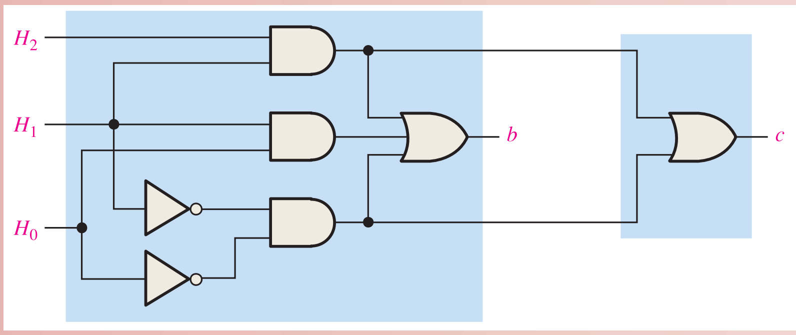


FIGURE 4–53 Segment-*b* and segment-*c* logic circuits.

Exercise

11. Show the logic for segment *d*.
12. Show the logic for segment *e*.
13. Show the logic for segment *f*.
14. Show the logic for segment *g*.

Describing the Decoding Logic with VHDL

The 7-segment decoding logic can be described using VHDL for implementation in a programmable logic device (PLD). The logic expressions for segments *a*, *b*, and *c* of the display are as follows:

$$a = H_0$$
$$b = \overline{H}_1\overline{H}_0 + H_1H_0 + H_2H_1$$
$$c = \overline{H}_1\overline{H}_0 + H_2H_1$$

- The VHDL code for segment *a* is

```
entity SEGLOGIC is
   port (H0: in bit; SEGa: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGa <= H0;
end architecture LogicFunction;
```

- The VHDL code for segment *b* is

```
entity SEGLOGIC is
   port (H0, H1, H2: in bit; SEGb: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGb <= (not H1 and not H0) or (H1 and H0) or (H2 and H1);
end architecture LogicFunction;
```

- The VHDL code for segment *c* is

```
entity SEGLOGIC is
   port (H0, H1, H2: in bit; SEGc: out bit);
end entity SEGLOGIC;
architecture LogicFunction of SEGLOGIC is
begin
   SEGc <= (not H1 and not H0) or (H2 and H1);
end architecture LogicFunction;
```

Exercise

15. Write the VHDL code for segments *d*, *e*, *f*, and *g*.

Simulation

The decoder simulation using Multisim is shown in Figure 4–54 with the letter E selected. Subcircuits are used for the segment logic to be developed as activities or in the lab. The purpose of simulation is to verify proper operation of the circuit.

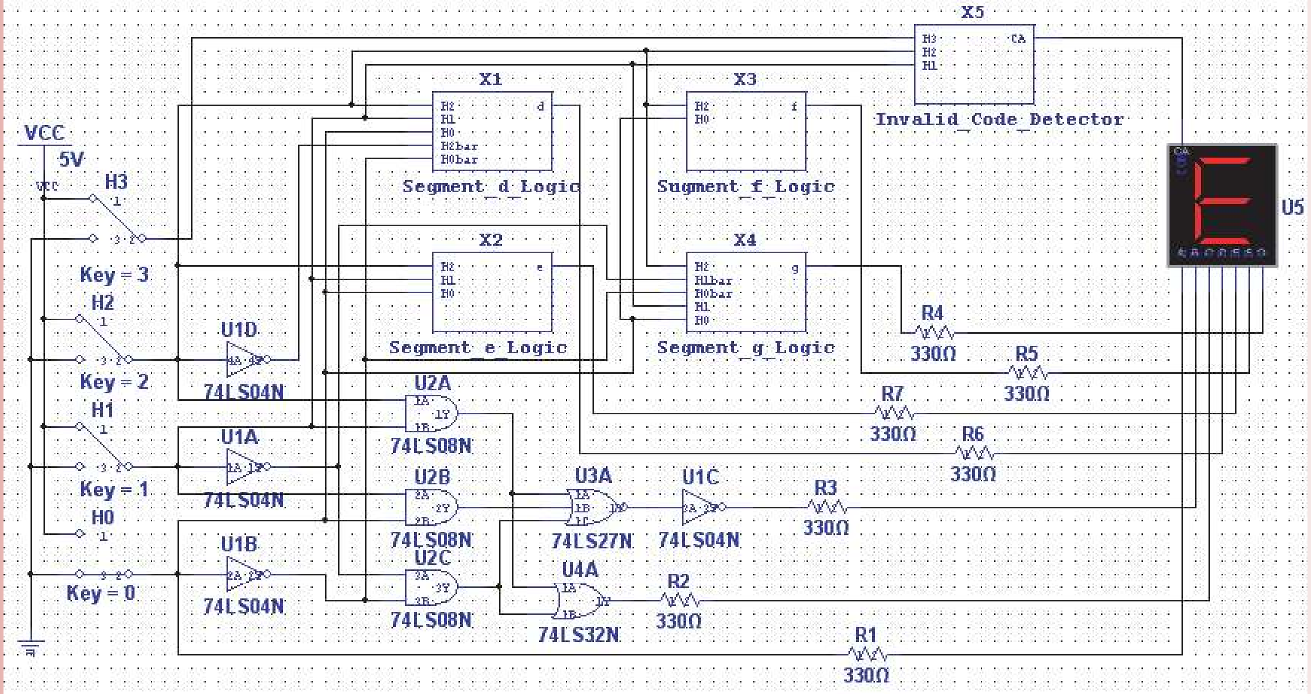


FIGURE 4–54 Multisim circuit screen for decoder and display.

MultiSim

> Open file AL04 in the Applied Logic folder on the website. Run the simulation of the decoder and display using your Multisim software. Observe the operation for the specified letters.

Putting Your Knowledge to Work

How would you modify the decoder for a common-cathode 7-segment display?

SUMMARY

- Gate symbols and Boolean expressions for the outputs of an inverter and 2-input gates are shown in Figure 4–55.

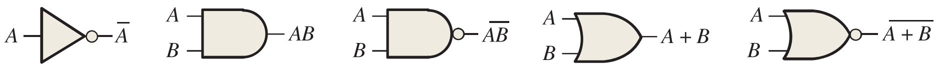


FIGURE 4–55

- Commutative laws: $A + B = B + A$
  $AB = BA$
- Associative laws: $A + (B + C) = (A + B) + C$
  $A(BC) = (AB)C$
- Distributive law: $A(B + C) = AB + AC$
- Boolean rules:
  1. $A + 0 = A$
  2. $A + 1 = 1$
  3. $A \cdot 0 = 0$
  4. $A \cdot 1 = A$
  5. $A + A = A$
  6. $A + \overline{A} = 1$
  7. $A \cdot A = A$
  8. $A \cdot \overline{A} = ($
  9. $\overline{\overline{A}} = A$
  10. $A + AB = A$
  11. $A + \overline{A}B = A + B$
  12. $(A + B)(A + C) = A + BC$
- DeMorgan's theorems:
  1. The complement of a product is equal to the sum of the complements of the terms in the product.

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

  2. The complement of a sum is equal to the product of the complements of the terms in the sum.

$$\overline{X + Y} = \overline{X}\,\overline{Y}$$

- Karnaugh maps for 3 variables have 8 cells and for 4 variables have 16 cells.
- Quinn-McCluskey is a method for simplification of Boolean expressions.
- The three levels of abstraction in VHDL are data flow, structural, and behavioral.

KEY TERMS

Key terms and other bold terms in the chapter are defined in the end-of-book glossary.

Complement The inverse or opposite of a number. In Boolean algebra, the inverse function, expressed with a bar over a variable. The complement of a 1 is 0, and vice versa.

"Don't care" A combination of input literals that cannot occur and can be used as a 1 or a 0 on a Karnaugh map for simplification.

Karnaugh map An arrangement of cells representing the combinations of literals in a Boolean expression and used for a systematic simplification of the expression.

Minimization The process that results in an SOP or POS Boolean expression that contains the fewest possible literals per term.

Product-of-sums (POS) A form of Boolean expression that is basically the ANDing of ORed terms.

Product term The Boolean product of two or more literals equivalent to an AND operation.

Sum-of-products (SOP) A form of Boolean expression that is basically the ORing of ANDed terms.

Sum term The Boolean sum of two or more literals equivalent to an OR operation.

Variable A symbol used to represent an action, a condition, or data that can have a value of 1 or 0, usually designated by an italic letter or word.

TRUE/FALSE QUIZ

Answers are at the end of the chapter.

1. *Variable, complement,* and *literal* are all terms used in Boolean algebra.

2. Addition in Boolean algebra is equivalent to the NOR function.

3. Multiplication in Boolean algebra is equivalent to the AND function.

4. The commutative law, associative law, and distributive law are all laws in Boolean algebra.

5. The complement of 0 is 0 itself.

6. When a Boolean variable is multiplied by its complement, the result is the variable.

7. "The complement of a product of variables is equal to the sum of the complements of each variable" is a statement of DeMorgan's theorem.
8. SOP means sum-of-products.
9. Karnaugh maps can be used to simplify Boolean expressions.
10. A 3-variable Karnaugh map has six cells.
11. VHDL is a type of hardware definition language.
12. A VHDL program consists of an entity and an architecture.

SELF-TEST

Answers are at the end of the chapter.

1. A variable is a symbol in Boolean algebra used to represent
(a) data **(b)** a condition
(c) an action **(d)** answers (a), (b), and (c)

2. The Boolean expression $A + B + C$ is
(a) a sum term **(b)** a literal term
(c) an inverse term **(d)** a product term

3. The Boolean expression $\overline{ABCD}$ is
(a) a sum term **(b)** a literal term
(c) an inverse term **(d)** a product term

4. The domain of the expression $A\overline{B}CD + A\overline{B} + \overline{C}D + B$ is
(a) A and D **(b)** B only
(c) A, B, C, and D **(d)** none of these

5. According to the associative law of addition,
(a) $A + B = B + A$ **(b)** $A = A + A$
(c) $(A + B) + C = A + (B + C)$ **(d)** $A + 0 = A$

6. According to commutative law of multiplication,
(a) $AB = BA$ **(b)** $A = AA$
(c) $(AB)C = A(BC)$ **(d)** $A0 = A$

7. According to the distributive law,
(a) $A(B + C) = AB + AC$ **(b)** $A(BC) = ABC$
(c) $A(A + 1) = A$ **(d)** $A + AB = A$

8. Which one of the following is *not* a valid rule of Boolean algebra?
(a) $A + 1 = 1$ **(b)** $A = \overline{A}$
(c) $AA = A$ **(d)** $A + 0 = A$

9. Which of the following rules states that if one input of an AND gate is always 1, the output is equal to the other input?
(a) $A + 1 = 1$ **(b)** $A + A = A$
(c) $A \cdot A = A$ **(d)** $A \cdot 1 = A$

10. According to DeMorgan's theorems, the complement of a product of variables is equal to
(a) the complement of the sum **(b)** the sum of the complements
(c) the product of the complements **(d)** answers (a), (b), and (c)

11. The Boolean expression $X = (A + B)(C + D)$ represents
(a) two ORs ANDed together **(b)** two ANDs ORed together
(c) A 4-input AND gate **(d)** a 4-input OR gate

12. An example of a sum-of-products expression is
(a) $A + B(C + D)$ **(b)** $\overline{A}B + A\overline{C} + A\overline{B}C$
(c) $(\overline{A} + B + C)(A + \overline{B} + C)$ **(d)** both answers (a) and (b)

13. An example of a product-of-sums expression is
(a) $A(B + C) + A\overline{C}$ **(b)** $(A + B)(\overline{A} + B + \overline{C})$
(c) $\overline{A} + \overline{B} + BC$ **(d)** both answers (a) and (b)

14. An example of a standard SOP expression is
(a) $\overline{A}B + A\overline{B}C + AB\overline{D}$ **(b)** $A\overline{B}C + A\overline{C}D$
(c) $A\overline{B} + \overline{A}B + AB$ **(d)** $A\overline{B}C\overline{D} + \overline{A}B + \overline{A}$

15. A 4-variable Karnaugh map has
(a) four cells **(b)** eight cells
(c) sixteen cells **(d)** thirty-two cells

16. In a 4-variable Karnaugh map, a 2-variable product term is produced by
(a) a 2-cell group of 1s **(b)** an 8-cell group of 1s
(c) a 4-cell group of 1s **(d)** a 4-cell group of 0s

17. The Quine-McCluskey method can be used to
(a) replace the Karnaugh map method **(b)** simplify expressions with 5 or more variables
(c) both (a) and (b) **(d)** none of the above

18. VHDL is a type of
(a) programmable logic **(b)** hardware description language
(c) programmable array **(d)** logical mathematics

19. In VHDL, a port is
(a) a type of entity **(b)** a type of architecture
(c) an input or output **(d)** a type of variable

20. Using VDHL, a logic circuit's inputs and outputs are described in the
(a) architecture **(b)** component
(c) entity **(d)** data flow

PROBLEMS

Answers to odd-numbered problems are at the end of the book.

Section 4–1 Boolean Operations and Expressions

1. Using Boolean notation, write an expression that is a 0 only when all of its variables (A, B, C, and D) are 0s.

2. Write an expression that is a 1 when one or more of its variables (A, B, C, D, and E) are 0s.

3. Write an expression that is a 0 when one or more of its variables (A, B, and C) are 0s.

4. Evaluate the following operations:

(a) $0 + 0 + 0 + 0$ **(b)** $0 + 0 + 0 + 1$ **(c)** $1 + 1 + 1 + 1$
(d) $1 \cdot 1 + 0 \cdot 0 + 1$ **(e)** $1 \cdot 0 \cdot 1 \cdot 0$ **(f)** $1 \cdot 0 + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 1$

5. Find the values of the variables that make each product term 1 and each sum term 0.

(a) ABC **(b)** $A + B + C$ **(c)** $\overline{A}\,\overline{B}C$ **(d)** $\overline{A} + \overline{B} + C$
(e) $A + \overline{B} + \overline{C}$ **(f)** $\overline{A} + \overline{B} + \overline{C}$

6. Find the value of X for all possible values of the variables.

(a) $X = A + B + C$ **(b)** $X = (A + B)C$ **(c)** $X = (A + B)(\overline{B + C})$
(d) $X = (A + B) + (\overline{AB + BC})$ **(e)** $X = (\overline{A} + \overline{B})(A + B)$

Section 4–2 Laws and Rules of Boolean Algebra

7. Identify the law of Boolean algebra upon which each of the following equalities is based:

(a) $A + AB + ABC + \overline{ABCD} = \overline{ABCD} + ABC + AB + A$
(b) $A + \overline{AB} + ABC + \overline{ABCD} = \overline{DCBA} + CBA + \overline{BA} + A$
(c) $AB(CD + \overline{CD} + EF + \overline{EF}) = ABCD + AB\overline{CD} + ABEF + AB\overline{EF}$

8. Identify the Boolean rule(s) on which each of the following equalities is based:

(a) $\overline{\overline{AB + CD}} + \overline{EF} = AB + CD + \overline{EF}$ **(b)** $A\overline{A}B + AB\overline{C} + AB\overline{B} = AB\overline{C}$
(c) $A(BC + BC) + AC = A(BC) + AC$ **(d)** $AB(C + \overline{C}) + AC = AB + AC$
(e) $A\overline{B} + A\overline{B}C = A\overline{B}$ **(f)** $ABC + \overline{AB} + \overline{ABC}D = ABC + \overline{AB} + D$

Section 4–3 DeMorgan's Theorems

9. Apply DeMorgan's theorems to each expression:

(a) $\overline{A + \overline{B}}$ **(b)** $\overline{\overline{A}B}$ **(c)** $\overline{A + B + C}$ **(d)** $\overline{ABC}$
(e) $\overline{A(B + C)}$ **(f)** $\overline{AB} + \overline{CD}$ **(g)** $\overline{AB + CD}$ **(h)** $\overline{(A + \overline{B})(\overline{C} + D)}$

10. Apply DeMorgan's theorems to each expression:

(a) $\overline{A\overline{B}(C + \overline{D})}$ **(b)** $\overline{AB(CD + EF)}$

(c) $\overline{(A + \overline{B} + C + \overline{D})} + \overline{ABC\overline{D}}$ **(d)** $\overline{\overline{(\overline{A} + B + C + D)}\,\overline{(A\overline{B}\,\overline{CD})}}$

(e) $\overline{\overline{AB}(CD + \overline{E}F)(\overline{AB} + \overline{CD})}$

11. Apply DeMorgan's theorems to the following:

(a) $\overline{\overline{(\overline{ABC})(\overline{EFG})} + \overline{(\overline{HIJ})(\overline{KLM})}}$ **(b)** $\overline{(A + \overline{B\overline{C}} + CD)} + \overline{\overline{BC}}$

(c) $\overline{\overline{(\overline{A + B})(\overline{C + D})(\overline{E + F})(\overline{G + H})}}$

Section 4–4 Boolean Analysis of Logic Circuits

12. Write the Boolean expression for each of the logic gates in Figure 4–56.

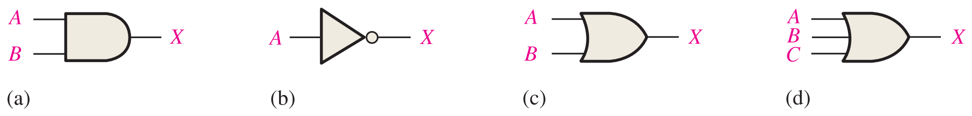


FIGURE 4–56

13. Write the Boolean expression for each of the logic circuits in Figure 4–57.

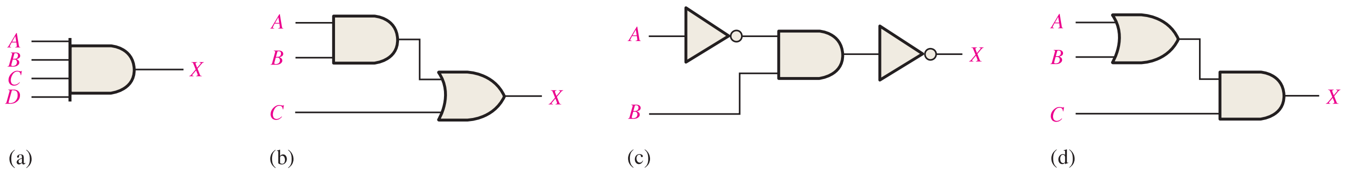


FIGURE 4–57

14. Draw the logic circuit represented by each of the following expressions:

(a) $A + B + C + D$ **(b)** $ABCD$

(c) $A + BC$ **(d)** $ABC + D$

15. Draw the logic circuit represented by each expression:

(a) $AB + \overline{AB}$ **(b)** $ABCD$

(c) $A + BC$ **(d)** $ABC + D$

16. (a) Draw a logic circuit for the case where the output, ENABLE, is HIGH only if the inputs, ASSERT and READY, are both LOW.

(b) Draw a logic circuit for the case where the output, HOLD, is HIGH only if the input, LOAD, is LOW and the input, READY, is HIGH.

17. Develop the truth table for each of the circuits in Figure 4–58.

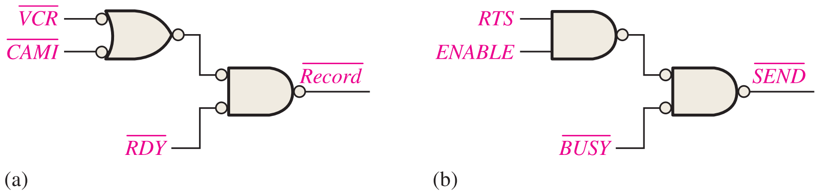


FIGURE 4–58

18. Construct a truth table for each of the following Boolean expressions:

(a) $A + B + C$ **(b)** ABC **(c)** $AB + BC + CA$

(d) $(A + B)(B + C)(C + A)$ **(e)** $A\overline{B} + B\overline{C} + C\overline{A}$

Section 4–5 Logic Simplification Using Boolean Algebra

19. Using Boolean algebra techniques, simplify the following expressions as much as possible:

(a) $A(A + B)$ **(b)** $A(\overline{A} + AB)$ **(c)** $BC + \overline{B}C$

(d) $A(A + \overline{A}B)$ **(e)** $A\overline{B}C + \overline{A}BC + \overline{A}\,\overline{B}C$

20. Using Boolean algebra, simplify the following expressions:

(a) $(\overline{A} + B)(A + C)$ **(b)** $A\overline{B} + A\overline{B}C + A\overline{B}CD + A\overline{B}CDE$
(c) $BC + \overline{BCD} + B$ **(d)** $(B + \overline{B})(BC + BC\overline{D})$
(e) $BC + (\overline{B} + \overline{C})D + BC$

21. Using Boolean algebra, simplify the following expressions:

(a) $CE + C(E + F) + \overline{E}(E + G)$ **(b)** $\overline{B}\,\overline{C}D + (\overline{B + C + D}) + \overline{B}\,\overline{C}\,\overline{D}E$
(c) $(C + CD)(C + \overline{C}D)(C + E)$ **(d)** $BCDE + BC(\overline{DE}) + (\overline{BC})DE$
(e) $BCD[BC + \overline{D}(CD + BD)]$

22. Determine which of the logic circuits in Figure 4–59 are equivalent.

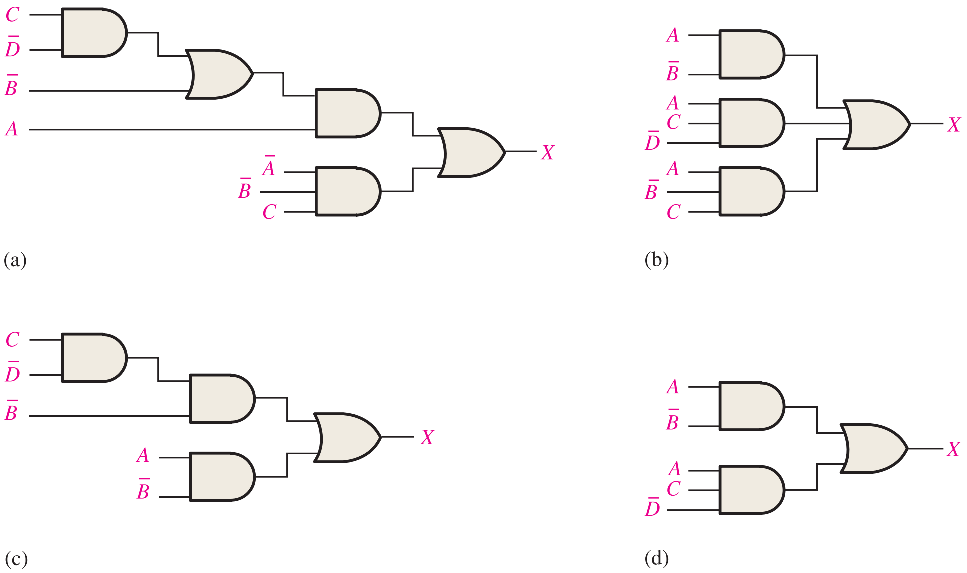


FIGURE 4–59

Section 4–6 Standard Forms of Boolean Expressions

23. Convert the following expressions to sum-of-product (SOP) forms:

(a) $(C + D)(A + \overline{D})$ **(b)** $A(A\overline{D} + C)$ **(c)** $(A + C)(CD + AC)$

24. Convert the following expressions to sum-of-product (SOP) forms:

(a) $BC + DE(B\overline{C} + DE)$ **(b)** $BC(\overline{C}\,\overline{D} + CE)$ **(c)** $B + C[BD + (C + \overline{D})E]$

25. Define the domain of each SOP expression in Problem 23 and convert the expression to standard SOP form.

26. Convert each SOP expression in Problem 24 to standard SOP form.

27. Determine the binary value of each term in the standard SOP expressions from Problem 25.

28. Determine the binary value of each term in the standard SOP expressions from Problem 26.

29. Convert each standard SOP expression in Problem 25 to standard POS form.

30. Convert each standard SOP expression in Problem 26 to standard POS form.

Section 4–7 Boolean Expressions and Truth Tables

31. Develop a truth table for each of the following standard SOP expressions:

(a) $ABC + \overline{A}\,\overline{B}C + AB\overline{C}$ **(b)** $\overline{X}\,\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z + \overline{X}YZ + XY\overline{Z}$

32. Develop a truth table for each of the following standard SOP expressions:

(a) $A\overline{B}C\overline{D} + AB\overline{C}\,\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$
(b) $WXYZ + \overline{W}X\overline{Y}Z + W\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}YZ + WX\overline{Y}\,\overline{Z}$

33. Develop a truth table for each of the SOP expressions:

(a) $\overline{A}B + AB\overline{C} + \overline{A}\,\overline{C} + A\overline{B}C$ **(b)** $\overline{X} + Y\overline{Z} + WZ + X\overline{Y}Z$

34. Develop a truth table for each of the standard POS expressions:

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

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

35. Develop a truth table for each of the standard POS expressions:

(a) $(A + B)(A + C)(A + B + C)$

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

36. For each truth table in Table 4–15, derive a standard SOP and a standard POS expression.

TABLE 4–15

ABC	*X*
000	0
001	1
010	0
011	0
100	1
101	1
110	0
111	1

(a)

ABC	*X*
000	0
001	0
010	0
011	0
100	0
101	1
110	1
111	1

(b)

ABCD	*X*
0000	1
0001	1
0010	0
0011	1
0100	0
0101	1
0110	1
0111	0
1000	0
1001	1
1010	0
1011	0
1100	1
1101	0
1110	0
1111	0

(c)

ABCD	*X*
0000	0
0001	0
0010	1
0011	0
0100	1
0101	1
0110	0
0111	1
1000	0
1001	0
1010	0
1011	1
1100	1
1101	0
1110	0
1111	1

(d)

Section 4–8 The Karnaugh Map

37. Draw a 3-variable Karnaugh map and label each cell according to its binary value.

38. Draw a 4-variable Karnaugh map and label each cell according to its binary value.

39. Write the standard product term for each cell in a 3-variable Karnaugh map.

Section 4–9 Karnaugh Map SOP Minimization

40. Use a Karnaugh map to find the minimum SOP form for each expression:

(a) $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + A\overline{B}C$ **(b)** $AC(\overline{B} + C)$

(c) $\overline{A}(BC + B\overline{C}) + A(BC + B\overline{C})$ **(d)** $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + AB\overline{C}$

41. Use a Karnaugh map to simplify each expression to a minimum SOP form:

(a) $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}C + \overline{A}BC + AB\overline{C}$ **(b)** $AC[\overline{B} + B(B + \overline{C})]$

(c) $DE\overline{F} + \overline{D}E\overline{F} + \overline{D}\,\overline{E}\,\overline{F}$

42. Expand each expression to a standard SOP form:

(a) $AB + A\overline{B}C + ABC$ **(b)** $A + BC$

(c) $A\overline{B}\,\overline{C}D + AC\overline{D} + B\overline{C}D + \overline{A}BC\overline{D}$ **(d)** $A\overline{B} + A\overline{B}\,\overline{C}D + CD + B\overline{C}D + ABCD$

43. Minimize each expression in Problem 42 with a Karnaugh map.

44. Use a Karnaugh map to reduce each expression to a minimum SOP form:

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

(b) $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + ABCD + ABC\overline{D}$

(c) $\overline{A}B(\overline{C}\,\overline{D} + \overline{C}D) + AB(\overline{C}\,\overline{D} + \overline{C}D) + A\overline{B}\,\overline{C}D$

(d) $(\overline{A}\,\overline{B} + A\overline{B})(CD + C\overline{D})$

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

45. Reduce the function specified in truth Table 4–16 to its minimum SOP form by using a Karnaugh map.

46. Use the Karnaugh map method to implement the minimum SOP expression for the logic function specified in truth Table 4–17.

47. Solve Problem 46 for a situation in which the last six binary combinations are not allowed.

TABLE 4–16

Inputs	Output
$A\ B\ C$	X
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	1

TABLE 4–17

Inputs	Output
$A\ B\ C\ D$	X
0 0 0 0	0
0 0 0 1	1
0 0 1 0	1
0 0 1 1	0
0 1 0 0	0
0 1 0 1	0
0 1 1 0	1
0 1 1 1	1
1 0 0 0	1
1 0 0 1	0
1 0 1 0	1
1 0 1 1	0
1 1 0 0	1
1 1 0 1	1
1 1 1 0	0
1 1 1 1	1

Section 4–10 Karnaugh Map POS Minimization

48. Use a Karnaugh map to find the minimum POS for each expression:

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

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

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

49. Use a Karnaugh map to simplify each expression to minimum POS form:

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

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

50. For the function specified in Table 4–16, determine the minimum POS expression using a Karnaugh map.

51. Determine the minimum POS expression for the function in Table 4–17.

52. Convert each of the following POS expressions to minimum SOP expressions using a Karnaugh map:

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

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

Section 4–11 The Quine-McCluskey Method

53. List the minterms in the expression

$$X = ABC + \overline{A}\,\overline{B}C + AB\overline{C} + A\overline{B}C + \overline{A}BC$$

54. List the minterms in the expression

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

55. Create a table for the number of 1s in the minterms for the expression in Problem 54 (similar to Table 4–10).

56. Create a table of first level minterms for the expression in Problem 54 (similar to Table 4–11).

57. Create a table of second level minterms for the expression in Problem 54 (similar to Table 4–12).

58. Create a table of prime implicants for the expression in Problem 54 (similar to Table 4–13).

59. Determine the final reduced expression for the expression in Problem 54.

Section 4–12 Boolean Expressions with VHDL

60. Write a VHDL program for the logic circuit in Figure 4–60.

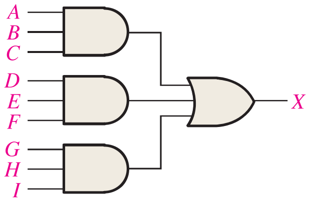


FIGURE 4–60

61. Write a program in VHDL for the expression

$$Y = A\overline{B}C + \overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + \overline{A}BC$$

Applied Logic

62. If you are required to choose a type of digital display for low light conditions, will you select LED or LCD 7-segment displays? Why?

63. Explain the purpose of the invalid code detector.

64. For segment *c*, how many fewer gates and inverters does it take to implement the minimum SOP expression than the standard SOP expression?

65. Repeat Problem 64 for the logic for segments *d* through *g*.

Special Design Problems

66. The logic for segments *b* and *c* in Figure 4–53 produces LOW outputs to activate the segments. If a type of 7-segment display is used that requires a HIGH to activate a segment, modify the logic accordingly.

67. Redesign the logic for segment *a* in the Applied Logic to include the letter F in the display.

68. Repeat Problem 67 for segments *b* through *g*.

69. Design the invalid code detector.

Multisim Troubleshooting Practice

70. Open file P04-70. For the specified fault, predict the effect on the circuit. Then introduce the fault and verify whether your prediction is correct.

71. Open file P04-71. For the specified fault, predict the effect on the circuit. Then introduce the fault and verify whether your prediction is correct.

72. Open file P04-72. For the observed behavior indicated, predict the fault in the circuit. Then introduce the suspected fault and verify whether your prediction is correct.

ANSWERS

SECTION CHECKUPS

Section 4–1 Boolean Operations and Expressions

1. $\overline{A} = \overline{0} = 1$

2. $A = 1, B = 1, C = 0; \overline{A} + \overline{B} + C = \overline{1} + \overline{1} + 0 = 0 + 0 + 0 = 0$

3. $A = 1, B = 0, C = 1; A\overline{B}C = 1 \cdot \overline{0} \cdot 1 = 1 \cdot 1 \cdot 1 =$

Section 4–2 Laws and Rules of Boolean Algebra

1. $A + (B + C + D) = (A + B + C) + D$

2. $A(B + C + D) = AB + AC + AD$

Section 4–3 DeMorgan's Theorems

1. (a) $\overline{ABC} + \overline{(\overline{D} + E)} = \overline{A} + \overline{B} + \overline{C} + D\overline{E}$ **(b)** $\overline{(A + B)C} = \overline{A}\,\overline{B} + \overline{C}$

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

Section 4–4 Boolean Analysis of Logic Circuits

1. $(C + D)B + A$

2. Abbreviated truth table: The expression is a 1 when A is 1 or when B and C are 1s or when B and D are 1s. The expression is 0 for all other variable combinations.

Section 4–5 Logic Simplification Using Boolean Algebra

1. (a) $A + AB + A\overline{B}C = A$ **(b)** $(\overline{A} + B)C + ABC = C(\overline{A} + B)$

(c) $A\overline{B}C(BD + CDE) + A\overline{C} = A(\overline{C} + \overline{B}DE)$

2. (a) *Original:* 2 AND gates, 1 OR gate, 1 inverter; *Simplified:* No gates (straight connection)

(b) *Original:* 2 OR gates, 2 AND gates, 1 inverter; *Simplified:* 1 OR gate, 1 AND gate, 1 inverter

(c) *Original:* 5 AND gates, 2 OR gates, 2 inverters; *Simplified:* 2 AND gates, 1 OR gate, 2 inverters

Section 4–6 Standard Forms of Boolean Expressions

1. (a) SOP **(b)** standard POS **(c)** standard SOP **(d)** POS

2. (a) $AB\overline{C}\,\overline{D} + AB\overline{C}D + ABC\overline{D} + ABCD + \overline{A}B\overline{C}D + \overline{A}BCD + \overline{A}\,\overline{B}C\overline{D} + \overline{A}BC\overline{D}$

(c) Already standard

3. (b) Already standard

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

Section 4–7 Boolean Expressions and Truth Tables

1. $2^5 = 32$ **2.** 0110 $\longrightarrow$ $\overline{W}XY\overline{Z}$ **3.** 1100 $\longrightarrow$ $\overline{W} + \overline{X} + Y + Z$

Section 4–8 The Karnaugh Map

1. (a) upper left cell: 000 **(b)** lower right cell: 101

(c) lower left cell: 100 **(d)** upper right cell: 001

2. (a) upper left cell: $\overline{X}\,\overline{Y}\,\overline{Z}$ **(b)** lower right cell: $X\overline{Y}Z$

(c) lower left cell: $X\overline{Y}\,\overline{Z}$ **(d)** upper right cell: $\overline{X}\,\overline{Y}Z$

3. (a) upper left cell: 0000 **(b)** lower right cell: 1010

(c) lower left cell: 1000 **(d)** upper right cell: 0010

4. (a) upper left cell: $\overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z}$ **(b)** lower right cell: $W\overline{X}Y\overline{Z}$

(c) lower left cell: $W\overline{X}\,\overline{Y}\,\overline{Z}$ **(d)** upper right cell: $\overline{W}\,\overline{X}Y\overline{Z}$

Section 4–9 Karnaugh Map SOP Minimization

1. 8-cell map for 3 variables; 16-cell map for 4 variables

2. $AB + B\overline{C} + \overline{A}\,\overline{B}C$

3. (a) $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + ABC + AB\overline{C}$

(b) $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + ABC + A\overline{B}\,\overline{C} + A\overline{B}C$

(c) $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}B\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + \overline{A}BCD + \overline{A}BC\overline{D} + A\overline{B}\,\overline{C}D + A\overline{B}CD$

(d) $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}B\overline{C}\,\overline{D} + AB\overline{C}\,\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + AB\overline{C}D + A\overline{B}CD + \overline{A}\,\overline{B}C\overline{D} +$ $\overline{A}BC\overline{D} + ABC\overline{D} + A\overline{B}C\overline{D}$

Section 4–10 Karnaugh Map POS Minimization

1. In mapping a POS expression, 0s are placed in cells whose value makes the standard sum term zero; and in mapping an SOP expression 1s are placed in cells having the same values as the product terms.

2. 0 in the 1011 cell: $\overline{A} + B + \overline{C} + \overline{D}$
3. 1 in the 0010 cell: $\overline{A}\,\overline{B}C\overline{D}$

Section 4–11 The Quine-McCluskey Method

1. A minterm is a product term in which each variable appears once, either complemented or uncomplemented.
2. An essential prime implicant is a product term that cannot be further simplified by combining with other terms.

Section 4–12 Boolean Expressions with VHDL

1. Simplification can make a VHDL program shorter, easier to read, and easier to modify.
2. Code simplification results in less space used in a target device, thus allowing capacity for more complex circuits.
3. Truth table: Behavioral
   Boolean expression: Data flow
   Logic diagram: Structural

RELATED PROBLEMS FOR EXAMPLES

4–1 $\overline{A} + B = 0$ when $A = 1$ and $B = 0$.

4–2 $\overline{A}\,\overline{B} = 1$ when $A = 0$ and $B = 0$.

4–3 XYZ

4–4 $W + X + Y + Z$

4–5 $ABC\overline{D}\,\overline{E}$

4–6 $(A + \overline{B} + \overline{C}D)\overline{E}$

4–7 $\overline{ABCD} = \overline{A} + \overline{B} + \overline{C} + \overline{D}$

4–8 Results should be same as example.

4–9 $A\overline{B}$

4–10 CD

4–11 $AB\overline{C} + \overline{A}C + \overline{A}\,\overline{B}$

4–12 $\overline{A} + \overline{B} + \overline{C}$

4–13 Results should be same as example.

4–14 $\overline{A}B\overline{C} + AB + A\overline{C} + A\overline{B} + \overline{B}\,\overline{C}$

4–15 $W\overline{X}YZ + W\overline{X}Y\overline{Z} + W\overline{X}Y\overline{Z} + \overline{W}\,\overline{X}Y\overline{Z} + WX\overline{Y}Z + WX\overline{Y}\,\overline{Z}$

4–16 011, 101, 110, 010, 111. Yes

4–17 $(A + \overline{B} + C)(A + \overline{B} + \overline{C})(A + B + C)(\overline{A} + B + C)$

4–18 010, 100, 001, 111, 011. Yes

4–19 SOP and POS expressions are equivalent.

4–20 See Table 4–18.

4–21 See Table 4–19.

TABLE 4–18

A	*B*	*C*	*X*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

TABLE 4–19

A	*B*	*C*	*X*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

4–22 The SOP and POS expressions are equivalent.

4–23 See Figure 4–61.

4–24 See Figure 4–62.

C AB	0	1
00		
01		1
11		
10	1	1

FIGURE 4–61

CD AB	00	01	11	10
00				
01				1
11	1		1	1
10				

FIGURE 4–62

4–25 See Figure 4–63.

4–26 See Figure 4–64.

C AB	0	1
00	1	
01	1	1
11		1
10		

FIGURE 4–63

CD AB	00	01	11	10
00		1		
01		1		1
11	1	1	1	1
10	1	1	1	1

FIGURE 4–64

4–27 No other ways

4–28 $X = B + \overline{A}C + A\overline{C}D + C\overline{D}$

4–29 $X = \overline{D} + A\overline{B}C + B\overline{C} + \overline{A}B$

4–30 $Q = X + Y$

4–31 $Q = \overline{X}\,\overline{Y}\,\overline{Z} + W\overline{X}Z + \overline{W}YZ$

4–32 See Figure 4–65.

4–33 See Figure 4–66.

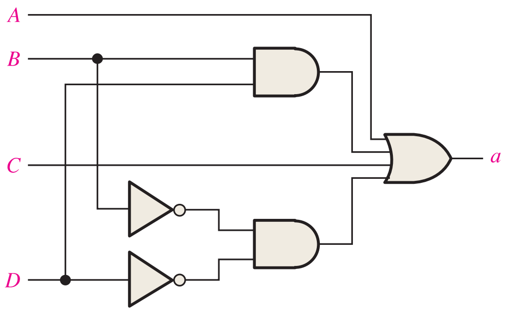


FIGURE 4–65

CD AB	00	01	11	10
00	0	0		
01				0
11				
10				0

FIGURE 4–66

4–34 $(X + \overline{Y})(X + \overline{Z})(\overline{X} + Y + Z)$

4–35 $(\overline{X} + \overline{Y} + Z)(\overline{W} + \overline{X} + Z)(W + X + Y + Z)(W + \overline{X} + Y + \overline{Z})$

4–36 $\overline{Y}\,\overline{Z} + \overline{X}\,\overline{Z} + \overline{W}Y + \overline{X}\,\overline{Y}Z$

4–37

```
architecture RelProb_1 of Example4_37 is
begin
  X <= (not A or B or C) and D;
end architecture RelProb_1;

architecture RelProb_2 of Example4_37 is
begin
  X <= (not A and D or B and D or C and D);
end architecture RelProb_2;
```

4–38

```
architecture RelProb of Example4_38 is
begin
  X <= not(A and ((B and C) or not D))
end architecture RelProb;
```

TRUE/FALSE QUIZ

1. T **2.** F **3.** T **4.** T **5.** F **6.** F
7. T **8.** T **9.** T **10.** F **11.** F **12.** T

SELF-TEST

1. (d) **2.** (a) **3.** (d) **4.** (c) **5.** (c) **6.** (a) **7.** (a)
8. (b) **9.** (d) **10.** (b) **11.** (a) **12.** (b) **13.** (b) **14.** (c)
15. (c) **16.** (c) **17.** (c) **18.** (b) **19.** (c) **20.** (c)

CHAPTER 5

Combinational Logic Analysis

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Analyze basic combinational logic circuits, such as AND-OR, AND-OR-Invert, exclusive-OR, and exclusive-NOR
- Use AND-OR and AND-OR-Invert circuits to implement sum-of-products (SOP) and product-of-sums (POS) expressions
- Write the Boolean output expression for any combinational logic circuit
- Develop a truth table from the output expression for a combinational logic circuit
- Use the Karnaugh map to expand an output expression containing terms with missing variables into a full SOP form
- Design a combinational logic circuit for a given Boolean output expression
- Design a combinational logic circuit for a given truth table
- Simplify a combinational logic circuit to its minimum form
- Use NAND gates to implement any combinational logic function
- Use NOR gates to implement any combinational logic function
- Analyze the operation of logic circuits with pulse inputs
- Write VHDL programs for simple logic circuits
- Troubleshoot faulty logic circuits
- Troubleshoot logic circuits by using signal tracing and waveform analysis
- Apply combinational logic to an application

KEY TERMS

Key terms are in order of appearance in the chapter.

- Universal gate
- Negative-OR
- Negative-AND
- Component
- Signal
- Node
- Signal tracing

VISIT THE WEBSITE

Study aids for this chapter are available at
http://www.pearsonglobaleditions.com/floyd

INTRODUCTION

In Chapters 3 and 4, logic gates were discussed on an individual basis and in simple combinations. You were introduced to SOP and POS implementations, which are basic forms of combinational logic. When logic gates are connected together to produce a specified output for certain specified combinations of input variables, with no storage involved, the resulting circuit is in the category of **combinational logic**. In combinational logic, the output level is at all times dependent on the combination of input levels. This chapter expands on the material introduced in earlier chapters with a coverage of the analysis, design, and troubleshooting of various combinational logic circuits. The VHDL structural approach is introduced and applied to combinational logic.

5–1 Basic Combinational Logic Circuits

In Chapter 4, you learned that SOP expressions are implemented with an AND gate for each product term and one OR gate for summing all of the product terms. As you know, this SOP implementation is called AND-OR logic and is the basic form for realizing standard Boolean functions. In this section, the AND-OR and the AND-OR-Invert are examined; the exclusive-OR and exclusive-NOR gates, which are actually a form of AND-OR logic, are also covered.

After completing this section, you should be able to

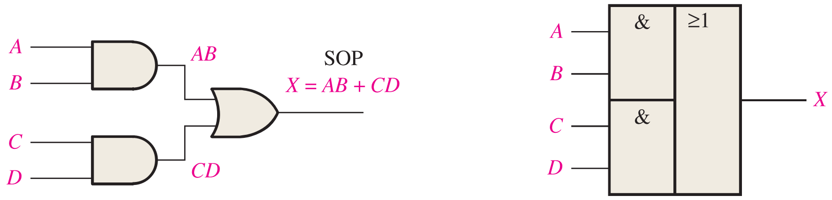
- Analyze and apply AND-OR circuits
- Analyze and apply AND-OR-Invert circuits
- Analyze and apply exclusive-OR gates
- Analyze and apply exclusive-NOR gates

AND-OR Logic

AND-OR logic produces an SOP expression.

Figure 5–1(a) shows an AND-OR circuit consisting of two 2-input AND gates and one 2-input OR gate; Figure 5–1(b) is the ANSI standard rectangular outline symbol. The Boolean expressions for the AND gate outputs and the resulting SOP expression for the output *X* are shown on the diagram. In general, an AND-OR circuit can have any number of AND gates, each with any number of inputs.

The truth table for a 4-input AND-OR logic circuit is shown in Table 5–1. The intermediate AND gate outputs (the *AB* and *CD* columns) are also shown in the table.



(a) Logic diagram (ANSI standard distinctive shape symbols)

(b) ANSI standard rectangular outline symbol

FIGURE 5–1 An example of AND-OR logic. Open file F05-01 to verify the operation. *A Multisim tutorial is available on the website.*

TABLE 5–1

Truth table for the AND-OR logic in Figure 5–1.

Inputs						Output
A	*B*	*C*	*D*	*AB*	*CD*	*X*
0	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	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1

An AND-OR circuit directly implements an SOP expression, assuming the complements (if any) of the variables are available. The operation of the AND-OR circuit in Figure 5–1 is stated as follows:

For a 4-input AND-OR logic circuit, the output X is HIGH (1) if both input A and input B are HIGH (1) or both input C and input D are HIGH (1).

EXAMPLE 5–1

In a certain chemical-processing plant, a liquid chemical is used in a manufacturing process. The chemical is stored in three different tanks. A level sensor in each tank produces a HIGH voltage when the level of chemical in the tank drops below a specified point.

Design a circuit that monitors the chemical level in each tank and indicates when the level in any two of the tanks drops below the specified point.

Solution

The AND-OR circuit in Figure 5–2 has inputs from the sensors on tanks A, B, and C as shown. The AND gate G_1 checks the levels in tanks A and B, gate G_2 checks tanks A and C, and gate G_3 checks tanks B and C. When the chemical level in any two of the tanks gets too low, one of the AND gates will have HIGHs on both of its inputs, causing its output to be HIGH; and so the final output X from the OR gate is HIGH. This HIGH input is then used to activate an indicator such as a lamp or audible alarm, as shown in the figure.

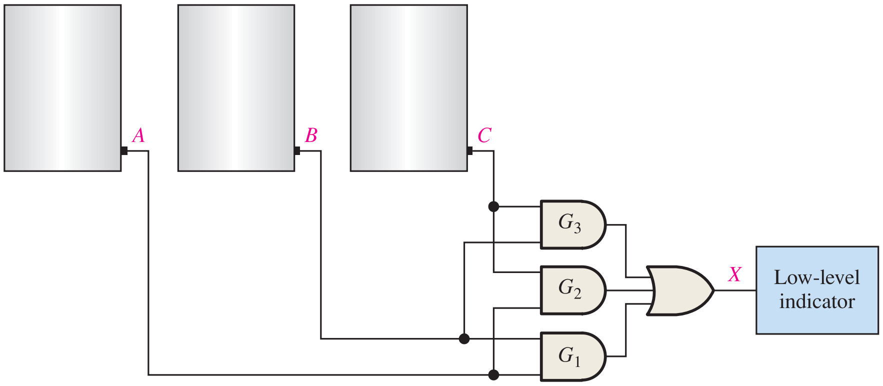


FIGURE 5–2

Related Problem*

Write the Boolean SOP expression for the AND-OR logic in Figure 5–2.

*Answers are at the end of the chapter.

AND-OR-Invert Logic

When the output of an AND-OR circuit is complemented (inverted), it results in an AND-OR-Invert circuit. Recall that AND-OR logic directly implements SOP expressions. POS expressions can be implemented with AND-OR-Invert logic. This is illustrated as follows, starting with a POS expression and developing the corresponding AND-OR-Invert (AOI) expression.

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

The logic diagram in Figure 5–3(a) shows an AND-OR-Invert circuit with four inputs and the development of the POS output expression. The ANSI standard rectangular outline symbol is shown in part (b). In general, an AND-OR-Invert circuit can have any number of AND gates, each with any number of inputs.

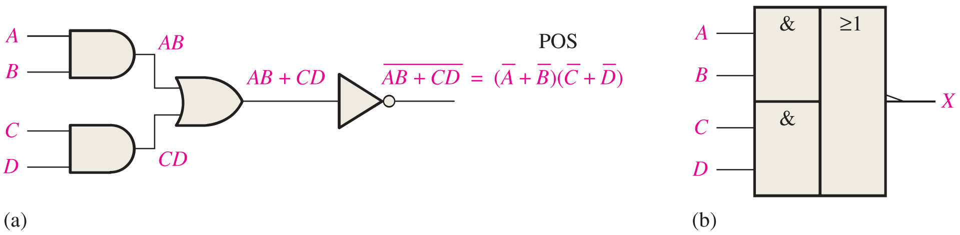

A
B
AB
AB + CD
C
D
CD
POS
$\overline{AB + CD} = (\overline{A} + \overline{B})(\overline{C} + \overline{D})$
(a)
A
B
C
D
&
&
≥1
X
(b)

FIGURE 5–3 An AND-OR-Invert circuit produces a POS output. Open file F05-03 to verify the operation.

The operation of the AND-OR-Invert circuit in Figure 5–3 is stated as follows:

For a 4-input AND-OR-Invert logic circuit, the output *X* is LOW (0) if both input *A* and input *B* are HIGH (1) or both input *C* and input *D* are HIGH (1).

A truth table can be developed from the AND-OR truth table in Table 5–1 by simply changing all 1s to 0s and all 0s to 1s in the output column.

EXAMPLE 5–2

The sensors in the chemical tanks of Example 5–1 are being replaced by a new model that produces a LOW voltage instead of a HIGH voltage when the level of the chemical in the tank drops below a critical point.

Modify the circuit in Figure 5–2 to operate with the different input levels and still produce a HIGH output to activate the indicator when the level in any two of the tanks drops below the critical point. Show the logic diagram.

Solution

The AND-OR-Invert circuit in Figure 5–4 has inputs from the sensors on tanks *A*, *B*, and *C* as shown. The AND gate G_1 checks the levels in tanks *A* and *B*, gate G_2 checks tanks *A* and *C,* and gate G_3 checks tanks *B* and *C*. When the chemical level in any two of the tanks gets too low, each AND gate will have a LOW on at least one input, causing its output to be LOW and, thus, the final output *X* from the inverter is HIGH. This HIGH output is then used to activate an indicator.

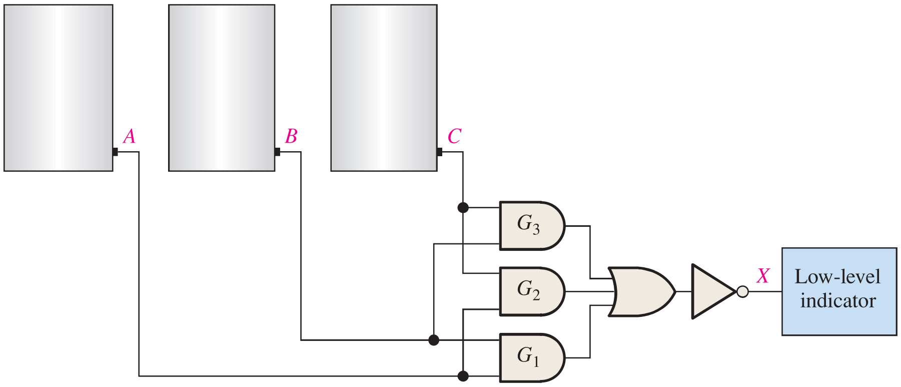


FIGURE 5–4

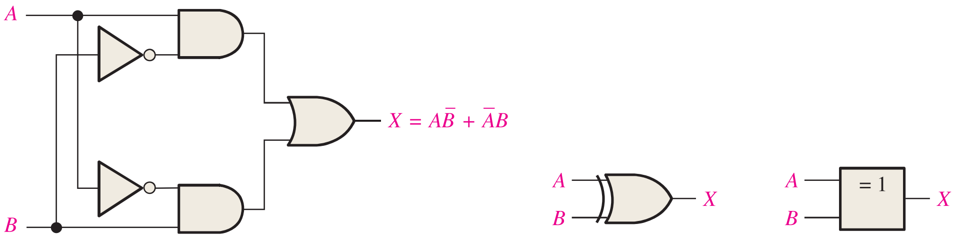
Related Problem

Write the Boolean expression for the AND-OR-Invert logic in Figure 5–4 and show that the output is HIGH (1) when any two of the inputs *A*, *B*, and *C* are LOW (0).

Exclusive-OR Logic

The exclusive-OR gate was introduced in Chapter 3. Although this circuit is considered a type of logic gate with its own unique symbol, it is actually a combination of two AND gates, one OR gate, and two inverters, as shown in Figure 5–5(a). The two ANSI standard exclusive-OR logic symbols are shown in parts (b) and (c).

The XOR gate is actually a combination of other gates.



(a) Logic diagram

(b) ANSI distinctive shape symbol

(c) ANSI rectangular outline symbol

FIGURE 5–5 Exclusive-OR logic diagram and symbols. Open file F05-05 to verify the operation.

The output expression for the circuit in Figure 5–5 is

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

Evaluation of this expression results in the truth table in Table 5–2. Notice that the output is HIGH only when the two inputs are at opposite levels. A special exclusive-OR operator $\oplus$ is often used, so the expression $X = A\overline{B} + \overline{A}B$ can be stated as "*X* is equal to *A* exclusive-OR *B*" and can be written as

$$X = A \oplus B$$

TABLE 5–2

Truth table for an exclusive-OR.

A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

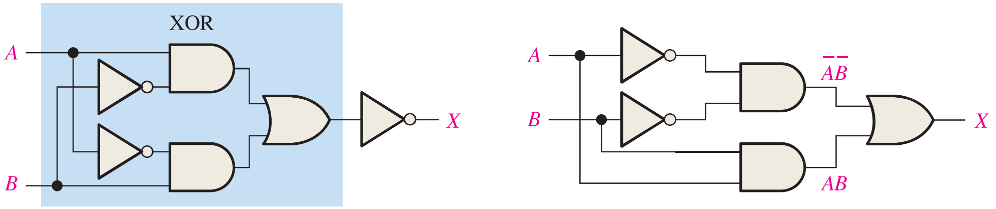
Exclusive-NOR Logic

As you know, the complement of the exclusive-OR function is the exclusive-NOR, which is derived as follows:

$$X = \overline{A\overline{B} + \overline{A}B} = \overline{(A\overline{B})}\,\overline{(\overline{A}B)} = (\overline{A} + B)(A + \overline{B}) = \overline{A}\,\overline{B} + AB$$

Notice that the output *X* is HIGH only when the two inputs, *A* and *B*, are at the same level.

The exclusive-NOR can be implemented by simply inverting the output of an exclusive-OR, as shown in Figure 5–6(a), or by directly implementing the expression $\overline{A}\,\overline{B} + AB$, as shown in part (b).



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

(b) $X = \overline{A}\,\overline{B} + AB$

FIGURE 5–6 Two equivalent ways of implementing the exclusive-NOR. Open files F05-06 (a) and (b) to verify the operation.

EXAMPLE 5–3

Use exclusive-OR gates to implement an even-parity code generator for an original 4-bit code.

Solution

Recall from Chapter 2 that a parity bit is added to a binary code in order to provide error detection. For even parity, a parity bit is added to the original code to make the total number of 1s in the code even. The circuit in Figure 5–7 produces a 1 output when there is an odd number of 1s on the inputs in order to make the total number of 1s in the output code even. A 0 output is produced when there is an even number of 1s on the inputs.

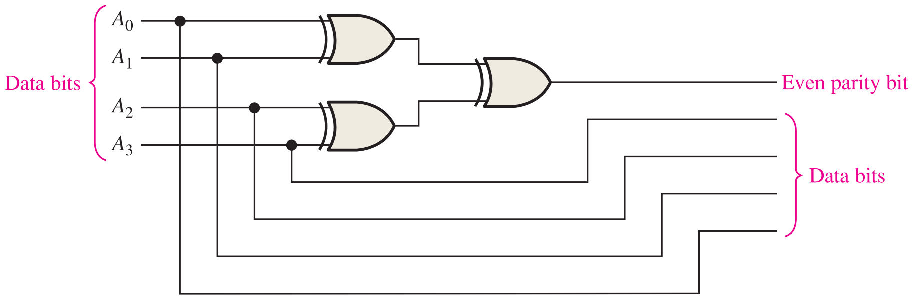


FIGURE 5–7 Even-parity generator.

Related Problem

How would you verify that a correct even-parity bit is generated for each combination of the four data bits?

EXAMPLE 5–4

Use exlusive-OR gates to implement an even-parity checker for the 5-bit code generated by the circuit in Example 5–3.

Solution

The circuit in Figure 5–8 produces a 1 output when there is an error in the five-bit code and a 0 when there is no error.

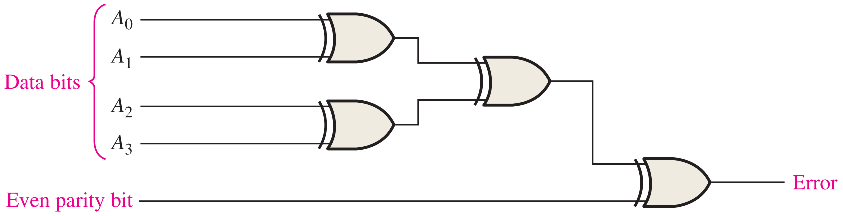


FIGURE 5–8 Even-parity checker.

Related Problem

How would you verify that an error is indicated when the input code is incorrect?