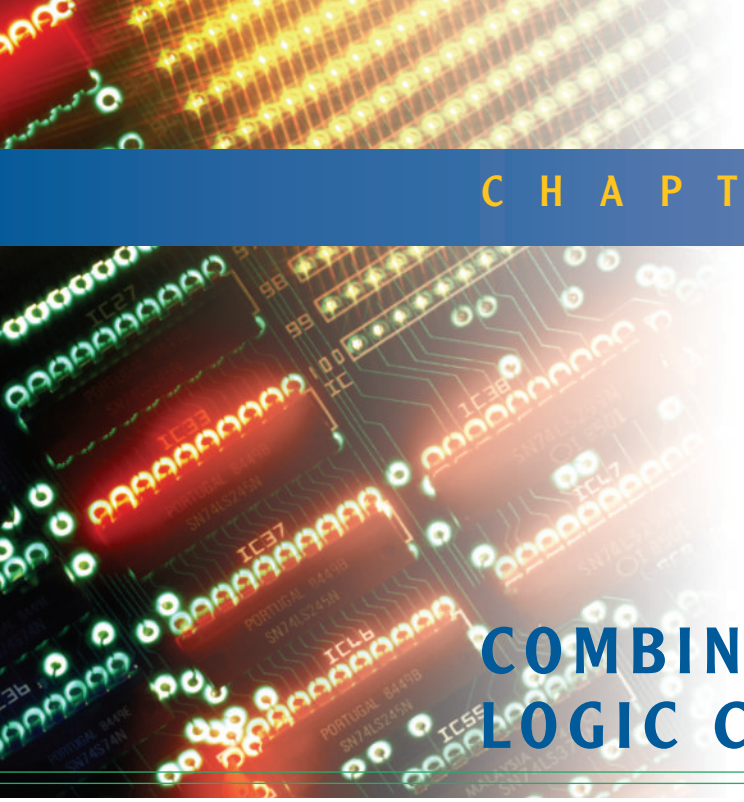


CHAPTER 4

COMBINATIONAL LOGIC CIRCUITS

OUTLINE

OBJECTIVES

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

- Convert a logic expression into a sum-of-products expression.
- Perform the necessary steps to reduce a sum-of-products expression to its simplest form.
- Use Boolean algebra and the Karnaugh map as tools to simplify and design logic circuits.
- Explain the operation of both exclusive-OR and exclusive-NOR circuits.
- Design simple logic circuits without the help of a truth table.
- Implement enable circuits.
- Cite the basic characteristics of TTL and CMOS digital ICs.
- Use the basic troubleshooting rules of digital systems.
- Deduce from observed results the faults of malfunctioning combinational logic circuits.
- Describe the fundamental idea of programmable logic devices (PLDs).
- Outline the steps involved in programming a PLD to perform a simple combinational logic function.
- Go to the Altera user manuals to acquire the information needed to do a simple programming experiment in the lab.
- Describe hierarchical design methods.
- Identify proper data types for single-bit, bit array, and numeric value variables.
- Describe logic circuits using HDL control structures IF/ELSE, IF/ELSIF, and CASE.
- Select the appropriate control structure for a given problem.

INTRODUCTION

In Chapter 3, we studied the operation of all the basic logic gates, and we used Boolean algebra to describe and analyze circuits that were made up of combinations of logic gates. These circuits can be classified as *combinational* logic circuits because, at any time, the logic level at the output depends on the combination of logic levels present at the inputs. A combinational circuit has no *memory* characteristic, so its output depends *only* on the current value of its inputs.

In this chapter, we will continue our study of combinational circuits. To start, we will go further into the simplification of logic circuits. Two methods will be used: one uses Boolean algebra theorems; the other uses a *mapping* technique. In addition, we will study simple techniques for designing

combinational logic circuits to satisfy a given set of requirements. A complete study of logic-circuit design is not one of our objectives, but the methods we introduce will provide a good introduction to logic design.

A good portion of the chapter is devoted to the troubleshooting of combinational circuits. This first exposure to troubleshooting should begin to develop the type of analytical skills needed for successful troubleshooting. To make this material as practical as possible, we will first present some of the basic characteristics of logic-gate ICs in the TTL and CMOS logic families along with a description of the most common types of faults encountered in digital IC circuits.

In the last sections of this chapter, we will extend our knowledge of programmable logic devices and hardware description languages. The concept of programmable hardware connections will be reinforced, and we will provide more details regarding the role of the development system. You will learn the steps followed in the design and development of digital systems today. Enough information will be provided to allow you to choose the correct types of data objects for use in simple projects to be presented later in this text. Finally, several control structures will be explained, along with some instruction regarding their appropriate use.

4-1 SUM-OF-PRODUCTS FORM

The methods of logic-circuit simplification and design that we will study require the logic expression to be in a **sum-of-products (SOP)** form. Some examples of this form are:

1. $ABC + \overline{A}B\overline{C}$
2. $AB + \overline{A}B\overline{C} + \overline{C}\,\overline{D} + D$
3. $\overline{A}B + C\overline{D} + EF + GK + H\overline{L}$

Each of these sum-of-products expressions consists of two or more AND terms (products) that are ORed together. Each AND term consists of one or more variables *individually* appearing in either complemented or uncomplemented form. For example, in the sum-of-products expression $ABC + \overline{A}B\overline{C}$, the first AND product contains the variables A, B, and C in their uncomplemented (not inverted) form. The second AND term contains A and C in their complemented (inverted) form. Note that in a sum-of-products expression, one inversion sign *cannot* cover more than one variable in a term (e.g., we cannot have $\overline{ABC}$ or $\overline{RST}$).

Product-of-Sums

Another general form for logic expressions is sometimes used in logic-circuit design. Called the **product-of-sums (POS)** form, it consists of two or more OR terms (sums) that are ANDed together. Each OR term contains one or more variables in complemented or uncomplemented form. Here are some product-of-sum expressions:

1. $(A + \overline{B} + C)(A + C)$
2. $(A + \overline{B})(\overline{C} + D)F$
3. $(A + C)(B + \overline{D})(\overline{B} + C)(A + \overline{D} + \overline{E})$

The methods of circuit simplification and design that we will be using are based on the sum-of-products (SOP) form, so we will not be doing much

with the product-of-sums (POS) form. It will, however, occur from time to time in some logic circuits that have a particular structure.

REVIEW QUESTIONS

1. Which of the following expressions is in SOP form?
   (a) $AB + CD + E$
   (b) $AB(C + D)$
   (c) $(A + B)(C + D + F)$
   (d) $\overline{MN} + PQ$
2. Repeat question 1 for the POS form.

4-2 SIMPLIFYING LOGIC CIRCUITS

Once the expression for a logic circuit has been obtained, we may be able to reduce it to a simpler form containing fewer terms or fewer variables in one or more terms. The new expression can then be used to implement a circuit that is equivalent to the original circuit but that contains fewer gates and connections.

To illustrate, the circuit of Figure 4-1(a) can be simplified to produce the circuit of Figure 4-1(b). Both circuits perform the same logic, so it should be obvious that the simpler circuit is more desirable because it contains fewer gates and will therefore be smaller and cheaper than the original. Furthermore, the circuit reliability will improve because there are fewer interconnections that can be potential circuit faults.

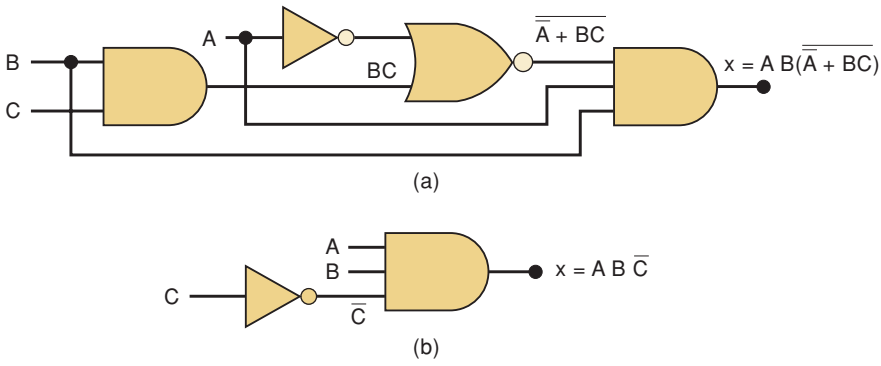


FIGURE 4-1 It is often possible to simplify a logic circuit such as that in part (a) to produce a more efficient implementation, shown in (b).

In subsequent sections, we will study two methods for simplifying logic circuits. One method will utilize the Boolean algebra theorems and, as we shall see, is greatly dependent on inspiration and experience. The other method (Karnaugh mapping) is a systematic, step-by-step approach. Some instructors may wish to skip over this latter method because it is somewhat mechanical and probably does not contribute to a better understanding of Boolean algebra. This can be done without affecting the continuity or clarity of the rest of the text.

4-3 ALGEBRAIC SIMPLIFICATION

We can use the Boolean algebra theorems that we studied in Chapter 3 to help us simplify the expression for a logic circuit. Unfortunately, it is not always obvious which theorems should be applied to produce the simplest

result. Furthermore, there is no easy way to tell whether the simplified expression is in its simplest form or whether it could have been simplified further. Thus, algebraic simplification often becomes a process of trial and error. With experience, however, one can become adept at obtaining reasonably good results.

The examples that follow will illustrate many of the ways in which the Boolean theorems can be applied in trying to simplify an expression. You should notice that these examples contain two essential steps:

1. The original expression is put into SOP form by repeated application of DeMorgan's theorems and multiplication of terms.
2. Once the original expression is in SOP form, the product terms are checked for common factors, and factoring is performed wherever possible. The factoring should result in the elimination of one or more terms.

EXAMPLE 4-1

Simplify the logic circuit shown in Figure 4-2(a).

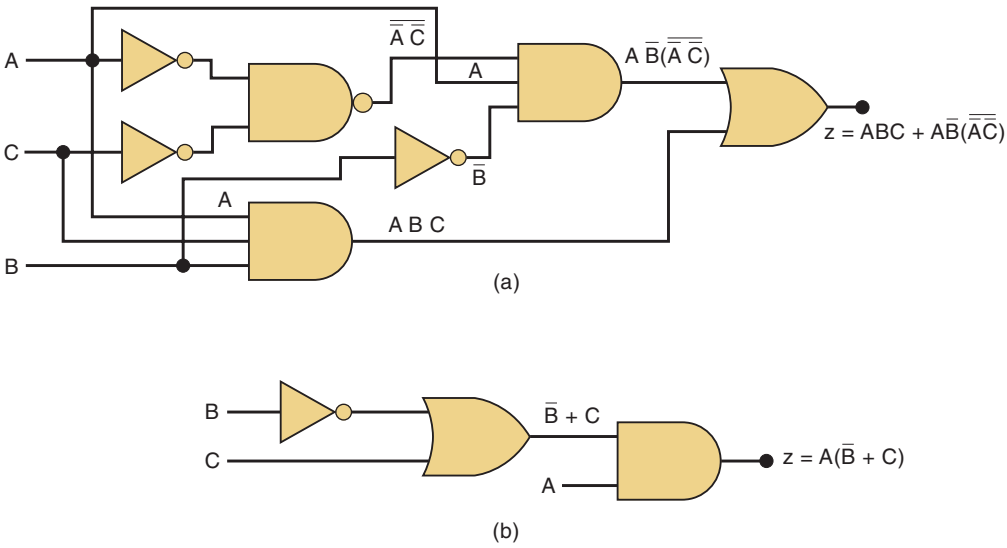


FIGURE 4-2 Example 4-1.

Solution

The first step is to determine the expression for the output using the method presented in Section 3-6. The result is

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

Once the expression is determined, it is usually a good idea to break down all large inverter signs using DeMorgan's theorems and then multiply out all terms.

$$\begin{aligned} z &= ABC + A\overline{B}(\overline{\overline{A}} + \overline{\overline{C}}) && \text{[theorem (17)]} \\ &= ABC + A\overline{B}(A + C) && \text{[cancel double inversions]} \\ &= ABC + A\overline{B}A + A\overline{B}C && \text{[multiply out]} \\ &= ABC + A\overline{B} + A\overline{B}C && [A \cdot A = A] \end{aligned}$$

With the expression now in SOP form, we should look for common variables among the various terms with the intention of factoring. The first and third terms above have AC in common, which can be factored out:

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

Since $B + \overline{B} = 1$, then

$$\begin{aligned} z &= AC(1) + A\overline{B} \\ &= AC + A\overline{B} \end{aligned}$$

We can now factor out A, which results in

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

This result can be simplified no further. Its circuit implementation is shown in Figure 4-2(b). It is obvious that the circuit in Figure 4-2(b) is a great deal simpler than the original circuit in Figure 4-2(a).

EXAMPLE 4-2

Simplify the expression $z = A\overline{B}\,\overline{C} + A\overline{B}C + ABC$.

Solution

The expression is already in SOP form.

Method 1: The first two terms in the expression have the product $A\overline{B}$ in common. Thus,

$$\begin{aligned} z &= A\overline{B}(\overline{C} + C) + ABC \\ &= A\overline{B}(1) + ABC \\ &= A\overline{B} + ABC \end{aligned}$$

We can factor the variable A from both terms:

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

Invoking theorem (15b):

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

Method 2: The original expression is $z = A\overline{B}\,\overline{C} + A\overline{B}C + ABC$. The first two terms have $A\overline{B}$ in common. The last two terms have AC in common. How do we know whether to factor $A\overline{B}$ from the first two terms or AC from the last two terms? Actually, we can do both by using the $A\overline{B}C$ term *twice.* In other words, we can rewrite the expression as:

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

where we have added an extra term $A\overline{B}C$. This is valid and will not change the value of the expression because $A\overline{B}C + A\overline{B}C = A\overline{B}C$ [theorem (7)]. Now we can factor $A\overline{B}$ from the first two terms and AC from the last two terms:

$$\begin{aligned} z &= A\overline{B}(C + \overline{C}) + AC(\overline{B} + B) \\ &= A\overline{B} \cdot 1 + AC \cdot 1 \\ &= A\overline{B} + AC = A(\overline{B} + C) \end{aligned}$$

Of course, this is the same result obtained with method 1. This trick of using the same term twice can always be used. In fact, the same term can be used more than twice if necessary.

EXAMPLE 4-3

Simplify $z = \overline{A}C(\overline{\overline{A}BD}) + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C$.

Solution

First, use DeMorgan's theorem on the first term:

$$z = \overline{A}C(A + \overline{B} + \overline{D}) + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad \textbf{(step 1)}$$

Multiplying out yields

$$z = \overline{A}CA + \overline{A}C\overline{B} + \overline{A}C\overline{D} + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad \textbf{(2)}$$

Because $\overline{A} \cdot A = 0$, the first term is eliminated:

$$z = \overline{A}\,\overline{B}C + \overline{A}C\overline{D} + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C \qquad \textbf{(3)}$$

This is the desired SOP form. Now we must look for common factors among the various product terms. The idea is to check for the largest common factor between any two or more product terms. For example, the first and last terms have the common factor $\overline{B}C$, and the second and third terms share the common factor $\overline{A}\,\overline{D}$. We can factor these out as follows:

$$z = \overline{B}C(\overline{A} + A) + \overline{A}\,\overline{D}(C + B\overline{C}) \qquad \textbf{(4)}$$

Now, because $\overline{A} + A = 1$, and $C + B\overline{C} = C + B$ [theorem (15a)], we have

$$z = \overline{B}C + \overline{A}\,\overline{D}(B + C) \qquad \textbf{(5)}$$

This same result could have been reached with other choices for the factoring. For example, we could have factored C from the first, second, and fourth product terms in step 3 to obtain

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

The expression inside the parentheses can be factored further:

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

Because $\overline{A} + A = 1$, this becomes

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

Multiplying out yields

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

Now we can factor $\overline{A}\,\overline{D}$ from the second and third terms to get

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

If we use theorem (15a), the expression in parentheses becomes $B + C$. Thus, we finally have

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

This is the same result that we obtained earlier, but it took us many more steps. This illustrates why you should look for the largest common factors: it will generally lead to the final expression in the fewest steps.

Example 4-3 illustrates the frustration often encountered in Boolean simplification. Because we have arrived at the same equation (which appears irreducible) by two different methods, it might seem reasonable to conclude that this final equation is the simplest form. In fact, the simplest form of this equation is

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

But there is no apparent way to reduce step (5) to reach this simpler version. In this case, we missed an operation earlier in the process that could have led to the simpler form. The question is, "How could we have known that we missed a step?" Later in this chapter, we will examine a mapping technique that will always lead to the simplest SOP form.

EXAMPLE 4-4

Simplify the expression $x = (\overline{A} + B)(A + B + D)\overline{D}$.

Solution

The expression can be put into sum-of-products form by multiplying out all the terms. The result is

$$x = \overline{A}A\overline{D} + \overline{A}B\overline{D} + \overline{A}D\overline{D} + BA\overline{D} + BB\overline{D} + BD\overline{D}$$

The first term can be eliminated because $\overline{A}A = 0$. Likewise, the third and sixth terms can be eliminated because $D\overline{D} = 0$. The fifth term can be simplified to $B\overline{D}$ because $BB = B$. This gives us

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

We can factor $B\overline{D}$ from each term to obtain

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

Clearly, the term inside the parentheses is always 1, so we finally have

$$x = B\overline{D}$$

EXAMPLE 4-5

Simplify the circuit of Figure 4-3(a).

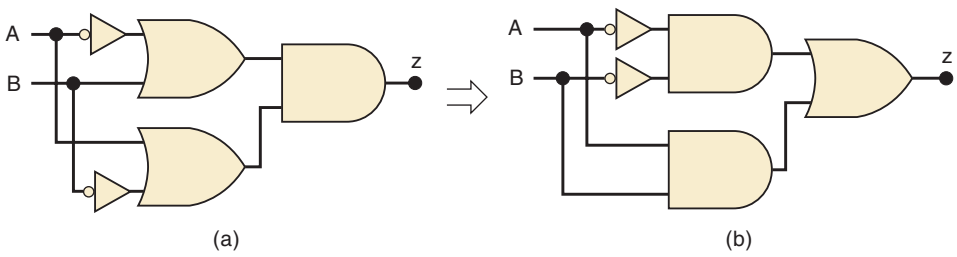


FIGURE 4-3 Example 4-5.

Solution

The expression for output z is

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

Multiplying out to get the sum-of-products form, we obtain

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

We can eliminate $\overline{A}A = 0$ and $B\overline{B} = 0$ to end up with

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

This expression is implemented in Figure 4-3(b), and if we compare it with the original circuit, we see that both circuits contain the same number of gates and connections. In this case, the simplification process produced an equivalent, but not simpler, circuit.

EXAMPLE 4-6

Simplify $x = A\overline{B}C + \overline{A}BD + \overline{C}\,\overline{D}$.

Solution

You can try, but you will not be able to simplify this expression any further.

REVIEW QUESTIONS

1. State which of the following expressions are *not* in the sum-of-products form:
   (a) $RS\overline{T} + \overline{R}S\overline{T} + \overline{T}$
   (b) $A\overline{DC} + \overline{A}DC$
   (c) $MN\overline{P} + (M + \overline{N})P$
   (d) $AB + \overline{A}B\overline{C} + A\,\overline{B}\,\overline{C}\,D$
2. Simplify the circuit in Figure 4-1(a) to arrive at the circuit of Figure 4-1(b).
3. Change each AND gate in Figure 4-1(a) to a NAND gate. Determine the new expression for x and simplify it.

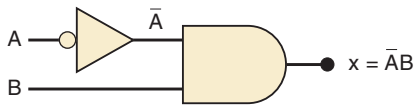
4-4 DESIGNING COMBINATIONAL LOGIC CIRCUITS

When the desired output level of a logic circuit is given for all possible input conditions, the results can be conveniently displayed in a truth table. The Boolean expression for the required circuit can then be derived from the truth table. For example, consider Figure 4-4(a), where a truth table is shown for a circuit that has two inputs, A and B, and output x. The table shows that output x is to be at the 1 level *only* for the case where $A = 0$ and $B = 1$. It now remains to determine what logic circuit will produce this desired operation. It should be apparent that one possible solution is that shown in Figure 4-4(b). Here an AND gate is used with inputs $\overline{A}$ and B, so that $x = \overline{A} \cdot B$. Obviously x will be 1 *only if* both inputs to the AND gate are 1, namely, $\overline{A} = 1$ (which means that $A = 0$) and $B = 1$. For all other values of A and B, the output x will be 0.

FIGURE 4-4 Circuit that produces a 1 output only for the $A = 0, B = 1$ condition.

A	B	x
0	0	0
0	1	1
1	0	0
1	1	0

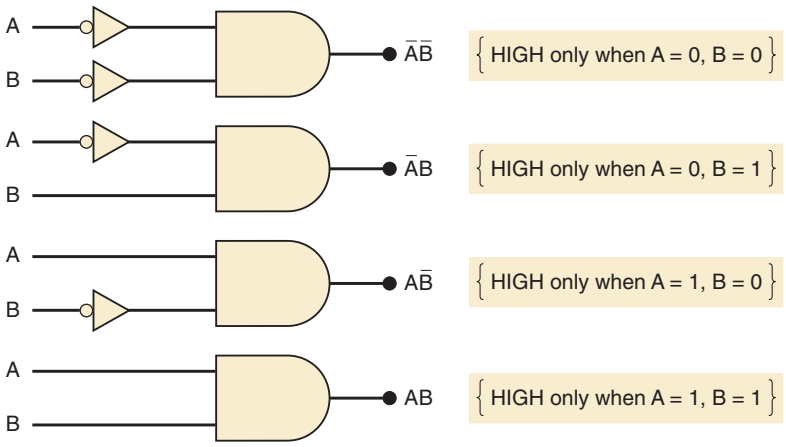
(a)



(b)

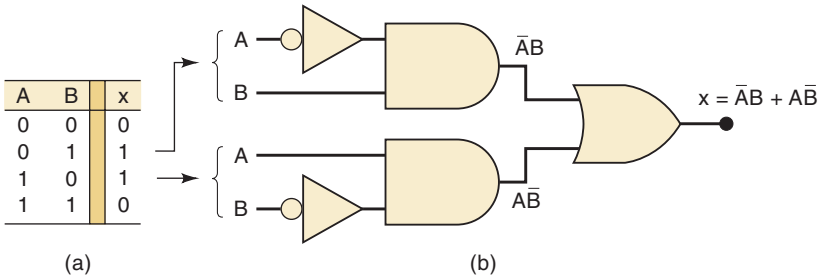
A similar approach can be used for the other input conditions. For instance, if x were to be high only for the $A = 1, B = 0$ condition, the resulting circuit would be an AND gate with inputs A and $\overline{B}$. In other words, for any of the four possible input conditions, we can generate a high x output by using an AND gate with appropriate inputs to generate the required AND product. The four different cases are shown in Figure 4-5. Each of the AND gates shown generates an output that is 1 *only* for one given input condition and the output is 0 for all other conditions. It should be noted that the AND inputs are inverted or not inverted depending on the values that the variables have for the given condition. If the variable is 0 for the given condition, it is inverted before entering the AND gate.

FIGURE 4-5 An AND gate with appropriate inputs can be used to produce a 1 output for a specific set of input levels.



Let us now consider the case shown in Figure 4-6(a), where we have a truth table that indicates that the output x is to be 1 for two different cases: $A = 0, B = 1$ and $A = 1, B = 0$. How can this be implemented? We know that

the AND term $\overline{A} \cdot B$ will generate a 1 only for the $A = 0, B = 1$ condition, and the AND term $A \cdot \overline{B}$ will generate a 1 for the $A = 1, B = 0$ condition. Because x must be HIGH for *either* condition, it should be clear that these terms should be ORed together to produce the desired output, x. This implementation is shown in Figure 4-6(b), where the resulting expression for the output is $x = \overline{A}B + A\overline{B}$.



A	B	x
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 4-6 Each set of input conditions that is to produce a HIGH output is implemented by a separate AND gate. The AND outputs are ORed to produce the final output.

In this example, an AND term is generated for each case in the table where the output x is to be a 1. The AND gate outputs are then ORed together to produce the total output x, which will be 1 when either AND term is 1. This same procedure can be extended to examples with more than two inputs. Consider the truth table for a three-input circuit (Table 4-1). Here there are three cases where the output x is to be 1. The required AND term for each of these cases is shown. Again, note that for each case where a variable is 0, it appears inverted in the AND term. The sum-of-products expression for x is obtained by ORing the three AND terms.

$$x = \overline{A}B\overline{C} + \overline{A}BC + ABC$$

TABLE 4-1

A	*B*	*C*	*x*	
0	0	0	0	
0	0	1	0	
0	1	0	1	$\rightarrow \overline{A}B\overline{C}$
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$\rightarrow ABC$

Complete Design Procedure

Any logic problem can be solved using the following step-by-step procedure.

1. Interpret the problem and set up a truth table to describe its operation.
2. Write the AND (product) term for each case where the output is 1.
3. Write the sum-of-products (SOP) expression for the output.
4. Simplify the output expression if possible.
5. Implement the circuit for the final, simplified expression.

The following example illustrates the complete design procedure.

EXAMPLE 4-7

Design a logic circuit that has three inputs, A, B, and C, and whose output will be HIGH only when a majority of the inputs are HIGH.

Solution

Step 1. Set up the truth table.

On the basis of the problem statement, the output x should be 1 whenever two or more inputs are 1; for all other cases, the output should be 0 (Table 4-2).

TABLE 4-2

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\overline{B}C$
1	1	0	1	$\rightarrow AB\overline{C}$
1	1	1	1	$\rightarrow ABC$

Step 2. Write the AND term for each case where the output is a 1.

There are four such cases. The AND terms are shown next to the truth table (Table 4-2). Again note that each AND term contains each input variable in either inverted or noninverted form.

Step 3. Write the sum-of-products expression for the output.

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

Step 4. Simplify the output expression.

This expression can be simplified in several ways. Perhaps the quickest way is to realize that the last term ABC has two variables in common with each of the other terms. Thus, we can use the ABC term to factor with each of the other terms. The expression is rewritten with the ABC term occurring three times (recall from Example 4-2 that this is legal in Boolean algebra):

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

Factoring the appropriate pairs of terms, we have

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

Each term in parentheses is equal to 1, so we have

$$x = BC + AC + AB$$

Step 5. Implement the circuit for the final expression.

This expression is implemented in Figure 4-7. Since the expression is in SOP form, the circuit consists of a group of AND gates working into a single OR gate.

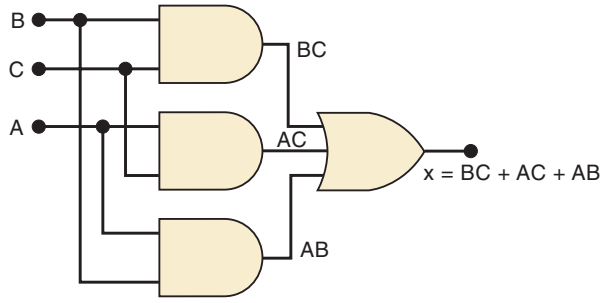
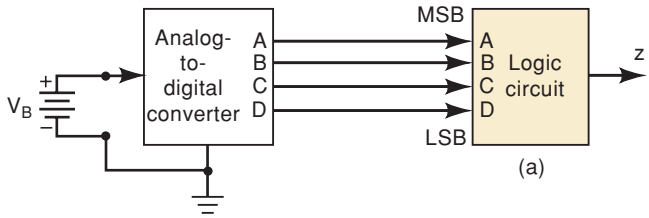


FIGURE 4-7 Example 4-7.

EXAMPLE 4-8

Refer to Figure 4-8(a), where an analog-to-digital converter is monitoring the dc voltage of a 12-V storage battery on an orbiting spaceship. The converter's output is a four-bit binary number, *ABCD*, corresponding to the battery voltage in steps of 1 V, with *A* as the MSB. The converter's binary outputs are fed to a logic circuit that is to produce a HIGH output as long as the binary value is greater than $0110_2 = 6_{10}$; that is, the battery voltage is greater than 6 V. Design this logic circuit.



	A	B	C	D	z	
(0)	0	0	0	0	0	
(1)	0	0	0	1	0	
(2)	0	0	1	0	0	
(3)	0	0	1	1	0	
(4)	0	1	0	0	0	
(5)	0	1	0	1	0	
(6)	0	1	1	0	0	
(7)	0	1	1	1	1	$\rightarrow \bar{A}BCD$
(8)	1	0	0	0	1	$\rightarrow A\bar{B}\bar{C}\bar{D}$
(9)	1	0	0	1	1	$\rightarrow A\bar{B}\bar{C}D$
(10)	1	0	1	0	1	$\rightarrow A\bar{B}C\bar{D}$
(11)	1	0	1	1	1	$\rightarrow A\bar{B}CD$
(12)	1	1	0	0	1	$\rightarrow AB\bar{C}\bar{D}$
(13)	1	1	0	1	1	$\rightarrow AB\bar{C}D$
(14)	1	1	1	0	1	$\rightarrow ABC\bar{D}$
(15)	1	1	1	1	1	$\rightarrow ABCD$

(b)

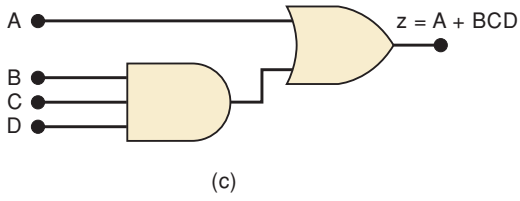


FIGURE 4-8 Example 4-8.

Solution

The truth table is shown in Figure 4-8(b). For each case in the truth table, we have indicated the decimal equivalent of the binary number represented by the *ABCD* combination.

The output *z* is set equal to 1 for all those cases where the binary number is greater than 0110. For all other cases, *z* is set equal to 0. This truth table gives us the following sum-of-products expression:

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

Simplification of this expression will be a formidable task, but with a little care it can be accomplished. The step-by-step process involves factoring and eliminating terms of the form $A + \overline{A}$:

$$\begin{aligned} z &= \overline{A}BCD + A\overline{B}\,\overline{C}(\overline{D} + D) + A\overline{B}C(\overline{D} + D) + AB\overline{C}(\overline{D} + D) + ABC(\overline{D} + D) \\ &= \overline{A}BCD + A\overline{B}\,\overline{C} + A\overline{B}C + AB\overline{C} + ABC \\ &= \overline{A}BCD + A\overline{B}(\overline{C} + C) + AB(\overline{C} + C) \\ &= \overline{A}BCD + A\overline{B} + AB \\ &= \overline{A}BCD + A(\overline{B} + B) \\ &= \overline{A}BCD + A \end{aligned}$$

This can be reduced further by invoking theorem (15a), which says that $x + \overline{x}y = x + y$. In this case $x = A$ and $y = BCD$. Thus,

$$z = \overline{A}BCD + A = BCD + A$$

This final expression is implemented in Figure 4-8(c).

As this example demonstrates, the algebraic simplification method can be quite lengthy when the original expression contains a large number of terms. This is a limitation that is not shared by the Karnaugh mapping method, as we will see later.

EXAMPLE 4-9

Refer to Figure 4-9(a). In a simple copy machine, a stop signal, S, is to be generated to stop the machine operation and energize an indicator light whenever either of the following conditions exists: (1) there is no paper in the paper feeder tray; or (2) the two microswitches in the paper path are

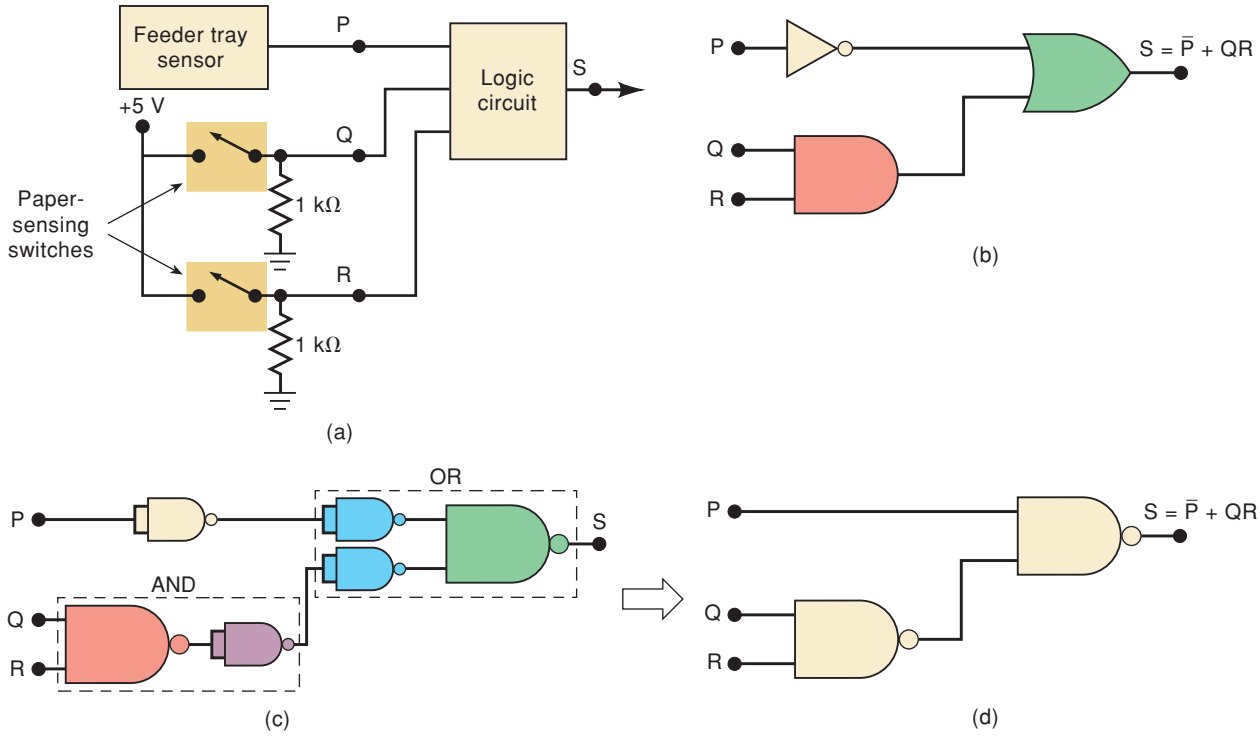


FIGURE 4-9 Example 4-9.

activated, indicating a jam in the paper path. The presence of paper in the feeder tray is indicated by a HIGH at logic signal *P*. Each of the microswitches produces a logic signal (*Q* and *R*) that goes HIGH whenever paper is passing over the switch to activate it. Design the logic circuit to produce a HIGH at output signal *S* for the stated conditions, and implement it using the 74HC00 CMOS quad two-input NAND chip.

Solution

We will use the five-step process used in Example 4-7. The truth table is shown in Table 4-3. The *S* output will be a logic 1 whenever $P = 0$ because this indicates no paper in the feeder tray. *S* will also be a 1 for the two cases where *Q* and *R* are both 1, indicating a paper jam. As the table shows, there are five different input conditions that produce a HIGH output. **(Step 1)**

TABLE 4-3

P	Q	R	S	
0	0	0	1	$\overline{P}\,\overline{Q}\,\overline{R}$
0	0	1	1	$\overline{P}\,\overline{Q}R$
0	1	0	1	$\overline{P}Q\overline{R}$
0	1	1	1	$\overline{P}QR$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	PQR

The AND terms for each of these cases are shown. **(Step 2)**
The sum-of-products expression becomes

$$S = \overline{P}\,\overline{Q}\,\overline{R} + \overline{P}\,\overline{Q}R + \overline{P}Q\overline{R} + \overline{P}QR + PQR \qquad \textbf{(Step 3)}$$

We can begin the simplification by factoring out $\overline{P}\,\overline{Q}$ from terms 1 and 2 and by factoring out $\overline{P}Q$ from terms 3 and 4:

$$S = \overline{P}\,\overline{Q}(\overline{R} + R) + \overline{P}Q(\overline{R} + R) + PQR \qquad \textbf{(Step 4)}$$

Now we can eliminate the $\overline{R} + R$ terms because they equal 1:

$$S = \overline{P}\,\overline{Q} + \overline{P}Q + PQR$$

Factoring $\overline{P}$ from terms 1 and 2 allows us to eliminate *Q* from these terms:

$$S = \overline{P} + PQR$$

Here we can apply theorem (15b) ($\overline{x} + xy = \overline{x} + y$) to obtain

$$S = \overline{P} + QR$$

As a double check of this simplified Boolean equation, let's see if it matches the truth table that we started out with. This equation says that the output *S* will be HIGH whenever *P* is LOW OR both *Q* AND *R* are HIGH. Look at Table 4-3 and observe that the output is HIGH for all four cases

when *P* is LOW. *S* is also HIGH when *Q* AND *R* are both HIGH, regardless of the state of *P*. This agrees with the equation.

The AND/OR implementation for this circuit is shown in Figure 4-9(b).

(Step 5)

To implement this circuit using the 74HC00 quad two-input NAND chip, we must convert each gate and the INVERTER by their NAND-gate equivalents (per Section 3-12). This is shown in Figure 4-9(c). Clearly, we can eliminate the double inverters to produce the NAND-gate implementation shown in Figure 4-9(d).

The final wired-up circuit is obtained by connecting two of the NAND gates on the 74HC00 chip. This CMOS chip has the same gate configuration and pin numbers as the TTL 74LS00 chip of Figure 3-31. Figure 4-10 shows the wired-up circuit with pin numbers, including the +5 V and GROUND pins. It also includes an output driver transistor and LED to indicate the state of output *S*.

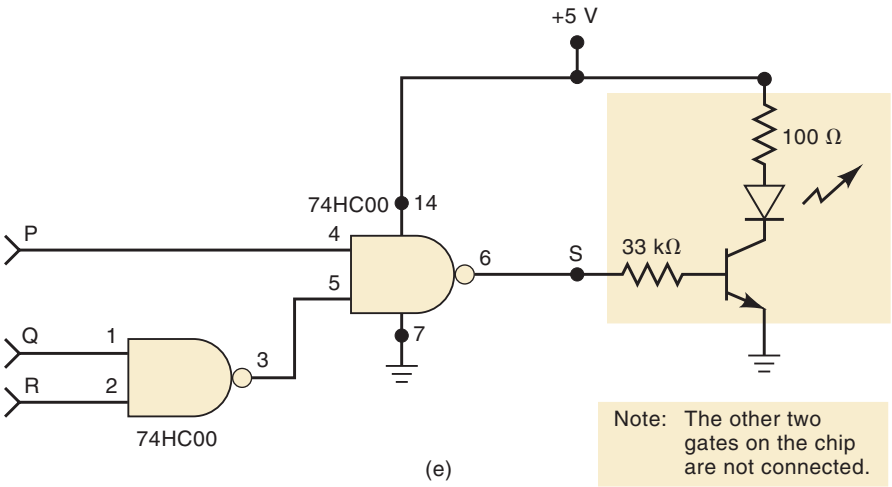


FIGURE 4-10 Circuit of Figure 4-9(d) implemented using 74HC00 NAND chip.

REVIEW QUESTIONS

1. Write the sum-of-products expression for a circuit with four inputs and an output that is to be HIGH only when input *A* is LOW at the same time that exactly two other inputs are LOW.
2. Implement the expression of question 1 using all four-input NAND gates. How many are required?

4-5 KARNAUGH MAP METHOD

The **Karnaugh map (K map)** is a graphical tool used to simplify a logic equation or to convert a truth table to its corresponding logic circuit in a simple, orderly process. Although a K map can be used for problems involving any number of input variables, its practical usefulness is limited to five or six variables. The following discussion will be limited to problems with up to four inputs because even five- and six-input problems are too involved and are best done by a computer program.

Karnaugh Map Format

The K map, like a truth table, is a means for showing the relationship between logic inputs and the desired output. Figure 4-11 shows three examples of K maps for two, three, and four variables, together with the corresponding truth tables. These examples illustrate the following important points:

1. The truth table gives the value of output X for each combination of input values. The K map gives the same information in a different format. Each case in the truth table corresponds to a square in the K map. For example, in Figure 4-11(a), the $A = 0, B = 0$ condition in the truth table corresponds to the $\overline{A}\,\overline{B}$ square in the K map. Because the truth table shows $X = 1$ for this case, a 1 is placed in the $\overline{A}\,\overline{B}$ square in the K map. Similarly, the $A = 1, B = 1$ condition in the truth table corresponds to the AB square of the K map. Because $X = 1$ for this case, a 1 is placed in the AB square. All other squares are filled with 0s. This same idea is used in the three- and four-variable maps shown in the figure.
2. The K-map squares are labeled so that horizontally adjacent squares differ only in one variable. For example, the upper left-hand square in the four-variable map is $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, while the square immediately to its right is $\overline{A}\,\overline{B}\,\overline{C}D$ (only the D variable is different). Similarly, vertically adjacent

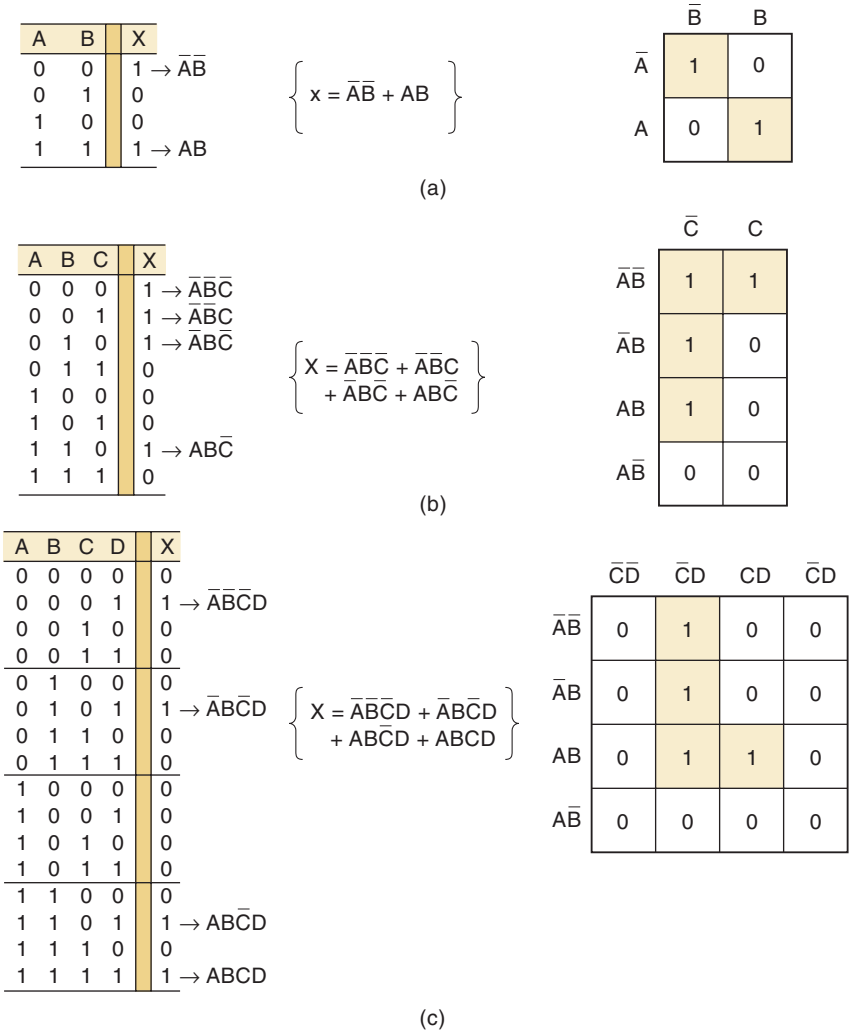


FIGURE 4-11 Karnaugh maps and truth tables for (a) two, (b) three, and (c) four variables.

squares differ only in one variable. For example, the upper left-hand square is $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, while the square directly below it is $A\overline{B}\overline{C}\,\overline{D}$ (only the B variable is different).

Note that each square in the top row is considered to be adjacent to a corresponding square in the bottom row. For example, the $\overline{A}\,\overline{B}CD$ square in the top row is adjacent to the $A\overline{B}CD$ square in the bottom row because they differ only in the A variable. You can think of the top of the map as being wrapped around to touch the bottom of the map. Similarly, squares in the leftmost column are adjacent to corresponding squares in the rightmost column.

3. In order for vertically and horizontally adjacent squares to differ in only one variable, the top-to-bottom labeling must be done in the order shown: $\overline{A}\,\overline{B}$, $\overline{A}B$, AB, $A\overline{B}$. The same is true of the left-to-right labeling: $\overline{C}\,\overline{D}$, $\overline{C}D$, CD, $C\overline{D}$.
4. Once a K map has been filled with 0s and 1s, the sum-of-products expression for the output X can be obtained by ORing together those squares that contain a 1. In the three-variable map of Figure 4-11(b), the $\overline{A}\,\overline{B}\,\overline{C}$, $\overline{A}\,\overline{B}C$, $\overline{A}B\overline{C}$, and $AB\overline{C}$ squares contain a 1, so that $X = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + AB\overline{C}$.

Looping

The expression for output X can be simplified by properly combining those squares in the K map that contain 1s. The process for combining these 1s is called **looping**.

Looping Groups of Two (Pairs)

Figure 4-12(a) is the K map for a particular three-variable truth table. This map contains a pair of 1s that are vertically adjacent to each other; the first

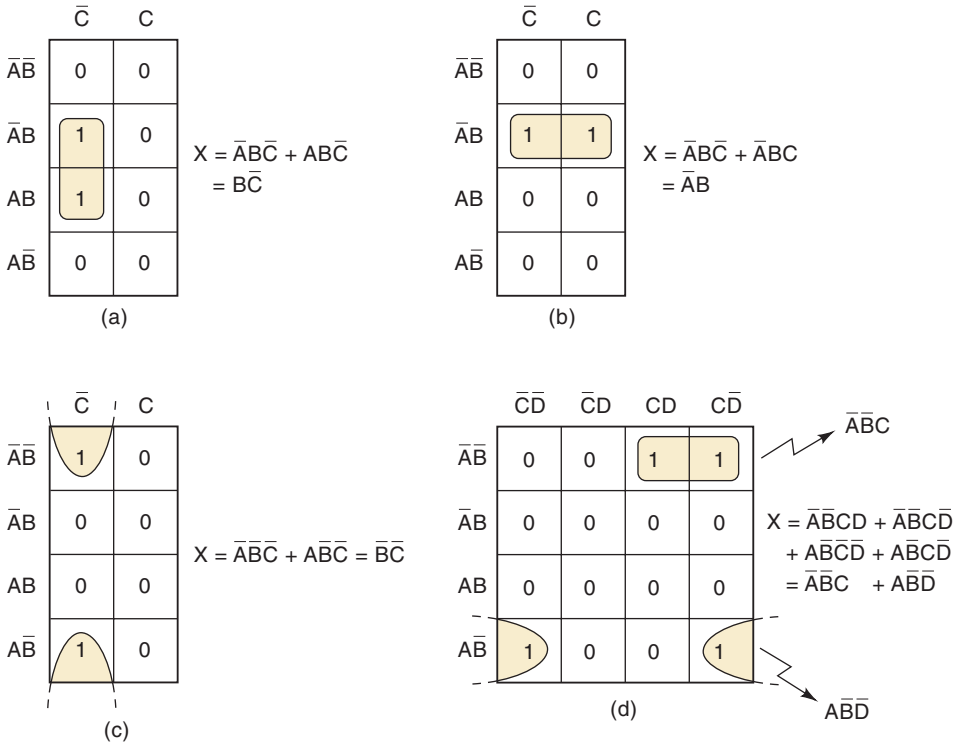


FIGURE 4-12 Examples of looping pairs of adjacent 1s.

represents $\overline{A}B\overline{C}$, and the second represents $AB\overline{C}$. Note that in these two terms only the A variable appears in both normal and complemented (inverted) form, while B and $\overline{C}$ remain unchanged. These two terms can be looped (combined) to give a resultant that eliminates the A variable because it appears in both uncomplemented and complemented forms. This is easily proved as follows:

$$\begin{aligned} X &= \overline{A}B\overline{C} + AB\overline{C} \\ &= B\overline{C}(\overline{A} + A) \\ &= B\overline{C}(1) = B\overline{C} \end{aligned}$$

This same principle holds true for any pair of vertically or horizontally adjacent 1s. Figure 4-12(b) shows an example of two horizontally adjacent 1s. These two can be looped and the C variable eliminated because it appears in both its uncomplemented and complemented forms to give a resultant of $X = \overline{A}B$.

Another example is shown in Figure 4-12(c). In a K map, the top row and bottom row of squares are considered to be adjacent. Thus, the two 1s in this map can be looped to provide a resultant of $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} = \overline{B}\,\overline{C}$.

Figure 4-12(d) shows a K map that has two pairs of 1s that can be looped. The two 1s in the top row are horizontally adjacent. The two 1s in the bottom row are also adjacent because, in a K map, the leftmost column and the rightmost column of squares are considered to be adjacent. When the top pair of 1s is looped, the D variable is eliminated (because it appears as both D and $\overline{D}$) to give the term $\overline{A}\,\overline{B}C$. Looping the bottom pair eliminates the C variable to give the term $A\overline{B}\,\overline{D}$. These two terms are ORed to give the final result for X.

To summarize:

Looping a pair of adjacent 1s in a K map eliminates the variable that appears in complemented and uncomplemented form.

Looping Groups of Four (Quads)

A K map may contain a group of four 1s that are adjacent to each other. This group is called a *quad*. Figure 4-13 shows several examples of quads. In Figure 4-13(a), the four 1s are vertically adjacent, and in Figure 4-13(b), they are horizontally adjacent. The K map in Figure 4-13(c) contains four 1s in a square, and they are considered adjacent to each other. The four 1s in Figure 4-13(d) are also adjacent, as are those in Figure 4-13(e), because, as pointed out earlier, the top and bottom rows are considered to be adjacent to each other, as are the leftmost and rightmost columns.

When a quad is looped, the resultant term will contain only the variables that do not change form for all the squares in the quad. For example, in Figure 4-13(a), the four squares that contain a 1 are $\overline{A}\,\overline{B}C$, $\overline{A}BC$, ABC, and $A\overline{B}C$. Examination of these terms reveals that only the variable C remains unchanged (both A and B appear in complemented and uncomplemented form). Thus, the resultant expression for X is simply $X = C$. This can be proved as follows:

$$\begin{aligned} X &= \overline{A}\,\overline{B}C + \overline{A}BC + ABC + A\overline{B}C \\ &= \overline{A}C(\overline{B} + B) + AC(B + \overline{B}) \\ &= \overline{A}C + AC \\ &= C(\overline{A} + A) = C \end{aligned}$$

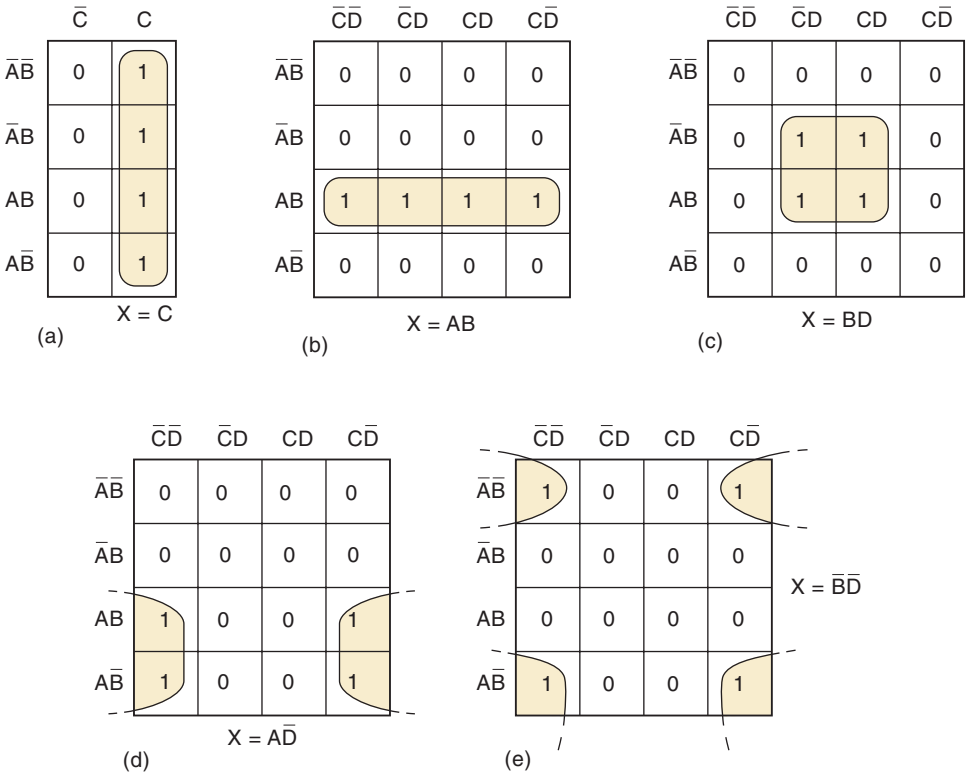


FIGURE 4-13 Examples of looping groups of four 1s (quads).

As another example, consider Figure 4-13(d), where the four squares containing 1s are $AB\overline{C}\,\overline{D}$, $A\,\overline{B}\,\overline{C}D$, $AB\overline{C}D$, and $A\overline{B}\overline{C}D$. Examination of these terms indicates that only the variables A and $\overline{D}$ remain unchanged, so that the simplified expression for X is

$$X = A\overline{D}$$

This can be proved in the same manner that was used above. The reader should check each of the other cases in Figure 4-13 to verify the indicated expressions for X.

To summarize:

Looping a quad of adjacent 1s eliminates the two variables that appear in both complemented and uncomplemented form.

Looping Groups of Eight (Octets)

A group of eight 1s that are adjacent to one another is called an *octet.* Several examples of octets are shown in Figure 4-14. When an octet is looped in a four-variable map, three of the four variables are eliminated because only one variable remains unchanged. For example, examination of the eight looped squares in Figure 4-14(a) shows that only the variable B is in the same form for all eight squares: the other variables appear in complemented and uncomplemented form. Thus, for this map, $X = B$. The reader can verify the results for the other examples in Figure 4-14.

To summarize:

Looping an octet of adjacent 1s eliminates the three variables that appear in both complemented and uncomplemented form.

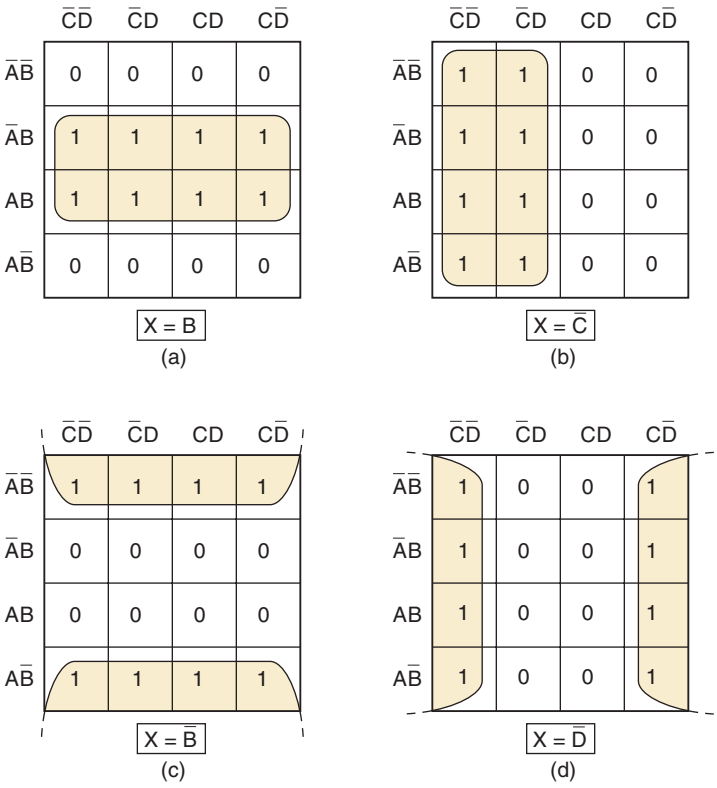


FIGURE 4-14 Examples of looping groups of eight 1s (octets).

Complete Simplification Process

We have seen how looping of pairs, quads, and octets on a K map can be used to obtain a simplified expression. We can summarize the rule for loops of *any* size:

> **When a variable appears in both complemented and uncomplemented form within a loop, that variable is eliminated from the expression. Variables that are the same for all squares of the loop must appear in the final expression.**

It should be clear that a larger loop of 1s eliminates more variables. To be exact, a loop of two eliminates one variable, a loop of four eliminates two variables, and a loop of eight eliminates three. This principle will now be used to obtain a simplified logic expression from a K map that contains any combination of 1s and 0s.

The procedure will first be outlined and then applied to several examples. The steps below are followed in using the K-map method for simplifying a Boolean expression:

Step 1 Construct the K map and place 1s in those squares corresponding to the 1s in the truth table. Place 0s in the other squares.

Step 2 Examine the map for adjacent 1s and loop those 1s that are *not* adjacent to any other 1s. These are called *isolated* 1s.

Step 3 Next, look for those 1s that are adjacent to only one other 1. Loop *any* pair containing such a 1.

Step 4 Loop any octet even if it contains some 1s that have already been looped.

Step 5 Loop any quad that contains one or more 1s that have not already been looped, *making sure to use the minimum number of loops.*

Step 6 Loop any pairs necessary to include any 1s that have not yet been looped, *making sure to use the minimum number of loops.*

Step 7 Form the OR sum of all the terms generated by each loop.

These steps will be followed exactly and referred to in the following examples. In each case, the resulting logic expression will be in its simplest sum-of-products form.

EXAMPLE 4-10

Figure 4-15(a) shows the K map for a four-variable problem. We will assume that the map was obtained from the problem truth table **(step 1)**. The squares are numbered for convenience in identifying each loop.

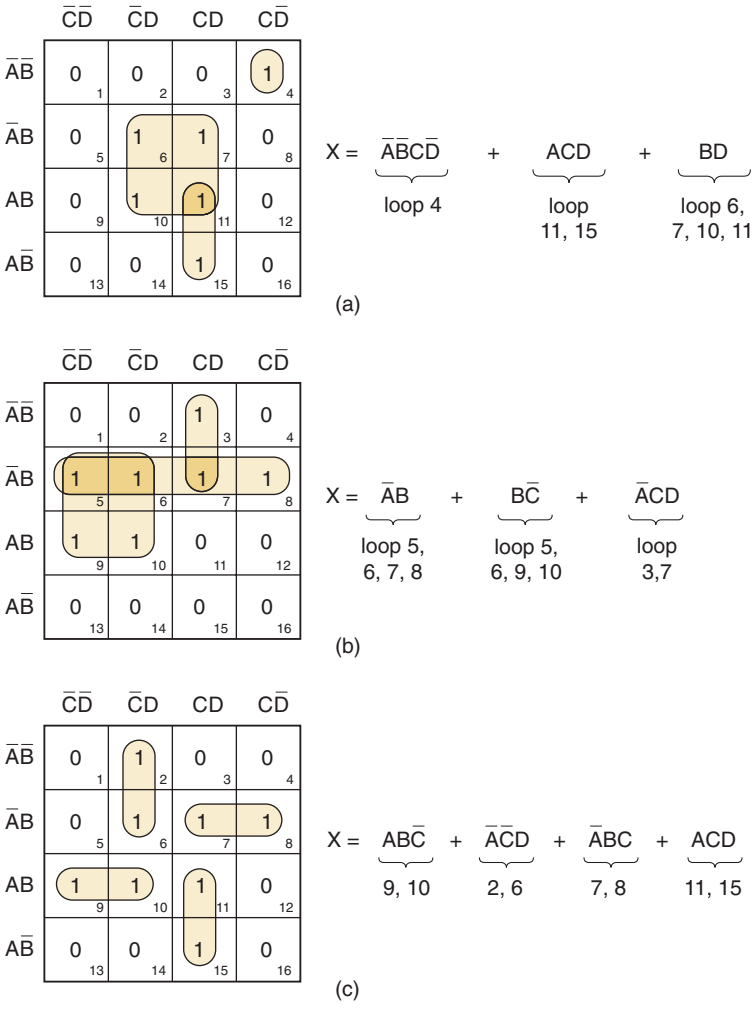


FIGURE 4-15 Examples 4-10 to 4-12.

Step 2 Square 4 is the only square containing a 1 that is not adjacent to any other 1. It is looped and is referred to as loop 4.

Step 3 Square 15 is adjacent *only* to square 11. This pair is looped and referred to as loop 11, 15.

Step 4 There are no octets.

Step 5 Squares 6, 7, 10, and 11 form a quad. This quad is looped (loop 6, 7, 10, 11). Note that square 11 is used again, even though it was part of loop 11, 15.

Step 6 All 1s have already been looped.

Step 7 Each loop generates a term in the expression for X. Loop 4 is simply $\overline{A}\,\overline{B}C\overline{D}$. Loop 11, 15 is ACD (the B variable is eliminated). Loop 6, 7, 10, 11 is BD (A and C are eliminated).

EXAMPLE 4-11

Consider the K map in Figure 4-15(b). Once again, we can assume that step 1 has already been performed.

Step 2 There are no isolated 1s.

Step 3 The 1 in square 3 is adjacent *only* to the 1 in square 7. Looping this pair (loop 3, 7) produces the term $\overline{A}CD$.

Step 4 There are no octets.

Step 5 There are two quads. Squares 5, 6, 7, and 8 form one quad. Looping this quad produces the term $\overline{A}B$. The second quad is made up of squares 5, 6, 9, and 10. This quad is looped because it contains two squares that have not been looped previously. Looping this quad produces $B\overline{C}$.

Step 6 All 1s have already been looped.

Step 7 The terms generated by the three loops are ORed together to obtain the expression for X.

EXAMPLE 4-12

Consider the K map in Figure 4-15(c).

Step 2 There are no isolated 1s.

Step 3 The 1 in square 2 is adjacent only to the 1 in square 6. This pair is looped to produce $\overline{A}\,\overline{C}D$. Similarly, square 9 is adjacent only to square 10. Looping this pair produces $AB\overline{C}$. Likewise, loop 7, 8 and loop 11, 15 produce the terms $\overline{A}BC$ and ACD, respectively.

Step 4 There are no octets.

Step 5 There is one quad formed by squares 6, 7, 10, and 11. This quad, however, is *not* looped because all the 1s in the quad have been included in other loops.

Step 6 All 1s have already been looped.

Step 7 The expression for X is shown in the figure.

EXAMPLE 4-13

Consider the K map in Figure 4-16(a).

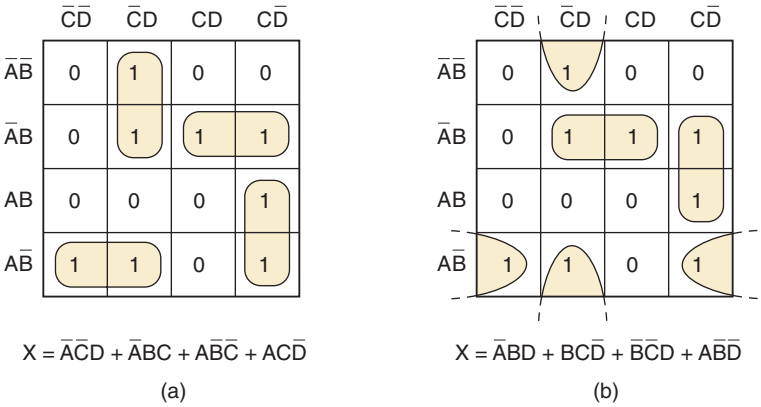


FIGURE 4-16 The same K map with two equally good solutions.

Step 2 There are no isolated 1s.

Step 3 There are no 1s that are adjacent to only one other 1.

Step 4 There are no octets.

Step 5 There are no quads.

Steps 6 and 7 There are many possible pairs. The looping must use the minimum number of loops to account for all the 1s. For this map, there are *two* possible loopings, which require only four looped pairs. Figure 4-16(a) shows one solution and its resultant expression. Figure 4-16(b) shows the other. Note that both expressions are of the same complexity, and so neither is better than the other.

Filling a K Map from an Output Expression

When the desired output is presented as a Boolean expression instead of a truth table, the K map can be filled by using the following steps:

1. Get the expression into SOP form if it is not already in that form.
2. For each product term in the SOP expression, place a 1 in each K-map square whose label contains the same combination of input variables. Place a 0 in all other squares.

The following example illustrates this procedure.

EXAMPLE 4-14

Use a K map to simplify $y = \overline{C}(\overline{A}\,\overline{B}\,\overline{D} + D) + A\overline{B}C + \overline{D}$.

Solution

1. Multiply out the first term to get $y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{C}D + A\overline{B}C + \overline{D}$, which is now in SOP form.
2. For the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ term, simply put a 1 in the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ square of the K map (Figure 4-17). For the $\overline{C}D$ term, place a 1 in all squares with $\overline{C}D$ in their labels, that is, $\overline{A}\,\overline{B}\,\overline{C}D$, $\overline{A}B\overline{C}D$, $AB\overline{C}D$, $A\,\overline{B}\,\overline{C}D$. For the $A\overline{B}C$ term, place a 1 in all squares that have an $A\overline{B}C$ in their labels, that is, $A\overline{B}C\overline{D}$, $A\overline{B}CD$. For the $\overline{D}$ term, place a 1 in all squares that have a $\overline{D}$ in their labels, that is, all squares in the leftmost and rightmost columns.

FIGURE 4-17 Example 4-14.

	$\overline{C}\overline{D}$	$\overline{C}D$	CD	$C\overline{D}$
$\overline{A}\overline{B}$	1	1	0	1
$\overline{A}B$	1	1	0	1
AB	1	1	0	1
$A\overline{B}$	1	1	1	1

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

The K map is now filled and can be looped for simplification. Verify that proper looping produces $y = A\overline{B} + \overline{C} + \overline{D}$.

Don't-Care Conditions

Some logic circuits can be designed so that there are certain input conditions for which there are no specified output levels, usually because these input conditions will never occur. In other words, there will be certain combinations of input levels where we "don't care" whether the output is HIGH or LOW. This is illustrated in the truth table of Figure 4-18(a).

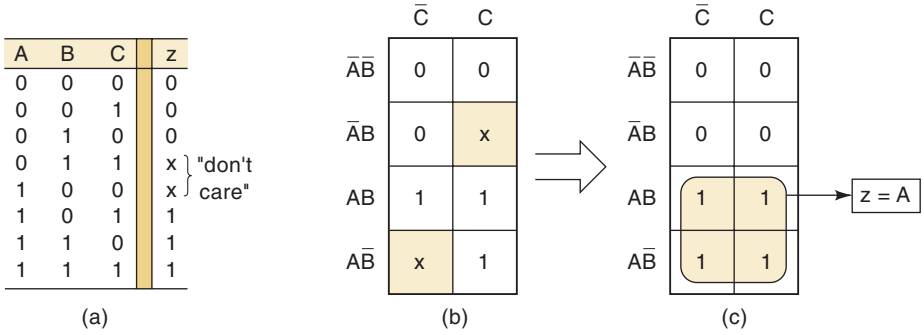


FIGURE 4-18 "Don't-care" conditions should be changed to 0 or 1 to produce K-map looping that yields the simplest expression.

Here the output z is not specified as either 0 or 1 for the conditions $A, B, C = 1, 0, 0$ and $A, B, C = 0, 1, 1$. Instead, an x is shown for these conditions. The x represents the **don't-care condition**. A don't-care condition can come about for several reasons, the most common being that in some situations certain input combinations can never occur, and so there is no specified output for these conditions.

A circuit designer is free to make the output for any don't-care condition either a 0 or a 1 to produce the simplest output expression. For example, the K map for this truth table is shown in Figure 4-18(b) with an x placed in the $A\overline{B}\,\overline{C}$ and $\overline{A}BC$ squares. The designer here would be wise to change the x in the $A\overline{B}\,\overline{C}$ square to a 1 and the x in the $\overline{A}BC$ square to a 0 because this would produce a quad that can be looped to produce $z = A$, as shown in Figure 4-18(c).

Whenever don't-care conditions occur, we must decide which x to change to 0 and which to 1 to produce the best K-map looping (i.e., the simplest expression). This decision is not always an easy one. Several end-of-chapter problems will provide practice in dealing with don't-care cases. Here's another example.

EXAMPLE 4-15

Let's design a logic circuit that controls an elevator door in a three-story building. The circuit in Figure 4-19(a) has four inputs. M is a logic signal that indicates when the elevator is moving ($M = 1$) or stopped ($M = 0$). $F1$, $F2$, and $F3$ are floor indicator signals that are normally LOW, and they go HIGH only when the elevator is positioned at the level of that particular floor. For example, when the elevator is lined up level with the second floor, $F2 = 1$ and $F1 = F3 = 0$. The circuit output is the *OPEN* signal, which is normally LOW and will go HIGH when the elevator door is to be opened.

We can fill in the truth table for the *OPEN* output [Figure 4-19(b)] as follows:

1. Because the elevator cannot be lined up with more than one floor at a time, only one of the floor inputs can be HIGH at any given time. This means that all those cases in the truth table where more than one floor

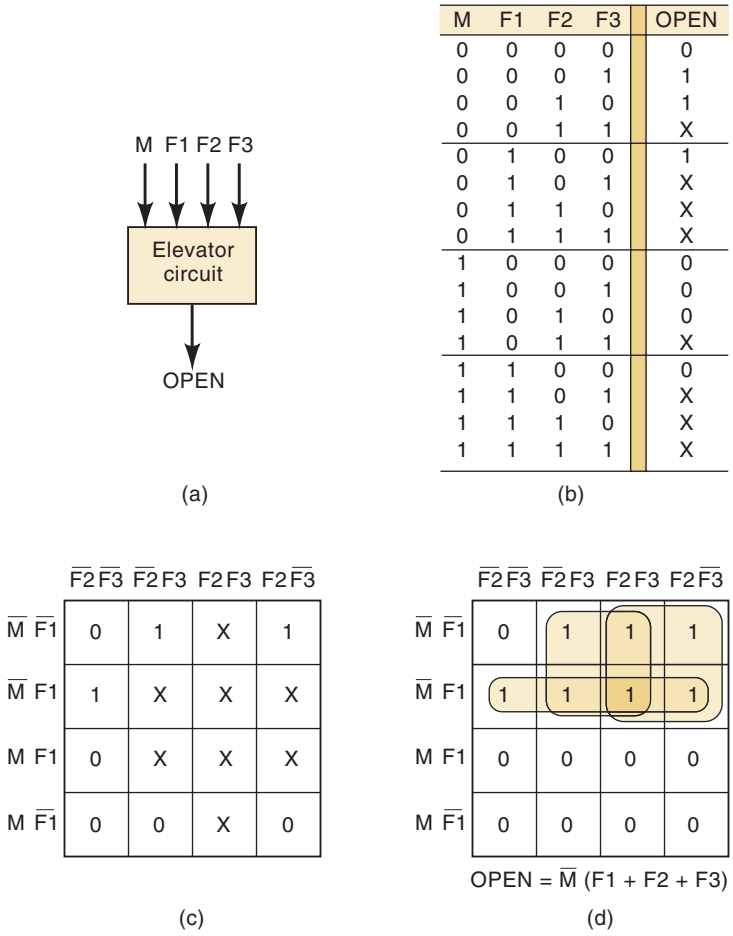


FIGURE 4-19 Example 4-15.

input is a 1 are don't-care conditions. We can place an *x* in the *OPEN* output column for those eight cases where more than one *F* input is 1.

2. Looking at the other eight cases, when $M = 1$ the elevator is moving, so *OPEN* must be a 0 because we do not want the elevator door to open. When $M = 0$ (elevator stopped) we want $OPEN = 1$ provided that one of the floor inputs is 1. When $M = 0$ and all floor inputs are 0, the elevator is stopped but is not properly lined up with any floor, so we want $OPEN = 0$ to keep the door closed.

The truth table is now complete and we can transfer its information to the K map in Figure 4-19(c). The map has only three 1s, but it has eight don't-cares. By changing four of these don't-care squares to 1s, we can produce quad loopings that contain the original 1s [Figure 4-19(d)]. This is the best we can do as far as minimizing the output expression. Verify that the loopings produce the *OPEN* output expression shown.

Summary

The K-map process has several advantages over the algebraic method. K mapping is a more orderly process with well-defined steps compared with the trial-and-error process sometimes used in algebraic simplification. K mapping usually requires fewer steps, especially for expressions containing many terms, and it always produces a minimum expression.

Nevertheless, some instructors prefer the algebraic method because it requires a thorough knowledge of Boolean algebra and is not simply a mechanical procedure. Each method has its advantages, and although most logic designers are adept at both, being proficient in one method is all that is necessary to produce acceptable results.

There are other, more complex techniques that designers use to minimize logic circuits with more than four inputs. These techniques are especially suited for circuits with large numbers of inputs where algebraic and K-mapping methods are not feasible. Most of these techniques can be translated into a computer program that will perform the minimization from input data that supply the truth table or the unsimplified expression.

REVIEW QUESTIONS

1. Use K mapping to obtain the expression of Example 4-7.
2. Use K mapping to obtain the expression of Example 4-8. This should emphasize the advantage of K mapping for expressions containing many terms.
3. Obtain the expression of Example 4-9 using a K map.
4. What is a don't-care condition?

4-6 EXCLUSIVE-OR AND EXCLUSIVE-NOR CIRCUITS

Two special logic circuits that occur quite often in digital systems are the *exclusive-OR* and *exclusive-NOR* circuits.

Exclusive-OR

Consider the logic circuit of Figure 4-20(a). The output expression of this circuit is

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

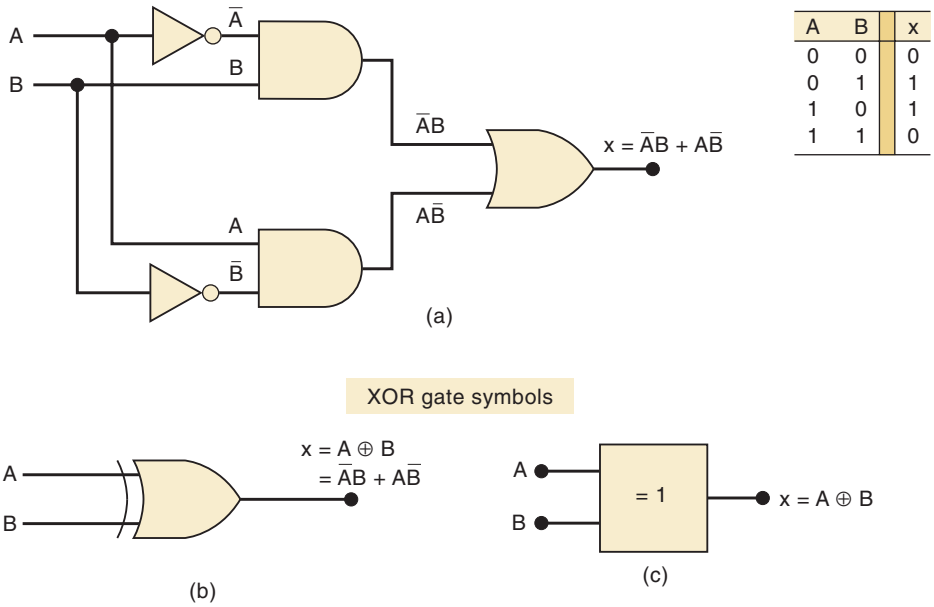


FIGURE 4-20
(a) Exclusive-OR circuit and truth table; (b) traditional XOR gate symbol; (c) IEEE/ANSI symbol for XOR gate.

The accompanying truth table shows that $x = 1$ for two cases: $A = 0, B = 1$ (the $\overline{A}B$ term) and $A = 1, B = 0$ (the $A\overline{B}$ term). In other words:

This circuit produces a HIGH output whenever the two inputs are at opposite levels.

This is the **exclusive-OR** circuit, which will hereafter be abbreviated **XOR**.

This particular combination of logic gates occurs quite often and is very useful in certain applications. In fact, the XOR circuit has been given a symbol of its own, shown in Figure 4-20(b). This symbol is assumed to contain all of the logic contained in the XOR circuit and therefore has the same logic expression and truth table. This XOR circuit is commonly referred to as an XOR *gate,* and we consider it as another type of logic gate. The IEEE/ANSI symbol for an XOR gate is shown in Figure 4-20(c). The dependency notation (= 1) inside the block indicates that the output will be active-HIGH *only* when a single input is HIGH.

An XOR gate has only *two* inputs; there are no three-input or four-input XOR gates. The two inputs are combined so that $x = \overline{A}B + A\overline{B}$. A shorthand way that is sometimes used to indicate the XOR output expression is

$$x = \mathrm{A} \oplus B$$

where the symbol $\oplus$ represents the XOR gate operation.

The characteristics of an XOR gate are summarized as follows:

1. It has only two inputs and its output is

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

2. Its output is *HIGH* only when the two inputs are at *different* levels.

Several ICs are available that contain XOR gates. Those listed below are *quad* XOR chips containing four XOR gates.

74LS86	Quad XOR (TTL family)
74C86	Quad XOR (CMOS family)
74HC86	Quad XOR (high-speed CMOS)

Exclusive-NOR

The **exclusive-NOR** circuit (abbreviated **XNOR**) operates completely opposite to the XOR circuit. Figure 4-21(a) shows an XNOR circuit and its accompanying truth table. The output expression is

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

which indicates along with the truth table that x will be 1 for two cases: $A = B = 1$ (the AB term) and $A = B = 0$ (the $\overline{A}\,\overline{B}$ term). In other words:

The XNOR produces a HIGH output whenever the two inputs are at the same level.

It should be apparent that the output of the XNOR circuit is the exact inverse of the output of the XOR circuit. The traditional symbol for an XNOR

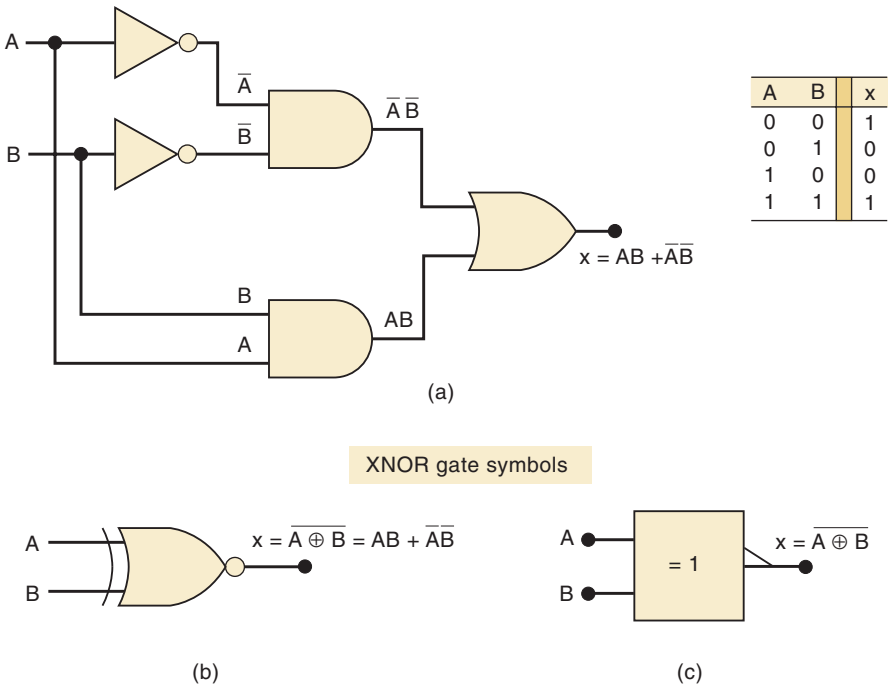


FIGURE 4-21
(a) Exclusive-NOR circuit; (b) traditional symbol for XNOR gate; (c) IEEE/ANSI symbol.

gate is obtained by simply adding a small circle at the output of the XOR symbol [Figure 4-21(b)]. The IEEE/ANSI symbol adds the small triangle on the output of the XOR symbol. Both symbols indicate an output that goes to its active-LOW state when *only one* input is HIGH.

The XNOR gate also has *only two* inputs, and it combines them so that its output is

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

A shorthand way to indicate the output expression of the XNOR is

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

which is simply the inverse of the XOR operation. The XNOR gate is summarized as follows:

1. It has only two inputs and its output is

$$x = AB + \overline{A}\,\overline{B} = \overline{A \oplus B}$$

2. Its output is HIGH only when the two inputs are at the *same* level.

Several ICs are available that contain XNOR gates. Those listed below are quad XNOR chips containing four XNOR gates.

74LS266	Quad XNOR (TTL family)
74C266	Quad XNOR (CMOS)
74HC266	Quad XNOR (high-speed CMOS)

Each of these XNOR chips, however, has special output circuitry that limits its use to special types of applications. Very often, a logic designer will obtain the XNOR function simply by connecting the output of an XOR to an INVERTER.

EXAMPLE 4-16

Determine the output waveform for the input waveforms given in Figure 4-22.

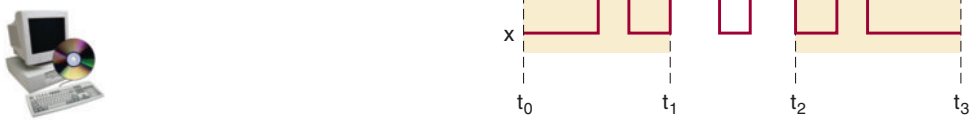


FIGURE 4-22
Example 4-16.

Solution

The output waveform is obtained using the fact that the XOR output will go HIGH only when its inputs are at different levels. The resulting output waveform reveals several interesting points:

1. The x waveform matches the A input waveform during those time intervals when $B = 0$. This occurs during the time intervals t_0 to t_1 and t_2 to t_3.
2. The x waveform is the *inverse* of the A input waveform during those time intervals when $B = 1$. This occurs during the interval t_1 to t_2.
3. These observations show that an XOR gate can be used as a *controlled INVERTER*; that is, one of its inputs can be used to control whether or not the signal at the other input will be inverted. This property will be useful in certain applications.

EXAMPLE 4-17

The notation x_1x_0 represents a two-bit binary number that can have any value (00, 01, 10, or 11); for example, when $x_1 = 1$ and $x_0 = 0$, the binary number is 10, and so on. Similarly, y_1y_0 represents another two-bit binary number. Design a logic circuit, using x_1, x_0, y_1, and y_0 inputs, whose output will be HIGH only when the two binary numbers x_1x_0 and y_1y_0 are *equal*.

Solution

The first step is to construct a truth table for the 16 input conditions (Table 4-4). The output z must be HIGH whenever the x_1x_0 values match the y_1y_0 values; that is, whenever $x_1 = y_1$ and $x_0 = y_0$. The table shows that there are four such cases. We could now continue with the normal procedure, which would be to obtain a sum-of-products expression for z, attempt to simplify it, and then implement the result. However, the nature of this problem makes it ideally suited for implementation using XNOR gates, and a little thought

TABLE 4-4

x_1	x_0	y_1	y_0	z (Output)
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

will produce a simple solution with minimum work. Refer to Figure 4-23; in this logic diagram, x_1 and y_1 are fed to one XNOR gate, and x_0 and y_0 are fed to another XNOR gate. The output of each XNOR will be HIGH only when its inputs are equal. Thus, for $x_0 = y_0$ and $x_1 = y_1$, both XNOR outputs will be HIGH. This is the condition we are looking for because it means that the two two-bit numbers are equal. The AND gate output will be HIGH only for this case, thereby producing the desired output.

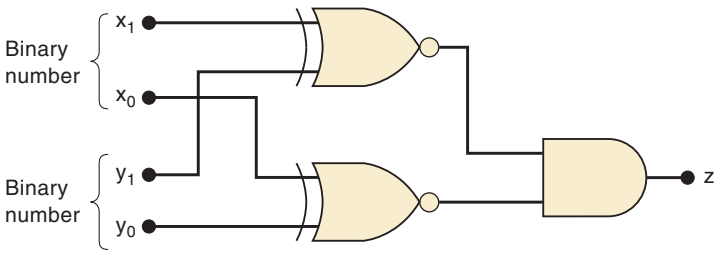


FIGURE 4-23 Circuit for detecting equality of two two-bit binary numbers.

EXAMPLE 4-18

When simplifying the expression for the output of a combinational logic circuit, you may encounter the XOR or XNOR operations as you are factoring. This will often lead to the use of XOR or XNOR gates in the implementation of the final circuit. To illustrate, simplify the circuit of Figure 4-24(a).

Solution

The unsimplified expression for the circuit is obtained as

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

We can factor AD from the first two terms:

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

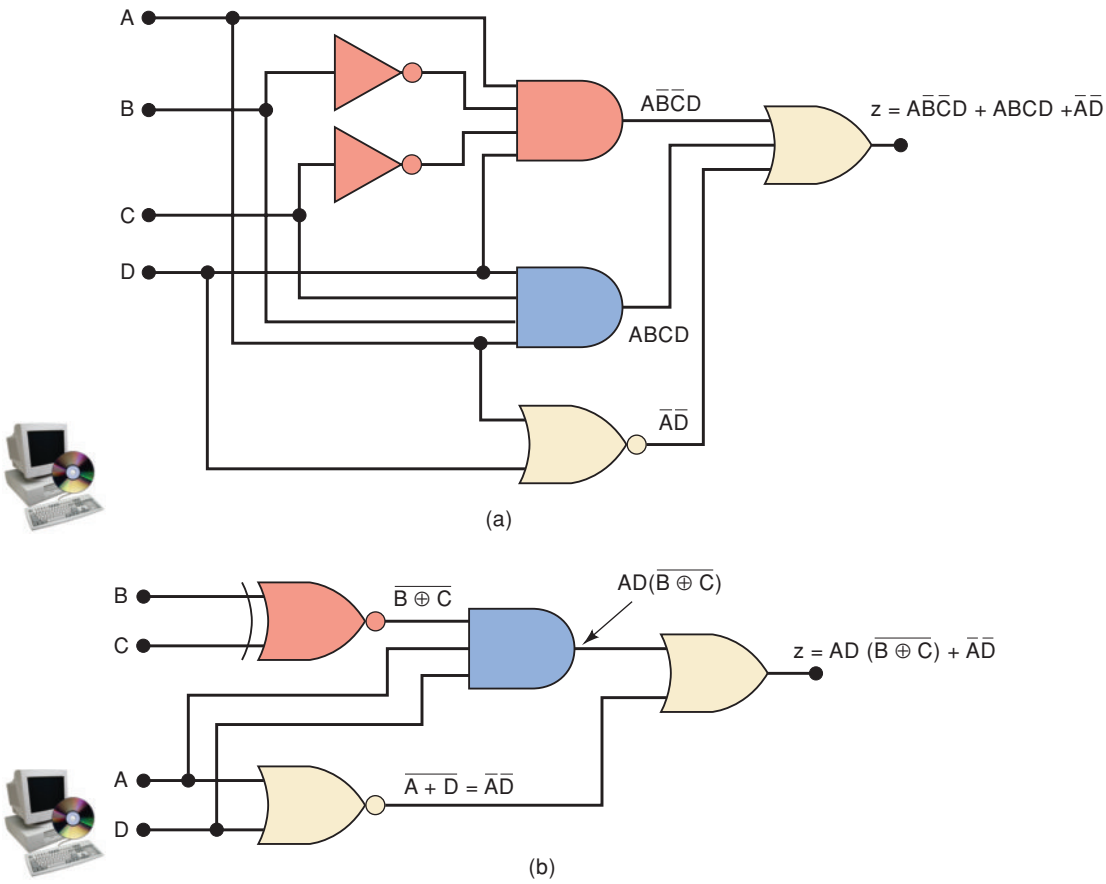


FIGURE 4-24 Example 4-18, showing how an XNOR gate may be used to simplify circuit implementation.

At first glance, you might think that the expression in parentheses can be replaced by 1. But that would be true only if it were $BC + \overline{BC}$. You should recognize the expression in parentheses as the XNOR combination of B and C. This fact can be used to reimplement the circuit as shown in Figure 4-24(b). This circuit is much simpler than the original because it uses gates with fewer inputs and two INVERTERs have been eliminated.

REVIEW QUESTIONS

1. Use Boolean algebra to prove that the XNOR output expression is the exact inverse of the XOR output expression.
2. What is the output of an XNOR gate when a logic signal and its exact inverse are connected to its inputs?
3. A logic designer needs an INVERTER, and all that is available is one XOR gate from a 74HC86 chip. Does he need another chip?

4-7 PARITY GENERATOR AND CHECKER

In Chapter 2, we saw that a transmitter can attach a parity bit to a set of data bits before transmitting the data bits to a receiver. We also saw how this allows the receiver to detect any single-bit errors that may have occurred during the

transmission. Figure 4-25 shows an example of one type of logic circuitry that is used for **parity generation** and **parity checking**. This particular example uses a group of four bits as the data to be transmitted, and it uses an even-parity bit. It can be readily adapted to use odd parity and any number of bits.

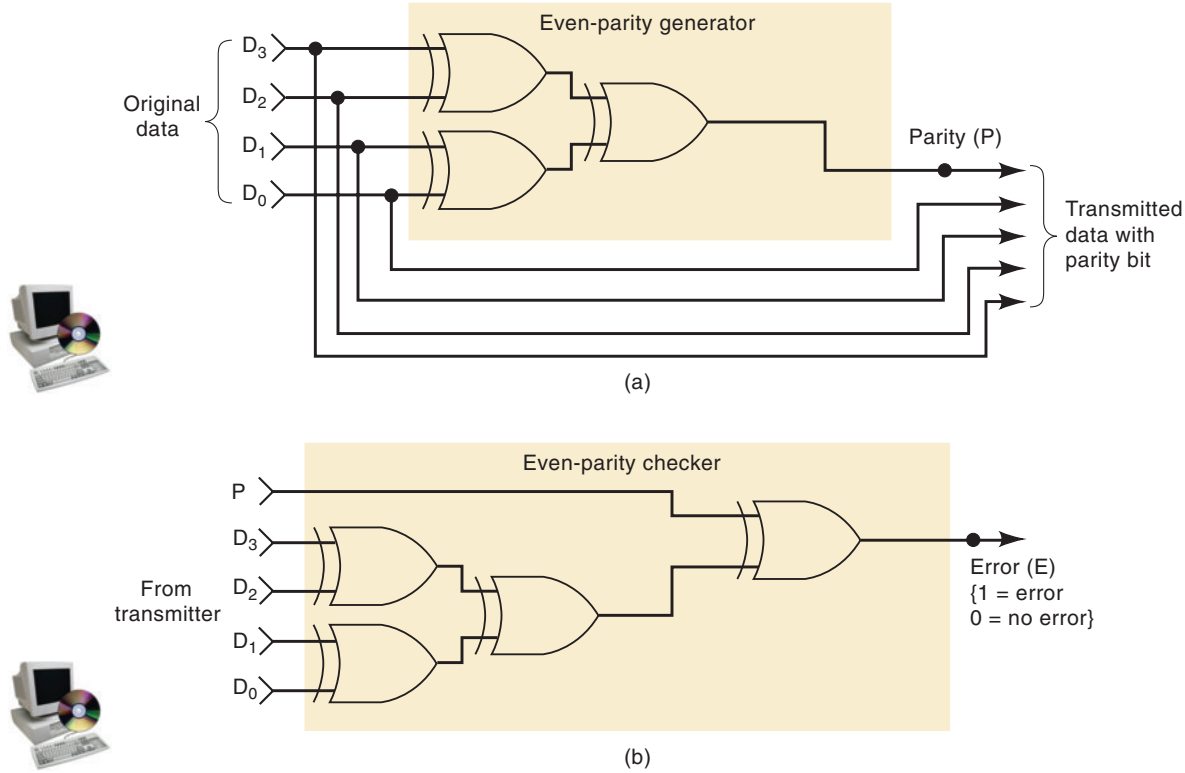


FIGURE 4-25 XOR gates used to implement (a) the parity generator and (b) the parity checker for an even-parity system.

In Figure 4-25(a), the set of data to be transmitted is applied to the parity-generator circuit, which produces the even-parity bit, *P*, at its output. This parity bit is transmitted to the receiver along with the original data bits, making a total of five bits. In Figure 4-25(b), these five bits (data + parity) enter the receiver's parity-checker circuit, which produces an error output, *E*, that indicates whether or not a single-bit error has occurred.

It should not be too surprising that both of these circuits employ XOR gates when we consider that a single XOR gate operates so that it produces a 1 output if an odd number of its inputs are 1, and a 0 output if an even number of its inputs are 1.

EXAMPLE 4-19

Determine the parity generator's output for each of the following sets of input data, $D_3D_2D_1D_0$: (a) 0111; (b) 1001; (c) 0000; (d) 0100. Refer to Figure 4-25(a).

Solution

For each case, apply the data levels to the parity-generator inputs and trace them through each gate to the *P* output. The results are: (a) 1; (b) 0; (c) 0; and (d) 1. Note that *P* is a 1 only when the original data contain an odd number of 1s. Thus, the total number of 1s sent to the receiver (data + parity) will be even.

EXAMPLE 4-20

Determine the parity checker's output [see Figure 4-25(b)] for each of the following sets of data from the transmitter:

	P	D_3	D_2	D_1	D_0
(a)	0	1	0	1	0
(b)	1	1	1	1	0
(c)	1	1	1	1	1
(d)	1	0	0	0	0

Solution

For each case, apply these levels to the parity-checker inputs and trace them through to the E output. The results are: (a) 0; (b) 0; (c) 1; (d) 1. Note that a 1 is produced at E only when an odd number of 1s appears in the inputs to the parity checker. This indicates that an error has occurred because even parity is being used.

4-8 ENABLE/DISABLE CIRCUITS

Each of the basic logic gates can be used to control the passage of an input logic signal through to the output. This is depicted in Figure 4-26, where a logic signal, A, is applied to one input of each of the basic logic gates. The

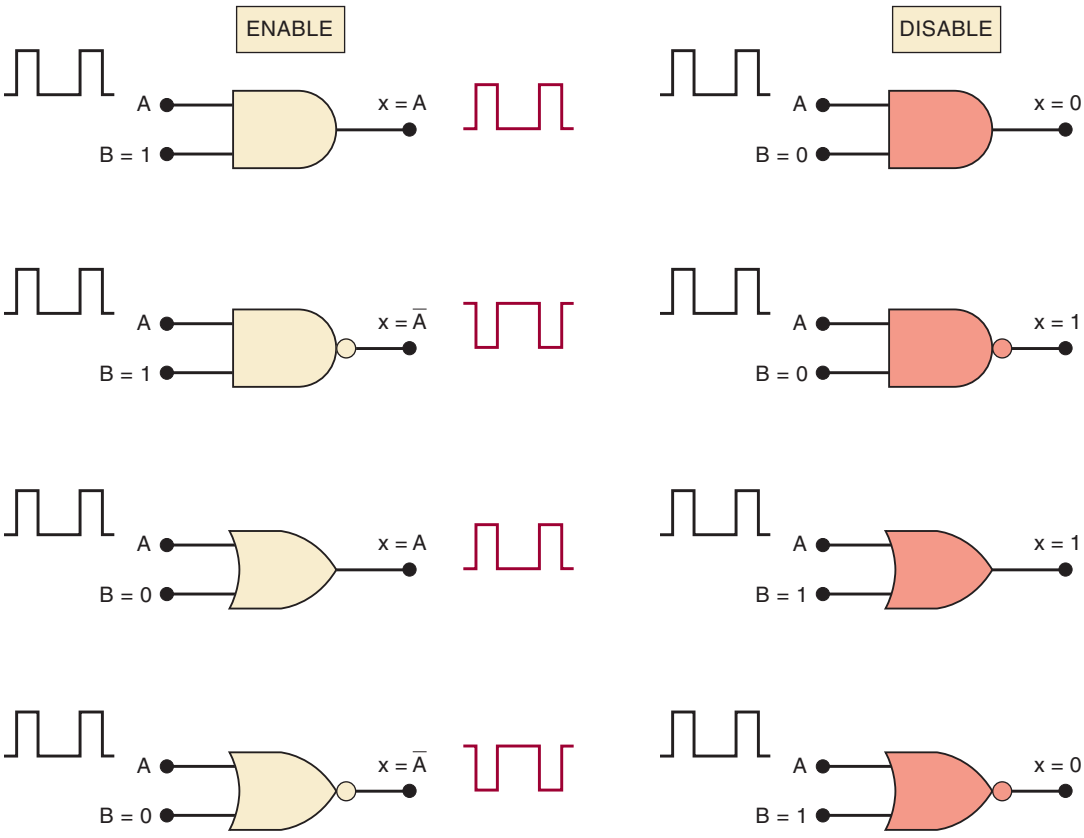


FIGURE 4-26 Four basic gates can either enable or disable the passage of an input signal, A, under control of the logic level at control input B.

other input of each gate is the control input, *B*. The logic level at this control input will determine whether the input signal is **enabled** to reach the output or **disabled** from reaching the output. This controlling action is why these circuits came to be called *gates.*

Examine Figure 4-26 and you should notice that when the noninverting gates (AND, OR) are enabled, the output will follow the *A* signal exactly. Conversely, when the inverting gates (NAND, NOR) are enabled, the output will be the exact inverse of the *A* signal.

Also notice in the figure that AND and NOR gates produce a constant LOW output when they are in the disabled condition. Conversely, the NAND and OR gates produce a constant HIGH output in the disabled condition.

There will be many situations in digital-circuit design where the passage of a logic signal is to be enabled or disabled, depending on conditions present at one or more control inputs. Several are shown in the following examples.

EXAMPLE 4-21

Design a logic circuit that will allow a signal to pass to the output only when control inputs *B* and *C* are both HIGH; otherwise, the output will stay LOW.

Solution

An AND gate should be used because the signal is to be passed without inversion, and the disable output condition is a LOW. Because the enable condition must occur only when $B = C = 1$, a three-input AND gate is used, as shown in Figure 4-27(a).

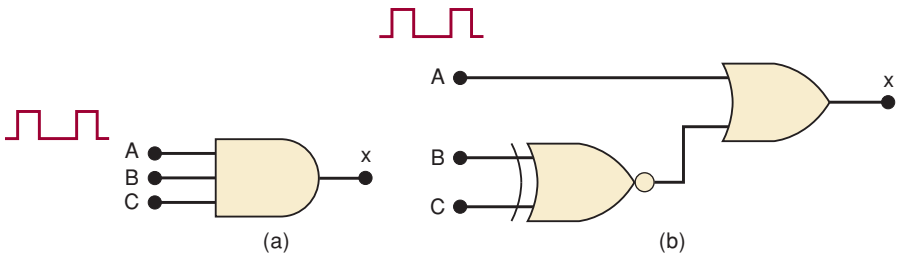


FIGURE 4-27 Examples 4-21 and 4-22.

EXAMPLE 4-22

Design a logic circuit that allows a signal to pass to the output only when one, but not both, of the control inputs are HIGH; otherwise, the output will stay HIGH.

Solution

The result is drawn in Figure 4-27(b). An OR gate is used because we want the output disable condition to be a HIGH, and we do not want to invert the signal. Control inputs *B* and *C* are combined in an XNOR gate. When *B* and *C* are different, the XNOR sends a LOW to enable the OR gate. When *B* and *C* are the same, the XNOR sends a HIGH to disable the OR gate.

EXAMPLE 4-23

Design a logic circuit with input signal *A*, control input *B*, and outputs *X* and *Y* to operate as follows:

1. When $B = 1$, output *X* will follow input *A*, and output *Y* will be 0.
2. When $B = 0$, output *X* will be 0, and output *Y* will follow input *A*.

Solution

The two outputs will be 0 when they are disabled and will follow the input signal when they are enabled. Thus, an AND gate should be used for each output. Because X is to be enabled when $B = 1$, its AND gate must be controlled by B, as shown in Figure 4-28. Because Y is to be enabled when $B = 0$, its AND gate is controlled by $\overline{B}$. The circuit in Figure 4-28 is called a *pulse-steering circuit* because it steers the input pulse to one output or the other, depending on B.

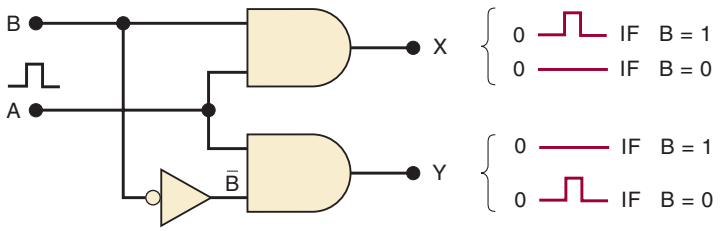


FIGURE 4-28 Example 4-23.

REVIEW QUESTIONS

1. Design a logic circuit with three inputs A, B, C and an output that goes LOW only when A is HIGH while B and C are different.
2. Which logic gates produce a 1 output in the disabled state?
3. Which logic gates pass the inverse of the input signal when they are enabled?

4-9 BASIC CHARACTERISTICS OF DIGITAL ICs

Digital ICs are a collection of resistors, diodes, and transistors fabricated on a single piece of semiconductor material (usually silicon) called a *substrate,* which is commonly referred to as a *chip.* The chip is enclosed in a protective plastic or ceramic package from which pins extend for connecting the IC to other devices. One of the more common types of package is the **dual-in-line package (DIP)**, shown in Figure 4-29(a), so called because it contains two parallel rows of pins. The pins are numbered counterclockwise when viewed from the top of the package with respect to an identifying notch or dot at one end of the package [see Figure 4-29(b)]. The DIP shown here is a 14-pin package that measures 0.75 in. by 0.25 in.; 16-, 20-, 24-, 28-, 40-, and 64-pin packages are also used.

Figure 4-29(c) shows that the actual silicon chip is much smaller than the DIP; typically, it might be a 0.05-in. square. The silicon chip is connected to the pins of the DIP by very fine wires (1-mil diameter).

The DIP is probably the most common digital IC package found in older digital equipment, but other types are becoming more and more popular. The IC shown in Figure 4-29(d) is only one of the many packages common to modern digital circuits. This particular package uses J-shaped leads that curl under the IC. We will take a look at some of these other types of IC packages in Chapter 8.

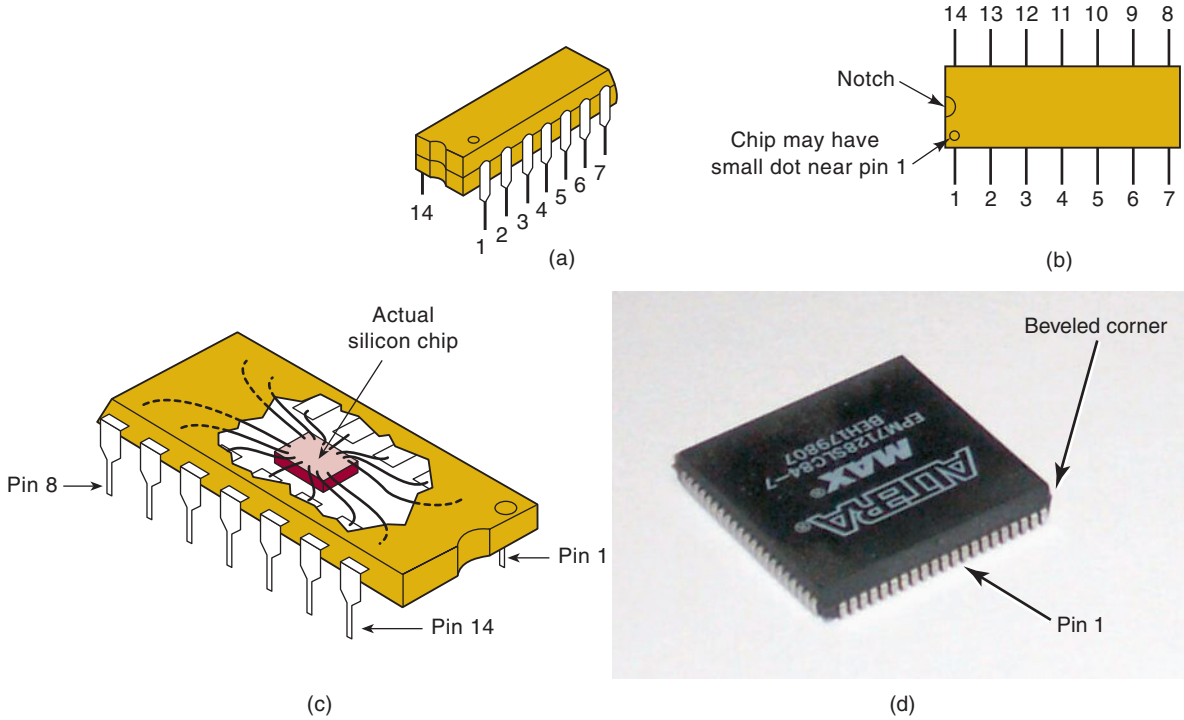


FIGURE 4-29 (a) Dual-in-line package (DIP); (b) top view; (c) actual silicon chip is much smaller than the protective package; (d) PLCC package.

Digital ICs are often categorized according to their circuit complexity as measured by the number of equivalent logic gates on the substrate. There are currently six levels of complexity that are commonly defined as shown in Table 4-5.

TABLE 4-5

Complexity	Gates per Chip
Small-scale integration (SSI)	Fewer than 12
Medium-scale integration (MSI)	12 to 99
Large-scale integration (LSI)	100 to 9999
Very large-scale integration (VLSI)	10,000 to 99,999
Ultra large-scale integration (ULSI)	100,000 to 999,999
Giga-scale integration (GSI)	1,000,000 or more

All of the specific ICs referred to in Chapter 3 and this chapter are **SSI** chips having a small number of gates. In modern digital systems, medium-scale integration **(MSI)** and large-scale integration devices **(LSI, VLSI, ULSI, GSI)** perform most of the functions that once required several circuit boards full of SSI devices. However, SSI chips are still used as the "interface," or "glue," between these more complex chips. The small-scale ICs also offer an excellent way to learn the basic building blocks of digital systems. Consequently, many laboratory-based courses use these ICs to build and test small projects.

The industrial world of digital electronics has now turned to programmable logic devices (PLDs) to implement a digital system of any significant size. Some simple PLDs are available in DIP packages, but the more complex

programmable logic devices require many more pins than are available in DIPs. Larger integrated circuits that may need to be removed from a circuit and replaced are typically manufactured in a plastic leaded chip carrier (PLCC) package. Figure 4-29(d) shows the Altera EPM 7128SLC84 in a PLCC package, which is a very popular PLD used in many educational laboratories. The key features of this chip are more pins, closer spacing, and pins around the entire periphery. Notice that pin 1 is not "on the corner" like the DIP but rather at the middle of the top of the package.

Bipolar and Unipolar Digital ICs

Digital ICs can also be categorized according to the principal type of electronic component used in their circuitry. *Bipolar ICs* are made using the bipolar junction transistor (NPN and PNP) as their main circuit element. *Unipolar ICs* use the unipolar field-effect transistor (P-channel and N-channel MOSFETs) as their main element.

The **transistor-transistor logic (TTL)** family has been the major family of bipolar digital ICs for over 30 years. The standard 74 series was the first series of TTL ICs. It is no longer used in new designs, having been replaced by several higher-performance TTL series, but its basic circuit arrangement forms the foundation for all the TTL series ICs. This circuit arrangement is shown in Figure 4-30(a) for the standard TTL INVERTER. Notice that the circuit contains several bipolar transistors as the main circuit element.

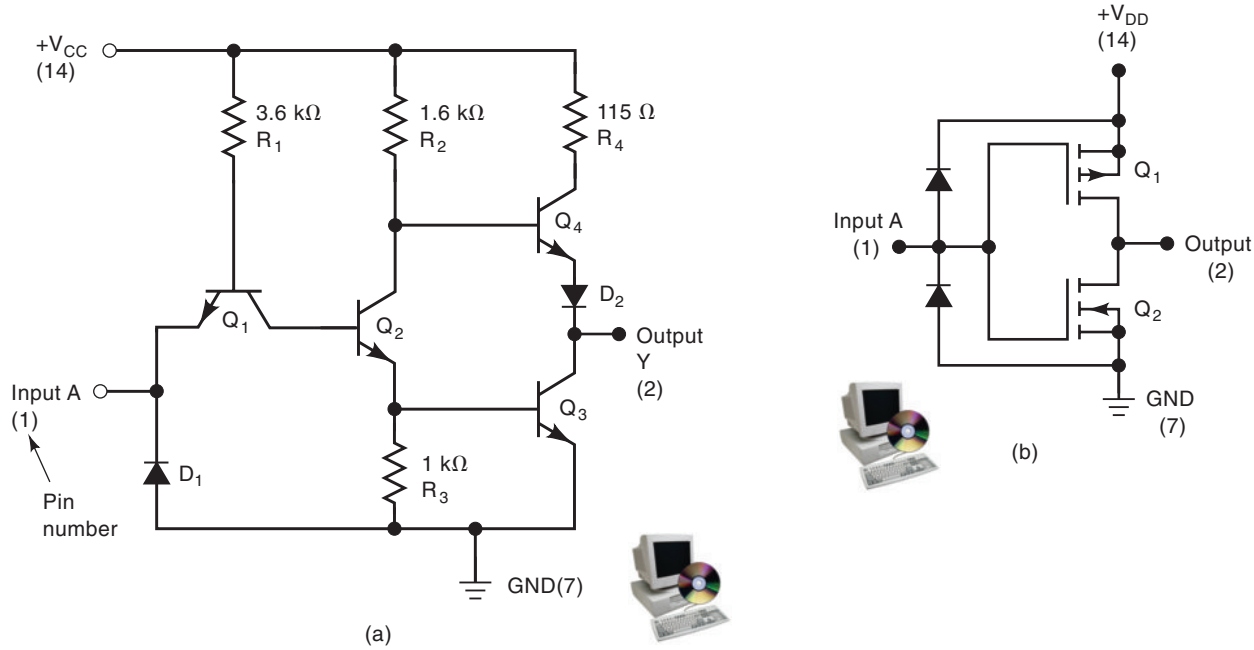


FIGURE 4-30 (a) TTL INVERTER circuit; (b) CMOS INVERTER circuit. Pin numbers are given in parentheses.

TTL had been the leading IC family in the SSI and MSI categories up until the last 12 or so years. Since then, its leading position has been challenged by the CMOS family, which has gradually displaced TTL from that position. The **complementary metal-oxide semiconductor (CMOS)** family belongs to the class of unipolar digital ICs because it uses P- and N-channel MOSFETs as the main circuit elements. Figure 4-30(b) is a standard CMOS INVERTER circuit. If we compare the TTL and CMOS circuits in Figure 4-30, it is apparent

that the CMOS version uses fewer components. This is one of the main advantages of CMOS over TTL.

Because of the simplicity and compactness as well as some other superior attributes of CMOS, the modern large-scale ICs are manufactured primarily using CMOS technology. Teaching laboratories that use SSI and MSI devices often use TTL due to its durability, although some use CMOS as well. Chapter 8 will provide a comprehensive study of the circuitry and characteristics of TTL and CMOS. For now, we need to look at only a few of their basic characteristics so that we can talk about troubleshooting simple combinational circuits.

TTL Family

The TTL logic family actually consists of several subfamilies or series. Table 4-6 lists the name of each TTL series together with the prefix designation used to identify different ICs as being part of that series. For example, ICs that are part of the standard TTL series have an identification number that starts with 74. The 7402, 7438, and 74123 are all ICs in this series. Likewise, ICs that are part of the low-power Schottky TTL series have an identification number that starts with 74LS. The 74LS02, 74LS38, and 74LS123 are examples of devices in the 74LS series.

TABLE 4-6 Various series within the TTL logic family.

TTL Series	Prefix	Example IC
Standard TTL	74	7404 (hex INVERTER)
Schottky TTL	74S	74S04 (hex INVERTER)
Low-power Schottky TTL	74LS	74LS04 (hex INVERTER)
Advanced Schottky TTL	74AS	74AS04 (hex INVERTER)
Advanced low-power Schottky TTL	74ALS	74ALS04 (hex INVERTER)

The principal differences in the various TTL series have to do with their electrical characteristics such as power dissipation and switching speed. They do not differ in the pin layout or logic operations performed by the circuits on the chip. For example, the 7404, 74S04, 74LS04, 74AS04, and 74ALS04 are all hex-INVERTER ICs, each containing *six* INVERTERs on a single chip.

CMOS Family

Several CMOS series are available, and some of these are listed in Table 4-7. The 4000 series is the oldest CMOS series. This series contains many of the same logic functions as the TTL family but was not designed to be *pin-compatible* with TTL devices. For example, the 4001 quad NOR chip contains four two-input NOR gates, as does the TTL 7402 chip, but the gate inputs and outputs on the CMOS chip will not have the same pin numbers as the corresponding signals on the TTL chip.

The 74C, 74HC, 74HCT, 74AC, and 74ACT series are newer CMOS series. The first three are pin-compatible with correspondingly numbered TTL devices. For example, the 74C02, 74HC02, and 74HCT02 have the same pin layout as the 7402, 74LS02, and so on. The 74HC and 74HCT series operate at a higher speed than 74C devices. The 74HCT series is designed to be *electrically compatible* with TTL devices; that is, a 74HCT integrated circuit can be connected directly to TTL devices without any interfacing circuitry. The 74AC and 74ACT series are advanced-performance ICs. Neither is pin-compatible with

TABLE 4-7 Various series within the CMOS logic family.

CMOS Series	Prefix	Example IC
Metal-gate CMOS	40	4001 (quad NOR gates)
Metal-gate, pin-compatible with TTL	74C	74C02 (quad NOR gates)
Silicon-gate, pin-compatible with TTL, high-speed	74HC	74HC02 (quad NOR gates)
Silicon-gate, high-speed, pin-compatible and electrically compatible with TTL	74HCT	74HCT02 (quad NOR gates)
Advanced-performance CMOS, not pin-compatible or electrically compatible with TTL	74AC	74AC02 (quad NOR)
Advanced-performance CMOS, not pin-compatible with TTL, but electrically compatible with TTL	74ACT	74ACT02 (quad NOR)

TTL. The 74ACT devices are electrically compatible with TTL. We explore the various TTL and CMOS series in greater detail in Chapter 8.

Power and Ground

To use digital ICs, it is necessary to make the proper connections to the IC pins. The most important connections are *dc power* and *ground*. These are required for the circuits on the chip to operate correctly. In Figure 4-30, you can see that both the TTL and the CMOS circuits have a dc power supply voltage connected to one of their pins, and ground to another. The power supply pin is labeled V_{CC} for the TTL circuit, and V_{DD} for the CMOS circuit. Many of the newer CMOS integrated circuits that are designed to be compatible with TTL integrated circuits also use the V_{CC} designation as their power pin.

If either the power or the ground connection is not made to the IC, the logic gates on the chip will not respond properly to the logic inputs, and the gates will not produce the expected output logic levels.

Logic-Level Voltage Ranges

For TTL devices, V_{CC} is nominally +5 V. For CMOS integrated circuits, V_{DD} can range from +3 to +18 V, although +5 V is most often used when CMOS integrated circuits are used in the same circuit with TTL integrated circuits.

For standard TTL devices, the acceptable input voltage ranges for the logic 0 and logic 1 levels are defined as shown in Figure 4-31(a). A logic 0 is any voltage in the range from 0 to 0.8 V; a logic 1 is any voltage from 2 to 5 V. Voltages that are not in either of these ranges are said to be **indeterminate** and should not be used as inputs to any TTL device. The IC manufacturers cannot guarantee how a TTL circuit will respond to input levels that are in the indeterminate range (between 0.8 and 2.0 V).

The logic input voltage ranges for CMOS integrated circuits operating with $V_{DD} = +5$ V are shown in Figure 4-31(b). Voltages between 0 and 1.5 V are defined as a logic 0, and voltages from 3.5 to 5 V are defined as a logic 1. The indeterminate range includes voltages between 1.5 and 3.5 V.

Unconnected (Floating) Inputs

What happens when the input to a digital IC is left unconnected? An unconnected input is often called a **floating** input. The answer to this question will be different for TTL and CMOS.

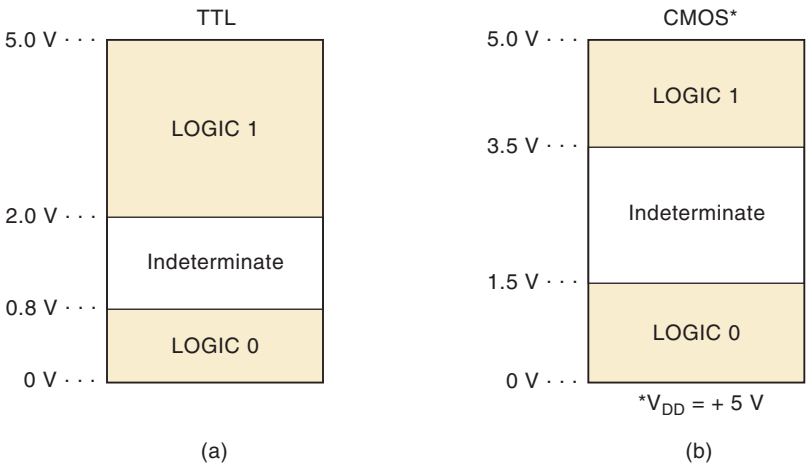


FIGURE 4-31 Logic-level input voltage ranges for (a) TTL and (b) CMOS digital ICs.

A floating TTL input acts just like a logic 1. In other words, the IC will respond as if the input had a logic HIGH level applied to it. This characteristic is often used when testing a TTL circuit. A lazy technician might leave certain inputs unconnected instead of connecting them to a logic HIGH. Although this is logically correct, it is not a recommended practice, especially in final circuit designs, because the floating TTL input is extremely susceptible to picking up noise signals that will probably adversely affect the device's operation.

A floating input on some TTL gates will measure a dc level of between 1.4 and 1.8 V when checked with a VOM or an oscilloscope. Even though this is in the indeterminate range for TTL, it will produce the same response as a logic 1. Being aware of this characteristic of a floating TTL input can be valuable when troubleshooting TTL circuits.

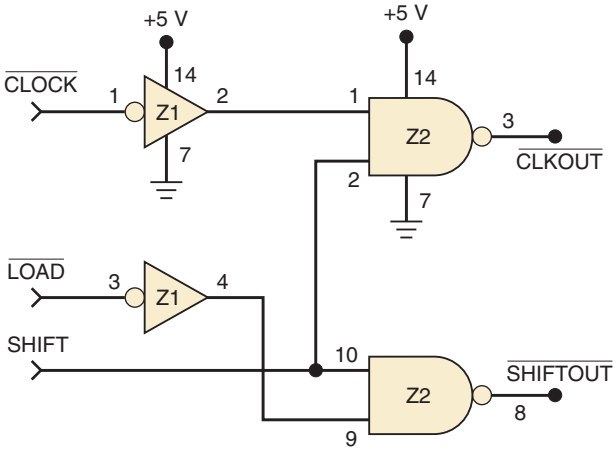
If a CMOS input is left floating, it may have disastrous results. The IC may become overheated and eventually destroy itself. For this reason all inputs to a CMOS integrated circuit must be connected to a LOW or a HIGH level or to the output of another IC. A floating CMOS input will not measure as a specific dc voltage but will fluctuate randomly as it picks up noise. Thus, it does not act as logic 1 or logic 0, and so its effect on the output is unpredictable. Sometimes the output will oscillate as a result of the noise picked up by the floating input.

Many of the more complex CMOS ICs have circuitry built into the inputs, which reduces the likelihood of any destructive reaction to an open input. With this circuitry, it is not necessary to ground each unused pin on these large ICs when experimenting. It is still good practice, however, to tie unused inputs to HIGH or LOW (whichever is appropriate) in the final circuit implementation.

Logic-Circuit Connection Diagrams

A connection diagram shows *all* electrical connections, pin numbers, IC numbers, component values, signal names, and power supply voltages. Figure 4-32 shows a typical connection diagram for a simple logic circuit. Examine it carefully and note the following important points:

1. The circuit uses logic gates from two different ICs. The two INVERTERs are part of a 74HC04 chip that has been given the designation Z1. The 74HC04 contains six INVERTERs; two of them are used in this circuit, and each is labeled as being part of chip Z1. Similarly, the two NAND gates are part of a 74HC00 chip that contains four NAND gates. All of



IC	Type
Z1	74HC04 hex inverter
Z2	74HC00 quad nand

FIGURE 4-32 Typical logic-circuit connection diagram.

the gates on this chip are designated with the label Z2. By numbering each gate as Z1, Z2, Z3, and so on, we can keep track of which gate is part of which chip. This is especially valuable in more complex circuits containing many ICs with several gates per chip.

2. Each gate input and output pin number is indicated on the diagram. These pin numbers and the IC labels are used to reference easily any point in the circuit. For example, Z1 pin 2 refers to the output pin of the top INVERTER. Similarly, we can say that Z1 pin 4 is connected to Z2 pin 9.
3. The power and ground connections to each IC (not each gate) are shown on the diagram. For example, Z1 pin 14 is connected to +5 V, and Z1 pin 7 is connected to ground. These connections provide power to *all* of the six INVERTERs that are part of Z1.
4. For the circuit contained in Figure 4-32, the signals that are inputs are on the left. The signals that are outputs are on the right. The bar over the signal name indicates that the signal is active when LOW. The bubbles are positioned on the diagram symbols also to indicate the active-LOW state. Each signal in this case is obviously a single bit.
5. Signals are defined graphically in Figure 4-32 as inputs and outputs, and the relationship between them (the operation of the circuit) is described graphically using interconnected logic symbols.

Manufacturers of electronic equipment generally supply detailed schematics that use a format similar to that in Figure 4-32. These connection diagrams are a virtual necessity when troubleshooting a faulty circuit. We have chosen to identify individual ICs as Z1, Z2, Z3, and so on. Other designations that are commonly used are IC1, IC2, IC3, and so on, and U1, U2, U3, and so on.

Personal computers with schematic diagram software can be used to draw logic circuits. Computer applications that can interpret these graphic symbols and signal connections and can translate them into logical relationships are often called schematic capture tools. The Altera MAX+PLUS development system for programmable logic allows the user to enter graphic design files (.gdf) using schematic capture techniques. Thus, designing the circuit is as easy as drawing the schematic diagram on the computer screen. Notice that in Figure 4-33 there are no pin numbers or chip designations on the logic symbols. The circuits will not be implemented using actual SSI or MSI chips, but rather the equivalent logic functionality will be "programmed" into a PLD. We will explain this further at a later point in this chapter.

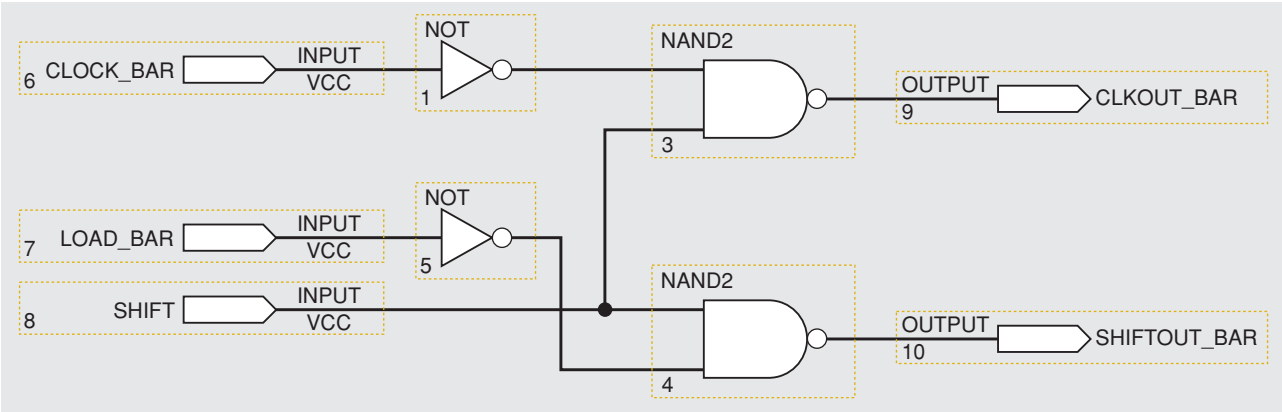


FIGURE 4-33 Logic diagram using schematic capture.

REVIEW QUESTIONS

1. What is the most common type of digital IC package?
2. Name the six common categories of digital ICs according to complexity.
3. *True or false:* A 74S74 chip will contain the same logic and pin layout as the 74LS74.
4. *True or false:* A 74HC74 chip will contain the same logic and pin layout as the 74AS74.
5. Which CMOS series are not pin-compatible with TTL?
6. What is the acceptable input voltage range of a logic 0 for TTL? What is it for a logic 1?
7. Repeat question 6 for CMOS operating at $V_{DD} = 5$ V.
8. How does a TTL integrated circuit respond to a floating input?
9. How does a CMOS integrated circuit respond to a floating input?
10. Which CMOS series can be connected directly to TTL with no interfacing circuitry?
11. What is the purpose of pin numbers on a logic circuit connection diagram?
12. What are the key similarities of graphic design files used for programmable logic and traditional logic circuit connection diagrams?

4-10 TROUBLESHOOTING DIGITAL SYSTEMS

There are three basic steps in fixing a digital circuit or system that has a fault (failure):

1. *Fault detection.* Observe the circuit/system operation and compare it with the expected correct operation.
2. *Fault isolation.* Perform tests and make measurements to isolate the fault.
3. *Fault correction.* Replace the faulty component, repair the faulty connection, remove the short, and so on.

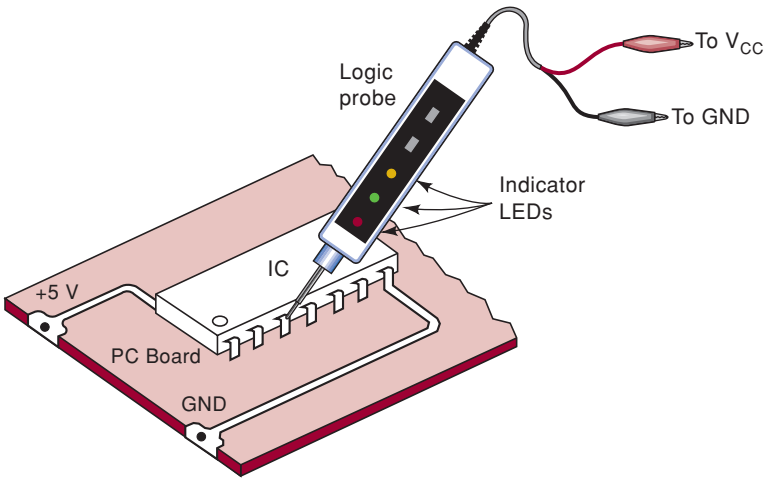
Although these steps may seem relatively apparent and straightforward, the actual troubleshooting procedure that is followed is highly dependent on the

type and complexity of the circuitry, and on the kinds of troubleshooting tools and documentation that are available.

Good troubleshooting techniques can be learned only in a laboratory environment through experimentation and actual troubleshooting of faulty circuits and systems. There is absolutely no better way to become an effective troubleshooter than to do as much troubleshooting as possible, and no amount of textbook reading can provide that kind of experience. We can, however, help you to develop the analytical skills that are the most essential part of effective troubleshooting. We will describe the types of faults that are common to systems that are made primarily from digital ICs and we will tell you how to recognize them. We will then present typical case studies to illustrate the analytical processes involved in troubleshooting. In addition, there will be end-of-chapter problems to provide you with the opportunity to go through these analytical processes to reach conclusions about faulty digital circuits.

For the troubleshooting discussions and exercises we will be doing in this book, it will be assumed that the troubleshooting technician has the basic troubleshooting tools available: *logic probe, oscilloscope, logic pulser.* Of course, the most important and effective troubleshooting tool is the technician's brain, and that's the tool we are hoping to develop by presenting troubleshooting principles and techniques, examples and problems, here and in the following chapters.

In the next three sections on troubleshooting, we will use only our brain and a **logic probe** such as the one illustrated in Figure 4-34. The logic probe has a pointy metal tip that is touched to the specific point you want to test. Here, it is shown probing (touching) pin 3 of an IC. It can also be touched to a printed circuit board trace, an uninsulated wire, a connector pin, a lead on a discrete component such as a transistor, or any other conducting point in a circuit. The logic level that is present at the probe tip will be indicated by the status of the indicator LEDs in the probe. The four possibilities are given in the table of Figure 4-34. Note that an *indeterminate* logic level produces no indicator light. This includes the condition where the probe tip is touched to a point in a circuit that is open or floating—that is, not connected to any source of voltage. This type of probe also offers a yellow LED to indicate the presence of a pulse train. Any transitions (LOW to HIGH or HIGH to LOW) will cause the yellow LED to flash on for a fraction of a second and then turn off. If the transitions are occurring frequently, the LED will continue to flash



LEDs			
Red	Green	Yellow	Logic Condition
OFF	ON	OFF	LOW
ON	OFF	OFF	HIGH
OFF	OFF	OFF	INDETERMINATE*
X	X	FLASHING	PULSING

* Includes open or floating condition

FIGURE 4-34 A logic probe is used to monitor the logic level activity at an IC pin or any other accessible point in a logic circuit.

at around 3 Hz. By observing the green and red LEDs along with the flashing yellow, you can tell whether the signal is mostly HIGH or mostly LOW.

4-11 INTERNAL DIGITAL IC FAULTS

The most common internal failures of digital ICs are:

1. Malfunction in the internal circuitry
2. Inputs or outputs shorted to ground or V_{CC}
3. Inputs or outputs open-circuited
4. Short between two pins (other than ground or V_{CC})

We will now describe each of these types of failure.

Malfunction in Internal Circuitry

This is usually caused by one of the internal components failing completely or operating outside its specifications. When this happens, the IC outputs do not respond properly to the IC inputs. There is no way to predict what the outputs will do because it depends on what internal component has failed. Examples of this type of failure would be a base-emitter short in transistor Q_4 or an extremely large resistance value for R_2 in the TTL INVERTER of Figure 4-30(a). This type of internal IC failure is not as common as the other three.

Input Internally Shorted to Ground or Supply

This type of internal failure will cause an IC input to be stuck in the LOW or HIGH state. Figure 4-35(a) shows input pin 2 of a NAND gate shorted to ground within the IC. This will cause pin 2 always to be in the LOW state. If this input pin is being driven by a logic signal B, it will effectively short B to ground. Thus, this type of fault will affect the output of the device that is generating the B signal.

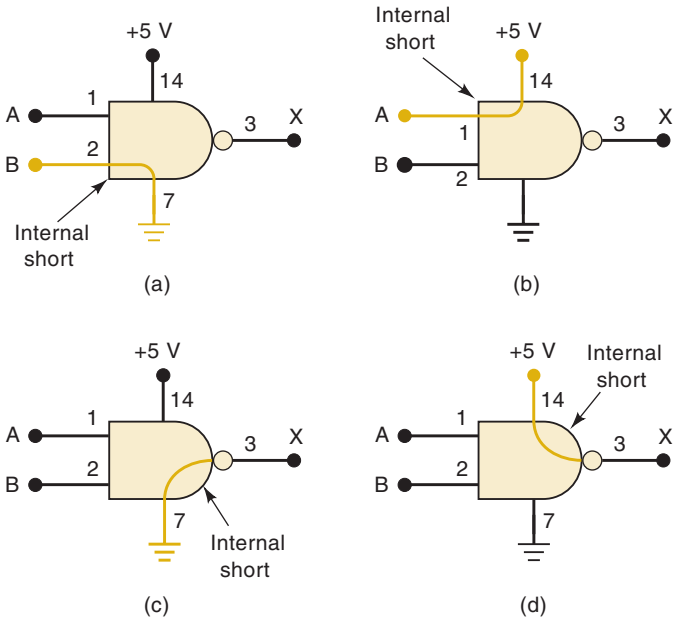


FIGURE 4-35 (a) IC input internally shorted to ground; (b) IC input internally shorted to supply voltage. These two types of failures force the input signal at the shorted pin to stay in the same state. (c) IC output internally shorted to ground; (d) output internally shorted to supply voltage. These two failures do not affect signals at the IC inputs.

Similarly, an IC input pin could be internally shorted to +5 V, as in Figure 4-35(b). This would keep that pin stuck in the HIGH state. If this input pin is being driven by a logic signal *A*, it will effectively short *A* to +5 V.

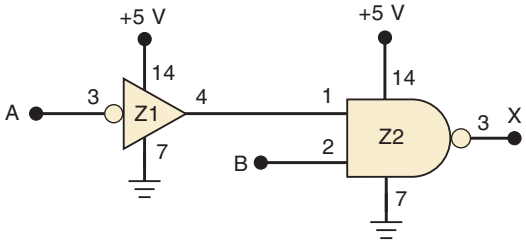
Output Internally Shorted to Ground or Supply

This type of internal failure will cause the output pin to be stuck in the LOW or HIGH state. Figure 4-35(c) shows pin 3 of the NAND gate shorted to ground within the IC. This output is stuck LOW, and it will not respond to the conditions applied to input pins 1 and 2; in other words, logic inputs *A* and *B* will have no effect on output *X*.

An IC output pin can also be shorted to +5 V within the IC, as shown in Figure 4-35(d). This forces the output pin 3 to be stuck HIGH regardless of the state of the signals at the input pins. Note that this type of failure has no effect on the logic signals at the IC inputs.

EXAMPLE 4-24

Refer to the circuit of Figure 4-36. A technician uses a logic probe to determine the conditions at the various IC pins. The results are recorded in the figure. Examine these results and determine if the circuit is working properly. If not, suggest some of the possible faults.



Pin	Condition
Z1-3	Pulsing
Z1-4	LOW
Z2-1	LOW
Z2-2	HIGH
Z2-3	HIGH

FIGURE 4-36
Example 4-24.

Solution

Output pin 4 of the INVERTER should be pulsing because its input is pulsing. The recorded results, however, show that pin 4 is stuck LOW. Because this is connected to Z2 pin 1, this keeps the NAND output HIGH. From our preceding discussion, we can list three possible faults that could produce this operation.

First, there could be an internal component failure in the INVERTER that prevents it from responding properly to its input. Second, pin 4 of the INVERTER could be shorted to ground internal to Z1, thereby keeping it stuck LOW. Third, pin 1 of Z2 could be shorted to ground internal to Z2. This would prevent the INVERTER output pin from changing.

In addition to these possible faults, there can be external shorts to ground anywhere in the conducting path between Z1 pin 4 and Z2 pin 1. We will see how to go about isolating the actual fault in a subsequent example.

Open-Circuited Input or Output

Sometimes the very fine conducting wire that connects an IC pin to the IC's internal circuitry will break, producing an open circuit. Figure 4-37 in Example 4-25 shows this for an input (pin 13) and an output (pin 6). If a signal is applied to pin 13, it will not reach the NAND-1 gate input and so will not have an effect

on the NAND-1 output. The open gate input will be in the floating state. As stated earlier, TTL devices will respond as if this floating input is a logic 1, and CMOS devices will respond erratically and may even become damaged from overheating.

The open at the NAND-4 output prevents the signal from reaching IC pin 6, so there will be no stable voltage present at that pin. If this pin is connected to the input of another IC, it will produce a floating condition at that input.

EXAMPLE 4-25

What would a logic probe indicate at pin 13 and at pin 6 of Figure 4-37?

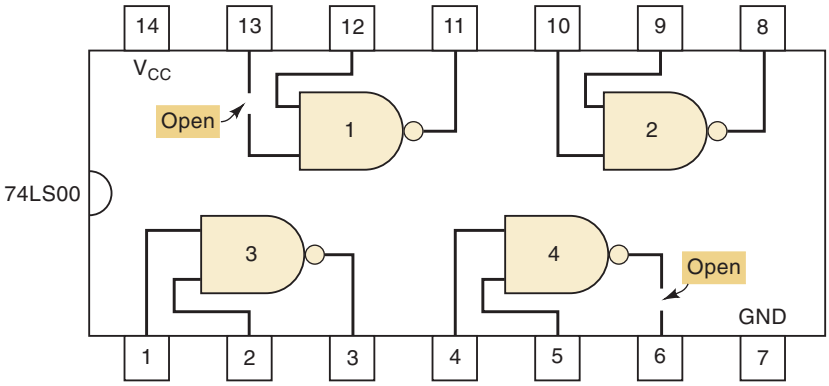


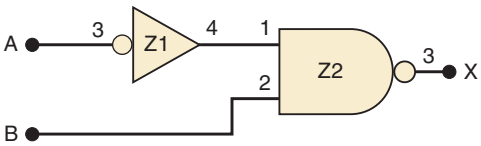
FIGURE 4-37 An IC with an internally open input will not respond to signals applied to that input pin. An internally open output will produce an unpredictable voltage at that output pin.

Solution

At pin 13, the logic probe will indicate the logic level of the external signal that is connected to pin 13 (which is not shown in the diagram). At pin 6, the logic probe will show no LED lit for an indeterminate logic level because the NAND output level never makes it to pin 6.

EXAMPLE 4-26

Refer to the circuit of Figure 4-38 and the recorded logic probe indications. What are some of the possible faults that could produce the recorded results? Assume that the ICs are TTL.



Note: V_{CC} and ground connections to each IC are not shown

Pin	Condition
Z1-3	HIGH
Z1-4	LOW
Z2-1	LOW
Z2-2	Pulsing
Z2-3	Pulsing

FIGURE 4-38 Example 4-26.

Solution

Examination of the recorded results indicates that the INVERTER appears to be working properly, but the NAND output is inconsistent with its inputs. The NAND output should be HIGH because its input pin 1 is LOW. This LOW should prevent the NAND gate from responding to the pulses at pin 2. It is probable that this LOW is not reaching the internal NAND gate circuitry

because of an internal open. Because the IC is TTL, this open circuit would produce the same effect as a logic HIGH at pin 1. If the IC had been CMOS, the internal open circuit at pin 1 might have produced an indeterminate output and possible overheating and destruction of the chip.

From our earlier statement regarding open TTL inputs, you might have expected that the voltage of pin 1 of Z2 would be 1.4 to 1.8 V and should have been registered as indeterminate by the logic probe. This would have been true if the open circuit had been *external* to the NAND chip. There is no open circuit between Z1 pin 4 and Z2 pin 1, and so the voltage at Z1 pin 4 is reaching Z2 pin 1, but it becomes disconnected *inside* the NAND chip.

Short Between Two Pins

An internal short between two pins of an IC will force the logic signals at those pins always to be identical. Whenever two signals that are supposed to be different show the same logic-level variations, there is a good possibility that the signals are shorted together.

Consider the circuit in Figure 4-39, where pins 5 and 6 of the NOR gate are internally shorted together. The short causes the two INVERTER output pins to be connected together so that the signals at Z1 pin 2 and Z1 pin 4 must be identical, even when the two INVERTER input signals are trying to produce different outputs. To illustrate, consider the input waveforms shown in the diagram. Even though these input waveforms are different, the waveforms at outputs Z1-2 and Z1-4 are the same.

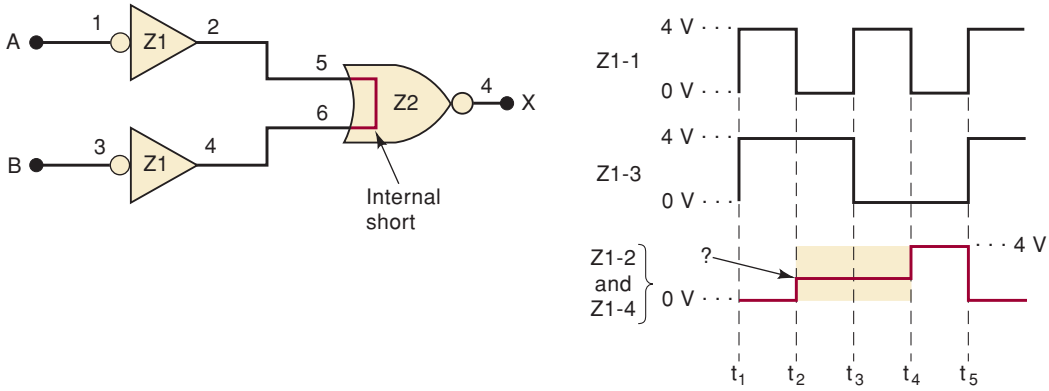


FIGURE 4-39 When two input pins are internally shorted, the signals driving these pins are forced to be identical, and usually a signal with three distinct levels results.

During the interval t_1 to t_2, both INVERTERs have a HIGH input and both are trying to produce a LOW output, so that their being shorted together makes no difference. During the interval t_4 to t_5, both INVERTERs have a LOW input and are trying to produce a HIGH output, so that again their being shorted has no effect. However, during the intervals t_2 to t_3 and t_3 to t_4, one INVERTER is trying to produce a HIGH output while the other is trying to produce a LOW output. This is called signal **contention** because the two signals are "fighting" each other. When this happens, the actual voltage level that appears at the shorted outputs will depend on the internal IC circuitry. For TTL devices, it will usually be a voltage in the high end of the logic 0 range (i.e., close to 0.8 V), although it may also be in the indeterminate range. For CMOS devices, it will often be a voltage in the indeterminate range.

Whenever you see a waveform like the Z1-2, Z1-4 signal in Figure 4-39 with three different levels, you should suspect that two output signals may be shorted together.

REVIEW QUESTIONS

1. List the different internal digital IC faults.
2. Which internal IC fault can produce signals that show three different voltage levels?
3. What would a logic probe indicate at Z1-2 and Z1-4 of Figure 4-39 if $A = 0$ and $B = 1$?
4. What is signal contention?

4-12 EXTERNAL FAULTS

We have seen how to recognize the effects of various faults internal to digital ICs. Many more things can go wrong external to the ICs; we will describe the most common ones in this section.

Open Signal Lines

This category includes any fault that produces a break or discontinuity in the conducting path such that a voltage level or signal is prevented from going from one point to another. Some of the causes of open signal lines are:

1. Broken wire
2. Poor solder connection; loose wire-wrap connection
3. Crack or cut trace on a printed circuit board (some of these are hairline cracks that are difficult to see without a magnifying glass)
4. Bent or broken pin on an IC
5. Faulty IC socket such that the IC pin does not make good contact with the socket

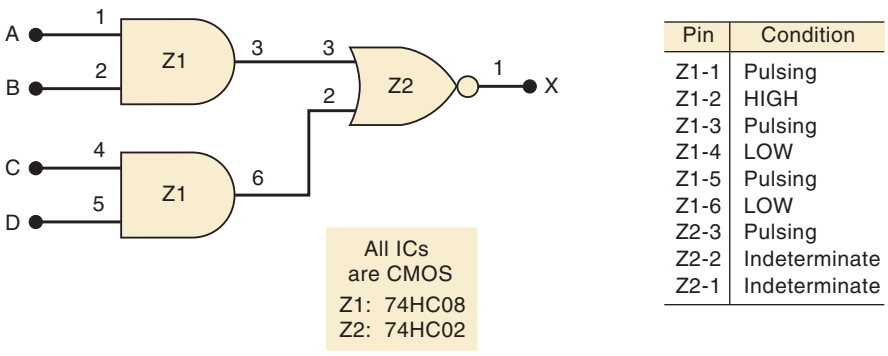
This type of circuit fault can often be detected by a careful visual inspection and then verified by disconnecting power from the circuit and checking for continuity (i.e., a low-resistance path) with an ohmmeter between the two points in question.

EXAMPLE 4-27

Consider the CMOS circuit of Figure 4-40 and the accompanying logic probe indications. What is the most probable circuit fault?

Solution

The indeterminate level at the NOR gate output is probably due to the indeterminate input at pin 2. Because there is a LOW at Z1-6, this LOW should also be at Z2-2. Clearly, the LOW from Z1-6 is not reaching Z2-2, and there must be an open circuit in the signal path between these two points. The location of this open circuit can be determined by starting at Z1-6 with the logic probe and tracing the LOW level along the signal path toward Z2-2 until it changes into an indeterminate level.



Pin	Condition
Z1-1	Pulsing
Z1-2	HIGH
Z1-3	Pulsing
Z1-4	LOW
Z1-5	Pulsing
Z1-6	LOW
Z2-3	Pulsing
Z2-2	Indeterminate
Z2-1	Indeterminate

FIGURE 4-40 Example 4-27.

Shorted Signal Lines

This type of fault has the same effect as an internal short between IC pins. It causes two signals to be exactly the same (signal contention). A signal line may be shorted to ground or V_{CC} rather than to another signal line. In those cases, the signal will be forced to the LOW or the HIGH state. The main causes for unexpected shorts between two points in a circuit are as follows:

1. *Sloppy wiring.* An example of this is stripping too much insulation from ends of wires that are in close proximity.
2. *Solder bridges.* These are splashes of solder that short two or more points together. They commonly occur between points that are very close together, such as adjacent pins on a chip.
3. *Incomplete etching.* The copper between adjacent conducting paths on a printed circuit board is not completely etched away.

Again, a careful visual inspection can very often uncover this type of fault, and an ohmmeter check can verify that the two points in the circuit are shorted together.

Faulty Power Supply

All digital systems have one or more dc power supplies that supply the V_{CC} and V_{DD} voltages required by the chips. A faulty power supply or one that is overloaded (supplying more than its rated amount of current) will provide poorly regulated supply voltages to the ICs, and the ICs either will not operate or will operate erratically.

A power supply may go out of regulation because of a fault in its internal circuitry, or because the circuits that it is powering are drawing more current than the supply is designed for. This can happen if a chip or a component has a fault that causes it to draw much more current than normal.

It is good troubleshooting practice to check the voltage levels at each power supply in the system to see that they are within their specified ranges. It is also a good idea to check them on an oscilloscope to verify that there is no significant amount of ac ripple on the dc levels and to verify that the voltage levels stay regulated during the system operation.

One of the most common signs of a faulty power supply is one or more chips operating erratically or not at all. Some ICs are more tolerant of power supply variations and may operate properly, while others do not. You should always check the power and ground levels at each IC that appears to be operating incorrectly.

Output Loading

When a digital IC has its output connected to too many IC inputs, its output current rating will be exceeded, and the output voltage can fall into the indeterminate range. This effect is called *loading* the output signal (actually it's overloading the output signal) and is usually the result of poor design or an incorrect connection.

REVIEW QUESTIONS

1. What are the most common types of external faults?
2. List some of the causes of signal-path open circuits.
3. What symptoms are caused by a faulty power supply?
4. How might loading affect an IC output voltage level?

4-13 TROUBLESHOOTING CASE STUDY

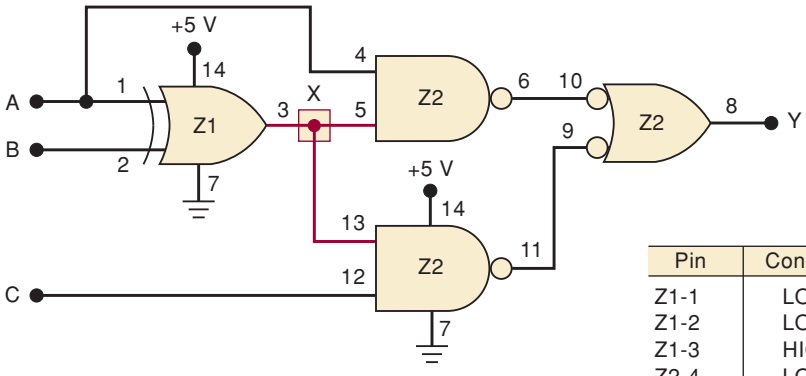
The following example will illustrate the analytical processes involved in troubleshooting digital circuits. Although the example is a fairly simple combinational logic circuit, the reasoning and the troubleshooting procedures used can be applied to the more complex digital circuits that we encounter in subsequent chapters.

EXAMPLE 4-28

Consider the circuit of Figure 4-41. Output Y is supposed to go HIGH for either of the following conditions:

1. $A = 1, B = 0$ regardless of the level on C
2. $A = 0, B = 1, C = 1$

You may wish to verify these results for yourself.



Pin	Condition
Z1-1	LOW
Z1-2	LOW
Z1-3	HIGH
Z2-4	LOW
Z2-5	HIGH
Z2-6,10	HIGH
Z2-13	HIGH
Z2-12	HIGH
Z2-9,11	LOW
Z2-8	HIGH

FIGURE 4-41 Example 4-28.

When the circuit is tested, the technician observes that output Y goes HIGH whenever A is HIGH or C is HIGH, regardless of the level at B. She takes logic probe measurements for the condition where $A = B = 0, C = 1$ and comes up with the indications recorded in Figure 4-41.

Examine the recorded levels and list the possible causes for the malfunction. Then develop a step-by-step procedure to determine the exact fault.

Solution

All of the NAND gate outputs are correct for the levels present at their inputs. The XOR gate, however, should be producing a LOW at output pin 3 because both of its inputs are at the same LOW level. It appears that Z1-3 is stuck HIGH, even though its inputs should produce a LOW. There are several possible causes for this:

1. An internal component failure in Z1 that prevents its output from going LOW
2. An external short to V_{CC} from any point along the conductors connected to node X (shaded in the diagram of the figure)
3. Pin 3 of Z1 internally shorted to V_{CC}
4. Pin 5 of Z2 internally shorted to V_{CC}
5. Pin 13 of Z2 internally shorted to V_{CC}

All of these possibilities except for the first one will short node X (and every IC pin connected to it) directly to V_{CC}.

The following procedure can be used to isolate the fault. This procedure is not the only approach that can be used and, as we stated earlier, the actual troubleshooting procedure that a technician uses is very dependent on what test equipment is available.

1. Check the V_{CC} and ground levels at the appropriate pins of Z1. Although it is unlikely that the absence of either of these might cause Z1-3 to stay HIGH, it is a good idea to make this check on any IC that is producing an incorrect output.
2. Turn off power to the circuit and use an ohmmeter to check for a short (resistance less than 1 Ω) between node X and any point connected to V_{CC} (such as Z1-14 or Z2-14). If no short is indicated, the last four possibilities in our list can be eliminated. This means that it is very likely that Z1 has an internal failure and should be replaced.
3. If step 2 shows that there is a short from node X to V_{CC}, perform a thorough visual examination of the circuit board and look for solder bridges, unetched copper slivers, uninsulated wires touching each other, and any other possible cause of an external short to V_{CC}. A likely spot for a solder bridge would be between adjacent pins 13 and 14 of Z2. Pin 14 is connected to V_{CC}, and pin 13 to node X. If an external short is found, remove it and perform an ohmmeter check to verify that node X is no longer shorted to V_{CC}.
4. If step 3 does not reveal an external short, the three possibilities that remain are internal shorts to V_{CC} at Z1-3, Z2-13, or Z2-5. One of these is shorting node X to V_{CC}.

To determine which of these IC pins is the culprit, we should disconnect each of them from node X *one at a time* and recheck for a short to V_{CC} after each disconnection. When the pin that is internally shorted to V_{CC} is disconnected, node X will no longer be shorted to V_{CC}.

The process of disconnecting each suspected pin from node *X* can be easy or difficult depending on how the circuit is constructed. If the ICs are in sockets, all you need to do is to pull the IC from its socket, bend out the suspected pin, and reinsert the IC into its socket. If the ICs are soldered into a printed circuit board, you will have to cut the trace that is connected to the pin and repair the cut trace when you are finished.

Example 4-28, although fairly simple, shows you the kinds of thinking that a troubleshooter must employ to isolate a fault. You will have the opportunity to begin developing your own troubleshooting skills by working on many end-of-chapter problems that have been designated with a **T** for troubleshooting.

4-14 PROGRAMMABLE LOGIC DEVICES*

In the previous sections, we briefly considered the class of ICs known as programmable logic devices. In Chapter 3, we introduced the concept of describing a circuit's operation using a hardware description language. In this section, we will explore these topics further and become prepared to use the tools of the trade to develop and implement digital systems using PLDs. Of course, it is impossible to understand all the complex details of how a PLD works before grasping the fundamentals of digital circuits. As we examine new fundamental concepts, we will expand our knowledge of the PLDs and the programming methods. The material is presented in such a way that anyone who is not interested in PLDs can easily skip over these sections without loss of continuity in the coverage of the basic principles.

Let's review the process we covered earlier of designing combinational digital circuits. The input devices are identified and assigned an algebraic name like *A*, *B*, *C*, or *LOAD, SHIFT, CLOCK*. Likewise, output devices are given names like *X*, *Z*, or *CLOCK_OUT, SHIFT_OUT.* Then a truth table is constructed that lists all the possible input combinations and identifies the required state of the outputs under each input condition. The truth table is one way of describing how the circuit is to operate. Another way to describe the circuit's operation is the Boolean expression. From this point the designer must find the simplest algebraic relationship and select digital ICs that can be wired together to implement the circuit. You have probably experienced that these last steps are the most tedious, time consuming, and prone to errors.

Programmable logic devices allow most of these tedious steps to be automated by a computer and PLD *development software.* Using programmable logic improves the efficiency of the design and development process. Consequently, most modern digital systems are implemented in this way. The job of the circuit designer is to identify inputs and outputs, specify the logical relationship in the most convenient manner, and select a programmable device that is capable of implementing the circuit at the lowest cost. The concept behind programmable logic devices is simple: put lots of logic gates in a single IC and control the interconnection of these gates electronically.

PLD Hardware

Recall from Chapter 3 that many digital circuits today are implemented using programmable logic devices (PLDs). These devices are configured electronically and their internal circuits are "wired" together electronically to

*All sections covering PLDs may be skipped without loss of continuity in the balance of Chapters 1–12.

form a logic circuit. This programmable wiring can be thought of as thousands of connections that are either connected (1) or not connected (0). It is very tedious to try to configure these devices by manually placing 1s and 0s in a grid, so the next logical question is, "How do we control the interconnection of gates in a PLD electronically?"

A common method of connecting one of many signals entering a network to one of many signal lines exiting the network is a switching matrix. Refer back to Figure 3-44, where this concept was introduced. A matrix is simply a grid of conductors (wires) arranged in rows and columns. Input signals are connected to the columns of the matrix, and the outputs are connected to the rows of the matrix. At each intersection of a row and a column is a switch that can electrically connect that row to that column. The switches that connect rows to columns can be mechanical switches, fusible links, electromagnetic switches (relays), or transistors. This is the general structure used in many applications and will be explored further when we study memory devices in Chapter 12.

PLDs also use a switch matrix that is often referred to as a programmable array. By deciding which intersections are connected and which ones are not, we can "program" the way the inputs are connected to the outputs of the array. In Figure 4-42, a programmable array is used to select the inputs for each AND gate. Notice that in this simple matrix, we can produce any logical product combination of variables A, B at any of the AND gate outputs. A matrix or programmable array such as the one shown in the figure can also be used to connect the AND outputs to OR gates. The details of various PLD architectures will be covered thoroughly in Chapter 13.

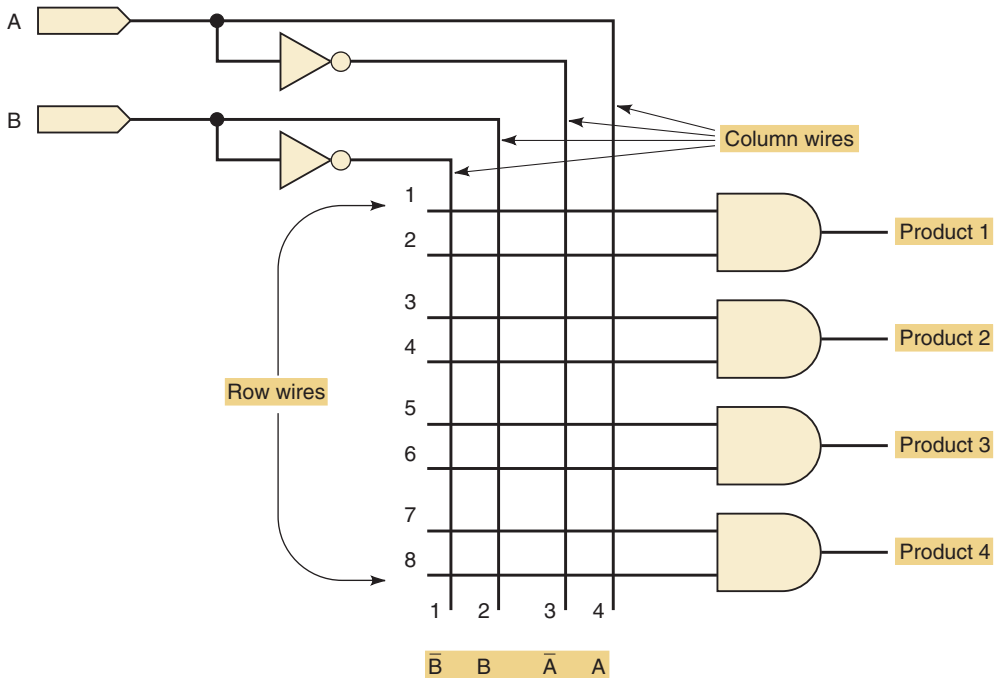


FIGURE 4-42 A programmable array for selecting inputs as product terms.

Programming a PLD

There are two ways to "program" a PLD IC. Programming means making the actual connections in the array. In other words, it means determining which of

those connections are supposed to be open (0) and which are supposed to be closed (1). The first method involves removing the PLD IC chip from its circuit board. The chip is then placed in a special fixture called a **programmer**, shown in Figure 4-43. Most modern programmers are connected to a personal computer that is running software containing libraries of information about the many types of programmable devices available.

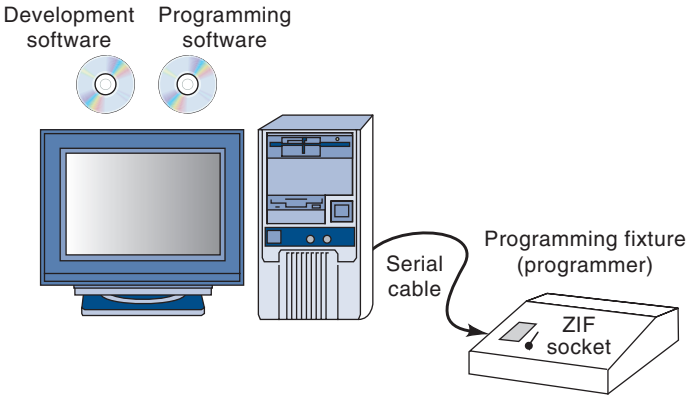


FIGURE 4-43 A PLD development system.

The programming software is invoked (called up and executed) on the PC to establish communication with the programmer. This software allows the user to set up the programmer for the type of device that is to be programmed, check if the device is blank, read the state of any programmable connection in the device, and provide instructions for the user to program a chip. Ultimately, the part is placed into a special socket that allows you to drop the chip in and then clamp the contacts onto the pins. This is called a **zero insertion force (ZIF)** socket. *Universal programmers* that can program any type of programmable device are available from numerous manufacturers.

Fortunately, as programmable parts began to proliferate, manufacturers saw the need to standardize pin assignments and programming methods. As a result, the Joint Electronic Device Engineering Council **(JEDEC)** was formed. One of the results was JEDEC standard 3, a format for transferring programming data for PLDs, independent of the PLD manufacturer or programming software. Pin assignments for various IC packages were also standardized, making universal programmers less complicated. Consequently, programming fixtures can program numerous types of PLDs. The software that allows the designer to specify a configuration for a PLD simply needs to produce an output file that conforms to the JEDEC standards. Then this JEDEC file can be loaded into any JEDEC-compatible PLD programmer that is capable of programming the desired type of PLD.

The second method is referred to as in system programming (ISP). As its name implies, the chip does not need to be removed from its circuit for storage of the programming information. A standard interface has been developed by the Joint Test Action Group **(JTAG)**. The interface was developed to allow ICs to be tested without actually connecting test equipment to every pin of the IC. It also allows for internal programming. Four pins on the IC are used as a portal to store data and retrieve information about the inner condition of the IC. Many ICs, including PLDs and microcontrollers, are manufactured today to include the JTAG interface. An interface cable connects the four JTAG pins on the IC to an output port (like the printer port) of a personal computer. Software running on the PC establishes contact with the IC and loads the information in the proper format.

Development Software

We have examined several methods of describing logic circuits now, including schematic capture, logic equations, truth tables, and HDL. We also described the fundamental methods of storing 1s and 0s into a PLD IC to connect the logic circuits in the desired way. The biggest challenge in getting a PLD programmed is converting from any form of description into the array of 1s and 0s. Fortunately, this task is accomplished quite easily by a computer running the development software. The development software that we will be referring to and using for examples is produced by Altera. This software allows the designer to enter a circuit description in any one of the many ways we have been discussing: graphic design files (schematics), AHDL, and VHDL. It also allows the use of another HDL, called Verilog, and the option of describing the circuit with timing diagrams. Circuit blocks described by any of these methods can also be "connected" together to implement a much larger digital system, as shown in Figure 4-44. Any logic diagram found in this text can be redrawn using the schematic entry tools in the Altera software to create a graphic design file. We will not focus on graphic design entry in this text because it is quite straightforward to pick up these skills in the laboratory. We will focus our examples on the methods that allow us to use HDL as an alternate means of describing a circuit. For more information on the Altera software, see the accompanying CD as well as user manuals from the Altera web site (http://www.altera.com).

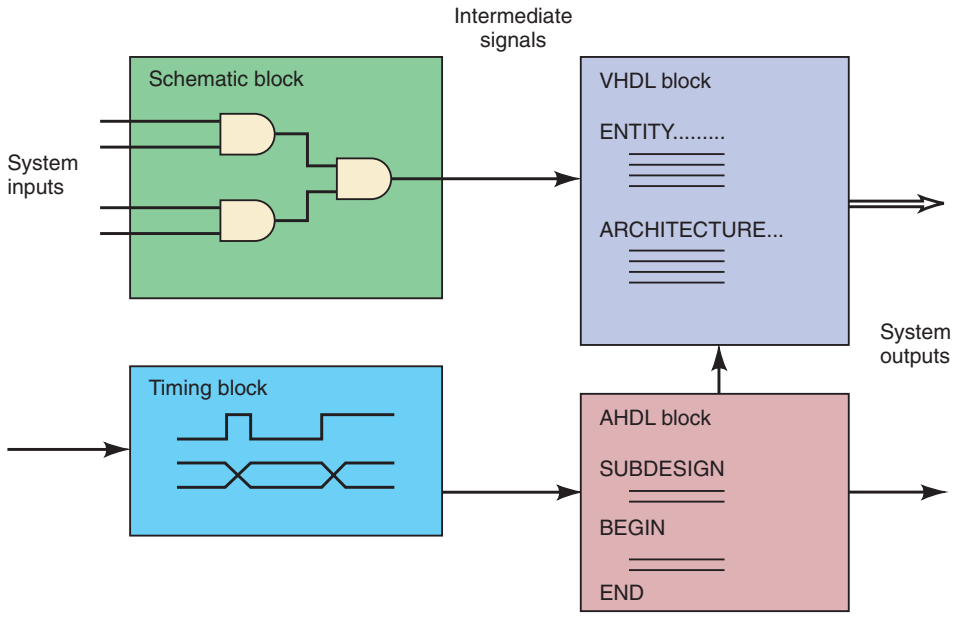


FIGURE 4-44 Combining blocks developed using different description methods.

This concept of using building blocks of circuits is called **hierarchical design**. Small, useful logic circuits can be defined in whatever manner is most convenient (graphic, HDL, timing, etc.) and then combined with other circuits to form a large section of a project. Sections can be combined and connected with other sections to form the whole system. Figure 4-45 shows the hierarchical structure of a CD player using a block diagram. The outer box encloses the entire system. The dashed lines identify each major subsection, and each subsection contains individual circuits. Although it is not shown in this diagram, each circuit may be made up of smaller building blocks of common

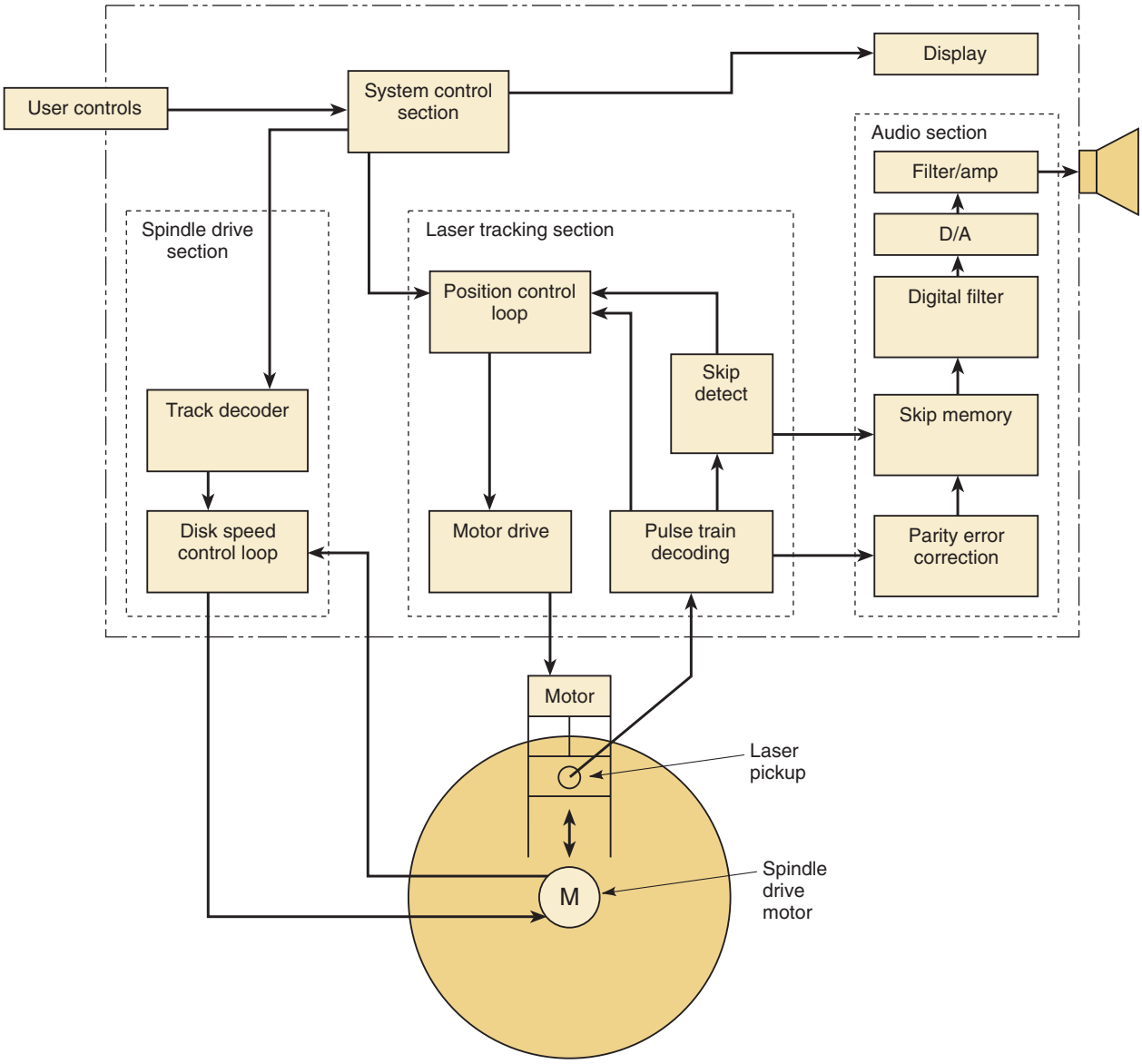


FIGURE 4-45 Block diagram of a CD player.

digital circuits. The Altera development software makes this type of modular, hierarchical design and development easy to accomplish.

Design and Development Process

Another way you might see the hierarchy of a system like the CD player just described is shown in Figure 4-46. The top level represents the entire system. It is made up of three subsections, each of which in turn is made up of the smaller circuits shown. Notice that this diagram does not show how the signals flow throughout the system but clearly identifies the various levels of the hierarchical structure of the project.

This type of diagram has led to the name for one of the most common methods of design: **top-down**. With this design approach, you start with the overall description of the entire system, such as the top box in Figure 4-46. Then you define several subsections that will make up the system. The subsections are further refined into individual circuits connected together.

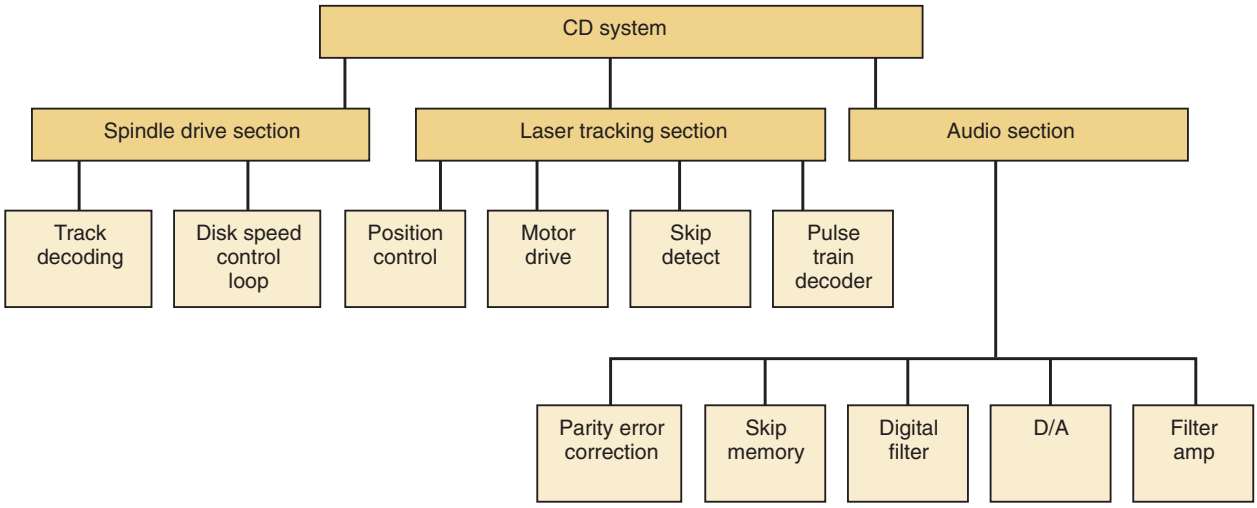


FIGURE 4-46 An organizational hierarchy chart.

Every one of these hierarchy levels has defined inputs, outputs, and behavior. Each can be tested individually before it is connected to the others.

After defining the blocks from the top down, the system is built from the bottom up. Each block in this system design has a design file that describes it. The lowest level blocks must be designed by opening a design file and writing a description of its operation. The designed block is then compiled using the development tools. The compiling process determines if you have made errors in your syntax. Until your syntax is correct, the computer cannot possibly translate your description into its proper form. After it has been compiled with no syntax errors, it should be tested to see if it operates correctly. Development systems offer simulator programs that run on the PC and simulate the way your circuit responds to inputs. A simulator is a computer program that calculates the correct output logic states based on a description of the logic circuit and the current inputs. A set of hypothetical inputs and their corresponding correct outputs are developed that will prove the block works as expected. These hypothetical inputs are often called **test vectors**. Thorough testing during simulation greatly increases the likelihood of the final system working reliably. Figure 4-47 shows the simulation file for the circuit described in Figure 3-13(a) of Chapter 3. Inputs *a*, *b*, and *c* were entered as test vectors, and the simulation produced output *y*.

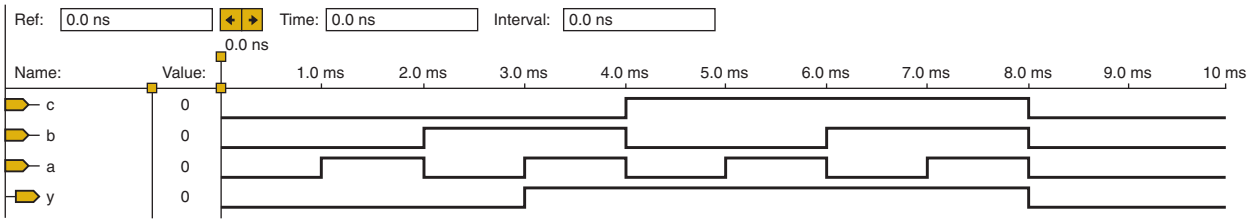


FIGURE 4-47 A timing simulation of a circuit described in HDL.

When the designer is satisfied that the design works, the design can be verified by actually programming a chip and testing. For a complex PLD, the designer can either let the development system assign pins and then lay out the final circuit board accordingly, or specify the pins for each signal using

the software features. If the compiler assigns the pins, the assignments can be found in the report file or pin-out file, which provides many details about the implementation of the design. If the designer specifies the pins, it is important to know the constraints and limitations of the chip's architecture. These details will be covered in Chapter 13. The flowchart of Figure 4-48 summarizes the design process for designing each block.

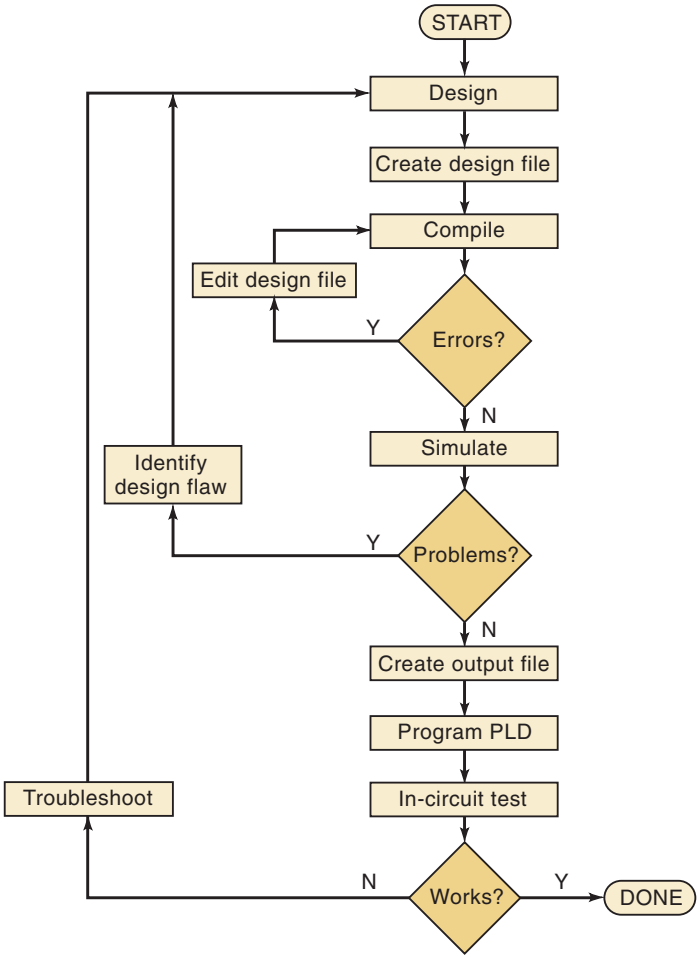


FIGURE 4-48 PLD development cycle flowchart.

After each circuit in a subsection has been tested, all can be combined and the subsection can be tested following the same process that was used for the small circuits. Then the subsections are combined and the system is tested. This approach lends itself very well to a typical project environment, where a team of people are working together, each responsible for his or her own circuits and sections that will ultimately come together to make up the system.

REVIEW QUESTIONS

1. What is actually being "programmed" in a PLD?
2. What bits (column, row) in Figure 4-42 must be connected to make Product 1 = AB?
3. What bits (column, row) in Figure 4-42 must be connected to make Product 3 = $A\overline{B}$?

4-15 REPRESENTING DATA IN HDL

Numeric data can be represented in various ways. We have studied the use of the hexadecimal number system as a convenient way to communicate bit patterns. We naturally prefer to use the decimal number system for numeric data, but computers and digital systems can operate only on binary information, as we studied in previous chapters. When we write in HDL, we often need to use these various number formats, and the computer must be able to understand which number system we are using. So far in this text, we have used a subscript to indicate the number system. For example, 101_2 was binary, 101_{16} was hexadecimal, and 101_{10} was decimal. Every programming language and HDL has its own unique way of identifying the various number systems, generally done with a prefix to indicate the number system. In most languages, a number with no prefix is assumed to be decimal. When we read one of these number designations, we must think of it as a symbol that represents a binary bit pattern. These numeric values are referred to as scalars or **literals**. Table 4-8 summarizes the methods of specifying values in binary, hex, and decimal for AHDL and VHDL.

TABLE 4-8 Designating number systems in HDL.

Number System	AHDL	VHDL	Bit Pattern	Decimal Equivalent
Binary	B"101"	B"101"	101	5
Hexadecimal	H"101"	X"101"	100000001	257
Decimal	101	101	1100101	101

EXAMPLE 4-29

Express the following bit pattern's numeric value in binary, hex, and decimal using AHDL and VHDL notation:

11001

Solution

Binary is designated the same in both AHDL and VHDL: **B "11001"**.
Converting the binary to hex, we have 19_{16}.
In AHDL: **H "19"**
In VHDL: **X "19"**
Converting the binary to decimal, we have 25_{10}.
Decimal is designated the same in both AHDL and VHDL: 25.

Bit Arrays/Bit Vectors

In Chapter 3, we declared names for inputs to and outputs of a very simple logic circuit. These were defined as bits, or single binary digits. What if we want to represent an input, output, or signal that is made up of several bits? In an HDL, we must define the type of the signal and its range of acceptable values.

To understand the concepts used in HDLs, let's first consider some conventions for describing bits of binary words in common digital systems. Suppose we have an eight-bit number representing the current temperature, and the number is coming into our digital system through an input port that we have named P1, as shown in Figure 4-49. We can refer to the individual bits of this port as P1 bit 0 for the least significant bit, on up to P1 bit 7 for the most significant bit.

We can also describe this port by saying that it is named P1, with bits numbered 7 down to 0. The terms **bit array** and **bit vector** are often used to describe this type of data structure. It simply means that the overall data structure (eight-bit port) has a name (P1) and that each individual element (bit) has a unique **index** number (0–7) to describe that bit's position (and possibly its numeric weight) in the overall structure. The HDLs and computer programming languages take advantage of this notation. For example, the third bit from the right is designated as P1[2], and it can be connected to another signal bit by using an assignment operator.

EXAMPLE 4-30

Assume there is an eight-bit array named P1, as shown in Figure 4-49, and another four-bit array is named P5.

(a) Write the bit designation for the most significant bit of P1.
(b) Write the bit designation for the least significant bit of P5.
(c) Write an expression that causes the least significant bit of P5 to drive the most significant bit of P1.

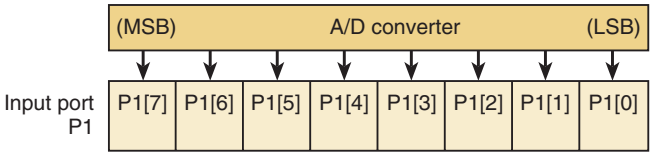


FIGURE 4-49 Bit array notation.

Solution

(a) The name of the port is P1 and the most significant bit is bit 7. The proper designation for P1 bit 7 is P1[7].
(b) The name of the port is P5 and the least significant bit is bit 0. The proper designation for P5 bit 0 is P5[0].
(c) The driving signal is placed on the right side of the assignment operator, and the driven signal is placed on the left: P1[7] = P5[0];.

AHDL

AHDL BIT ARRAY DECLARATIONS

In AHDL, port *p1* of Figure 4-49 is defined as an eight-bit input port, and the value on this port can be referred to using any number system, such as hex, binary, decimal, etc. The syntax for AHDL uses a name for the bit vector followed by the range of index designations, which are enclosed in square brackets. This declaration is included in the SUBDESIGN section. For example, to declare an eight-bit input port called *p1*, you would write

```
p1 [7..0] :INPUT; --define an 8-bit input port
```

EXAMPLE 4-31

Declare a four-bit input named *keypad* using AHDL.

Solution

```
keypad [3..0] :INPUT;
```

Intermediate variables can also be declared as an array of bits. As with single bits, they are declared just after the I/O declarations in SUBDESIGN. As an example, the eight-bit temperature port *p1* can be assigned (connected) to a node named *temp*, as follows:

```
VARIABLE temp [7..0] :NODE;
BEGIN
    temp[] = p1[];
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. Think of the term on the right of the equals sign as the source of the data and the term on the left as the destination. The empty brackets [] mean that each of the corresponding bits in the two arrays are being connected. Individual bits can also be "connected" by specifying the bits inside the brackets. For example, to connect only the least significant bit of p1 to the LSB of temp, the statement would be temp[0] = p1[0];.

VHDL BIT VECTOR DECLARATIONS

In VHDL, port *p1* of Figure 4-49 is defined as an eight-bit input port, and the value on this port can be referred to using only binary literals. The syntax for VHDL uses a name for the bit vector followed by the mode (:IN), the type **(BIT_VECTOR)**, and the range of index designations, which are enclosed in parentheses. This declaration is included in the ENTITY section. For example, to declare an eight-bit input port called *p1*, you would write

```
PORT (p1 :IN BIT_VECTOR (7 DOWNTO 0);
```

EXAMPLE 4-32

Declare a four-bit input named *keypad* using VHDL.

Solution

```
PORT(keypad :IN BIT_VECTOR (3 DOWNTO 0);
```

Intermediate signals can also be declared as an array of bits. As with single bits, they are declared just inside the ARCHITECTURE definition. As an example, the eight-bit temperature on port *p1* can be assigned (connected) to a signal named *temp*, as follows:

```
SIGNAL      temp :BIT_VECTOR (7 DOWNTO 0);
BEGIN
    temp <= p1;
END;
```

Notice that the input port *p1* has the data applied to it, and it is driving the signal wires named *temp*. No elements in the bit vector are specified, which means that all the bits are being connected. Individual bits can also be "connected" using signal assignments and by specifying the bit numbers inside parentheses. For example, to connect only the least significant bit of *p1* to the LSB of *temp*, the statement would be temp(0) <= p1(0);.

VHDL is very particular regarding the definitions of each type of the data. The type "bit_vector" describes an array of individual bits. This is interpreted differently than an eight-bit binary number (called a scaler quantity), which has the type **integer**. Unfortunately, VHDL does not allow us to assign an integer value to a BIT_VECTOR signal directly. Data can be represented by any of the types shown in Table 4-9, but data assignments and other operations must be done between objects of the same type. For example, the compiler will not allow you to take a number from a keypad declared as an integer and connect it to four LEDs that are declared as BIT_VECTOR outputs. Notice in Table 4-9, under Possible Values, that individual BIT and STD_LOGIC data **objects** (e.g., signals, variables, inputs, and outputs) are designated by single quotes, whereas values assigned to BIT_VECTOR and STD_LOGIC_VECTOR types are strings of valid bit values enclosed in double quotes.

TABLE 4-9 Common VHDL data types.

Data Type	Sample Declaration	Possible Values	Use
BIT	y :OUT BIT;	'0' '1'	y <= '0';
STD_LOGIC	driver :STD_LOGIC	'0' '1' 'z' 'x' '-'	driver <= 'z';
BIT_VECTOR	bcd_data :BIT_VECTOR	"0101" "1001"	digit <= bcd_data;
	(3 DOWNTO 0);	"0000"	
STD_LOGIC_VECTOR	dbus :STD_LOGIC_VECTOR (3 DOWNTO 0);	"0Z1X"	IF rd = '0' THEN dbus <= "zzzz";
INTEGER	SIGNAL z:INTEGER RANGE −32 TO 31;	−32..−2,−1,0,1,2 . . . 31	IF z > 5 THEN . . .

VHDL also offers some standardized data types that are necessary when using logic functions that are contained in the **libraries**. As you might have guessed, libraries are simply collections of little pieces of VHDL code that can be used in your hardware descriptions without reinventing the wheel. These libraries offer convenient functions, called **macrofunctions**, like many of the standard TTL devices that are described throughout this text. Rather than writing a new description of a familiar TTL device, we can simply pull its macrofunction out of the library and use it in our system. Of course, you need to get signals into and out of these macrofunctions, and the types of the signals in your code must match the types in the functions (which someone else wrote). This means that everyone must use the same standard data types.

When VHDL was standardized through the IEEE, many data types were created at the same time. The two that we will use in this text are **STD_LOGIC**, which is equivalent to BIT type, and **STD_LOGIC_VECTOR**, which is equivalent to BIT_VECTOR. As you recall, BIT type can have values of only '0' and '1'. The standard logic types are defined in the IEEE library and have a broader range of possible values than their built-in counterparts. The possible values for a STD_LOGIC type or for any element in a STD_LOGIC_VECTOR are given in Table 4-10. The names of these categories will make much more sense after we study the characteristics of logic circuits in Chapter 8. For now, we will show examples using values of only '1' and '0'.

TABLE 4-10 STD_LOGIC values.

'1'	Logic 1 (just like BIT type)
'0'	Logic 0 (just like BIT type)
'z'	High impedance*
'-'	don't care (just like you used in your K maps)
'U'	Uninitialized
'X'	Unknown
'W'	Weak unknown
'L'	Weak '0'
'H'	Weak '1'

*We will study tristate logic in Chapter 8.

REVIEW QUESTIONS

1. How would you declare a six-bit input array named push_buttons in (a) AHDL or (b) VHDL?
2. What statement would you use to take the MSB from the array in question 1 and put it on a single-bit output port named z? Use (a) AHDL or (b) VHDL.
3. In VHDL, what is the IEEE standard type that is equivalent to the BIT type?
4. In VHDL, what is the IEEE standard type that is equivalent to the BIT_VECTOR type?

4-16 TRUTH TABLES USING HDL

We have learned that a truth table is another way of expressing the operation of a circuit block. It relates the output of the circuit to every possible combination of its inputs. As we saw in Section 4-4, a truth table is the starting point for a designer to define how the circuit should operate. Then a Boolean expression is derived from the truth table and simplified using K maps or Boolean algebra. Finally the circuit is implemented from the final Boolean equation. Wouldn't it be great if we could go from the truth table directly to the final circuit without all those steps? We can do exactly that by entering the truth table using HDL.

TRUTH TABLES USING AHDL

The code in Figure 4-50 uses AHDL to implement a circuit and uses a truth table to describe its operation. The truth table for this design was presented in Example 4-7. The key point of this example is the use of the TABLE keyword in AHDL. It allows the designer to specify the operation of the circuit just like you would fill out a truth table. On the first line after TABLE, the input variables (*a*,*b*,*c*) are listed exactly like you would create a column heading on a truth table. By including the three binary variables in parentheses, we tell the compiler that we want to use these three bits as a group and to refer to them as a three-bit binary number or bit pattern. The specific values for this bit pattern are listed below the group and are referred to as binary literals. The special operator (=>) is used in truth tables to separate the inputs from the output (*y*).

AHDL

FIGURE 4-50 AHDL design file for Figure 4-7

```
%     Figure 4-7 in AHDL
      Digital Systems 10th ed
      Neal Widmer
      MAY 23, 2005                    %
SUBDESIGN FIG4_50
(
      a,b,c :INPUT;              --define inputs to block
      y     :OUTPUT;             --define block output
)
BEGIN
      TABLE
            (a,b,c)              =>    y;   --column headings
            (0,0,0)              =>    0;
            (0,0,1)              =>    0;
            (0,1,0)              =>    0;
            (0,1,1)              =>    1;
            (1,0,0)              =>    0;
            (1,0,1)              =>    1;
            (1,1,0)              =>    1;
            (1,1,1)              =>    1;
      END TABLE;
END;
```

The TABLE in Figure 4-50 is intended to show the relationship between the HDL code and a truth table. A more common way of representing the input data heading is to use a variable bit array to represent the value on *a*, *b*, *c*. This method involves a declaration of the bit array on the line before BEGIN, such as:

```
VARIABLE in_bits[2..0]     :NODE;
```

Just before the TABLE keyword, the input bits can be assigned to the array, *inbits[ ]*:

```
in_bits[] = (a,b,c);
```

Grouping three independent bits in order like this is referred to as **concatenating**, and it is often done to connect individual bits to a bit array. The table heading on the input bit sets can be represented by *in_bits[ ]*, in this case. Note that as we list the possible combinations of the inputs, we have several options. We can make up a group of 1s and 0s in parentheses, as shown in Figure 4-50, or we can represent the same bit pattern using the equivalent binary, hex, or decimal number. It is up to the designer to decide which format is most appropriate depending on what the input variables represent.

VHDL

TRUTH TABLES USING VHDL: SELECTED SIGNAL ASSIGNMENT

The code in Figure 4-51 uses VHDL to implement a circuit using a **selected signal assignment** to describe its operation. It allows the designer to specify the operation of the circuit, just like you would fill out a truth table. The truth table for this design was presented in Example 4-7. The primary point of this example is the use of the WITH signal_name SELECT statement in VHDL. A secondary point presented here shows how to put the data into a

```
--    Figure 4-7 in VHDL
--    Digital Systems 10th ed
--    Neal Widmer
--    MAY 23, 2005
ENTITY fig4_51 IS
PORT(
      a,b,c :IN BIT;          --declare individual input bits
      y     :OUT BIT);
END fig4_51;

ARCHITECTURE truth OF fig4_51 IS
      SIGNAL in_bits :BIT_VECTOR(2 DOWNTO 0);
      BEGIN
      in_bits <= a & b & c;    --concatenate input bits into bit_vector
            WITH in_bits SELECT
            y     <=    '0'  WHEN "000",   --Truth Table
                        '0'  WHEN "001",
                        '0'  WHEN "010",
                        '1'  WHEN "011",
                        '0'  WHEN "100",
                        '1'  WHEN "101",
                        '1'  WHEN "110",
                        '1'  WHEN "111";
END truth;
```

FIGURE 4-51 VHDL design file for Figure 4-7.

format that can be used conveniently with the selected signal assignment. Notice that the inputs are defined in the ENTITY declaration as three independent bits *a*, *b*, and *c*. Nothing in this declaration makes one of these more significant than another. The order in which they are listed does not matter. We want to compare the current value of these bits with each of the possible combinations that could be present. If we drew out a truth table, we would decide which bit to place on the left (MSB) and which to place on the right (LSB). This is accomplished in VHDL by **concatenating** (connecting in order) the bit variables to form a bit vector. The concatenation operator is "&". A signal is declared as a BIT_VECTOR to receive the ordered set of input bits and is used to compare the input's value with the string literals contained in quotes. The output (*y*) is assigned (<=) a bit value ('0' or '1') WHEN *in_bits* contains the value listed in double quotes.

VHDL is very strict in the way it allows us to assign and compare objects such as signals, variables, constants, and literals. The output *y* is a BIT, and so it must be assigned a value of '0' or '1'. The SIGNAL *in_bits* is a three-bit BIT_VECTOR, so it must be compared with a three-bit string literal value. VHDL will not allow *in_bits* (a BIT_VECTOR) to be compared with a hex number like X "5" or a decimal number like 3. These scalar quantities would be valid for assignment or comparison with integers.

EXAMPLE 4-33

Declare three signals in VHDL that are single bits named *too_hot, too_cold,* and *just_right*. Combine (concatenate) these three bits into a three-bit signal called *temp_status,* with hot on the left and cold on the right.

Solution

1. Declare signals first in Architecture.

```
SIGNAL too_hot, too_cold, just_right :BIT;
SIGNAL temp_status :BIT_VECTOR (2 DOWNTO 0);
```

2. Write concurrent assignment statements between BEGIN and END.

```
temp_status <= too_hot & just_right & too_cold;
```

REVIEW QUESTIONS

1. How would you concatenate three bits *x*, *y*, and *z* into a three-bit array named *omega*? Use AHDL or VHDL.
2. How are truth tables implemented in AHDL?
3. How are truth tables implemented in VHDL?

4-17 DECISION CONTROL STRUCTURES IN HDL

In this section, we will examine methods that allow us to tell the digital system how to make "logical" decisions in much the same way that we make decisions every day. In Chapter 3, we learned that concurrent assignment statements are evaluated such that the order in which they are written has no effect on the circuit being described. When using **decision control structures**, the order in which we ask the questions does matter. To summarize this concept in the terms used in HDL documentation, statements that can be written in any sequence are called **concurrent**, and statements that are evaluated in the sequence in which they are written are called **sequential**. The sequence of sequential statements affects the circuit's operation.

The examples we have considered so far involve several individual bits. Many digital systems require inputs that represent a numeric value. Refer again to Example 4-8, in which the purpose of the logic circuit is to monitor the battery voltage measured by an A/D converter. The digital value is represented by a four-bit number coming from the A/D into the logic circuit. These inputs are not independent binary variables but rather four binary digits of a number representing battery voltage. We need to give the data the correct type that will allow us to use it as a number.

IF/ELSE

Truth tables are great for listing all the possible combinations of independent variables, but there are better ways to handle numeric data. As an example, when a person leaves for school or work in the morning, she must make a logical decision about wearing a coat. Let's assume she decides this issue based only on the current temperature. How many of us would reason as follows?

I will wear a coat if the temperature is 0.
I will wear a coat if the temperature is 1.
I will wear a coat if the temperature is 2. . . .
I will wear a coat if the temperature is 55.

I will *not* wear a coat if the temperature is 56.
I will *not* wear a coat if the temperature is 57.
I will *not* wear a coat if the temperature is 58. . . .
I will *not* wear a coat if the temperature is 99.

This method is similar to the truth table approach of describing the decision. For every possible input, she decides what the output should be. Of course, what she would really do is decide as follows:

I will wear a coat if the temperature is less than 56 degrees.
Otherwise, I will *not* wear a coat.

An HDL gives us the power to describe logic circuits using this type of reasoning. First, we must describe the inputs as a *number within a given range,* and then we can write statements that decide what to do to the outputs based on the *value* of the incoming number. In most computer programming languages, as well as HDLs, these types of decisions are made using an IF/THEN/ELSE control structure. Whenever the decision is between doing something and doing nothing, an **IF/THEN** construct is used. The keyword IF is followed by a statement that is true or false. IF it is true, THEN do whatever is specified. In the event that the statement is false, no action is taken. Figure 4-52(a) shows graphically how this decision works. The diamond shape represents the decision being made by evaluating the statement contained within the diamond. Every decision has two possible outcomes: true or false. In this example, if the statement is false, no action is taken.

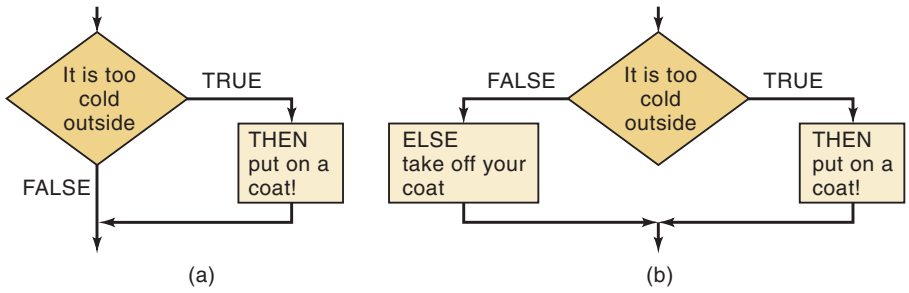


FIGURE 4-52 Logical flow of (a) IF/THEN and (b) IF/THEN/ELSE constructs.

In some cases it is not enough only to decide to act or not to act, but rather we must choose between two different actions. For example, in our analogy about the decision to wear a coat, if the person already has her coat on when making this decision, she will not be taking it off. The use of IF/THEN logic assumes that she is not wearing her coat initially.

When decisions demand two possible actions, the IF/THEN/**ELSE** control structure is used, as shown in Figure 4-52(b). Here again, the statement is evaluated as true or false. The difference is that, when the statement is false, a different action is performed. One of the two actions must occur with this construct. We can state it verbally as, "IF the statement is true, THEN do this. ELSE do that." In our coat analogy, this control structure would work, regardless of whether the person's coat was on or off initially.

Example 4-8 gave a simple example of a logic circuit that has as its input a numeric value representing battery voltage from an A/D converter. The inputs *A*, *B*, *C*, *D* are actually binary digits in a four-bit number, with *A* being the MSB and *D* being the LSB. Figure 4-53 shows the same circuit with the

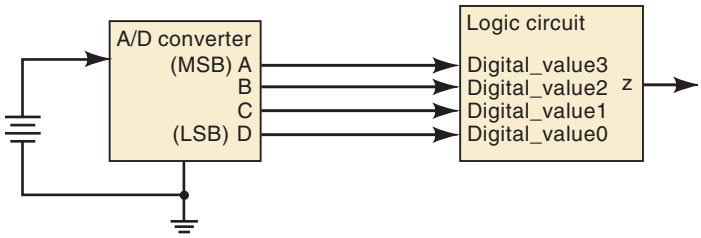


FIGURE 4-53 Logic circuit similar to Example 4-8.

inputs labeled as a four-bit number called *digital_value.* The relationship between bits is as follows:

A	*digital_value[3]*	digital value bit 3 (MSB)
B	*digital_value[2]*	digital value bit 2
C	*digital_value[1]*	digital value bit 1
D	*digital_value[0]*	digital value bit 0 (LSB)

The input can be treated as a decimal number between 0 and 15 if we specify the correct type of the input variable.

AHDL

IF/THEN/ELSE USING AHDL

In AHDL, the inputs can be specified as a binary number made up of multiple bits by assigning a variable name followed by a list of the bit positions, as shown in Figure 4-54. The name is *digital_value*, and the bit positions range from 3 down to 0. Notice how simple the code becomes using this method along with an IF/ELSE construct. The IF is followed by a statement that refers to the value of the entire four-bit input variable and compares it with the number 6. Of course, 6 is a decimal form of a scalar quantity and *digital_value[ ]* actually represents a binary number. The compiler can interpret numbers in any system, so it creates a logic circuit that compares the binary value of *digital_value* with the binary number for 6 and decides if this statement is true or false. If it is true, THEN the next statement (z = VCC) is used to assign *z* a value. Notice that in AHDL, we must use VCC for a logic 1 and GND for a logic 0 when assigning a logic level to a single bit. When *digital_value* is 6 or less, it follows the statement after ELSE (z = GND). The END IF; terminates the control structure.

FIGURE 4-54 AHDL version.

```
SUBDESIGN FIG4_54
(
   digital_value[3..0]  :INPUT;  -- define inputs to block
   z                    :OUTPUT; -- define block output
)
BEGIN
   IF digital_value[] > 6 THEN
         z = VCC;                -- output a 1
   ELSE  z = GND;                -- output a 0
   END IF;
END;
```

IF/THEN/ELSE USING VHDL

In VHDL, the critical issue is the declaration of the type of inputs. Refer to Figure 4-55. The input is treated as a single variable called *digital_value*. Because its type is declared as INTEGER, the compiler knows to treat it as a number. By specifying a range of 0 to 15, the compiler knows it is a four-bit number. Notice that RANGE does not specify the index number of a bit vector but rather the limits of the numeric value of the integer. Integers are treated differently than bit arrays (BIT_VECTOR) in VHDL. An integer can be compared with other numbers using inequality operators. A BIT_VECTOR cannot be used with inequality operators.

FIGURE 4-55 VHDL version.

```
ENTITY fig4_55 IS
PORT( digital_value :IN INTEGER RANGE 0 TO 15; -- 4-bit input
      z             :OUT BIT);
END fig4_55;

ARCHITECTURE decision OF fig4_55 IS

BEGIN
   PROCESS (digital_value)
      BEGIN
         IF (digital_value > 6) THEN
            z <= '1';
         ELSE
            z <= '0';
      END IF;
END PROCESS ;
END decision;
```

To use the IF/THEN/ELSE control structure, VHDL requires that the code be put inside a "PROCESS." The statements that occur within a process are *sequential*, meaning that the order in which they are written affects the operation of the circuit. The keyword **PROCESS** is followed by a list of variables called a **sensitivity list**, which is a list of variables to which the process code must respond. Whenever *digital_value* changes, it causes the process code to be reevaluated. Even though we know *digital_value* is really a four-bit binary number, the compiler will evaluate it as a number between the equivalent decimal values of 0 and 15. IF the statement in parentheses is true, THEN the next statement is applied (*z* is assigned a value of logic 1). If this statement is not true, the logic follows the ELSE clause and assigns a value of 0 to *z*. The END IF; terminates the control structure, and the END PROCESS; terminates the evaluation of the sequential statements.

ELSIF

We often need to choose among many possible actions, depending on the situation. The IF construct chooses whether to perform a set of actions or not. The IF/ELSE construct selects one out of two possible actions. By combining

IF and ELSE decisions, we can create a control structure referred to as **ELSIF**, which chooses one of many possible outcomes. The decision structure is shown graphically in Figure 4-56.

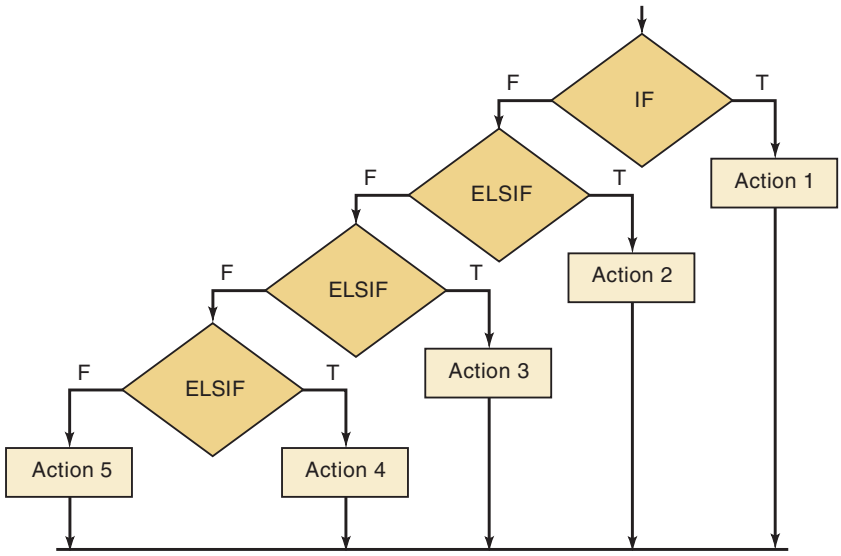


FIGURE 4-56 Flowchart for multiple decisions using IF/ELSIF.

Notice that as each condition is evaluated, it either performs an action if true or goes on to evaluate the next condition. Each action is associated with one condition, and there is no chance to select more than one action. Note also that the conditions used to decide the appropriate action can be any expression that evaluates as true or false. This fact allows the designer to use the inequality operators to choose an action based on a range of input values. As an example of this application, let's consider the temperature-measuring system that uses an A/D converter, as described in Figure 4-57. Suppose that we want to indicate when the temperature is in a certain range, which we will refer to as Too Cold, Just Right, and Too Hot.

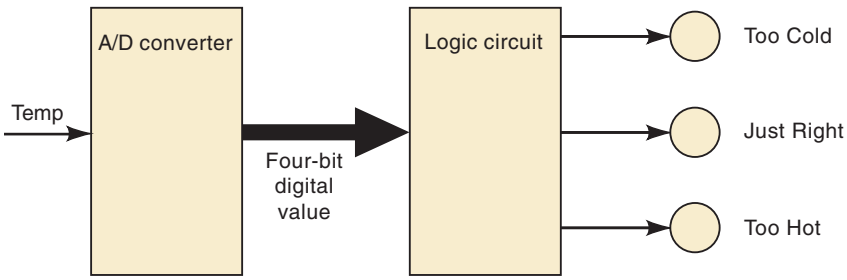


FIGURE 4-57 Temperature range indicator circuit.

The relationship between the digital values for temperature and the categories is

Digital Values	*Category*
0000–1000	Too Cold
1001–1010	Just Right
1011–1111	Too Hot

We can express the decision-making process for this logic circuit as follows:

IF the digital value is less than or equal to 8, THEN light only the Too Cold indicator.

ELSE IF the digital value is greater than 8 AND less than 11, THEN light only the Just Right indicator.

ELSE light only the Too Hot indicator.

AHDL

ELSIF USING AHDL

The AHDL code in Figure 4-58 defines the inputs as a four-bit binary number. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate variable (*status*) that allows us to assign a bit pattern representing the three conditions of *too_cold, just_right,* and *too_hot.* The sequential section of the code uses the IF, ELSIF, ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to status. In the last statement, the bits of *status* are connected to the actual output port bits. These bits have been ordered in a group that relates to the bit patterns assigned to *status[ ].* This could also have been written as three concurrent statements: too_cold = status[2]; just_right = status[1]; too_hot = status[0];.

```
SUBDESIGN fig4_58
(
   digital_value[3..0]           :INPUT;  --define inputs to block
   too_cold, just_right, too_hot :OUTPUT;--define outputs
)
VARIABLE
status[2..0]    :NODE;--holds state of too_cold, just_right, too_hot
BEGIN
   IF       digital_value[] <= 8 THEN status[] = b"100";
   ELSIF    digital_value[] > 8 AND digital_value[] < 11 THEN
         status[] = b"010";
   ELSE  status[] = b"001";
   END IF;
   (too_cold, just_right, too_hot) = status[]; -- update output bits
END;
```

FIGURE 4-58 Temperature range example in AHDL using ELSIF.

VHDL

ELSIF USING VHDL

The VHDL code in Figure 4-59 defines the inputs as a four-bit integer. The outputs are three individual bits that drive the three range indicators. This example uses an intermediate signal (*status*) that allows us to assign a bit pattern representing all three conditions of *too_cold, just_right,* and *too_hot.* The process section of the code uses the IF, ELSIF, and ELSE to identify the range in which the temperature lies and assigns the correct bit pattern to status. In the last three statements, each bit of *status* is connected to the correct output port bit.

```
ENTITY fig4_59  IS
PORT(digital_value:IN INTEGER RANGE 0 TO 15;    -- declare 4-bit input
      too_cold, just_right, too_hot :OUT BIT);
END fig4_59 ;

ARCHITECTURE howhot OF fig4_59  IS
SIGNAL status   :BIT_VECTOR (2 downto 0);
BEGIN
   PROCESS (digital_value)
      BEGIN
         IF (digital_value <= 8) THEN status <= "100";
         ELSIF (digital_value > 8 AND digital_value < 11) THEN
               status <= "010";
         ELSE  status <= "001";
         END IF;
      END PROCESS ;
   too_cold    <= status(2);     -- assign status bits to output
   just_right  <= status(1);
   too_hot     <= status(0);
END howhot;
```

FIGURE 4-59 Temperature range example in VHDL using ELSIF.

CASE

One more important control structure is useful for choosing actions based on current conditions. It is called by various names, depending on the programming language, but it nearly always involves the word **CASE**. This construct determines the value of an expression or object and then goes through a list of possible values (cases) for the expression or object being evaluated. Each case has a list of actions that should take place. A CASE construct is different from an IF/ELSIF because a case correlates one unique value of an object with a set of actions. Recall that an IF/ELSIF correlates a set of actions with a true statement. There can be only one match for a CASE statement. An IF/ELSIF can have more than one statement that is true, but will THEN perform the action associated with the first true statement it evaluates.

Another important point in the examples that follow is the need to combine several independent variables into a set of bits, called a bit vector. Recall that this action of linking several bits in a particular order is called *concatenation.* It allows us to consider the bit pattern as an ordered group.

AHDL

CASE USING AHDL

The AHDL example in Figure 4-60 demonstrates a case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status*. The CASE statement evaluates the variable *status* and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this example, it simply assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

FIGURE 4-60 Figure 4-9 represented in AHDL.

```
SUBDESIGN fig4_60
(
   p, q, r        :INPUT;      -- define inputs to block
   s              :OUTPUT;     -- define outputs
)
VARIABLE
  status[2..0]    :NODE;
BEGIN
   status[]= (p, q, r);  -- link input bits in order
   CASE status[] IS
      WHEN b"100"      => s = GND;
      WHEN b"101"      => s = GND;
      WHEN b"110"      => s = GND;
      WHEN OTHERS      => s = VCC;
   END CASE;
END;
```


CASE USING VHDL

The VHDL example in Figure 4-61 demonstrates the case construct implementing the circuit of Figure 4-9 (see also Table 4-3). It uses individual bits as its inputs. In the first statement after BEGIN, these bits are concatenated and assigned to the intermediate variable called *status* using the & operator. The CASE statement evaluates the variable status and finds the bit pattern (following the keyword WHEN) that matches the value of *status*. It then performs the action described following =>. In this simple example, it merely assigns logic 0 to the output for each of the three specified cases. All *other* cases result in a logic 1 on the output.

FIGURE 4-61 Figure 4-9 represented in VHDL.

```
ENTITY fig4_61 IS
PORT( p, q, r     :IN bit;              --declare 3 bits input
      s           :OUT BIT);
END fig4_61;

ARCHITECTURE copy OF fig4_61 IS
SIGNAL status     :BIT_VECTOR (2 downto 0);
BEGIN
   status <= p & q & r;                 --link bits in order.
   PROCESS (status)
      BEGIN
         CASE status IS
            WHEN "100" =>  s <= '0';
            WHEN "101" =>  s <= '0';
            WHEN "110" =>  s <= '0';
            WHEN OTHERS => s <= '1';
         END CASE;
      END PROCESS;
END copy;
```

EXAMPLE 4-34

A coin detector in a vending machine accepts quarters, dimes, and nickels and activates the corresponding digital signal (Q, D, N) only when the correct coin is present. It is physically impossible for multiple coins to be present at the same time. A digital circuit must use the Q, D, and N signals as inputs and produce a binary number representing the value of the coin as shown in Figure 4-62. Write the AHDL and VHDL code.

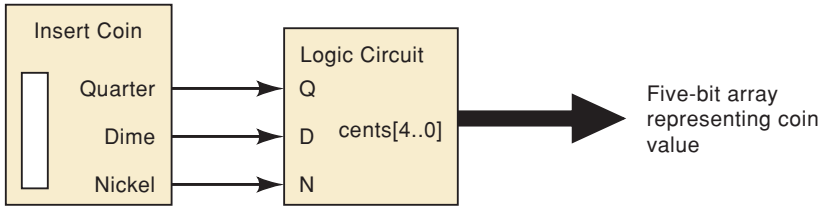


FIGURE 4-62 A coin detector circuit for a vending machine.

Solution

This is an ideal application of the CASE construct to describe the correct operation. The outputs must be declared as five-bit numbers in order to represent up to 25 cents. Figure 4-63 shows the AHDL solution and Figure 4-64 shows the VHDL solution.

AHDL

```
SUBDESIGN   fig4_63
(
   q, d, n         :INPUT;      -- define quarter, dime, nickel
   cents[4..0]     :OUTPUT;     -- define binary value of coins
)
BEGIN
   CASE (q, d, n) IS            -- group coins in an ordered set
      WHEN b"001" => cents[] = 5;
      WHEN b"010" => cents[] = 10;
      WHEN b"100" => cents[] = 25;
      WHEN others => cents[] = 0;
   END CASE;
END;
```

FIGURE 4-63 An AHDL coin detector.

VHDL

```
ENTITY   fig4_64 IS
PORT( q, d, n:IN BIT;                        -- quarter, dime, nickel
      cents  :OUT INTEGER RANGE 0 TO 25);  -- binary value of coins
END fig4_64;
ARCHITECTURE detector of fig4_64 IS
   SIGNAL   coins :BIT_VECTOR(2 DOWNTO 0);-- group the coin sensors
   BEGIN
      coins <= (q & d & n);                  -- assign sensors to group
      PROCESS (coins)
         BEGIN
            CASE (coins) IS
               WHEN "001"  => cents <= 5;
               WHEN "010"  => cents <= 10;
               WHEN "100"  => cents <= 25;
               WHEN others => cents <= 0;
            END CASE;
         END PROCESS;
   END detector;
```

FIGURE 4-64 A VHDL coin detector.

REVIEW QUESTIONS

1. Which control structure decides to do or not to do?
2. Which control structure decides to do this or to do that?
3. Which control structure(s) decides which one of several different actions to take?
4. Declare an input named *count* that can represent a numeric quantity as big as 205. Use AHDL or VHDL.

SUMMARY

1. The two general forms for logic expressions are the sum-of-products form and the product-of-sums form.
2. One approach to the design of a combinatorial logic circuit is to (1) construct its truth table, (2) convert the truth table to a sum-of-products expression, (3) simplify the expression using Boolean algebra or K mapping, (4) implement the final expression.
3. The K map is a graphical method for representing a circuit's truth table and generating a simplified expression for the circuit output.
4. An exclusive-OR circuit has the expression $x = A\overline{B} + \overline{A}B$. Its output x will be HIGH only when inputs A and B are at opposite logic levels.
5. An exclusive-NOR circuit has the expression $x = \overline{A}\,\overline{B} + AB$. Its output x will be HIGH only when inputs A and B are at the same logic level.
6. Each of the basic gates (AND, OR, NAND, NOR) can be used to enable or disable the passage of an input signal to its output.
7. The main digital IC families are the TTL and CMOS families. Digital ICs are available in a wide range of complexities (gates per chip), from the basic to the high-complexity logic functions.
8. To perform basic troubleshooting requires—at a minimum—an understanding of circuit operation, a knowledge of the types of possible faults, a complete logic-circuit connection diagram, and a logic probe.
9. A programmable logic device (PLD) is an IC that contains a large number of logic gates whose interconnections can be programmed by the user to generate the desired logic relationship between inputs and outputs.
10. To program a PLD, you need a development system that consists of a computer, PLD development software, and a programmer fixture that does the actual programming of the PLD chip.
11. The Altera system allows convenient hierarchical design techniques using any form of hardware description.
12. The type of data objects must be specified so that the HDL compiler knows the range of numbers to be represented.
13. Truth tables can be entered directly into the source file using the features of HDL.
14. Logical control structures such as IF, ELSE, and CASE can be used to describe the operation of a logic circuit, making the code and the problem's solution much more straightforward.

IMPORTANT TERMS

sum-of-products (SOP)
product-of-sums (POS)
Karnaugh map (K map)
looping
don't-care condition
exclusive-OR (XOR)
exclusive-NOR (XNOR)
parity generator
parity checker
enable/disable
dual-in-line package (DIP)
SSI, MSI, LSI, VLSI, ULSI, GSI
transistor-transistor logic (TTL)
complementary metal-oxide-semiconductor (CMOS)
indeterminate
floating
logic probe
contention
programmer
ZIF socket
JEDEC
JTAG
hierarchical design
top-down
test vectors
literals
bit array
bit vector
BIT_VECTOR
index
integer
objects
libraries
macrofunction
STD_LOGIC
STD_LOGIC_VECTOR
concatenate
selected signal assignment
decision control structure
concurrent
sequential
IF/THEN
ELSE
PROCESS
sensitivity list
ELSIF
CASE

PROBLEMS

SECTIONS 4-2 AND 4-3

B 4-1.* Simplify the following expressions using Boolean algebra.

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

(b) $y = (Q + R)(\overline{Q} + \overline{R})$

(c) $w = ABC + A\overline{B}C + \overline{A}$

(d) $q = \overline{RST}(\overline{R + S + T})$

(e) $x = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + ABC + A\,\overline{B}\,\overline{C} + A\overline{B}C$

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

(g) $y = (\overline{C + D}) + \overline{A}C\overline{D} + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}CD + AC\overline{D}$

(h) $x = AB(\overline{\overline{C}D}) + \overline{A}BD + \overline{B}\,\overline{C}\,\overline{D}$

B 4-2. Simplify the circuit of Figure 4-65 using Boolean algebra.

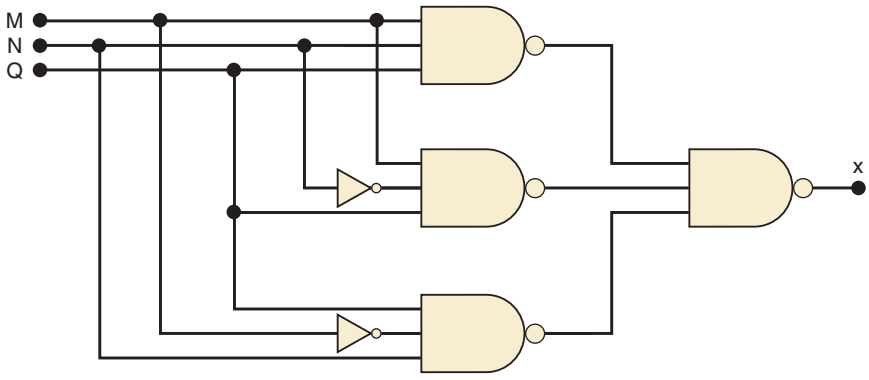


FIGURE 4-65 Problems 4-2 and 4-3.

*Answers to problems marked with an asterisk can be found in the back of the text.

B 4-3.* Change each gate in Problem 4-2 to a NOR gate, and simplify the circuit using Boolean algebra.

SECTION 4-4

B, D 4-4.* Design the logic circuit corresponding to the truth table shown in Table 4-11.

TABLE 4-11

A	B	C	x
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

B, D 4-5. Design a logic circuit whose output is HIGH *only* when a majority of inputs *A*, *B*, and *C* are LOW.

D 4-6. A manufacturing plant needs to have a horn sound to signal quitting time. The horn should be activated when either of the following conditions is met:

1. It's after 5 o'clock and all machines are shut down.
2. It's Friday, the production run for the day is complete, and all machines are shut down.

Design a logic circuit that will control the horn. (*Hint:* Use four logic input variables to represent the various conditions; for example, input *A* will be HIGH only when the time of day is 5 o'clock or later.)

D 4-7.* A four-bit binary number is represented as $A_3A_2A_1A_0$, where A_3, A_2, A_1, and A_0 represent the individual bits and A_0 is equal to the LSB. Design a logic circuit that will produce a HIGH output whenever the binary number is greater than 0010 and less than 1000.

D 4-8. Figure 4-66 shows a diagram for an automobile alarm circuit used to detect certain undesirable conditions. The three switches are used to

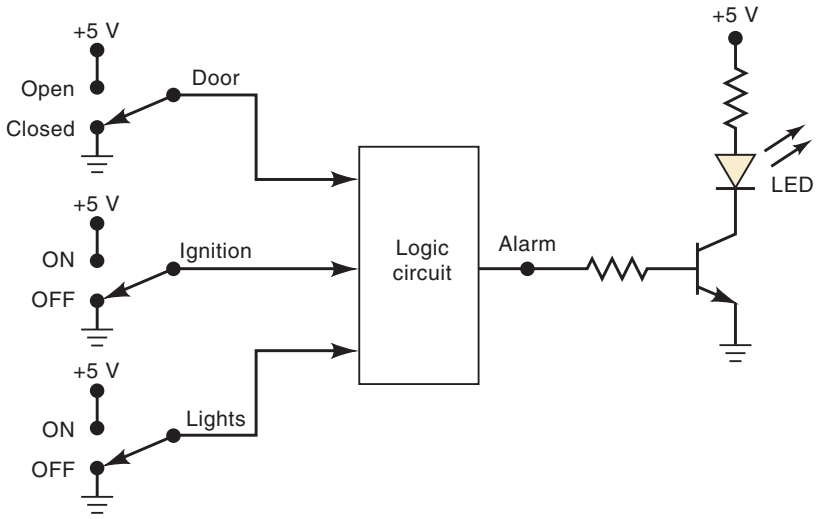


FIGURE 4-66 Problem 4-8.

indicate the status of the door by the driver's seat, the ignition, and the headlights, respectively. Design the logic circuit with these three switches as inputs so that the alarm will be activated whenever either of the following conditions exists:

- The headlights are on while the ignition is off.
- The door is open while the ignition is on.

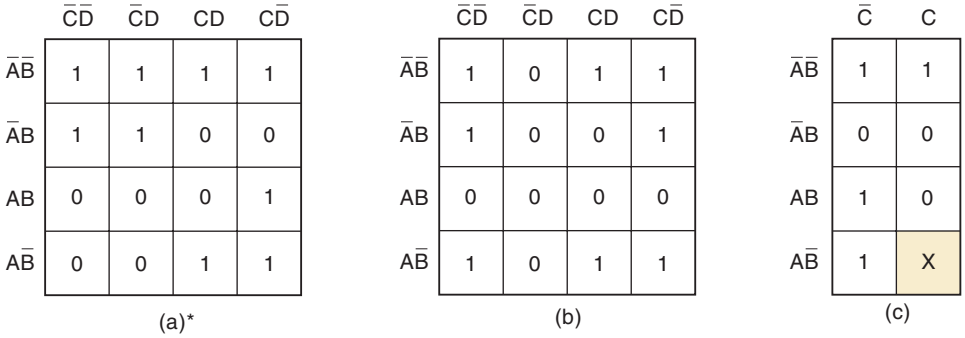
4-9.* Implement the circuit of Problem 4-4 using all NAND gates.

4-10. Implement the circuit of Problem 4-5 using all NAND gates.

SECTION 4-5

B 4-11. Determine the minimum expression for each K map in Figure 4-67. Pay particular attention to step 5 for the map in (a).

FIGURE 4-67 Problem 4-11.



B 4-12. For the truth table below, create a 2 × 2 K map, group terms, and simplify. Then look at the truth table again to see if the expression is true for every entry in the table.

A	*B*	*y*
0	0	1
0	1	1
1	0	0
1	1	0

B 4-13. Starting with the truth table in Table 4-11, use a K map to find the simplest SOP equation.

B 4-14. Simplify the expression in (a)* Problem 4-1(e) using a K map. (b) Problem 4-1(g) using a K map. (c)* Problem 4-1(h) using a K map.

B 4-15.* Obtain the output expression for Problem 4-7 using a K map.

C, D 4-16. Figure 4-68 shows a *BCD counter* that produces a four-bit output representing the BCD code for the number of pulses that have been applied to the counter input. For example, after four pulses have occurred, the counter outputs are $DCBA = 0100_2 = 4_{10}$. The counter resets to 0000 on the tenth pulse and starts counting over again. In other words, the *DCBA* outputs will never represent a number greater than $1001_2 = 9_{10}$.

(a)* Design the logic circuit that produces a HIGH output whenever the count is 2, 3, or 9. Use K mapping and take advantage of the don't-care conditions.

(b) Repeat for $x = 1$ when $DCBA = 3, 4, 5, 8$.

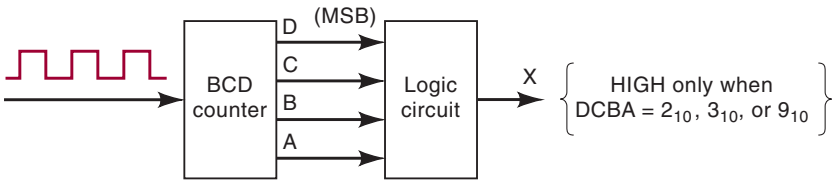


FIGURE 4-68 Problem 4-16.

D 4-17.* Figure 4-69 shows four switches that are part of the control circuitry in a copy machine. The switches are at various points along the path of the copy paper as the paper passes through the machine. Each switch is normally open, and as the paper passes over a switch, the switch closes. It is impossible for switches SW1 and SW4 to be closed at the same time. Design the logic circuit to produce a HIGH output whenever *two or more* switches are closed at the same time. Use K mapping and take advantage of the don't-care conditions.

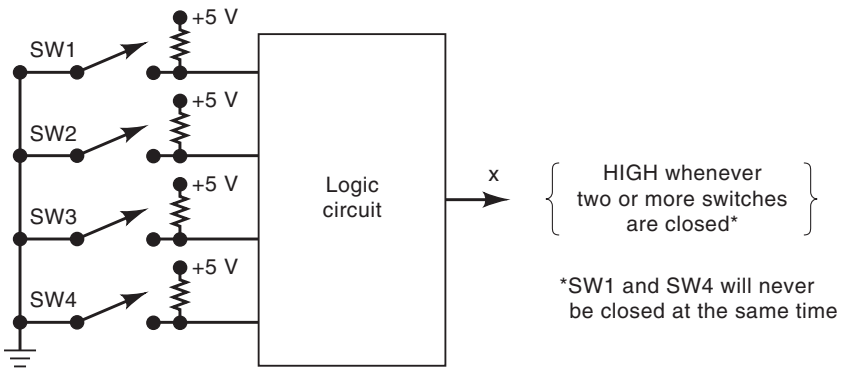


FIGURE 4-69 Problem 4-17.

B 4-18. Example 4-3 demonstrated algebraic simplification. Step 3 resulted in the SOP equation $z = \overline{A}\,\overline{B}C + \overline{A}C\overline{D} + \overline{A}B\overline{C}\,\overline{D} + A\overline{B}C$. Use a K map to prove that this equation can be simplified further than the answer shown in the example.

C 4-19. Use Boolean algebra to arrive at the same result obtained by the K map method of Problem 4-18.

SECTION 4-6

B 4-20. (a) Determine the output waveform for the circuit of Figure 4-70.
(b) Repeat with the *B* input held LOW.
(c) Repeat with *B* held HIGH.

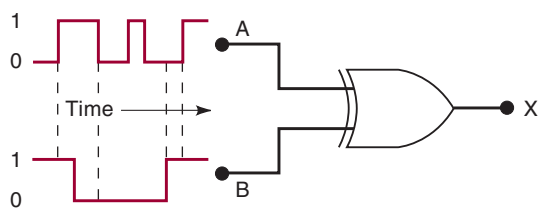


FIGURE 4-70 Problem 4-20.

B 4-21.* Determine the input conditions needed to produce $x = 1$ in Figure 4-71.

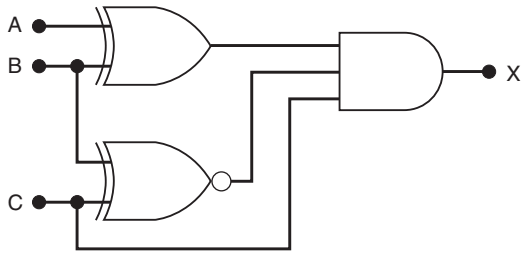


FIGURE 4-71 Problem 4-21.

B 4-22. Design a circuit that produces a HIGH out only when all three inputs are the same level.

(a) Use a truth table and K map to produce the SOP solution.

(b) Use two-input XOR and other gates to find a solution. (*Hint:* Recall the transitive property from algebra. . . if $a = b$ and $b = c$ then $a = c$.)

B 4-23.* A 7486 chip contains four XOR gates. Show how to make an XNOR gate using only a 7486 chip. *Hint:* See Example 4-16.

B 4-24.* Modify the circuit of Figure 4-23 to compare two four-bit numbers and produce a HIGH output when the two numbers match exactly.

B 4-25. Figure 4-72 represents a *relative-magnitude detector* that takes two three-bit binary numbers, $x_2x_1x_0$ and $y_2y_1y_0$, and determines whether they are equal and, if not, which one is larger. There are three outputs, defined as follows:

1. $M = 1$ only if the two input numbers are equal.
2. $N = 1$ only if $x_2x_1x_0$ is greater than $y_2y_1y_0$.
3. $P = 1$ only if $y_2y_1y_0$ is greater than $x_2x_1x_0$.

Design the logic circuitry for this detector. The circuit has *six* inputs and *three* outputs and is therefore much too complex to handle using the truth-table approach. Refer to Example 4-17 as a hint about how you might start to solve this problem.

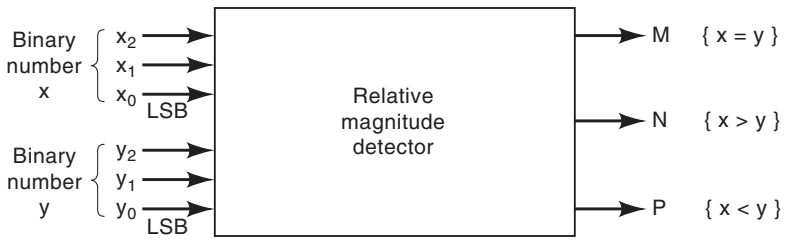


FIGURE 4-72 Problem 4-25.

MORE DESIGN PROBLEMS

C, D 4-26.* Figure 4-73 represents a multiplier circuit that takes two-bit binary numbers, x_1x_0 and y_1y_0, and produces an output binary number $z_3z_2z_1z_0$ that is equal to the arithmetic product of the two input numbers. Design the logic circuit for the multiplier. (*Hint:* The logic circuit will have four inputs and four outputs.)

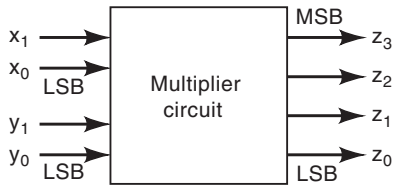


FIGURE 4-73 Problem 4-26.

D 4-27. A BCD code is being transmitted to a remote receiver. The bits are A_3, A_2, A_1, and A_0, with A_3 as the MSB. The receiver circuitry includes a *BCD error detector* circuit that examines the received code to see if it is a legal BCD code (i.e., ≤1001). Design this circuit to produce a HIGH for any error condition.

D 4-28.* Design a logic circuit whose output is HIGH whenever *A* and *B* are both HIGH as long as *C* and *D* are either both LOW or both HIGH. Try to do this without using a truth table. Then check your result by constructing a truth table from your circuit to see if it agrees with the problem statement.

D 4-29. Four large tanks at a chemical plant contain different liquids being heated. Liquid-level sensors are being used to detect whenever the level in tank *A* or tank *B* rises above a predetermined level. Temperature sensors in tanks *C* and *D* detect when the temperature in either of these tanks drops below a prescribed temperature limit. Assume that the liquid-level sensor outputs *A* and *B* are LOW when the level is satisfactory and HIGH when the level is too high. Also, the temperature-sensor outputs *C* and *D* are LOW when the temperature is satisfactory and HIGH when the temperature is too low. Design a logic circuit that will detect whenever the level in tank *A* or tank *B* is too high at the same time that the temperature in either tank *C* or tank *D* is too low.

C, D 4-30.* Figure 4-74 shows the intersection of a main highway with a secondary access road. Vehicle-detection sensors are placed along lanes *C* and *D* (main road) and lanes *A* and *B* (access road). These sensor

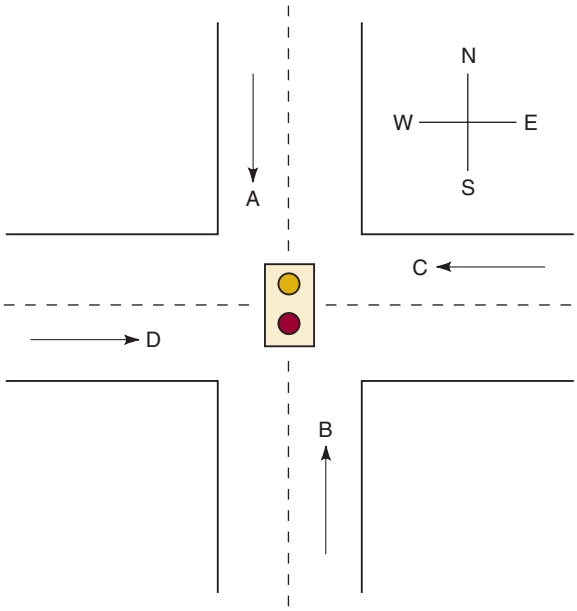


FIGURE 4-74 Problem 4-30.

outputs are LOW (0) when no vehicle is present and HIGH (1) when a vehicle is present. The intersection traffic light is to be controlled according to the following logic:

1. The east-west (E-W) traffic light will be green whenever *both* lanes *C* and *D* are occupied.
2. The E-W light will be green whenever *either C* or *D* is occupied but lanes *A* and *B* are not *both* occupied.
3. The north-south (N-S) light will be green whenever *both* lanes *A* and *B* are occupied but *C* and *D* are not *both* occupied.
4. The N-S light will also be green when *either A* or *B* is occupied while *C* and *D* are *both* vacant.
5. The E-W light will be green when *no* vehicles are present.

Using the sensor outputs *A*, *B*, *C*, and *D* as inputs, design a logic circuit to control the traffic light. There should be two outputs, N-S and E-W, that go HIGH when the corresponding light is to be *green*. Simplify the circuit as much as possible and show *all* steps.

SECTION 4-7

D 4-31. Redesign the parity generator and checker of Figure 4-25 to (a) operate using odd parity. (*Hint:* What is the relationship between an odd-parity bit and an even-parity bit for the same set of data bits?) (b) Operate on eight data bits.

SECTION 4-8

B 4-32. (a) Under what conditions will an OR gate allow a logic signal to pass through to its output unchanged?

(b) Repeat (a) for an AND gate.

(c) Repeat for a NAND gate.

(d) Repeat for a NOR gate.

B 4-33.* (a) Can an INVERTER be used as an enable/disable circuit? Explain.

(b) Can an XOR gate be used as an enable/disable circuit? Explain.

D 4-34. Design a logic circuit that will allow input signal *A* to pass through to the output only when control input *B* is LOW while control input *C* is HIGH; otherwise, the output is LOW.

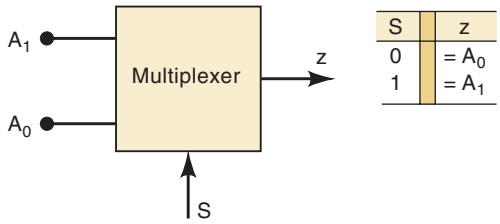
D 4-35.* Design a circuit that will *disable* the passage of an input signal only when control inputs *B*, *C*, and *D* are all HIGH; the output is to be HIGH in the disabled condition.

D 4-36. Design a logic circuit that controls the passage of a signal *A* according to the following requirements:

1. Output *X* will equal *A* when control inputs *B* and *C* are the same.
2. *X* will remain HIGH when *B* and *C* are different.

D 4-37. Design a logic circuit that has two signal inputs, A_1 and A_0, and a control input *S* so that it functions according to the requirements given in Figure 4-75. (This type of circuit is called a *multiplexer* and will be covered in Chapter 9.)

FIGURE 4-75 Problem 4-37.



D 4-38.* Use K mapping to design a circuit to meet the requirements of Example 4-17. Compare this circuit with the solution in Figure 4-23. This points out that the K-map method cannot take advantage of the XOR and XNOR gate logic. The designer must be able to determine when these gates are applicable.

SECTIONS 4-9 TO 4-13

T* 4-39. (a) A technician testing a logic circuit sees that the output of a particular INVERTER is stuck LOW while its input is pulsing. List as many possible reasons as you can for this faulty operation.

(b) Repeat part (a) for the case where the INVERTER output is stuck at an indeterminate logic level.

T 4-40.* The signals shown in Figure 4-76 are applied to the inputs of the circuit of Figure 4-32. Suppose that there is an internal open circuit at Z1-4.

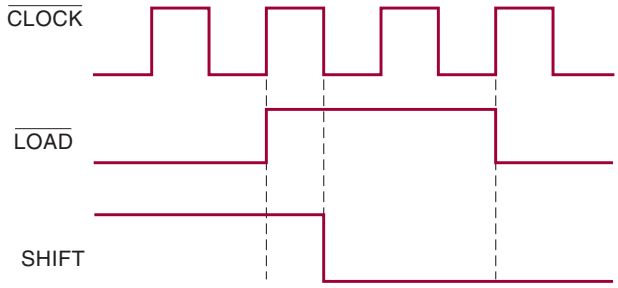
(a) What will a logic probe indicate at Z1-4?

(b) What dc voltage reading would you expect at Z1-4? (Remember that the ICs are TTL.)

(c) Sketch what you think the $\overline{CLKOUT}$ and $\overline{SHIFTOUT}$ signals will look like.

(d) Instead of the open at Z1-4, suppose that pins 9 and 10 of Z2 are internally shorted. Sketch the probable signals at Z2-10, $\overline{CLOCKOUT}$, and $\overline{SHIFTOUT}$.

FIGURE 4-76 Problem 4-40.



T 4-41. Assume that the ICs in Figure 4-32 are CMOS. Describe how the circuit operation would be affected by an open circuit in the conductor connecting Z2-2 and Z2-10.

T 4-42. In Example 4-24, we listed three possible faults for the situation of Figure 4-36. What procedure would you follow to determine which of the faults is the actual one?

T 4-43.* Refer to the circuit of Figure 4-38. Assume that the devices are CMOS. Also assume that the logic probe indication at Z2-3 is "indeterminate"

*Recall that T indicates a troubleshooting exercise.

rather than "pulsing." List the possible faults, and write a procedure to follow to determine the actual fault.

T 4-44.* Refer to the logic circuit of Figure 4-41. Recall that output Y is supposed to be HIGH for either of the following conditions:

1. $A = 1, B = 0$, regardless of C
2. $A = 0, B = 1, C = 1$

When testing the circuit, the technician observes that Y goes HIGH only for the first condition but stays LOW for all other input conditions. Consider the following list of possible faults. For each one, write yes or no to indicate whether or not it could be the actual fault. Explain your reasoning for each no response.

(a) An internal short to ground at Z2-13

(b) An open circuit in the connection to Z2-13

(c) An internal short to V_{CC} at Z2-11

(d) An open circuit in the V_{CC} connection to Z2

(e) An internal open circuit at Z2-9

(f) An open in the connection from Z2-11 to Z2-9

(g) A solder bridge between pins 6 and 7 of Z2

T 4-45. Develop a procedure for isolating the fault that is causing the malfunction described in Problem 4-44.

T 4-46.* Assume that the gates in Figure 4-41 are all CMOS. When the technician tests the circuit, he finds that it operates correctly except for the following conditions:

1. $A = 1, B = 0, C = 0$
2. $A = 0, B = 1, C = 1$

For these conditions, the logic probe indicates indeterminate levels at Z2-6, Z2-11, and Z2-8. What do you think is the probable fault in the circuit? Explain your reasoning.

T 4-47. Figure 4-77 is a combinational logic circuit that operates an alarm in a car whenever the driver and/or passenger seats are occupied and the seatbelts are not fastened when the car is started. The active-HIGH signals *DRIV* and *PASS* indicate the presence of the driver and passenger, respectively, and are taken from pressure-actuated switches in the seats. The signal *IGN* is active-HIGH when the ignition switch is on. The signal $\overline{BELTD}$ is active-LOW and indicates that the driver's seatbelt is

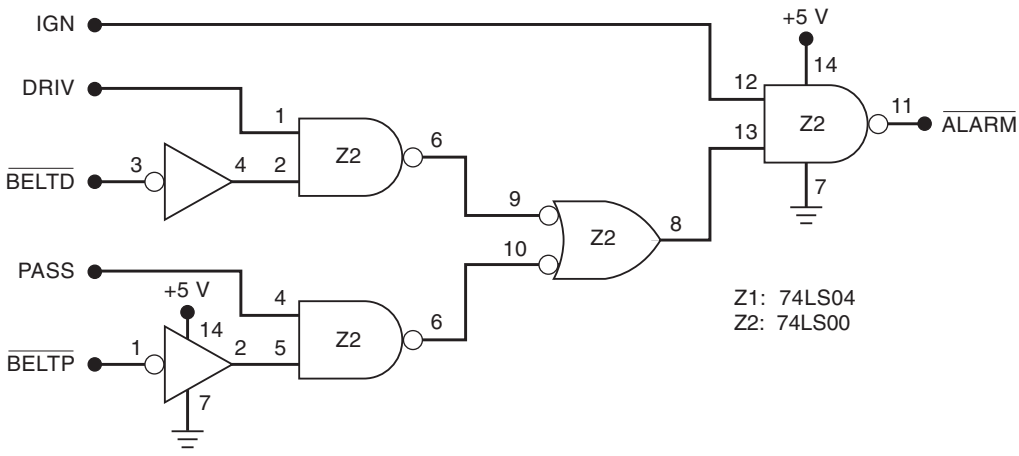


FIGURE 4-77 Problems 4-47, 4-48, and 4-49.

unfastened; $\overline{BELTP}$ is the corresponding signal for the passenger seatbelt. The alarm will be activated (LOW) whenever the car is started and either of the front seats is occupied and its seatbelt is not fastened.

(a) Verify that the circuit will function as described.

(b) Describe how this alarm system would operate if Z1-2 were internally shorted to ground.

(c) Describe how it would operate if there were an open connection from Z2-6 to Z2-10.

T 4-48.* Suppose that the system of Figure 4-77 is functioning so that the alarm is activated as soon as the driver and/or passenger are seated and the car is started, regardless of the status of the seatbelts. What are the possible faults? What procedure would you follow to find the actual fault?

T 4-49.* Suppose that the alarm system of Figure 4-77 is operating so that the alarm goes on continuously as soon as the car is started, regardless of the state of the other inputs. List the possible faults and write a procedure to isolate the fault.

DRILL QUESTIONS ON PLDs (50 THROUGH 55)

4-50.* *True or false:*

(a) Top-down design begins with an overall description of the entire system and it specifications.

(b) A JEDEC file can be used as the input file for a programmer.

(c) If an input file compiles with no errors, it means the PLD circuit will work correctly.

(d) A compiler can interpret code in spite of syntax errors.

(e) Test vectors are used to simulate and test a device.

H, B 4-51. What are the % characters used for in the AHDL design file?

H, B 4-52. How are comments indicated in a VHDL design file?

B 4-53. What is a ZIF socket?

B 4-54.* Name three entry modes used to input a circuit description into PLD development software.

B 4-55. What do JEDEC and HDL stand for?

SECTION 4-15

H, B 4-56. Declare the following data objects in AHDL or VHDL.

(a)* An array of eight output bits named *gadgets.*

(b) A single-output bit named *buzzer.*

(c) A 16-bit numeric input port named *altitude.*

(d) A single, intermediate bit within a hardware description file named *wire2.*

H, B 4-57. Express the following literal numbers in hex, binary, and decimal using the syntax of AHDL or VHDL.

(a)* 152_{10}

(b) 1001010100_2

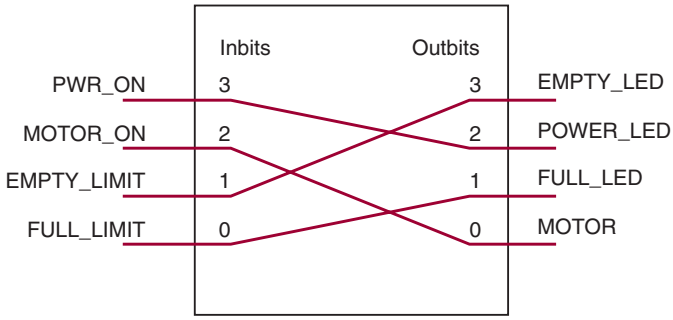
(c) $3C4_{16}$

H, B 4-58.* The following similar I/O definition is given for AHDL and VHDL. Write four concurrent assignment statements that will connect the inputs to the outputs as shown in Figure 4-78.

FIGURE 4-78 Problem 4-58.

```
SUBDESIGN hw
(
    inbits[3..0]    :INPUT;
    outbits[3..0]   :OUTPUT;
)
```

```
ENTITY hw IS
PORT (
     inbits      :IN BIT_VECTOR (3 downto 0);
     outbits     :OUT BIT_VECTOR (3 downto 0)
     );
END hw;
```



SECTION 4-16

H, D 4-59. Modify the AHDL truth table of Figure 4-50 to implement $AB + A\overline{C} + \overline{A}B$.

H, D 4-60.* Modify the AHDL design in Figure 4-54 so that $z = 1$ only when the digital value is less than 1010_2.

H, D 4-61. Modify the VHDL truth table of Figure 4-51 to implement $AB + A\overline{C} + \overline{A}B$.

H, D 4-62.* Modify the VHDL design in Figure 4-55 so that $z = 1$ only when the digital value is less than 1010_2.

H, B 4-63. Modify the code of (a) Figure 4-54 or (b) Figure 4-55 such that the output z is LOW only when digital_value is between 6 and 11 (inclusive).

H, D 4-64. Modify (a) the AHDL design in Figure 4-60 to implement Table 4-1. (b) the VHDL design in Figure 4-61 to implement Table 4-1.

H, D 4-65.* Write the hardware description design file Boolean equation to implement Example 4-9.

4-66. Write the hardware description design file Boolean equation to implement a four-bit parity generator as shown in Figure 4-25(a).

DRILL QUESTION

B 4-67. Define each of the following terms.

(a) Karnaugh map

(b) Sum-of-products form

(c) Parity generator

(d) Octet

(e) Enable circuit
(f) Don't-care condition
(g) Floating input
(h) Indeterminate voltage level
(i) Contention
(j) PLD
(k) TTL
(l) CMOS

MICROCOMPUTER APPLICATIONS

C 4-68. In a microcomputer, the microprocessor unit (MPU) is always communicating with one of the following: (1) random-access memory (RAM), which stores programs and data that can be readily changed; (2) read-only memory (ROM), which stores programs and data that never change; and (3) external input/output (I/O) devices such as keyboards, video displays, printers, and disk drives. As it is executing a program, the MPU will generate an address code that selects which type of device (RAM, ROM, or I/O) it wants to communicate with. Figure 4-79 shows a typical arrangement where the MPU outputs an eight-bit address code A_{15} through A_8. Actually, the MPU outputs a 16-bit address code, but the low-order bits A_7 through A_0 are not used in the device selection process. The address code is applied to a logic circuit that uses it to generate the device select signals: $\overline{RAM}$, $\overline{ROM}$, and $\overline{I/O}$.

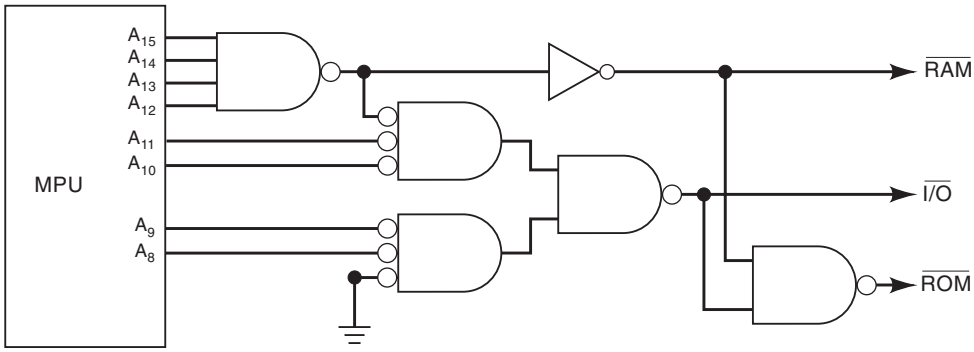


FIGURE 4-79 Problem 4-68.

Analyze this circuit and determine the following.

(a)* The range of addresses A_{15} through A_8 that will activate $\overline{\text{RAM}}$
(b) The range of addresses that activate $\overline{\text{I/O}}$
(c) The range of addresses that activate $\overline{\text{ROM}}$

Express the addresses in binary and hexadecimal. For example, the answer to (a) is A_{15} to $A_8 = 00000000_2$ to $11101111_2 = 00_{16}$ to EF_{16}.

C, D 4-69. In some microcomputers, the MPU can be *disabled* for short periods of time while another device controls the RAM, ROM, and I/O. During these intervals, a special control signal ($\overline{DMA}$) is activated by the MPU and is used to disable (deactivate) the device select logic so that the $\overline{\text{RAM}}$, $\overline{\text{ROM}}$, and $\overline{\text{I/O}}$ are all in their inactive state. Modify the circuit of Figure 4-79 so that $\overline{\text{RAM}}$, $\overline{\text{ROM}}$, and $\overline{\text{I/O}}$ will be deactivated whenever the signal $\overline{DMA}$ is active, regardless of the state of the address code.

ANSWERS TO SECTION REVIEW QUESTIONS

SECTION 4-1

1. Only (a) 2. Only (c)

SECTION 4-3

1. Expression (b) is not in sum-of-products form because of the inversion sign over both the C and D variables (i.e., the $A\overline{CD}$ term). Expression (c) is not in sum-of-products form because of the $(M + \overline{N})P$ term. 3. $x = \overline{A} + \overline{B} + \overline{C}$

SECTION 4-4

1. $x = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}C\overline{D} + \overline{A}B\overline{C}\,\overline{D}$ 2. Eight

SECTION 4-5

1. $x = AB + AC + BC$ 2. $x = A + BCD$ 3. $S = \overline{P} + QR$ 4. An input condition for which there is no specific required output condition; i.e., we are free to make it 0 or 1.

SECTION 4-6

2. A constant LOW 3. No; the available XOR gate can be used as an INVERTER by connecting one of its inputs to a constant HIGH (see Example 4-16).

SECTION 4-8

1. $x = \overline{A(B \oplus C)}$ 2. OR, NAND 3. NAND, NOR

SECTION 4-9

1. DIP 2. SSI, MSI, LSI, VLSI, ULSI, GSI 3. True 4. True 5. 40, 74AC, 74ACT series 6. 0 to 0.8 V; 2.0 to 5.0 V 7. 0 to 1.5 V; 3.5 to 5.0 V 8. As if the input were HIGH 9. Unpredictably; it may overheat and be destroyed. 10. 74HCT and 74ACT 11. They describe exactly how to interconnect the chips for laying out the circuit and troubleshooting. 12. Inputs and outputs are defined, and logical relationships are described.

SECTION 4-11

1. Open inputs or outputs; inputs or outputs shorted to V_{CC}; inputs or outputs shorted to ground; pins shorted together; internal circuit failures 2. Pins shorted together 3. For TTL, a LOW; for CMOS, indeterminate 4. Two or more outputs connected together

SECTION 4-12

1. Open signal lines; shorted signal lines; faulty power supply; output loading 2. Broken wires; poor solder connections; cracks or cuts in PC board; bent or broken IC pins; faulty IC sockets 3. ICs operating erratically or not at all 4. Logic level indeterminate

SECTION 4-14

1. Electrically controlled connections are being programmed as open or closed. 2. (4, 1) (2, 2) or (2, 1) (4, 2) 3. (4, 5) (1, 6) or (4, 6) (1, 5)

SECTION 4-15

1. (a) push_buttons[5..0] :INPUT; (b) push_buttons :IN BIT_VECTOR (5 DOWNTO 0), 2. (a) z = push_buttons[5]; (b) z <= push_buttons(5); 3. STD_LOGIC 4. STD_LOGIC_VECTOR

SECTION 4-16

1. (AHDL) omega[] = (x, y, z); (VHDL) omega <= x & y & z; 2. Using the keyword TABLE 3. Using selected signal assignments

SECTION 4-17

1. IF/THEN 2. IF/THEN/ELSE 3. CASE or IF/ELSIF
4. (AHDL) count[7..0] :INPUT; (VHDL) count :IN INTEGER RANGE 0 TO 205