## CHAPTER 3

# DESCRIBING LOGIC CIRCUITS

## OUTLINE

- 3-1 Boolean Constants and Variables
- 3-2 Truth Tables
- 3-3 OR Operation with OR Gates
- 3-4 AND Operation with AND Gates
- 3-5 NOT Operation
- 3-6 Describing Logic Circuits Algebraically
- 3-7 Evaluating Logic-Circuit Outputs
- 3-8 Implementing Circuits from Boolean Expressions
- 3-9 NOR Gates and NAND Gates
- **3-10** Boolean Theorems
- 3-11 DeMorgan's Theorems

- 3-12 Universality of NAND Gates and NOR Gates
- 3-13 Alternate Logic-Gate Representations
- 3-14 Which Gate Representation to Use
- 3-15 IEEE/ANSI Standard Logic Symbols
- 3-16 Summary of Methods to Describe Logic Circuits
- 3-17 Description Languages Versus Programming Languages
- 3-18 Implementing Logic Circuits with PLDs
- 3-19 HDL Format and Syntax
- 3-20 Intermediate Signals

## **OBJECTIVES**

Upon completion of this chapter, you will be able to:

- Perform the three basic logic operations.
- Describe the operation of and construct the truth tables for the AND, NAND, OR, and NOR gates, and the NOT (INVERTER) circuit.
- Draw timing diagrams for the various logic-circuit gates.
- Write the Boolean expression for the logic gates and combinations of logic gates.
- Implement logic circuits using basic AND, OR, and NOT gates.
- Appreciate the potential of Boolean algebra to simplify complex logic circuits.
- Use DeMorgan's theorems to simplify logic expressions.
- Use either of the universal gates (NAND or NOR) to implement a circuit represented by a Boolean expression.
- Explain the advantages of constructing a logic-circuit diagram using the alternate gate symbols versus the standard logic-gate symbols.
- Describe the concept of active-LOW and active-HIGH logic signals.
- Draw and interpret the IEEE/ANSI standard logic-gate symbols.
- Use several methods to describe the operation of logic circuits.
- Interpret simple circuits defined by a hardware description language (HDL).
- Explain the difference between an HDL and a computer programming language.
- Create an HDL file for a simple logic gate.
- Create an HDL file for combinational circuits with intermediate variables.

## INTRODUCTION

Chapters 1 and 2 introduced the concepts of logic levels and logic circuits. In logic, only two possible conditions exist for any input or output: true and false. The binary number system uses only two digits, 1 and 0, so it is perfect for representing logical relationships. Digital logic circuits use predefined voltage ranges to represent these binary states. Using these concepts, we can create circuits made of little more than processed beach sand and wire that make consistent, intelligent, logical decisions. It is vitally important that we have a method to describe the logical decisions made by these circuits. In other words, we must describe how they operate. In this chapter, we will discover many ways to describe their operation. Each description method is important because all these methods commonly appear in technical literature and system documentation and are used in conjunction with modern design and development tools.

Life is full of examples of circumstances that are in one state or another. For example, a creature is either alive or dead, a light is either on or off, a door is locked or unlocked, and it is either raining or it is not. In 1854, a mathematician named George Boole wrote *An Investigation of the Laws of Thought*, in which he described the way we make logical decisions based on true or false circumstances. The methods he described are referred to today as Boolean logic, and the system of using symbols and operators to describe these decisions is called Boolean algebra. In the same way we use symbols such as x and y to represent unknown numerical values in regular algebra, Boolean algebra uses symbols to represent a logical expression that has one of two possible values: true or false. The logical expression might be *door is closed, button is pressed,* or *fuel is low.* Writing these expressions is very tedious, and so we tend to substitute symbols such as *A*, *B*, and *C*.

The main purpose of these logical expressions is to describe the relationship between a logic circuit's output (the decision) and its inputs (the circumstances). In this chapter, we will study the most basic logic circuits *logic gates*—which are the fundamental building blocks from which all other logic circuits and digital systems are constructed. We will see how the operation of the different logic gates and the more complex circuits formed from combinations of logic gates can be described and analyzed using Boolean algebra. We will also get a glimpse of how Boolean algebra can be used to simplify a circuit's Boolean expression so that the circuit can be rebuilt using fewer logic gates and/or fewer connections. Much more will be done with circuit simplification in Chapter 4.

Boolean algebra is not only used as a tool for analysis and simplification of logic systems. It can also be used as a tool to create a logic circuit that will produce the desired input/output relationship. This process is often called synthesis of logic circuits as opposed to analysis. Other techniques have been used in the analysis, synthesis, and documentation of logic systems and circuits including truth tables, schematic symbols, timing diagrams, and—last but by no means least—language. To categorize these methods, we could say that Boolean algebra is a mathematic tool, truth tables are data organizational tools, schematic symbols are drawing tools, timing diagrams are graphing tools, and language is the universal description tool.

Today, any of these tools can be used to provide input to computers. The computers can be used to simplify and translate between these various forms of description and ultimately provide an output in the form necessary to implement a digital system. To take advantage of the powerful benefits of computer software, we must first fully understand the acceptable ways for describing these systems in terms the computer can understand. This chapter will lay the groundwork for further study of these vital tools for synthesis and analysis of digital systems.

Clearly the tools described here are invaluable tools in describing, analyzing, designing, and implementing digital circuits. The student who expects to work in the digital field must work hard at understanding and becoming comfortable with Boolean algebra (believe us, it's much, much easier than conventional algebra) and all the other tools. Do *all* of the examples, exercises, and problems, even the ones your instructor doesn't assign. When those run out, make up your own. The time you spend will be well worth it because you will see your skills improve and your confidence grow. **TABLE 3-1** 

## **3-1 BOOLEAN CONSTANTS AND VARIABLES**

Boolean algebra differs in a major way from ordinary algebra because Boolean constants and variables are allowed to have only two possible values, 0 or 1. A Boolean variable is a quantity that may, at different times, be equal to either 0 or 1. Boolean variables are often used to represent the voltage level present on a wire or at the input/output terminals of a circuit. For example, in a certain digital system, the Boolean value of 0 might be assigned to any voltage in the range from 0 to 0.8 V, while the Boolean value of 1 might be assigned to any voltage in the range 2 to 5 V.\*

Thus, Boolean 0 and 1 do not represent actual numbers but instead represent the state of a voltage variable, or what is called its **logic level**. A voltage in a digital circuit is said to be at the logic 0 level or the logic 1 level, depending on its actual numerical value. In digital logic, several other terms are used synonymously with 0 and 1. Some of the more common ones are shown in Table 3-1. We will use the 0/1 and LOW/HIGH designations most of the time.

Logic (	0 Logic 1
False	True
Off	On
Low	High
No	Yes
Open swi	itch Closed switch

As we said in the introduction, **Boolean algebra** is a means for expressing the relationship between a logic circuit's inputs and outputs. The inputs are considered logic variables whose logic levels at any time determine the output levels. In all our work to follow, we shall use letter symbols to represent logic variables. For example, the letter A might represent a certain digital circuit input or output, and at any time we must have either A = 0 or A = 1: if not one, then the other.

Because only two values are possible, Boolean algebra is relatively easy to work with compared with ordinary algebra. In Boolean algebra, there are no fractions, decimals, negative numbers, square roots, cube roots, logarithms, imaginary numbers, and so on. In fact, in Boolean algebra there are only *three* basic operations: *OR*, *AND*, and *NOT*.

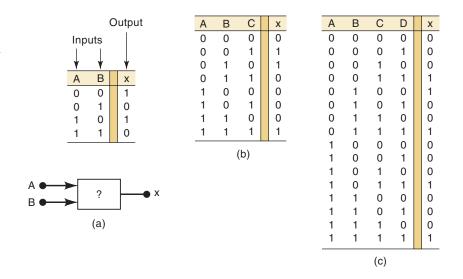
These basic operations are called *logic operations*. Digital circuits called *logic gates* can be constructed from diodes, transistors, and resistors connected so that the circuit output is the result of a basic logic operation (*OR*, *AND*, *NOT*) performed on the inputs. We will be using Boolean algebra first to describe and analyze these basic logic gates, then later to analyze and design combinations of logic gates connected as logic circuits.

## **3-2 TRUTH TABLES**

A **truth table** is a means for describing how a logic circuit's output depends on the logic levels present at the circuit's inputs. Figure 3-1(a) illustrates a truth table for one type of two-input logic circuit. The table lists all possible

-	_
h	
J	

<sup>\*</sup>Voltages between 0.8 and 2 V are undefined (neither 0 nor 1) and should not occur under normal circumstances.



combinations of logic levels present at inputs A and B, along with the corresponding output level x. The first entry in the table shows that when A and B are both at the 0 level, the output x is at the 1 level or, equivalently, in the 1 state. The second entry shows that when input B is changed to the 1 state, so that A = 0 and B = 1, the output x becomes a 0. In a similar way, the table shows what happens to the output state for any set of input conditions.

Figures 3-1(b) and (c) show samples of truth tables for three- and fourinput logic circuits. Again, each table lists all possible combinations of input logic levels on the left, with the resultant logic level for output x on the right. Of course, the actual values for x will depend on the type of logic circuit.

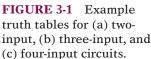
Note that there are 4 table entries for the two-input truth table, 8 entries for a three-input truth table, and 16 entries for the four-input truth table. The number of input combinations will equal  $2^N$  for an *N*-input truth table. Also note that the list of all possible input combinations follows the binary counting sequence, and so it is an easy matter to write down all of the combinations without missing any.

#### **REVIEW QUESTIONS**

- 1. What is the output state of the four-input circuit represented in Figure 3-1(c) when all inputs except *B* are 1?
- 2. Repeat question 1 for the following input conditions: A = 1, B = 0, C = 1, D = 0.
- 3. How many table entries are needed for a five-input circuit?

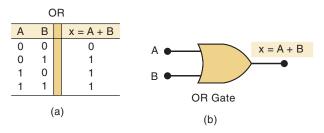
## **3-3 OR OPERATION WITH OR GATES**

The **OR operation** is the first of the three basic Boolean operations to be learned. An example can be found in the kitchen oven. The light inside the oven should turn on if either the oven light switch is on OR if the door is opened. The letter A could be used to represent the oven light switch is on and B could represent door is opened. The letter x could represent the light is on. The truth table in Figure 3-2(a) shows what happens when two logic inputs, A and B, are combined using the OR operation to produce the output x. The table shows that x is a logic 1 for every combination of input levels where one or more inputs are 1. The only case where x is a 0 is when both inputs are 0.



**FIGURE 3-2** (a) Truth table defining the OR operation; (b) circuit symbol for a two-input OR gate.





The Boolean expression for the OR operation is

$$x = A + B$$

In this expression, the + sign does not stand for ordinary addition; it stands for the OR operation. The OR operation is similar to ordinary addition except for the case where A and B are both 1; the OR operation produces 1 + 1 = 1, not 1 + 1 = 2. In Boolean algebra, 1 is as high as we go, so we can never have a result greater than 1. The same holds true for combining three inputs using the OR operation. Here we have x = A + B + C. If we consider the case where all three inputs are 1, we have

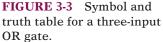
$$x = 1 + 1 + 1 = 1$$

The expression x = A + B is read as "x equals A OR B," which means that x will be 1 when A or B or both are 1. Likewise, the expression x = A + B + C is read as "x equals A OR B OR C," which means that x will be 1 when A or B or C or any combination of them are 1. To describe this circuit in the English language we could say that x is true (1) **WHEN** A is true (1) **OR** B is true (1) **OR** C is true (1).

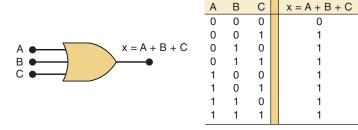
## **OR Gate**

In digital circuitry, an **OR** gate\* is a circuit that has two or more inputs and whose output is equal to the OR combination of the inputs. Figure 3-2(b) is the logic symbol for a two-input OR gate. The inputs *A* and *B* are logic voltage levels, and the output *x* is a logic voltage level whose value is the result of the OR operation on *A* and *B*; that is, x = A + B. In other words, the OR gate operates so that its output is HIGH (logic 1) if either input *A* or *B* or both are at a logic 1 level. The OR gate output will be LOW (logic 0) only if all its inputs are at logic 0.

This same idea can be extended to more than two inputs. Figure 3-3 shows a three-input OR gate and its truth table. Examination of this truth table shows again that the output will be 1 for every case where one or more inputs are 1. This general principle is the same for OR gates with any number of inputs.







\*The term gate comes from the inhibit/enable operation discussed in Chapter 4.

Using the language of Boolean algebra, the output x can be expressed as x = A + B + C, where again it must be emphasized that the + represents the OR operation. The output of any OR gate, then, can be expressed as the OR combination of its various inputs. We will put this to use when we subsequently analyze logic circuits.

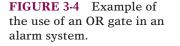
## Summary of the OR Operation

The important points to remember concerning the OR operation and OR gates are:

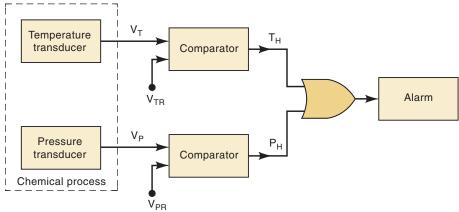
- 1. The OR operation produces a result (output) of 1 whenever *any* input is a 1. Otherwise the output is 0.
- 2. An OR gate is a logic circuit that performs an OR operation on the circuit's inputs.
- 3. The expression x = A + B is read as "x equals A OR B."

#### **EXAMPLE 3-1**

In many industrial control systems, it is required to activate an output function whenever any one of several inputs is activated. For example, in a chemical process it may be desired that an alarm be activated whenever the process temperature exceeds a maximum value *or* whenever the pressure goes above a certain limit. Figure 3-4 is a block diagram of this situation. The temperature transducer circuit produces an output voltage proportional to the process temperature. This voltage,  $V_{\rm T}$ , is compared with a temperature reference voltage,  $V_{\rm TR}$ , in a voltage comparator circuit. The comparator output,  $T_{\rm H}$ , is normally a low voltage (logic 0), but it switches to a high voltage (logic 1) when  $V_{\rm T}$  exceeds  $V_{\rm TR}$ , indicating that the process temperature is too high. A similar arrangement is used for the pressure measurement, so that its associated comparator output,  $P_{\rm H}$ , goes from LOW to HIGH when the pressure is too high.



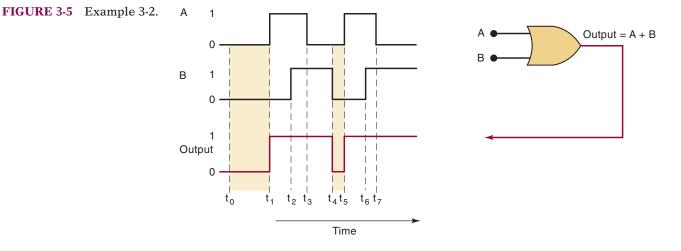




Since we want the alarm to be activated when either temperature *or* pressure is too high, it should be apparent that the two comparator outputs can be fed to a two-input OR gate. The OR gate output thus goes HIGH (1) for either alarm condition and will activate the alarm. This same idea can obviously be extended to situations with more than two process variables.

## EXAMPLE 3-2

Determine the OR gate output in Figure 3-5. The OR gate inputs A and B are varying according to the timing diagrams shown. For example, A starts out LOW at time  $t_0$ , goes HIGH at  $t_1$ , back to LOW at  $t_3$ , and so on.



#### Solution

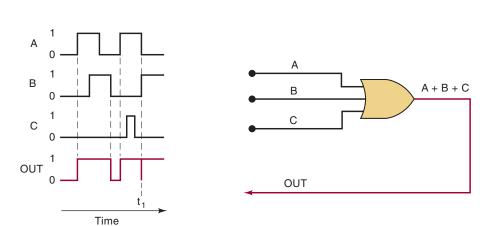
gate output.

The OR gate output will be HIGH whenever *any* input is HIGH. Between time  $t_0$  and  $t_1$ , both inputs are LOW, so OUTPUT = LOW. At  $t_1$ , input *A* goes HIGH while *B* remains LOW. This causes OUTPUT to go HIGH at  $t_1$  and stay HIGH until  $t_4$  because, during this interval, one or both inputs are HIGH. At  $t_4$ , input *B* goes from 1 to 0 so that now both inputs are LOW, and this drives OUTPUT back to LOW. At  $t_5$ , *A* goes HIGH, sending OUTPUT back HIGH, where it stays for the rest of the shown time span.

For the situation depicted in Figure 3-6, determine the waveform at the OR

#### **EXAMPLE 3-3A**

**FIGURE 3-6** Examples 3-3A and B.



#### Solution

The three OR gate inputs *A*, *B*, and *C* are varying, as shown by their waveform diagrams. The OR gate output is determined by realizing that it will be

HIGH whenever *any* of the three inputs is at a HIGH level. Using this reasoning, the OR output waveform is as shown in the figure. Particular attention should be paid to what occurs at time  $t_1$ . The diagram shows that, at that instant of time, input *A* is going from HIGH to LOW while input *B* is going from LOW to HIGH. Since these inputs are making their transitions at approximately the same time, and since these transitions take a certain amount of time, there is a short interval when these OR gate inputs are both in the undefined range between 0 and 1. When this occurs, the OR gate output also becomes a value in this range, as evidenced by the glitch or spike on the output waveform at  $t_1$ . The occurrence of this glitch and its size (amplitude and width) depend on the speed with which the input transitions occur.

#### **EXAMPLE 3-3B**

What would happen to the glitch in the output in Figure 3-6 if input *C* sat in the HIGH state while *A* and *B* were changing at time  $t_1$ ?

#### Solution

With the *C* input HIGH at  $t_1$ , the OR gate output will remain HIGH, regardless of what is occurring at the other inputs, because any HIGH input will keep an OR gate output HIGH. Therefore, the glitch will not appear in the output.

#### **REVIEW QUESTIONS**

- 1. What is the only set of input conditions that will produce a LOW output for any OR gate?
- 2. Write the Boolean expression for a six-input OR gate.
- 3. If the *A* input in Figure 3-6 is permanently kept at the 1 level, what will the resultant output waveform be?

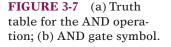
## 3-4 AND OPERATION WITH AND GATES

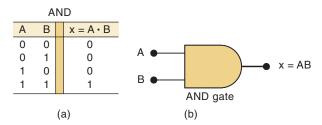
The **AND operation** is the second basic Boolean operation. As an example of the use of AND logic, consider a typical clothes dryer. It is drying clothes (heating, tumbling) only if the *timer is set above zero* AND the *door is closed*. Let's assign A to represent *timer is set*, B to represent *door is closed*, and x can represent the *heater and motor are on*. The truth table in Figure 3-7(a) shows what happens when two logic inputs, A and B, are combined using the AND operation to produce output x. The table shows that x is a logic 1 only when both A and B are at the logic 1 level. For any case where one of the inputs is 0, the output is 0.

The Boolean expression for the AND operation is

$$x = A \cdot B$$

In this expression, the  $\cdot$  sign stands for the Boolean AND operation and not the multiplication operation. However, the AND operation on Boolean variables operates the same as ordinary multiplication, as examination of the truth table shows, so we can think of them as being the same. This characteristic can be helpful when evaluating logic expressions that contain AND operations.





The expression  $x = A \cdot B$  is read as "x equals A AND B," which means that x will be 1 only when A and B are both 1. The  $\cdot$  sign is usually omitted so that the expression simply becomes x = AB. For the case when three inputs are ANDed, we have  $x = A \cdot B \cdot C = ABC$ . This is read as "x equals A AND B AND C," which means that x will be 1 only when A and B and C are all 1.

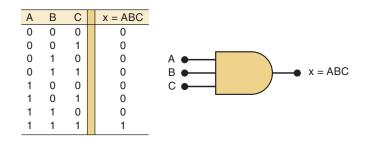
## AND Gate

The logic symbol for a two-input **AND** gate is shown in Figure 3-7(b). The AND gate output is equal to the AND product of the logic inputs; that is, x = AB. In other words, the AND gate is a circuit that operates so that its output is HIGH only when all its inputs are HIGH. For all other cases, the AND gate output is LOW.

This same operation is characteristic of AND gates with more than two inputs. For example, a three-input AND gate and its accompanying truth table are shown in Figure 3-8. Once again, note that the gate output is 1 only for the case where A = B = C = 1. The expression for the output is x = ABC. For a four-input AND gate, the output is x = ABCD, and so on.

**FIGURE 3-8** Truth table and symbol for a three-input AND gate.





Note the difference between the symbols for the AND gate and the OR gate. Whenever you see the AND symbol on a logic-circuit diagram, it tells you that the output will go HIGH *only* when *all* inputs are HIGH. Whenever you see the OR symbol, it means that the output will go HIGH when *any* input is HIGH.

#### Summary of the AND Operation

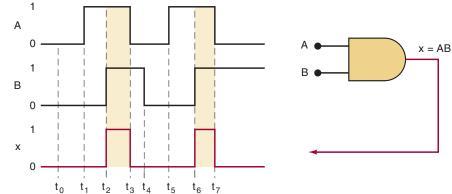
- 1. The AND operation is performed the same as ordinary multiplication of 1s and 0s.
- 2. An AND gate is a logic circuit that performs the AND operation on the circuit's inputs.
- 3. An AND gate output will be 1 *only* for the case when *all* inputs are 1; for all other cases, the output will be 0.
- 4. The expression x = AB is read as "x equals A AND B."

**EXAMPLE 3-4** 

Determine the output *x* from the AND gate in Figure 3-9 for the given input waveforms.

FIGURE 3-9 Example 3-4.





#### Solution

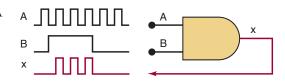
The output of an AND gate is determined by realizing that it will be HIGH only when all inputs are HIGH at the same time. For the input waveforms given, this condition is met only during intervals  $t_2-t_3$  and  $t_6-t_7$ . At all other times, one or more of the inputs are 0, thereby producing a LOW output. Note that input level changes that occur while the other input is LOW have no effect on the output.



Determine the output waveform for the AND gate shown in Figure 3-10.



FIGURE 3-10 Examples 3-5A and B.



#### **Solution**

The output *x* will be at 1 only when *A* and *B* are both HIGH at the same time. Using this fact, we can determine the *x* waveform as shown in the figure.

Notice that the *x* waveform is 0 whenever *B* is 0, regardless of the signal at *A*. Also notice that whenever *B* is 1, the *x* waveform is the same as *A*. Thus, we can think of the *B* input as a *control* input whose logic level determines whether or not the *A* waveform gets through to the *x* output. In this situation, the AND gate is used as an *inhibit circuit*. We can say that B = 0 is the inhibit condition producing a 0 output. Conversely, B = 1 is the *enable* condition, which enables *A* to reach the output. This inhibit operation is an important application of AND gates, which will be encountered later.

#### **EXAMPLE 3-5B**

What will happen to the *x* output waveform in Figure 3-10 if the *B* input is kept at the 0 level?

#### Solution

With *B* kept LOW, the *x* output will also stay LOW. This can be reasoned in two different ways. First, with B = 0 we have  $x = A \cdot B = A \cdot 0 = 0$  because

anything multiplied (ANDed) by 0 will be 0. Another way to look at it is that an AND gate requires that all inputs be HIGH for the output to be HIGH, and this cannot happen if *B* is kept LOW.

#### **REVIEW QUESTIONS**

- 1. What is the only input combination that will produce a HIGH at the output of a five-input AND gate?
- 2. What logic level should be applied to the second input of a two-input AND gate if the logic signal at the first input is to be inhibited (prevented) from reaching the output?
- 3. *True or false:* An AND gate output will always differ from an OR gate output for the same input conditions.

## 3-5 NOT OPERATION

The **NOT operation** is unlike the OR and AND operations because it can be performed on a single input variable. For example, if the variable *A* is subjected to the NOT operation, the result *x* can be expressed as

 $x = \overline{A}$ 

where the overbar represents the NOT operation. This expression is read as "*x* equals NOT *A*" or "*x* equals the *inverse* of *A*" or "*x* equals the *complement* of *A*." Each of these is in common usage, and all indicate that the logic value of  $x = \overline{A}$  is opposite to the logic value of *A*. The truth table in Figure 3-11(a) clarifies this for the two cases A = 0 and A = 1. That is,

$$0 = 1$$
 because 0 is not 1

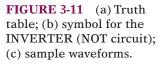
and

$$1 = \overline{0}$$
 because 1 is not 0

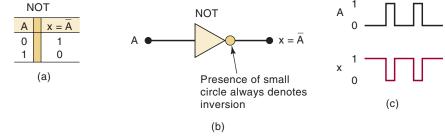
The NOT operation is also referred to as **inversion** or **complementation**, and these terms will be used interchangeably throughout the book. Although we will always use the overbar indicator to represent inversion, it is important to mention that another indicator for inversion is the prime symbol ('). That is,

$$A' = \overline{A}$$

Both should be recognized as indicating the inversion operation.







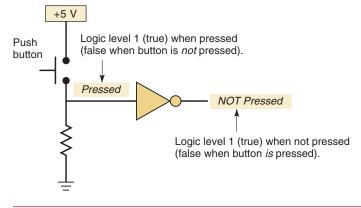
## **NOT Circuit (INVERTER)**

Figure 3-11(b) shows the symbol for a **NOT circuit**, which is more commonly called an **INVERTER**. This circuit *always* has only a single input, and its output logic level is always opposite to the logic level of this input. Figure 3-11(c) shows how the INVERTER affects an input signal. It inverts (complements) the input signal at all points on the waveform so that whenever the input = 0, output = 1, and vice versa.

#### **APPLICATION 3-1**

Figure 3-12 shows a typical application of the NOT gate. The push button is wired to produce a logic 1 (true) when it is pressed. Sometimes we want to know if the push button is not being pressed, and so this circuit provides an expression that is true when the button is not pressed.

**FIGURE 3-12** A NOT gate indicating a button is *not* pressed when its output is true.



## **Summary of Boolean Operations**

The rules for the OR, AND, and NOT operations may be summarized as follows:

OR	AND	NOT
0 + 0 = 0	$0\cdot 0=0$	$\overline{0} = 1$
0 + 1 = 1	$0 \cdot 1 = 0$	$\overline{1} = 0$
1 + 0 = 1	$1 \cdot 0 = 0$	
1 + 1 = 1	$1 \cdot 1 = 1$	

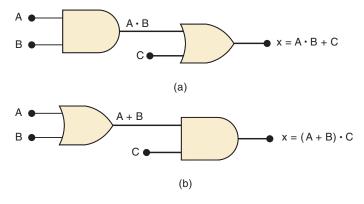
#### **REVIEW QUESTIONS**

- 1. The output of the INVERTER of Figure 3-11 is connected to the input of a second INVERTER. Determine the output level of the second INVERTER for each level of input *A*.
- 2. The output of the AND gate in Figure 3-7 is connected to the input of an INVERTER. Write the truth table showing the INVERTER output, *y*, for each combination of inputs *A* and *B*.

## 3-6 DESCRIBING LOGIC CIRCUITS ALGEBRAICALLY

Any logic circuit, no matter how complex, can be described completely using the three basic Boolean operations because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital systems. For example, consider FIGURE 3-13 (a) Logic circuit with its Boolean expression; (b) logic circuit whose expression requires parentheses.





the circuit in Figure 3-13(a). This circuit has three inputs, A, B, and C, and a single output, x. Utilizing the Boolean expression for each gate, we can easily determine the expression for the output.

The expression for the AND gate output is written  $A \cdot B$ . This AND output is connected as an input to the OR gate along with C, another input. The OR gate operates on its inputs so that its output is the OR sum of the inputs. Thus, we can express the OR output as  $x = A \cdot B + C$ . (This final expression could also be written as  $x = C + A \cdot B$  because it does not matter which term of the OR sum is written first.)

#### **Operator Precedence**

Occasionally, there may be confusion about which operation in an expression is performed first. The expression  $A \cdot B + C$  can be interpreted in two different ways: (1)  $A \cdot B$  is ORed with *C*, or (2) *A* is ANDed with the term B + C. To avoid this confusion, it will be understood that if an expression contains both AND and OR operations, the AND operations are performed first, unless there are *parentheses* in the expression, in which case the operation inside the parentheses is to be performed first. This is the same rule that is used in ordinary algebra to determine the order of operations.

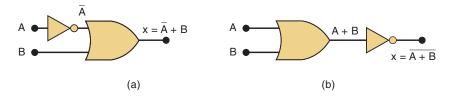
To illustrate further, consider the circuit in Figure 3-13(b). The expression for the OR gate output is simply A + B. This output serves as an input to the AND gate along with another input, *C*. Thus, we express the output of the AND gate as  $x = (A + B) \cdot C$ . Note the use of parentheses here to indicate that *A* and *B* are ORed *first*, before their OR sum is ANDed with *C*. Without the parentheses it would be interpreted *incorrectly*, because  $A + B \cdot C$  means that *A* is ORed with the product  $B \cdot C$ .

#### **Circuits Containing INVERTERs**

Whenever an INVERTER is present in a logic-circuit diagram, its output expression is simply equal to the input expression with a bar over it. Figure 3-14 shows two examples using INVERTERs. In Figure 3-14(a), input A is fed through an INVERTER, whose output is therefore  $\overline{A}$ . The INVERTER output is fed to an OR gate together with B, so that the OR output is equal to  $\overline{A} + B$ . Note that the bar is over the A alone, indicating that A is first inverted and then ORed with B.

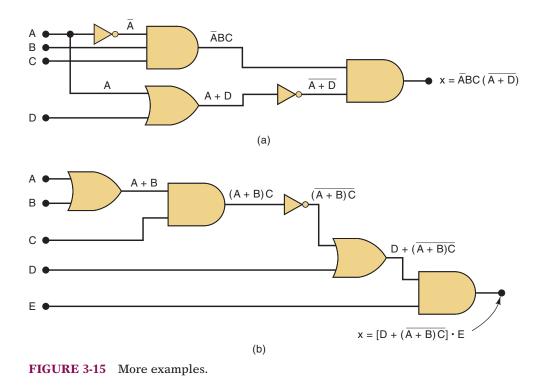
**FIGURE 3-14** Circuits using INVERTERs.





In Figure 3-14(b), the output of the OR gate is equal to A + B and is fed through an INVERTER. The INVERTER output is therefore equal to  $(\overline{A + B})$ because it inverts the *complete* input expression. Note that the bar covers the entire expression (A + B). This is important because, as will be shown later, the expressions  $(\overline{A + B})$  and  $(\overline{A + B})$  are *not* equivalent. The expression  $(\overline{A + B})$  means that A is ORed with B and then their OR sum is inverted, whereas the expression  $(\overline{A + B})$  indicates that A is inverted and B is inverted and the results are then ORed together.

Figure 3-15 shows two more examples, which should be studied carefully. Note especially the use of *two* separate sets of parentheses in Figure 3-15(b). Also notice in Figure 3-15(a) that the input variable A is connected as an input to two different gates.





#### **REVIEW QUESTIONS**

- 1. In Figure 3-15(a), change each AND gate to an OR gate, and change the OR gate to an AND gate. Then write the expression for output *x*.
- 2. In Figure 3-15(b), change each AND gate to an OR gate, and each OR gate to an AND gate. Then write the expression for *x*.

## 3-7 EVALUATING LOGIC-CIRCUIT OUTPUTS

Once we have the Boolean expression for a circuit output, we can obtain the output logic level for any set of input levels. For example, suppose that we want to know the logic level of the output x for the circuit in Figure 3-15(a) for the case where A = 0, B = 1, C = 1, and D = 1. As in ordinary algebra,

the value of *x* can be found by "plugging" the values of the variables into the expression and performing the indicated operations as follows:

$$x = ABC(A + D)$$
  
=  $\overline{0} \cdot 1 \cdot 1 \cdot (\overline{0} + 1)$   
=  $1 \cdot 1 \cdot 1 \cdot (\overline{0} + 1)$   
=  $1 \cdot 1 \cdot 1 \cdot (\overline{1})$   
=  $1 \cdot 1 \cdot 1 \cdot 0$   
=  $0$ 

As another illustration, let us evaluate the output of the circuit in Figure 3-15(b) for A = 0, B = 0, C = 1, D = 1, and E = 1.

$$x = [D + \overline{(A + B)C}] \cdot E$$
  
=  $[1 + \overline{(0 + 0) \cdot 1}] \cdot 1$   
=  $[1 + \overline{0} \cdot 1] \cdot 1$   
=  $[1 + \overline{0}] \cdot 1$   
=  $[1 + 1] \cdot 1$   
=  $1 \cdot 1$   
=  $1$ 

In general, the following rules must always be followed when evaluating a Boolean expression:

- 1. First, perform all inversions of single terms; that is,  $\overline{0} = 1$  or  $\overline{1} = 0$ .
- 2. Then perform all operations within parentheses.
- 3. Perform an AND operation before an OR operation unless parentheses indicate otherwise.
- 4. If an expression has a bar over it, perform the operations inside the expression first and then invert the result.

For practice, determine the outputs of both circuits in Figure 3-15 for the case where all inputs are 1. The answers are x = 0 and x = 1, respectively.

#### Analysis Using a Table

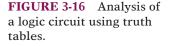
Whenever you have a combinational logic circuit and you want to know how it works, the best way to analyze it is to use a truth table. The advantages of this method are:

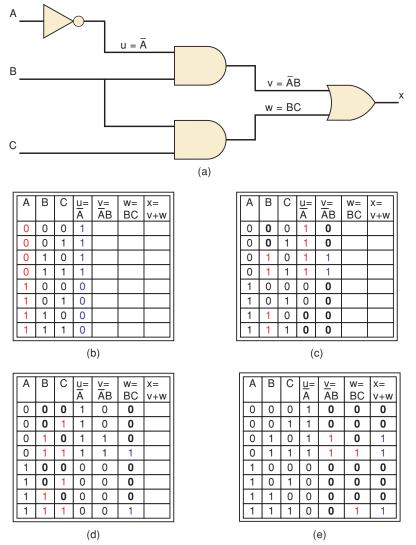
It allows you to analyze one gate or logic combination at a time.

It allows you to easily double-check your work.

When you are done, you have a table that is of tremendous benefit in troubleshooting the logic circuit.

Recall that a truth table lists all the possible input combinations in numerical order. For each possible input combination, we can determine the logic state at every point (node) in the logic circuit including the output. For example refer to Figure 3-16(a). There are several intermediate nodes in this circuit that are neither inputs nor outputs to the circuit. They are simply connections between one gate's output and another gate's input. In this diagram they have been labeled u, v, and w. The first step after listing all the input combinations is to create a column in the truth table for each intermediate signal (node) as shown in Figure 3-16(b). Node u has been filled in as the complement of A.





The next step is to fill in the values for column v as shown in Figure 3-16(c). From the diagram we can see that  $v = \overline{AB}$ . The node v should be HIGH when  $\overline{A}$  (node u) is HIGH AND B is HIGH. This occurs whenever A is LOW AND B is HIGH. The third step is to predict the values at node w which is the logical product of BC. This column is HIGH whenever B is HIGH AND C is HIGH as shown in Figure 3-16(d). The final step is to logically combine columns v and w to predict the output x. Since x = v + w, the x output will be HIGH when v is HIGH OR w is HIGH as shown in Figure 3-16(e).

If you built this circuit and it was not producing the correct output for *x* under all conditions, this table could be used to find the trouble. The general procedure is to test the circuit under each combination of inputs. If any input combination produces an incorrect output (i.e., a fault), compare the actual logic state of each intermediate node in the circuit with the correct theoretical value in the table while applying that input condition. If the logic state for an intermediate node is *correct*, the problem must be farther to the right of that node. If the logic state for an intermediate node is *shorted* to something). Detailed troubleshooting procedures and possible circuit faults will be covered more extensively in Chapter 4.

## EXAMPLE 3-6

Analyze the operation of Figure 3-15(a) by creating a table showing the logic state at each node of the circuit.

#### Solution

Fill in the column for *t* by entering a 1 for all entries where A = 0 and B = 1 and C = 1.

Fill in the column for u by entering a 1 for all entries where A = 1 or D = 1. Fill in the column for v by complementing all entries in column u.

Fill in the column for *x* by entering a 1 for all entries where t = 1 and v = 1.

Α	В	С	D	$t = \overline{A}BC$	u = A + D	$v = \overline{A + D}$	x = tv
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	0	1	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	1
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0
1	1	1	1	0	1	0	0

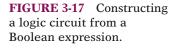
#### **REVIEW QUESTIONS**

- 1. Use the expression for x to determine the output of the circuit in Figure 3-15(a) for the conditions A = 0, B = 1, C = 1, and D = 0.
- 2. Use the expression for x to determine the output of the circuit in Figure 3-15(b) for the conditions A = B = E = 1, C = D = 0.
- 3. Determine the answers to Questions 1 and 2 by finding the logic levels present at each gate output using a table as in Figure 3-16.

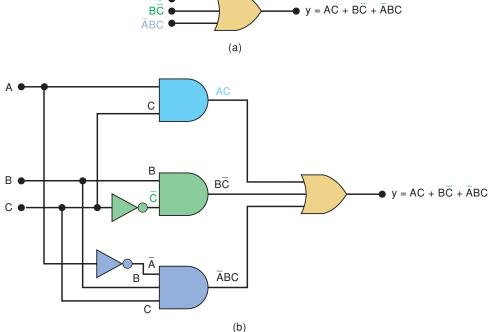
## **3-8 IMPLEMENTING CIRCUITS FROM BOOLEAN EXPRESSIONS**

When the operation of a circuit is defined by a Boolean expression, we can draw a logic-circuit diagram directly from that expression. For example, if we needed a circuit that was defined by  $x = A \cdot B \cdot C$ , we would immediately know that all that was needed was a three-input AND gate. If we needed a circuit that was defined by  $x = A + \overline{B}$ , we would use a two-input OR gate with an INVERTER on one of the inputs. The same reasoning used for these simple cases can be extended to more complex circuits.

Suppose that we wanted to construct a circuit whose output is  $y = AC + B\overline{C} + \overline{ABC}$ . This Boolean expression contains three terms  $(AC, B\overline{C}, \overline{ABC})$ , which are ORed together. This tells us that a three-input OR gate is required with inputs that are equal to  $AC, B\overline{C}$ , and  $\overline{ABC}$ . This is illustrated in Figure 3-17(a), where a three-input OR gate is drawn with inputs labeled as  $AC, B\overline{C}$ , and  $\overline{ABC}$ .







Each OR gate input is an AND product term, which means that an AND gate with appropriate inputs can be used to generate each of these terms. This is shown in Figure 3-17(b), which is the final circuit diagram. Note the use of INVERTERs to produce the  $\overline{A}$  and  $\overline{C}$  terms required in the expression.

This same general approach can always be followed, although we shall find that there are some clever, more efficient techniques that can be employed. For now, however, this straightforward method will be used to minimize the number of new items that are to be learned.

#### **EXAMPLE 3-7**

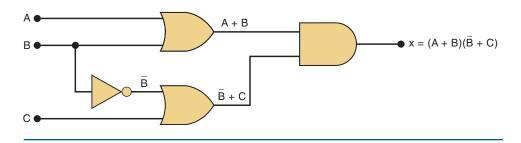
Draw the circuit diagram to implement the expression  $x = (A + B)(\overline{B} + C)$ .

#### Solution

This expression shows that the terms A + B and  $\overline{B} + C$  are inputs to an AND gate, and each of these two terms is generated from a separate OR gate. The result is drawn in Figure 3-18.

**FIGURE 3-18** Example 3-7.





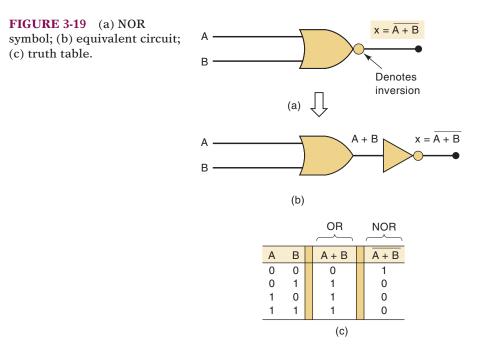
3. Draw the circuit diagram for  $x = [D + (\overline{A + B})C)] \cdot E$ .

## **3-9 NOR GATES AND NAND GATES**

Two other types of logic gates, NOR gates and NAND gates, are widely used in digital circuits. These gates actually combine the basic AND, OR, and NOT operations, so it is a relatively simple matter to write their Boolean expressions.

## **NOR Gate**

The symbol for a two-input **NOR gate** is shown in Figure 3-19(a). It is the same as the OR gate symbol except that it has a small circle on the output. The small circle represents the inversion operation. Thus, the NOR gate operates like an OR gate followed by an INVERTER, so that the circuits in Figure 3-19(a) and (b) are equivalent, and the output expression for the NOR gate is  $x = \overline{A + B}$ .



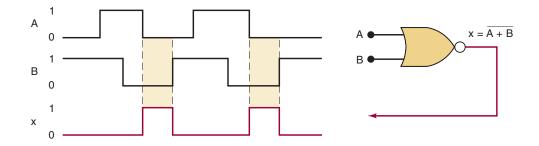
The truth table in Figure 3-19(c) shows that the NOR gate output is the exact inverse of the OR gate output for all possible input conditions. An OR gate output goes HIGH when any input is HIGH; the NOR gate output goes LOW when any input is HIGH. This same operation can be extended to NOR gates with more than two inputs.

## EXAMPLE 3-8

Determine the waveform at the output of a NOR gate for the input waveforms shown in Figure 3-20.

**FIGURE 3-20** Example 3-8.





#### Solution

One way to determine the NOR output waveform is to find first the OR output waveform and then invert it (change all 1s to 0s, and vice versa). Another way utilizes the fact that a NOR gate output will be HIGH *only* when all inputs are LOW. Thus, you can examine the input waveforms, find those time intervals where they are all LOW, and make the NOR output HIGH for those intervals. The NOR output will be LOW for all other time intervals. The resultant output waveform is shown in the figure.

#### **EXAMPLE 3-9**

Determine the Boolean expression for a three-input NOR gate followed by an INVERTER.

#### Solution

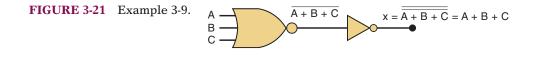
Refer to Figure 3-21, where the circuit diagram is shown. The expression at the NOR output is  $(\overline{A + B + C})$ , which is then fed through an INVERTER to produce

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

The presence of the double inversion signs indicates that the quantity (A + B + C) has been inverted and then inverted again. It should be clear that this simply results in the expression (A + B + C) being unchanged. That is,

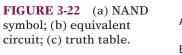
$$x = (\overline{A + B + C}) = (A + B + C)$$

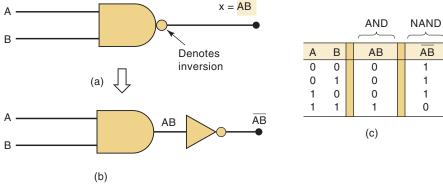
Whenever two inversion bars are over the same variable or quantity, they <u>cancel</u> each other out, as in the example above. However, in cases such as  $\overline{A} + \overline{B}$  the inversion bars do not cancel. This is because the smaller inversion bars invert the single variables A and B, while the wide bar inverts the quantity ( $\overline{A} + \overline{B}$ ). Thus,  $\overline{A} + \overline{B} \neq A + B$ . Similarly,  $\overline{A}\overline{B} \neq AB$ .



## **NAND Gate**

The symbol for a two-input **NAND** gate is shown in Figure 3-22(a). It is the same as the AND gate symbol except for the small circle on its output. Once again, this small circle denotes the inversion operation. Thus, the NAND operates like an AND gate followed by an INVERTER, so that the circuits of Figure 3-22(a) and (b) are equivalent, and the output expression for the NAND gate is  $x = \overline{AB}$ .





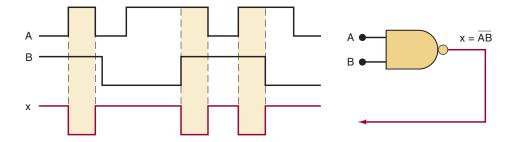
The truth table in Figure 3-22(c) shows that the NAND gate output is the exact inverse of the AND gate for all possible input conditions. The AND output goes HIGH only when all inputs are HIGH, while the NAND output goes LOW only when all inputs are HIGH. This same characteristic is true of NAND gates having more than two inputs.

Determine the output waveform of a NAND gate having the inputs shown in Figure 3-23.



EXAMPLE 3-10





#### Solution

One way is to draw first the output waveform for an AND gate and then invert it. Another way utilizes the fact that a NAND output will be LOW only when all inputs are HIGH. Thus, you can find those time intervals during which the inputs are all HIGH, and make the NAND output LOW for those intervals. The output will be HIGH at all other times.

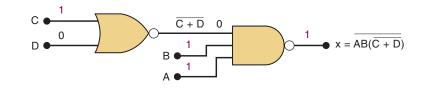
## EXAMPLE 3-11

Implement the logic circuit that has the expression  $x = AB \cdot (C + D)$  using only NOR and NAND gates.

#### Solution

The  $(\overline{C} + \overline{D})$  term is the expression for the output of a NOR gate. This term is ANDed with *A* and *B*, and the result is inverted; this, of course, is the NAND operation. Thus, the circuit is implemented as shown in Figure 3-24. Note that the NAND gate first ANDs the *A*, *B*, and  $(\overline{C} + \overline{D})$  terms, and then it inverts the *complete* result.





#### **EXAMPLE 3-12**

Determine the output level in Figure 3-24 for A = B = C = 1 and D = 0.

#### Solution

In the first method we use the expression for *x*.

$$x = \underline{AB(\overline{C+D})} = \overline{\underbrace{\frac{1 \cdot 1 \cdot (\overline{1}+0)}{1 \cdot 1 \cdot (\overline{1})}}} = \overline{\underbrace{\frac{1 \cdot 1 \cdot (\overline{1})}{1 \cdot 1 \cdot 0}} = \overline{0} = 1$$

In the second method, we write down the input logic levels on the circuit diagram (shown in color in Figure 3-24) and follow these levels through each gate to the final output. The NOR gate has inputs of 1 and 0 to produce an output of 0 (an OR would have produced an output of 1). The NAND gate thus has input levels of 0, 1, and 1 to produce an output of 1 (an AND would have produced an output of 0).

#### **REVIEW QUESTIONS**

- 1. What is the only set of input conditions that will produce a HIGH output from a three-input NOR gate?
- 2. Determine the output level in Figure 3-24 for A = B = 1, C = D = 0.
- 3. Change the NOR gate of Figure 3-24 to a NAND gate, and change the NAND to a NOR. What is the new expression for *x*?

#### 3-10 BOOLEAN THEOREMS

We have seen how Boolean algebra can be used to help analyze a logic circuit and express its operation mathematically. We will continue our study of Boolean algebra by investigating the various **Boolean theorems** (rules) that can help us to simplify logic expressions and logic circuits. The first group of theorems is given in Figure 3-25. In each theorem, x is a logic variable that

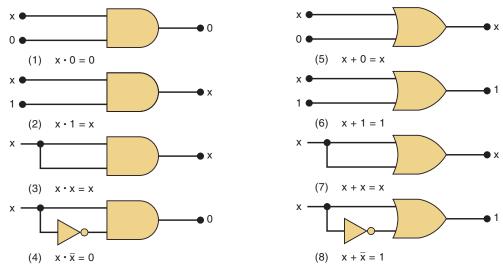


FIGURE 3-25 Single-variable theorems.

can be either a 0 or a 1. Each theorem is accompanied by a logic-circuit diagram that demonstrates its validity.

Theorem (1) states that if any variable is ANDed with 0, the result must be 0. This is easy to remember because the AND operation is just like ordinary multiplication, where we know that anything multiplied by 0 is 0. We also know that the output of an AND gate will be 0 whenever any input is 0, regardless of the level on the other input.

Theorem (2) is also obvious by comparison with ordinary multiplication. Theorem (3) can be proved by trying each case. If x = 0, then  $0 \cdot 0 = 0$ ; if

x = 1, then  $1 \cdot 1 = 1$ . Thus,  $x \cdot x = x$ .

Theorem (4) can be proved in the same manner. However, it can also be reasoned that at any time either x or its inverse  $\overline{x}$  must be at the 0 level, and so their AND product always must be 0.

Theorem (5) is straightforward, since 0 *added* to anything does not affect its value, either in regular addition or in OR addition.

Theorem (6) states that if any variable is ORed with 1, the result will always be 1. We check this for both values of x: 0 + 1 = 1 and 1 + 1 = 1. Equivalently, we can remember that an OR gate output will be 1 when *any* input is 1, regardless of the value of the other input.

Theorem (7) can be proved by checking for both values of x: 0 + 0 = 0 and 1 + 1 = 1.

Theorem (8) can be proved similarly, or we can just reason that at any time either x or  $\overline{x}$  must be at the 1 level so that we are always ORing a 0 and a 1, which always results in 1.

Before introducing any more theorems, we should point out that when theorems (1) through (8) are applied, the variable *x* may actually represent an expression containing more than one variable. For example, if we have  $A\overline{B}(A\overline{B})$ , we can invoke theorem (4) by letting  $x = A\overline{B}$ . Thus, we can say that  $A\overline{B}(A\overline{B}) = 0$ . The same idea can be applied to the use of any of these theorems.

## **Multivariable Theorems**

The theorems presented below involve more than one variable:

(9)	x + y = y + x
(10)	$x \cdot y = y \cdot x$

(11)	x + (y + z) = (x + y) + z = x + y + z
(12)	x(yz) = (xy)z = xyz
(13 <i>a</i> )	x(y+z) = xy + xz
(13 <i>b</i> )	(w+x)(y+z) = wy + xy + wz + xz
(14)	x + xy = x
(15a)	$x + \overline{x}y = x + y$
(15 <i>b</i> )	$\overline{x} + xy = \overline{x} + y$

Theorems (9) and (10) are called the *commutative laws*. These laws indicate that the order in which we OR or AND two variables is unimportant; the result is the same.

Theorems (11) and (12) are the *associative laws*, which state that we can group the variables in an AND expression or OR expression any way we want.

Theorem (13) is the *distributive law*, which states that an expression can be expanded by multiplying term by term just the same as in ordinary algebra. This theorem also indicates that we can factor an expression. That is, if we have a sum of two (or more) terms, each of which contains a common variable, the common variable can be factored out just as in ordinary algebra. For example, if we have the expression  $\overline{ABC} + \overline{ABC}$ , we can factor out the  $\overline{B}$  variable:

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

As another example, consider the expression ABC + ABD. Here the two terms have the variables A and B in common, and so  $A \cdot B$  can be factored out of both terms. That is,

$$ABC + ABD = AB(C + D)$$

Theorems (9) to (13) are easy to remember and use because they are identical to those of ordinary algebra. Theorems (14) and (15), on the other hand, do not have any counterparts in ordinary algebra. Each can be proved by trying all possible cases for x and y. This is illustrated (for theorem 14) by creating an analysis table for the equation x + xy as follows:

x	У	ху	x + xy
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Notice that the value of the entire expression (x + xy) is always the same as *x*.

Theorem (14) can also be proved by factoring and using theorems (6) and (2) as follows:

$$x + xy = x(1 + y)$$
  
= x \cdot 1 [using theorem (6)]  
= x [using theorem (2)]

All of these Boolean theorems can be useful in simplifying a logic expression—that is, in reducing the number of terms in the expression. When this is done, the reduced expression will produce a circuit that is less complex than the one that the original expression would have produced. A good portion of the next chapter will be devoted to the process of circuit simplification. For now, the following examples will serve to illustrate how the Boolean theorems can be applied. **Note:** You can find all the Boolean theorems on the inside back cover.

Solution Factor out the common variables  $A\overline{B}$  using theorem (13):  $y = A\overline{B}(D + \overline{D})$ Using theorem (8), the term in parentheses is equivalent to 1. Thus,  $y = A\overline{B} \cdot 1$  $= A\overline{B}$  [using theorem (2)]

Simplify the expression  $y = A\overline{B}D + A\overline{B}\overline{D}$ .

**EXAMPLE 3-14** Simplify  $z = (\overline{A} + B)(A + B)$ .

#### Solution

The expression can be expanded by multiplying out the terms [theorem (13)]:

$$z = A \cdot A + A \cdot B + B \cdot A + B \cdot B$$

Invoking theorem (4), the term  $\overline{A} \cdot A = 0$ . Also,  $B \cdot B = B$  [theorem (3)]:

$$z = 0 + \overline{A} \cdot B + B \cdot A + B = \overline{A}B + AB + B$$

Factoring out the variable B [theorem (13)], we have

$$z = B(\overline{A} + A + 1)$$

Finally, using theorems (2) and (6),

z = B

**EXAMPLE 3-15** 

EXAMPLE 3-13

Simplify  $x = ACD + \overline{ABCD}$ .

#### Solution

Factoring out the common variables CD, we have

$$x = CD(A + AB)$$

Utilizing theorem (15a), we can replace  $A + \overline{AB}$  by A + B, so

$$x = CD(A + B)$$
$$= ACD + BCD$$

**REVIEW QUESTIONS** 

- 1. Use theorems (13) and (14) to simplify  $y = A\overline{C} + AB\overline{C}$ .
- 2. Use theorems (13) and (8) to simplify  $y = \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D}$ .
  - 3. Use theorems (13) and (15b) to simplify  $y = \overline{AD} + ABD$ .

## 3-11 DEMORGAN'S THEOREMS

Two of the most important theorems of Boolean algebra were contributed by a great mathematician named DeMorgan. **DeMorgan's theorems** are extremely useful in simplifying expressions in which a product or sum of variables is inverted. The two theorems are:

(16) 
$$(\overline{x+y}) = \overline{x} \cdot \overline{y}$$
  
(17)  $(\overline{x \cdot y}) = \overline{x} + \overline{y}$ 

Theorem (16) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables. Theorem (17) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them. Each of DeMorgan's theorems can readily be proven by checking for all possible combinations of x and y. This will be left as an end-of-chapter exercise.

Although these theorems have been stated in terms of single variables x and y, they are equally valid for situations where x and/or y are expressions that contain more than one variable. For example, let's apply them to the expression  $(\overline{AB} + C)$  as shown below:

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

Note that we used theorem (16) and treated  $A\overline{B}$  as x and C as y. The result can be further simplified because we have a product  $A\overline{B}$  that is inverted. Using theorem (17), the expression becomes

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

Notice that we can replace  $\overline{B}$  by *B*, so that we finally have

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

This final result contains only inverter signs that invert a single variable.

## EXAMPLE 3-16

Simplify the expression  $z = (\overline{A} + C) \cdot (B + \overline{D})$  to one having only single variables inverted.

#### Solution

Using theorem (17), and treating  $(\overline{A} + C)$  as x and  $(B + \overline{D})$  as y, we have

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

We can think of this as breaking the large inverter sign down the middle and changing the AND sign ( $\cdot$ ) to an OR sign (+). Now the term ( $\overline{\overline{A} + C}$ ) can be simplified by applying theorem (16). Likewise, ( $\overline{B + D}$ ) can be simplified:

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

Here we have broken the larger inverter signs down the middle and replaced the (+) with a  $(\cdot)$ . Canceling out the double inversions, we have finally

$$z = AC + BD$$

Example 3-16 points out that when using DeMorgan's theorems to reduce an expression, we may break an inverter sign at any point in the expression and change the operator sign at that point in the expression to its opposite (+ is changed to  $\cdot$ , and vice versa). This procedure is continued until the expression is reduced to one in which only single variables are inverted. Two more examples are given below.

Example 1	Example 2
$z = \overline{A + \overline{B} \cdot C}$	$\omega = \overline{(A + BC) \cdot (D + EF)}$
$=\overline{A}\cdot(\overline{\overline{B}\cdot C})$	$= (\overline{A + BC}) + (\overline{D + EF})$
$=\overline{A}\cdot(\overline{\overline{B}}+\overline{C})$	$= (\overline{A} \cdot \overline{BC}) + (\overline{D} \cdot \overline{EF})$
$=\overline{A}\cdot(B+\overline{C})$	$= [\overline{A} \cdot (\overline{B} + \overline{C})] + [\overline{D} \cdot (\overline{E} + \overline{F})]$
	$=\overline{A}\overline{B}+\overline{A}\overline{C}+\overline{D}\overline{E}+\overline{D}\overline{F}$

DeMorgan's theorems are easily extended to more than two variables. For example, it can be proved that

$$\overline{x + \underbrace{y + z}_{\overline{x} \cdot \overline{y} \cdot \overline{z}}} = \overline{x} \cdot \overline{y} \cdot \overline{z}$$

Here, we see that the large inverter sign is broken at *two* points in the expression and the operator sign is changed to its opposite. This can be extended to any number of variables. Again, realize that the variables can themselves be expressions rather than single variables. Here is another example.

$$x = \overline{AB} \cdot \overline{CD} \cdot \overline{EF}$$
  
=  $\overline{\overline{AB}} + \overline{\overline{CD}} + \overline{\overline{EF}}$   
=  $AB + CD + EF$ 

## **Implications of DeMorgan's Theorems**

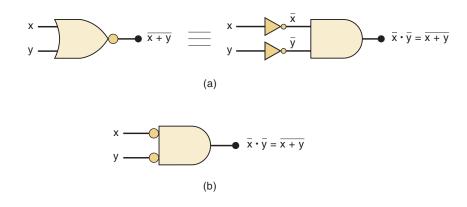
Let us examine theorems (16) and (17) from the standpoint of logic circuits. First, consider theorem (16):

$$\overline{x+y} = \overline{x} \cdot \overline{y}$$

The left-hand side of the equation can be viewed as the output of a NOR gate whose inputs are x and y. The right-hand side of the equation, on the other

#### **FIGURE 3-26**

(a) Equivalent circuits implied by theorem (16); (b) alternative symbol for the NOR function.

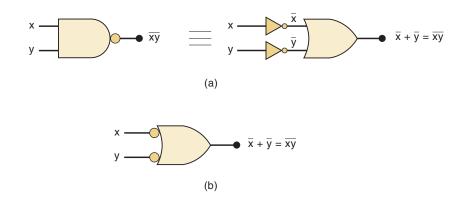


hand, is the result of first inverting both x and y and then putting them through an AND gate. These two representations are equivalent and are illustrated in Figure 3-26(a). What this means is that an AND gate with INVERTERs on each of its inputs is equivalent to a NOR gate. In fact, both representations are used to represent the NOR function. When the AND gate with inverted inputs is used to represent the NOR function, it is usually drawn as shown in Figure 3-26(b), where the small circles on the inputs represent the inversion operation.

Now consider theorem (17):

$$\overline{x \cdot y} = \overline{x} + \overline{y}$$

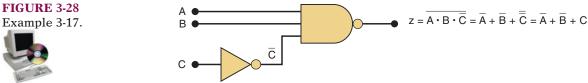
The left side of the equation can be implemented by a NAND gate with inputs x and y. The right side can be implemented by first inverting inputs xand y and then putting them through an OR gate. These two equivalent representations are shown in Figure 3-27(a). The OR gate with INVERTERs on each of its inputs is equivalent to the NAND gate. In fact, both representations are used to represent the NAND function. When the OR gate with inverted inputs is used to represent the NAND function, it is usually drawn as shown in Figure 3-27(b), where the circles again represent inversion.



#### **EXAMPLE 3-17**

**FIGURE 3-28** 

Determine the output expression for the circuit of Figure 3-28 and simplify it using DeMorgan's theorems.



#### **FIGURE 3-27** (a) Equivalent circuits

implied by theorem (17); (b) alternative symbol for the NAND function.

#### Solution

The expression for z is  $z = AB\overline{C}$ . Use DeMorgan's theorem to break the large inversion sign:

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

Cancel the double inversions over C to obtain

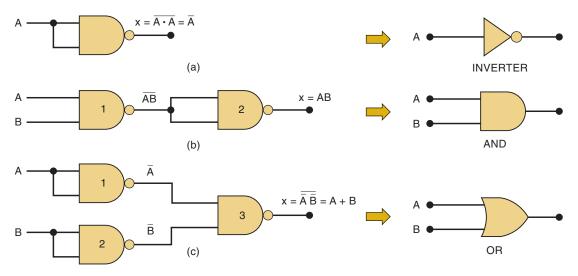
$$z = \overline{A} + \overline{B} + C$$

## **REVIEW QUESTIONS**

- 1. Use DeMorgan's theorems to convert the expression  $z = (A + B) \cdot \overline{C}$  to one that has only single-variable inversions.
- 2. Repeat question 1 for the expression  $y = R\overline{S}T + \overline{Q}$ .
- 3. Implement a circuit having output expression  $z = \overline{ABC}$  using only a NOR gate and an INVERTER.
- 4. Use DeMorgan's theorems to convert y = A + B + CD to an expression containing only single-variable inversions.

## 3-12 UNIVERSALITY OF NAND GATES AND NOR GATES

All Boolean expressions consist of various combinations of the basic operations of OR, AND, and INVERT. Therefore, any expression can be implemented using combinations of OR gates, AND gates, and INVERTERs. It is possible, however, to implement any logic expression using *only* NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations OR, AND, and INVERT. This is demonstrated in Figure 3-29.



#### FIGURE 3-29 NAND gates can be used to implement any Boolean function.

First, in Figure 3-29(a), we have a two-input NAND gate whose inputs are purposely connected together so that the variable A is applied to both. In this configuration, the NAND simply acts as INVERTER because its output is  $x = \overline{A \cdot A} = \overline{A}$ .

In Figure 3-29(b), we have two NAND gates connected so that the AND operation is performed. NAND gate 2 is used as an INVERTER to change  $\overline{AB}$  to  $\overline{AB} = AB$ , which is the desired AND function.

The OR operation can be implemented using NAND gates connected as shown in Figure 3-29(c). Here NAND gates 1 and 2 are used as INVERTERs to invert the inputs, so that the final output is  $x = \overline{A} \cdot \overline{B}$ , which can be simplified to x = A + B using DeMorgan's theorem.

In a similar manner, it can be shown that NOR gates can be arranged to implement any of the Boolean operations. This is illustrated in Figure 3-30. Part (a) shows that a NOR gate with its inputs connected together behaves as an INVERTER because the output is  $x = \overline{A} + \overline{A} = \overline{A}$ .

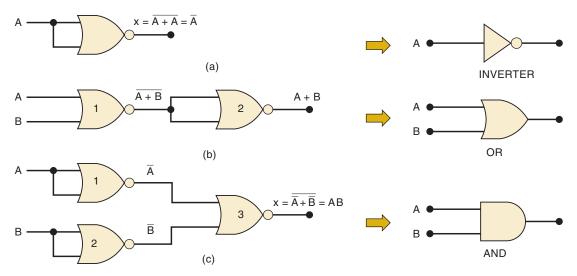


FIGURE 3-30 NOR gates can be used to implement any Boolean operation.

In Figure 3-30(b), two NOR gates are arranged so that the OR operation is performed. NOR gate 2 is used as an INVERTER to change  $\overline{A + B}$  to  $\overline{\overline{A + B}} = A + B$ , which is the desired OR function.

The AND operation can be implemented with NOR gates as shown in Figure 3-30(c). Here, NOR gates 1 and 2 are used as INVERTERs to invert the inputs, so that the final output is  $x = \overline{A} + \overline{B}$ , which can be simplified to  $x = A \cdot B$  by use of DeMorgan's theorem.

Since any of the Boolean operations can be implemented using only NAND gates, any logic circuit can be constructed using only NAND gates. The same is true for NOR gates. This characteristic of NAND and NOR gates can be very useful in logic-circuit design, as Example 3-18 illustrates.

#### **EXAMPLE 3-18**

In a certain manufacturing process, a conveyor belt will shut down whenever specific conditions occur. These conditions are monitored and reflected by the states of four logic signals as follows: signal A will be HIGH whenever the conveyor belt speed is too fast; signal B will be HIGH whenever the collection bin at the end of the belt is full; signal C will be HIGH when the belt tension is too high; signal D will be HIGH when the manual override is off.

A logic circuit is needed to generate a signal x that will go HIGH whenever conditions A and B exist simultaneously or whenever conditions Cand D exist simultaneously. Clearly, the logic expression for x will be x = AB + CD. The circuit is to be implemented with a minimum number of ICs. The TTL integrated circuits shown in Figure 3-31 are available. Each IC is a *quad*, which means that it contains *four* identical gates on one chip.

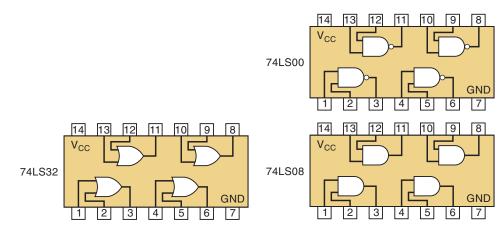


FIGURE 3-31 ICs available for Example 3-18.

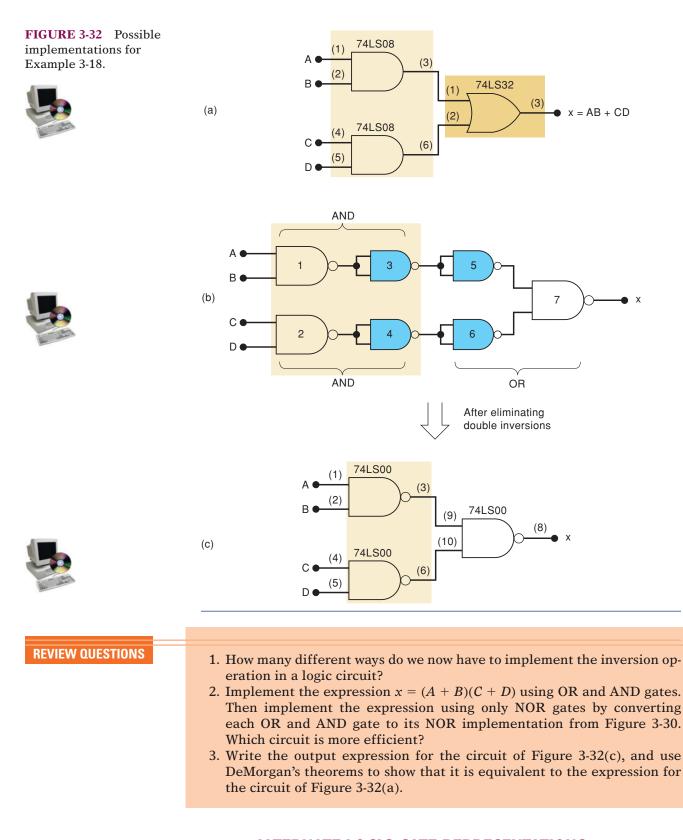
#### Solution

The straightforward method for implementing the given expression uses two AND gates and an OR gate, as shown in Figure 3-32(a). This implementation uses two gates from the 74LS08 IC and a single gate from the 74LS32 IC. The numbers in parentheses at each input and output are the pin numbers of the respective IC. These are always shown on any logic-circuit wiring diagram. For our purposes, most logic diagrams will not show pin numbers unless they are needed in the description of circuit operation.

Another implementation can be accomplished by taking the circuit of Figure 3-32(a) and replacing each AND gate and OR gate by its equivalent NAND gate implementation from Figure 3-29. The result is shown in Figure 3-32(b).

At first glance, this new circuit looks as if it requires seven NAND gates. However, NAND gates 3 and 5 are connected as INVERTERs in series and can be eliminated from the circuit because they perform a double inversion of the signal out of NAND gate 1. Similarly, NAND gates 4 and 6 can be eliminated. The final circuit, after eliminating the double INVERTERs, is drawn in Figure 3-32(c).

This final circuit is more efficient than the one in Figure 3-32(a) because it uses three two-input NAND gates that can be implemented from one IC, the 74LS00.



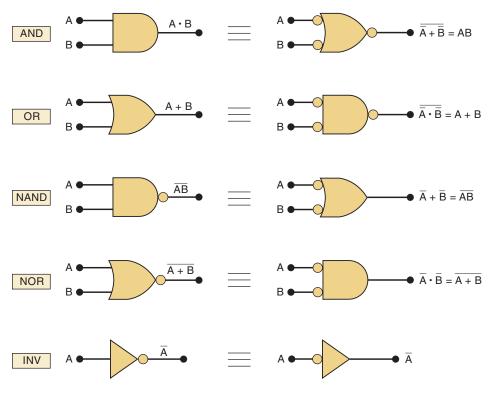
## 3-13 ALTERNATE LOGIC-GATE REPRESENTATIONS

We have introduced the five basic logic gates (AND, OR, INVERTER, NAND, and NOR) and the standard symbols used to represent them on logic-circuit diagrams. Although you may find that some circuit diagrams still use these

standard symbols exclusively, it has become increasingly more common to find circuit diagrams that utilize **alternate logic symbols** *in addition* to the standard symbols.

Before discussing the reasons for using an alternate symbol for a logic gate, we will present the alternate symbols for each gate and show that they are equivalent to the standard symbols. Refer to Figure 3-33; the left side of the illustration shows the standard symbol for each logic gate, and the right side shows the alternate symbol. The alternate symbol for each gate is obtained from the standard symbol by doing the following:

- 1. Invert each input and output of the standard symbol. This is done by adding bubbles (small circles) on input and output lines that do not have bubbles and by removing bubbles that are already there.
- 2. Change the operation symbol from AND to OR, or from OR to AND. (In the special case of the INVERTER, the operation symbol is not changed.)



For example, the standard NAND symbol is an AND symbol with a bubble on its output. Following the steps outlined above, remove the bubble from the output, and add a bubble to each input. Then change the AND symbol to an OR symbol. The result is an OR symbol with bubbles on its inputs.

We can easily prove that this alternate symbol is equivalent to the standard symbol by using DeMorgan's theorems and recalling that the bubble represents an inversion operation. The output expression from the standard NAND symbol is  $\overline{AB} = \overline{A} + \overline{B}$ , which is the same as the output expression for the alternate symbol. This same procedure can be followed for each pair of symbols in Figure 3-33.

Several points should be stressed regarding the logic symbol equivalences:

- 1. The equivalences can be extended to gates with any number of inputs.
- 2. None of the standard symbols have bubbles on their inputs, and all the alternate symbols do.

FIGURE 3-33 Standard and alternate symbols for various logic gates and inverter.

- 3. The standard and alternate symbols for each gate represent the same physical circuit; *there is no difference in the circuits represented by the two symbols.*
- 4. NAND and NOR gates are inverting gates, and so both the standard and the alternate symbols for each will have a bubble on *either* the input or the output. AND and OR gates are *noninverting* gates, and so the alternate symbols for each will have bubbles on *both* inputs and output.

#### **Logic-Symbol Interpretation**

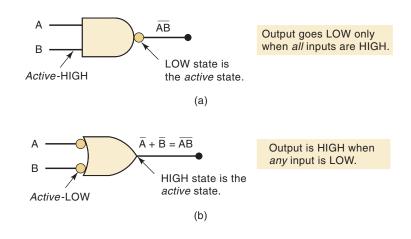
Each of the logic-gate symbols of Figure 3-33 provides a unique interpretation of how the gate operates. Before we can demonstrate these interpretations, we must first establish the concept of **active logic levels**.

When an input or output line on a logic circuit symbol has *no bubble* on it, that line is said to be **active-HIGH**. When an input or output line *does* have a *bubble* on it, that line is said to be **active-LOW**. The presence or absence of a bubble, then, determines the active-HIGH/active-LOW status of a circuit's inputs and output, and is used to interpret the circuit operation.

To illustrate, Figure 3-34(a) shows the standard symbol for a NAND gate. The standard symbol has a bubble on its output and no bubbles on its inputs. Thus, it has an active-LOW output and active-HIGH inputs. The logic operation represented by this symbol can therefore be interpreted as follows:

#### The output goes LOW only when all of the inputs are HIGH.

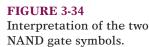
Note that this says that the output will go to its active state only when *all* of the inputs are in their active states. The word *all* is used because of the AND symbol.



The alternate symbol for a NAND gate shown in Figure 3-34(b) has an active-HIGH output and active-LOW inputs, and so its operation can be stated as follows:

#### The output goes HIGH when any input is LOW.

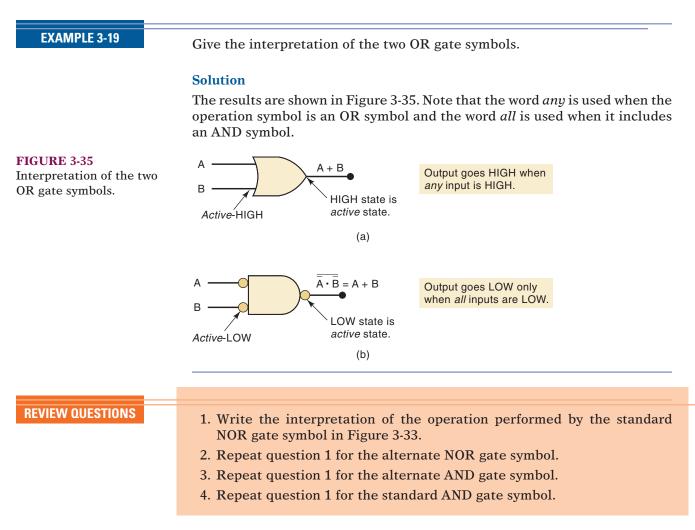
This says that the output will be in its active state whenever *any* of the inputs is in its active state. The word *any* is used because of the OR symbol. With a little thought, you can see that the two interpretations for the NAND symbols in Figure 3-34 are different ways of saying the same thing.



#### Summary

At this point you are probably wondering why there is a need to have two different symbols and interpretations for each logic gate. We hope the reasons will become clear after reading the next section. For now, let us summarize the important points concerning the logic-gate representations.

- 1. To obtain the alternate symbol for a logic gate, take the standard symbol and change its operation symbol (OR to AND, or AND to OR), and change the bubbles on both inputs and output (i.e., delete bubbles that are present, and add bubbles where there are none).
- 2. To interpret the logic-gate operation, first note which logic state, 0 or 1, is the active state for the inputs and which is the active state for the output. Then realize that the output's active state is produced by having *all* of the inputs in their active state (if an AND symbol is used) or by having *any* of the inputs in its active state (if an OR symbol is used).

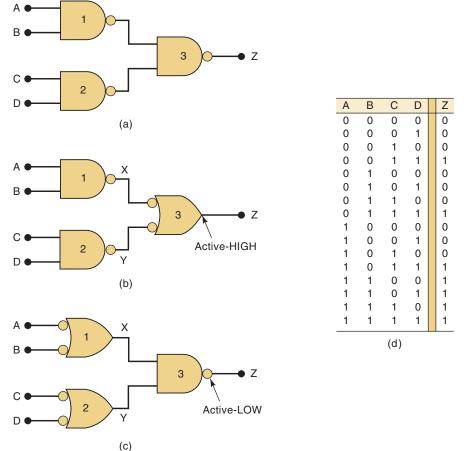


## 3-14 WHICH GATE REPRESENTATION TO USE

Some logic-circuit designers and some textbooks use only the standard logicgate symbols in their circuit schematics. While this practice is not incorrect, it does nothing to make the circuit operation easier to follow. Proper use of the alternate gate symbols in the circuit diagram can make the circuit operation

89

FIGURE 3-36 (a) Original circuit using standard NAND symbols; (b) equivalent representation where output Z is active-HIGH; (c) equivalent representation where output Z is active-LOW; (d) truth table.



much clearer. This can be illustrated by considering the example shown in Figure 3-36.

The circuit in Figure 3-36(a) contains three NAND gates connected to produce an output Z that depends on inputs A, B, C, and D. The circuit diagram uses the standard symbol for each of the NAND gates. While this diagram is logically correct, it does not facilitate an understanding of how the circuit functions. The circuit representations given in Figures 3-36(b) and (c), however, can be analyzed more easily to determine the circuit operation.

The representation of Figure 3-36(b) is obtained from the original circuit diagram by replacing NAND gate 3 with its alternate symbol. In this diagram, output Z is taken from a NAND gate symbol that has an active-HIGH output. Thus, we can say that Z will go HIGH when either X or Y is LOW. Now, since X and Y each appear at the output of NAND symbols having active-LOW outputs, we can say that X will go LOW only if A = B = 1, and Y will go LOW only if C = D = 1. Putting this all together, we can describe the circuit operation as follows:

# Output Z will go HIGH whenever either A = B = 1 or C = D = 1 (or both).

This description can be translated to truth-table form by setting Z = 1 for those cases where A = B = 1 and for those cases where C = D = 1. For all other cases, *Z* is made a 0. The resultant truth table is shown in Figure 3-36(d).

The representation of Figure 3-36(c) is obtained from the original circuit diagram by replacing NAND gates 1 and 2 by their alternate symbols. In this

equivalent representation, the Z output is taken from a NAND gate that has an active-LOW output. Thus, we can say that Z will go LOW only when X = Y = 1. Because X and Y are active-HIGH outputs, we can say that X will be HIGH when either A or B is LOW, and Y will be HIGH when either C or D is LOW. Putting this all together, we can describe the circuit operation as follows:

## Output Z will go LOW only when A or B is LOW and C or D is LOW.

This description can be translated to truth-table form by making Z = 0 for all cases where at least one of the *A* or *B* inputs is LOW at the same time that at least one of the *C* or *D* inputs is LOW. For all other cases, *Z* is made a 1. The resultant truth table is the same as that obtained for the circuit diagram of Figure 3-36(b).

# Which Circuit Diagram Should Be Used?

The answer to this question depends on the particular function being performed by the circuit output. If the circuit is being used to cause some action (e.g., turn on an LED or activate another logic circuit) when output Z goes to the 1 state, then we say that Z is to be active-HIGH, and the circuit diagram of Figure 3-36(b) should be used. On the other hand, if the circuit is being used to cause some action when Z goes to the 0 state, then Z is to be active-LOW, and the diagram of Figure 3-36(c) should be used.

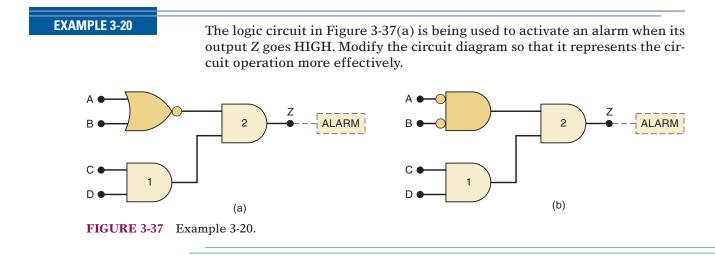
Of course, there will be situations where *both* output states are used to produce different actions and either one can be considered to be the active state. For these cases, either circuit representation can be used.

# **Bubble Placement**

Refer to the circuit representation of Figure 3-36(b) and note that the symbols for NAND gates 1 and 2 were chosen to have active-LOW outputs to match the active-LOW inputs of NAND gate 3. Refer to the circuit representation of Figure 3-36(c) and note that the symbols for NAND gates 1 and 2 were chosen to have active-HIGH outputs to match the active-HIGH inputs of NAND gate 3. This leads to the following general rule for preparing logic-circuit schematics:

Whenever possible, choose gate symbols so that bubble outputs are connected to bubble inputs, and nonbubble outputs to nonbubble inputs.

The following examples will show how this rule can be applied.



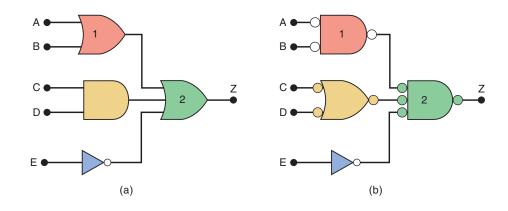
#### Solution

Because Z = 1 will activate the alarm, Z is to be active-HIGH. Thus, the AND gate 2 symbol does not have to be changed. The NOR gate symbol should be changed to the alternate symbol with a nonbubble (active-HIGH) output to match the nonbubble input of AND gate 2, as shown in Figure 3-37(b). Note that the circuit now has nonbubble outputs connected to the nonbubble inputs of gate 2.

# EXAMPLE 3-21

FIGURE 3-38 Example 3-21.

When the output of the logic circuit in Figure 3-38(a) goes LOW, it activates another logic circuit. Modify the circuit diagram to represent the circuit operation more effectively.



#### Solution

Because Z is to be active-LOW, the symbol for OR gate 2 must be changed to its alternate symbol, as shown in Figure 3-38(b). The new OR gate 2 symbol has bubble inputs, and so the AND gate and OR gate 1 symbols must be changed to bubbled outputs, as shown in Figure 3-38(b). The INVERTER already has a bubble output. Now the circuit has all bubble outputs connected to bubble inputs of gate 2.

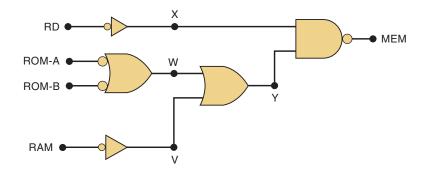
# **Analyzing Circuits**

When a logic-circuit schematic is drawn using the rules we followed in these examples, it is much easier for an engineer or technician (or student) to follow the signal flow through the circuit and to determine the input conditions that are needed to activate the output. This will be illustrated in the following examples—which, incidentally, use circuit diagrams taken from the logic schematics of an actual microcomputer.

# EXAMPLE 3-22

The logic circuit in Figure 3-39 generates an output, *MEM*, that is used to activate the memory ICs in a particular microcomputer. Determine the input conditions necessary to activate *MEM*.





## Solution

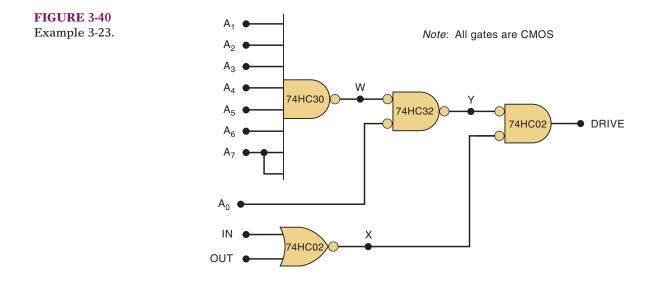
One way to do this would be to write the expression for *MEM* in terms of the inputs *RD*, *ROM-A*, *ROM-B*, and *RAM*, and to evaluate it for the 16 possible combinations of these inputs. While this method would work, it would require a lot more work than is necessary.

A more efficient method is to interpret the circuit diagram using the ideas we have been developing in the last two sections. These are the steps:

- 1. *MEM* is active-LOW, and it will go LOW only when X and Y are HIGH.
- 2. *X* will be HIGH only when RD = 0.
- 3. Y will be HIGH when either *W* or *V* is HIGH.
- 4. *V* will be HIGH when RAM = 0.
- 5. *W* will be HIGH when either *ROM-A* or *ROM-B* = 0.
- 6. Putting this all together, *MEM* will go LOW only when *RD* = 0 *and* at least one of the three inputs *ROM-A*, *ROM-B*, or *RAM* is LOW.

#### EXAMPLE 3-23

The logic circuit in Figure 3-40 is used to control the drive spindle motor for a floppy disk drive when the microcomputer is sending data to or receiving data from the disk. The circuit will turn on the motor when DRIVE = 1. Determine the input conditions necessary to turn on the motor.



#### Solution

Once again, we will interpret the diagram in a step-by-step fashion:

- 1. *DRIVE* is active-HIGH, and it will go HIGH only when X = Y = 0.
- 2. *X* will be LOW when either *IN* or *OUT* is HIGH.
- 3. Y will be LOW only when W = 0 and  $A_0 = 0$ .
- 4. *W* will be LOW only when  $A_1$  through  $A_7$  are all HIGH.
- 5. Putting this all together, *DRIVE* will be HIGH when  $A_1 = A_2 = A_3 = A_4 = A_5 = A_6 = A_7 = 1$  and  $A_0 = 0$ , and either *IN* or *OUT* or both are 1.

Note the strange symbol for the eight-input CMOS NAND gate (74HC30); also note that signal  $A_7$  is connected to two of the NAND inputs.

# Asserted Levels

We have been describing logic signals as being active-LOW or active-HIGH. For example, the output *MEM* in Figure 3-39 is active-LOW, and the output *DRIVE* in Figure 3-40 is active-HIGH because these are the output states that cause something to happen. Similarly, Figure 3-40 has active-HIGH inputs  $A_1$  to  $A_7$ , and active-LOW input  $A_0$ .

When a logic signal is in its active state, it can be said to be **asserted**. For example, when we say that input  $A_0$  is asserted, we are saying that it is in its active-LOW state. When a logic signal is not in its active state, it is said to be **unasserted**. Thus, when we say that *DRIVE* is unasserted, we mean that it is in its in its inactive state (low).

Clearly, the terms *asserted* and *unasserted* are synonymous with *active* and *inactive*, respectively:

asserted = active unasserted = inactive

Both sets of terms are in common use in the digital field, so you should recognize both ways of describing a logic signal's active state.

# Labeling Active-LOW Logic Signals

It has become common practice to use an overbar to label active-LOW signals. The overbar serves as another indication that the signal is active-LOW; of course, the absence of an overbar means that the signal is active-HIGH.

To illustrate, all of the signals in Figure 3-39 are active-LOW, and so they can be labeled as follows:

 $\overline{RD}$ ,  $\overline{ROM}$ - $\overline{A}$ ,  $\overline{ROM}$ - $\overline{B}$ ,  $\overline{RAM}$ ,  $\overline{MEM}$ 

Remember, the overbar is simply a way to emphasize that these are active-LOW signals. We will employ this convention for labeling logic signals whenever appropriate.

# **Labeling Bistate Signals**

Very often, an output signal will have two active states; that is, it will have one important function in the HIGH state and another in the LOW state. It is customary to label such signals so that both active states are apparent. A common example is the read/write signal, RD/WR, which is interpreted as follows: when this signal is HIGH, the read operation (RD) is performed; when it is LOW, the write operation (WR) is performed.

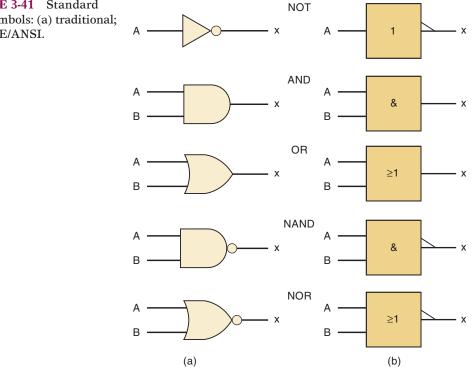
REVIEW QUESTIONS	1. Use the method of Examples 3-22 and 3-23 to determine the input con- ditions needed to activate the output of the circuit in Figure 3-37(b).
	2. Repeat question 1 for the circuit of Figure 3-38(b).
	3. How many NAND gates are shown in Figure 3-39?
	4 How many NOR gates are shown in Figure 3.402

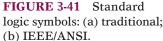
- 5. What will be the output level in Figure 3-38(b) when all of the inputs are asserted?
- 6. What inputs are required to assert the alarm output in Figure 3-37(b)?
- 7. Which of the following signals is active-LOW:  $RD, \overline{W}, R/\overline{W}$ ?

#### **IEEE/ANSI STANDARD LOGIC SYMBOLS** 3-15

The logic symbols we have used so far in this chapter are the *traditional* standard symbols used in the digital industry for many, many years. These traditional symbols use a distinctive shape for each logic gate. A newer standard for logic symbols was developed in 1984; it is called the IEEE/ANSI Standard 91-1984 for logic symbols. The IEEE/ANSI standard uses rectangular symbols to represent all logic gates and circuits. A special *dependency notation* inside the rectangular symbol indicates how the device outputs depend on the device inputs. Figure 3-41 shows the IEEE/ANSI symbols alongside the traditional symbols for the basic logic gates. Note the following points:

1. The rectangular symbols use a small right triangle  $(\square)$  in place of the small bubble of the traditional symbols to indicate the inversion of the logic level. The presence or absence of the triangle also signifies whether an input or output is active-LOW or active-HIGH.





- 2. A special notation inside each rectangular symbol describes the logic relation between inputs and output. The "1" inside the INVERTER symbol denotes a device with only *one* input; the triangle on the output indicates that the output will go to its active-LOW state when that one input is in its active-HIGH state. The "&" inside the AND symbol means that the output will go to its active-HIGH state when all of the inputs are in their active-HIGH state. The "≥" inside the OR gate means that the output will go to its active state (HIGH) whenever *one or more* inputs are in their active state (HIGH).
- 3. The rectangular symbols for the NAND and the NOR are the same as those for the AND and the OR, respectively, with the addition of the small inversion triangle on the output.

## **Traditional or IEEE/ANSI?**

The IEEE/ANSI standard has not yet been widely accepted for use in the digital field, although you will run across it in some newer equipment schematics. Most digital IC data books include both the traditional and IEEE/ANSI symbols, and it is possible that the newer standard might eventually become more widely used. We will employ the traditional symbols in most of the circuit diagrams throughout this book.

# **REVIEW QUESTIONS**

- 1. Draw all of the basic logic gates using both the traditional symbols and the IEEE/ANSI symbols.
- 2. Draw the IEEE/ANSI symbol for a NOR gate with active-HIGH output.

# 3-16 SUMMARY OF METHODS TO DESCRIBE LOGIC CIRCUITS

The topics we have covered so far in this chapter have all centered around just three simple logic functions that we refer to as AND, OR, and NOT. The concept is not new to anyone because we all use these logical functions every day as we make decisions. Here are some logical examples. If it is raining OR the newspaper says that it could rain, then I will take my umbrella. If I get my paycheck today AND I make it to the bank, then I will have money to spend this evening. If I have a passing grade in lecture AND I have NOT failed in lab, then I will pass my digital class. At this point, you may be wondering why we have spent so much effort in describing such familiar concepts. The answer can be summed up in two key points:

- 1. We must be able to represent these logical decisions.
- 2. We must be able to combine these logic functions and implement a decisionmaking system.

We have learned how to represent each of the basic logic functions using:

Logical statements in our own language Truth tables Traditional graphic logic symbols IEEE/ANSI standard logic symbols Boolean algebra expressions Timing diagrams

## **EXAMPLE 3-24**

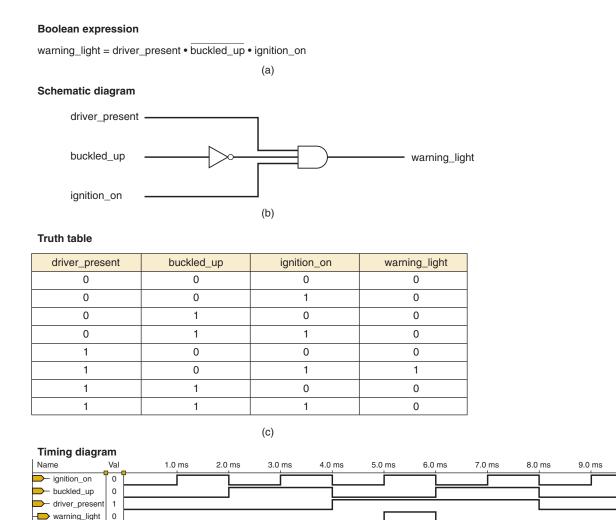
The following English expression describes the way a logic circuit needs to operate in order to drive a seatbelt warning indicator in a car.

# If the driver is present AND the driver is NOT buckled up AND the ignition switch is on, THEN turn on the warning light.

Describe the circuit using Boolean algebra, schematic diagrams with logic symbols, truth tables, and timing diagrams.

## Solution

See Figure 3-42.



**FIGURE 3-42** Methods of describing logic circuits: (a) Boolean expression; (b) schematic diagram; (c) truth table; (d) timing diagram.

(d)

10 ms

Figure 3-42 shows four different ways of representing the logic circuit that was described in English as the problem statement of Example 3-24. There are many other ways in which we could represent the logic of this decision. As an example we could dream up an entirely new set of graphic symbols, or state the logical relationship in French or Japanese. Of course, we cannot cover all the possible ways of describing a logic circuit, but we must understand the most common methods to be able to communicate with others in this profession. Furthermore, certain situations are easier to describe using one method over another. In some cases, a picture is worth a thousand words, and in other cases words are concise enough and are more easily communicated to others. The important point here is that we need ways to describe and communicate the operation of digital systems.

## **REVIEW QUESTION**

1. Name five ways to describe the operation of logic circuits.

# 3-17 DESCRIPTION LANGUAGES VERSUS PROGRAMMING LANGUAGES\*

Recent trends in the field of digital systems are favoring text-based language description of digital circuits. You probably noticed that each description method in Figure 3-42 offers challenges to computer entry, whether it is due to overbars, symbols, formatting, or line-drawing issues. In this section, we will begin to learn some of the more advanced tools that professionals in the digital field use to describe the circuits that implement their ideas. These tools are referred to as **hardware description languages (HDLs)**. Even with the powerful computers we have today, it is not possible to describe a logic circuit in English prose and expect the computer to understand what you mean. Computers need a more rigidly defined language. We will focus on two languages in this text: **Altera hardware description language (AHDL)** and **very high speed integrated circuit (VHSIC) hardware description language (VHDL)**.

## **VHDL and AHDL**

VHDL is not a new language. It was developed by the Department of Defense in the early 1980s as a concise way to document the designs in the very high speed integrated circuit (VHSIC) program. Appending HDL onto this acronym was too much, even for the military, and so the language was abbreviated to VHDL. Computer programs were developed to take the VHDL language files and simulate the operation of the circuits. With the growth of complex programmable logic devices in digital systems, VHDL has evolved into one of the primary high-level hardware description languages for designing and implementing digital circuits (synthesis). The language has been standardized by the IEEE, making it universally appealing for engineers as well as the makers of software tools that translate designs into the bit patterns used to program actual devices.

AHDL is a language that the Altera Corporation developed to provide a convenient way to configure the logic devices that they offer. Altera was one of the first companies to introduce logic devices that can be reconfigured

<sup>\*</sup>All sections covering hardware description languages may by skipped without loss of continuity in the balance of Chapters 1–12.

electronically. These devices are called **programmable logic devices (PLDs**). Unlike VHDL, this language is not intended to be used as a universal language for describing any logic circuit. It is intended to be used for programming complex digital systems into Altera PLDs in a language that is generally perceived to be easier to learn yet very similar to VHDL. It also has features that take full advantage of the architecture of Altera devices. All of the examples in this text will use the Altera MAX+PLUS II or Quartus II software to develop both AHDL and VHDL design files. You will see the advantage of using Altera's development system for both languages when you program an actual device. The Altera system makes circuit development very easy and contains all the necessary tools to translate from the HDL design file to a file ready to load into an Altera PLD. It also allows you to develop building blocks using schematic entry, AHDL, VHDL, and other methods and then interconnect them to form a complete system.

Other HDLs are available that are more suitable for programming simple programmable logic devices. You will find any of these languages easy to use after learning the basics of AHDL or VHDL as covered in this text.

# **Computer Programming Languages**

It is important to distinguish between hardware description languages intended to describe the hardware configuration of a circuit and programming languages that represent a sequence of instructions intended to be carried out by a computer to accomplish some task. In both cases, we use a *language* to *program* a device. However, computers are complex digital systems that are made up of logic circuits. Computers operate by following a laundry list of tasks (i.e., instructions, or "the program"), each of which must be done in sequential order. The speed of operation is determined by how fast the computer can execute each instruction. For example, if a computer were to respond to four different inputs, it would require at least four separate instructions (sequential tasks) to detect and identify which input changed state. A digital logic circuit, on the other hand, is limited in its speed only by how quickly the circuitry can change the outputs in response to changes in the inputs. It is monitoring all inputs **concurrently** (at the same time) and responding to any changes.

The following analogy will help you understand the difference between computer operation and digital logic circuit operation and the role of language elements used to describe what the systems do. Consider the challenge of describing what is done to an Indy 500 car during a pit stop. If a single person performed all the necessary tasks one at a time, he or she would need to be very fast at each task. This is the way a computer operates: one task at a time but very quickly. Of course, at Indy, there is an entire pit crew that swarms the car, and each member of the crew does his or her task while the others do theirs. All crew members operate concurrently, like the elements of a digital circuit. Now consider how you would describe to someone else what is being done to the Indy car during the pit stop using (1) the individualmechanic approach or (2) the pit-crew approach. Wouldn't the two English language descriptions of what is being done sound very similar? As we will see, the languages used to describe digital hardware (HDL) are very similar to languages that describe computer programs (e.g., BASIC, C, JAVA), even though the resulting implementation operates quite differently. Knowledge of any of these computer programming languages is not necessary to understand HDL. The important thing is that when you have learned both an HDL and a computer language, you must understand their different roles in digital systems.

# EXAMPLE 3-25

**FIGURE 3-43** Decision process of a computer

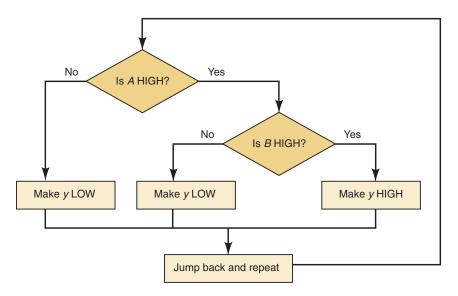
program.

Compare the operation of a computer and a logic circuit in performing the simple logical operation of y = AB.

# Solution

The logic circuit is a simple AND gate. The output y will be HIGH within approximately 10 nanoseconds of the point when A and B are HIGH simultaneously. Within approximately 10 nanoseconds after either input goes LOW, the output y will be LOW.

The computer must run a program of instructions that makes decisions. Suppose each instruction takes 20 ns (that's pretty fast!). Each shape in the flowchart shown in Figure 3-43 represents one instruction. Clearly, it will take a minimum of two or three instructions (40–60 ns) to respond to changes in the inputs.



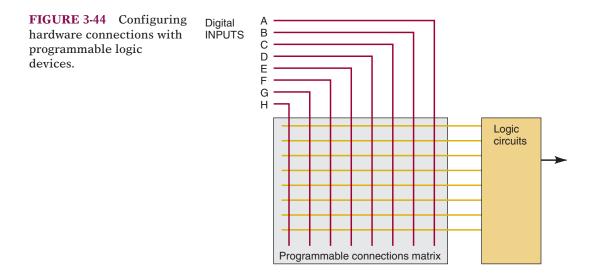
# **REVIEW QUESTIONS**

#### 1. What does HDL stand for?

- 2. What is the purpose of an HDL?
- 3. What is the purpose of a computer programming language?
- 4. What is the key difference between HDL and computer programming languages?

# 3-18 IMPLEMENTING LOGIC CIRCUITS WITH PLDs

Many digital circuits today are implemented using programmable logic devices (PLDs). These devices are not like microcomputers or microcontrollers that "run" the program of instructions. Instead, they are configured electronically, and their internal circuits are "wired" together electronically to form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). Figure 3-44 shows a small area of programmable connections. Each intersection between a row (horizontal wire) and a column (vertical wire) is a programmable connection. You can imagine how difficult it would be to try to



configure these devices by placing 1s and 0s in a grid manually (which is how they did it back in the 1970s).

The role of the hardware description language is to provide a concise and convenient way for the designer to describe the operation of the circuit in a format that a personal computer can handle and store conveniently. The computer runs a special software application called a **compiler** to translate from the hardware description language into the grid of 1s and 0s that can be loaded into the PLD. If a person can master the higher-level hardware description language, it actually makes programming the PLDs much easier than trying to use Boolean algebra, schematic drawings, or truth tables. In much the same way that you learned the English language, we will start by expressing simple things and gradually learn the more complicated aspects of these languages. Our objective is to learn enough of HDL to be able to communicate with others and perform simple tasks. A full understanding of all the details of these languages is beyond the scope of this text and can really be mastered only by regular use.

In the sections throughout this book that cover the HDLs, we will present both AHDL and VHDL in a format that allows you to skip over one language and concentrate on the other without missing important information. Of course, this setup means there will be some redundant information presented if you choose to read about both languages. We feel this redundancy is worth the extra effort to provide you with the flexibility of focusing on either of the two languages or learning both by comparing and contrasting similar examples. The recommended way to use the text is to focus on one language. It is true that the easiest way to become bilingual, and fluent in both languages, is to be raised in an environment where both languages are spoken routinely. It is also very easy, however, to confuse details, so we will keep the specific examples separate and independent. We hope this format provides you with the opportunity to learn one language now and then use this book as a reference later in your career should you need to pick up the second language.

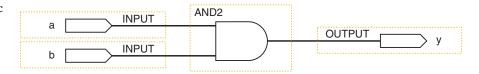
## **REVIEW QUESTIONS**

- 1. What does PLD stand for?
- 2. How are the circuits reconfigured electronically in a PLD?
- 3. What does a compiler do?

# 3-19 HDL FORMAT AND SYNTAX

Any language has its unique properties, similarities to other languages, and its proper syntax. When we study grammar in school, we learn conventions such as the order of words as elements in a sentence and proper punctuation. This is referred to as the **syntax** of language. A language designed to be interpreted by a computer must follow strict rules of syntax. A computer is just an assortment of processed beach sand and wire that has no idea what you "meant" to say, so you must present the instructions using the exact syntax that the computer language expects and understands. The basic format of any hardware circuit description (in any language) involves two vital elements:

- 1. The definition of what goes into it and what comes out of it (i.e., input/output specs)
- 2. The definition of how the outputs respond to the inputs (i.e., its operation)

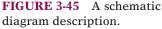


A circuit schematic diagram such as Figure 3-45 can be read and understood by a competent engineer or technician because both would understand the meaning of each symbol in the drawing. If you understand how each element works and how the elements are connected to each other, you can understand how the circuit operates. On the left side of the diagram is the set of inputs, and on the right is the set of outputs. The symbols in the middle define its operation. The text-based language must convey the same information. All HDLs use the format shown in Figure 3-46.

FIGURE 3-46	Format of
HDL files.	

Documentation	
I/O definitions	
Functional description	

In a text-based language, the circuit being described must be given a name. The inputs and outputs (sometimes called ports) must be assigned names and defined according to the nature of the port. Is it a single bit from a toggle switch? Or is it a four-bit number coming from a keypad? The text-based language must somehow convey the nature of these inputs and outputs. The **mode** of a port defines whether it is input, output, or both. The **type** refers to the number of bits and how those bits are grouped and interpreted. If the *type* of input is a single bit, then it can have only two possible values: 0 and 1. If the type of input is a four-bit binary number from a keypad, it can have any one of 16 different values  $(0000_2-1111_2)$ . The type determines the range of possible values. The definition of the circuit's operation in a



text-based language is contained in a set of statements that follow the circuit input/output (I/O) definition. The following two sections describe the very simple circuit of Figure 3-45 and illustrate the critical elements of AHDL and VHDL.

# BOOLEAN DESCRIPTION USING AHDL

Refer to Figure 3-47. The keyword **SUBDESIGN** gives a name to the circuit block, which in this case is *and\_gate*. The name of the file must also be and\_gate.tdf. Notice that the keyword SUBDESIGN is capitalized. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide that is provided with the Altera compiler for AHDL suggests the use of capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

**FIGURE 3-47** Essential elements in AHDL.

SUBDESIGN an	nd_gate	
a, b	: INPUT;	
У	:OUTPUT;	
)		
BEGIN		
y = a &	b;	
END;		

The SUBDESIGN section defines the inputs and outputs of the logic circuit block. Something must enclose the circuit that we are trying to describe, much the same way that a block diagram encloses everything that makes up that part of the design. In AHDL, this input/output definition is enclosed in parentheses. The list of variables used for inputs to this block are separated by commas and followed by :INPUT;. In AHDL, the single-bit type is assumed unless the variable is designated as multiple bits. The single-output bit is declared with the mode :OUTPUT;. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them.

The set of statements that describe the operation of the AHDL circuit are contained in the logic section between the keywords BEGIN and END. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output (y) is assigned (=) the logic level produced by *a* AND *b*. This Boolean algebra equation is referred to as a **concurrent assignment statement**. Any statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference. The basic Boolean operators are:

&	AND
#	OR
!	NOT
\$	XOR

1. What appears inside the parentheses () after SUBDESIGN?

2. What appears between BEGIN and END?

**REVIEW QUESTIONS** 

AHDL

# **BOOLEAN DESCRIPTION USING VHDL**

Refer to Figure 3-48. The keyword **ENTITY** gives a name to the circuit block, which in this case is and\_gate. Notice that the keyword ENTITY is capitalized but and\_gate is not. This is not required by the software, but use of a consistent style in capitalization makes the code much easier to read. The style guide provided with the Altera compiler for VHDL suggests using capital letters for the keywords in the language. Variables that are named by the designer should be lowercase.

FIGURE 3-48 Essential elements in VHDL.

The ENTITY declaration can be thought of as a block description. Something must enclose the circuit we are trying to describe, much the same way a block diagram encloses everything that makes up that part of the design. In VHDL, the keyword PORT tells the compiler that we are defining inputs and outputs to this circuit block. The names used for inputs (separated by commas) are listed, ending with a colon and a description of the mode and type of input (:IN BIT;). In VHDL, the **BIT** description tells the compiler that each variable in the list is a single bit. We will learn the proper way to describe other types of inputs, outputs, and variables as we need to use them. The line containing END and gate; terminates the ENTITY declaration.

The **ARCHITECTURE** declaration is used to describe the operation of everything inside the block. The designer makes up a name for this architectural description of the inner workings of the ENTITY block (*ckt* in this example). Every ENTITY must have at least one ARCHITECTURE associated with it. The words OF and IS are keywords in this declaration. The body of the architecture description is enclosed between the BEGIN and END keywords. END is followed by the name that has been assigned to this architecture. Within the body (between BEGIN and END) is the description of the block's operation. In this example, the operation of the hardware is described by a very simple Boolean algebra equation that states that the output (*y*) is assigned (<=) the logic level produced by *a* AND *b*. This is referred to as a **concurrent assignment statement**, which means that all the statements (there is only one in this example) between BEGIN and END are evaluated constantly and concurrently. The order in which they are listed makes no difference.

### **REVIEW QUESTIONS**

- 1. What is the role of the ENTITY declaration?
- 2. Which key section defines the operation of the circuit?
- 3. What is the assignment operator used to give a value to a logic signal?

# 3-20 INTERMEDIATE SIGNALS

In many designs, there is a need to define signal points "inside" the circuit block. They are points in the circuit that are neither inputs nor outputs for the block but may be useful as a reference point. It may be a signal that needs to be connected to many other places within the block. In an analog or digital schematic diagram, they would be called test points or *nodes*. In an HDL, they are referred to as **buried nodes** or **local signals**. Figure 3-49 shows a very simple circuit that uses an intermediate signal named *m*. In the HDL, these nodes (signals) are not defined with the inputs and outputs but rather in the section that describes the operation of the block. The inputs and outputs are available to other circuit blocks in the system, but these local signals are recognized only within this block.

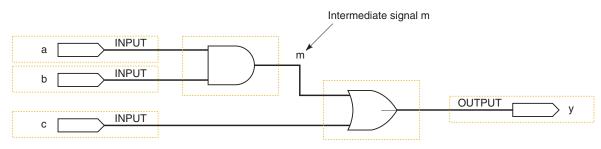


FIGURE 3-49 A logic circuit diagram with an intermediate variable.

In the example code that follows, notice the information at the top. The purpose of this information is strictly for documentation purposes. It is absolutely vital that the design is documented thoroughly. At a minimum, it should describe the project it is being used in, who wrote it, and the date. This information is often referred to as a header. We are keeping our headers brief to make this book a little lighter to carry to class, but remember: memory space is cheap and information is valuable. So don't be afraid to *document thoroughly!* There are also comments next to many of the statements in the code. These comments help the designer remember what she or he was trying to do and to help any other person to understand what was intended.

# **AHDL BURIED NODES**

The AHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-50. The **comments** in AHDL can be enclosed between % characters, as you can see in the figure between lines 1 and 4. This section of the code allows the designer to write many lines of information that will be ignored by computer programs using this file but can be read by any person trying to decipher the code. Notice that the comments at the end of lines 9, 10, 13, 15, and 16 are preceded by two dashes (--). The text following the dashes is for documentation only. Either type of comment symbol may be used, but percent signs must be used in pairs to open and close a comment. Double dashes indicate a comment that extends to the end of the line.

In AHDL, local signals are declared in the VARIABLE section, which is placed between the SUBDESIGN section and the logic section. The intermediate signal *m* is defined on line 11, following the keyword **VARIABLE**. The

FIGURE 3-50 Intermediate variables

in AHDL described in Figure 3-49.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

```
Intermediate variables in AHDL (Figure 3-49)
8
   Digital Systems 10th ed
   NS Widmer
   MAY 23, 2005
                         8
SUBDESIGN fig3_50
(
             : INPUT;
                        -- define inputs to block
   a,b,c
             :OUTPUT;
                       -- define block output
   V
)
VARIABLE
   m
           :NODE;
                        -- name an intermediate signal
BEGIN
   m = a \& b;
                        -- generate buried product term
   y = m \# c;
                        -- generate sum on output
END;
```

keyword **NODE** designates the nature of the variable. Notice that a colon separates the variable name from its node designation. In the hardware description on line 13, the intermediate variable is assigned (connected to) a value (m = a & b;) and then m is used in the second statement on line 14 to assign (connect) a value to y (y = m # c;). Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate variables before they are used in other assignment statements, as shown here.

# **REVIEW QUESTIONS**

- 1. What is the designation used for intermediate variables?
- 2. Where are these variables declared?
- 3. Does it matter whether the *m* or *y* equation comes first?
- 4. What character is used to limit a block of comments?
- 5. What characters are used to comment a single line?

# **VHDL LOCAL SIGNALS**

The VHDL code that describes the circuit in Figure 3-49 is shown in Figure 3-51. The **comments** in VHDL follow two dashes (--). Typing two successive dashes allows the designer to write information from that point to the end of the line. The information following the two successive dashes will be ignored by computer programs using this file, but can be read by any person trying to decipher the code.

The intermediate signal m is defined on line 13 following the keyword SIGNAL. The keyword BIT designates the type of the signal. Notice that a colon separates the signal name from its type designation. In the hardware description on line 16, the intermediate signal is assigned (connected to) a value

```
1
       -- Intermediate variables in VHDL (Figure 3-49)
2
       -- Digital Systems 10th ed
       -- NS Widmer
3
       -- MAY 23, 2005
4
5
6
       ENTITY fig3_51 IS
7
       PORT(a, b, c :IN BIT;
                                   -- define inputs to block
                       :OUT BIT); -- define block output
8
       У
9
       END fig3_51;
10
      ARCHITECTURE ckt OF fig3_51 IS
11
12
13
          SIGNAL m
                       :BIT;
                                    -- name an intermediate signal
14
15
       BEGIN
            m \ll a AND b;
                                    -- generate buried product term
16
17
            y <= m OR c;
                                    -- generate sum on output
18
       END ckt;
```

FIGURE 3-51 Intermediate signals in VHDL described in Figure 3-49.

 $(m \le a \text{ AND } b;)$  and then m is used in the statement on line 17 to assign (connect) a value to y ( $y \le m \text{ OR } c;$ ). Remember that the assignment statements are concurrent and, thus, the order in which they are given does not matter. For human readability, it may seem more logical to assign values to intermediate signals before they are used in other assignment statements, as shown here.

# **REVIEW QUESTIONS**

- 1. What is the designation used for intermediate signals?
- 2. Where are these signals declared?
- 3. Does it matter whether the m or y equation comes first?
- 4. What characters are used to comment a single line?

# **SUMMARY**

- 1. Boolean algebra is a mathematical tool used in the analysis and design of digital circuits.
- 2. The basic Boolean operations are the OR, AND, and NOT operations.
- 3. An OR gate produces a HIGH output when any input is HIGH. An AND gate produces a HIGH output only when all inputs are HIGH. A NOT circuit (INVERTER) produces an output that is the opposite logic level compared to the input.
- 4. A NOR gate is the same as an OR gate with its output connected to an IN-VERTER. A NAND gate is the same as an AND gate with its output connected to an INVERTER.

- 5. Boolean theorems and rules can be used to simplify the expression of a logic circuit and can lead to a simpler way of implementing the circuit.
- 6. NAND gates can be used to implement any of the basic Boolean operations. NOR gates can be used likewise.
- 7. Either standard or alternate symbols can be used for each logic gate, depending on whether the output is to be active-HIGH or active-LOW.
- 8. The IEEE/ANSI standard for logic symbols uses rectangular symbols for each logic device, with special notations inside the rectangles to show how the outputs depend on the inputs.
- 9. Hardware description languages have become an important method of describing digital circuits.
- 10. HDL code should always contain comments that document its vital characteristics so a person reading it later can understand what it does.
- 11. Every HDL circuit description contains a definition of the inputs and outputs, followed by a section that describes the circuit's operation.
- 12. In addition to inputs and outputs, intermediate connections that are buried within the circuit can be defined. These intermediate connections are called nodes or signals.

# **IMPORTANT TERMS**

active logic levels concurrent logic level Boolean algebra active-HIGH compiler truth table active-LOW syntax **OR** operation asserted mode OR gate unasserted type AND operation **IEEE/ANSI** SUBDESIGN AND gate hardware description concurrent NOT operation languages (HDLs) assignment inversion Altera hardware statement (complementation) description ENTITY NOT circuit language (AHDL) BIT ARCHITECTURE (INVERTER) very high speed NOR gate integrated circuit buried nodes (local NAND gate (VHSIC) hardware signals) Boolean theorems description comments VARIABLE DeMorgan's theorems language (VHDL) alternate logic programmable logic NODE symbols devices (PLDs)

# PROBLEMS

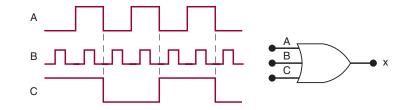
The color letters preceding some of the problems are used to indicate the nature or type of problem as follows:

- **B** basic problem
- T troubleshooting problem
- **D** design or circuit-modification problem
- N new concept or technique not covered in text
- C challenging problem
- H HDL problem

# **SECTION 3-3**

**B** 3-1\* Draw the output waveform for the OR gate of Figure 3-52.

#### **FIGURE 3-52**



- **B** 3-2. Suppose that the *A* input in Figure 3-52 is unintentionally shorted to ground (i.e., A = 0). Draw the resulting output waveform.
- **B** 3-3.\*Suppose that the A input in Figure 3-52 is unintentionally shorted to the +5 V supply line (i.e., A = 1). Draw the resulting output waveform.
- **C** 3-4. Read the statements below concerning an OR gate. At first, they may appear to be valid, but after some thought you should realize that neither one is *always* true. Prove this by showing a specific example to refute each statement.
  - (a) If the output waveform from an OR gate is the same as the waveform at one of its inputs, the other input is being held permanently LOW.
  - (b) If the output waveform from an OR gate is always HIGH, one of its inputs is being held permanently HIGH.
- **B** 3-5. How many different sets of input conditions will produce a HIGH output from a five-input OR gate?

#### **SECTION 3-4**

- **B** 3-6. Change the OR gate in Figure 3-52 to an AND gate.
  - (a)\*Draw the output waveform.
  - (b) Draw the output waveform if the *A* input is permanently shorted to ground.
  - (c) Draw the output waveform if A is permanently shorted to +5 V.
- D 3-7.\*Refer to Figure 3-4. Modify the circuit so that the alarm is to be activated only when the pressure and the temperature exceed their maximum limits at the same time.
- **B** 3-8.\*Change the OR gate in Figure 3-6 to an AND gate and draw the output waveform.
- **B** 3-9. Suppose that you have an unknown two-input gate that is either an OR gate or an AND gate. What combination of input levels should you apply to the gate's inputs to determine which type of gate it is?
- **B** 3-10. *True or false:* No matter how many inputs it has, an AND gate will produce a HIGH output for only one combination of input levels.

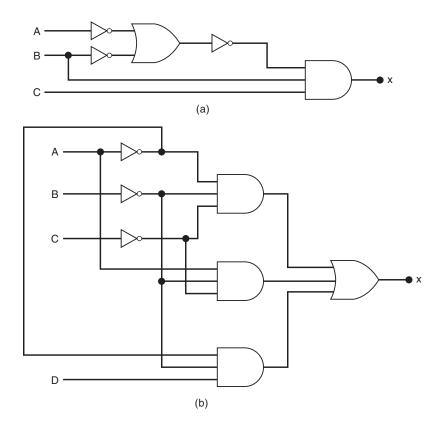
<sup>\*</sup>Answers to problems marked with an asterisk can be found in the back of the text.

## **SECTIONS 3-5 TO 3-7**

В

- **B** 3-11. Apply the *A* waveform from Figure 3-23 to the input of an INVERTER. Draw the output waveform. Repeat for waveform *B*.
  - 3-12. (a)\* Write the Boolean expression for output x in Figure 3-53(a). Determine the value of x for all possible input conditions, and list the values in a truth table.
    - (b) Repeat for the circuit in Figure 3-53(b).

#### FIGURE 3-53



- **B** 3-13.\*Create a complete analysis table for the circuit of Figure 3-15(b) by finding the logic levels present at each gate output for each of the 32 possible input combinations.
- B 3-14. (a)\*Change each OR to an AND, and each AND to an OR, in Figure 3-15(b). Then write the expression for the output.
  - (b) Complete an analysis table.
- B 3-15. Create a complete analysis table for the circuit of Figure 3-16 by finding the logic levels present at each gate output for each of the 16 possible combinations of input levels.

#### **SECTION 3-8**

**B** 3-16. For each of the following expressions, construct the corresponding logic circuit, using AND and OR gates and INVERTERs.

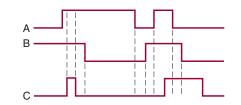
(a)\*
$$x = AB(C + D)$$
  
(b)\* $z = \overline{A + B + \overline{C}D\overline{E}}$ ) +  $\overline{B}C\overline{D}$   
(c)  $y = (\overline{M + N} + \overline{P}Q)$ 

(d)  $x = \overline{W + P\overline{Q}}$ (e)  $z = MN(P + \overline{N})$ (f)  $x = (A + B)(\overline{A} + \overline{B})$ 

# **SECTION 3-9**

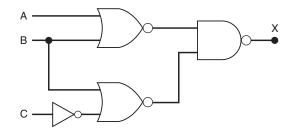
- **B** 3-17<sup>\*</sup>(a) Apply the input waveforms of Figure 3-54 to a NOR gate, and draw the output waveform.
  - (b) Repeat with C held permanently LOW.
  - (c) Repeat with *C* held HIGH.

# FIGURE 3-54



- **B** 3-18. Repeat Problem 3-17 for a NAND gate.
- C 3-19.\*Write the expression for the output of Figure 3-55, and use it to determine the complete truth table. Then apply the waveforms of Figure 3-54 to the circuit inputs, and draw the resulting output waveform.

FIGURE 3-55



- **B** 3-20. Determine the truth table for the circuit of Figure 3-24.
- **B** 3-21. Modify the circuits that were constructed in Problem 3-16 so that NAND gates and NOR gates are used wherever appropriate.

## **SECTION 3-10**

- **C** 3-22. Prove theorems (15a) and (15b) by trying all possible cases.
- **B** 3-23\* DRILL QUESTION

Complete each expression.

 (a) A + 1 =\_\_\_\_\_
 (f)  $D \cdot 1 =$ \_\_\_\_\_

 (b)  $A \cdot A =$ \_\_\_\_\_
 (g) D + 0 =\_\_\_\_\_

 (c)  $B \cdot \overline{B} =$ \_\_\_\_\_
 (h)  $C + \overline{C} =$ \_\_\_\_\_

 (d) C + C =\_\_\_\_\_
 (i) G + GF =\_\_\_\_\_

 (e)  $x \cdot 0 =$ \_\_\_\_\_
 (j)  $y + \overline{w}y =$ \_\_\_\_\_

**C** 3-24. (a)\* Simplify the following expression using theorems (13b), (3), and (4):

$$x = (M + N)(\overline{M} + P)(\overline{N} + \overline{P})$$

(b) Simplify the following expression using theorems (13a), (8), and (6):

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

## **SECTIONS 3-11 AND 3-12**

- **C** 3-25. Prove DeMorgan's theorems by trying all possible cases.
- **B** 3-26. Simplify each of the following expressions using DeMorgan's theorems.

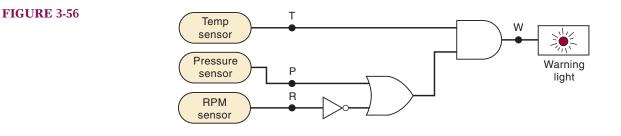
$$\begin{array}{lll} (a)^{\star}\overline{\overline{A}B\overline{C}} & (d) \ \overline{A} + \overline{B} & (g)^{\star}\overline{A(\overline{B} + \overline{C})}D \\ (b) \ \overline{\overline{A} + \overline{B}C} & (e)^{\star}\overline{\overline{AB}} & (h) \ \overline{(M + \overline{N})}(\overline{M} + N) \\ (c)^{\star}\overline{AB\overline{CD}} & (f) \ \overline{\overline{A} + \overline{C} + \overline{D}} & (i) \ \overline{\overline{\overline{ABCD}}} \end{array}$$

- **B** 3-27.\*Use DeMorgan's theorems to simplify the expression for the output of Figure 3-55.
- C 3-28. Convert the circuit of Figure 3-53(b) to one using only NAND gates. Then write the output expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.
- C 3-29. Convert the circuit of Figure 3-53(a) to one using only NOR gates. Then write the expression for the new circuit, simplify it using DeMorgan's theorems, and compare it with the expression for the original circuit.
- **B** 3-30. Show how a two-input NAND gate can be constructed from two-input NOR gates.
- **B** 3-31. Show how a two-input NOR gate can be constructed from two-input NAND gates.
- **C** 3-32. A jet aircraft employs a system for monitoring the rpm, pressure, and temperature values of its engines using sensors that operate as follows:

RPM sensor output = 0 only when speed < 4800 rpm P sensor output = 0 only when pressure < 220 psi T sensor output = 0 only when temperature < 200°F

Figure 3-56 shows the logic circuit that controls a cockpit warning light for certain combinations of engine conditions. Assume that a HIGH at output *W* activates the warning light.

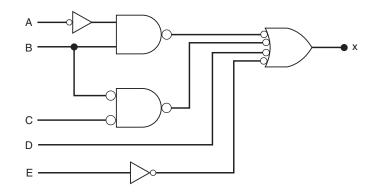
- (a)\*Determine what engine conditions will give a warning to the pilot.
- (b) Change this circuit to one using all NAND gates.



**FIGURE 3-57** 

# SECTIONS 3-13 AND 3-14

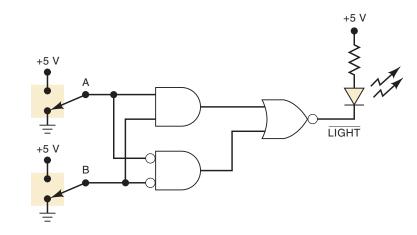
- **B** 3-33. For each statement below, draw the appropriate logic-gate symbol—standard or alternate—for the given operation.
  - (a) A HIGH output occurs only when all three inputs are LOW.
  - (b) A LOW output occurs when any of the four inputs is LOW.
  - (c) A LOW output occurs only when all eight inputs are HIGH.
- **B** 3-34. Draw the standard representations for each of the basic logic gates. Then draw the alternate representations.
- **C** 3-35. The circuit of Figure 3-55 is supposed to be a simple digital combination lock whose output will generate an active-LOW *UNLOCK* signal for only one combination of inputs.
  - (a)<sup>\*</sup>Modify the circuit diagram so that it represents more effectively the circuit operation.
  - (b) Use the new circuit diagram to determine the input combination that will activate the output. Do this by working back from the output using the information given by the gate symbols, as was done in Examples 3-22 and 3-23. Compare the results with the truth table obtained in Problem 3-19.
- C 3-36. (a) Determine the input conditions needed to activate output Z in Figure 3-37(b). Do this by working back from the output, as was done in Examples 3-22 and 3-23.
  - (b) Assume that it is the LOW state of Z that is to activate the alarm. Change the circuit diagram to reflect this, and then use the revised diagram to determine the input conditions needed to activate the alarm.
- **D** 3-37. Modify the circuit of Figure 3-40 so that  $A_1 = 0$  is needed to produce DRIVE = 1 instead of  $A_1 = 1$ .
- B 3-38\* Determine the input conditions needed to cause the output in Figure 3-57 to go to its active state.



- **B** 3-39.\* What is the asserted state for the output of Figure 3-57? For the output of Figure 3-36(c)?
- **B** 3-40. Use the results of Problem 3-38 to obtain the complete truth table for the circuit of Figure 3-57.
- **N** 3-41.\*Figure 3-58 shows an application of logic gates that simulates a twoway switch like the ones used in our homes to turn a light on or off



from two different switches. Here the light is an LED that will be ON (conducting) when the NOR gate output is LOW. Note that this output is labeled  $\overline{LIGHT}$  to indicate that it is active-LOW. Determine the input conditions needed to turn on the LED. Then verify that the circuit operates as a two-way switch using switches A and B. (In Chapter 4, you will learn how to design circuits like this one to produce a given relationship between inputs and outputs.)



#### **SECTION 3-15**

**B** 3-42. Redraw the circuits of (a)\* Figure 3-57 and (b) Figure 3-58 using the IEEE/ANSI symbols.

# SECTION 3-17 HDL DRILL QUESTIONS

- **H** 3-43*\* True or false:* 
  - (a) VHDL is a computer programming language.
  - (b) VHDL can accomplish the same thing as AHDL.
  - (c) AHDL is an IEEE standard language.
  - (d) Each intersection in a switch matrix can be programmed as an open or short circuit between a row and column wire.
  - (e) The first item that appears at the top of an HDL listing is the functional description.
  - (f) The type of an object indicates if it is an input or an output.
  - (g) The mode of an object determines if it is an input or an output.
  - (h) Buried nodes are nodes that have been deleted and will never be used again.
  - (i) Local signals are another name for intermediate variables.
  - (j) The header is a block of comments that document vital information about the project.

## **SECTION 3-18**

**B** 3-44. Redraw the programmable connection matrix from Figure 3-44. Label the output signals (horizontal lines) from the connection matrix (from

## **FIGURE 3-58**

top row to bottom row) as follows: AAABADHE. Draw an X in the appropriate intersections to short-circuit a row to a column and create these connections to the logic circuit.

**H** 3-45.\*Write the HDL code in the language of your choice that will produce the following output functions:

$$X = A + B$$
$$Y = AB$$
$$Z = A + B + C$$

- H 3-46. Write the HDL code in the language of your choice that will implement the logic circuit of Figure 3-39.
  - (a) Use a single Boolean equation.
  - (b) Use the intermediate variables V, W, X, and Y.

## **MICROCOMPUTER APPLICATION**

**C** 3-47.\*Refer to Figure 3-40 in Example 3-23. Inputs  $A_7$  through  $A_0$  are *address* inputs that are supplied to this circuit from outputs of the microprocessor chip in a microcomputer. The eight-bit address code  $A_7$  to  $A_0$  selects which device the microprocessor wants to activate. In Example 3-23, the required address code to activate the disk drive was  $A_7$  through  $A_0 = 11111110_2 = FE_{16}$ .

Modify the circuit so that the microprocessor must supply an address code of  $4A_{16}$  to activate the disk drive.

#### **CHALLENGING EXERCISES**

- C 3-48. Show how x = ABC can be implemented with one two-input NOR and one two-input NAND gate.
- **C** 3-49.\*Implement y = ABCD using only two-input NAND gates.

# ANSWERS TO SECTION REVIEW QUESTIONS

## **SECTION 3-2**

1. x = 1 2. x = 0 3. 32

#### **SECTION 3-3**

1. All inputs LOW 2. x = A + B + C + D + E + F 3. Constant HIGH

## **SECTION 3-4**

1. All five inputs = 1 2. A LOW input will keep the output LOW. 3. False; see truth table of each gate.

#### **SECTION 3-5**

1. Output of second INVERTER will be the same as input A. 2. y will be LOW only for A = B = 1.

#### **SECTION 3-6**

1.  $x = \overline{A} + B + C + \overline{AD}$  2.  $x = D(\overline{AB + C}) + E$ 

## **SECTION 3-7**

1. x = 1 2. x = 1 3. x = 1 for both.

#### SECTION 3-8

1. See Figure 3-15(a). 2. See Figure 3-17(b). 3. See Figure 3-15(b).

#### **SECTION 3-9**

1. All inputs LOW. 2. x = 0 3.  $x = A + B + \overline{CD}$ 

#### **SECTION 3-10**

1.  $y = A\overline{C}$  2.  $y = \overline{A}\overline{B}\overline{D}$  3.  $y = \overline{A}D + BD$ 

#### **SECTION 3-11**

1.  $z = \overline{AB} + C$  2.  $y = (\overline{R} + S + \overline{T})Q$  3. Same as Figure 3-28 except NAND is replaced by NOR. 4.  $y = \overline{AB}(C + \overline{D})$ 

#### SECTION 3-12

1. Three. 2. NOR circuit is more efficient because it can be implemented with one 74LS02 IC. 3.  $x = (\overline{AB})(\overline{CD}) = \overline{AB} + \overline{(CD)} + AB + CD$ 

#### **SECTION 3-13**

1. Output goes LOW when any input is HIGH. 2. Output goes HIGH only when all inputs are LOW. 3. Output goes LOW when any input is LOW. 4. Output goes HIGH only when all inputs are HIGH.

#### SECTION 3-14

1. Z will go HIGH when A = B = 0 and C = D = 1. A = B = 0, E = 1, and either C or D or both are 0. A = B = 0, E = 1, and either C or D or both are 0. A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = B = 0, C = D = 1 A = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0 A = 0, C = 0, C = 0, C = 0A = 0, C = 0,

#### **SECTION 3-15**

1. See Figure 3-41. 2. Rectangle with & inside, and triangles on inputs.

#### SECTION 3-16

1. Boolean equation, truth table, logic diagram, timing diagram, language.

## **SECTION 3-17**

1. Hardware description language2. To describe a digital circuit and itsoperation.3. To give a computer a sequential list of tasks.4. HDL describesconcurrent hardware circuits; computer instructions execute one at a time.

#### **SECTION 3-18**

 Programmable logic device
 By making and breaking connections in a switching matrix
 It translates HDL code into a pattern of bits to configure the switching matix.

#### SECTION 3-19

#### AHDL

1. The input and output definitions. 2. The description of how it operates.

1. To give a name to the circuit and define its inputs and outputs. 2. The ARCHITECTURE description. 3. <=

# **SECTION 3-20**

AHDL1. NODE2. After the I/O definition and before BEGIN.3. No4. %5. --VHDL1. SIGNAL2. Inside ARCHITECTURE before BEGIN.3. No4. --