CHAPTER 6

Functions of Combinational Logic

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Distinguish between half-adders and full-adders
- Use full-adders to implement multibit parallel binary adders
- Explain the differences between ripple carry and look-ahead carry parallel adders
- Use the magnitude comparator to determine the relationship between two binary numbers and use cascaded comparators to handle the comparison of larger numbers
- Implement a basic binary decoder
- Use BCD-to-7-segment decoders in display systems
- Apply a decimal-to-BCD priority encoder in a simple keyboard application
- Convert from binary to Gray code, and Gray code to binary by using logic devices
- Apply data selectors/multiplexers in multiplexed displays and as a function generator
- Use decoders as demultiplexers
- Explain the meaning of parity
- Use parity generators and checkers to detect bit errors in digital systems
- Describe a simple data communications system
- Write VHDL programs for several logic functions
- Identify glitches, common bugs in digital systems

KEY TERMS

Key terms are in order of appearance in the chapter.

- Half-adder
- Full-adder
- Cascading
- Ripple carry
- Look-ahead carry
- Comparator
- Decoder
- Encoder
- Priority encoder
- Multiplexer (MUX)
- Demultiplexer (DEMUX)
- Parity bit
- Glitch

VISIT THE WEBSITE

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

INTRODUCTION

In this chapter, several types of combinational logic functions are introduced including adders, comparators, decoders, encoders, code converters, multiplexers (data selectors), demultiplexers, and parity generators/checkers. VHDL implementation of each logic function is provided, and examples of fixed-function IC devices are included. Each device introduced may also be available in other logic families.

6–1 Half and Full Adders

Adders are important in computers and also in other types of digital systems in which numerical data are processed. An understanding of the basic adder operation is fundamental to the study of digital systems. In this section, the half-adder and the full-adder are introduced.

After completing this section, you should be able to

- Describe the function of a half-adder
- Draw a half-adder logic diagram
- Describe the function of the full-adder
- Draw a full-adder logic diagram using half-adders
- Implement a full-adder using AND-OR logic

The Half-Adder

A half-adder adds two bits and produces a sum and an output carry.

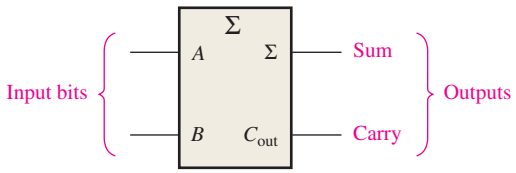
Recall the basic rules for binary addition as stated in Chapter 2.

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

The operations are performed by a logic circuit called a **half-adder**.

The half-adder accepts two binary digits on its inputs and produces two binary digits on its outputs—a sum bit and a carry bit.

A half-adder is represented by the logic symbol in Figure 6–1.



MultiSim

FIGURE 6–1 Logic symbol for a half-adder. Open file F06-01 to verify operation.
A Multisim tutorial is available on the website.

Half-Adder Logic

From the operation of the half-adder as stated in Table 6–1, expressions can be derived for the sum and the output carry as functions of the inputs. Notice that the output carry (C_{out}) is a 1 only when both A and B are 1s; therefore, C_{out} can be expressed as the AND of the input variables.

$$C_{out} = AB \qquad \text{Equation 6–1}$$

Now observe that the sum output (Σ) is a 1 only if the input variables, A and B, are not equal. The sum can therefore be expressed as the exclusive-OR of the input variables.

$$\Sigma = A \oplus B \qquad \text{Equation 6–2}$$

From Equations 6–1 and 6–2, the logic implementation required for the half-adder function can be developed. The output carry is produced with an AND gate with A and B on the

TABLE 6–1

Half-adder truth table.

A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Σ = sum
C_{out} = output carry
A and B = input variables (operands)

inputs, and the sum output is generated with an exclusive-OR gate, as shown in Figure 6–2. Remember that the exclusive-OR can be implemented with AND gates, an OR gate, and inverters.

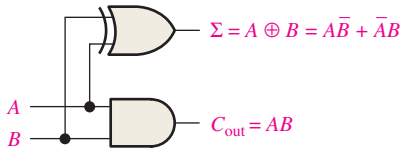


FIGURE 6–2 Half-adder logic diagram.

The Full-Adder

The second category of adder is the **full-adder**.

A full-adder has an input carry while the half-adder does not.

The full-adder accepts two input bits and an input carry and generates a sum output and an output carry.

The basic difference between a full-adder and a half-adder is that the full-adder accepts an input carry. A logic symbol for a full-adder is shown in Figure 6–3, and the truth table in Table 6–2 shows the operation of a full-adder.

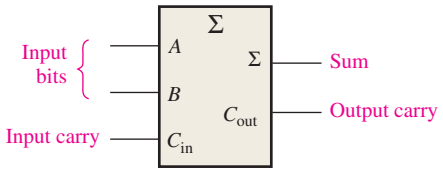


FIGURE 6–3 Logic symbol for a full-adder. Open file F06-03 to verify operation.

MultiSim

TABLE 6–2

Full-adder truth table.

A	*B*	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

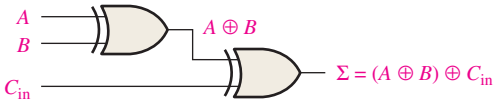
C_{in} = input carry, sometimes designated as *CI*
C_{out} = output carry, sometimes designated as *CO*
Σ = sum
A and *B* = input variables (operands)

Full-Adder Logic

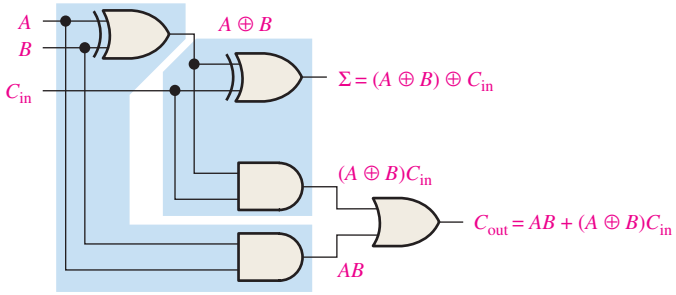
The full-adder must add the two input bits and the input carry. From the half-adder you know that the sum of the input bits *A* and *B* is the exclusive-OR of those two variables, $A \oplus B$. For the input carry (C_{in}) to be added to the input bits, it must be exclusive-ORed with $A \oplus B$, yielding the equation for the sum output of the full-adder.

$$\Sigma = (A \oplus B) \oplus C_{in} \qquad \text{Equation 6–3}$$

This means that to implement the full-adder sum function, two 2-input exclusive-OR gates can be used. The first must generate the term $A \oplus B$, and the second has as its inputs the output of the first XOR gate and the input carry, as illustrated in Figure 6–4(a).



(a) Logic required to form the sum of three bits



(b) Complete logic circuit for a full-adder (each half-adder is enclosed by a shaded area)

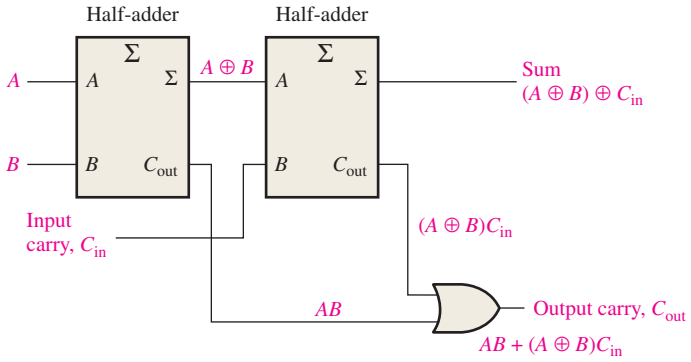
MultiSim

FIGURE 6–4 Full-adder logic. Open file F06-04 to verify operation.

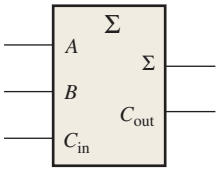
The output carry is a 1 when both inputs to the first XOR gate are 1s or when both inputs to the second XOR gate are 1s. You can verify this fact by studying Table 6–2. The output carry of the full-adder is therefore produced by input A ANDed with input B and $A \oplus B$ ANDed with C_{in}. These two terms are ORed, as expressed in Equation 6–4. This function is implemented and combined with the sum logic to form a complete full-adder circuit, as shown in Figure 6–4(b).

$$C_{out} = AB + (A \oplus B)C_{in} \qquad \text{Equation 6–4}$$

Notice in Figure 6–4(b) there are two half-adders, connected as shown in the block diagram of Figure 6–5(a), with their output carries ORed. The logic symbol shown in Figure 6–5(b) will normally be used to represent the full-adder.



(a) Arrangement of two half-adders to form a full-adder

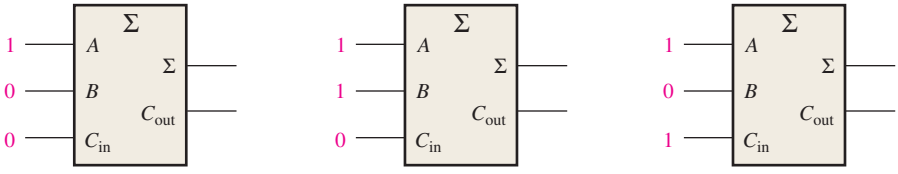


(b) Full-adder logic symbol

FIGURE 6–5 Full-adder implemented with half-adders.

EXAMPLE 6–1

For each of the three full-adders in Figure 6–6, determine the outputs for the inputs shown.



(a) (b) (c)

FIGURE 6–6

Solution

(a) The input bits are $A = 1$, $B = 0$, and $C_{in} = 0$.

$$1 + 0 + 0 = 1 \text{ with no carry}$$

Therefore, $\Sigma = \mathbf{1}$ and $C_{out} = \mathbf{0}$.

(b) The input bits are $A = 1$, $B = 1$, and $C_{in} = 0$.

$$1 + 1 + 0 = 0 \text{ with a carry of 1}$$

Therefore, $\Sigma = \mathbf{0}$ and $C_{out} = \mathbf{1}$.

(c) The input bits are $A = 1$, $B = 0$, and $C_{in} = 1$.

$$1 + 0 + 1 = 0 \text{ with a carry of 1}$$

Therefore, $\Sigma = \mathbf{0}$ and $C_{out} = \mathbf{1}$.

Related Problem*

What are the full-adder outputs for $A = 1$, $B = 1$, and $C_{in} = 1$?

*Answers are at the end of the chapter.

SECTION 6–1 CHECKUP

Answers are at the end of the chapter.

1. Determine the sum (Σ) and the output carry (C_{out}) of a half-adder for each set of input bits:
   (a) 01 **(b)** 00 **(c)** 10 **(d)** 11
2. A full-adder has $C_{in} = 1$. What are the sum (Σ) and the output carry (C_{out}) when $A = 1$ and $B = 1$?

6–2 Parallel Binary Adders

Two or more full-adders are connected to form parallel binary adders. In this section, you will learn the basic operation of this type of adder and its associated input and output functions.

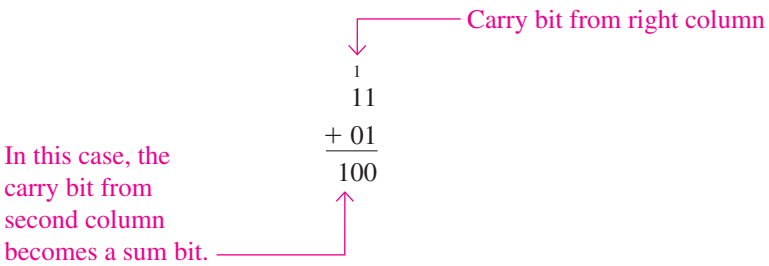
After completing this section, you should be able to

- Use full-adders to implement a parallel binary adder
- Explain the addition process in a parallel binary adder
- Use the truth table for a 4-bit parallel adder
- Apply two 74HC283s for the addition of two 8-bit numbers
- Expand the 4-bit adder to accommodate 8-bit or 16-bit addition
- Use VHDL to describe a 4-bit parallel adder

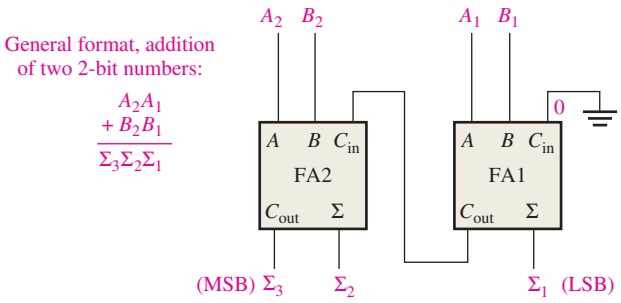
As you learned in Section 6–1, a single full-adder is capable of adding two 1-bit numbers and an input carry. To add binary numbers with more than one bit, you must use additional full-adders. When one binary number is added to another, each column generates a sum bit and a 1 or 0 carry bit to the next column to the left, as illustrated here with 2-bit numbers.

InfoNote

Addition is performed by processors on two numbers at a time, called *operands.* The *source operand* is a number that is to be added to an existing number called the *destination operand,* which is held in an ALU register, such as the accumulator. The sum of the two numbers is then stored back in the accumulator. Addition is performed on integer numbers or floating-point numbers using ADD or FADD instructions respectively.



To add two binary numbers, a full-adder (FA) is required for each bit in the numbers. So for 2-bit numbers, two adders are needed; for 4-bit numbers, four adders are used; and so on. The carry output of each adder is connected to the carry input of the next higher-order adder, as shown in Figure 6–7 for a 2-bit adder. Notice that either a half-adder can be used for the least significant position or the carry input of a full-adder can be made 0 (grounded) because there is no carry input to the least significant bit position.



MultiSim

FIGURE 6–7 Block diagram of a basic 2-bit parallel adder using two full-adders. Open file F06-07 to verify operation.

In Figure 6–7 the least significant bits (LSB) of the two numbers are represented by A_1 and B_1. The next higher-order bits are represented by A_2 and B_2. The three sum bits are Σ_1, Σ_2, and Σ_3. Notice that the output carry from the left-most full-adder becomes the most significant bit (MSB) in the sum, Σ_3.

EXAMPLE 6–2

Determine the sum generated by the 3-bit parallel adder in Figure 6–8 and show the intermediate carries when the binary numbers 101 and 011 are being added.

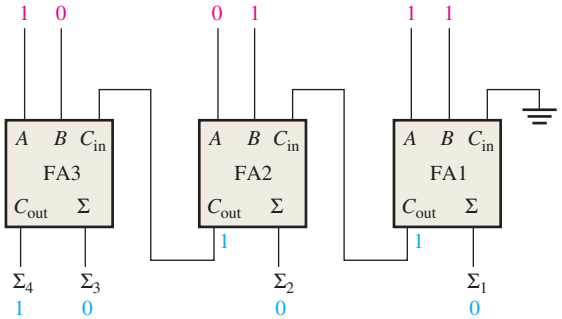


FIGURE 6–8

Solution

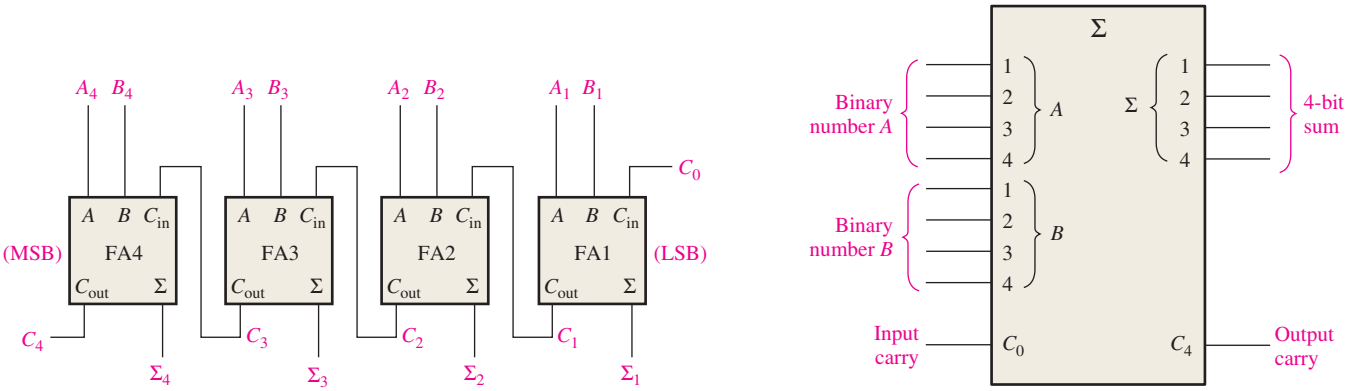
The LSBs of the two numbers are added in the right-most full-adder. The sum bits and the intermediate carries are indicated in blue in Figure 6–8.

Related Problem

What are the sum outputs when 111 and 101 are added by the 3-bit parallel adder?

Four-Bit Parallel Adders

A group of four bits is called a **nibble**. A basic 4-bit parallel adder is implemented with four full-adder stages as shown in Figure 6–9. Again, the LSBs (A_1 and B_1) in each number being added go into the right-most full-adder; the higher-order bits are applied as shown to the successively higher-order adders, with the MSBs (A_4 and B_4) in each number being applied to the left-most full-adder. The carry output of each adder is connected to the carry input of the next higher-order adder as indicated. These are called *internal carries.*



(a) Block diagram

(b) Logic symbol

FIGURE 6–9 A 4-bit parallel adder.

In keeping with most manufacturers' data sheets, the input labeled C_0 is the input carry to the least significant bit adder; C_4, in the case of four bits, is the output carry of the most significant bit adder; and Σ_1 (LSB) through Σ_4 (MSB) are the sum outputs. The logic symbol is shown in Figure 6–9(b).

In terms of the method used to handle carries in a parallel adder, there are two types: the *ripple carry* adder and the *carry look-ahead* adder. These are discussed in Section 6–3.

Truth Table for a 4-Bit Parallel Adder

Table 6–3 is the truth table for a 4-bit adder. On some data sheets, truth tables may be called *function tables* or *functional truth tables.* The subscript n represents the adder bits and can be 1, 2, 3, or 4 for the 4-bit adder. C_{n-1} is the carry from the previous adder. Carries C_1, C_2, and C_3 are generated internally. C_0 is an external carry input and C_4 is an output. Example 6–3 illustrates how to use Table 6–3.

TABLE 6–3

Truth table for each stage of a 4-bit parallel adder.

C_{n-1}	A_n	B_n	Σ_n	C_n
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

EXAMPLE 6–3

Use the 4-bit parallel adder truth table (Table 6–3) to find the sum and output carry for the addition of the following two 4-bit numbers if the input carry (C_{n-1}) is 0:

$$A_4A_3A_2A_1 = 1100 \quad \text{and} \quad B_4B_3B_2B_1 = 1100$$

Solution

For $n = 1$: $A_1 = 0$, $B_1 = 0$, and $C_{n-1} = 0$. From the 1st row of the table,

$$\Sigma_1 = \mathbf{0} \quad \text{and} \quad C_1 = 0$$

For $n = 2$: $A_2 = 0$, $B_2 = 0$, and $C_{n-1} = 0$. From the 1st row of the table,

$$\Sigma_2 = \mathbf{0} \quad \text{and} \quad C_2 = 0$$

For $n = 3$: $A_3 = 1$, $B_3 = 1$, and $C_{n-1} = 0$. From the 4th row of the table,

$$\Sigma_3 = \mathbf{0} \quad \text{and} \quad C_3 = 1$$

For $n = 4$: $A_4 = 1$, $B_4 = 1$, and $C_{n-1} = 1$. From the last row of the table,

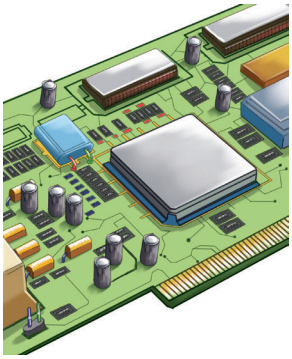
$$\Sigma_4 = \mathbf{1} \quad \text{and} \quad C_4 = \mathbf{1}$$

C_4 becomes the output carry; the sum of 1100 and 1100 is 11000.

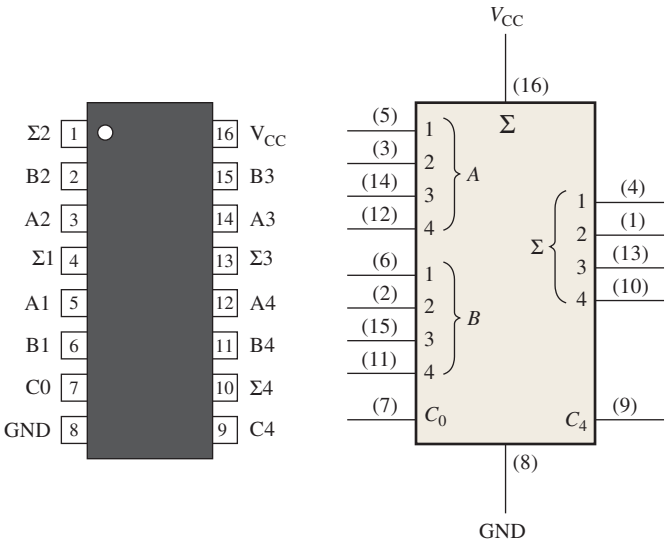
Related Problem

Use the truth table (Table 6–3) to find the result of adding the binary numbers 1011 and 1010.

IMPLEMENTATION: 4-BIT PARALLEL ADDER



Fixed-Function Device The 74HC283 and the 74LS283 are 4-bit parallel adders with identical package pin configurations. The logic symbol and package pin configuration are shown in Figure 6–10. Go to *ti.com* for data sheet information.



(a) Pin diagram (b) Logic symbol

FIGURE 6–10 The 74HC283/74LS283 4-bit parallel adder.

Programmable Logic Device (PLD) A 4-bit adder can be described using VHDL and implemented in a PLD. First, the data flow approach is used to describe the full adder, which is shown in Figure 6–4(b), for use as a component. (Blue text comments are not part of the program.)

```
entity FullAdder is
  port (A, B, CIN: in bit; SUM, COUT: out bit);        Inputs and outputs declared
end entity FullAdder;
```

```
architecture LogicOperation of FullAdder is
begin
SUM <= (A xor B) xor CIN;
COUT <= ((A xor B) and CIN) or (A and B);
end architecture LogicOperation;
```

Boolean expressions for the outputs

Next, the FullAdder program code is used as a component in a VHDL structural approach to the 4-bit full-adder in Figure 6–9(a).

A1-A4: Inputs
B1-B4: Inputs
C0: Carry input
S1-S4: Sum outputs
C4: Carry output

```
entity 4BitFullAdder is
  port (A1, A2, A3, A4, B1, B2, B3, B4, C0: in bit; S1, S2, S3, S4, C4: out bit);
end entity 4BitFullAdder;

architecture LogicOperation of 4BitFullAdder is
  component FullAdder is
    port (A, B, CIN: in bit; SUM, COUT: out bit);
  end component FullAdder;
  signal Cl, C2, C3: bit;
begin
FA1: FullAdder port map (A => A1, B => B1, CIN => C0, SUM => S1, COUT => Cl);
FA2: FullAdder port map (A => A2, B => B2, CIN => C1, SUM => S2, COUT => C2);
FA3: FullAdder port map (A => A3, B => B3, CIN => C2, SUM => S3, COUT => C3);
FA4: FullAdder port map (A => A4, B => B4, CIN => C3, SUM => S4, COUT => C4);
end architecture LogicOperation;
```

Full-adder component declaration

Instantiations for each of the four full adders

Adder Expansion

The 4-bit parallel adder can be expanded to handle the addition of two 8-bit numbers by using two 4-bit adders. The carry input of the low-order adder (C_0) is connected to ground because there is no carry into the least significant bit position, and the carry output of the low-order adder is connected to the carry input of the high-order adder, as shown in Figure 6–11. This process is known as **cascading**. Notice that, in this case, the output carry is designated C_8 because it is generated from the eighth bit position. The low-order adder is

Adders can be expanded to handle more bits by cascading.

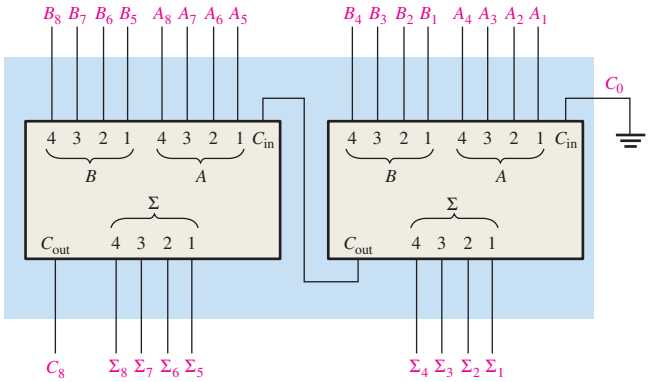


FIGURE 6–11 Cascading of two 4-bit adders to form an 8-bit adder.

the one that adds the lower or less significant four bits in the numbers, and the high-order adder is the one that adds the higher or more significant four bits in the 8-bit numbers. Similarly, four 4-bit adders can be cascaded to handle two 16-bit numbers.

EXAMPLE 6–4

Show how two 74HC283 adders can be connected to form an 8-bit parallel adder. Show output bits for the following 8-bit input numbers:

$$A_8A_7A_6A_5A_4A_3A_2A_1 = 10111001 \quad \text{and} \quad B_8B_7B_6B_5B_4B_3B_2B_1 = 10011110$$

Solution

Two 74HC283 4-bit parallel adders are used to implement the 8-bit adder. The only connection between the two 74HC283s is the carry output (pin 9) of the low-order adder to the carry input (pin 7) of the high-order adder, as shown in Figure 6–12. Pin 7 of the low-order adder is grounded (no carry input).

The sum of the two 8-bit numbers is

$$\Sigma_9\Sigma_8\Sigma_7\Sigma_6\Sigma_5\Sigma_4\Sigma_3\Sigma_2\Sigma_1 = 101010111$$

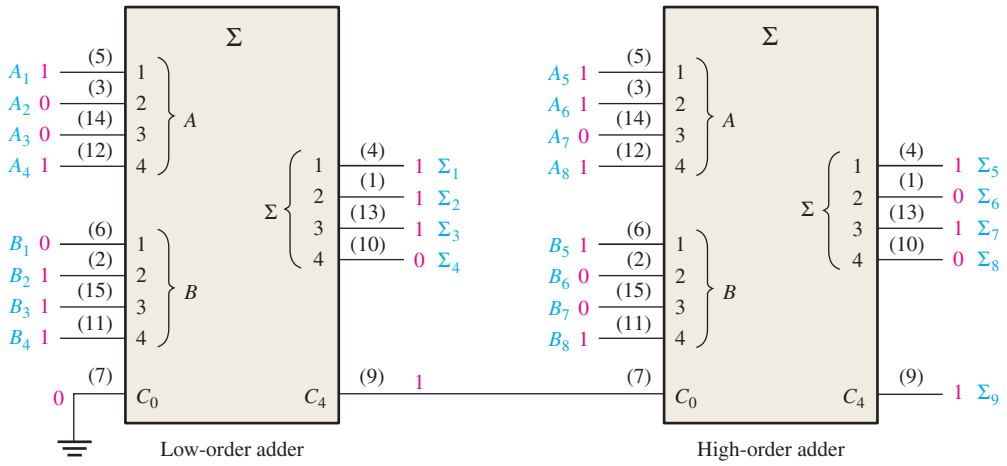


FIGURE 6–12 Two 74HC283 adders connected as an 8-bit parallel adder (pin numbers are in parentheses).

Related Problem

Use 74HC283 adders to implement a 12-bit parallel adder.

An Application

An example of full-adder and parallel adder application is a simple voting system that can be used to simultaneously provide the number of "yes" votes and the number of "no" votes. This type of system can be used where a group of people are assembled and there is a need for immediately determining opinions (for or against), making decisions, or voting on certain issues or other matters.

In its simplest form, the system includes a switch for "yes" or "no" selection at each position in the assembly and a digital display for the number of yes votes and one for the number of no votes. The basic system is shown in Figure 6–13 for a 6-position setup, but it can be expanded to any number of positions with additional 6-position modules and additional parallel adder and display circuits.

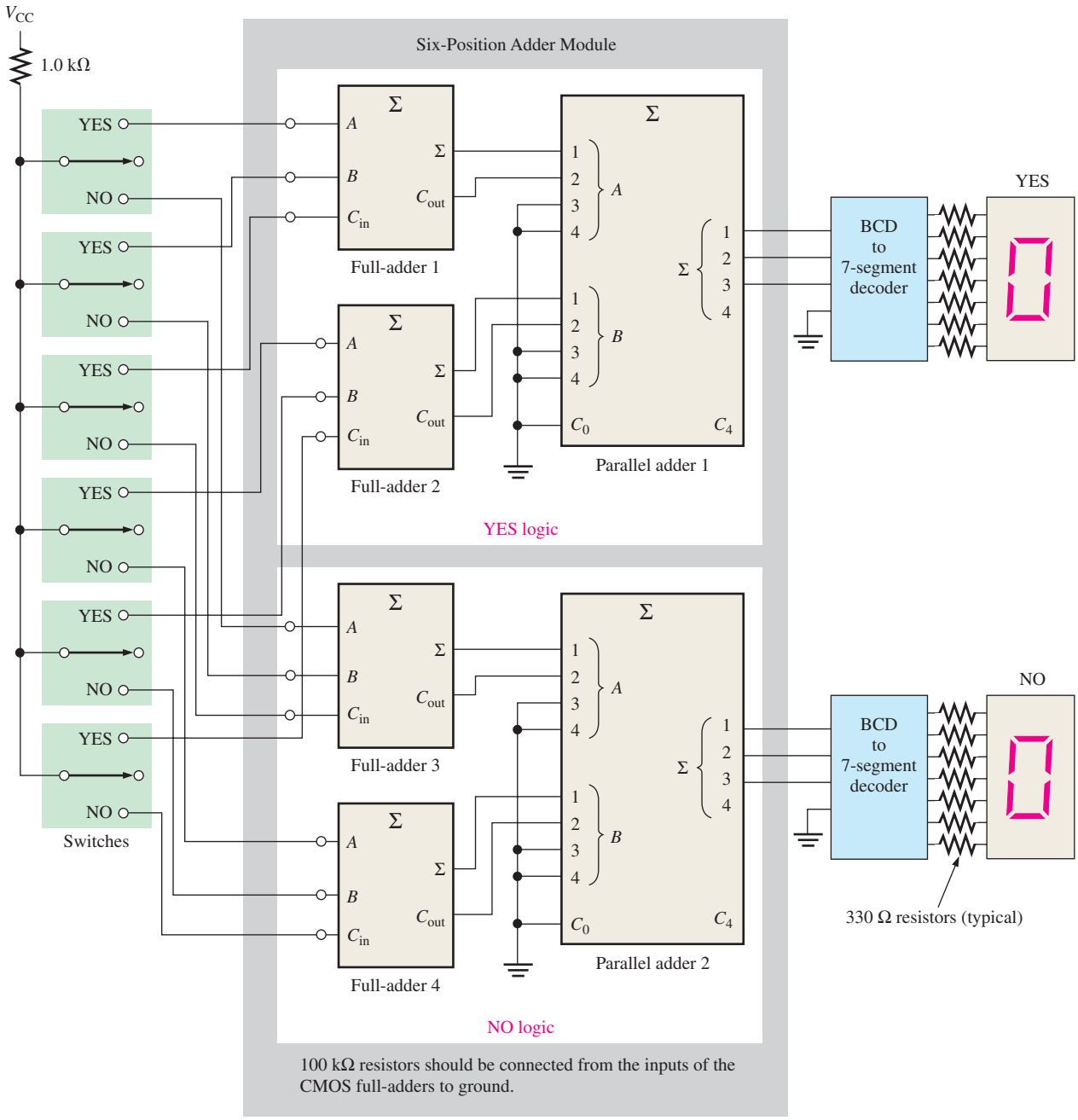


FIGURE 6–13 A voting system using full-adders and parallel binary adders.

In Figure 6–13 each full-adder can produce the sum of up to three votes. The sum and output carry of each full-adder then goes to the two lower-order inputs of a parallel binary adder. The two higher-order inputs of the parallel adder are connected to ground (0) because there is never a case where the binary input exceeds 0011 (decimal 3). For this basic 6-position system, the outputs of the parallel adder go to a BCD-to-7-segment decoder that drives the 7-segment display. As mentioned, additional circuits must be included when the system is expanded.

The resistors from the inputs of each full-adder to ground assure that each input is LOW when the switch is in the neutral position (CMOS logic is used). When a switch is moved to the "yes" or to the "no" position, a HIGH level (V_{CC}) is applied to the associated full-adder input.

SECTION 6–2 CHECKUP

1. Two 4-bit numbers (1101 and 1011) are applied to a 4-bit parallel adder. The input carry is 1. Determine the sum (Σ) and the output carry.
2. How many 74HC283 adders would be required to add two binary numbers each representing decimal numbers up through 1000_{10}?

6–3 Ripple Carry and Look-Ahead Carry Adders

As mentioned in the last section, parallel adders can be placed into two categories based on the way in which internal carries from stage to stage are handled. Those categories are ripple carry and look-ahead carry. Externally, both types of adders are the same in terms of inputs and outputs. The difference is the speed at which they can add numbers. The look-ahead carry adder is much faster than the ripple carry adder.

After completing this section, you should be able to

- Discuss the difference between a ripple carry adder and a look-ahead carry adder
- State the advantage of look-ahead carry addition
- Define *carry generation* and *carry propagation* and explain the difference
- Develop look-ahead carry logic
- Explain why cascaded 74HC283s exhibit both ripple carry and look-ahead carry properties

The Ripple Carry Adder

A **ripple carry** adder is one in which the carry output of each full-adder is connected to the carry input of the next higher-order stage (a stage is one full-adder). The sum and the output carry of any stage cannot be produced until the input carry occurs; this causes a time delay in the addition process, as illustrated in Figure 6–14. The carry propagation delay for each full-adder is the time from the application of the input carry until the output carry occurs, assuming that the *A* and *B* inputs are already present.

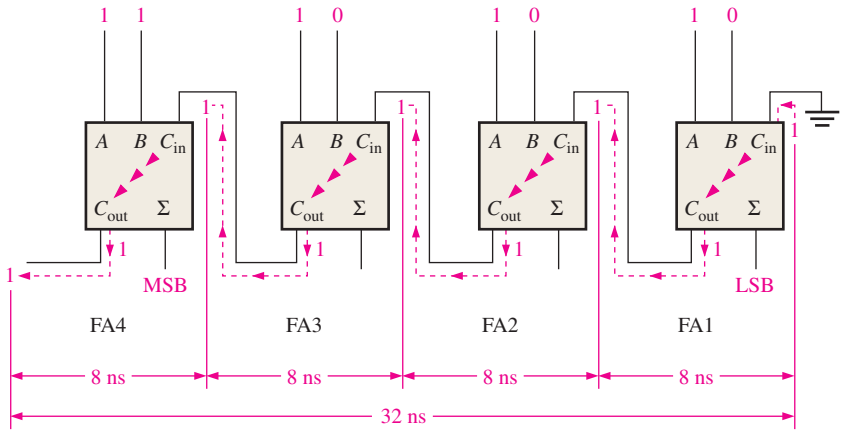


FIGURE 6–14 A 4-bit parallel ripple carry adder showing "worst-case" carry propagation delays.

Full-adder 1 (FA1) cannot produce a potential output carry until an input carry is applied. Full-adder 2 (FA2) cannot produce a potential output carry until FA1 produces an output carry. Full-adder 3 (FA3) cannot produce a potential output carry until an output

carry is produced by FA1 followed by an output carry from FA2, and so on. As you can see in Figure 6–14, the input carry to the least significant stage has to ripple through all the adders before a final sum is produced. The cumulative delay through all the adder stages is a "worst-case" addition time. The total delay can vary, depending on the carry bit produced by each full-adder. If two numbers are added such that no carries (0) occur between stages, the addition time is simply the propagation time through a single full-adder from the application of the data bits on the inputs to the occurrence of a sum output; however, worst-case addition time must always be assumed.

The Look-Ahead Carry Adder

The speed with which an addition can be performed is limited by the time required for the carries to propagate, or ripple, through all the stages of a parallel adder. One method of speeding up the addition process by eliminating this ripple carry delay is called **look-ahead carry** addition. The look-ahead carry adder anticipates the output carry of each stage, and based on the inputs, produces the output carry by either carry generation or carry propagation.

Carry generation occurs when an output carry is produced (generated) internally by the full-adder. A carry is generated only when both input bits are 1s. The generated carry, C_g, is expressed as the AND function of the two input bits, A and B.

$$C_g = AB \qquad \text{Equation 6–5}$$

Carry propagation occurs when the input carry is rippled to become the output carry. An input carry may be propagated by the full-adder when either or both of the input bits are 1s. The propagated carry, C_p, is expressed as the OR function of the input bits.

$$C_p = A + B \qquad \text{Equation 6–6}$$

The conditions for carry generation and carry propagation are illustrated in Figure 6–15. The three arrowheads symbolize ripple (propagation).

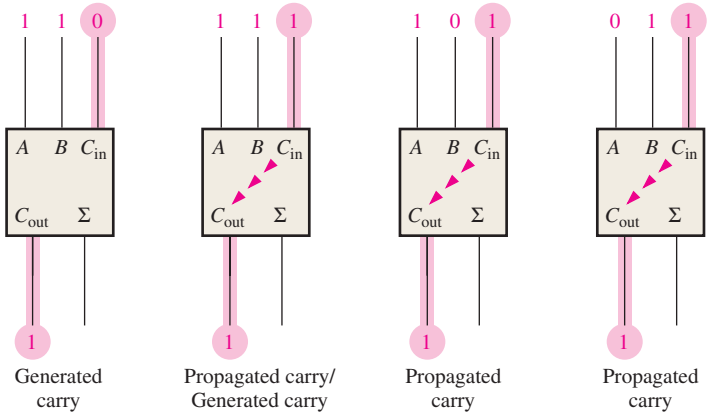


FIGURE 6–15 Illustration of conditions for carry generation and carry propagation.

The output carry of a full-adder can be expressed in terms of both the generated carry (C_g) and the propagated carry (C_p). The output carry (C_{out}) is a 1 if the generated carry is a 1 OR if the propagated carry is a 1 AND the input carry (C_{in}) is a 1. In other words, we get an output carry of 1 if it is generated by the full-adder ($A = 1$ AND $B = 1$) or if the adder propagates the input carry ($A = 1$ OR $B = 1$) AND $C_{in} = 1$. This relationship is expressed as

$$C_{out} = C_g + C_pC_{in} \qquad \text{Equation 6–7}$$

Now let's see how this concept can be applied to a parallel adder, whose individual stages are shown in Figure 6–16 for a 4-bit example. For each full-adder, the output carry is

dependent on the generated carry (C_g), the propagated carry (C_p), and its input carry (C_{in}). The C_g and C_p functions for each stage are *immediately* available as soon as the input bits A and B and the input carry to the LSB adder are applied because they are dependent only on these bits. The input carry to each stage is the output carry of the previous stage.

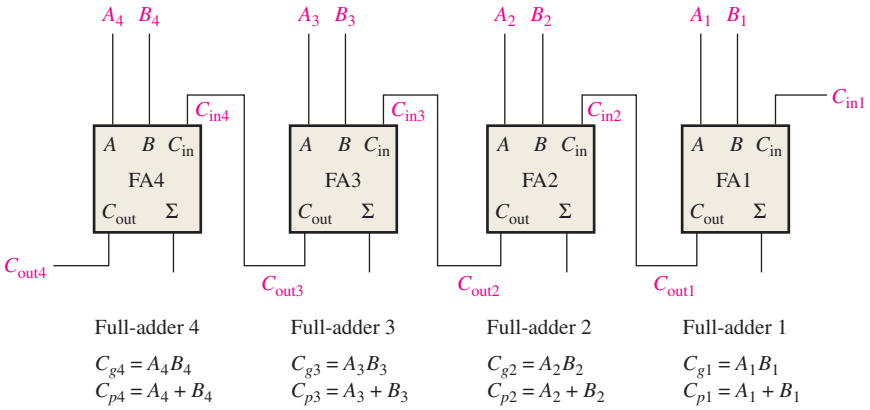


FIGURE 6–16 Carry generation and carry propagation in terms of the input bits to a 4-bit adder.

Based on this analysis, we can now develop expressions for the output carry, C_{out}, of each full-adder stage for the 4-bit example.

Full-adder 1:

$$C_{out1} = C_{g1} + C_{p1}C_{in1}$$

Full-adder 2:

$$C_{in2} = C_{out1}$$

$$\begin{aligned} C_{out2} &= C_{g2} + C_{p2}C_{in2} = C_{g2} + C_{p2}C_{out1} = C_{g2} + C_{p2}(C_{g1} + C_{p1}C_{in1}) \\ &= C_{g2} + C_{p2}C_{g1} + C_{p2}C_{p1}C_{in1} \end{aligned}$$

Full-adder 3:

$$C_{in3} = C_{out2}$$

$$\begin{aligned} C_{out3} &= C_{g3} + C_{p3}C_{in3} = C_{g3} + C_{p3}C_{out2} = C_{g3} + C_{p3}(C_{g2} + C_{p2}C_{g1} + C_{p2}C_{p1}C_{in1}) \\ &= C_{g3} + C_{p3}C_{g2} + C_{p3}C_{p2}C_{g1} + C_{p3}C_{p2}C_{p1}C_{in1} \end{aligned}$$

Full-adder 4:

$$C_{in4} = C_{out3}$$

$$\begin{aligned} C_{out4} &= C_{g4} + C_{p4}C_{in4} = C_{g4} + C_{p4}C_{out3} \\ &= C_{g4} + C_{p4}(C_{g3} + C_{p3}C_{g2} + C_{p3}C_{p2}C_{g1} + C_{p3}C_{p2}C_{p1}C_{in1}) \\ &= C_{g4} + C_{p4}C_{g3} + C_{p4}C_{p3}C_{g2} + C_{p4}C_{p3}C_{p2}C_{g1} + C_{p4}C_{p3}C_{p2}C_{p1}C_{in1} \end{aligned}$$

Notice that in each of these expressions, the output carry for each full-adder stage is dependent only on the initial input carry (C_{in1}), the C_g and C_p functions of that stage, and the C_g and C_p functions of the preceding stages. Since each of the C_g and C_p functions can be expressed in terms of the A and B inputs to the full-adders, all the output carries are immediately available (except for gate delays), and you do not have to wait for a carry to ripple through all the stages before a final result is achieved. Thus, the look-ahead carry technique speeds up the addition process.

The C_{out} equations are implemented with logic gates and connected to the full-adders to create a 4-bit look-ahead carry adder, as shown in Figure 6–17.

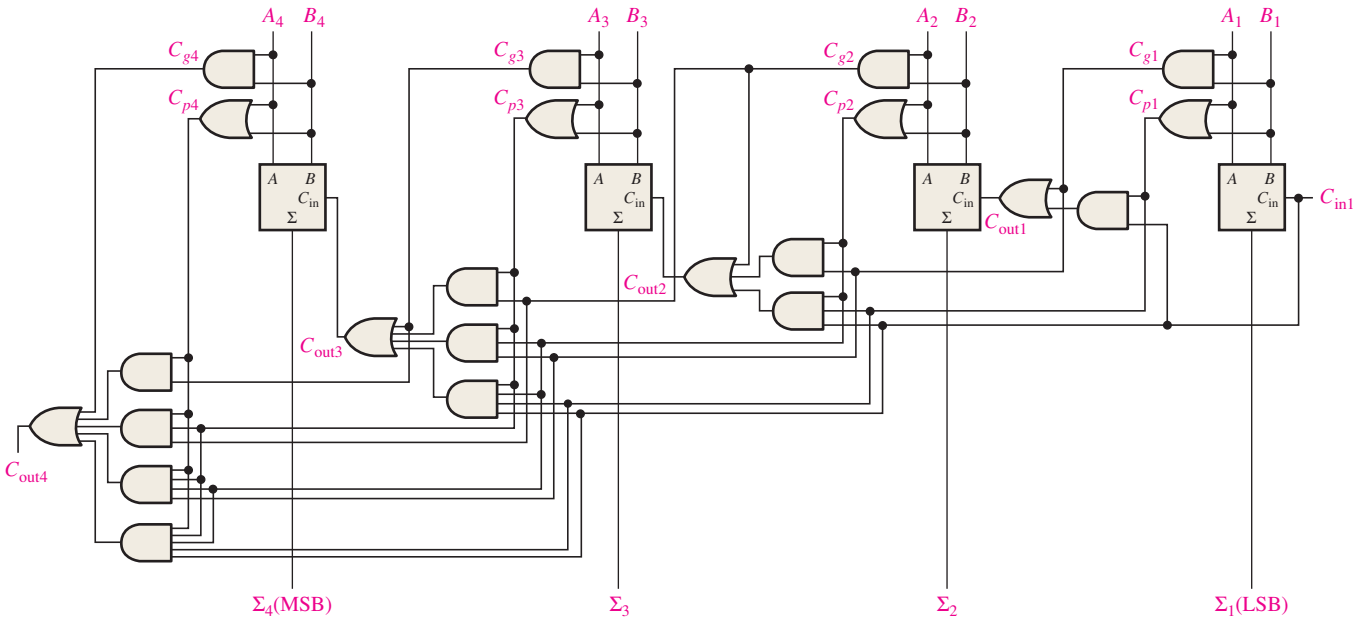


FIGURE 6–17 Logic diagram for a 4-stage look-ahead carry adder.

Combination Look-Ahead and Ripple Carry Adders

As with most fixed-function IC adders, the 74HC283 4-bit adder that was introduced in Section 6–2 is a look-ahead carry adder. When these adders are cascaded to expand their capability to handle binary numbers with more than four bits, the output carry of one adder is connected to the input carry of the next. This creates a ripple carry condition between the 4-bit adders so that when two or more 74HC283s are cascaded, the resulting adder is actually a combination look-ahead and ripple carry adder. The look-ahead carry operation is internal to each MSI adder and the ripple carry feature comes into play when there is a carry out of one of the adders to the next one.

SECTION 6–3 CHECKUP

1. The input bits to a full-adder are $A = 1$ and $B = 0$. Determine C_g and C_p.
2. Determine the output carry of a full-adder when $C_{in} = 1$, $C_g = 0$, and $C_p = 1$.

6–4 Comparators

The basic function of a **comparator** is to compare the magnitudes of two binary quantities to determine the relationship of those quantities. In its simplest form, a comparator circuit determines whether two numbers are equal.

After completing this section, you should be able to

- Use the exclusive-NOR gate as a basic comparator
- Analyze the internal logic of a magnitude comparator that has both equality and inequality outputs
- Apply the 74HC85 comparator to compare the magnitudes of two 4-bit numbers
- Cascade 74HC85s to expand a comparator to eight or more bits
- Use VHDL to describe a 4-bit magnitude comparator

Equality

As you learned in Chapter 3, the exclusive-NOR gate can be used as a basic comparator because its output is a 0 if the two input bits are not equal and a 1 if the input bits are equal. Figure 6–18 shows the exclusive-NOR gate as a 2-bit comparator.

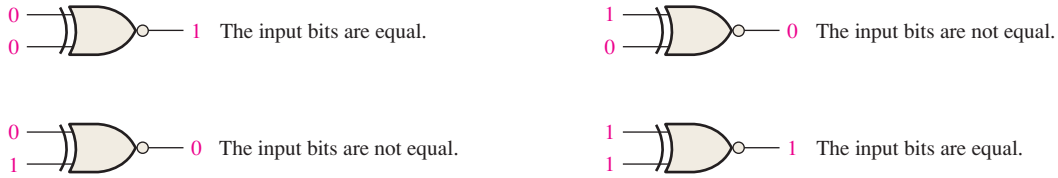
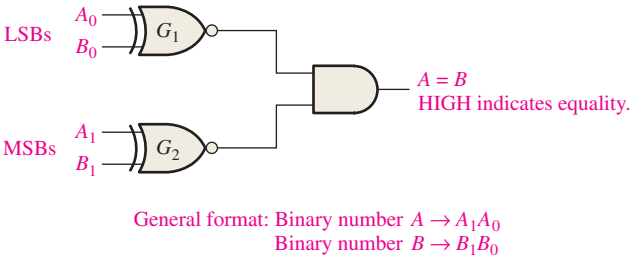


FIGURE 6–18 Basic comparator operation.

In order to compare binary numbers containing two bits each, an additional exclusive-NOR gate is necessary. The two least significant bits (LSBs) of the two numbers are compared by gate G_1, and the two most significant bits (MSBs) are compared by gate G_2, as shown in Figure 6–19. If the two numbers are equal, their corresponding bits are the same, and the output of each exclusive-NOR gate is a 1. If the corresponding sets of bits are not equal, a 0 occurs on that exclusive-NOR gate output.



MultiSim

FIGURE 6–19 Logic diagram for equality comparison of two 2-bit numbers. Open file F06-19 to verify operation.

In order to produce a single output indicating an equality or inequality of two numbers, an AND gate can be combined with XNOR gates, as shown in Figure 6–19. The output of each exclusive-NOR gate is applied to the AND gate input. When the two input bits for each exclusive-NOR are equal, the corresponding bits of the numbers are equal, producing a 1 on both inputs to the AND gate and thus a 1 on the output. When the two numbers are not equal, one or both sets of corresponding bits are unequal, and a 0 appears on at least one input to the AND gate to produce a 0 on its output. Thus, the output of the AND gate indicates equality (1) or inequality (0) of the two numbers. Example 6–5 illustrates this operation for two specific cases.

A comparator determines if two binary numbers are equal or unequal.

EXAMPLE 6–5

Apply each of the following sets of binary numbers to the comparator inputs in Figure 6–20, and determine the output by following the logic levels through the circuit.

(a) 10 and 10 **(b)** 11 and 10

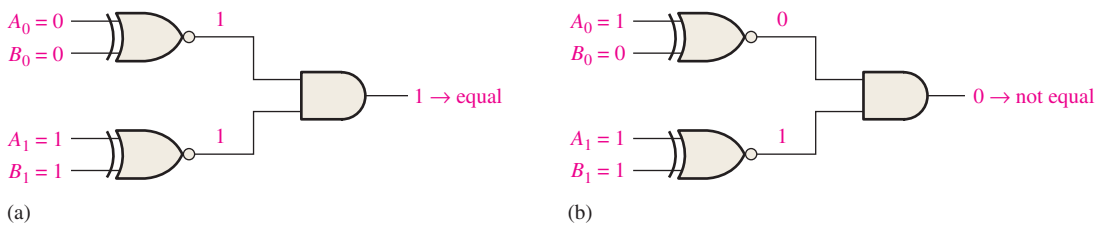


FIGURE 6–20

Solution

(a) The output is **1** for inputs 10 and 10, as shown in Figure 6–20(a).

(b) The output is **0** for inputs 11 and 10, as shown in Figure 6–20(b).

Related Problem

Repeat the process for binary inputs of 01 and 10.

As you know from Chapter 3, the basic comparator can be expanded to any number of bits. The AND gate sets the condition that all corresponding bits of the two numbers must be equal if the two numbers themselves are equal.

Inequality

In addition to the equality output, fixed-function comparators can provide additional outputs that indicate which of the two binary numbers being compared is the larger. That is, there is an output that indicates when number A is greater than number B ($A > B$) and an output that indicates when number A is less than number B ($A < B$), as shown in the logic symbol for a 4-bit comparator in Figure 6–21.

To determine an inequality of binary numbers A and B, you first examine the highest-order bit in each number. The following conditions are possible:

1. If $A_3 = 1$ and $B_3 = 0$, number A is greater than number B.
2. If $A_3 = 0$ and $B_3 = 1$, number A is less than number B.
3. If $A_3 = B_3$, then you must examine the next lower bit position for an inequality.

These three operations are valid for each bit position in the numbers. The general procedure used in a comparator is to check for an inequality in a bit position, starting with the highest-order bits (MSBs). When such an inequality is found, the relationship of the two numbers is established, and any other inequalities in lower-order bit positions must be ignored because it is possible for an opposite indication to occur; *the highest-order indication must take precedence.*

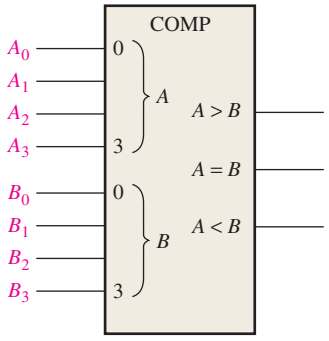


FIGURE 6–21 Logic symbol for a 4-bit comparator with inequality indication.

InfoNote

In a computer, the *cache* is a very fast intermediate memory between the central processing unit (CPU) and the slower main memory. The CPU requests data by sending out its *address* (unique location) in memory. Part of this address is called a *tag.* The *tag address comparator* compares the tag from the CPU with the tag from the cache directory. If the two agree, the addressed data is already in the cache and is retrieved very quickly. If the tags disagree, the data must be retrieved from the main memory at a much slower rate.

EXAMPLE 6–6

Determine the $A = B$, $A > B$, and $A < B$ outputs for the input numbers shown on the comparator in Figure 6–22.

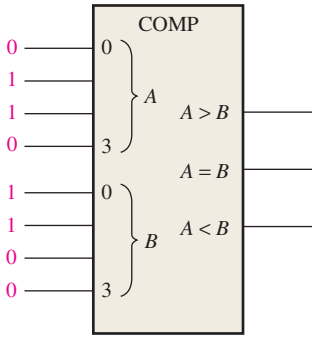


FIGURE 6–22

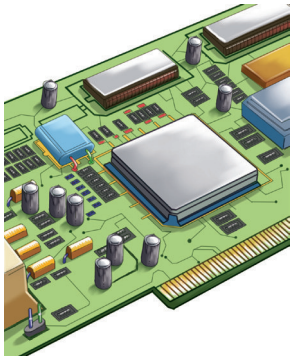
Solution

The number on the A inputs is 0110 and the number on the B inputs is 0011. The **$A > B$ output is HIGH and the other outputs are LOW**.

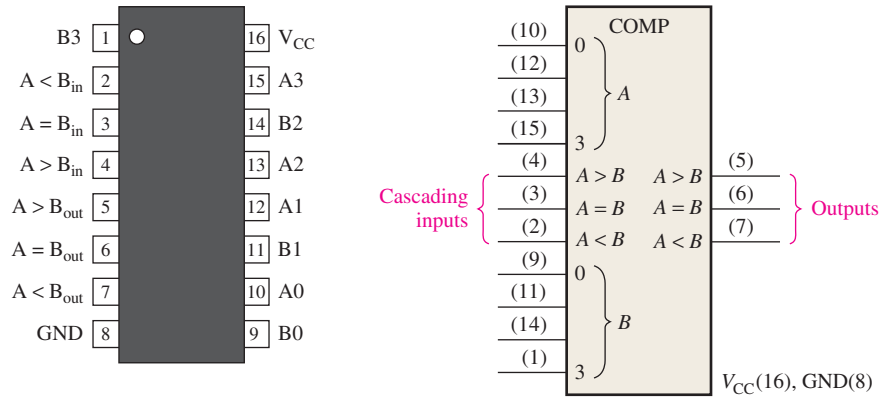
Related Problem

What are the comparator outputs when $A_3A_2A_1A_0 = 1001$ and $B_3B_2B_1B_0 = 1010$?

IMPLEMENTATION: 4-BIT MAGNITUDE COMPARATOR



Fixed-Function Device The 74HC85/74LS85 pin diagram and logic symbol are shown in Figure 6–23. Notice that this device has all the inputs and outputs of the generalized comparator previously discussed and, in addition, has three cascading inputs: $A < B, A = B, A > B$. These inputs allow several comparators to be cascaded for comparison of any number of bits greater than four. To expand the comparator, the $A < B$, $A = B$, and $A > B$ outputs of the lower-order comparator are connected to the corresponding cascading inputs of the next higher-order comparator. The lowest-order comparator must have a HIGH on the $A = B$ input and LOWs on the $A < B$ and $A > B$ inputs.



(a) Pin diagram (b) Logic symbol

FIGURE 6–23 The 74HC85/74LS85 4-bit magnitude comparator.

Programmable Logic Device (PLD) A 4-bit magnitude comparator can be described using VHDL and implemented in a PLD. The following VHDL program uses the data flow approach to implement a simplified comparator ($A = B$ output only) in Figure 6–24. (The blue comments are not part of the program.)

```
entity 4BitComparator is
   port (A0, A1, A2, A3, B0, B1, B2, B3: in bit; AequalB: out bit);
end entity 4BitComparator;                          Inputs and outputs declared

architecture LogicOperation of 4BitComparator is
begin
AequalB <= (A0 xnor B0) and (A1 xnor B1) and        } Output in terms of a
(A2 xnor B2) and (A3 xnor B);                       } Boolean expression

end architecture LogicOperation;
```

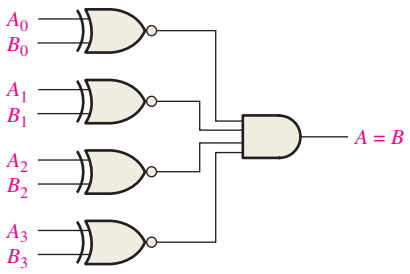


FIGURE 6–24

EXAMPLE 6–7

Use 74HC85 comparators to compare the magnitudes of two 8-bit numbers. Show the comparators with proper interconnections.

Solution

Two 74HC85s are required to compare two 8-bit numbers. They are connected as shown in Figure 6–25 in a cascaded arrangement.

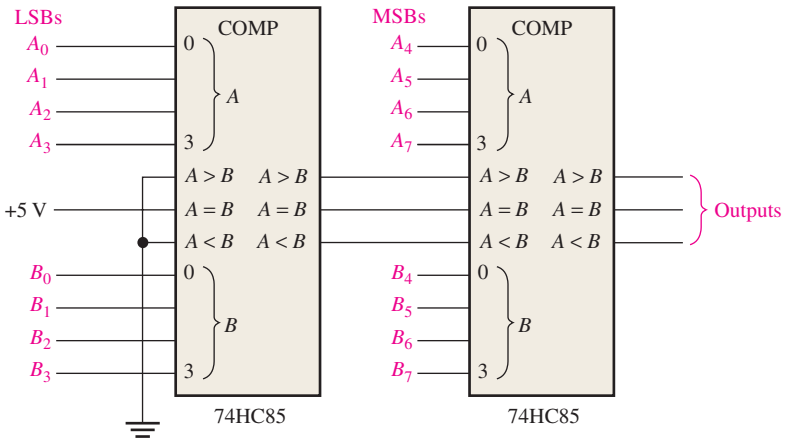


FIGURE 6–25 An 8-bit magnitude comparator using two 74HC85s.

Related Problem

Expand the circuit in Figure 6–25 to a 16-bit comparator.

Most CMOS devices contain protection circuitry to guard against damage from high static voltages or electric fields. However, precautions must be taken to avoid applications of any voltages higher than maximum rated voltages. For proper operation, input and output voltages should be between ground and V_{CC}. Also, remember that unused inputs must always be connected to an appropriate logic level (ground or V_{CC}). Unused outputs may be left open.

SECTION 6–4 CHECKUP

1. The binary numbers $A = 1011$ and $B = 1010$ are applied to the inputs of a 74HC85. Determine the outputs.
2. The binary numbers $A = 11001011$ and $B = 11010100$ are applied to the 8-bit comparator in Figure 6–25. Determine the states of the outputs on each comparator.

6–5 Decoders

A **decoder** is a digital circuit that detects the presence of a specified combination of bits (code) on its inputs and indicates the presence of that code by a specified output level. In its general form, a decoder has n input lines to handle n bits and from one to 2^n output lines to indicate the presence of one or more n-bit combinations. In this section, three fixed-function IC decoders are introduced. The basic principles can be extended to other types of decoders.

After completing this section, you should be able to

- Define *decoder*
- Design a logic circuit to decode any combination of bits
- Describe the 74HC154 binary-to-decimal decoder
- Expand decoders to accommodate larger numbers of bits in a code
- Describe the 74HC42 BCD-to-decimal decoder
- Describe the 74HC47 BCD-to-7-segment decoder
- Discuss zero suppression in 7-segment displays
- Use VHDL to describe various types of decoders
- Apply decoders to specific applications

The Basic Binary Decoder

Suppose you need to determine when a binary 1001 occurs on the inputs of a digital circuit. An AND gate can be used as the basic decoding element because it produces a HIGH output only when all of its inputs are HIGH. Therefore, you must make sure that all of the inputs to the AND gate are HIGH when the binary number 1001 occurs; this can be done by inverting the two middle bits (the 0s), as shown in Figure 6–26.

InfoNote

An *instruction* tells the processor what operation to perform. Instructions are in machine code (1s and 0s) and, in order for the processor to carry out an instruction, the instruction must be decoded. Instruction decoding is one of the steps in instruction *pipelining*, which are as follows: Instruction is read from the memory (instruction fetch), instruction is decoded, operand(s) is (are) read from memory (operand fetch), instruction is executed, and result is written back to memory. Basically, pipelining allows the next instruction to begin processing before the current one is completed.

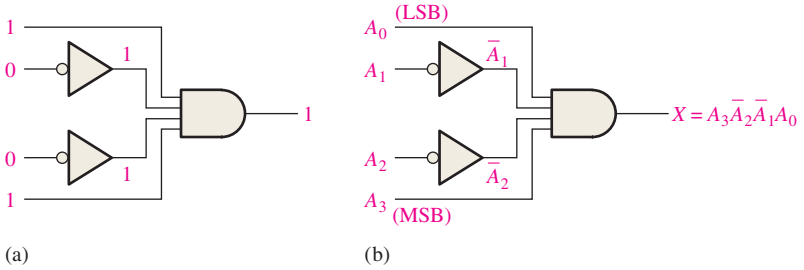


FIGURE 6–26 Decoding logic for the binary code 1001 with an active-HIGH output.

The logic equation for the decoder of Figure 6–26(a) is developed as illustrated in Figure 6–26(b). You should verify that the output is 0 except when $A_0 = 1$, $A_1 = 0$, $A_2 = 0$, and $A_3 = 1$ are applied to the inputs. A_0 is the LSB and A_3 is the MSB. *In the representation of a binary number or other weighted code in this book, the LSB is the right-most bit in a horizontal arrangement and the topmost bit in a vertical arrangement, unless specified otherwise.*

If a NAND gate is used in place of the AND gate in Figure 6–26, a LOW output will indicate the presence of the proper binary code, which is 1001 in this case.

EXAMPLE 6–8

Determine the logic required to decode the binary number 1011 by producing a HIGH level on the output.

Solution

The decoding function can be formed by complementing only the variables that appear as 0 in the desired binary number, as follows:

$$X = A_3\overline{A}_2A_1A_0 \qquad (1011)$$

This function can be implemented by connecting the true (uncomplemented) variables A_0, A_1, and A_3 directly to the inputs of an AND gate, and inverting the variable A_2 before applying it to the AND gate input. The decoding logic is shown in Figure 6–27.

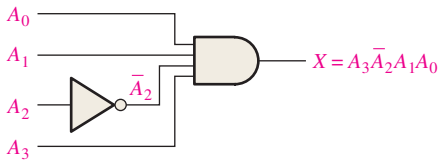


FIGURE 6–27 Decoding logic for producing a HIGH output when 1011 is on the inputs.

Related Problem

Develop the logic required to detect the binary code 10010 and produce an active-LOW output.

The 4-Bit Decoder

In order to decode all possible combinations of four bits, sixteen decoding gates are required ($2^4 = 16$). This type of decoder is commonly called either a *4-line-to-16-line decoder* because there are four inputs and sixteen outputs or a *1-of-16 decoder* because for any given code on the inputs, one of the sixteen outputs is activated. A list of the sixteen binary codes and their corresponding decoding functions is given in Table 6–4.

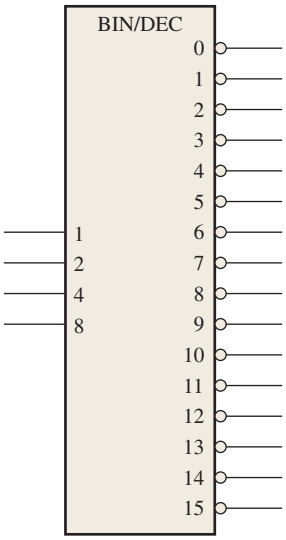
TABLE 6–4

Decoding functions and truth table for a 4-line-to-16-line (1-of-16) decoder with active-LOW outputs.

Decimal Digit	Binary Inputs				Decoding Function	Outputs														
	A_3	A_2	A_1	A_0		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	$\overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	$\overline{A}_3\overline{A}_2\overline{A}_1A_0$	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	0	$\overline{A}_3\overline{A}_2A_1\overline{A}_0$	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	0	1	1	$\overline{A}_3\overline{A}_2A_1A_0$	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
4	0	1	0	0	$\overline{A}_3A_2\overline{A}_1\overline{A}_0$	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
5	0	1	0	1	$\overline{A}_3A_2\overline{A}_1A_0$	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
6	0	1	1	0	$\overline{A}_3A_2A_1\overline{A}_0$	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
7	0	1	1	1	$\overline{A}_3A_2A_1A_0$	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
8	1	0	0	0	$A_3\overline{A}_2\overline{A}_1\overline{A}_0$	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
9	1	0	0	1	$A_3\overline{A}_2\overline{A}_1A_0$	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
10	1	0	1	0	$A_3\overline{A}_2A_1\overline{A}_0$	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
11	1	0	1	1	$A_3\overline{A}_2A_1A_0$	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
12	1	1	0	0	$A_3A_2\overline{A}_1\overline{A}_0$	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
13	1	1	0	1	$A_3A_2\overline{A}_1A_0$	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
14	1	1	1	0	$A_3A_2A_1\overline{A}_0$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
15	1	1	1	1	$A_3A_2A_1A_0$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

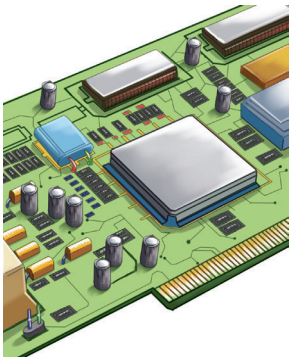
If an active-LOW output is required for each decoded number, the entire decoder can be implemented with NAND gates and inverters. In order to decode each of the sixteen binary codes, sixteen NAND gates are required (AND gates can be used to produce active-HIGH outputs).

A logic symbol for a 4-line-to-16-line (1-of-16) decoder with active-LOW outputs is shown in Figure 6–28. The BIN/DEC label indicates that a binary input makes the corresponding decimal output active. The input labels 8, 4, 2, and 1 represent the binary weights of the input bits ($2^3 2^2 2^1 2^0$).

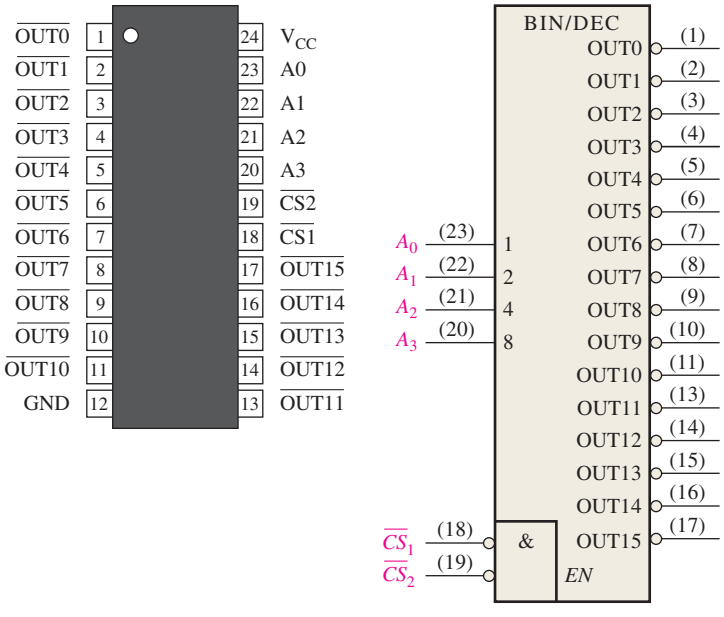


MultiSim **FIGURE 6–28** Logic symbol for a 4-line-to-16-line (1-of-16) decoder. Open file F06-28 to verify operation.

IMPLEMENTATION: 1-OF-16 DECODER



Fixed-Function Device The 74HC154 is a good example of a fixed-function IC decoder. The logic symbol is shown in Figure 6–29. There is an enable function (*EN*) provided on this device, which is implemented with a NOR gate used as a negative-AND. A LOW level on each chip select input, $\overline{CS}_1$ and $\overline{CS}_2$, is required in order to make the enable gate output (*EN*) HIGH. The enable gate output is connected to an input of *each* NAND gate in the decoder, so it must be HIGH for the NAND gates to be enabled. If the enable gate is not activated by a LOW on both inputs, then all sixteen decoder outputs (*OUT*) will be HIGH regardless of the states of the four input variables, A_0, A_1, A_2, and A_3.



(a) Pin diagram (b) Logic symbol

FIGURE 6–29 The 74HC154 1-of-16 decoder.

Programmable Logic Device (PLD) The 1-of-16 decoder can be described using VHDL and implemented as hardware in a PLD. The decoder consists of sixteen 5-input NAND gates for decoding, a 2-input negative-AND for the enable function, and four inverters. The following VHDL program code uses the data flow approach. (Blue text comments are not part of the program.)

```
entity 1of16Decoder is
  port (A0, A1, A2, A3, CS1, CS2: in bit; OUT0, OUT1, OUT2,
  OUT3, OUT4, OUT5, OUT6, OUT7, OUT8, OUT9, OUT10,
  OUT11, OUT12, OUT13, OUT14, OUT15: out bit);
end entity 1of16Decoder;

architecture LogicOperation of 1of16Decoder is
signal EN: bit;
begin
  OUT0 <= not(not A0 and not A1 and not A2 and not A3 and EN);
  OUT1 <= not(A0 and not A1 and not A2 and not A3 and EN);
  OUT2 <= not(not A0 and A1 and not A2 and not A3 and EN);
  OUT3 <= not(A0 and A1 and not A2 and not A3 and EN);
  OUT4 <= not(not A0 and not A1 and A2 and not A3 and EN);
  OUT5 <= not(A0 and not A1 and A2 and not A3 and EN);
  OUT6 <= not(not A0 and A1 and A2 and not A3 and EN);
  OUT7 <= not(A0 and A1 and A2 and not A3 and EN);
  OUT8 <= not(not A0 and not A1 and not A2 and A3 and EN);
  OUT9 <= not(A0 and not A1 and not A2 and A3 and EN);
  OUT10 <= not(not A0 and A1 and not A2 and A3 and EN);
  OUT11 <= not(A0 and A1 and not A2 and A3 and EN);
  OUT12 <= not(not A0 and not A1 and A2 and A3 and EN);
  OUT13 <= not(A0 and not A1 and A2 and A3 and EN);
  OUT14 <= not(not A0 and A1 and A2 and A3 and EN);
  OUT15 <= not(A0 and A1 and A2 and A3 and EN);
  EN <= not CS1 and not CS2;
end architecture LogicOperation;
```

Inputs and outputs declared

Boolean expressions for the sixteen outputs

EXAMPLE 6–9

A certain application requires that a 5-bit number be decoded. Use 74HC154 decoders to implement the logic. The binary number is represented by the format $A_4A_3A_2A_1A_0$.

Solution

Since the 74HC154 can handle only four bits, two decoders must be used to form a 5-bit expansion. The fifth bit, A_4, is connected to the chip select inputs, $\overline{CS}_1$ and $\overline{CS}_2$, of one decoder, and $\overline{A}_4$ is connected to the $\overline{CS}_1$ and $\overline{CS}_2$ inputs of the other decoder, as shown in Figure 6–30. When the decimal number is 15 or less, $A_4 = 0$, the low-order decoder is enabled, and the high-order decoder is disabled. When the decimal number is greater than 15, $A_4 = 1$ so $\overline{A}_4 = 0$, the high-order decoder is enabled, and the low-order decoder is disabled.

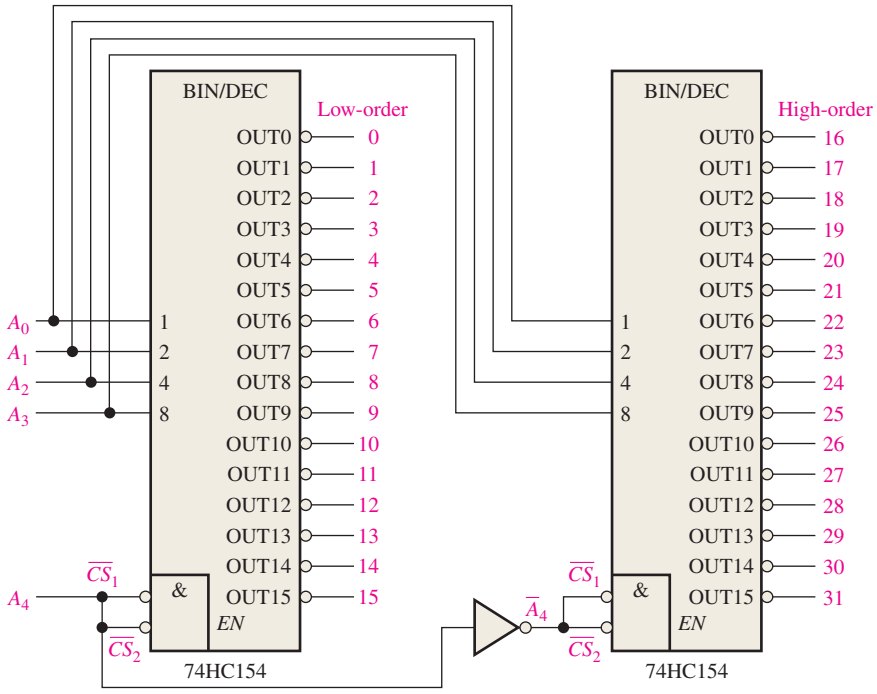


FIGURE 6–30 A 5-bit decoder using 74HC154s.

Related Problem

Determine the output in Figure 6–30 that is activated for the binary input 10110.

The BCD-to-Decimal Decoder

The BCD-to-decimal decoder converts each BCD code (8421 code) into one of ten possible decimal digit indications. It is frequently referred as a *4-line-to-10-line decoder* or a *1-of-10 decoder.*

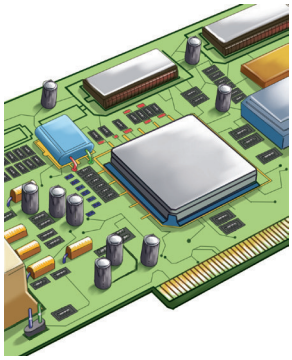
The method of implementation is the same as for the 1-of-16 decoder previously discussed, except that only ten decoding gates are required because the BCD code represents only the ten decimal digits 0 through 9. A list of the ten BCD codes and their corresponding decoding functions is given in Table 6–5. Each of these decoding functions is implemented with NAND gates to provide active-LOW outputs. If an active-HIGH output is required, AND gates are used for decoding. The logic is identical to that of the first ten decoding gates in the 1-of-16 decoder (see Table 6–4).

TABLE 6–5

BCD decoding functions.

Decimal Digit	BCD Code A_3	A_2	A_1	A_0	Decoding Function
0	0	0	0	0	$\overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$
1	0	0	0	1	$\overline{A}_3\overline{A}_2\overline{A}_1A_0$
2	0	0	1	0	$\overline{A}_3\overline{A}_2A_1\overline{A}_0$
3	0	0	1	1	$\overline{A}_3\overline{A}_2A_1A_0$
4	0	1	0	0	$\overline{A}_3A_2\overline{A}_1\overline{A}_0$
5	0	1	0	1	$\overline{A}_3A_2\overline{A}_1A_0$
6	0	1	1	0	$\overline{A}_3A_2A_1\overline{A}_0$
7	0	1	1	1	$\overline{A}_3A_2A_1A_0$
8	1	0	0	0	$A_3\overline{A}_2\overline{A}_1\overline{A}_0$
9	1	0	0	1	$A_3\overline{A}_2\overline{A}_1A_0$

IMPLEMENTATION: BCD-TO-DECIMAL DECODER



Fixed-Function Device The 74HC42 is a fixed-function IC decoder with four BCD inputs and ten active-LOW decimal outputs. The logic symbol is shown in Figure 6–31.

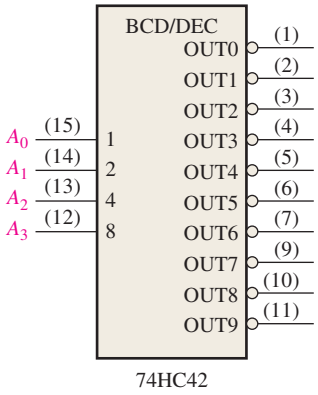


FIGURE 6–31 The 74HC42 BCD-to-decimal decoder.

Programmable Logic Device (PLD) The logic of the BCD-to-decimal decoder is similar to the 1-of-16 decoder except simpler. In this case, there are ten gates and four inverters instead of sixteen gates and four inverters. This decoder does not have an enable function. Using the data flow approach, the VHDL program code for the 1-of-16 decoder can be simplified to implement the BCD-to-decimal decoder.

```
entity BCDdecoder is
  port (A0, A1, A2, A3: in bit; OUT0, OUT1, OUT2, OUT3,
  OUT4, OUT5, OUT6, OUT7, OUT8, OUT9: out bit);
end entity BCDdecoder;

architecture LogicOperation of BCDdecoder is
begin
  OUT0 <= not(not A0 and not A1 and not A2 and not A3);
  OUT1 <= not(A0 and not A1 and not A2 and not A3);
  OUT2 <= not(not A0 and A1 and not A2 and not A3);
  OUT3 <= not(A0 and A1 and not A2 and not A3);
  OUT4 <= not(not A0 and not A1 and A2 and not A3);
  OUT5 <= not(A0 and not A1 and A2 and not A3);
  OUT6 <= not(not A0 and A1 and A2 and not A3);
  OUT7 <= not(A0 and A1 and A2 and not A3);
  OUT8 <= not(not A0 and not A1 and not A2 and A3);
  OUT9 <= not(A0 and not A1 and not A2 and A3);
end architecture LogicOperation;
```

Inputs and outputs declared

Boolean expressions for the ten outputs

EXAMPLE 6–10

If the input waveforms in Figure 6–32(a) are applied to the inputs of the 74HC42, show the output waveforms.

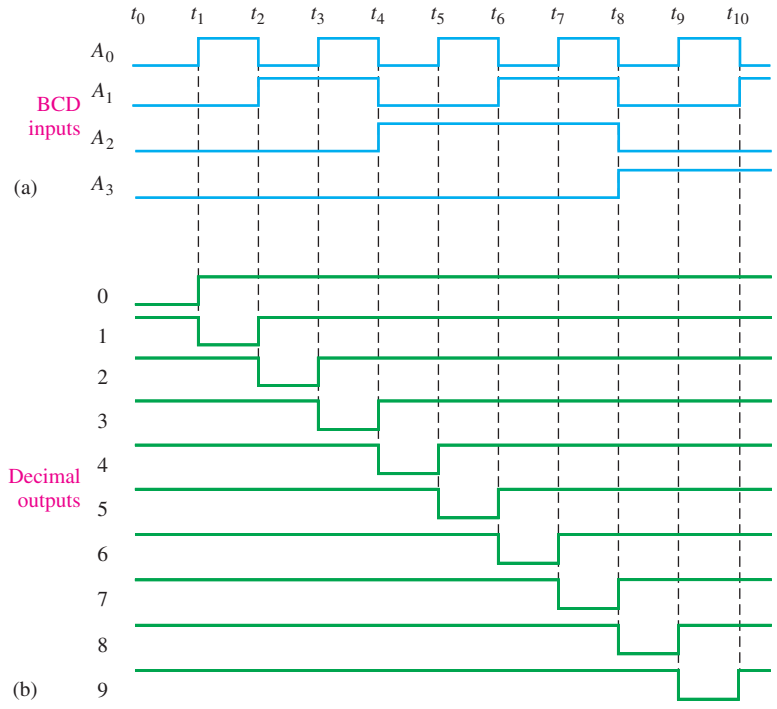


FIGURE 6–32

Solution

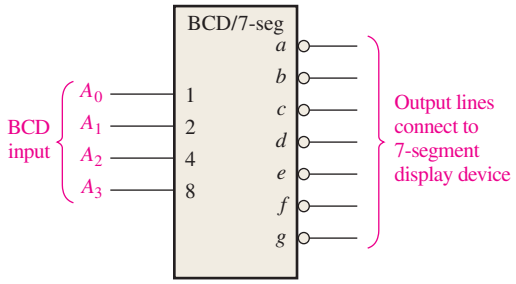
The output waveforms are shown in Figure 6–32(b). As you can see, the inputs are sequenced through the BCD for digits 0 through 9. The output waveforms in the timing diagram indicate that sequence on the decimal-value outputs.

Related Problem

Construct a timing diagram showing input and output waveforms for the case where the BCD inputs sequence through the decimal numbers as follows: 0, 2, 4, 6, 8, 1, 3, 5, and 9.

The BCD-to-7-Segment Decoder

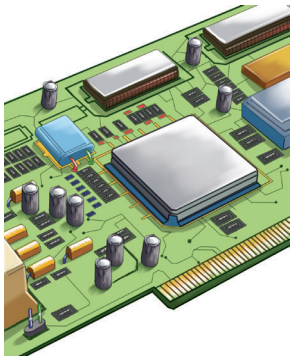
The BCD-to-7-segment decoder accepts the BCD code on its inputs and provides outputs to drive 7-segment display devices to produce a decimal readout. The logic diagram for a basic 7-segment decoder is shown in Figure 6–33.



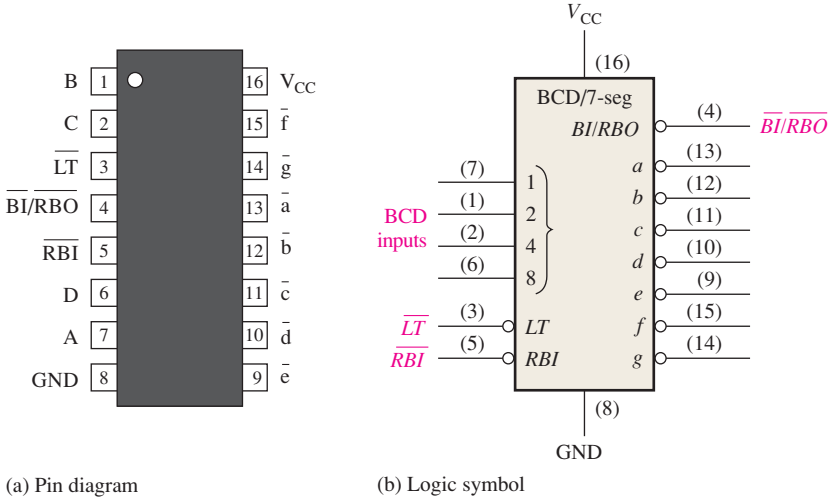
MultiSim

FIGURE 6–33 Logic symbol for a BCD-to-7-segment decoder/driver with active-LOW outputs. Open file F06-33 to verify operation.

IMPLEMENTATION: BCD-TO-7-SEGMENT DECODER/DRIVER



Fixed-Function Device The 74HC47 is an example of an IC device that decodes a BCD input and drives a 7-segment display. In addition to its decoding and segment drive capability, the 74HC47 has several additional features as indicated by the $\overline{LT}$, $\overline{RBI}$, $\overline{BI}/\overline{RBO}$ functions in the logic symbol of Figure 6–34. As indicated by the bubbles on the logic symbol, all of the outputs (*a* through *g*) are active-LOW as are the $\overline{LT}$ (lamp test), $\overline{RBI}$ (ripple blanking input), and $\overline{BI}/\overline{RBO}$ (blanking input/ripple blanking output) functions. The outputs can drive a common-anode 7-segment display directly. Recall that 7-segment displays were discussed in Chapter 4. In addition to decoding a BCD input and producing the appropriate 7-segment outputs, the 74HC47 has lamp test and zero suppression capability.



(a) Pin diagram (b) Logic symbol

FIGURE 6–34 The 74HC47 BCD-to-7-segment decoder/driver.

Lamp Test When a LOW is applied to the $\overline{LT}$ input and the $\overline{BI}/\overline{RBO}$ is HIGH, all of the seven segments in the display are turned on. Lamp test is used to verify that no segments are burned out.

Zero Suppression **Zero suppression** is a feature used for multidigit displays to blank out unnecessary zeros. For example, in a 6-digit display the number 6.4 may be displayed as 006.400 if the zeros are not blanked out. Blanking the zeros at the front of a number is called *leading zero suppression* and blanking the zeros at the back of the number is called *trailing zero suppression.* Keep in mind that only nonessential zeros are blanked. With zero suppression, the number 030.080 will be displayed as 30.08 (the essential zeros remain).

Zero suppression in the 74HC47 is accomplished using the $\overline{RBI}$ and $\overline{BI}/\overline{RBO}$ functions. $\overline{RBI}$ is the ripple blanking input and $\overline{RBO}$ is the ripple blanking output on the 74HC47; these are used for zero suppression. $\overline{BI}$ is the blanking input that shares the same pin with $\overline{RBO}$; in other words, the $\overline{BI}/\overline{RBO}$ pin can be used as an input or an output. When used as a $\overline{BI}$ (blanking input), all segment outputs are HIGH (nonactive) when $\overline{BI}$ is LOW, which overrides all other inputs. The $\overline{BI}$ function is not part of the zero suppression capability of the device.

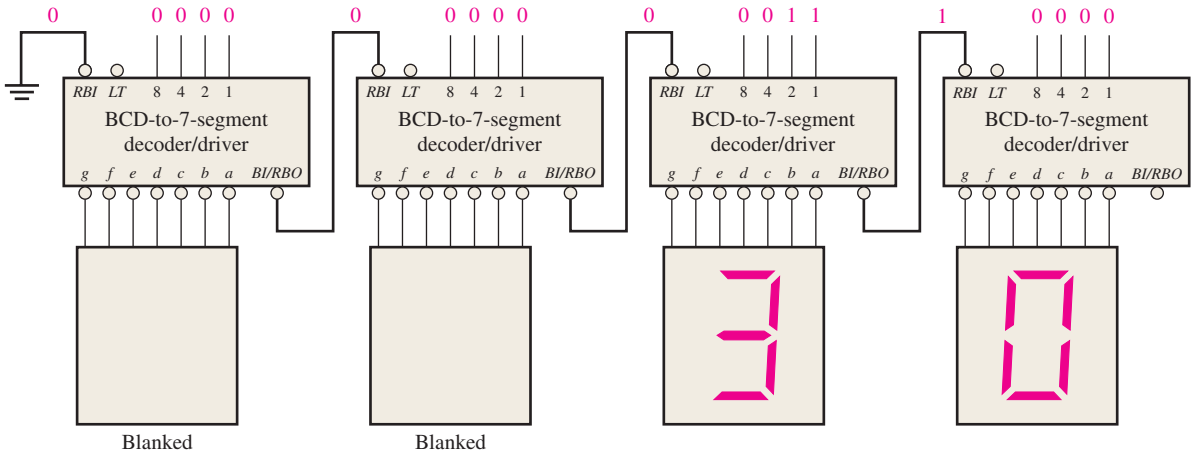
All of the segment outputs of the decoder are nonactive (HIGH) if a zero code (0000) is on its BCD inputs and if its $\overline{RBI}$ is LOW. This causes the display to be blank and produces a LOW $\overline{RBO}$.

Programmable Logic Device (PLD) The VHDL program code is the same as for the 74HC42 BCD-to-decimal decoder, except the 74HC47 has fewer outputs.

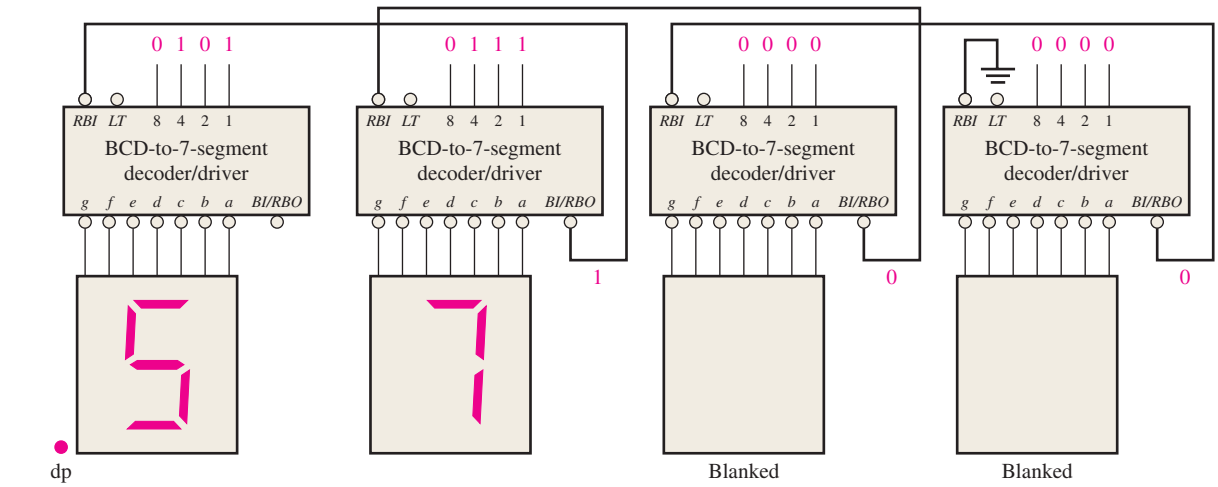
Zero Suppression for a 4-Digit Display

Zero suppression results in leading or trailing zeros in a number not showing on a display.

The logic diagram in Figure 6–35(a) illustrates leading zero suppression for a whole number. The highest-order digit position (left-most) is always blanked if a zero code is on its BCD inputs because the $\overline{RBI}$ of the most-significant decoder is made LOW by connecting it to ground. The $\overline{RBO}$ of each decoder is connected to the $\overline{RBI}$ of the next lowest-order decoder so that all zeros to the left of the first nonzero digit are blanked. For example, in part (a) of the figure the two highest-order digits are zeros and therefore are blanked. The remaining two digits, 3 and 0 are displayed.



(a) Illustration of leading zero suppression



(b) Illustration of trailing zero suppression

FIGURE 6–35 Examples of zero suppression using a BCD-to-7-segment decoder/driver.

The logic diagram in Figure 6–35(b) illustrates trailing zero suppression for a fractional number. The lowest-order digit (right-most) is always blanked if a zero code is on its BCD inputs because the $\overline{RBI}$ is connected to ground. The $\overline{RBO}$ of each decoder is connected to the $\overline{RBI}$ of the next highest-order decoder so that all zeros to the right of the first nonzero digit are blanked. In part (b) of the figure, the two lowest-order digits are zeros and therefore are blanked. The remaining two digits, 5 and 7 are displayed. To combine both leading and trailing zero suppression in one display and to have decimal point capability, additional logic is required.

SECTION 6–5 CHECKUP

1. A 3-line-to-8-line decoder can be used for octal-to-decimal decoding. When a binary 101 is on the inputs, which output line is activated?
2. How many 74HC154 1-of-16 decoders are necessary to decode a 6-bit binary number?
3. Would you select a decoder/driver with active-HIGH or active-LOW outputs to drive a common-cathode 7-segment LED display?

6–6 Encoders

An **encoder** is a combinational logic circuit that essentially performs a "reverse" decoder function. An encoder accepts an active level on one of its inputs representing a digit, such as a decimal or octal digit, and converts it to a coded output, such as BCD or binary. Encoders can also be devised to encode various symbols and alphabetic characters. The process of converting from familiar symbols or numbers to a coded format is called *encoding.*

After completing this section, you should be able to

- Determine the logic for a decimal-to-BCD encoder
- Explain the purpose of the priority feature in encoders
- Describe the 74HC147 decimal-to-BCD priority encoder
- Use VHDL to describe a decimal-to-BCD encoder
- Apply the encoder to a specific application

The Decimal-to-BCD Encoder

This type of encoder has ten inputs—one for each decimal digit—and four outputs corresponding to the BCD code, as shown in Figure 6–36. This is a basic 10-line-to-4-line encoder.

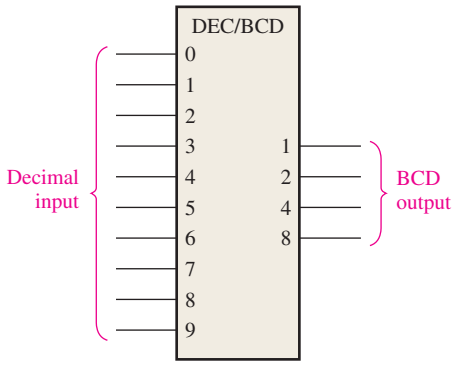


FIGURE 6–36 Logic symbol for a decimal-to-BCD encoder.

The BCD (8421) code is listed in Table 6–6. From this table you can determine the relationship between each BCD bit and the decimal digits in order to analyze the logic. For instance, the most significant bit of the BCD code, A_3, is always a 1 for decimal digit 8 or 9. An OR expression for bit A_3 in terms of the decimal digits can therefore be written as

$$A_3 = 8 + 9$$

TABLE 6–6

Decimal Digit	BCD Code A_3	A_2	A_1	A_0
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

InfoNote

An *assembler* can be thought of as a software encoder because it interprets the mnemonic instructions with which a program is written and carries out the applicable encoding to convert each mnemonic to a machine code instruction (series of 1s and 0s) that the processor can understand. Examples of mnemonic instructions for a processor are ADD, MOV (move data), MUL (multiply), XOR, JMP (jump), and OUT (output to a port).

Bit A_2 is always a 1 for decimal digit 4, 5, 6 or 7 and can be expressed as an OR function as follows:

$$A_2 = 4 + 5 + 6 + 7$$

Bit A_1 is always a 1 for decimal digit 2, 3, 6, or 7 and can be expressed as

$$A_1 = 2 + 3 + 6 + 7$$

Finally, A_0 is always a 1 for decimal digit 1, 3, 5, 7, or 9. The expression for A_0 is

$$A_0 = 1 + 3 + 5 + 7 + 9$$

Now let's implement the logic circuitry required for encoding each decimal digit to a BCD code by using the logic expressions just developed. It is simply a matter of ORing the appropriate decimal digit input lines to form each BCD output. The basic encoder logic resulting from these expressions is shown in Figure 6–37.

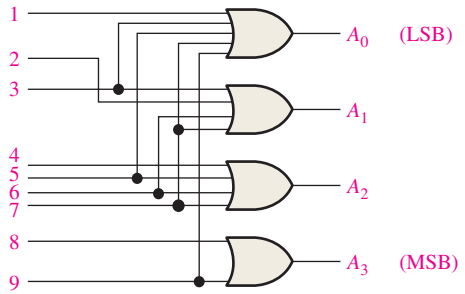


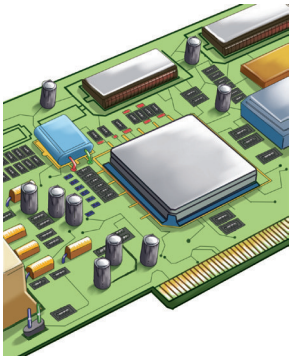
FIGURE 6–37 Basic logic diagram of a decimal-to-BCD encoder. A 0-digit input is not needed because the BCD outputs are all LOW when there are no HIGH inputs.

The basic operation of the circuit in Figure 6–37 is as follows: When a HIGH appears on *one* of the decimal digit input lines, the appropriate levels occur on the four BCD output lines. For instance, if input line 9 is HIGH (assuming all other input lines are LOW), this condition will produce a HIGH on outputs A_0 and A_3 and LOWs on outputs A_1 and A_2, which is the BCD code (1001) for decimal 9.

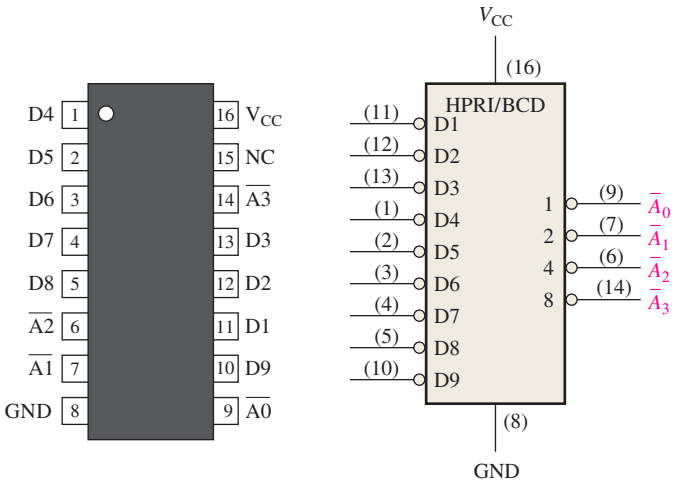
The Decimal-to-BCD Priority Encoder

This type of encoder performs the same basic encoding function as previously discussed. A **priority encoder** also offers additional flexibility in that it can be used in applications that require priority detection. The priority function means that the encoder will produce a BCD output corresponding to the *highest-order decimal digit* input that is active and will ignore any other lower-order active inputs. For instance, if the 6 and the 3 inputs are both active, the BCD output is 0110 (which represents decimal 6).

IMPLEMENTATION: DECIMAL-TO-BCD ENCODER



Fixed-Function Device The 74HC147 is a priority encoder with active-LOW inputs (0) for decimal digits 1 through 9 and active-LOW BCD outputs as indicated in the logic symbol in Figure 6–38. A BCD zero output is represented when none of the inputs is active. The device pin numbers are in parentheses.



(a) Pin diagram (b) Logic diagram

FIGURE 6–38 The 74HC147 decimal-to-BCD encoder (HPRI means highest value input has priority.

Programmable Logic Device (PLD) The logic of the decimal-to-BCD encoder shown in Figure 6–38 can be described in VHDL for implementation in a PLD. The data flow approach is used in this case.

```
entity DecBCDencoder is
    port (D1, D2, D3, D4, D5, D6, D7, D8, D9:
    in bit; A0, A1, A2, A3: out bit);
end entity DecBCDencoder;

architecture LogicFunction of DecBCDencoder is
begin
    A0 <= (D1 or D3 or D5 or D7 or D9);
    A1 <= (D2 or D3 or D6 or D7);
    A2 <= (D4 or D5 or D6 or D7);
    A3 <= (D8 or D9);
end architecture LogicFunction;
```

Inputs and outputs declared

Boolean expressions for the four BCD outputs

EXAMPLE 6–11

If LOW levels appear on pins, 1, 4, and 13 of the 74HC147 shown in Figure 6–38, indicate the state of the four outputs. All other inputs are HIGH.

Solution

Pin 4 is the highest-order decimal digit input having a LOW level and represents decimal 7. Therefore, the output levels indicate the BCD code for decimal 7 where $\overline{A}_0$ is the LSB and $\overline{A}_3$ is the MSB. Output $\overline{A}_0$ is LOW, $\overline{A}_1$ is LOW, $\overline{A}_2$ is LOW, and $\overline{A}_3$ is HIGH.

Related Problem

What are the outputs of the 74HC147 if all its inputs are LOW? If all its inputs are HIGH?

An Application

The ten decimal digits on a numeric keypad must be encoded for processing by the logic circuitry. In this example, when one of the keys is pressed, the decimal digit is encoded to the corresponding BCD code. Figure 6–39 shows a simple keyboard encoder arrangement using a priority encoder. The keys are represented by ten push-button switches, each with a **pull-up resistor** to $+V$. The pull-up resistor ensures that the line is HIGH when a key is not depressed. When a key is depressed, the line is connected to ground, and a LOW is applied to the corresponding encoder input. The zero key is not connected because the BCD output represents zero when none of the other keys is depressed.

The BCD complement output of the encoder goes into a storage device, and each successive BCD code is stored until the entire number has been entered. Methods of storing BCD numbers and binary data are covered in Chapter 11.

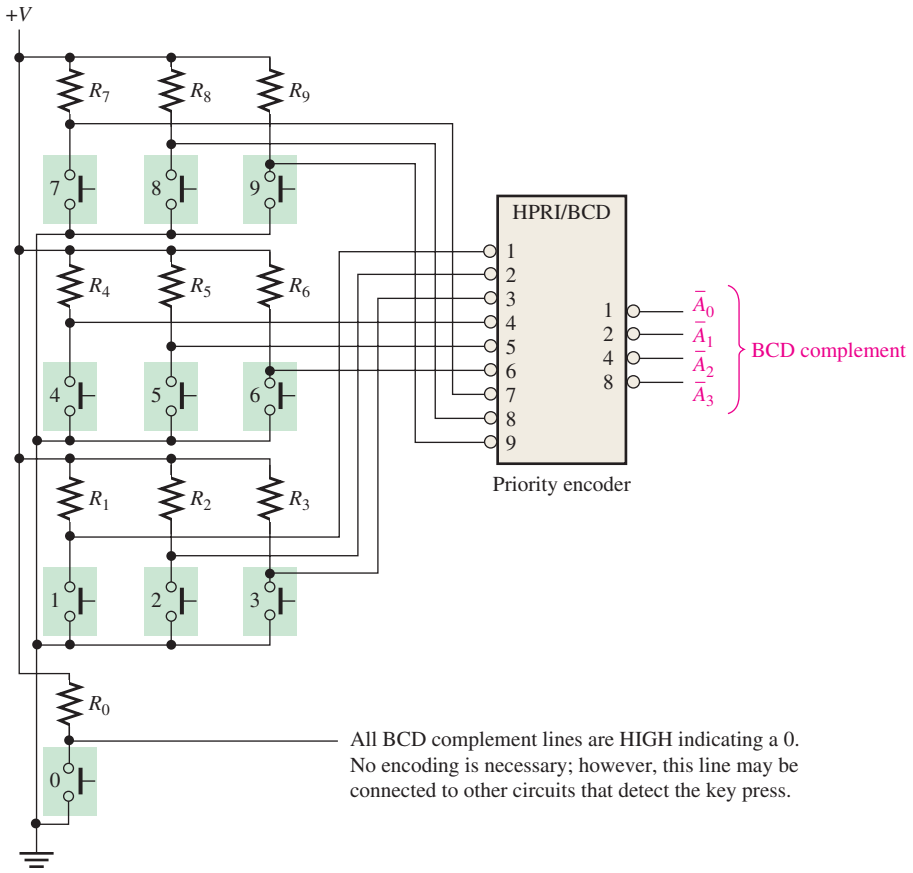


FIGURE 6–39 A simplified keyboard encoder.

SECTION 6–6 CHECKUP

1. Suppose the HIGH levels are applied to the 2 input and the 9 input of the circuit in Figure 6–37.
   - **(a)** What are the states of the output lines?
   - **(b)** Does this represent a valid BCD code?
   - **(c)** What is the restriction on the encoder logic in Figure 6–37?
2. **(a)** What is the $\overline{A}_3\,\overline{A}_2\,\overline{A}_1\,\overline{A}_0$ output when LOWs are applied to pins 1 and 5 of the 74HC147 in Figure 6–38?
   - **(b)** What does this output represent?

6–7 Code Converters

In this section, we will examine some methods of using combinational logic circuits to convert from one code to another.

After completing this section, you should be able to

- Explain the process for converting BCD to binary
- Use exclusive-OR gates for conversions between binary and Gray codes

BCD-to-Binary Conversion

One method of BCD-to-binary code conversion uses adder circuits. The basic conversion process is as follows:

1. The value, or weight, of each bit in the BCD number is represented by a binary number.
2. All of the binary representations of the weights of bits that are 1s in the BCD number are added.
3. The result of this addition is the binary equivalent of the BCD number.

A more concise statement of this operation is

The binary numbers representing the weights of the BCD bits are summed to produce the total binary number.

Let's examine an 8-bit BCD code (one that represents a 2-digit decimal number) to understand the relationship between BCD and binary. For instance, you already know that the decimal number 87 can be expressed in BCD as

$$\underbrace{1000}_{8} \quad \underbrace{0111}_{7}$$

The left-most 4-bit group represents 80, and the right-most 4-bit group represents 7. That is, the left-most group has a weight of 10, and the right-most group has a weight of 1. Within each group, the binary weight of each bit is as follows:

	Tens Digit				**Units Digit**			
Weight:	80	40	20	10	8	4	2	1
Bit designation:	B_3	B_2	B_1	B_0	A_3	A_2	A_1	A_0

The binary equivalent of each BCD bit is a binary number representing the weight of that bit within the total BCD number. This representation is given in Table 6–7.

TABLE 6–7

Binary representations of BCD bit weights.

BCD Bit	**BCD Weight**	**(MSB)**		**Binary Representation**				**(LSB)**
		64	**32**	**16**	**8**	**4**	**2**	**1**
A_0	1	0	0	0	0	0	0	1
A_1	2	0	0	0	0	0	1	0
A_2	4	0	0	0	0	1	0	0
A_3	8	0	0	0	1	0	0	0
B_0	10	0	0	0	1	0	1	0
B_1	20	0	0	1	0	1	0	0
B_2	40	0	1	0	1	0	0	0
B_3	80	1	0	1	0	0	0	0

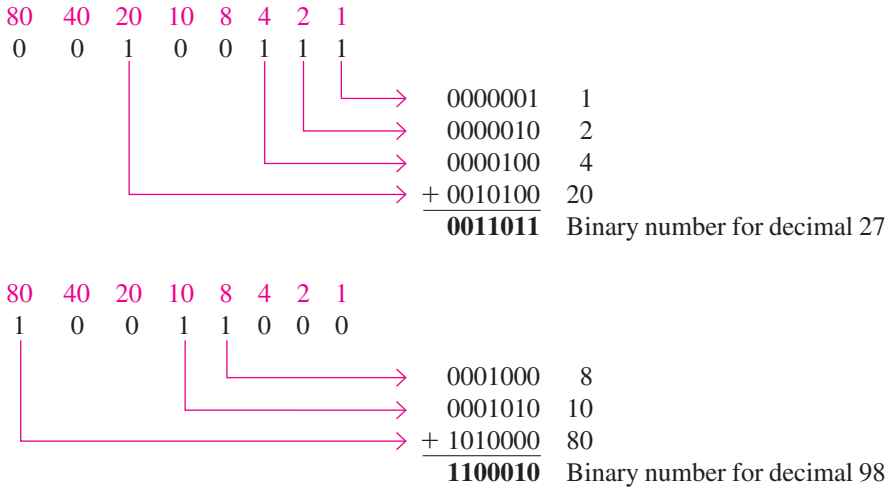
If the binary representations for the weights of all the 1s in the BCD number are added, the result is the binary number that corresponds to the BCD number. Example 6–12 illustrates this.

EXAMPLE 6–12

Convert the BCD numbers 00100111 (decimal 27) and 10011000 (decimal 98) to binary.

Solution

Write the binary representations of the weights of all 1s appearing in the numbers, and then add them together.



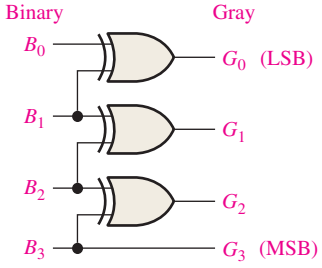
Related Problem

Show the process of converting 01000001 in BCD to binary.

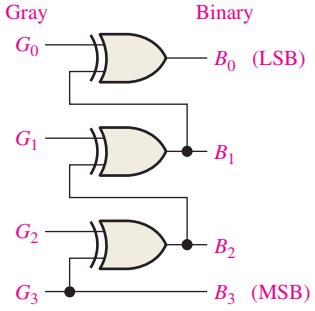
MultiSim Open file EX06-12 and run the simulation to observe the operation of a BCD-to-binary logic circuit.

Binary-to-Gray and Gray-to-Binary Conversion

The basic process for Gray-binary conversions was covered in Chapter 2. Exclusive-OR gates can be used for these conversions. Programmable logic devices (PLDs) can also be programmed for these code conversions. Figure 6–40 shows a 4-bit binary-to-Gray code converter, and Figure 6–41 illustrates a 4-bit Gray-to-binary converter.



MultiSim **FIGURE 6–40** Four-bit binary-to-Gray conversion logic. Open file F06-40 to verify operation.



MultiSim **FIGURE 6–41** Four-bit Gray-to-binary conversion logic. Open file F06-41 to verify operation.

EXAMPLE 6–13

(a) Convert the binary number 0101 to Gray code with exclusive-OR gates.

(b) Convert the Gray code 1011 to binary with exclusive-OR gates.

Solution

(a) 0101_2 is 0111 Gray. See Figure 6–42(a).

(b) 1011 Gray is 1101_2. See Figure 6–42(b).

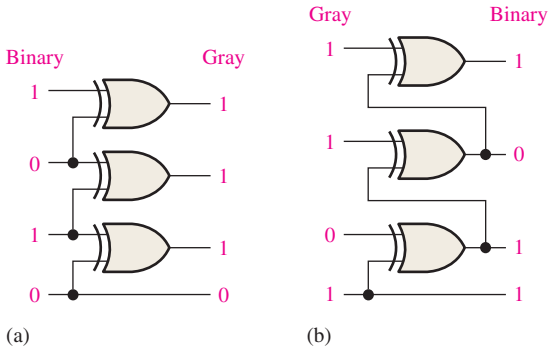


FIGURE 6–42

Related Problem

How many exclusive-OR gates are required to convert 8-bit binary to Gray?

SECTION 6–7 CHECKUP

1. Convert the BCD number 10000101 to binary.
2. Draw the logic diagram for converting an 8-bit binary number to Gray code.

6–8 Multiplexers (Data Selectors)

A **multiplexer (MUX)** is a device that allows digital information from several sources to be routed onto a single line for transmission over that line to a common destination. The basic multiplexer has several data-input lines and a single output line. It also has data-select inputs, which permit digital data on any one of the inputs to be switched to the output line. Multiplexers are also known as data selectors.

After completing this section, you should be able to

- Explain the basic operation of a multiplexer
- Describe the 74HC153 and the 74HC151 multiplexers
- Expand a multiplexer to handle more data inputs
- Use the multiplexer as a logic function generator
- Use VHDL to describe 4-input and 8-input multiplexers

A logic symbol for a 4-input multiplexer (MUX) is shown in Figure 6–43. Notice that there are two data-select lines because with two select bits, any one of the four data-input lines can be selected.

In a multiplexer, data are switched from several lines to one line.

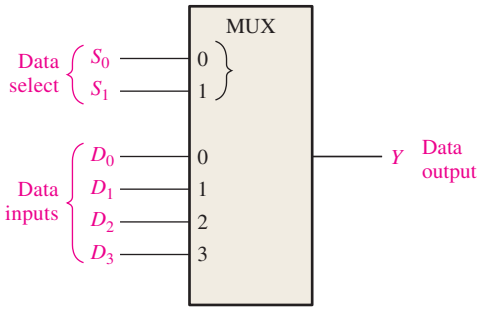


FIGURE 6–43 Logic symbol for a 1-of-4 data selector/multiplexer.

In Figure 6–43, a 2-bit code on the data-select (S) inputs will allow the data on the selected data input to pass through to the data output. If a binary 0 ($S_1 = 0$ and $S_0 = 0$) is applied to the data-select lines, the data on input D_0 appear on the data-output line. If a binary 1 ($S_1 = 0$ and $S_0 = 1$) is applied to the data-select lines, the data on input D_1 appear on the data output. If a binary 2 ($S_1 = 1$ and $S_0 = 0$) is applied, the data on D_2 appear on the output. If a binary 3 ($S_1 = 1$ and $S_0 = 1$) is applied, the data on D_3 are switched to the output line. A summary of this operation is given in Table 6–8.

TABLE 6–8

Data selection for a 1-of-4-multiplexer.

Data-Select Inputs		
S_1	S_0	Input Selected
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

InfoNote

A *bus* is a multiple conductor pathway along which electrical signals are sent from one part of a computer to another. In computer networks, a *shared bus* is one that is connected to all the microprocessors in the system in order to exchange data. A shared bus may contain memory and input/output devices that can be accessed by all the microprocessors in the system. Access to the shared bus is controlled by a *bus arbiter* (a multiplexer of sorts) that allows only one microprocessor at a time to use the system's shared bus.

Now let's look at the logic circuitry required to perform this multiplexing operation. The data output is equal to the state of the *selected* data input. You can therefore, derive a logic expression for the output in terms of the data input and the select inputs.

The data output is equal to D_0 only if $S_1 = 0$ and $S_0 = 0$: $Y = D_0\overline{S}_1\overline{S}_0$.
The data output is equal to D_1 only if $S_1 = 0$ and $S_0 = 1$: $Y = D_1\overline{S}_1S_0$.
The data output is equal to D_2 only if $S_1 = 1$ and $S_0 = 0$: $Y = D_2S_1\overline{S}_0$.
The data output is equal to D_3 only if $S_1 = 1$ and $S_0 = 1$: $Y = D_3S_1S_0$.

When these terms are ORed, the total expression for the data output is

$$Y = D_0\overline{S}_1\overline{S}_0 + D_1\overline{S}_1S_0 + D_2S_1\overline{S}_0 + D_3S_1S_0$$

The implementation of this equation requires four 3-input AND gates, a 4-input OR gate, and two inverters to generate the complements of S_1 and S_0, as shown in Figure 6–44. Because data can be selected from any one of the input lines, this circuit is also referred to as a **data selector**.

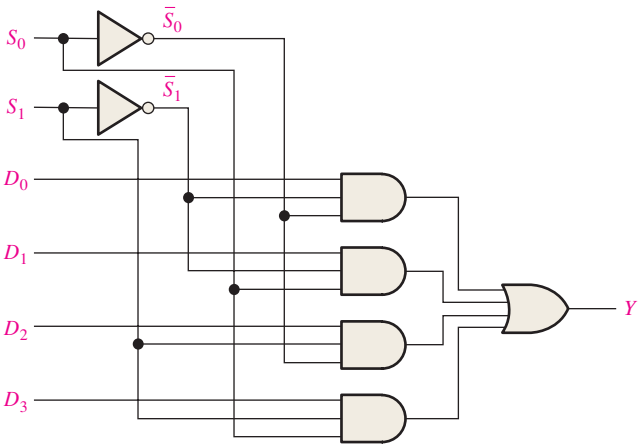


FIGURE 6–44 Logic diagram for a 4-input multiplexer. Open file F06-44 to verify operation.

MultiSim

EXAMPLE 6–14

The data-input and data-select waveforms in Figure 6–45(a) are applied to the multiplexer in Figure 6–44. Determine the output waveform in relation to the inputs.

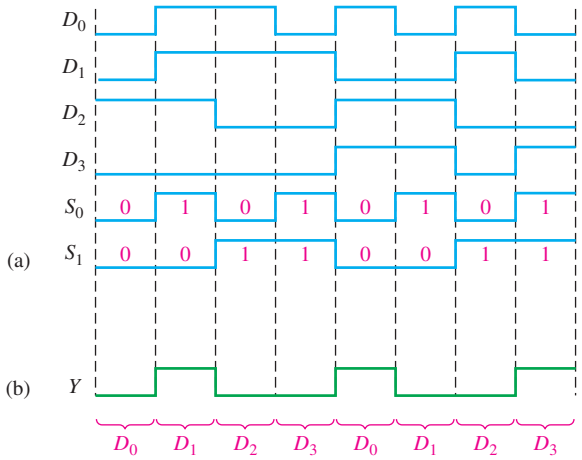


FIGURE 6–45

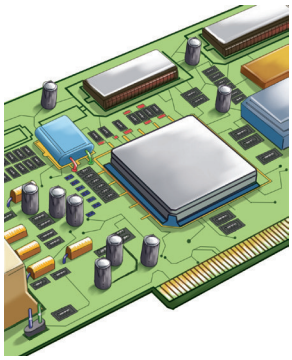
Solution

The binary state of the data-select inputs during each interval determines which data input is selected. Notice that the data-select inputs go through a repetitive binary sequence 00, 01, 10, 11, 00, 01, 10, 11, and so on. The resulting output waveform is shown in Figure 6–45(b).

Related Problem

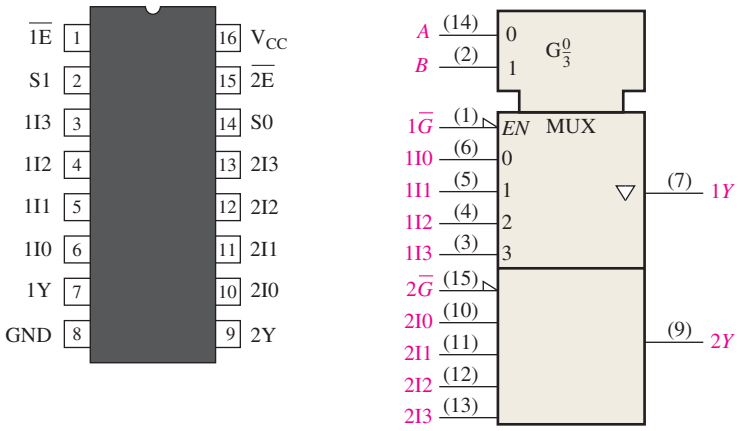
Construct a timing diagram showing all inputs and the output if the S_0 and S_1 waveforms in Figure 6–45 are interchanged.

IMPLEMENTATION: DATA SELECTOR/MULTIPLEXER



Fixed-Function Device The 74HC153 is a dual four-input data selector/multiplexer. The pin diagram is shown in Figure 6–46(a). The inputs to one of the multiplexers are 1I0 through 1I3 and the inputs to the second multiplexer are 2I0 through 2I3. The data select inputs are S0 and S1 and the active-LOW enable inputs are 1E and 2E. Each of the multiplexers has an active-LOW enable input.

The ANSI/IEEE logic symbol with dependency notation is shown in Figure 6–46(b). The two multiplexers are indicated by the partitioned outline, and the inputs common to both multiplexers are inputs to the notched block (common control block) at the top. The G_3^0 dependency notation indicates an AND relationship between the two select inputs (A and B) and the inputs to each multiplexer block.



(a) Pin diagram (b) Logic symbol

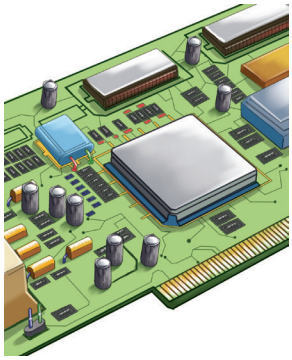
FIGURE 6–46 The 74HC153 dual four-input data selector/multiplexer.

Programmable Logic Device (PLD) The logic for a four-input multiplexer like the one shown in the logic diagram of Figure 6–44 can be described with VHDL. The data flow approach is used for this particular circuit. Keep in mind that once you have written the VHDL program for a given logic, the code is then downloaded into a PLD device and becomes actual hardware just as fixed-function devices are hardware.

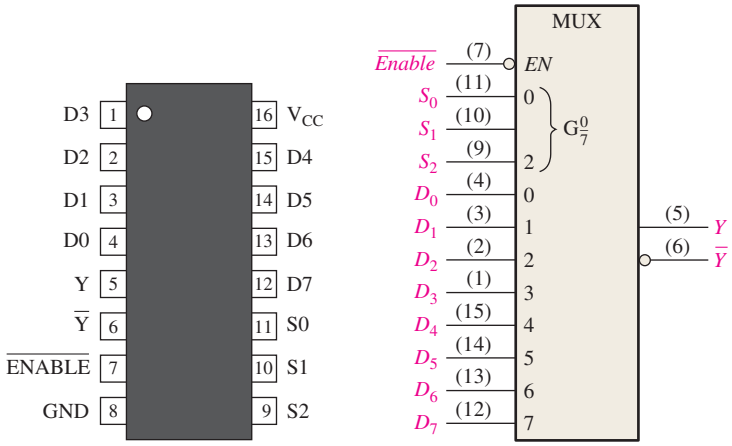
```
entity FourInputMultiplexer is
    port (S0, S1, D0, D1, D2, D3; in bit; Y: out bit);        -- Inputs and outputs declared
end entity FourInputMultiplexer;

architecture LogicFunction of FourInputMultiplexer is
begin
    Y <= (D0 and not S0 and not S1) or (Dl and S0 and not S1)
    or (D2 and not S0 and S1) or (D3 and S0 and S1);          -- Boolean expression for the output
end architecture LogicFunction;
```

IMPLEMENTATION: EIGHT-INPUT DATA SELECTOR/MULTIPLEXER



Fixed-Function Device The 74HC151 has eight data inputs (D_0–D_7) and, therefore, three data-select or address input lines (S_0–S_2). Three bits are required to select any one of the eight data inputs ($2^3 = 8$). A LOW on the *Enable* input allows the selected input data to pass through to the output. Notice that the data output and its complement are both available. The pin diagram is shown in Figure 6–47(a), and the ANSI/IEEE logic symbol is shown in part (b). In this case there is no need for a common control block on the logic symbol because there is only one multiplexer to be controlled, not two as in the 74HC153. The G_7^0 label within the logic symbol indicates the AND relationship between the data-select inputs and each of the data inputs 0 through 7.



(a) Pin diagram (b) Logic symbol

FIGURE 6–47 The 74HC151 eight-input data selector/multiplexer.

Programmable Logic Device (PLD) The logic for the eight-input multiplexer is implemented by first writing the VHDL code. For the 74HC151, eight 5-input AND gates, one 8-input OR gate, and four inverters are required.

```
entity EightInputMUX is
    port (S0, S1, S2, D0, D1, D2, D3, D4, D5, D6, D7,
    EN: in bit; Y: inout bit; YI: out bit);
end entity EightInputMUX;

architecture LogicOperation of EightInputMUX is
    signal AND0, AND1, AND2, AND3, AND4, AND5, AND6, AND7: bit;
    begin
    AND0 <= not S0 and not S1 and not S2 and D0 and not EN;
    AND1 <= S0 and not S1 and not S2 and D1 and not EN;
    AND2 <= not S0 and S1 and not S2 and D2 and not EN;
    AND3 <= S0 and S1 and not S2 and D3 and not EN;
    AND4 <= not S0 and not S1 and S2 and D4 and not EN;
    AND5 <= S0 and not S1 and S2 and D5 and not EN;
    AND6 <= not S0 and S1 and S2 and D6 and not EN;
    AND7 <= S0 and S1 and S2 and D7 and not EN;
    Y <= AND0 or AND1 or AND2 or AND3 or AND4 or AND5 or AND6 or AND7;
    YI <= not Y;
end architecture LogicOperation;
```

Inputs and outputs declared

Internal signals (outputs of AND gates) declared

Boolean expressions for internal AND gate outputs

Boolean expressions for fixed outputs

EXAMPLE 6–15

Use 74HC151s and any other logic necessary to multiplex 16 data lines onto a single data-output line.

Solution

An expansion of two 74HC151s is shown in Figure 6–48. Four bits are required to select one of 16 data inputs ($2^4 = 16$). In this application the $\overline{Enable}$ input is used as the most significant data-select bit. When the MSB in the data-select code is LOW, the left 74HC151 is enabled, and one of the data inputs (D_0 through D_7) is selected by the other three data-select bits. When the data-select MSB is HIGH, the right 74HC151 is enabled, and one of the data inputs (D_8 through D_{15}) is selected. The selected input data are then passed through to the negative-OR gate and onto the single output line.

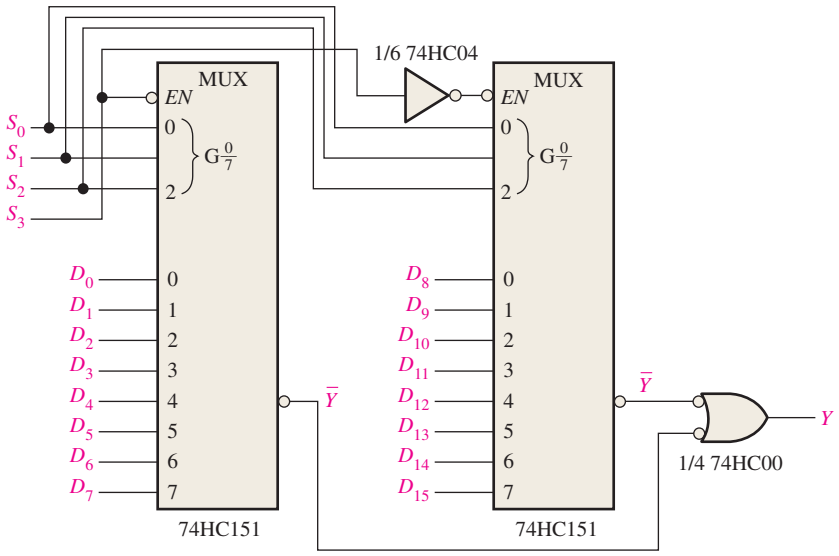


FIGURE 6–48 A 16-input multiplexer.

Related Problem

Determine the codes on the select inputs required to select each of the following data inputs: D_0, D_4, D_8, and D_{13}.

Applications

A 7-Segment Display Multiplexer

Figure 6–49 shows a simplified method of multiplexing BCD numbers to a 7-segment display. In this example, 2-digit numbers are displayed on the 7-segment readout by the use of a single BCD-to-7-segment decoder. This basic method of display multiplexing can be extended to displays with any number of digits. The 74HC157 is a quad 2-input multiplexer.

The basic operation is as follows. Two BCD digits ($A_3A_2A_1A_0$ and $B_3B_2B_1B_0$) are applied to the multiplexer inputs. A square wave is applied to the data-select line, and when it is LOW, the A bits ($A_3A_2A_1A_0$) are passed through to the inputs of the 74HC47 BCD-to-7-segment decoder. The LOW on the data-select also puts a LOW on the A_1 input of the 74HC139 2-line-to-4-line decoder, thus activating its 0 output and enabling the A-digit display by effectively connecting its common terminal to ground. The A digit is now *on* and the B digit is *off*.

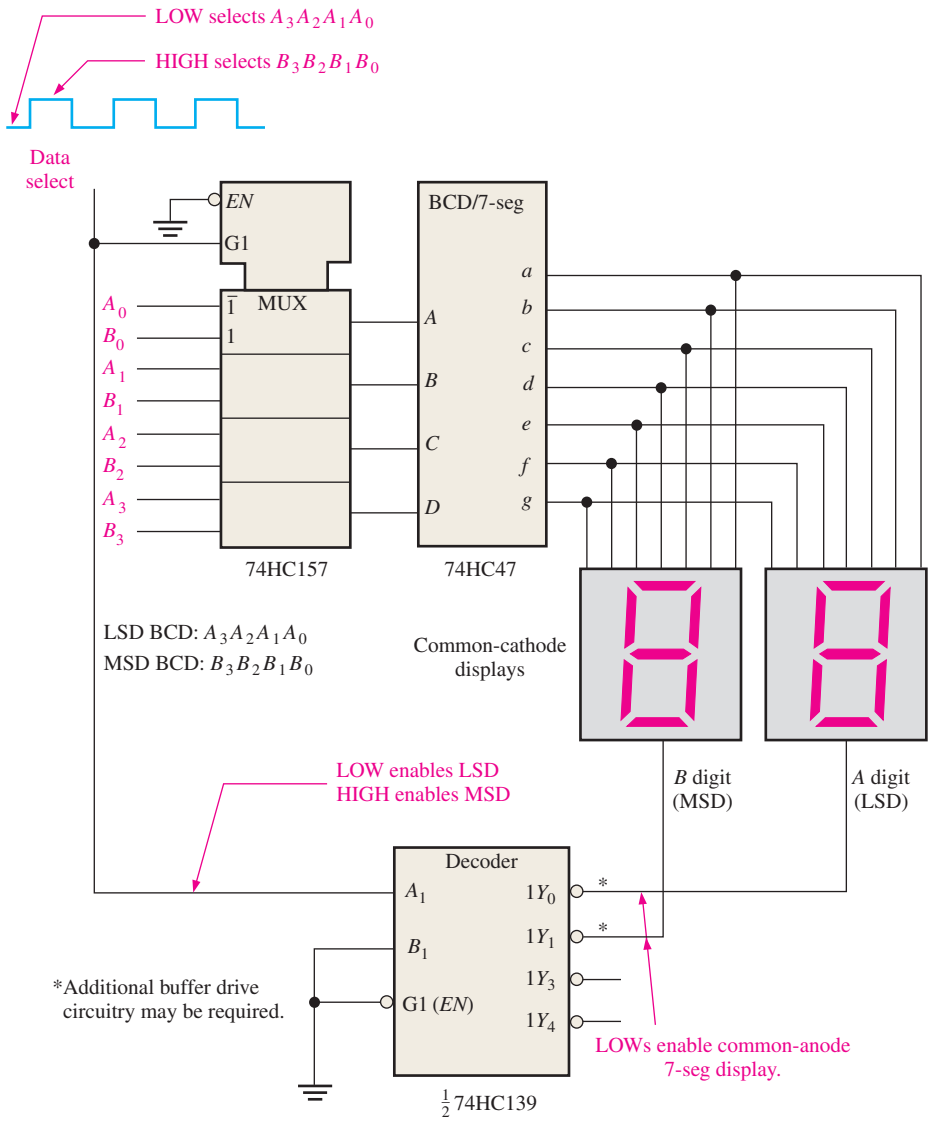


FIGURE 6–49 Simplified 7-segment display multiplexing logic.

When the data-select line goes HIGH, the *B* bits ($B_3B_2B_1B_0$) are passed through to the inputs of the BCD-to-7-segment decoder. Also, the 74HC139 decoder's 1 output is activated, thus enabling the *B*-digit display. The *B* digit is now *on* and the *A* digit is *off*. The cycle repeats at the frequency of the data-select square wave. This frequency must be high enough to prevent visual flicker as the digit displays are multiplexed.

A Logic Function Generator

A useful application of the data selector/multiplexer is in the generation of combinational logic functions in sum-of-products form. When used in this way, the device can replace discrete gates, can often greatly reduce the number of ICs, and can make design changes much easier.

To illustrate, a 74HC151 8-input data selector/multiplexer can be used to implement any specified 3-variable logic function if the variables are connected to the data-select inputs and each data input is set to the logic level required in the truth table for that function. For example, if the function is a 1 when the variable combination is $\overline{A}_2A_1\overline{A}_0$, the 2 input (selected by 010) is connected to a HIGH. This HIGH is passed through to the output when this particular combination of variables occurs on the data-select lines. Example 6–16 will help clarify this application.

EXAMPLE 6–16

Implement the logic function specified in Table 6–9 by using a 74HC151 8-input data selector/multiplexer. Compare this method with a discrete logic gate implementation.

TABLE 6–9

Inputs			Output
A_2	A_1	A_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Solution

Notice from the truth table that Y is a 1 for the following input variable combinations: 001, 011, 101, and 110. For all other combinations, Y is 0. For this function to be implemented with the data selector, the data input selected by each of the above-mentioned combinations must be connected to a HIGH (5 V). All the other data inputs must be connected to a LOW (ground), as shown in Figure 6–50.

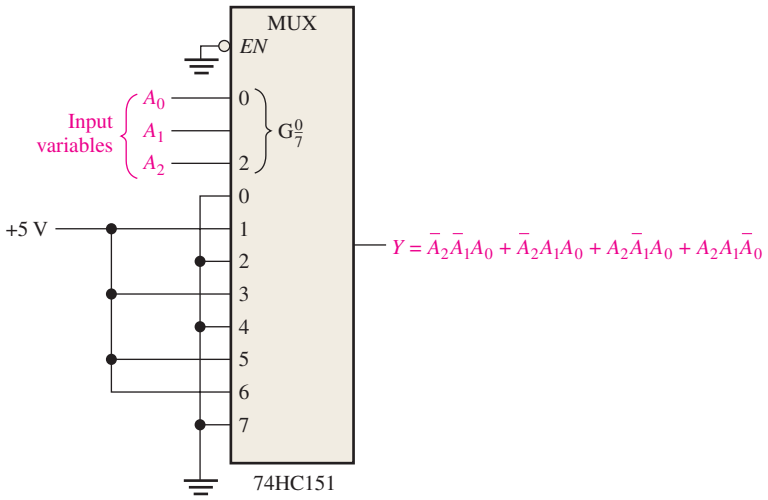


FIGURE 6–50 Data selector/multiplexer connected as a 3-variable logic function generator.

The implementation of this function with logic gates would require four 3-input AND gates, one 4-input OR gate, and three inverters unless the expression can be simplified.

Related Problem

Use the 74HC151 to implement the following expression:

$$Y = \overline{A}_2\overline{A}_1\overline{A}_0 + A_2\overline{A}_1\overline{A}_0 + \overline{A}_2A_1\overline{A}_0$$

Example 6–16 illustrated how the 8-input data selector can be used as a logic function generator for three variables. Actually, this device can be also used as a 4-variable logic function generator by the utilization of one of the bits (A_0) in conjunction with the data inputs.

A 4-variable truth table has sixteen combinations of input variables. When an 8-bit data selector is used, each input is selected twice: the first time when A_0 is 0 and the second time when A_0 is 1. With this in mind, the following rules can be applied (Y is the output, and A_0 is the least significant bit):

1. If $Y = 0$ both times a given data input is selected by a certain combination of the input variables, $A_3A_2A_1$, connect that data input to ground (0).
2. If $Y = 1$ both times a given data input is selected by a certain combination of the input variables, $A_3A_2A_1$, connect the data input to $+V$ (1).
3. If Y is different the two times a given data input is selected by a certain combination of the input variables, $A_3A_2A_1$, and if $Y = A_0$, connect that data input to A_0.
4. If Y is different the two times a given data input is selected by a certain combination of the input variables, $A_3A_2A_1$, and if $Y = \overline{A}_0$, connect that data input to $\overline{A}_0$.

EXAMPLE 6–17

Implement the logic function in Table 6–10 by using a 74HC151 8-input data selector/multiplexer. Compare this method with a discrete logic gate implementation.

TABLE 6–10

Decimal Digit	Inputs				Output
	A_3	A_2	A_1	A_0	Y
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	1

Solution

The data-select inputs are $A_3A_2A_1$. In the first row of the table, $A_3A_2A_1 = 000$ and $Y = A_0$. In the second row, where $A_3A_2A_1$ again is 000, $Y = A_0$. Thus, A_0 is connected to the 0 input. In the third row of the table, $A_3A_2A_1 = 001$ and $Y = \overline{A}_0$. Also, in the fourth row, when $A_3A_2A_1$ again is 001, $Y = \overline{A}_0$. Thus, A_0 is inverted and connected to the 1 input. This analysis is continued until each input is properly connected according to the specified rules. The implementation is shown in Figure 6–51.

If implemented with logic gates, the function would require as many as ten 4-input AND gates, one 10-input OR gate, and four inverters, although possible simplification would reduce this requirement.

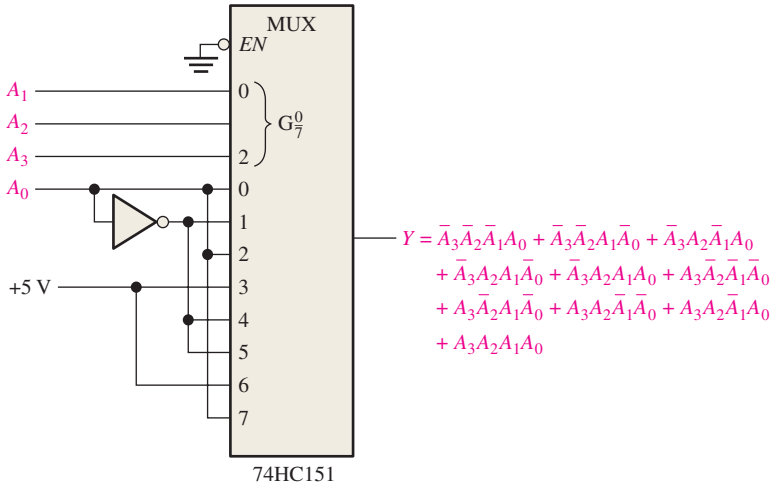


FIGURE 6–51 Data selector/multiplexer connected as a 4-variable logic function generator.

Related Problem

In Table 6–10, if $Y = 0$ when the inputs are all zeros and is alternately a 1 and a 0 for the remaining rows in the table, use a 74HC151 to implement the resulting logic function.

SECTION 6–8 CHECKUP

1. In Figure 6–44, $D_0 = 1$, $D_1 = 0$, $D_2 = 1$, $D_3 = 0$, $S_0 = 1$, and $S_1 = 0$. What is the output?
2. Identify each device.
   (a) 74HC153 **(b)** 74HC151
3. A 74HC151 has alternating LOW and HIGH levels on its data inputs beginning with $D_0 = 0$. The data-select lines are sequenced through a binary count (000, 001, 010, and so on) at a frequency of 1 kHz. The enable input is LOW. Describe the data output waveform.
4. Briefly describe the purpose of each of the following devices in Figure 6–49:
   (a) 74HC157 **(b)** 74HC47 **(c)** 74HC139

6–9 Demultiplexers

A **demultiplexer (DEMUX)** basically reverses the multiplexing function. It takes digital information from one line and distributes it to a given number of output lines. For this reason, the demultiplexer is also known as a data distributor. As you will learn, decoders can also be used as demultiplexers.

After completing this section, you should be able to

- Explain the basic operation of a demultiplexer
- Describe how a 4-line-to-16-line decoder can be used as a demultiplexer
- Develop the timing diagram for a demultiplexer with specified data and data selection inputs

In a demultiplexer, data are switched from one line to several lines.

Figure 6–52 shows a 1-line-to-4-line demultiplexer (DEMUX) circuit. The data-input line goes to all of the AND gates. The two data-select lines enable only one gate at a time, and the data appearing on the data-input line will pass through the selected gate to the associated data-output line.

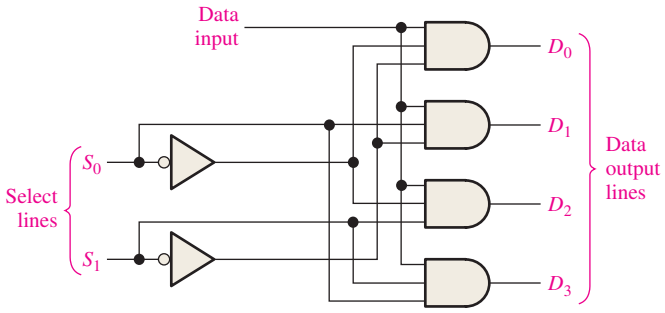


FIGURE 6–52 A 1-line-to-4-line demultiplexer.

EXAMPLE 6–18

The serial data-input waveform (Data in) and data-select inputs (S_0 and S_1) are shown in Figure 6–53. Determine the data-output waveforms on D_0 through D_3 for the demultiplexer in Figure 6–52.

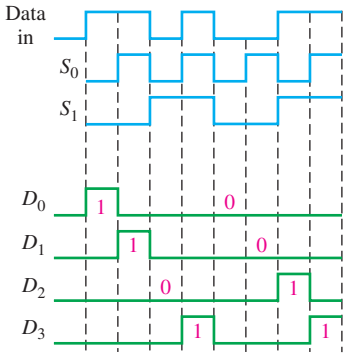


FIGURE 6–53

Solution

Notice that the select lines go through a binary sequence so that each successive input bit is routed to D_0, D_1, D_2, and D_3 in sequence, as shown by the output waveforms in Figure 6–53.

Related Problem

Develop the timing diagram for the demultiplexer if the S_0 and S_1 waveforms are both inverted.

4-Line-to-16-Line Decoder as a Demultiplexer

We have already discussed a 4-line-to-16-line decoder (Section 6–5). This device and other decoders can also be used in demultiplexing applications. The logic symbol for this device when used as a demultiplexer is shown in Figure 6–54. In demultiplexer applications, the input lines are used as the data-select lines. One of the chip select inputs is used as the data-input line, with the other chip select input held LOW to enable the internal negative-AND gate at the bottom of the diagram.

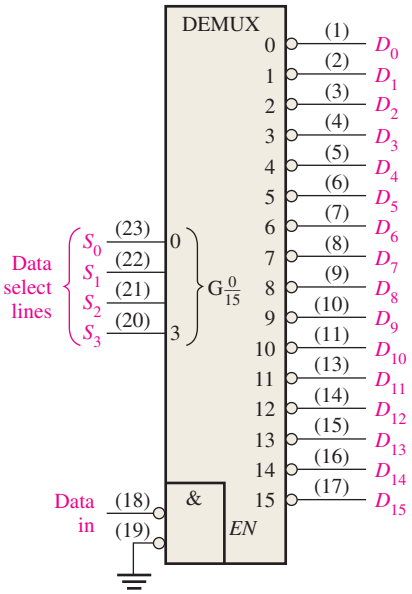


FIGURE 6–54 The decoder used as a demultiplexer.

SECTION 6–9 CHECKUP

1. Generally, how can a decoder be used as a demultiplexer?
2. The demultiplexer in Figure 6–54 has a binary code of 1010 on the data-select lines, and the data-input line is LOW. What are the states of the output lines?

6–10 Parity Generators/Checkers

Errors can occur as digital codes are being transferred from one point to another within a digital system or while codes are being transmitted from one system to another. The errors take the form of undesired changes in the bits that make up the coded information; that is, a 1 can change to a 0, or a 0 to a 1, because of component malfunctions or electrical noise. In most digital systems, the probability that even a single bit error will occur is very small, and the likelihood that more than one will occur is even smaller. Nevertheless, when an error occurs undetected, it can cause serious problems in a digital system.

After completing this section, you should be able to

- Explain the concept of parity
- Implement a basic parity circuit with exclusive-OR gates
- Describe the operation of basic parity generating and checking logic
- Discuss the 74HC280 9-bit parity generator/checker
- Use VHDL to describe a 9-bit parity generator/checker
- Discuss how error detection can be implemented in a data transmission system

The parity method of error detection in which a **parity bit** is attached to a group of information bits in order to make the total number of 1s either even or odd (depending on the system) was covered in Chapter 2. In addition to parity bits, several specific codes also provide inherent error detection.

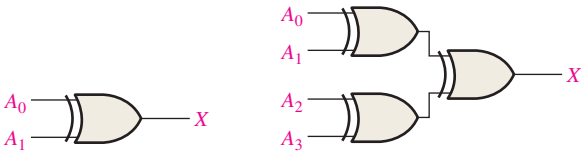
Basic Parity Logic

In order to check for or to generate the proper parity in a given code, a basic principle can be used:

The sum (disregarding carries) of an even number of 1s is always 0, and the sum of an odd number of 1s is always 1.

A parity bit indicates if the number of 1s in a code is even or odd for the purpose of error detection.

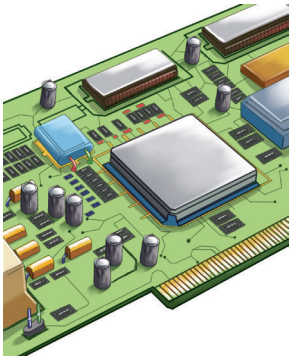
Therefore, to determine if a given code has **even parity** or **odd parity,** all the bits in that code are summed. As you know, the modulo-2 sum of two bits can be generated by an exclusive-OR gate, as shown in Figure 6–55(a); the modulo-2 sum of four bits can be formed by three exclusive-OR gates connected as shown in Figure 6–55(b); and so on. When the number of 1s on the inputs is even, the output X is 0 (LOW). When the number of 1s is odd, the output X is 1 (HIGH).



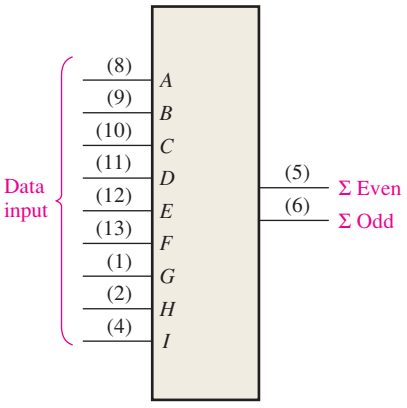
(a) Summing of two bits (b) Summing of four bits

FIGURE 6–55

IMPLEMENTATION: 9-BIT PARITY GENERATOR/CHECKER



Fixed-Function Device The logic symbol and function table for a 74HC280 are shown in Figure 6–56. This particular device can be used to check for odd or even parity on a 9-bit code (eight data bits and one parity bit), or it can be used to generate a parity bit for a binary code with up to nine bits. The inputs are A through I; when there is an even number of 1s on the inputs, the Σ Even output is HIGH and the Σ Odd output is LOW.



(a) Traditional logic symbol

Number of Inputs A–I that Are High	Outputs Σ Even	Outputs Σ Odd
0, 2, 4, 6, 8	H	L
1, 3, 5, 7, 9	L	H

(b) Function table

FIGURE 6–56 The 74HC280 9-bit parity generator/checker.

Parity Checker When this device is used as an even parity checker, the number of input bits should always be even; and when a parity error occurs, the Σ Even output goes LOW and the Σ Odd output goes HIGH. When it is used as an odd parity checker, the number of input bits should always be odd; and when a parity error occurs, the Σ Odd output goes LOW and the Σ Even output goes HIGH.

Parity Generator If this device is used as an even parity generator, the parity bit is taken at the Σ Odd output because this output is a 0 if there is an even number of input bits and it is a 1 if there is an odd number. When used as an odd parity generator, the parity bit is taken at the Σ Even output because it is a 0 when the number of inputs bits is odd.

Programmable Logic Device (PLD) The 9-bit parity generator/checker can be described using VHDL and implemented in a PLD. We will expand the 4-bit logic circuit in Figure 6–55(b) as shown in Figure 6–57. The data flow approach is used.

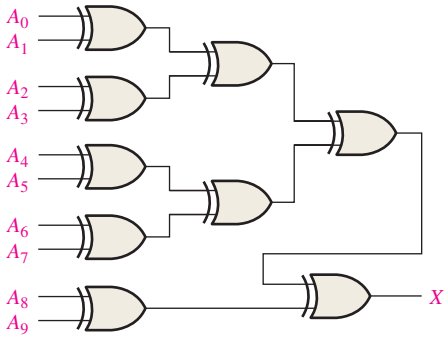


FIGURE 6–57

```
entity ParityCheck is
   port (A0, A1, A2, A3, A4, A5, A6, A7, A8, A9: in bit;
   X: out bit);                                              } Inputs and output declared
end entity ParityCheck;

architecture LogicOperation of ParityCheck is
begin
   X <= ((A0 xor A1) xor (A2 xor A3)) xor ((A4 xor A5) xor   } Output defined by
   (A6 xor A7)) xor (A8 xor A9);                             } Boolean expression
end architecture LogicOperation;
```

A Data Transmission System with Error Detection

A simplified data transmission system is shown in Figure 6–58 to illustrate an application of parity generators/checkers, as well as multiplexers and demultiplexers, and to illustrate the need for data storage in some applications.

In this application, digital data from seven sources are multiplexed onto a single line for transmission to a distant point. The seven data bits (D_0 through D_6) are applied to the multiplexer data inputs and, at the same time, to the even parity generator inputs. The Σ Odd output of the parity generator is used as the even parity bit. This bit is 0 if the number of 1s on the inputs *A* through *I* is even and is a 1 if the number of 1s on *A* through *I* is odd. This bit is D_7 of the transmitted code.

The data-select inputs are repeatedly cycled through a binary sequence, and each data bit, beginning with D_0, is serially passed through and onto the transmission line ($\overline{Y}$). In this example, the transmission line consists of four conductors: one carries the serial data and three carry the timing signals (data selects). There are more sophisticated ways of sending the timing information, but we are using this direct method to illustrate a basic principle.

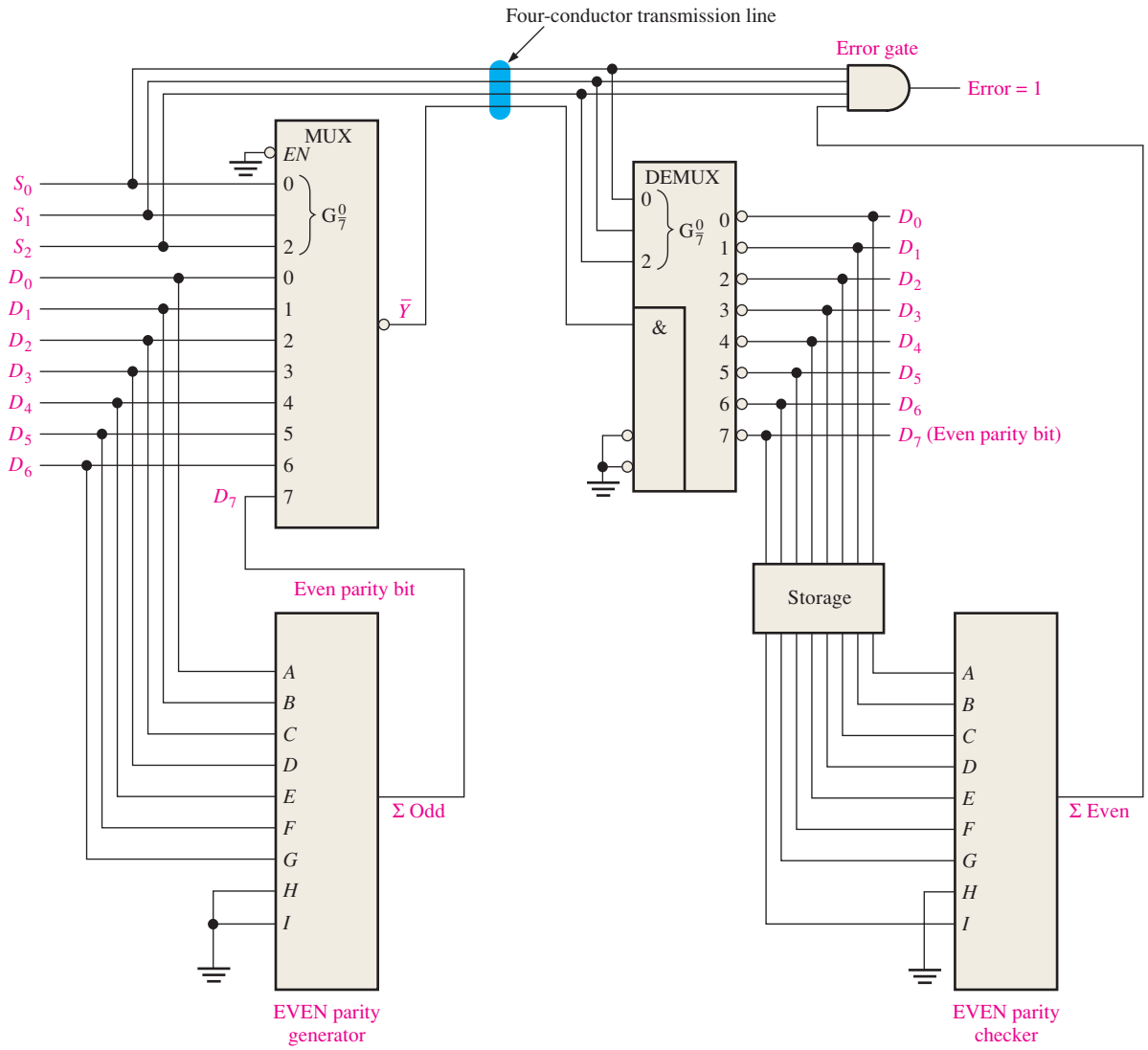


FIGURE 6–58 Simplified data transmission system with error detection.

At the demultiplexer end of the system, the data-select signals and the serial data stream are applied to the demultiplexer. The data bits are distributed by the demultiplexer onto the output lines in the order in which they occurred on the multiplexer inputs. That is, D_0 comes out on the D_0 output, D_1 comes out on the D_1 output, and so on. The parity bit comes out on the D_7 output. These eight bits are temporarily stored and applied to the even parity checker. Not all of the bits are present on the parity checker inputs until the parity bit D_7 comes out and is stored. At this time, the error gate is enabled by the data-select code 111. If the parity is correct, a 0 appears on the Σ Even output, keeping the Error output at 0. If the parity is incorrect, all 1s appear on the error gate inputs, and a 1 on the Error output results.

This particular application has demonstrated the need for data storage. Storage devices will be introduced in Chapter 7 and covered in Chapter 11.

The timing diagram in Figure 6–59 illustrates a specific case in which two 8-bit words are transmitted, one with correct parity and one with an error.

InfoNote

Microprocessors perform internal parity checks as well as parity checks of the external data and address buses. In a read operation, the external system can transfer the parity information together with the data bytes. The microprocessor checks whether the resulting parity is even and sends out the corresponding signal. When it sends out an address code, the microprocessor does not perform an address parity check, but it does generate an even parity bit for the address.

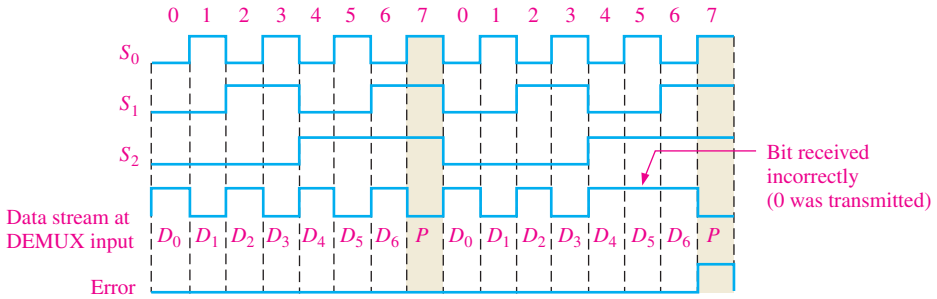


FIGURE 6–59 Example of data transmission with and without error for the system in Figure 6–58.

SECTION 6–10 CHECKUP

1. Add an even parity bit to each of the following codes:
   (a) 110100 **(b)** 01100011
2. Add an odd parity bit to each of the following codes:
   (a) 1010101 **(b)** 1000001
3. Check each of the even parity codes for an error.
   (a) 100010101 **(b)** 1110111001