CHAPTER 5

Combinational Logic Analysis

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Analyze basic combinational logic circuits, such as AND-OR, AND-OR-Invert, exclusive-OR, and exclusive-NOR
- Use AND-OR and AND-OR-Invert circuits to implement sum-of-products (SOP) and product-of-sums (POS) expressions
- Write the Boolean output expression for any combinational logic circuit
- Develop a truth table from the output expression for a combinational logic circuit
- Use the Karnaugh map to expand an output expression containing terms with missing variables into a full SOP form
- Design a combinational logic circuit for a given Boolean output expression
- Design a combinational logic circuit for a given truth table
- Simplify a combinational logic circuit to its minimum form
- Use NAND gates to implement any combinational logic function
- Use NOR gates to implement any combinational logic function
- Analyze the operation of logic circuits with pulse inputs
- Write VHDL programs for simple logic circuits
- Troubleshoot faulty logic circuits
- Troubleshoot logic circuits by using signal tracing and waveform analysis
- Apply combinational logic to an application

KEY TERMS

Key terms are in order of appearance in the chapter.

- Universal gate
- Negative-OR
- Negative-AND
- Component
- Signal
- Node
- Signal tracing

VISIT THE WEBSITE

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

INTRODUCTION

In Chapters 3 and 4, logic gates were discussed on an individual basis and in simple combinations. You were introduced to SOP and POS implementations, which are basic forms of combinational logic. When logic gates are connected together to produce a specified output for certain specified combinations of input variables, with no storage involved, the resulting circuit is in the category of **combinational logic**. In combinational logic, the output level is at all times dependent on the combination of input levels. This chapter expands on the material introduced in earlier chapters with a coverage of the analysis, design, and troubleshooting of various combinational logic circuits. The VHDL structural approach is introduced and applied to combinational logic.

5–1 Basic Combinational Logic Circuits

In Chapter 4, you learned that SOP expressions are implemented with an AND gate for each product term and one OR gate for summing all of the product terms. As you know, this SOP implementation is called AND-OR logic and is the basic form for realizing standard Boolean functions. In this section, the AND-OR and the AND-OR-Invert are examined; the exclusive-OR and exclusive-NOR gates, which are actually a form of AND-OR logic, are also covered.

After completing this section, you should be able to

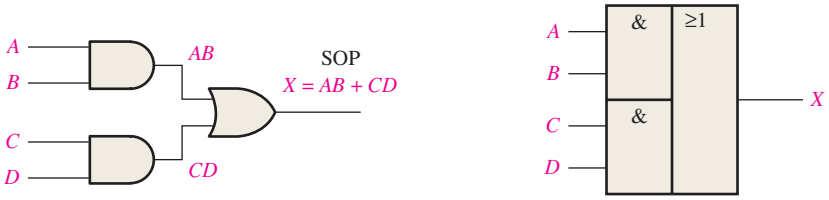
- Analyze and apply AND-OR circuits
- Analyze and apply AND-OR-Invert circuits
- Analyze and apply exclusive-OR gates
- Analyze and apply exclusive-NOR gates

AND-OR Logic

AND-OR logic produces an SOP expression.

Figure 5–1(a) shows an AND-OR circuit consisting of two 2-input AND gates and one 2-input OR gate; Figure 5–1(b) is the ANSI standard rectangular outline symbol. The Boolean expressions for the AND gate outputs and the resulting SOP expression for the output X are shown on the diagram. In general, an AND-OR circuit can have any number of AND gates, each with any number of inputs.

The truth table for a 4-input AND-OR logic circuit is shown in Table 5–1. The intermediate AND gate outputs (the *AB* and *CD* columns) are also shown in the table.



(a) Logic diagram (ANSI standard distinctive shape symbols)

(b) ANSI standard rectangular outline symbol

MultiSim

FIGURE 5–1 An example of AND-OR logic. Open file F05-01 to verify the operation. *A Multisim tutorial is available on the website.*

TABLE 5–1

Truth table for the AND-OR logic in Figure 5–1.

Inputs						Output
A	*B*	*C*	*D*	*AB*	*CD*	*X*
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	1	0	1
1	1	1	1	1	1	1

An AND-OR circuit directly implements an SOP expression, assuming the complements (if any) of the variables are available. The operation of the AND-OR circuit in Figure 5–1 is stated as follows:

For a 4-input AND-OR logic circuit, the output *X* is HIGH (1) if both input *A* and input *B* are HIGH (1) or both input *C* and input *D* are HIGH (1).

EXAMPLE 5–1

In a certain chemical-processing plant, a liquid chemical is used in a manufacturing process. The chemical is stored in three different tanks. A level sensor in each tank produces a HIGH voltage when the level of chemical in the tank drops below a specified point.

Design a circuit that monitors the chemical level in each tank and indicates when the level in any two of the tanks drops below the specified point.

Solution

The AND-OR circuit in Figure 5–2 has inputs from the sensors on tanks *A*, *B*, and *C* as shown. The AND gate G_1 checks the levels in tanks *A* and *B*, gate G_2 checks tanks *A* and *C*, and gate G_3 checks tanks *B* and *C*. When the chemical level in any two of the tanks gets too low, one of the AND gates will have HIGHs on both of its inputs, causing its output to be HIGH; and so the final output *X* from the OR gate is HIGH. This HIGH input is then used to activate an indicator such as a lamp or audible alarm, as shown in the figure.

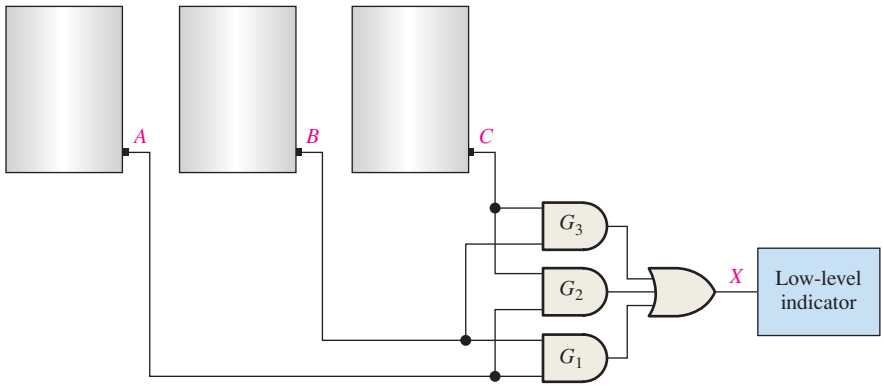


FIGURE 5–2

Related Problem*

Write the Boolean SOP expression for the AND-OR logic in Figure 5–2.

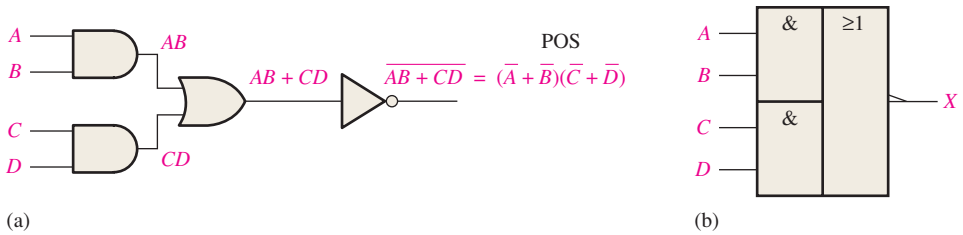
*Answers are at the end of the chapter.

AND-OR-Invert Logic

When the output of an AND-OR circuit is complemented (inverted), it results in an AND-OR-Invert circuit. Recall that AND-OR logic directly implements SOP expressions. POS expressions can be implemented with AND-OR-Invert logic. This is illustrated as follows, starting with a POS expression and developing the corresponding AND-OR-Invert (AOI) expression.

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

The logic diagram in Figure 5–3(a) shows an AND-OR-Invert circuit with four inputs and the development of the POS output expression. The ANSI standard rectangular outline symbol is shown in part (b). In general, an AND-OR-Invert circuit can have any number of AND gates, each with any number of inputs.



MultiSim

FIGURE 5–3 An AND-OR-Invert circuit produces a POS output. Open file F05-03 to verify the operation.

The operation of the AND-OR-Invert circuit in Figure 5–3 is stated as follows:

For a 4-input AND-OR-Invert logic circuit, the output *X* is LOW (0) if both input *A* and input *B* are HIGH (1) or both input *C* and input *D* are HIGH (1).

A truth table can be developed from the AND-OR truth table in Table 5–1 by simply changing all 1s to 0s and all 0s to 1s in the output column.

EXAMPLE 5–2

The sensors in the chemical tanks of Example 5–1 are being replaced by a new model that produces a LOW voltage instead of a HIGH voltage when the level of the chemical in the tank drops below a critical point.

Modify the circuit in Figure 5–2 to operate with the different input levels and still produce a HIGH output to activate the indicator when the level in any two of the tanks drops below the critical point. Show the logic diagram.

Solution

The AND-OR-Invert circuit in Figure 5–4 has inputs from the sensors on tanks *A*, *B*, and *C* as shown. The AND gate G_1 checks the levels in tanks *A* and *B*, gate G_2 checks tanks *A* and *C*, and gate G_3 checks tanks *B* and *C*. When the chemical level in any two of the tanks gets too low, each AND gate will have a LOW on at least one input, causing its output to be LOW and, thus, the final output *X* from the inverter is HIGH. This HIGH output is then used to activate an indicator.

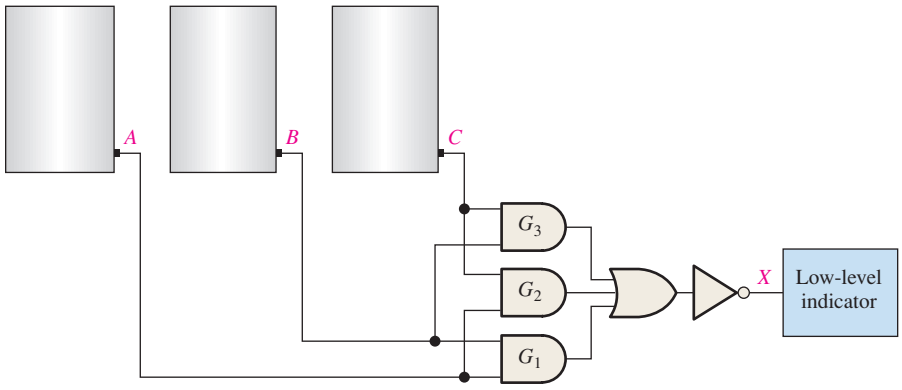


FIGURE 5–4

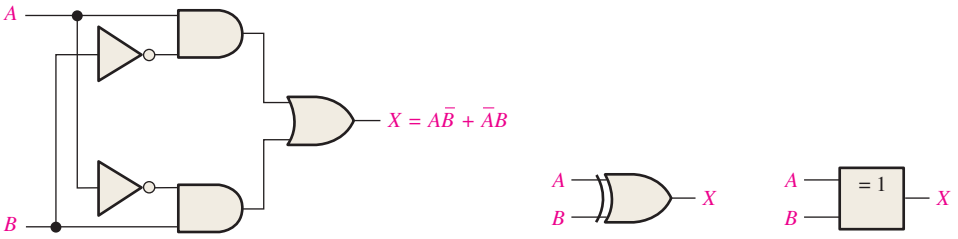
Related Problem

Write the Boolean expression for the AND-OR-Invert logic in Figure 5–4 and show that the output is HIGH (1) when any two of the inputs *A*, *B*, and *C* are LOW (0).

Exclusive-OR Logic

The exclusive-OR gate was introduced in Chapter 3. Although this circuit is considered a type of logic gate with its own unique symbol, it is actually a combination of two AND gates, one OR gate, and two inverters, as shown in Figure 5–5(a). The two ANSI standard exclusive-OR logic symbols are shown in parts (b) and (c).

The XOR gate is actually a combination of other gates.



(a) Logic diagram (b) ANSI distinctive shape symbol (c) ANSI rectangular outline symbol

FIGURE 5–5 Exclusive-OR logic diagram and symbols. Open file F05-05 to verify the operation.

MultiSim

The output expression for the circuit in Figure 5–5 is

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

Evaluation of this expression results in the truth table in Table 5–2. Notice that the output is HIGH only when the two inputs are at opposite levels. A special exclusive-OR operator $\oplus$ is often used, so the expression $X = A\overline{B} + \overline{A}B$ can be stated as "X is equal to A exclusive-OR B" and can be written as

$$X = A \oplus B$$

TABLE 5–2

Truth table for an exclusive-OR.

A	*B*	*X*
0	0	0
0	1	1
1	0	1
1	1	0

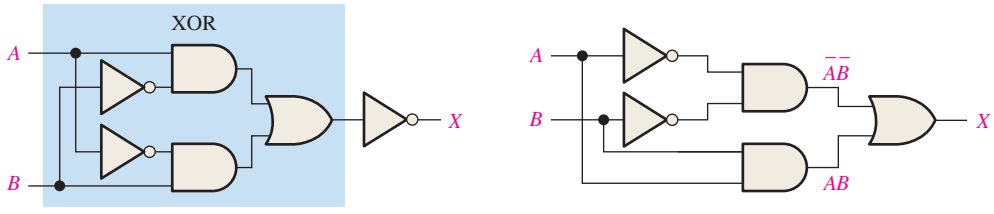
Exclusive-NOR Logic

As you know, the complement of the exclusive-OR function is the exclusive-NOR, which is derived as follows:

$$X = \overline{A\overline{B} + \overline{A}B} = \overline{(A\overline{B})}\,\overline{(\overline{A}B)} = (\overline{A} + B)(A + \overline{B}) = \overline{A}\,\overline{B} + AB$$

Notice that the output X is HIGH only when the two inputs, A and B, are at the same level.

The exclusive-NOR can be implemented by simply inverting the output of an exclusive-OR, as shown in Figure 5–6(a), or by directly implementing the expression $\overline{A}\,\overline{B} + AB$, as shown in part (b).



(a) $X = \overline{A\overline{B} + \overline{A}B}$ (b) $X = \overline{A}\,\overline{B} + AB$

FIGURE 5–6 Two equivalent ways of implementing the exclusive-NOR. Open files F05-06 (a) and (b) to verify the operation.

MultiSim

EXAMPLE 5–3

Use exclusive-OR gates to implement an even-parity code generator for an original 4-bit code.

Solution

Recall from Chapter 2 that a parity bit is added to a binary code in order to provide error detection. For even parity, a parity bit is added to the original code to make the total number of 1s in the code even. The circuit in Figure 5–7 produces a 1 output when there is an odd number of 1s on the inputs in order to make the total number of 1s in the output code even. A 0 output is produced when there is an even number of 1s on the inputs.

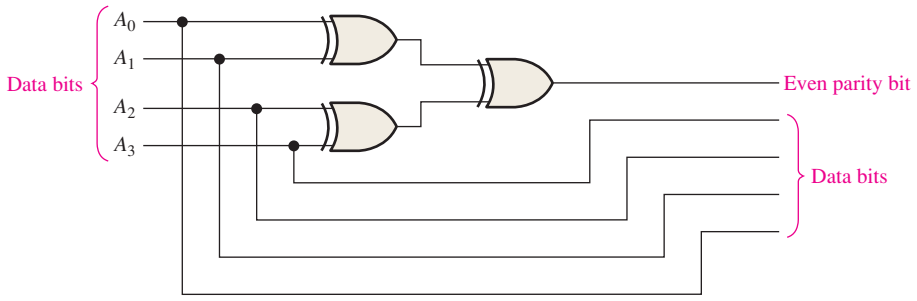


FIGURE 5–7 Even-parity generator.

Related Problem

How would you verify that a correct even-parity bit is generated for each combination of the four data bits?

EXAMPLE 5–4

Use exlusive-OR gates to implement an even-parity checker for the 5-bit code generated by the circuit in Example 5–3.

Solution

The circuit in Figure 5–8 produces a 1 output when there is an error in the five-bit code and a 0 when there is no error.

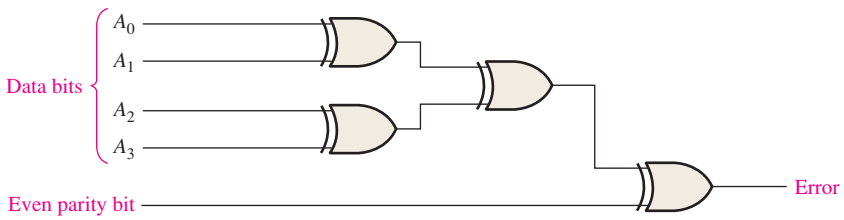


FIGURE 5–8 Even-parity checker.

Related Problem

How would you verify that an error is indicated when the input code is incorrect?

SECTION 5–1 CHECKUP

Answers are at the end of the chapter.

1. Determine the output (1 or 0) of a 4-variable AND-OR-Invert circuit for each of the following input conditions:
   (a) $A = 1, B = 0, C = 1, D = 0$ **(b)** $A = 1, B = 1, C = 0, D = 1$
   (c) $A = 0, B = 1, C = 1, D = 1$
2. Determine the output (1 or 0) of an exclusive-OR gate for each of the following input conditions:
   (a) $A = 1, B = 0$ **(b)** $A = 1, B = 1$
   (c) $A = 0, B = 1$ **(d)** $A = 0, B = 0$
3. Develop the truth table for a certain 3-input logic circuit with the output expression $X = A\overline{B}C + \overline{A}BC + \overline{A}\,\overline{B}\,\overline{C} + AB\overline{C} + ABC$.
4. Draw the logic diagram for an exclusive-NOR circuit.

5–2 Implementing Combinational Logic

In this section, examples are used to illustrate how to implement a logic circuit from a Boolean expression or a truth table. Minimization of a logic circuit using the methods covered in Chapter 4 is also included.

After completing this section, you should be able to

- Implement a logic circuit from a Boolean expression
- Implement a logic circuit from a truth table
- Minimize a logic circuit

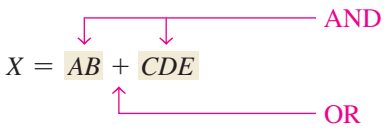
For every Boolean expression there is a logic circuit, and for every logic circuit there is a Boolean expression.

From a Boolean Expression to a Logic Circuit

Let's examine the following Boolean expression:

$$X = AB + CDE$$

A brief inspection shows that this expression is composed of two terms, AB and CDE, with a domain of five variables. The first term is formed by ANDing A with B, and the second term is formed by ANDing C, D, and E. The two terms are then ORed to form the output X. These operations are indicated in the structure of the expression as follows:



Note that in this particular expression, the AND operations forming the two individual terms, AB and CDE, must be performed *before* the terms can be ORed.

To implement this Boolean expression, a 2-input AND gate is required to form the term AB, and a 3-input AND gate is needed to form the term CDE. A 2-input OR gate is then required to combine the two AND terms. The resulting logic circuit is shown in Figure 5–9.

As another example, let's implement the following expression:

$$X = AB(C\overline{D} + EF)$$

InfoNote

Many control programs require logic operations to be performed by a computer. A driver program is a control program that is used with computer peripherals. For example, a mouse driver requires logic tests to determine if a button has been pressed and further logic operations to determine if it has moved, either horizontally or vertically. Within the heart of a microprocessor is the arithmetic logic unit (ALU), which performs these logic operations as directed by program instructions. All of the logic described in this chapter can also be performed by the ALU, given the proper instructions.

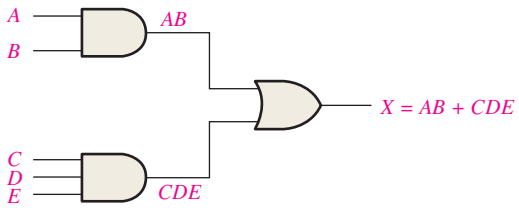
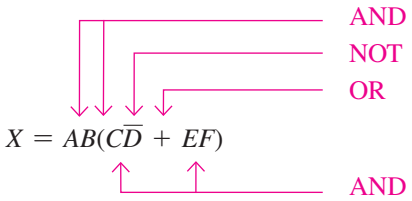


FIGURE 5–9 Logic circuit for $X = AB + CDE$.

A breakdown of this expression shows that the terms AB and $(C\overline{D} + EF)$ are ANDed. The term $C\overline{D} + EF$ is formed by first ANDing C and $\overline{D}$ and ANDing E and F, and then ORing these two terms. This structure is indicated in relation to the expression as follows:



Before you can implement the final expression, you must create the sum term $C\overline{D} + EF$; but before you can get this term; you must create the product terms $C\overline{D}$ and EF; but before you can get the term $C\overline{D}$, you must create $\overline{D}$. So, as you can see, the logic operations must be done in the proper order.

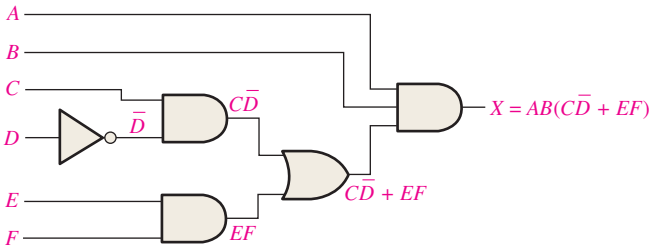
The logic gates required to implement $X = AB(C\overline{D} + EF)$ are as follows:

1. One inverter to form $\overline{D}$
2. Two 2-input AND gates to form $C\overline{D}$ and EF
3. One 2-input OR gate to form $C\overline{D} + EF$
4. One 3-input AND gate to form X

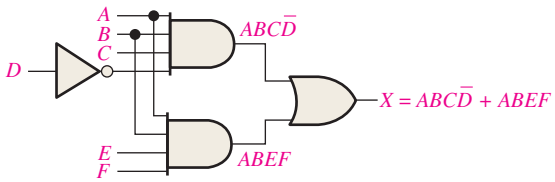
The logic circuit for this expression is shown in Figure 5–10(a). Notice that there is a maximum of four gates and an inverter between an input and output in this circuit (from input D to output). Often the total propagation delay time through a logic circuit is a major consideration. Propagation delays are additive, so the more gates or inverters between input and output, the greater the propagation delay time.

Unless an intermediate term, such as $C\overline{D} + EF$ in Figure 5–10(a), is required as an output for some other purpose, it is usually best to reduce a circuit to its SOP form in order to reduce the overall propagation delay time. The expression is converted to SOP as follows, and the resulting circuit is shown in Figure 5–10(b).

$$AB(C\overline{D} + EF) = ABC\overline{D} + ABEF$$



(a)



(b) Sum-of-products implementation of the circuit in part (a)

FIGURE 5–10 Logic circuits for $X = AB(C\overline{D} + EF) = ABC\overline{D} + ABEF$.

TABLE 5–3

Inputs			Output	
A	*B*	*C*	*X*	Product Term
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC$
1	0	0	1	$A\overline{B}\,\overline{C}$
1	0	1	0	
1	1	0	0	
1	1	1	0	

From a Truth Table to a Logic Circuit

If you begin with a truth table instead of an expression, you can write the SOP expression from the truth table and then implement the logic circuit. Table 5–3 specifies a logic function.

The Boolean SOP expression obtained from the truth table by ORing the product terms for which $X = 1$ is

$$X = \overline{A}BC + A\overline{B}\,\overline{C}$$

The first term in the expression is formed by ANDing the three variables $\overline{A}$, B, and C. The second term is formed by ANDing the three variables A, $\overline{B}$, and $\overline{C}$.

The logic gates required to implement this expression are as follows: three inverters to form the $\overline{A}$, $\overline{B}$, and $\overline{C}$ variables; two 3-input AND gates to form the terms $\overline{A}BC$ and $A\overline{B}\,\overline{C}$; and one 2-input OR gate to form the final output function, $\overline{A}BC + A\overline{B}\,\overline{C}$.

The implementation of this logic function is illustrated in Figure 5–11.

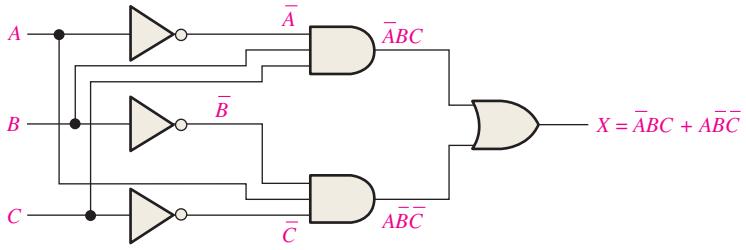


FIGURE 5–11 Logic circuit for $X = \overline{A}BC + A\overline{B}\,\overline{C}$. Open file F05-11 to verify the operation.

MultiSim

EXAMPLE 5–5

Design a logic circuit to implement the operation specified in the truth table of Table 5–4.

TABLE 5–4

Inputs			Output	
A	*B*	*C*	*X*	Product Term
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC$
1	0	0	0	
1	0	1	1	$A\overline{B}C$
1	1	0	1	$AB\overline{C}$
1	1	1	0	

Solution

Notice that $X = 1$ for only three of the input conditions. Therefore, the logic expression is

$$X = \overline{A}BC + A\overline{B}C + AB\overline{C}$$

The logic gates required are three inverters, three 3-input AND gates and one 3-input OR gate. The logic circuit is shown in Figure 5–12.

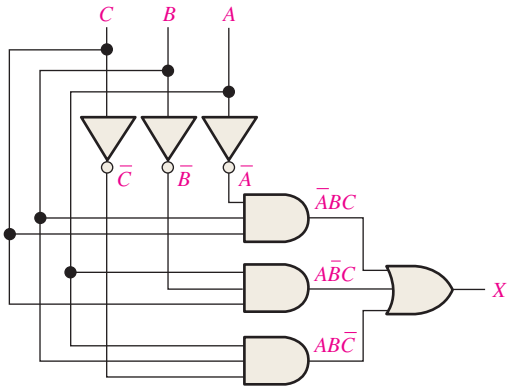


FIGURE 5–12 Open file F05-12 to verify the operation.

MultiSim

Related Problem

Determine if the logic circuit of Figure 5–12 can be simplified.

EXAMPLE 5–6

Develop a logic circuit with four input variables that will only produce a 1 output when exactly three input variables are 1s.

Solution

Out of sixteen possible combinations of four variables, the combinations in which there are exactly three 1s are listed in Table 5–5, along with the corresponding product term for each.

TABLE 5–5

A	*B*	*C*	*D*	Product Term
0	1	1	1	$\overline{A}BCD$
1	0	1	1	$A\overline{B}CD$
1	1	0	1	$AB\overline{C}D$
1	1	1	0	$ABC\overline{D}$

The product terms are ORed to get the following expression:

$$X = \overline{A}BCD + A\overline{B}CD + AB\overline{C}D + ABC\overline{D}$$

This expression is implemented in Figure 5–13 with AND-OR logic.

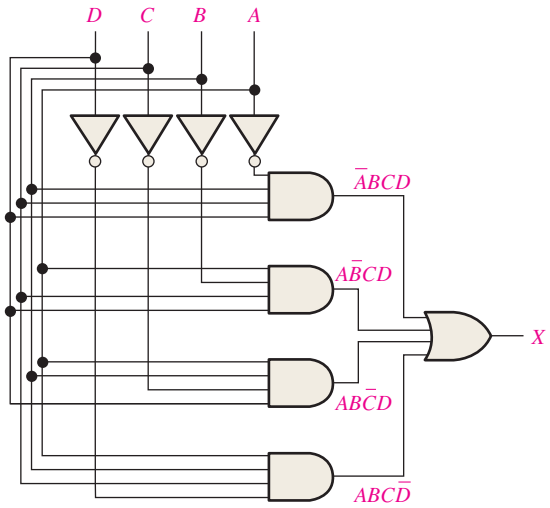


FIGURE 5–13 Open file F05-13 to verify the operation.

MultiSim

Related Problem

Determine if the logic circuit of Figure 5–13 can be simplified.

EXAMPLE 5–7

Reduce the combinational logic circuit in Figure 5–14 to a minimum form.

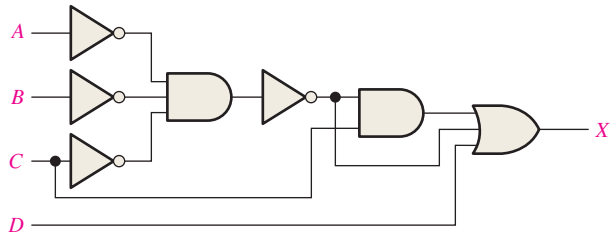


FIGURE 5–14
Open file F05-14 to verify that this circuit is equivalent to the gate in Figure 5–15.

MultiSim

Solution

The expression for the output of the circuit is

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

Applying DeMorgan's theorem and Boolean algebra,

$$\begin{aligned} X &= (\overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}})C + \overline{\overline{A}} + \overline{\overline{B}} + \overline{\overline{C}} + D \\ &= AC + BC + CC + A + B + C + D \\ &= AC + BC + C + A + B + \not{C} + D \\ &= C(A + B + 1) + A + B + D \\ X &= A + B + C + D \end{aligned}$$

The simplified circuit is a 4-input OR gate as shown in Figure 5–15.

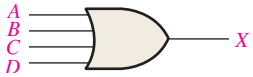


FIGURE 5–15

Related Problem

Verify the minimized expression $A + B + C + D$ using a Karnaugh map.

EXAMPLE 5–8

Minimize the combinational logic circuit in Figure 5–16. Inverters for the complemented variables are not shown.

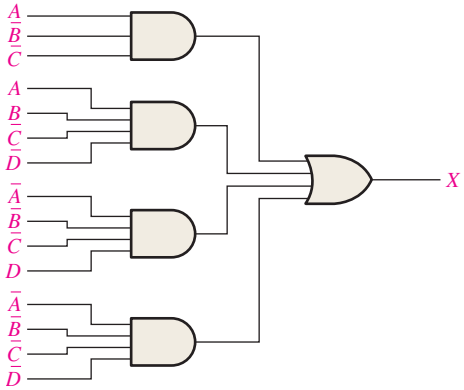


FIGURE 5–16

Solution

The output expression is

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

Expanding the first term to include the missing variables D and $\overline{D}$,

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

This expanded SOP expression is mapped and simplified on the Karnaugh map in Figure 5–17(a). The simplified implementation is shown in part (b). Inverters are not shown.

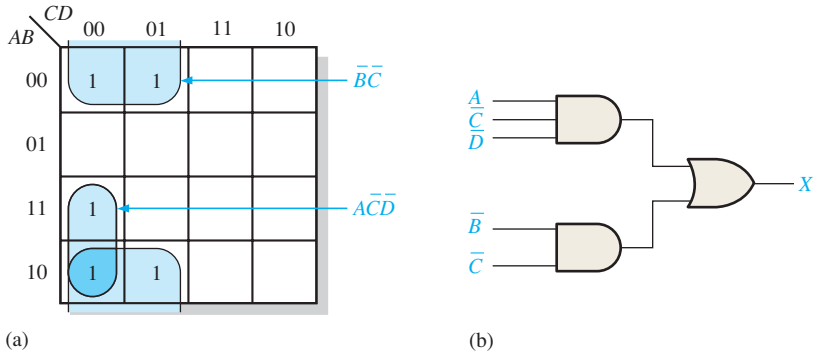


FIGURE 5–17

Related Problem

Develop the POS equivalent of the circuit in Figure 5–17(b). See Section 4–10.

SECTION 5–2 CHECKUP

1. Implement the following Boolean expressions as they are stated:
   (a) $X = ABC + AB + AC$ **(b)** $X = AB(C + DE)$
2. Develop a logic circuit that will produce a 1 on its output only when all three inputs are 1s or when all three inputs are 0s.
3. Reduce the circuits in Question 1 to minimum SOP form.

5–3 The Universal Property of NAND and NOR Gates

Up to this point, you have studied combinational circuits implemented with AND gates, OR gates, and inverters. In this section, the universal property of the NAND gate and the NOR gate is discussed. The universality of the NAND gate means that it can be used as an inverter and that combinations of NAND gates can be used to implement the AND, OR, and NOR operations. Similarly, the NOR gate can be used to implement the inverter (NOT), AND, OR, and NAND operations.

After completing this section, you should be able to

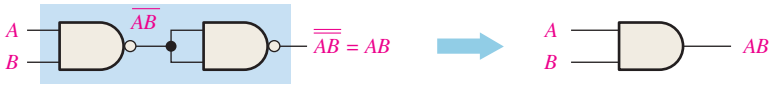
- Use NAND gates to implement the inverter, the AND gate, the OR gate, and the NOR gate
- Use NOR gates to implement the inverter, the AND gate, the OR gate, and the NAND gate

The NAND Gate as a Universal Logic Element

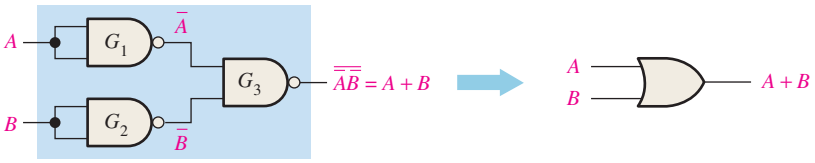
Combinations of NAND gates can be used to produce any logic function.

The NAND gate is a **universal gate** because it can be used to produce the NOT, the AND, the OR, and the NOR functions. An inverter can be made from a NAND gate by connecting all of the inputs together and creating, in effect, a single input, as shown in Figure 5–18(a) for a 2-input gate. An AND function can be generated by the use of NAND gates alone, as shown in Figure 5–18(b). An OR function can be produced with only NAND gates, as illustrated in part (c). Finally, a NOR function is produced as shown in part (d).

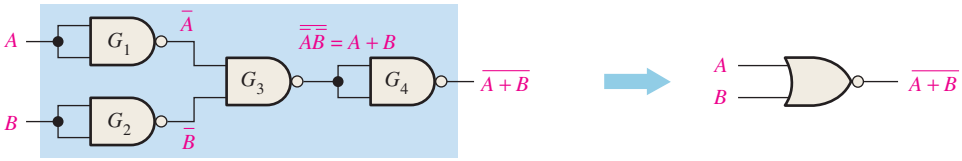
(a) One NAND gate used as an inverter



(b) Two NAND gates used as an AND gate



(c) Three NAND gates used as an OR gate



(d) Four NAND gates used as a NOR gate

FIGURE 5–18 Universal application of NAND gates. Open files F05-18(a), (b), (c), and (d) to verify each of the equivalencies.

MultiSim

In Figure 5–18(b), a NAND gate is used to invert (complement) a NAND output to form the AND function, as indicated in the following equation:

$$X = \overline{\overline{AB}} = AB$$

In Figure 5–18(c), NAND gates G_1 and G_2 are used to invert the two input variables before they are applied to NAND gate G_3. The final OR output is derived as follows by application of DeMorgan's theorem:

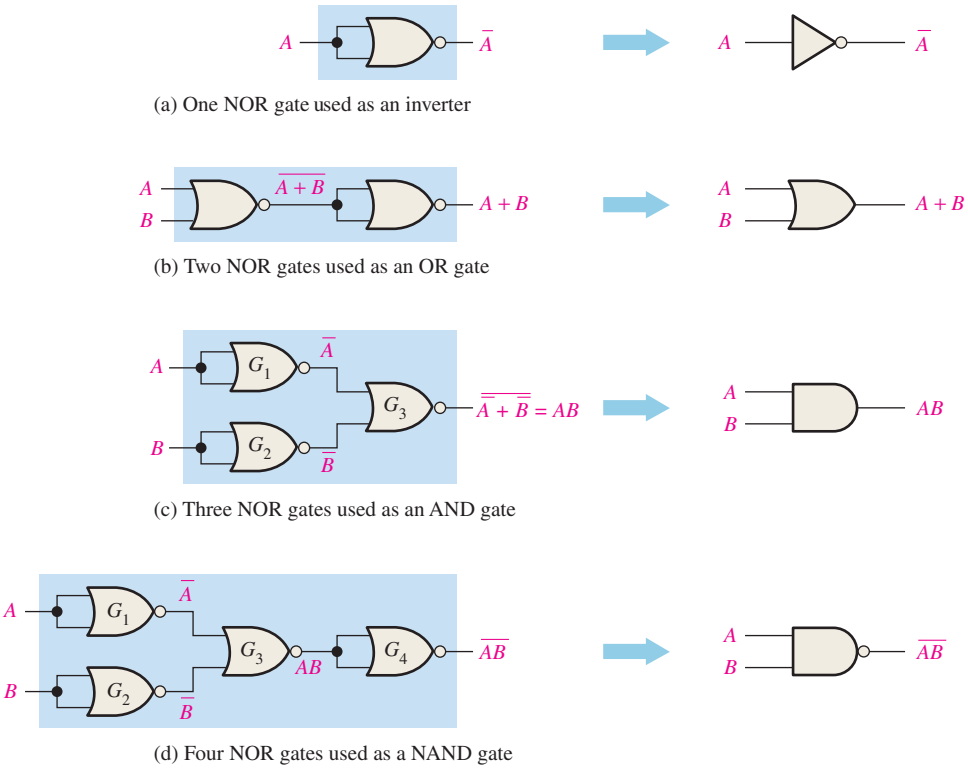
$$X = \overline{\overline{A}\,\overline{B}} = A + B$$

In Figure 5–18(d), NAND gate G_4 is used as an inverter connected to the circuit of part (c) to produce the NOR operation $\overline{A + B}$.

The NOR Gate as a Universal Logic Element

Combinations of NOR gates can be used to produce any logic function.

Like the NAND gate, the NOR gate can be used to produce the NOT, AND, OR, and NAND functions. A NOT circuit, or inverter, can be made from a NOR gate by connecting all of the inputs together to effectively create a single input, as shown in Figure 5–19(a) with a 2-input example. Also, an OR gate can be produced from NOR gates, as illustrated in Figure 5–19(b). An AND gate can be constructed by the use of NOR gates, as shown in



(a) One NOR gate used as an inverter

(b) Two NOR gates used as an OR gate

(c) Three NOR gates used as an AND gate

(d) Four NOR gates used as a NAND gate

MultiSim

FIGURE 5–19 Universal application of NOR gates. Open files F05-19(a), (b), (c), and (d) to verify each of the equivalencies.

Figure 5–19(c). In this case the NOR gates G_1 and G_2 are used as inverters, and the final output is derived by the use of DeMorgan's theorem as follows:

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

Figure 5–19(d) shows how NOR gates are used to form a NAND function.

SECTION 5–3 CHECKUP

1. Use NAND gates to implement each expression:
   (a) $X = \overline{A} + B$ **(b)** $X = A\overline{B}$
2. Use NOR gates to implement each expression:
   (a) $X = \overline{A} + B$ **(b)** $X = A\overline{B}$

5–4 Combinational Logic Using NAND and NOR Gates

In this section, you will see how NAND and NOR gates can be used to implement a logic function. Recall from Chapter 3 that the NAND gate also exhibits an equivalent operation called the negative-OR and that the NOR gate exhibits an equivalent operation called the negative-AND. You will see how the use of the appropriate symbols to represent the equivalent operations makes "reading" a logic diagram easier.

After completing this section, you should be able to

- Use NAND gates to implement a logic function
- Use NOR gates to implement a logic function
- Use the appropriate dual symbol in a logic diagram

NAND Logic

As you have learned, a NAND gate can function as either a NAND or a negative-OR because, by DeMorgan's theorem,

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

NAND — negative-OR

Consider the NAND logic in Figure 5–20. The output expression is developed in the following steps:

$$\begin{aligned} X &= \overline{(\overline{AB})(\overline{CD})} \\ &= \overline{(\overline{A} + \overline{B})(\overline{C} + \overline{D})} \\ &= (\overline{\overline{A} + \overline{B}}) + (\overline{\overline{C} + \overline{D}}) \\ &= \overline{\overline{A}}\,\overline{\overline{B}} + \overline{\overline{C}}\,\overline{\overline{D}} \\ &= AB + CD \end{aligned}$$

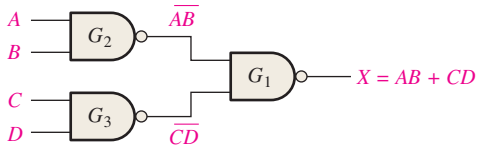
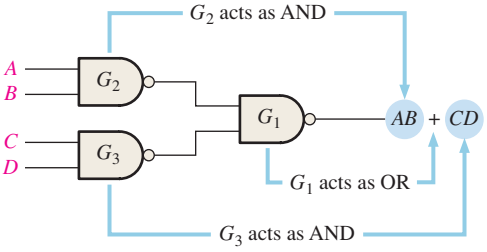


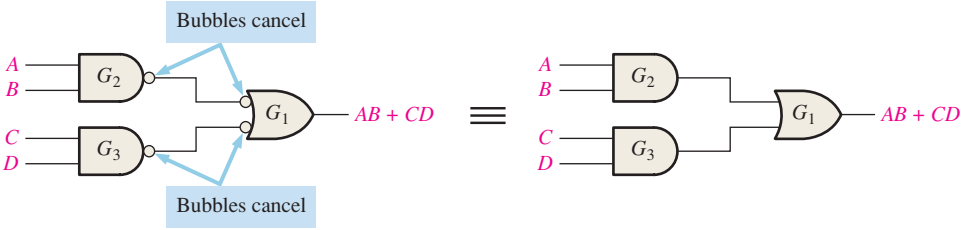
FIGURE 5–20 NAND logic for $X = AB + CD$.

As you can see in Figure 5–20, the output expression, $AB + CD$, is in the form of two AND terms ORed together. This shows that gates G_2 and G_3 act as AND gates and that gate G_1 acts as an OR gate, as illustrated in Figure 5–21(a). This circuit is redrawn in part (b) with NAND symbols for gates G_2 and G_3 and a negative-OR symbol for gate G_1.

Notice in Figure 5–21(b) the bubble-to-bubble connections between the outputs of gates G_2 and G_3 and the inputs of gate G_1. *Since a bubble represents an inversion, two*



(a) Original NAND logic diagram showing effective gate operation relative to the output expression



(b) Equivalent NAND/Negative-OR logic diagram

(c) AND-OR equivalent

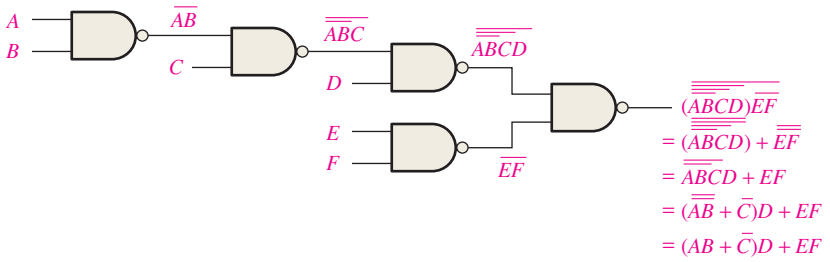
FIGURE 5–21 Development of the AND-OR equivalent of the circuit in Figure 5–20.

connected bubbles represent a double inversion and therefore cancel each other. This inversion cancellation can be seen in the previous development of the output expression $AB + CD$ and is indicated by the absence of barred terms in the output expression. Thus, the circuit in Figure 5–21(b) is *effectively* an AND-OR circuit, as shown in Figure 5–21(c).

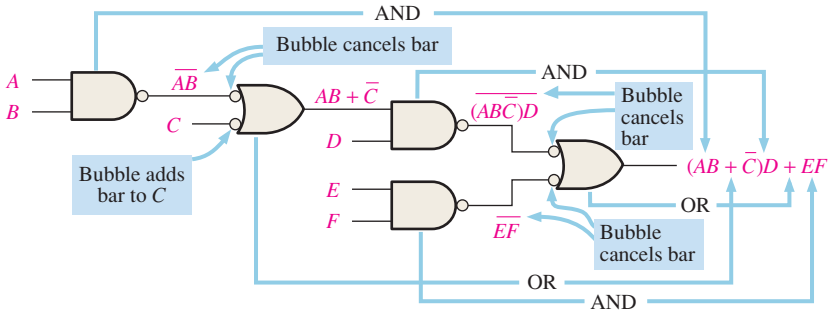
NAND Logic Diagrams Using Dual Symbols

All logic diagrams using NAND gates should be drawn with each gate represented by either a NAND symbol or the equivalent negative-OR symbol to reflect the operation of the gate within the logic circuit. The NAND symbol and the **negative-OR** symbol are called *dual symbols.* When drawing a NAND logic diagram, always use the gate symbols in such a way that every connection between a gate output and a gate input is either bubble-to-bubble or nonbubble-to-nonbubble. In general, a bubble output should not be connected to a nonbubble input or vice versa in a logic diagram.

Figure 5–22 shows an arrangement of gates to illustrate the procedure of using the appropriate dual symbols for a NAND circuit with several gate levels. Although using all NAND symbols as in Figure 5–22(a) is correct, the diagram in part (b) is much easier to "read" and is the preferred method. As shown in Figure 5–22(b), the output gate is represented with a negative-OR symbol. Then the NAND symbol is used for the level of gates right before the output gate and the symbols for successive levels of gates are alternated as you move away from the output.



(a) Several Boolean steps are required to arrive at final output expression.



(b) Output expression can be obtained directly from the function of each gate symbol in the diagram.

FIGURE 5–22 Illustration of the use of the appropriate dual symbols in a NAND logic diagram.

The shape of the gate indicates the way its inputs will appear in the output expression and thus shows how the gate functions within the logic circuit. For a NAND symbol, the inputs appear ANDed in the output expression; and for a negative-OR symbol, the inputs appear ORed in the output expression, as Figure 5–22(b) illustrates. The dual-symbol diagram in part (b) makes it easier to determine the output expression directly from the logic diagram because each gate symbol indicates the relationship of its input variables as they appear in the output expression.

EXAMPLE 5–9

Redraw the logic diagram and develop the output expression for the circuit in Figure 5–23 using the appropriate dual symbols.

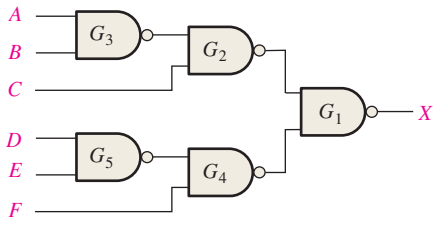


FIGURE 5–23

Solution

Redraw the logic diagram in Figure 5–23 with the use of equivalent negative-OR symbols as shown in Figure 5–24. Writing the expression for X directly from the indicated logic operation of each gate gives $X = (\overline{A} + \overline{B})C + (\overline{D} + \overline{E})F$.

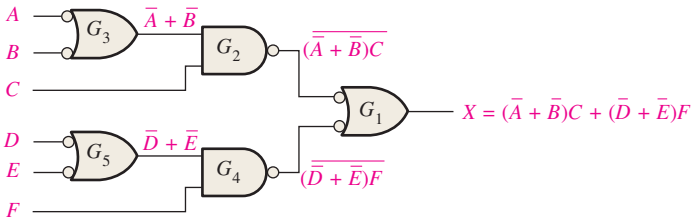


FIGURE 5–24

Related Problem

Derive the output expression from Figure 5–23 and show it is equivalent to the expression in the solution.

EXAMPLE 5–10

Implement each expression with NAND logic using appropriate dual symbols:

(a) $ABC + DE$ **(b)** $ABC + \overline{D} + \overline{E}$

Solution

See Figure 5–25.

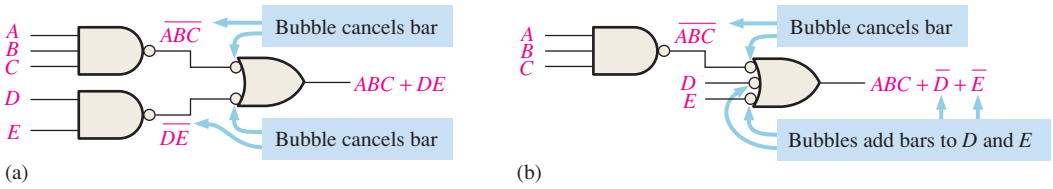


FIGURE 5–25

Related Problem

Convert the NAND circuits in Figure 5–25(a) and (b) to equivalent AND-OR logic.

NOR Logic

A NOR gate can function as either a NOR or a **negative-AND**, as shown by DeMorgan's theorem.

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

NOR — negative-AND

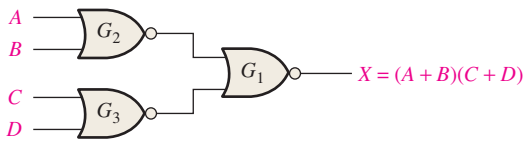


FIGURE 5–26 NOR logic for $X = (A + B)(C + D)$.

Consider the NOR logic in Figure 5–26. The output expression is developed as follows:

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

As you can see in Figure 5–26, the output expression $(A + B)(C + D)$ consists of two OR terms ANDed together. This shows that gates G_2 and G_3 act as OR gates and gate G_1 acts as an AND gate, as illustrated in Figure 5–27(a). This circuit is redrawn in part (b) with a negative-AND symbol for gate G_1.

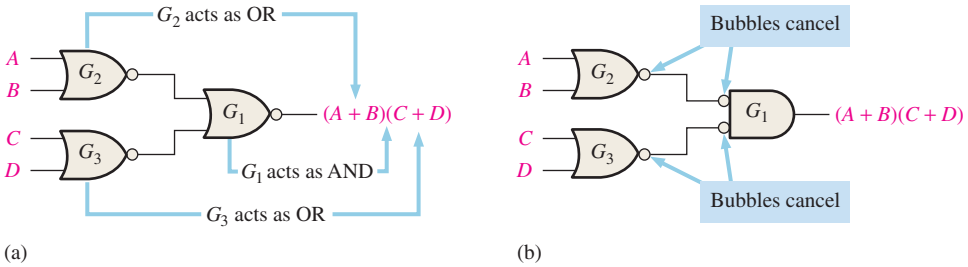
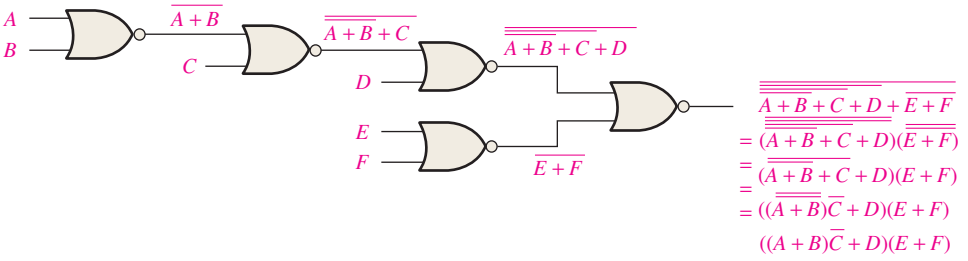


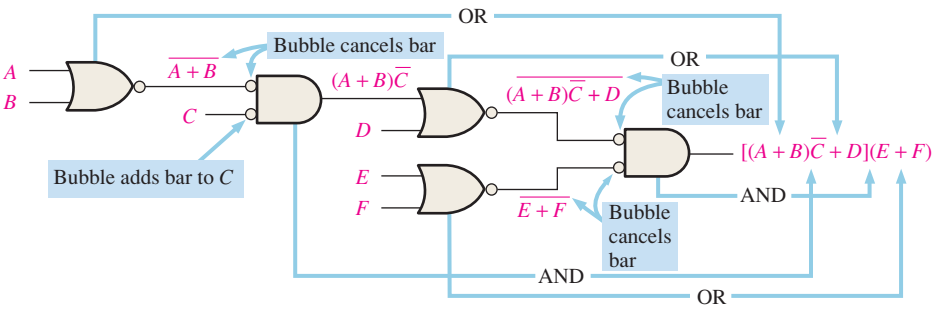
FIGURE 5–27

NOR Logic Diagram Using Dual Symbols

As with NAND logic, the purpose for using the dual symbols is to make the logic diagram easier to read and analyze, as illustrated in the NOR logic circuit in Figure 5–28. When the circuit in part (a) is redrawn with dual symbols in part (b), notice that all output-to-input



(a) Final output expression is obtained after several Boolean steps.



(b) Output expression can be obtained directly from the function of each gate symbol in the diagram.

FIGURE 5–28 Illustration of the use of the appropriate dual symbols in a NOR logic diagram.

connections between gates are bubble-to-bubble or nonbubble-to-nonbubble. Again, you can see that the shape of each gate symbol indicates the type of term (AND or OR) that it produces in the output expression, thus making the output expression easier to determine and the logic diagram easier to analyze.

EXAMPLE 5–11

Using appropriate dual symbols, redraw the logic diagram and develop the output expression for the circuit in Figure 5–29.

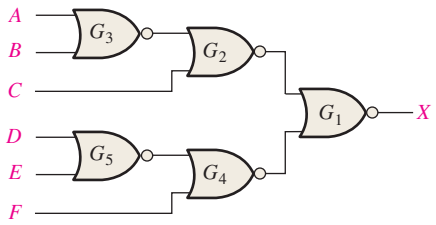


FIGURE 5–29

Solution

Redraw the logic diagram with the equivalent negative-AND symbols as shown in Figure 5–30. Writing the expression for X directly from the indicated operation of each gate,

$$X = (\overline{A}\,\overline{B} + C)(\overline{D}\,\overline{E} + F)$$

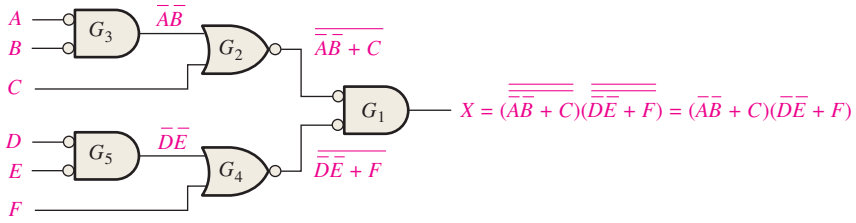


FIGURE 5–30

Related Problem

Prove that the output of the NOR circuit in Figure 5–29 is the same as for the circuit in Figure 5–30.

SECTION 5–4 CHECKUP

1. Implement the expression $X = \overline{(\overline{A} + \overline{B} + \overline{C})DE}$ by using NAND logic.
2. Implement the expression $X = \overline{\overline{A}\,\overline{B}\,\overline{C} + (D + E)}$ with NOR logic.

5–5 Pulse Waveform Operation

General combinational logic circuits with pulse waveform inputs are examined in this section. Keep in mind that the operation of each gate is the same for pulse waveform inputs as for constant-level inputs. The output of a logic circuit at any given time depends on the inputs at that particular time, so the relationship of the time-varying inputs is of primary importance.

After completing this section, you should be able to

- Analyze combinational logic circuits with pulse waveform inputs
- Develop a timing diagram for any given combinational logic circuit with specified inputs

The operation of any gate is the same regardless of whether its inputs are pulsed or constant levels. The nature of the inputs (pulsed or constant levels) does not alter the truth table of a circuit. The examples in this section illustrate the analysis of combinational logic circuits with pulse waveform inputs.

The following is a review of the operation of individual gates for use in analyzing combinational circuits with pulse waveform inputs:

1. The output of an AND gate is HIGH only when all inputs are HIGH at the same time.
2. The output of an OR gate is HIGH only when at least one of its inputs is HIGH.
3. The output of a NAND gate is LOW only when all inputs are HIGH at the same time.
4. The output of a NOR gate is LOW only when at least one of its inputs is HIGH.

EXAMPLE 5–12

Determine the final output waveform X for the circuit in Figure 5–31, with input waveforms A, B, and C as shown.

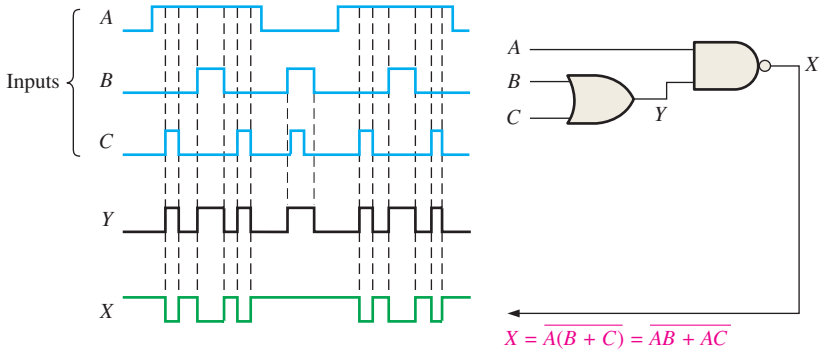


FIGURE 5–31

Solution

The output expression, $\overline{AB + AC}$, indicates that the output X is LOW when both A and B are HIGH or when both A and C are HIGH or when all inputs are HIGH. The output waveform X is shown in the timing diagram of Figure 5–31. The intermediate waveform Y at the output of the OR gate is also shown.

Related Problem

Determine the output waveform if input A is a constant HIGH level.

EXAMPLE 5–13

Draw the timing diagram for the circuit in Figure 5–32 showing the outputs of G_1, G_2, and G_3 with the input waveforms, A, and B, as indicated.

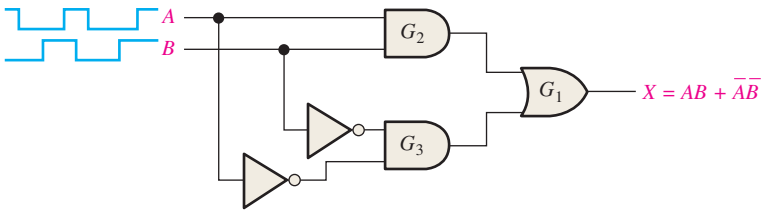


FIGURE 5–32

Solution

When both inputs are HIGH or when both inputs are LOW, the output X is HIGH as shown in Figure 5–33. Notice that this is an exclusive-NOR circuit. The intermediate outputs of gates G_2 and G_3 are also shown in Figure 5–33.

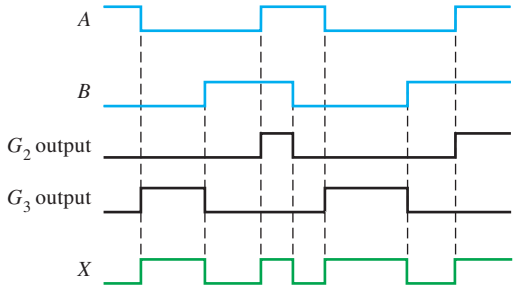


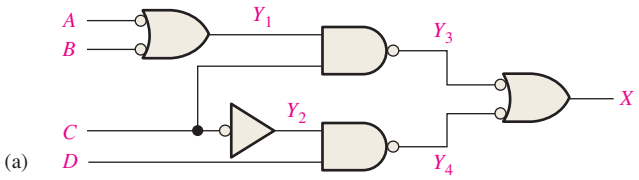
FIGURE 5–33

Related Problem

Determine the output X in Figure 5–32 if input B is inverted.

EXAMPLE 5–14

Determine the output waveform X for the logic circuit in Figure 5–34(a) by first finding the intermediate waveform at each of points Y_1, Y_2, Y_3, and Y_4. The input waveforms are shown in Figure 5–34(b).



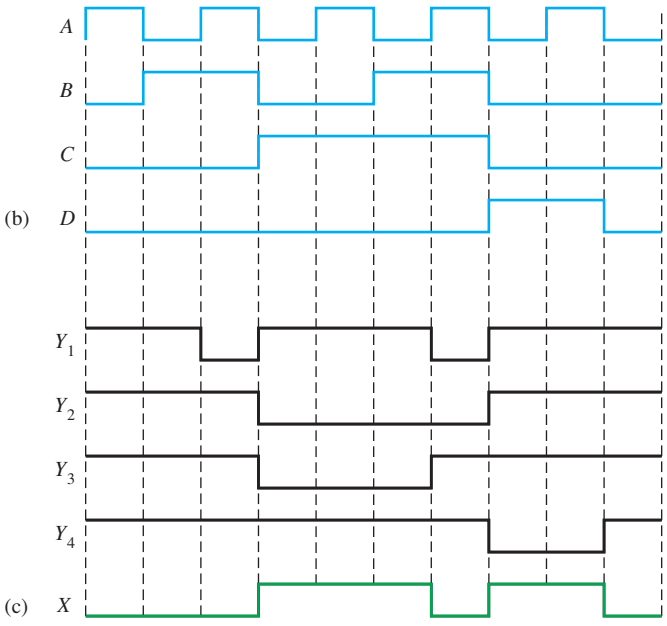


FIGURE 5–34

Solution

All the intermediate waveforms and the final output waveform are shown in the timing diagram of Figure 5–34(c).

Related Problem

Determine the waveforms Y_1, Y_2, Y_3, Y_4 and X if input waveform A is inverted.

EXAMPLE 5–15

Determine the output waveform X for the circuit in Example 5–14, Figure 5–34(a), directly from the output expression.

Solution

The output expression for the circuit is developed in Figure 5–35. The SOP form indicates that the output is HIGH when A is LOW and C is HIGH or when B is LOW and C is HIGH or when C is LOW and D is HIGH.

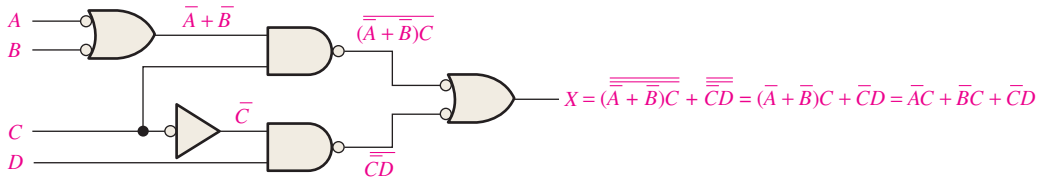


FIGURE 5–35

The result is shown in Figure 5–36 and is the same as the one obtained by the intermediate-waveform method in Example 5–14. The corresponding product terms for each waveform condition that results in a HIGH output are indicated.

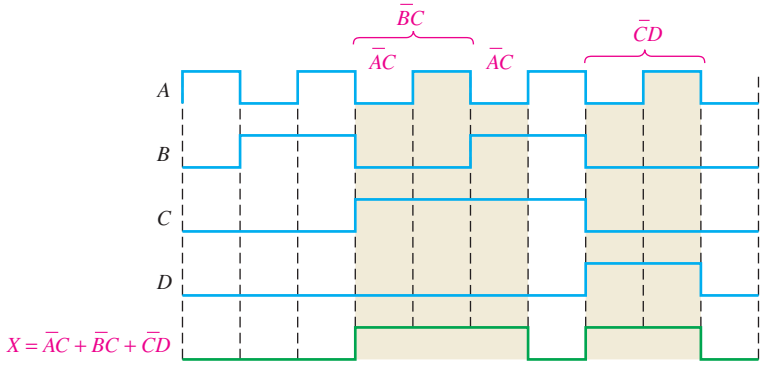


FIGURE 5–36

Related Problem

Repeat this example if all the input waveforms are inverted.

SECTION 5–5 CHECKUP

1. One pulse with $t_W = 50\ \mu s$ is applied to one of the inputs of an exclusive-OR circuit. A second positive pulse with $t_W = 10\ \mu s$ is applied to the other input beginning 15 μs after the leading edge of the first pulse. Show the output in relation to the inputs.
2. The pulse waveforms A and B in Figure 5–31 are applied to the exclusive-NOR circuit in Figure 5–32. Develop a complete timing diagram.

5–6 Combinational Logic with VHDL

The purpose of describing logic using VHDL is so that it can be programmed into a PLD. The data flow approach to writing a VHDL program was described in Chapter 4. In this section, both the data flow approach using Boolean expressions and the structural approach are used to develop VHDL code for describing logic circuits. The VHDL component is introduced and used to illustrate structural descriptions. Some aspects of software development tools are discussed.

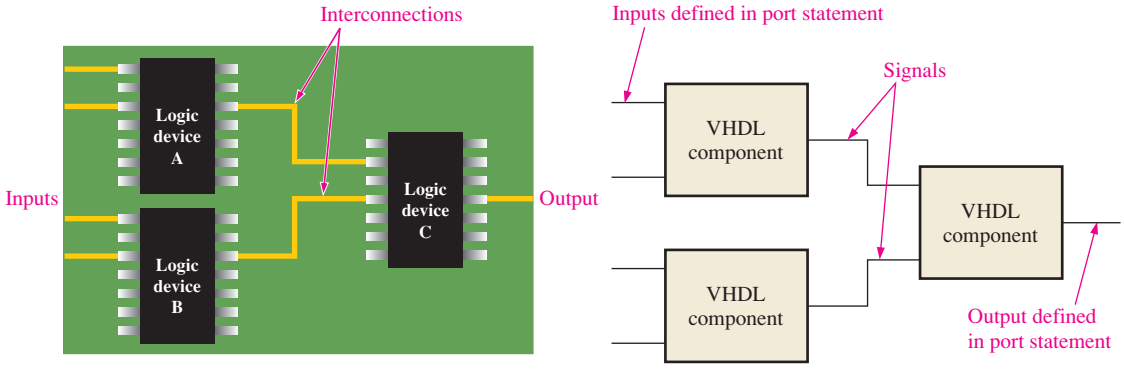
After completing this section, you should be able to

- Describe a VHDL component and discuss how it is used in a program
- Apply the structural approach and the data flow approach to writing VHDL code
- Describe two basic software development tools

Structural Approach to VHDL Programming

The structural approach to writing a VHDL description of a logic function can be compared to installing IC devices on a circuit board and interconnecting them with wires. With the structural approach, you describe logic functions and specify how they are connected together. The VHDL **component** is a way to predefine a logic function for repeated use in a program or in other programs. The component can be used to describe anything from a simple logic gate to a complex logic function. The VHDL **signal** can be thought of as a way to specify a "wire" connection between components.

Figure 5–37 provides a simplified comparison of the structural approach to a hardware implementation on a circuit board.



(a) Hardware implementation with fixed-function logic

(b) VHDL structural implementation

FIGURE 5–37 Simplified comparison of the VHDL structural approach to a hardware implementation. The VHDL signals correspond to the interconnections on the circuit board, and the VHDL components correspond to the 74 series IC devices.

VHDL Components

A VHDL component describes predefined logic that can be stored as a package declaration in a VHDL library and called as many times as necessary in a program. You can use components to avoid repeating the same code over and over within a program. For example, you can create a VHDL component for an AND gate and then use it as many times as you wish without having to write a program for an AND gate every time you need one.

VHDL components are stored and are available for use when you write a program. This is similar to having, for example, a storage bin of ICs available when you are constructing a circuit. Every time you need to use one in your circuit, you reach into the storage bin and place it on the circuit board.

The VHDL program for any logic function can become a component and used whenever necessary in a larger program with the use of a component declaration of the following general form. **Component** is a VHDL keyword.

```
component name_of_component is
   port (port definitions);
end component name_of_component;
```

For simplicity, let's assume that there are predefined VHDL descriptions of a 2-input AND gate with the entity name AND_gate and a 2-input OR gate with the entity name OR_gate, as shown in Figure 5–38.

2-input AND gate

```
entity AND_gate is
   port (A, B: in bit; X: out bit);
end entity AND_gate;

architecture ANDfunction of AND_gate is
begin
   X <= A and B;
end architecture ANDfunction;
```

2-input OR gate

```
entity OR_gate is
   port (A, B: in bit; X: out bit);
end entity OR_gate;

architecture ORfunction of OR_gate is
begin
   X <= A or B;
end architecture ORfunction;
```

FIGURE 5–38 Predefined programs for a 2-input AND gate and a 2-input OR gate to be used as components in the structural approach.

Using Components in a Program

Assume that you are writing a program for a logic circuit that has several AND gates. Instead of rewriting the program in Figure 5–38 over and over, you can use a component declaration to specify the AND gate. The port statement in the component declaration must correspond to the port statement in the entity declaration of the AND gate.

```
component AND_gate is
   port (A, B: in bit; X: out bit);
end component AND_gate;
```

To use a component in a program, you must write a component instantiation statement for each instance in which the component is used. You can think of a component instantiation as a request or call for the component to be used in the main program. For example, the simple SOP logic circuit in Figure 5–39 has two AND gates and one OR gate. Therefore, the VHDL program for this circuit will have two components and three component instantiations or calls.

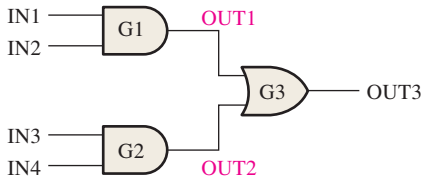


FIGURE 5–39

Signals

In VHDL, signals are analogous to wires that interconnect components on a circuit board. The signals in Figure 5–39 are named OUT1 and OUT2. Signals are the *internal* connections in the logic circuit and are treated differently than the inputs and outputs. Whereas the inputs and outputs are declared in the entity declaration using the port statement, the signals are declared within the architecture using the signal statement. **Signal** is a VHDL keyword.

The Program

The program for the logic in Figure 5–39 begins with an entity declaration as follows:

```
entity AND_OR_Logic is
   port (IN1, IN2, IN3, IN4: in bit; OUT3: out bit);
end entity AND_OR_Logic;
```

The architecture declaration contains the component declarations for the AND gate and the OR gate, the signal definitions, and the component instantiations.

```
architecture LogicOperation of AND_OR_Logic is

  component AND_gate is
     port (A, B: in bit; X: out bit);
  end component AND_gate;

  component OR_gate is
     port (A, B: in bit; X: out bit);
  end component OR_gate;

  signal OUT1, OUT2: bit;

begin
  G1: AND_gate port map (A => IN1, B => IN2, X => OUT1);
  G2: AND_gate port map (A => IN3, B => IN4, X => OUT2);
  G3: OR_gate port map (A => OUT1, B => OUT2, X => OUT3);

end architecture LogicOperation;
```

Component declaration for the AND gate

Component declaration for the OR gate

Signal declaration

Component instantiations describe how the three gates are connected.

Component Instantiations

Let's look at the component instantiations. First, notice that the component instantiations appear between the keyword **begin** and the **end architecture** statement. For each instantiation an identifier is defined, such as G1, G2, and G3 in this case. Then the component name is specified. The keyword **port map** essentially makes all the connections for the logic function using the operator =>. For example, the first instantiation,

```
G1: AND_gate port map (A => IN1, B => IN2, X => OUT1);
```

can be explained as follows: *Input A of AND gate G1 is connected to input IN1, input B of the gate is connected to input IN2, and the output X of the gate is connected to the signal OUT1.*

The three instantiation statements together completely describe the logic circuit in Figure 5–39, as illustrated in Figure 5–40.

Although the data flow approach using Boolean expressions would have been easier and probably the best way to describe this particular circuit, we have used this simple circuit to explain the concept of the structural approach. Example 5–16 compares the structural and data flow approaches to writing a VHDL program for an SOP logic circuit.

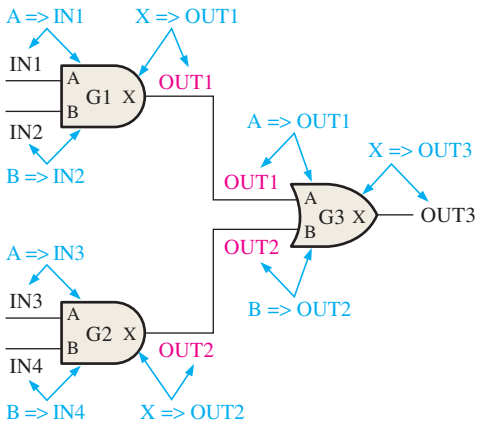


FIGURE 5–40 Illustration of the instantiation statements and port mapping applied to the AND-OR logic. Signals are shown in red.

EXAMPLE 5–16

Write a VHDL program for the SOP logic circuit in Figure 5–41 using the structural approach and compare with the data flow approach. Assume that VHDL components for a 3-input NAND gate and for a 2-input NAND are available. Notice the NAND gate G4 is shown as a negative-OR.

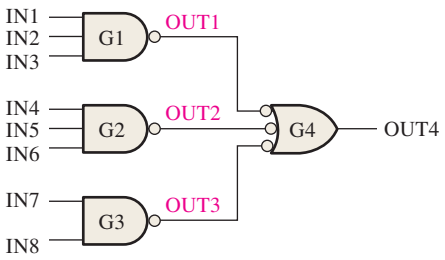


FIGURE 5–41

Solution

The structural approach:

The components and component instantiations are highlighted. Lines preceded by two hyphens are comment lines and are not part of the program.

```
--Program for the logic circuit in Figure 5–41
entity SOP_Logic is
   port (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8: in bit; OUT4: out bit);
end entity SOP_Logic;
architecture LogicOperation of SOP_Logic is
--component declaration for 3-input NAND gate
   component NAND_gate3 is
      port (A, B, C: in bit X: out bit);
   end component NAND_gate3;
--component declaration for 2-input NAND gate
   component NAND_gate2 is
      port (A, B: in bit; X: out bit);
   end component NAND_gate2;
   signal OUT1, OUT2, OUT3: bit;
```

```
begin
G1: NAND_gate3 port map   (A => IN1, B => IN2, C => IN3, X => OUT1);
G2: NAND_gate3 port map   (A => IN4, B => IN5, C => IN6, X => OUT2);
G3: NAND_gate2 port map   (A => IN7, B => IN8, X => OUT3);
G4: NAND_gate3 port map   (A => OUT1, B => OUT2, C => OUT3, X => OUT4);
end architecture LogicOperation;
```

The data flow approach:

The program for the logic circuit in Figure 5–41 using the data flow approach is written as follows:

```
entity SOP_Logic is
  port (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8: in bit; OUT4: out bit);
end entity SOP_Logic;

architecture LogicOperation of SOP_Logic is
begin
  OUT4 <= (IN1 and IN2 and IN3) or (IN4 and IN5 and IN6) or (IN7 and IN8);
end architecture LogicOperation;
```

As you can see, the data flow approach results in a much simpler code for this particular logic function. However, in situations where a logic function consists of many blocks of complex logic, the structural approach might have an advantage over the data flow approach.

Related Problem

If another NAND gate is added to the circuit in Figure 5–41 with inputs IN9 and IN10, write a component instantiation to add to the program.

Applying Software Development Tools

A software development package must be used to implement an HDL design in a target device. Once the logic has been described using an HDL and entered via a software tool called a code or text editor, it can be tested using a simulation to verify that it performs properly before actually programming the target device. Using software development tools allows for the design, development, and testing of combinational logic before it is committed to hardware.

Typical software development tools allow you to input VHDL code on a text-based editor specific to the particular development tool that you are using. The VHDL code for a combinational logic circuit has been written using a text-based editor for illustration and appears on the computer screen as shown in Figure 5–42. Many code editors provide enhanced features such as the highlighting of keywords.

After the program has been written into the text editor, it is passed to the compiler. The compiler takes the high-level VHDL code and converts it into a file that can be downloaded to the target device. Once the program has been compiled, you can create a simulation for testing. Simulated input values are inserted into the logic design and allow for verification of the output(s).

You specify the input waveforms on a software tool called a waveform editor, as shown in Figure 5–43. The output waveforms are generated by a simulation of the VHDL code that you entered on the text editor in Figure 5–42. The waveform simulation provides the resulting outputs X and Y for the inputs A, B, C, and D in all sixteen combinations from $0\,0\,0\,0_2$ to $1\,1\,1\,1_2$.

Recall from Chapter 3 that there are several performance characteristics of logic circuits to be considered in the creation of any digital system. Propagation delay, for example, determines the speed or frequency at which a logic circuit can operate. A timing simulation can be used to mimic the propagation delay through the logic design in the target device.

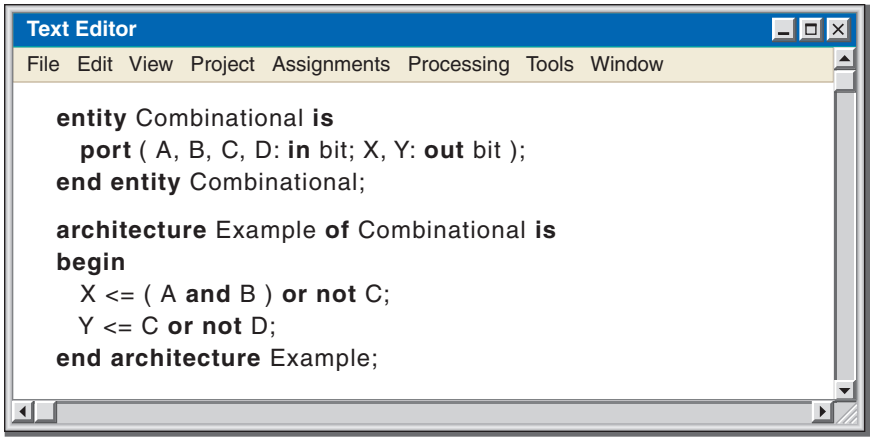


FIGURE 5–42 A VHDL program for a combinational logic circuit after entry on a generic text editor screen that is part of a software development tool.

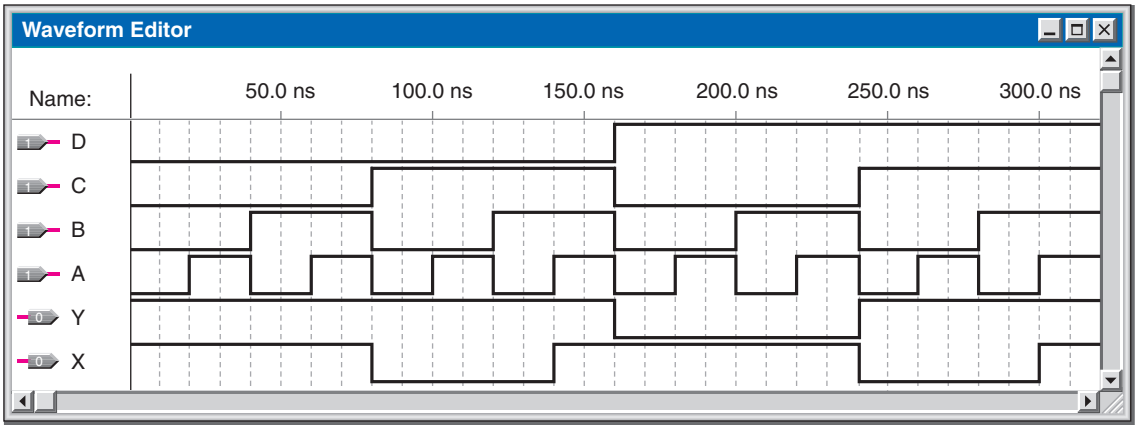


FIGURE 5–43 A typical waveform editor tool showing the simulated waveforms for the logic circuit described by the VHDL code in Figure 5–42.

SECTION 5–6 CHECKUP

1. What is a VHDL component?
2. State the purpose of a component instantiation in a program architecture.
3. How are interconnections made between components in VHDL?
4. The use of components in a VHDL program represents what approach?

5–7 Troubleshooting

The preceding sections have given you some insight into the operation of combinational logic circuits and the relationships of inputs and outputs. This type of understanding is essential when you troubleshoot digital circuits because you must know what logic levels or waveforms to look for throughout the circuit for a given set of input conditions.

In this section, an oscilloscope is used to troubleshoot a fixed-function logic circuit when a device output is connected to several device inputs. Also, an example of signal tracing and waveform analysis methods is presented using a scope or logic analyzer for locating a fault in a combinational logic circuit.

After completing this section, you should be able to

- Define a circuit node
- Use an oscilloscope to find a faulty circuit node
- Use an oscilloscope to find an open input or output
- Use an oscilloscope to find a shorted input or output
- Discuss how to use an oscilloscope or a logic analyzer for signal tracing in a combinational logic circuit

In a combinational logic circuit, the output of a driving device may be connected to two or more load devices as shown in Figure 5–44. The interconnecting paths share a common electrical point known as a **node**.

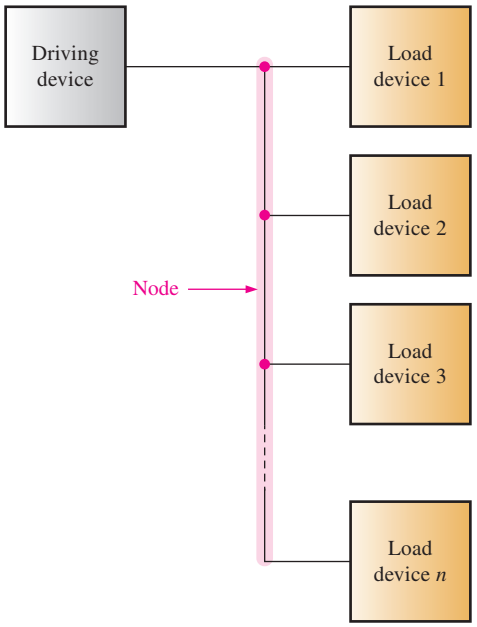


FIGURE 5–44 Illustration of a node in a logic circuit.

The driving device in Figure 5–44 is driving the node, and the other devices represent loads connected to the node. A driving device can drive a number of load device inputs up to its specified fan-out. Several types of failures are possible in this situation. Some of these failure modes are difficult to isolate to a single bad device because all the devices connected to the node are affected. Common types of failures are the following:

1. ***Open output in driving device.*** This failure will cause a loss of signal to all load devices.
2. ***Open input in a load device.*** This failure will not affect the operation of any of the other devices connected to the node, but it will result in loss of signal output from the faulty device.
3. ***Shorted output in driving device.*** This failure can cause the node to be stuck in the LOW state (short to ground) or in the HIGH state (short to V_{CC}).
4. ***Shorted input in a load device.*** This failure can also cause the node to be stuck in the LOW state (short to ground) or in the HIGH state (short to V_{CC}).

Troubleshooting Common Faults

Open Output in Driving Device

In this situation there is no pulse activity on the node. With circuit power on, an open node will normally result in a "floating" level, as illustrated in Figure 5–45.

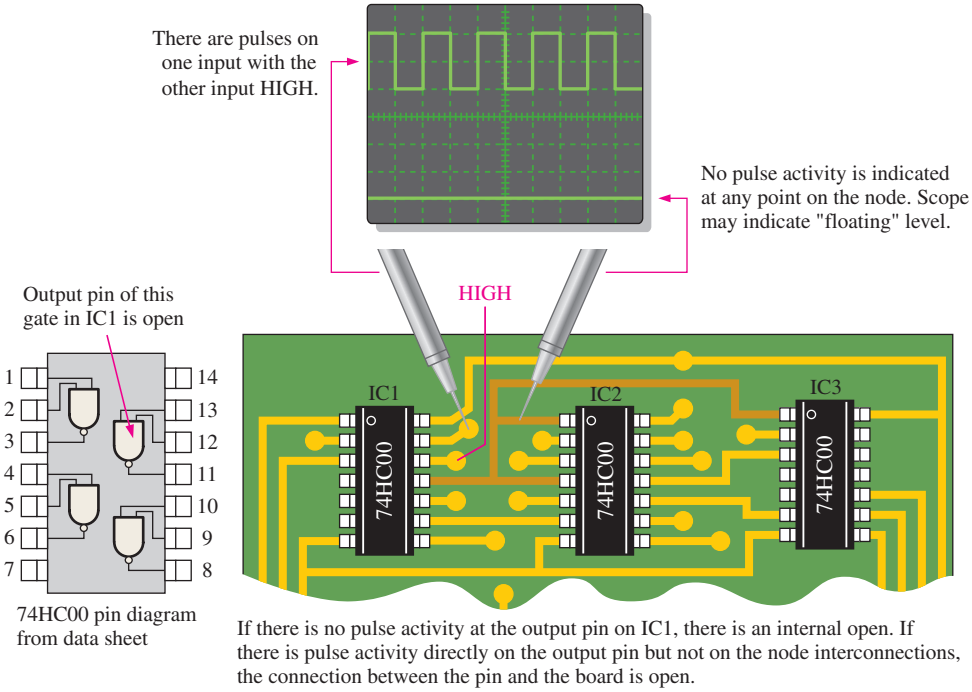


FIGURE 5–45 Open output in driving device. Assume a HIGH is on one input.

Open Input in a Load Device

If the check for an open driver output in IC1 is negative (there is pulse activity), then a check for an open input in a load device should be performed. Check the output of each device for pulse activity, as illustrated in Figure 5–46. If one of the inputs that is normally connected to the node is open, no pulses will be detected on that device's output.

Output or Input Shorted to Ground

When the output is shorted to ground in the driving device or the input to a load device is shorted to ground, it will cause the node to be stuck LOW, as previously mentioned. A quick check with a scope probe will indicate this, as shown in Figure 5–47. A short to ground in the driving device's output or in any load input will cause this symptom, and further checks must therefore be made to isolate the short to a particular device.

Signal Tracing and Waveform Analysis

Although the methods of isolating an open or a short at a node point are useful from time to time, a more general troubleshooting technique called **signal tracing** is of value in just

When troubleshooting logic circuits, begin with a visual check, looking for obvious problems. In addition to components, visual inspection should include connectors. Edge connectors are frequently used to bring power, ground, and signals to a circuit board. The mating surfaces of the connector need to be clean and have a good mechanical fit. A dirty connector can cause intermittent or complete failure of the circuit. Edge connectors can be cleaned with a common pencil eraser and wiped clean with a Q-tip soaked in alcohol. Also, all connectors should be checked for loose-fitting pins.

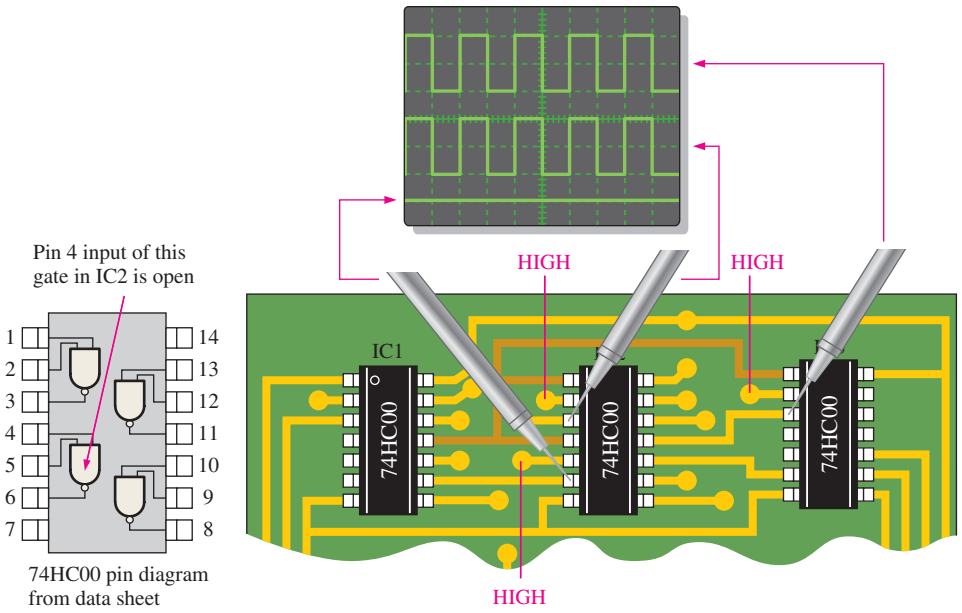


FIGURE 5–46 Open input in a load device.

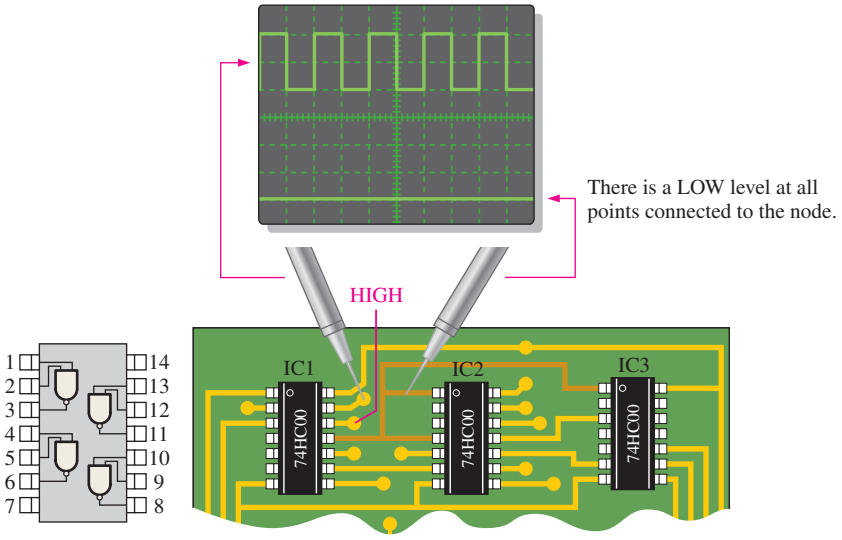


FIGURE 5–47 Shorted output in the driving device or shorted input in a load.

about every troubleshooting situation. Waveform measurement is accomplished with an oscilloscope or a logic analyzer.

Basically, the signal tracing method requires that you observe the waveforms and their time relationships at all accessible points in the logic circuit. You can begin at the inputs and, from an analysis of the waveform timing diagram for each point, determine where an incorrect waveform first occurs. With this procedure you can usually isolate the fault to a specific device. A procedure beginning at the output and working back toward the inputs can also be used.

The general procedure for signal tracing starting at the inputs is outlined as follows:

- Within a system, define the section of logic that is suspected of being faulty.
- Start at the inputs to the section of logic under examination. We assume, for this discussion, that the input waveforms coming from other sections of the system have been found to be correct.

- For each device, beginning at the input and working toward the output of the logic circuit, observe the output waveform of the device and compare it with the input waveforms by using the oscilloscope or the logic analyzer.
- Determine if the output waveform is correct, using your knowledge of the logical operation of the device.
- If the output is incorrect, the device under test may be faulty. Pull the IC device that is suspected of being faulty, and test it out-of-circuit. If the device is found to be faulty, replace the IC. If it works correctly, the fault is in the external circuitry or in another IC to which the tested one is connected.
- If the output is correct, go to the next device. Continue checking each device until an incorrect waveform is observed.

Figure 5–48 is an example that illustrates the general procedure for a specific logic circuit in the following steps:

Step 1: Observe the output of gate G_1 (test point 5) relative to the inputs. If it is correct, check the inverter next. If the output is not correct, the gate or its

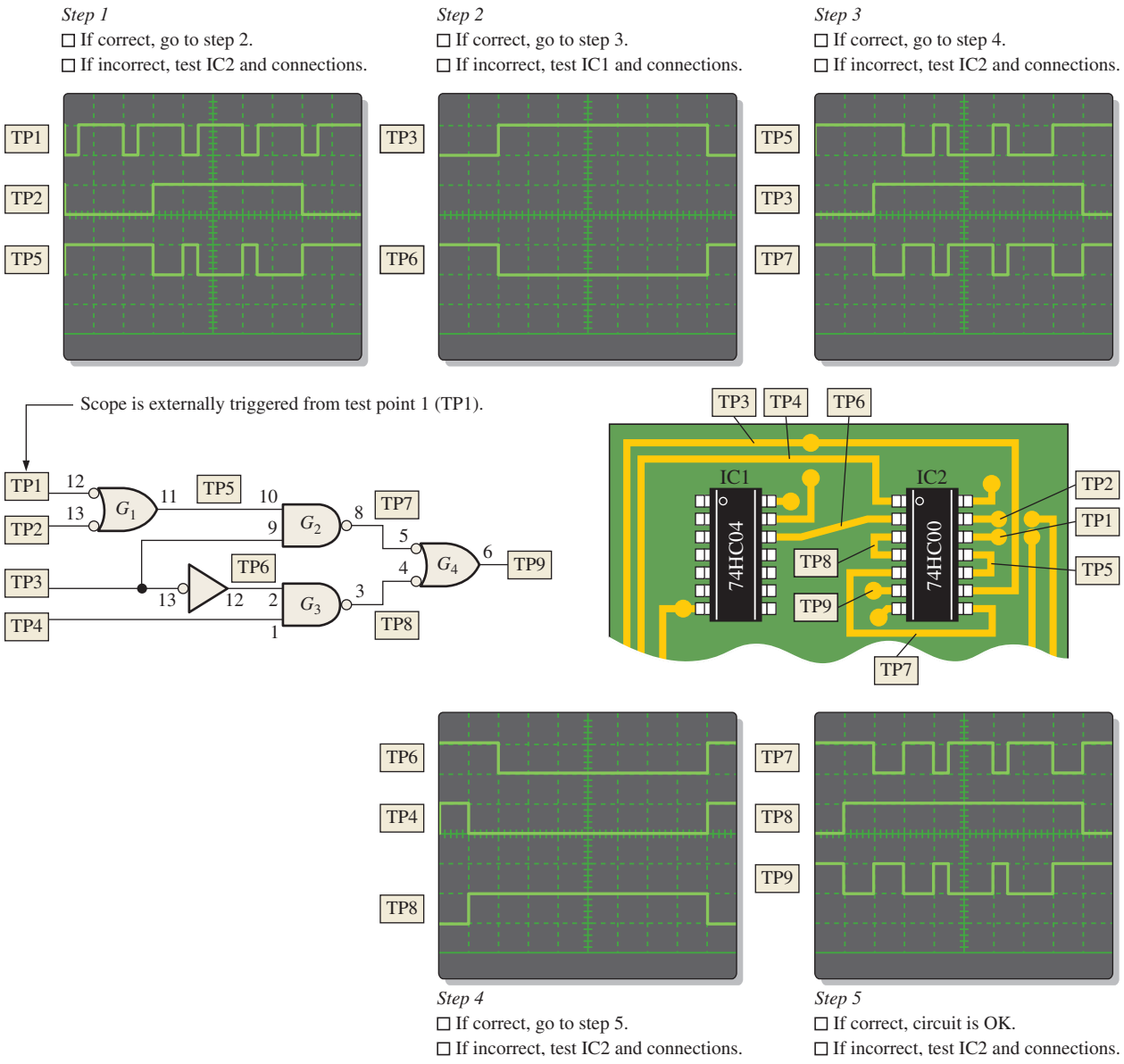


FIGURE 5–48 Example of signal tracing and waveform analysis in a portion of a printed circuit board. TP indicates test point.

connections are bad; or, if the output is LOW, the input to gate G_2 may be shorted.

Step 2: Observe the output of the inverter (TP6) relative to the input. If it is correct, check gate G_2 next. If the output is not correct, the inverter or its connections are bad; or, if the output is LOW, the input to gate G_3 may be shorted.

Step 3: Observe the output of gate G_2 (TP7) relative to the inputs. If it is correct, check gate G_3 next. If the output is not correct, the gate or its connections are bad; or, if the output is LOW, the input to gate G_4 may be shorted.

Step 4: Observe the output of gate G_3 (TP8) relative to the inputs. If it is correct, check gate G_4 next. If the output is not correct, the gate or its connections are bad; or, if the output is LOW, the input to gate G_4 (TP7) may be shorted.

Step 5: Observe the output of gate G_4 (TP9) relative to the inputs. If it is correct, the circuit is okay. If the output is not correct, the gate or its connections are bad.

EXAMPLE 5–17

Determine the fault in the logic circuit of Figure 5–49(a) by using waveform analysis. You have observed the waveforms shown in green in Figure 5–49(b). The red waveforms are correct and are provided for comparison.

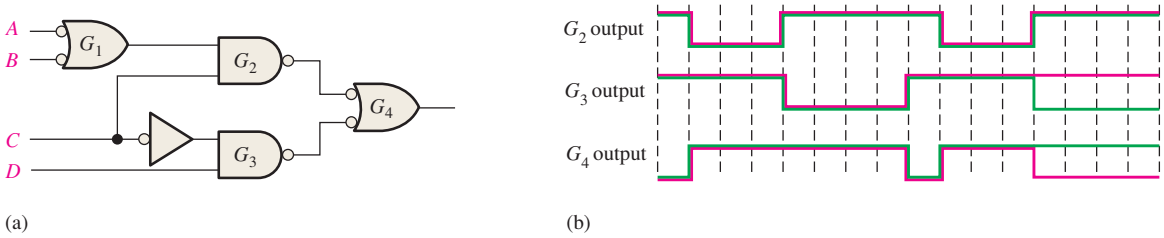


FIGURE 5–49

Solution

1. Determine what the correct waveform should be for each gate. The correct waveforms are shown in red, superimposed on the actual measured waveforms, in Figure 5–49(b).
2. Compare waveforms gate by gate until you find a measured waveform that does not match the correct waveform.

In this example, everything tested is correct until gate G_3 is checked. The output of this gate is not correct as the differences in the waveforms indicate. An analysis of the waveforms indicates that if the *D* input to gate G_3 is open and acting as a HIGH, you will get the output waveform measured (shown in red). Notice that the output of G_4 is also incorrect due to the incorrect input from G_3.

Replace the IC containing G_3, and check the circuit's operation again.

Related Problem

For the inputs in Figure 5–49(b), determine the output waveform for the logic circuit (output of G_4) if the inverter has an open output.

As you know, testing and troubleshooting logic circuits often require observing and comparing two digital waveforms simultaneously, such as an input and the output of a device, on an oscilloscope. For digital waveforms, the scope should always be set to DC coupling on each channel input to avoid "shifting" the ground level. You should determine where the 0 V level is on the screen for both channels.

To compare the timing of the waveforms, the scope should be triggered from only one channel (don't use vertical mode or composite triggering). The channel selected for triggering should always be the one that has the lowest frequency waveform, if possible.

SECTION 5–7 CHECKUP

1. List four common internal failures in logic gates.
2. One input of a NOR gate is externally shorted to $+V_{CC}$. How does this condition affect the gate operation?
3. Determine the output of gate G_4 in Figure 5–49(a), with inputs as shown in part (b), for the following faults:
   - **(a)** one input to G_1 shorted to ground
   - **(b)** the inverter input shorted to ground
   - **(c)** an open output in G_3

Applied Logic

Tank Control

A storage tank system for a pancake syrup manufacturing company is shown in Figure 5–50. The control logic allows a volume of corn syrup to be preheated to a specified temperature to achieve the proper viscosity prior to being sent to a mixing vat where ingredients such as sugar, flavoring, preservative, and coloring are added. Level and temperature sensors in the tank and the flow sensor provide the inputs for the logic.

System Operation and Analysis

The tank holds corn syrup for use in a pancake syrup manufacturing process. In preparation for mixing, the temperature of the corn syrup when released from the tank into a mixing vat must be at a specified value for proper viscosity to produce required flow characteristics. This temperature can be selected via a keypad input. The control logic maintains the temperature at this value by turning a heater *on* and *off*. The analog output from the temperature transducer (T_{analog}) is converted to an 8-bit binary code by an analog-to-digital converter and then to an 8-bit BCD code. A temperature controller detects when the temperature falls below the specified value and turns the heater *on*. When the temperature reaches the specified value, the heater is turned *off*.

The level sensors produce a HIGH when the corn syrup is at or above the minimum or at the maximum level. The valve control logic detects when the maximum level (L_{max}) or minimum level (L_{min}) has been reached and when mixture is flowing into the tank (F_{inlet}). Based on these inputs, the control logic opens or closes each valve (V_{inlet} and V_{outlet}). New corn syrup can be

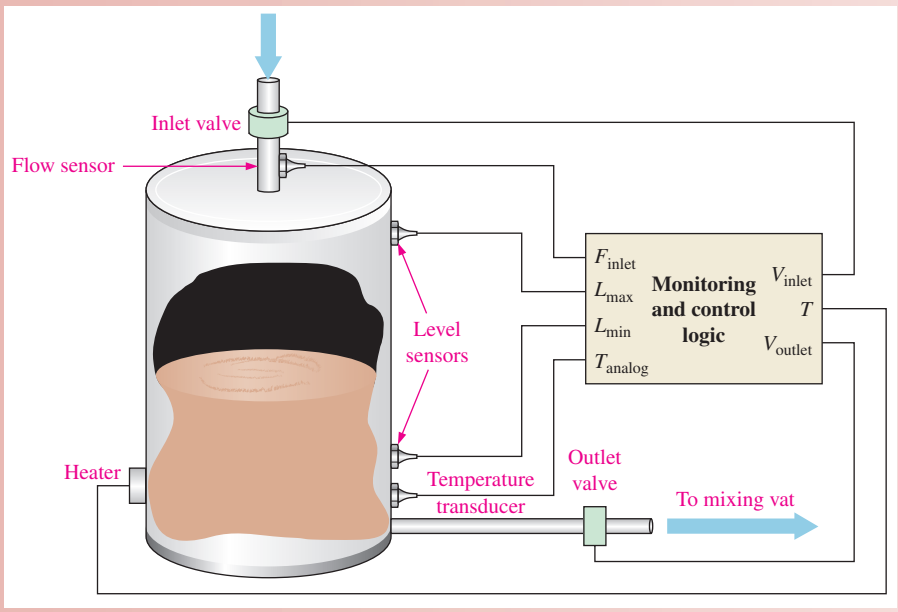


FIGURE 5–50 Tank with level and temperature sensors and controls.

added to the tank via the inlet valve only when the minimum level is reached. Once the inlet valve is opened, the level in the tank must reach the maximum point before the inlet valve is closed. Also, once the outlet valve is opened, the level must reach the minimum point before the outlet valve is closed. New syrup is always cooler than the syrup in the tank. Syrup cannot be released from the tank while it is being filled or its temperature is below the specified value.

Inlet Valve Control The conditions for which the inlet valve is open, allowing the tank to fill, are

- The solution level is at minimum (L_{min}).
- The tank is filling (F_{inlet}) but the maximum level has not been reached ($\overline{L}_{max}$).

Table 5–6 is the truth table for the inlet valve. A HIGH (1) is the active level for the inlet valve to be open (*on*).

TABLE 5–6

Truth table for inlet valve control.

Inputs			Output	
L_{max}	L_{min}	F_{inlet}	V_{inlet}	Description
0	0	0	1	Level below minimum. No inlet flow.
0	0	1	1	Level below minimum. Inlet flow.
0	1	0	0	Level above min and below max. No inlet flow.
0	1	1	1	Level above min and below max. Inlet flow.
1	0	0	X	Invalid
1	0	1	X	Invalid
1	1	0	0	Level at maximum. No inlet flow.
1	1	1	0	Level at maximum. Inlet flow.

Exercise

1. Explain why the two conditions indicated in the truth table are invalid.
2. Under how many input conditions is the inlet valve open?
3. Once the level drops below minimum and the tank starts refilling, when does the inlet valve turn *off*?

From the truth table, an expression for the inlet valve control output can be written.

$$V_{\text{inlet}} = \overline{L}_{\text{max}}\overline{L}_{\text{min}}\overline{F}_{\text{inlet}} + \overline{L}_{\text{max}}\overline{L}_{\text{min}}F_{\text{inlet}} + \overline{L}_{\text{max}}L_{\text{min}}F_{\text{inlet}}$$

The SOP expression for the inlet valve logic can be reduced to the following simplified expression using Boolean methods:

$$V_{\text{inlet}} = \overline{L}_{\text{min}} + \overline{L}_{\text{max}}F_{\text{inlet}}$$

Exercise

4. Using a K-map, prove that the simplified expression is correct.
5. Using the simplified expression, draw the logic diagram for the inlet valve control.

Outlet Valve Control The conditions for which the outlet valve is open allowing the tank to drain are

- The syrup level is above minimum and the tank is not filling.
- The temperature of the syrup is at the specified value.

Table 5–7 is the truth table for the outlet valve. A HIGH (1) is the active level for the outlet valve to be open (*on*). (*Note: T* is both an input and an output, T = Temp).

TABLE 5–7

Truth table for outlet valve control.

Inputs				Output	
L_{max}	L_{min}	F_{inlet}	T	V_{outlet}	Description
0	0	0	0	0	Level below minimum. No inlet flow. Temp low.
0	0	0	1	0	Level below minimum. No inlet flow. Temp correct.
0	0	1	0	0	Level below minimum. Inlet flow. Temp low.
0	0	1	1	0	Level below minimum. Inlet flow. Temp correct.
0	1	0	0	0	Level above min and below max. No inlet flow. Temp low.
0	1	0	1	1	Level above min and below max. No inlet flow. Temp correct.
0	1	1	0	0	Level above min and below max. Inlet flow. Temp low.
0	1	1	1	0	Level above min and below max. Inlet flow. Temp correct
1	0	0	0	X	Invalid
1	0	0	1	X	Invalid
1	0	1	0	X	Invalid
1	0	1	1	X	Invalid
1	1	0	0	0	Level at maximum. No inlet flow. Temp low.
1	1	0	1	1	Level at maximum. No inlet flow. Temp correct.
1	1	1	0	0	Level at maximum. Inlet flow. Temp low.
1	1	1	1	0	Level at maximum. Inlet flow. Temp correct.

Exercise

6. Why does the outlet valve control require four inputs and the inlet valve only three?
7. Under how many input conditions is the outlet valve open?
8. Once the level reaches maximum and the tank starts draining, when does the outlet valve turn off?

From the truth table, an expression for the outlet valve control can be written.

$$V_{\text{outlet}} = \overline{L}_{\text{max}}L_{\text{min}}\overline{F}_{\text{inlet}}\,T + L_{\text{max}}L_{\text{min}}\overline{F}_{\text{inlet}}T$$

The SOP expression for the outlet valve logic can be reduced to the following simplified expression:

$$V_{\text{outlet}} = L_{\text{min}}\overline{F}_{\text{inlet}}T$$

Exercise

9. Using a K-map, prove that the simplified expression is correct.

10. Using the simplified expression, draw the logic diagram for the outlet valve control.

Temperature Control The temperature control logic accepts an 8-bit BCD code representing the measured temperature and compares it to the BCD code for the specified temperature. A block diagram is shown in Figure 5–51.

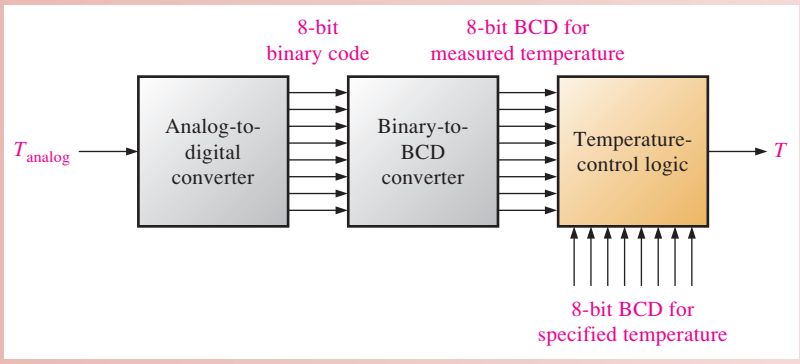


FIGURE 5–51 Block diagram for temperature control circuit.

When the measured temperature and the specified temperature are the same, the two BCD codes are equal and the T output is LOW (0). When the measured temperature falls below the specified value, there is a difference in the BCD codes and the T output is HIGH (1), which turns on the heater. The temperature control logic can be implemented with exclusive-OR gates, as shown in Figure 5–52. Each pair of corresponding bits from the two

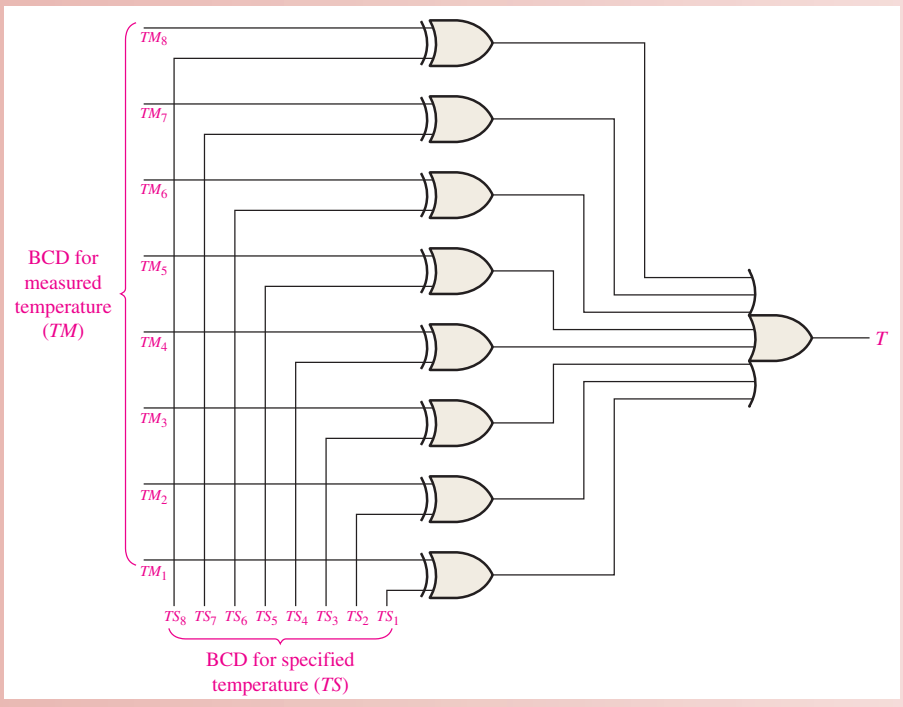


FIGURE 5–52 Logic diagram of the temperature control logic.

BCD codes is applied to an exclusive-OR gate. If the bits are the same, the output of the XOR gate is 0; and if they are different, the output of the XOR gate is 1. When one or more XOR outputs equal 1, the *T* output of the OR gate equals 1, causing the heater to turn on.

VHDL Code for Tank Control Logic

The control logic for the inlet valve, outlet valve, and temperature is described with VHDL using the data flow approach (which is based on the Boolean description of the logic). Exercise 11 requires the structural approach (which is based on the gates and how they are connected) for comparison.

```
entity TankControl is
    port (Finlet, Lmax, Lmin, TS1, TS2, TS3, TS4, TS5, TS6, TS7, TS8, TM1, TM2,
    TM3, TM4, TM5, TM6, TM7, TM8: in bit; Vinlet, Voutlet, T: out bit);
end entity TankControl;

architecture ValveTempLogic of Tank Control is

begin

    Vinlet <= not Lmin or (not Lmax and Finlet);
    Voutlet <= Lmin and not Finlet and T;
    T <= (TS1 xor TM1) or (TS2 xor TM2) or (TS3 xor TM3) or (TS4 xor TM4)
    or (TS5 xor TM5) or (TS6 xor TM6) or (TS7 xor TM7) or (TS8 xor TM8);

end architecture ValveTempLogic;
```

Exercise

11. Write the VHDL code for the tank control logic using the structural approach.

Simulation of the Valve Control Logic

The inlet and outlet valve control logic simulation screen is shown in Figure 5–53. SPDT switches are used to represent the level and flow sensor inputs and the temperature indication. Probes are used to indicate the output states.

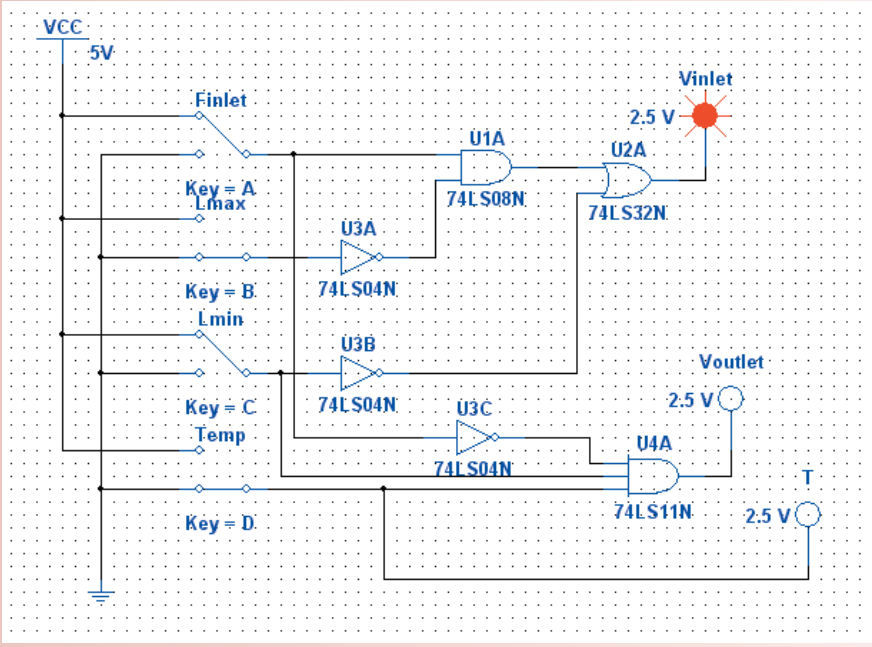


FIGURE 5–53 Multisim circuit screen for the valve control logic.

Open file AL05 in the Applied Logic folder on the website. Run the simulation of the valve-control logic using your Multisim software and observe the operation. Create a new Multisim file, connect the temperature control logic, and run the simulation.

MultiSim

Putting Your Knowledge to Work

If the temperature of the syrup can never be more than 9°C below the specified value, can the temperature control circuit be simplified? If so, how?

SUMMARY

- AND-OR logic produces an output expression in SOP form.
- AND-OR-Invert logic produces a complemented SOP form, which is actually a POS form.
- The operational symbol for exclusive-OR is $\oplus$. An exclusive-OR expression can be stated in two equivalent ways:

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

- To do an analysis of a logic circuit, start with the logic circuit, and develop the Boolean output expression or the truth table or both.
- Implementation of a logic circuit is the process in which you start with the Boolean output expressions or the truth table and develop a logic circuit that produces the output function.
- All NAND or NOR logic diagrams should be drawn using appropriate dual symbols so that bubble outputs are connected to bubble inputs and nonbubble outputs are connected to nonbubble inputs.
- When two negation indicators (bubbles) are connected, they effectively cancel each other.
- A VHDL component is a predefined logic function stored for use throughout a program or in other programs.
- A component instantiation is used to call for a component in a program.
- A VHDL signal effectively acts as an internal interconnection in a VHDL structural description.

KEY TERMS

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

Component A VHDL feature that can be used to predefine a logic function for multiple use throughout a program or programs.

Negative-AND The dual operation of a NOR gate when the inputs are active-LOW.

Negative-OR The dual operation of a NAND gate when the inputs are active-LOW.

Node A common connection point in a circuit in which a gate output is connected to one or more gate inputs.

Signal A waveform; a type of VHDL object that holds data.

Signal tracing A troubleshooting technique in which waveforms are observed in a step-by-step manner beginning at the input and working toward the output or vice versa. At each point the observed waveform is compared with the correct signal for that point.

Universal gate Either a NAND gate or a NOR gate. The term *universal* refers to the property of a gate that permits any logic function to be implemented by that gate or by a combination of that kind.