Null03 1/7/03 4:35 PM Page 93

"I've always loved that word, Boolean."

—Claude Shannon

CHAPTER 3 Boolean Algebra and Digital Logic

3.1 INTRODUCTION

George Boole lived in England during the time Abraham Lincoln was getting involved in politics in the United States. Boole was a mathematician and logician who developed ways of expressing logical processes using algebraic symbols, thus creating a branch of mathematics known as *symbolic logic*, or *Boolean algebra*. It wasn't until years later that Boolean algebra was applied to computing by John Vincent Atanasoff. He was attempting to build a machine based on the same technology used by Pascal and Babbage, and wanted to use this machine to solve linear algebraic equations. After struggling with repeated failures, Atanasoff was so frustrated he decided to take a drive. He was living in Ames, Iowa, at the time, but found himself 200 miles away in Illinois before he suddenly realized how far he had driven.

Atanasoff had not intended to drive that far, but since he was in Illinois where he could legally buy a drink in a tavern, he sat down, ordered a bourbon, and realized he had driven quite a distance to get a drink! (Atanasoff reassured the author that it was not the drink that led him to the following revelations—in fact, he left the drink untouched on the table.) Exercising his physics and mathematics backgrounds and focusing on the failures of his previous computing machine, he made four critical breakthroughs necessary in the machine's new design.

He would use electricity instead of mechanical movements (vacuum tubes would allow him to do this).

Because he was using electricity, he would use base 2 numbers instead of base 10 (this correlated directly with switches that were either "on" or "off"), resulting in a digital, rather than an analog, machine.

He would use capacitors (condensers) for memory because they store electrical charges with a regenerative process to avoid power leakage.

Computations would be done by what Atanasoff termed "direct logical action" (which is essentially equivalent to Boolean algebra) and not by enumeration as all previous computing machines had done.

It should be noted that at the time, Atanasoff did not recognize the application of Boolean algebra to his problem and that he devised his own direct logical action by trial and error. He was unaware that in 1938, Claude Shannon proved that two-valued Boolean algebra could describe the operation of two-valued electrical switching circuits. Today, we see the significance of Boolean algebra's application in the design of modern computing systems. It is for this reason that we include a chapter on Boolean logic and its relationship to digital computers.

This chapter contains a brief introduction to the basics of logic design. It provides minimal coverage of Boolean algebra and this algebra's relationship to logic gates and basic digital circuits. You may already be familiar with the basic Boolean operators from a previous programming class. It is a fair question, then, to ask why you must study this material in more detail. The relationship between Boolean logic and the actual physical components of any computer system is very strong, as you will see in this chapter. As a computer scientist, you may never have to design digital circuits or other physical components—in fact, this chapter will not prepare you to design such items. Rather, it provides sufficient background for you to understand the basic motivation underlying computer design and implementation. Understanding how Boolean logic affects the design of various computer system components will allow you to use, from a programming perspective, any computer system more effectively. For the interested reader, there are many resources listed at the end of the chapter to allow further investigation into these topics.

3.2 BOOLEAN ALGEBRA

Boolean algebra is an algebra for the manipulation of objects that can take on only two values, typically true and false, although it can be any pair of values. Because computers are built as collections of switches that are either "on" or "off," Boolean algebra is a very natural way to represent digital information. In reality, digital circuits use low and high voltages, but for our level of understanding, 0 and 1 will suffice. It is common to interpret the digital value 0 as false and the digital value 1 as true.

3.2.1 Boolean Expressions

In addition to binary objects, Boolean algebra also has operations that can be performed on these objects, or variables. Combining the variables and operators yields *Boolean expressions*. A *Boolean function* typically has one or more input values and yields a result, based on these input values, in the range {0,1}.

Three common Boolean operators are *AND*, *OR*, and *NOT*. To better understand these operators, we need a mechanism to allow us to examine their behav-

Null03 1/7/03 4:35 PM Page 95

Inputs		Outputs
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 3.1 The Truth Table for AND

Inputs		Outputs
x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 3.2 The Truth Table for OR

iors. A Boolean operator can be completely described using a table that lists the inputs, all possible values for these inputs, and the resulting values of the operation for all possible combinations of these inputs. This table is called a *truth table*. A truth table shows the relationship, in tabular form, between the input values and the result of a specific Boolean operator or function on the input variables. Let's look at the Boolean operators AND, OR, and NOT to see how each is represented, using both Boolean algebra and truth tables.

The logical operator AND is typically represented by either a dot or no symbol at all. For example, the Boolean expression xy is equivalent to the expression $x \cdot y$ and is read "x and y." The expression xy is often referred to as a *Boolean product*. The behavior of this operator is characterized by the truth table shown in Table 3.1.

The result of the expression xy is 1 only when both inputs are 1, and 0 otherwise. Each row in the table represents a different Boolean expression, and all possible combinations of values for x and y are represented by the rows in the table.

The Boolean operator OR is typically represented by a plus sign. Therefore, the expression $x + y$ is read "x or y." The result of $x + y$ is 0 only when both of its input values are 0. The expression $x + y$ is often referred to as a *Boolean sum*. The truth table for OR is shown in Table 3.2.

The remaining logical operator, NOT, is represented typically by either an overscore or a prime. Therefore, both $\bar{x}$ and x' are read as "NOT x." The truth table for NOT is shown in Table 3.3.

We now understand that Boolean algebra deals with binary variables and logical operations on those variables. Combining these two concepts, we can examine Boolean expressions composed of Boolean variables and multiple logic operators. For example, the Boolean function:

$$F(x, y, z) = x + \bar{y}z$$

Inputs	Outputs
x	$\bar{x}$
0	1
1	0

TABLE 3.3 The Truth Table for NOT

Null03 1/7/03 4:35 PM Page 96

Inputs					Outputs
x	y	z	$\bar{y}$	$\bar{y}z$	$x + \bar{y}z = F$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

TABLE 3.4 **The Truth Table for F(x,y,z) = $x + \bar{y}z$**

is represented by a Boolean expression involving the three Boolean variables x, y, and z and the logical operators OR, NOT, and AND. How do we know which operator to apply first? The rules of precedence for Boolean operators give NOT top priority, followed by AND, and then OR. For our previous function F, we would negate y first, then perform the AND of $\bar{y}$ and z, and lastly OR this result with x.

We can also use a truth table to represent this expression. It is often helpful, when creating a truth table for a more complex function such as this, to build the table representing different pieces of the function, one column at a time, until the final function can be evaluated. The truth table for our function F is shown in Table 3.4.

The last column in the truth table indicates the values of the function for all possible combinations of x, y, and z. We note that the real truth table for our function F consists of only the first three columns and the last column. The shaded columns show the intermediate steps necessary to arrive at our final answer. Creating truth tables in this manner makes it easier to evaluate the function for all possible combinations of the input values.

3.2.2 Boolean Identities

Frequently, a Boolean expression is not in its simplest form. Recall from algebra that an expression such as $2x + 6x$ is not in its simplest form; it can be reduced (represented by fewer or simpler terms) to $8x$. Boolean expressions can also be simplified, but we need new *identities*, or laws, that apply to Boolean algebra instead of regular algebra. These identities, which apply to single Boolean variables as well as Boolean expressions, are listed in Table 3.5. Note that each relationship (with the exception of the last one) has both an AND (or product) form and an OR (or sum) form. This is known as the *duality principle*.

The Identity Law states that any Boolean variable ANDed with 1 or ORed with 0 simply results in the original variable. (1 is the identity element for AND; 0 is the identity element for OR.) The Null Law states that any Boolean variable ANDed with 0 is 0, and a variable ORed with 1 is always 1. The Idempotent Law states that ANDing or ORing a variable with itself produces the original variable. The Inverse Law states that ANDing or ORing a variable with its complement

Null103 1/7/03 4:35 PM Page 97

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x}+\bar{y}$	$\overline{(x+y)} = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

TABLE 3.5 Basic Identities of Boolean Algebra

produces the identity for that given operation. You should recognize the Commutative Law and Associative Law from algebra. Boolean variables can be reordered (commuted) and regrouped (associated) without affecting the final result. The Distributive Law shows how OR distributes over AND and vice versa.

The Absorption Law and DeMorgan's Law are not so obvious, but we can prove these identities by creating a truth table for the various expressions: If the right-hand side is equal to the left-hand side, the expressions represent the same function and result in identical truth tables. Table 3.6 depicts the truth table for both the left-hand side and the right-hand side of DeMorgan's Law for AND. It is left as an exercise to prove the validity of the remaining laws, in particular, the OR form of DeMorgan's Law and both forms of the Absorption Law.

The Double Complement Law formalizes the idea of the double negative, which evokes rebuke from high school teachers. The Double Complement Law can be useful in digital circuits as well as in your life. For example, let x be the amount of cash you have (assume a positive quantity). If you have no cash, you have $\bar{x}$. When an untrustworthy acquaintance asks to borrow some cash, you can truthfully say that you don't have no money. That is, $x = \overline{(\bar{x})}$ even if you just got paid.

One of the most common errors that beginners make when working with Boolean logic is to assume the following:

$\overline{(x\,y)} = \bar{x}\,\bar{y}$ Please note that this is not a valid equality!

DeMorgan's Law clearly indicates that the above statement is incorrect; however, it is a very easy mistake to make, and one that should be avoided.

x	y	(xy)	$\overline{(xy)}$	$\bar{x}$	$\bar{y}$	$\bar{x}+\bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

TABLE 3.6 Truth Tables for the AND Form of DeMorgan's Law

Null03 1/7/03 4:35 PM Page 98

3.2.3 Simplification of Boolean Expressions

The algebraic identities we studied in algebra class allow us to reduce algebraic expressions (such as $10x + 2y - x + 3y$) to their simplest forms ($9x + 5y$). The Boolean identities can be used to simplify Boolean expressions in a similar fashion. We apply these identities in the following examples.

EXAMPLE 3.1 Suppose we have the function $F(x,y) = xy + xy$. Using the OR form of the Idempotent Law and treating the expression xy as a Boolean variable, we simplify the original expression to xy. Therefore, $F(x,y) = xy + xy = xy$.

EXAMPLE 3.2 Given the function $F(x,y,z) = \bar{x}yz + \bar{x}y\bar{z} + xz$, we simplify as follows:

$$
\begin{aligned}
F(x,y,z) &= \bar{x}yz + \bar{x}y\bar{z} + xz \\
&= \bar{x}y(z + \bar{z}) + xz && \text{(Distributive)} \\
&= \bar{x}y(1) + xz && \text{(Inverse)} \\
&= \bar{x}y + xz && \text{(Identity)}
\end{aligned}
$$

At times, the simplification is reasonably straightforward, as in the preceding examples. However, using the identities can be tricky, as we see in this next example.

EXAMPLE 3.3 Given the function $F(x,y,z) = xy + \bar{x}z + yz$, we simplify as follows:

$$
\begin{aligned}
&= xy + \bar{x}z + yz(1) && \text{(Identity)} \\
&= xy + \bar{x}z + yz(x + \bar{x}) && \text{(Inverse)} \\
&= xy + \bar{x}z + (yz)x + (yz)\bar{x} && \text{(Distributive)} \\
&= xy + \bar{x}z + x(yz) + \bar{x}(zy) && \text{(Commutative)} \\
&= xy + \bar{x}z + (xy)z + (\bar{x}z)y && \text{(Associative)} \\
&= xy + (xy)z + \bar{x}z + (\bar{x}z)y && \text{(Commutative)} \\
&= xy(1 + z) + \bar{x}z(1 + y) && \text{(Distributive)} \\
&= xy(1) + \bar{x}z(1) && \text{(Null)} \\
&= xy + \bar{x}z && \text{(Identity)}
\end{aligned}
$$

Example 3.3 illustrates what is commonly known as the *Consensus Theorem.*

How did we know to insert additional terms to simplify the function? Unfortunately, there is no defined set of rules for using these identities to minimize a Boolean expression; it is simply something that comes with experience. There are other methods that can be used to simplify Boolean expressions; we mention these later in this section.

Nul103 1/7/03 4:35 PM Page 99

Proof	Identity Name
$(x+y)(\overline{x}+y) = x\overline{x}+xy+y\overline{x}+yy$	Distributive Law
$= 0+xy+y\overline{x}+yy$	Inverse Law
$= 0+xy+y\overline{x}+y$	Idempotent Law
$= xy+y\overline{x}+y$	Identity Law
$= y(x+\overline{x})+y$	Distributive Law (and Commutative Law)
$= y(1)+y$	Inverse Law
$= y+y$	Identity Law
$= y$	Idempotent Law

TABLE 3.7 Example Using Identities

We can also use these identities to prove Boolean equalities. Suppose we want to prove that $(x + y)(\overline{x} + y) = y$. The proof is given in Table 3.7.

To prove the equality of two Boolean expressions, you can also create the truth tables for each and compare. If the truth tables are identical, the expressions are equal. We leave it as an exercise to find the truth tables for the equality in Table 3.7.

3.2.4 Complements

As you saw in Example 3.1, the Boolean identities can be applied to Boolean expressions, not simply Boolean variables (we treated *xy* as a Boolean variable and then applied the Idempotent Law). The same is true for the Boolean operators. The most common Boolean operator applied to more complex Boolean expressions is the NOT operator, resulting in the *complement* of the expression. Later we will see that there is a one-to-one correspondence between a Boolean function and its physical implementation using electronic circuits. Quite often, it is cheaper and less complicated to implement the complement of a function rather than the function itself. If we implement the complement, we must invert the final output to yield the original function; this is accomplished with one simple NOT operation. Therefore, complements are quite useful.

To find the complement of a Boolean function, we use DeMorgan's Law. The OR form of this law states that $\overline{(x + y)} = \overline{x}\,\overline{y}$. We can easily extend this to three or more variables as follows:

Given the function:

$F(x,y,z) = \overline{(x + y + z)}$

Let w = $(x + y)$. Then

$F(x,y,z) = \overline{(w + z)} = \overline{w}\,\overline{\overline{z}}$

Now, applying DeMorgan's Law again, we get:

$\overline{w}\,\overline{z} = \overline{(x + y)}\;\overline{z} = \overline{x}\,\overline{y}\,\overline{z} = \overline{F}(x,y,z)$

Null03 1/7/03 4:35 PM Page 100

x	y	z	$y\bar{z}$	$\bar{x}+y\bar{z}$	$\bar{y}+z$	$x(\bar{y}+z)$
0	0	0	0	1	1	0
0	0	1	0	1	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	0	1	1

TABLE 3.8 Truth Table Representation for a Function and Its Complement

Therefore, if $F(x,y,z) = (x + y + z)$, then $\overline{F}(x,y,z) = \bar{x}\bar{y}\bar{z}$. Applying the principle of duality, we see that $\overline{(x\ y\ z)} = \bar{x} + \bar{y} + \bar{z}$.

It appears that to find the complement of a Boolean expression, we simply replace each variable by its complement (x is replaced by $\bar{x}$) and interchange ANDs and ORs. In fact, this is exactly what DeMorgan's Law is telling us to do. For example, the complement of $\bar{x} + y\bar{z}$ is $x(\bar{y} + z)$. We have to add the parentheses to ensure the correct precedence.

You can verify that this simple rule of thumb for finding the complement of a Boolean expression is correct by examining the truth tables for both the expression and its complement. The complement of any expression, when represented as a truth table, should have 0s for output everywhere the original function has 1s, and 1s in those places where the original function has 0s. Table 3.8 depicts the truth tables for $F(x,y,z) = \bar{x} + y\bar{z}$ and its complement, $\overline{F}(x,y,z) = x(\bar{y} + z)$. The shaded portions indicate the final results for F and $\overline{F}$.

3.2.5 Representing Boolean Functions

We have seen that there are many different ways to represent a given Boolean function. For example, we can use a truth table or we can use one of many different Boolean expressions. In fact, there are an infinite number of Boolean expressions that are *logically equivalent* to one another. Two expressions that can be represented by the same truth table are considered logically equivalent. See Example 3.4.

EXAMPLE 3.4 Suppose $F(x,y,z) = x + x\bar{y}$. We can also express F as $F(x,y,z) = x + x + x\bar{y}$ because the Idempotent Law tells us these two expressions are the same. We can also express F as $F(x,y,z) = x(1 + \bar{y})$ using the Distributive Law.

To help eliminate potential confusion, logic designers specify a Boolean function using a *canonical*, or *standardized*, form. For any given Boolean function, there exists a unique standardized form. However, there are different "standards" that designers use. The two most common are the sum-of-products form and the product-of-sums form.

Null03 1/7/03 4:35 PM Page 101

The *sum-of-products form* requires that the expression be a collection of ANDed variables (or product terms) that are ORed together. The function $F_1(x,y,z) = xy + y\bar{z} + xyz$ is in sum-of-products form. The function $F_2(x,y,z) = x\bar{y} + x(y + \bar{z})$ is not in sum-of-products form. We apply the Distributive Law to distribute the x variable in F_2, resulting in the expression $x\bar{y} + xy + x\bar{z}$, which is now in sum-of-products form.

Boolean expressions stated in *product-of-sums* form consist of ORed variables (sum terms) that are ANDed together. The function $F_1(x,y,z) = (x + y)(x + \bar{z})(y + \bar{z})(y + z)$ is in product-of-sums form. The product-of-sums form is often preferred when the Boolean expression evaluates true in more cases than it evaluates false. This is not the case with the function, F_1, so the sum-of-products form is appropriate. Also, the sum-of-products form is usually easier to work with and to simplify, so we use this form exclusively in the sections that follow.

Any Boolean expression can be represented in sum-of-products form. Because any Boolean expression can also be represented as a truth table, we conclude that any truth table can also be represented in sum-of-products form. It is a simple matter to convert a truth table into sum-of-products form, as indicated in the following example.

EXAMPLE 3.5 Consider a simple majority function. This is a function that, when given three inputs, outputs a 0 if less than half of its inputs are 1, and a 1 if at least half of its inputs are 1. Table 3.9 depicts the truth table for this majority function over three variables.

To convert the truth table to sum-of-products form, we start by looking at the problem in reverse. If we want the expression $x + y$ to equal 1, then either x or y (or both) must be equal to 1. If $xy + yz = 1$, then either $xy = 1$ or $yz = 1$ (or both). Using this logic in reverse and applying it to Example 3.5, we see that the function must output a 1 when $x = 0$, $y = 1$, and $z = 1$. The product term that satisfies this is $\bar{x}yz$ (clearly this is equal to 1 when $x = 0$, $y = 1$, and $z = 1$). The second occurrence of an output value of 1 is when $x = 1$, $y = 0$, and $z = 1$. The product term to guarantee an output of 1 is $x\bar{y}z$. The third product term we need is $xy\bar{z}$, and the last is xyz. In summary, to generate a sum-of-products expression using

x	y	z	F
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

TABLE 3.9 Truth Table Representation for the Majority Function

Null03 1/7/03 4:35 PM Page 102

the truth table for any Boolean expression, you must generate a product term of the input variables corresponding to each row where the value of the output variable in that row is 1. In each product term, you must then complement any variables that are 0 for that row.

Our majority function can be expressed in sum-of-products form as $F(x,y,z) = \overline{x}yz + x\overline{y}z + xy\overline{z} + xyz$. Please note that this expression may not be in simplest form; we are only guaranteeing a standard form. The sum-of-products and product-of-sums standard forms are equivalent ways of expressing a Boolean function. One form can be converted to the other through an application of Boolean identities. Whether using sum-of-products or product-of-sums, the expression must eventually be converted to its simplest form, which means reducing the expression to the minimum number of terms. Why must the expressions be simplified? A one-to-one correspondence exists between a Boolean expression and its implementation using electrical circuits, as we shall see in the next section. Unnecessary product terms in the expression lead to unnecessary components in the physical circuit, which in turn yield a suboptimal circuit.

3.3 LOGIC GATES

The logical operators AND, OR, and NOT that we have discussed have been represented thus far in an abstract sense using truth tables and Boolean expressions. The actual physical components, or *digital circuits*, such as those that perform arithmetic operations or make choices in a computer, are constructed from a number of primitive elements called *gates*. Gates implement each of the basic logic functions we have discussed. These gates are the basic building blocks for digital design. Formally, a gate is a small, electronic device that computes various functions of two-valued signals. More simply stated, a gate implements a simple Boolean function. To physically implement each gate requires from one to six or more transistors (described in Chapter 1), depending on the technology being used. To summarize, the basic physical component of a computer is the transistor; the basic logic element is the gate.

3.3.1 Symbols for Logic Gates

We initially examine the three simplest gates. These correspond to the logical operators AND, OR, and NOT. We have discussed the functional behavior of each of these Boolean operators. Figure 3.1 depicts the graphical representation of the gate that corresponds to each operator.

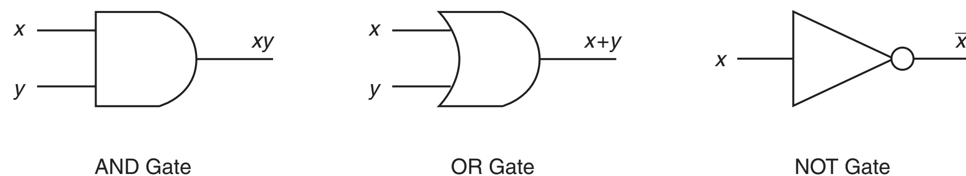
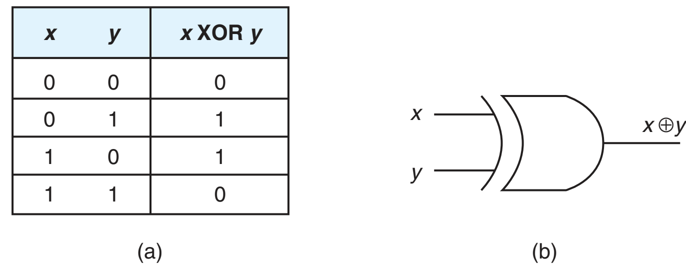


FIGURE 3.1 The Three Basic Gates

Null03 1/7/03 4:35 PM Page 103



x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

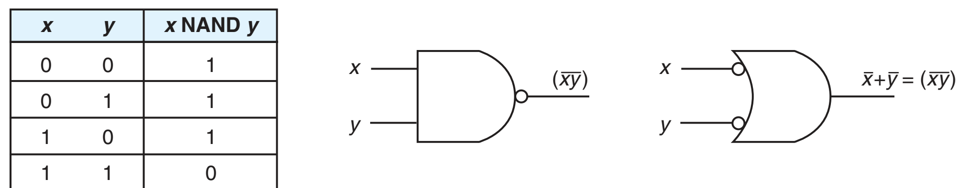
FIGURE 3.2 a) The Truth Table for XOR
b) The Logic Symbol for XOR

Note the circle at the output of the NOT gate. Typically, this circle represents the complement operation.

Another common gate is the exclusive-OR (XOR) gate, represented by the Boolean expression: $x \oplus y$. XOR is false if both of the input values are equal and true otherwise. Figure 3.2 illustrates the truth table for XOR as well as the logic diagram that specifies its behavior.

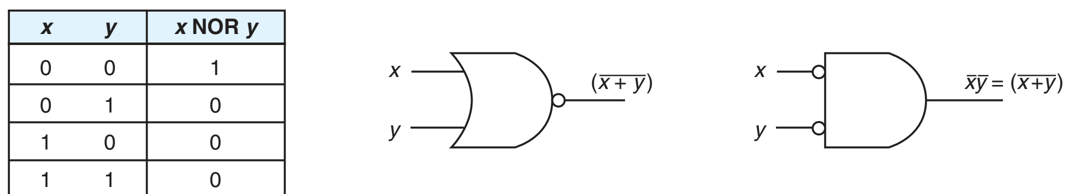
3.3.2 Universal Gates

Two other common gates are NAND and NOR, which produce complementary output to AND and OR, respectively. Each gate has two different logic symbols that can be used for gate representation. (It is left as an exercise to prove that the symbols are logically equivalent. Hint: Use DeMorgan's Law.) Figures 3.3 and 3.4 depict the logic diagrams for NAND and NOR along with the truth tables to explain the functional behavior of each gate.



x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0

FIGURE 3.3 The Truth Table and Logic Symbols for NAND



x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

FIGURE 3.4 The Truth Table and Logic Symbols for NOR

Null03 1/7/03 4:35 PM Page 104

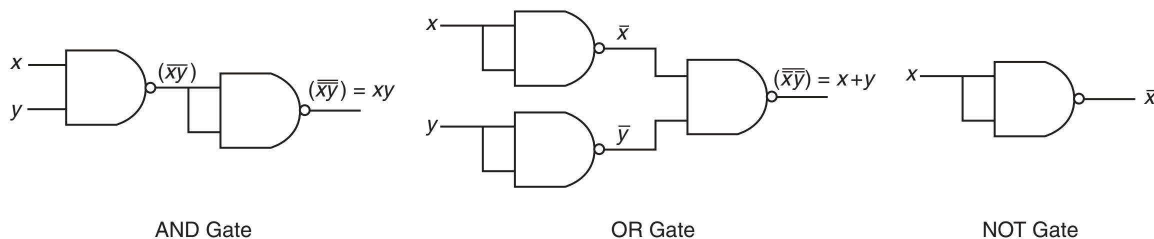


FIGURE 3.5 Three Circuits Constructed Using Only NAND Gates

The NAND gate is commonly referred to as a *universal gate*, because any electronic circuit can be constructed using only NAND gates. To prove this, Figure 3.5 depicts an AND gate, an OR gate, and a NOT gate using only NAND gates.

Why not simply use the AND, OR, and NOT gates we already know exist? There are two reasons to investigate using only NAND gates to build any given circuit. First, NAND gates are cheaper to build than the other gates. Second, complex integrated circuits (which are discussed in the following sections) are often much easier to build using the same building block (i.e., several NAND gates) rather than a collection of the basic building blocks (i.e., a combination of AND, OR, and NOT gates).

Please note that the duality principle applies to universality as well. One can build any circuit using only NOR gates. NAND and NOR gates are related in much the same way as the sum-of-products form and the product-of-sums form presented earlier. One would use NAND for implementing an expression in sum-of-products form and NOR for those in product-of-sums form.

3.3.3 Multiple Input Gates

In our examples thus far, all gates have accepted only two inputs. Gates are not limited to two input values, however. There are many variations in the number and types of inputs and outputs allowed for various gates. For example, we can represent the expression $x + y + z$ using one OR gate with three inputs, as in Figure 3.6.

Figure 3.7 represents the expression $x\bar{y}z$.

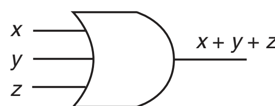


FIGURE 3.6 A Three-Input OR Gate Representing $x + y + z$

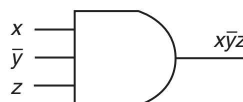


FIGURE 3.7 A Three-Input AND Gate Representing $x\bar{y}z$

Null03 1/7/03 4:35 PM Page 105

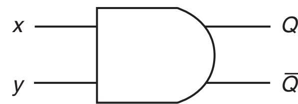


FIGURE 3.8 **AND Gate with Two Inputs and Two Outputs**

We shall see later in this chapter that it is sometimes useful to depict the output of a gate as Q along with its complement $\overline{Q}$, as shown in Figure 3.8. Note that Q always represents the actual output.

3.4 DIGITAL COMPONENTS

Upon opening a computer and looking inside, one would realize that there is a lot to know about all of the digital components that make up the system. Every computer is built using collections of gates that are all connected by way of wires acting as signal gateways. These collections of gates are often quite standard, resulting in a set of building blocks that can be used to build the entire computer system. Surprisingly, these building blocks are all constructed using the basic AND, OR, and NOT operations. In the next few sections, we discuss digital circuits, their relationship to Boolean algebra, the standard building blocks, and examples of the two different categories, combinational logic and sequential logic, into which these building blocks can be placed.

3.4.1 Digital Circuits and Their Relationship to Boolean Algebra

What is the connection between Boolean functions and digital circuits? We have seen that a simple Boolean operation (such as AND or OR) can be represented by a simple logic gate. More complex Boolean expressions can be represented as combinations of AND, OR, and NOT gates, resulting in a logic diagram that describes the entire expression. This logic diagram represents the physical implementation of the given expression, or the actual digital circuit. Consider the function $F(x,y,z) = x + \overline{y}z$ (which we looked at earlier). Figure 3.9 represents a logic diagram that implements this function.

We can build logic diagrams (which in turn lead to digital circuits) for any Boolean expression.

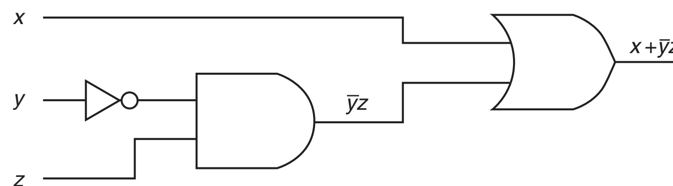


FIGURE 3.9 **A Logic Diagram for $F(x,y,z) = x + \overline{y}z$**

Null03 1/7/03 4:35 PM Page 106

Boolean algebra allows us to analyze and design digital circuits. Because of the relationship between Boolean algebra and logic diagrams, we simplify our circuit by simplifying our Boolean expression. Digital circuits are implemented with gates, but gates and logic diagrams are not the most convenient forms for representing digital circuits during the design phase. Boolean expressions are much better to use during this phase because they are easier to manipulate and simplify.

The complexity of the expression representing a Boolean function has a direct impact on the complexity of the resulting digital circuit; the more complex the expression, the more complex the resulting circuit. We should point out that we do not typically simplify our circuits using Boolean identities; we have already seen that this can sometimes be quite difficult and time consuming. Instead, designers use a more automated method to do this. This method involves the use of *Karnaugh maps* (or *Kmaps*). The interested reader is referred to the focus section following this chapter to learn how Kmaps help to simplify digital circuits.

3.4.2 Integrated Circuits

Computers are composed of various digital components, connected by wires. Like a good program, the actual hardware of a computer uses collections of gates to create larger modules, which, in turn, are used to implement various functions. The number of gates required to create these "building blocks" depends on the technology being used. Because the circuit technology is beyond the scope of this text, the reader is referred to the reading list at the end of this chapter for more information on circuit technology.

Typically, gates are not sold individually; they are sold in units called *integrated circuits* (*ICs*). A chip (a small silicon semiconductor crystal) is a small electronic device consisting of the necessary electronic components (transistors, resistors, and capacitors) to implement various gates. As described in Chapter 1, components are etched directly on the chip, allowing them to be smaller and to require less power for operation than their discrete component counterparts. This chip is then mounted in a ceramic or plastic container with external pins. The necessary connections are welded from the chip to the external pins to form an IC. The first ICs contained very few transistors. As we learned in Chapter 1, the first ICs were called SSI chips and contained up to 100 electronic components per chip. We now have ULSI (ultra large-scale integration) with more than 1 million electronic components per chip. Figure 3.10 illustrates a simple SSI IC.

3.5 COMBINATIONAL CIRCUITS

Digital logic chips are combined to give us useful circuits. These logic circuits can be categorized as either *combinational logic* or *sequential logic***.** This section introduces combinational logic. Sequential logic is covered in Section 3.6.

Nul103 1/7/03 4:35 PM Page 107

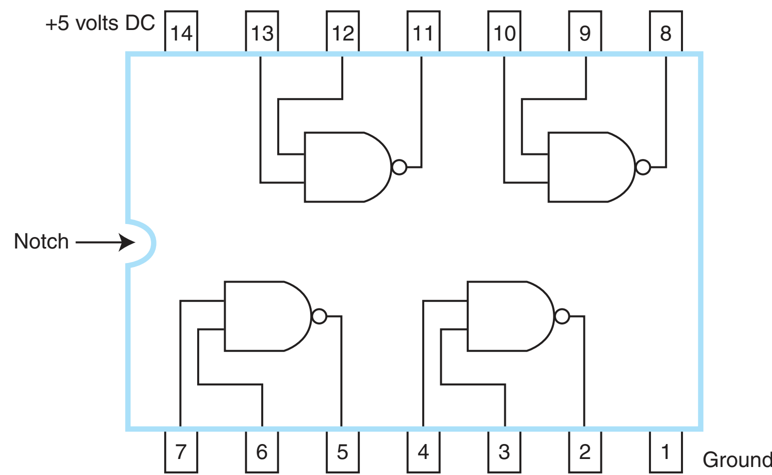


FIGURE 3.10 **A Simple SSI Integrated Circuit**

3.5.1 Basic Concepts

Combinational logic is used to build circuits that contain basic Boolean operators, inputs, and outputs. The key concept in recognizing a combinational circuit is that an output is always based entirely on the given inputs. Thus, the output of a combinational circuit is a function of its inputs, and the output is uniquely determined by the values of the inputs at any given moment. A given combinational circuit may have several outputs. If so, each output represents a different Boolean function.

3.5.2 Examples of Typical Combinational Circuits

Let's begin with a very simple combinational circuit called a *half-adder*. Consider the problem of adding two binary digits together. There are only three things to remember: 0 + 0 = 0, 0 + 1 = 1 + 0 = 1, and 1 + 1 = 10. We know the behavior this circuit exhibits, and we can formalize this behavior using a truth table. We need to specify two outputs, not just one, because we have a sum and a carry to address. The truth table for a half-adder is shown in Table 3.10.

A closer look reveals that Sum is actually an XOR. The Carry output is equivalent to that of an AND gate. We can combine an XOR gate and an AND gate, resulting in the logic diagram for a half-adder shown in Figure 3.11.

The half-adder is a very simple circuit and not really very useful because it can only add two bits together. However, we can extend this adder to a circuit that allows the addition of larger binary numbers. Consider how you add base 10 numbers: You add up the rightmost column, note the units digit, and carry the tens digit. Then you add that carry to the current column, and continue in a similar fashion. We can add binary numbers in the same way. However, we need a

Null03 1/7/03 4:35 PM Page 108

Inputs		Outputs	
x	*y*	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLE 3.10 The Truth Table for a Half-Adder

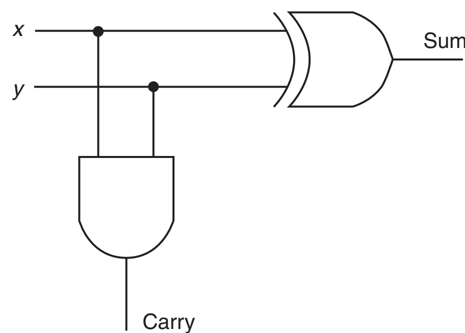


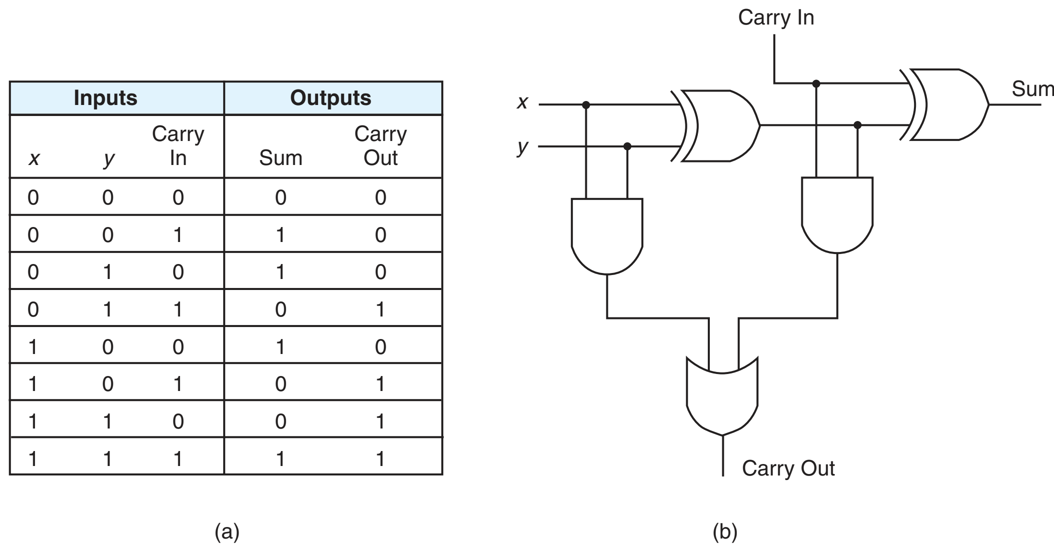
FIGURE 3.11 The Logic Diagram for a Half-Adder

circuit that allows three inputs (*x*, *y*, and Carry In), and two outputs (Sum and Carry Out). Figure 3.12 illustrates the truth table and corresponding logic diagram for a *full-adder*. Note that this full-adder is composed of two half-adders and an OR gate.

Given this full-adder, you may be wondering how this circuit can add binary numbers, since it is capable of adding only three bits. The answer is, it can't. However, we can build an adder capable of adding two 16-bit words, for example, by replicating the above circuit 16 times, feeding the Carry Out of one circuit into the Carry In of the circuit immediately to its left. Figure 3.13 illustrates this idea. This type of circuit is called a *ripple-carry adder* because of the sequential generation of carries that "ripple" through the adder stages. Note that instead of drawing all the gates that constitute a full-adder, we use a *black box* approach to depict our adder. A black box approach allows us to ignore the details of the actual gates. We concern ourselves only with the inputs and outputs of the circuit. This is typically done with most circuits, including decoders, multiplexers, and adders, as we shall see very soon.

Because this adder is very slow, it is not normally implemented. However, it is easy to understand and should give you some idea of how addition of larger binary numbers can be achieved. Modifications made to adder designs have resulted in the carry-look-ahead adder, the carry-select adder, and the carry-save adder, as well as others. Each attempts to shorten the delay required to add two binary numbers.

Nul103 1/7/03 4:35 PM Page 109



Inputs			Outputs	
x	*y*	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

FIGURE 3.12 a) A Truth Table for a Full-Adder
b) A Logic Diagram for a Full-Adder

FIGURE 3.13 The Logic Diagram for a Ripple-Carry Adder

In fact, these newer adders achieve speeds 40% to 90% faster than the ripple-carry adder by performing additions in parallel and reducing the maximum carry path.

Adders are very important circuits—a computer would not be very useful if it could not add numbers. An equally important operation that all computers use frequently is decoding binary information from a set of n inputs to a maximum of 2^n outputs. A *decoder* uses the inputs and their respective values to select one specific output line. What do we mean by "select an output line"? It simply means that one unique output line is asserted, or set to 1, while the other output lines are set to zero. Decoders are normally defined by the number of inputs and the number of outputs. For example, a decoder that has 3 inputs and 8 outputs is called a 3-to-8 decoder.

We mentioned that this decoder is something the computer uses frequently. At this point, you can probably name many arithmetic operations the computer must be able to perform, but you might find it difficult to propose an example of decoding. If so, it is because you are not familiar with how a computer accesses memory.

All memory addresses in a computer are specified as binary numbers. When a memory address is referenced (whether for reading or for writing), the computer

Null03 1/7/03 4:35 PM Page 110

first has to determine the actual address. This is done using a decoder. The following example should clarify any questions you may have about how a decoder works and what it might be used for.

EXAMPLE 3.6 A 3-to-8 decoder circuit

Imagine memory consisting of 8 chips, each containing 8K bytes. Let's assume chip 0 contains memory addresses 0–8191, chip 1 contains memory addresses 8192–16,383, and so on. We have a total of 8K $\times$ 8, or 64K (65,536) addresses available. We will not write down all 64K addresses as binary numbers; however, writing a few addresses in binary form (as we illustrate in the following paragraphs) will illustrate why a decoder is necessary.

Given $64 = 2^6$ and $1K = 2^{10}$, then $64K = 2^6 \times 2^{10} = 2^{16}$, which indicates we need 16 bits to represent each address. If you have trouble understanding this, start with a smaller number of addresses. For example, if you have 4 addresses—addresses 0, 1, 2, and 3, the binary equivalent of these addresses is 00, 01, 10, and 11, requiring two bits. We know $2^2 = 4$. Now consider eight addresses. We have to be able to count from 0 to 7 in binary. How many bits does that require? The answer is 3. You can either write them all down, or you recognize that $8 = 2^3$. The exponent tells us the minimum number of bits necessary to represent the addresses.

All addresses on chip 0 have the format: 000*xxxxxxxxxxxxx*. Because chip 0 contains the addresses 0–8191, the binary representation of these addresses is in the range 0000000000000000 to 0001111111111111. Similarly, all addresses on chip 1 have the format 001*xxxxxxxxxxxxx*, and so on for the remaining chips. The leftmost 3 bits determine on which chip the address is actually located. We need 16 bits to represent the entire address, but on each chip, we only have 2^{13} addresses. Therefore, we need only 13 bits to uniquely identify an address on a given chip. The rightmost 13 bits give us this information.

When a computer is given an address, it must first determine which chip to use; then it must find the actual address on that specific chip. In our example, the computer would use the 3 leftmost bits to pick the chip and then find the address on the chip using the remaining 13 bits. These 3 high-order bits are actually used as the inputs to a decoder so the computer can determine which chip to activate for reading or writing. If the first 3 bits are 000, chip 0 should be activated. If the first 3 bits are 111, chip 7 should be activated. Which chip would be activated if the first 3 bits were 010? It would be chip 2. Turning on a specific wire activates a chip. The output of the decoder is used to activate one, and only one, chip as the addresses are decoded.

Figure 3.14 illustrates the physical components in a decoder and the symbol often used to represent a decoder. We will see how a decoder is used in memory in Section 3.6.

Another common combinational circuit is a *multiplexer*. This circuit selects binary information from one of many input lines and directs it to a single output line. Selection of a particular input line is controlled by a set of selection vari-

Null03 1/7/03 4:35 PM Page 111

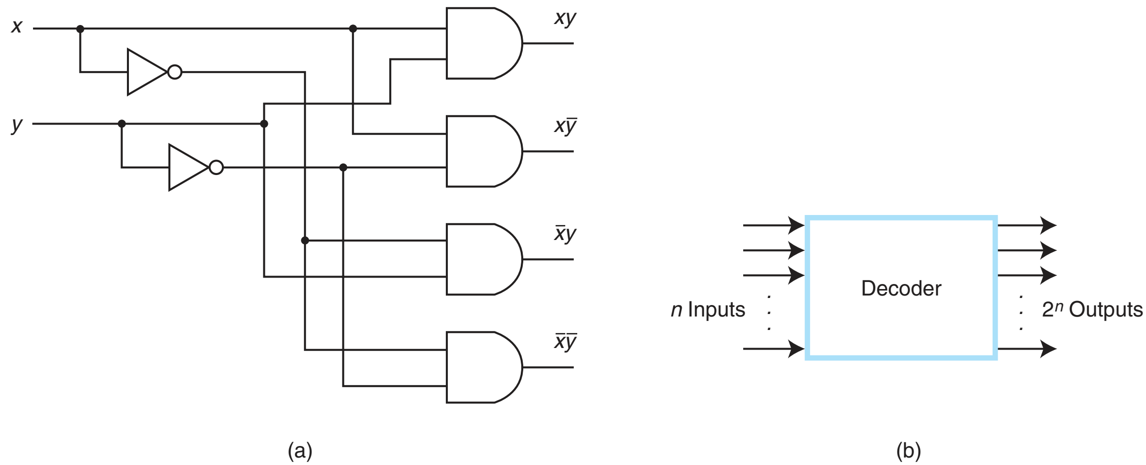


FIGURE 3.14 a) A Look Inside a Decoder
b) A Decoder Symbol

ables, or control lines. At any given time, only one input (the one selected) is routed through the circuit to the output line. All other inputs are "cut off." If the values on the control lines change, the input actually routed through changes as well. Figure 3.15 illustrates the physical components in a multiplexer and the symbol often used to represent a multiplexer.

Can you think of some situations that require multiplexers? Time-sharing computers multiplex the input from user terminals. Modem pools multiplex the modem lines entering the computer.

Another useful set of combinational circuits to study includes a parity generator and a parity checker (recall we studied parity in Chapter 2). A *parity generator* is a circuit that creates the necessary parity bit to add to a word; a *parity*

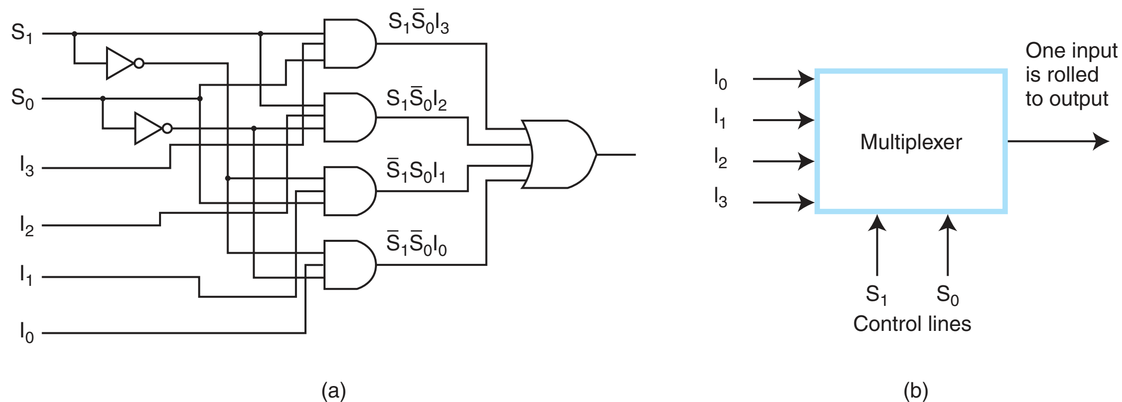


FIGURE 3.15 a) A Look Inside a Multiplexer
b) A Multiplexer Symbol

Null03 1/7/03 4:35 PM Page 112

checker checks to make sure proper parity (odd or even) is present in the word, detecting an error if the parity bit is incorrect.

Typically parity generators and parity checkers are constructed using XOR functions. Assuming we are using odd parity, the truth table for a parity generator for a 3-bit word is given in Table 3.11. The truth table for a parity checker to be used on a 4-bit word with 3 information bits and 1 parity bit is given in Table 3.12. The parity checker outputs a 1 if an error is detected and 0 otherwise. We leave it as an exercise to draw the corresponding logic diagrams for both the parity generator and the parity checker.

There are far too many combinational circuits for us to be able to cover them all in this brief chapter. Comparators, shifters, programmable logic devices—these are all valuable circuits and actually quite easy to understand. The interested reader is referred to the references at the end of this chapter for more information on combinational circuits. However, before we finish the topic of combinational logic, there is one more combinational circuit we need to introduce. We have covered all of the components necessary to build an *arithmetic logic unit* (*ALU*).

Figure 3.16 illustrates a very simple ALU with four basic operations—AND, OR, NOT, and addition—carried out on two machine words of 2 bits each. The control lines, f_0 and f_1, determine which operation is to be performed by the

x	*y*	*z*	Parity Bit
0	0	0	1
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	0

TABLE 3.11 Parity Generator

x	*y*	*z*	*P*	Error Detected?
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

TABLE 3.12 Parity Checker

Null03 1/7/03 4:35 PM Page 113

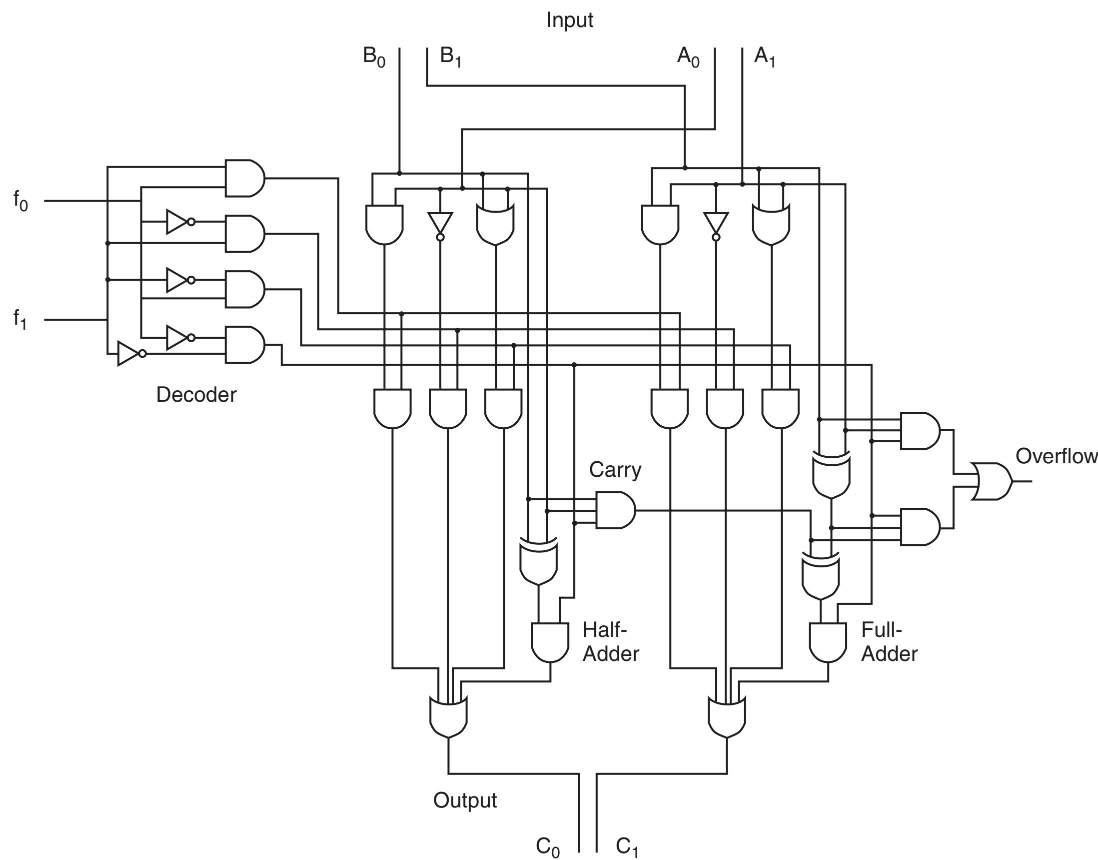


FIGURE 3.16 A Simple Two-Bit ALU

CPU. The signal 00 is used for addition (A + B); 01 for NOT A; 10 for A OR B, and 11 for A AND B. The input lines A_0 and A_1 indicate 2 bits of one word, while B_0 and B_1 indicate the second word. C_0 and C_1 represent the output lines.

3.6 SEQUENTIAL CIRCUITS

In the previous section we studied combinational logic. We have approached our study of Boolean functions by examining the variables, the values for those variables, and the function outputs that depend solely on the values of the inputs to the functions. If we change an input value, this has a direct and immediate impact on the value of the output. The major weakness of combinational circuits is that there is no concept of storage—they are memoryless. This presents us with a bit of a dilemma. We know that computers must have a way to remember values. Consider a much simpler digital circuit needed for a soda machine. When you put

Null03 1/7/03 4:35 PM Page 114

money into a soda machine, the machine remembers how much you have put in at any given instant. Without this ability to remember, it would be very difficult to use. A soda machine cannot be built using only combinational circuits. To understand how a soda machine works, and ultimately how a computer works, we must study sequential logic.

3.6.1 Basic Concepts

A sequential circuit defines its output as a function of both its current inputs and its previous inputs. Therefore, the output depends on past inputs. To remember previous inputs, sequential circuits must have some sort of storage element. We typically refer to this storage element as a *flip-flop*. The state of this flip-flop is a function of the previous inputs to the circuit. Therefore, pending output depends on both the current inputs and the current state of the circuit. In the same way that combinational circuits are generalizations of gates, sequential circuits are generalizations of flip-flops.

3.6.2 Clocks

Before we discuss sequential logic, we must first introduce a way to order events. (The fact that a sequential circuit uses past inputs to determine present outputs indicates we must have event ordering.) Some sequential circuits are *asynchronous*, which means they become active the moment any input value changes. *Synchronous* sequential circuits use clocks to order events. A *clock* is a circuit that emits a series of pulses with a precise pulse width and a precise interval between consecutive pulses. This interval is called the *clock cycle time*. Clock speed is generally measured in megahertz (MHz), or millions of pulses per second. Common cycle times are from one to several hundred MHz.

A clock is used by a sequential circuit to decide when to update the state of the circuit (when do "present" inputs become "past" inputs?). This means that inputs to the circuit can only affect the storage element at given, discrete instances of time. In this chapter we examine synchronous sequential circuits because they are easier to understand than their asynchronous counterparts. From this point, when we refer to "sequential circuit," we are implying "synchronous sequential circuit."

Most sequential circuits are edge-triggered (as opposed to being level-triggered). This means they are allowed to change their states on either the rising or falling edge of the clock signal, as seen in Figure 3.17.

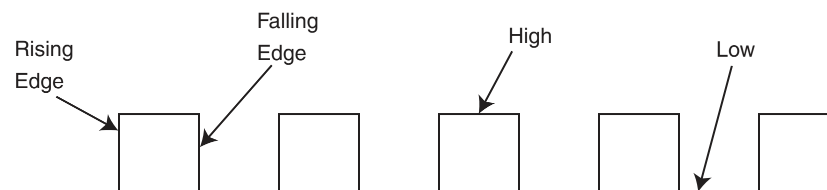


FIGURE 3.17 A Clock Signal Indicating Discrete Instances of Time

Null03 1/7/03 4:35 PM Page 115

3.6.3 Flip-Flops

A level-triggered circuit is allowed to change state whenever the clock signal is either high or low. Many people use the terms *latch* and *flip-flop* interchangeably. Technically, a latch is level triggered, whereas a flip-flop is edge triggered. In this book, we use the term *flip-flop*.

In order to "remember" a past state, sequential circuits rely on a concept called *feedback*. This simply means the output of a circuit is fed back as an input to the same circuit. A very simple feedback circuit uses two NOT gates, as shown in Figure 3.18.

In this figure, if Q is 0, it will always be 0. If Q is 1, it will always be 1. This is not a very interesting or useful circuit, but it allows you to see how feedback works.

A more useful feedback circuit is composed of two NOR gates resulting in the most basic memory unit called an *SR flip-flop*. SR stands for "set/reset." The logic diagram for the SR flip-flop is given in Figure 3.19.

We can describe any flip-flop by using a *characteristic table*, which indicates what the next state should be based on the inputs and the current state, Q. The notation Q(t) represents the current state, and $Q(t + 1)$ indicates the next state, or the state the flip-flop should enter after the clock has been pulsed. Figure 3.20 shows the actual implementation of the SR sequential circuit and its characteristic table.

An SR flip-flop exhibits interesting behavior. There are three inputs: S, R, and the current output Q(t). We create the truth table shown in Table 3.13 to illustrate how this circuit works.

For example, if S is 0 and R is 0, and the current state, Q(t), is 0, then the next state, Q(t + 1), is also 0. If S is 0 and R is 0, and Q(t) is 1, then Q(t+1) is 1. Actual inputs of (0,0) for (S,R) result in no change when the clock is pulsed. Following a similar argument, we can see that inputs (S,R) = (0,1) force the next state, Q(t + 1), to 0 regardless of the current state (thus forcing a *reset* on the circuit output). When (S,R) = (1,0), the circuit output is *set* to 1.

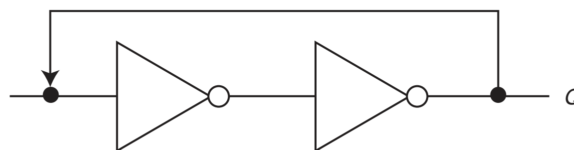


FIGURE 3.18 **Example of Simple Feedback**

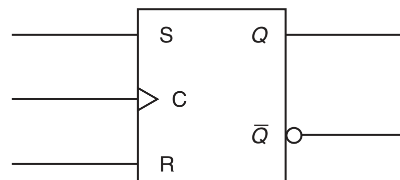
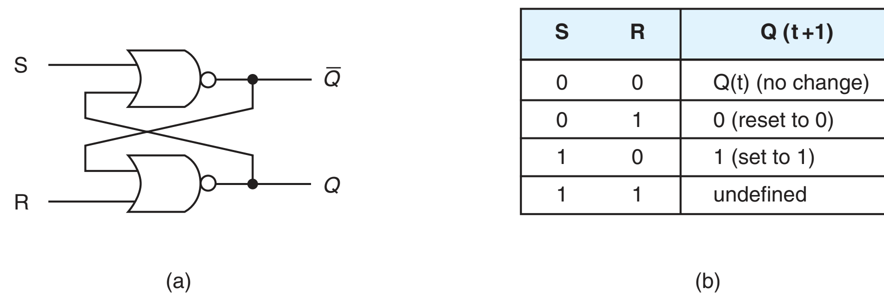


FIGURE 3.19 **An SR Flip-Flop Logic Diagram**

Nul103 1/7/03 4:35 PM Page 116



S	R	Q (t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

FIGURE 3.20 a) The Actual SR Flip-Flop
b) The Characteristic Table for the SR Flip-Flop

S	R	Present State Q(t)	Next State Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

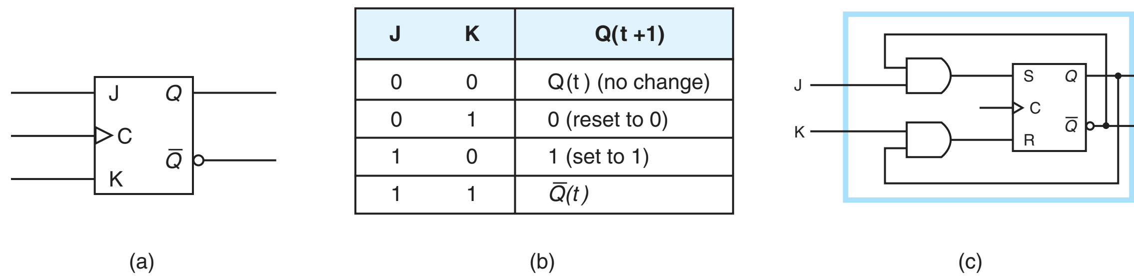
TABLE 3.13 Truth Table for SR Flip-Flop

There is one oddity with this particular flip-flop. What happens if both S and R are set to 1 at the same time? This forces both Q and $\overline{Q}$ to 1, but how can Q = 1 = $\overline{Q}$? This results in an unstable circuit. Therefore, this combination of inputs is not allowed in an SR flip-flop.

We can add some conditioning logic to our SR flip-flop to ensure that the illegal state never arises—we simply modify the SR flip-flop as shown in Figure 3.21. This results in a *JK flip-flop*. JK flip-flops were named after the Texas Instruments engineer, Jack Kilby, who invented the integrated circuit in 1958.

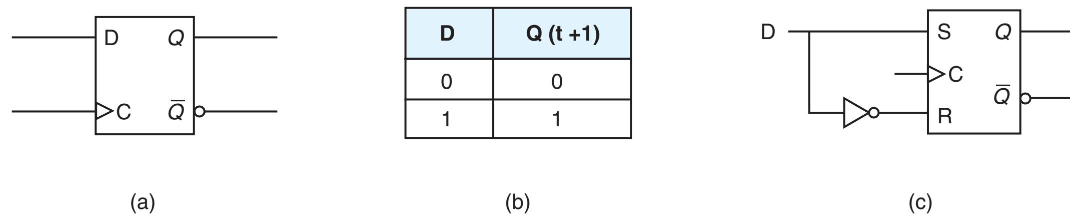
Another variant of the SR flip-flop is the *D (data) flip-flop*. A D flip-flop is a true representation of physical computer memory. This sequential circuit stores one bit of information. If a 1 is asserted on the input line D, and the clock is pulsed, the output line Q becomes a 1. If a 0 is asserted on the input line and the clock is pulsed, the output becomes 0. Remember that output Q represents the current state of the circuit. Therefore, an output value of 1 means the circuit is currently "storing" a value of 1. Figure 3.22 illustrates the D flip-flop, lists its characteristic table, and reveals that the D flip-flop is actually a modified SR flip-flop.

Null03 1/7/03 4:35 PM Page 117



J	K	Q(t +1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}(t)$

FIGURE 3.21 **a) A JK Flip-Flop**
b) The JK Characteristic Table
c) A JK Flip-Flop as a Modified SR Flip-Flop



D	Q (t +1)
0	0
1	1

FIGURE 3.22 **a) A D Flip-Flop**
b) The D Characteristic Table
c) A D Flip-Flop as a Modified SR Flip-Flop

3.6.4 Examples of Sequential Circuits

Latches and flip-flops are used to implement more complex sequential circuits. Registers, counters, memories, and shift registers all require the use of storage, and are therefore implemented using sequential logic.

Our first example of a sequential circuit is a simple 4-bit register implemented using four D flip-flops. (To implement registers for larger words, we would simply need to add flip-flops.) There are four input lines, four output lines, and a clock signal line. The clock is very important from a timing standpoint; the registers must all accept their new input values and change their storage elements at the same time. Remember that a synchronous sequential circuit cannot change state unless the clock pulses. The same clock signal is tied into all four D flip-flops, so they change in unison. Figure 3.23 depicts the logic diagram for our 4-bit register, as well as a block diagram for the register. In reality, physical components have additional lines for power and for ground, as well as a clear line (which gives the ability to reset the entire register to all zeros). However, in this text, we are willing to leave those concepts to the computer engineers and focus on the actual digital logic present in these circuits.

Another useful sequential circuit is a binary counter, which goes through a predetermined sequence of states as the clock pulses. In a straight binary counter, these states reflect the binary number sequence. If we begin counting in binary:

Null03 1/7/03 4:35 PM Page 118

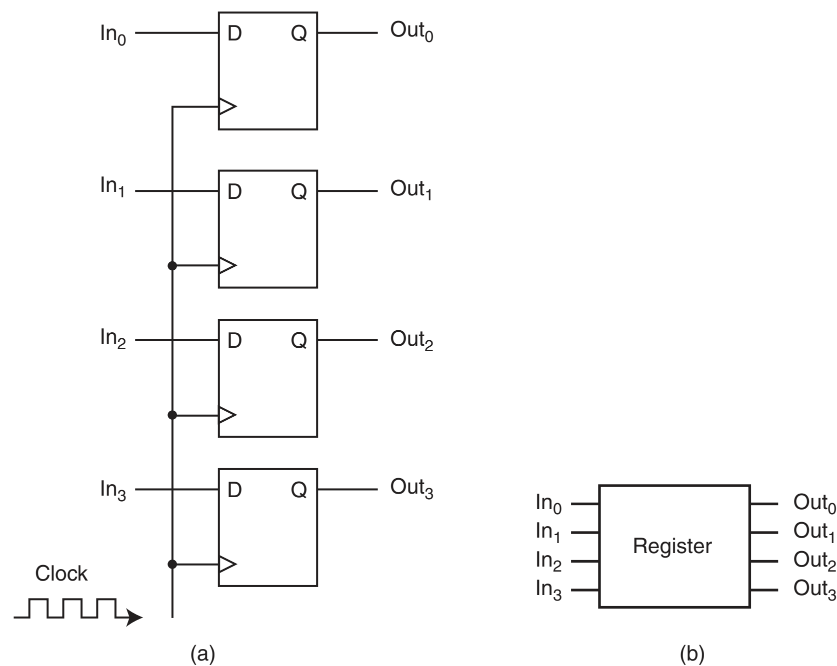


FIGURE 3.23 a) A 4-Bit Register
b) A Block Diagram for a 4-Bit Register

0000, 0001, 0010, 0011, . . . , we can see that as the numbers increase, the low-order bit is complemented each time. Whenever it changes state from 1 to 0, the bit to the left is then complemented. Each of the other bits changes state from 0 to 1 when all bits to the right are equal to 1. Because of this concept of complementing states, our binary counter is best implemented using a JK flip-flop (recall that when J and K are both equal to 1, the flip-flop complements the present state). Instead of independent inputs to each flip-flop, there is a *count enable line* that runs to each flip-flop. The circuit counts only when the clock pulses and this count enable line is set to 1. If count enable is set to 0 and the clock pulses, the circuit does not change state. You should examine Figure 3.24 very carefully, tracing the circuit with various inputs to make sure you understand how this circuit outputs the binary numbers from 0000 to 1111. You should also check to see which state the circuit enters if the current state is 1111 and the clock is pulsed.

We have looked at a simple register and a binary counter. We are now ready to examine a very simple memory circuit.

The memory depicted in Figure 3.25 holds four 3-bit words (this is typically denoted as a 4×3 memory). Each column in the circuit represents one 3-bit word. Notice that the flip-flops storing the bits for each word are synchronized via the clock signal, so a read or write operation always reads or writes a complete word. The inputs In_0, In_1, and In_2 are the lines used to store, or write, a 3-bit word to memory. The lines S_0 and S_1 are the address lines used to select which word in

Null03 1/7/03 4:35 PM Page 119

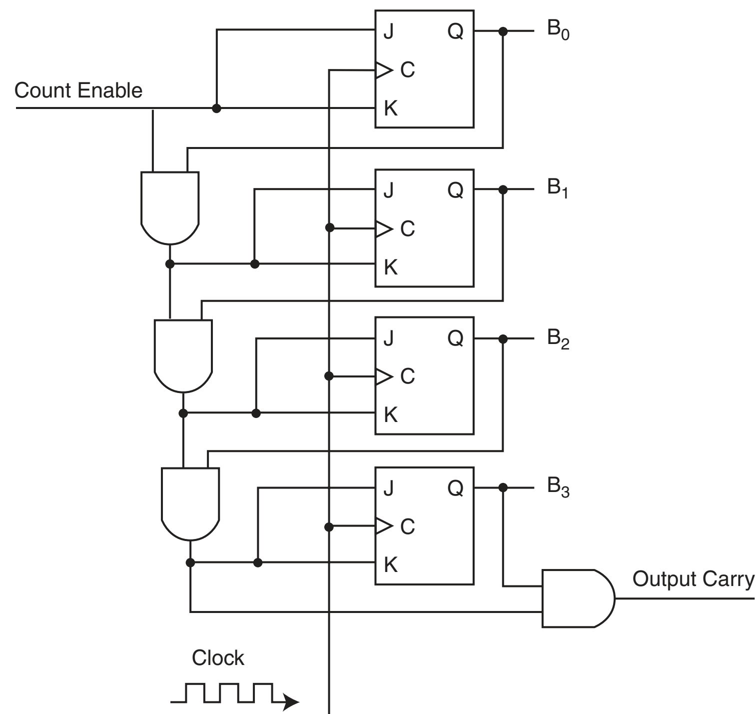


FIGURE 3.24 **A 4-Bit Synchronous Counter Using JK Flip-Flops**

memory is being referenced. (Notice that S_0 and S_1 are the input lines to a 2-to-4 decoder that is responsible for selecting the correct memory word.) The three output lines (Out_1,Out_2, and Out_3) are used when reading words from memory.

You should notice another control line as well. The *write enable* control line indicates whether we are reading or writing. Note that in this chip, we have separated the input and output lines for ease of understanding. In practice, the input lines and output lines are the same lines.

To summarize our discussion of this memory circuit, here are the steps necessary to write a word to memory:

1. An address is asserted on S_0 and S_1.
2. WE (write enable) is set to high.
3. The decoder using S_0 and S_1 enables only one AND gate, selecting a given word in memory.
4. The line selected in Step 3 combined with the clock and WE select only one word.
5. The write gate enabled in Step 4 drives the clock for the selected word.
6. When the clock pulses, the word on the input lines is loaded into the D flip-flops.

We leave it as an exercise to create a similar list of the steps necessary to read a word from this memory. Another interesting exercise is to analyze this circuit

Null03 1/7/03 4:35 PM Page 120

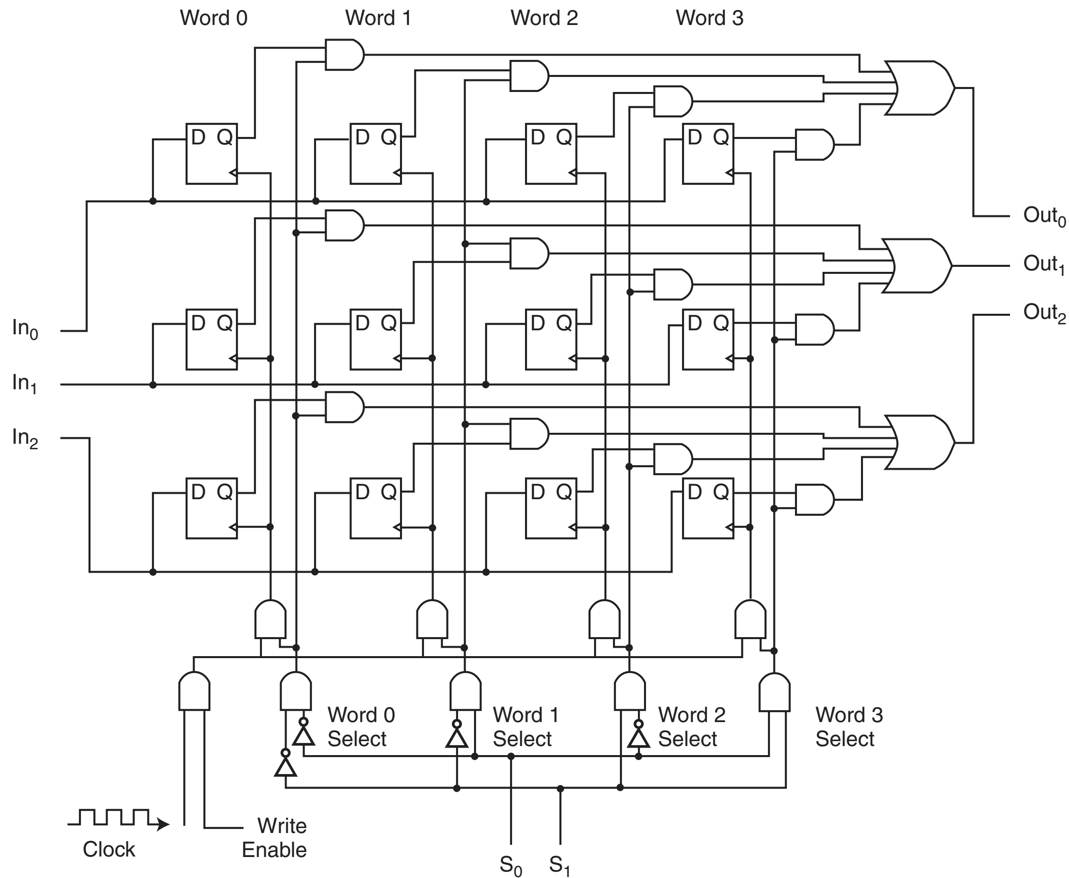


FIGURE 3.25 **A 4 × 3 Memory**

and determine what additional components would be necessary to extend the memory from, say, a 4 × 3 memory to an 8 × 3 memory or a 4 × 8 memory.

3.7 DESIGNING CIRCUITS

In the preceding sections, we introduced many different components used in computer systems. We have, by no means, provided enough detail to allow you to start designing circuits or systems. Digital logic design requires someone not only familiar with digital logic, but also well versed in *digital analysis* (analyzing the relationship between inputs and outputs), *digital synthesis* (starting with a truth table and determining the logic diagram to implement the given logic function), and the use of CAD (computer-aided design) software. Recall from our previous discussions that great care needs to be taken when designing the circuits to ensure that they are minimized. A circuit designer faces many problems, including find-

ing efficient Boolean functions, using the smallest number of gates, using an inexpensive combination of gates, organizing the gates of a circuit board to use the smallest surface area and minimal power requirements, and attempting to do all of this using a standard set of modules for implementation. Add to this the many problems we have not discussed, such as signal propagation, fan out, synchronization issues, and external interfacing, and you can see that digital circuit design is quite complicated.

Up to this point, we have discussed how to design registers, counters, memory, and various other digital building blocks. Given these components, a circuit designer can implement any given algorithm in hardware (recall the Principle of Equivalence of Hardware and Software from Chapter 1). When you write a program, you are specifying a sequence of Boolean expressions. Typically, it is much easier to write a program than it is to design the hardware necessary to implement the algorithm. However, there are situations in which the hardware implementation is better (for example, in a real-time system, the hardware implementation is faster, and faster is definitely better.) However, there are also cases in which a software implementation is better. It is often desirable to replace a large number of digital components with a single programmed microcomputer chip, resulting in an *embedded system*. Your microwave oven and your car most likely contain embedded systems. This is done to replace additional hardware that could present mechanical problems. Programming these embedded systems requires design software that can read input variables and send output signals to perform such tasks as turning a light on or off, emitting a beep, sounding an alarm, or opening a door. Writing this software requires an understanding of how Boolean functions behave.

CHAPTER SUMMARY

The main purpose of this chapter is to acquaint you with the basic concepts involved in logic design and to give you a general understanding of the basic circuit configurations used to construct computer systems. This level of familiarity will not enable you to design these components; rather, it gives you a much better understanding of the architectural concepts discussed in the following chapters.

In this chapter we examined the behaviors of the standard logical operators AND, OR, and NOT and looked at the logic gates that implement them. Any Boolean function can be represented as a truth table, which can then be transformed into a logic diagram, indicating the components necessary to implement the digital circuit for that function. Thus, truth tables provide us with a means to express the characteristics of Boolean functions as well as logic circuits. In practice, these simple logic circuits are combined to create components such as adders, ALUs, decoders, multiplexers, registers, and memory.

There is a one-to-one correspondence between a Boolean function and its digital representation. Boolean identities can be used to reduce Boolean expressions, and thus, to minimize both combinational and sequential circuits. Minimization is extremely important in circuit design. From a chip designer's point of

Null03 1/7/03 4:35 PM Page 122

view, the two most important factors are speed and cost: minimizing the circuits helps to both lower the cost and increase performance.

Digital logic is divided into two categories: combinational logic and sequential logic. Combinational logic devices, such as adders, decoders, and multiplexers, produce outputs that are based strictly on the current inputs. The AND, OR, and NOT gates are the building blocks for combinational logic circuits, although universal gates, such as NAND and NOR, could also be used. Sequential logic devices, such as registers, counters, and memory, produce outputs based on the combination of current inputs and the current state of the circuit. These circuits are built using SR, D, and JK flip-flops.

These logic circuits are the building blocks necessary for computer systems. In the next chapter we put these blocks together and take a closer, more detailed look at how a computer actually functions.

If you are interested in learning more about Kmaps, there is a special section that focuses on Kmaps located at the end of this chapter, after the exercises.

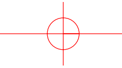
FURTHER READING

Most computer organization and architecture books have a brief discussion of digital logic and Boolean algebra. The books by Stallings (2000) and Patterson and Hennessy (1997) contain good synopses of digital logic. Mano (1993) presents a good discussion on using Kmaps for circuit simplification (discussed in the focus section of this chapter) and programmable logic devices, as well as an introduction to the various circuit technologies. For more in-depth information on digital logic, see the Wakerly (2000), Katz (1994), or Hayes (1993) books.

For a good discussion of Boolean algebra in lay terms, check out the book by Gregg (1998). The book by Maxfield (1995) is an absolute delight to read and contains informative and sophisticated concepts on Boolean logic, as well as a trove of interesting and enlightening bits of trivia (including a wonderful recipe for seafood gumbo!). For a very straightforward and easy book to read on gates and flip-flops (as well as a terrific explanation of what computers are and how they work), see the book by Petgold (1989). Davidson (1979) presents a method of decomposing NAND-based circuits (of interest because NAND is a universal gate).

If you are interested in actually designing some circuits, there is a nice simulator freely available. The set of tools is called the Chipmunk System. It performs a wide variety of applications, including electronic circuit simulation, graphics editing, and curve plotting. It contains four main tools, but for circuit simulation, *Log* is the program you need. The *Diglog* portion of Log allows you to create and actually test digital circuits. If you are interested in downloading the program and running it on your machine, the general Chipmunk distribution can be found at www.cs.berkeley.edu/~lazzaro/chipmunk/. The distribution is available for a wide variety of platforms (including PCs and Unix machines).

Null03 1/7/03 4:35 PM Page 123



REFERENCES

Davidson, E. S. "An Algorithm for NAND Decomposition under Network Constraints," *IEEE Transactions on Computing:* C-18, 1098, 1979.

Gregg, John. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets.* New York: IEEE Press, 1998.

Hayes, J. P. *Digital Logic Design.* Reading, MA: Addison-Wesley, 1993.

Katz, R. H. *Contemporary Logic Design.* Redwood City, CA: Benjamin Cummings, 1994.

Mano, Morris M. *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

Maxfield, Clive. *Bebop to the Boolean Boogie*. Solana Beach, CA: High Text Publications, 1995.

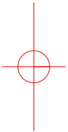
Patterson, D. A. and Hennessy, J. L. *Computer Organization and Design*, *The Hardware/Software Interface*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1997.

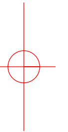
Petgold, Charles. *Code: The Hidden Language of Computer Hardware and Software*, Redmond, WA: Microsoft Press, 1989.

Stallings, W. *Computer Organization and Architecture*, 5th ed. New York: Macmillan Publishing Company, 2000.

Tanenbaum, Andrew. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

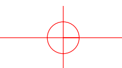
Wakerly, J. F. *Digital Design Principles and Practices*, Upper Saddle River, NJ: Prentice Hall, 2000.





REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. Why is an understanding of Boolean algebra important to computer scientists?
2. Which Boolean operation is referred to as a Boolean product?
3. Which Boolean operation is referred to as a Boolean sum?
4. Create truth tables for the Boolean operators OR, AND, and NOT.
5. What is the Boolean duality principle?
6. Why is it important for Boolean expressions to be minimized in the design of digital circuits?
7. What is the relationship between transistors and gates?
8. Name the four basic logic gates.
9. What are the two universal gates described in this chapter? Why are these universal gates important?
10. Describe the basic construction of a digital logic chip.
11. Describe the operation of a ripple-carry adder. Why are ripple-carry adders not used in most computers today?
12. What do we call a circuit that takes several inputs and their respective values to select one specific output line? Name one important application for these devices.
13. What kind of circuit selects binary information from one of many input lines and directs it to a single output line?



14. How are sequential circuits different from combinational circuits?
15. What is the basic element of a sequential circuit?
16. What do we mean when we say that a sequential circuit is edge-triggered rather than level-triggered?
17. What is feedback?
18. How is a JK flip-flop related to an SR flip-flop?
19. Why are JK flip-flops often preferred to SR flip-flops?
20. Which flip-flop gives a true representation of computer memory?

EXERCISES

◆ 1. Construct a truth table for the following:
   - ◆ a) $xyz + (\overline{xyz})$
   - ◆ b) $x(y\overline{z} + \overline{x}y)$

2. Construct a truth table for the following:
   - a) $xyz + x\overline{yz} + \overline{xyz}$
   - b) $(x + y)(x + z)(\overline{x} + z)$

◆ 3. Using DeMorgan's Law, write an expression for the complement of F if $F(x,y,z) = x(\overline{y} + z)$.

4. Using DeMorgan's Law, write an expression for the complement of F if $F(x,y,z) = xy + \overline{x}z + y\overline{z}$.

◆ 5. Using DeMorgan's Law, write an expression for the complement of F if $F(w,x,y,z) = xy\overline{z}\,(\overline{\overline{y}z + x}) + (\overline{w}yz + \overline{x})$.

6. Use the Boolean identities to prove the following:
   - a) The absorption laws
   - b) DeMorgan's laws

◆ 7. Is the following distributive law valid or invalid? Prove your answer.

   x XOR (y AND z) = (x XOR y) AND (x XOR z)

8. Show that $x = xy + x\overline{y}$
   - a) Using truth tables
   - b) Using Boolean identities

9. Show that $xz = (x + y)(x + \overline{y})(\overline{x} + z)$
   - a) Using truth tables
   - ◆ b) Using Boolean identities

Null03 1/7/03 4:35 PM Page 125

10. Simplify the following functional expressions using Boolean algebra and its identities. List the identity used at each step.

a) $F(x,y,z) = \bar{x}y + xy\bar{z} + xyz$

b) $F(w,x,y,z) = (x\bar{y} + \bar{w}z)(w\bar{x} + y\bar{z})$

c) $F(x,y,z) = (\overline{x + y})(\overline{\bar{x} + \bar{y}})$

11. Simplify the following functional expressions using Boolean algebra and its identities. List the identity used at each step.

◆ **a)** $\bar{x}yz + xz$

◆ **b)** $(\overline{x + y})(\overline{\bar{x} + \bar{y}})$

◆ **c)** $\overline{\bar{x}\bar{\bar{x}}y}$

12. Simplify the following functional expressions using Boolean algebra and its identities. List the identity used at each step.

a) $(ab + c + df)ef$

b) $x + xy$

c) $(x\bar{y} + \bar{x}z)(w\bar{x} + y\bar{z})$

13. Simplify the following functional expressions using Boolean algebra and its identities. List the identity used at each step.

◆ **a)** $xy + x\bar{y}$

b) $\bar{x}yz + xz$

c) $wx + w(xy + y\bar{z})$

14. Use any method to prove the following either true or false:

$yz + xy\bar{z} + \bar{x}\bar{y}z = xy + \bar{x}z$

◆ **15.** Using the basic identities of Boolean algebra, show that:

$x(\bar{x} + y) = xy$

***16.** Using the basic identities of Boolean algebra, show that:

$x + \bar{x}y = x + y$

◆ **17.** Using the basic identities of Boolean algebra, show that:

$xy + \bar{x}z + yz = xy + \bar{x}z$

◆ **18.** The truth table for a Boolean expression is shown below. Write the Boolean expression in sum-of-products form.

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Null03 1/7/03 4:35 PM Page 126

19. The truth table for a Boolean expression is shown below. Write the Boolean expression in sum-of-products form.

x	y	z	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

20. Draw the truth table and rewrite the expression below as the complemented sum of two products:

$x\bar{z} + \bar{y}z + \bar{x}y$

21. Given the Boolean function $F(x,y,z) = \bar{x}y + xy\bar{z}$

◆ **a)** Derive an algebraic expression for the complement of F. Express in sum-of-products form.

b) Show that $F\bar{F} = 0$.

c) Show that $F + \bar{F} = 1$.

22. Given the function $F(xy,z) = x\bar{y}z + \bar{x}\bar{y}z + xyz$

a) List the truth table for F.

b) Draw the logic diagram using the original Boolean expression.

c) Simplify the expression using Boolean algebra and identities.

d) List the truth table for your answer in Part c.

e) Draw the logic diagram for the simplified expression in Part c.

23. Construct the XOR operator using only AND, OR, and NOT gates.

***24.** Construct the XOR operator using only NAND gates.

Hint: x XOR $y = \overline{(\overline{\bar{x}y})(\overline{x\bar{y}})}$

25. Design a circuit with three inputs (x,y, and z) representing the bits in a binary number, and three outputs (a,b, and c) also representing bits in a binary number. When the input is 0, 1, 2, or 3, the binary output should be one less than the input. When the binary input is 4, 5, 6, or 7, the binary output should be one greater than the input. Show your truth table, all computations for simplification, and the final circuit.

26. Draw the combinational circuit that directly implements the following Boolean expression:

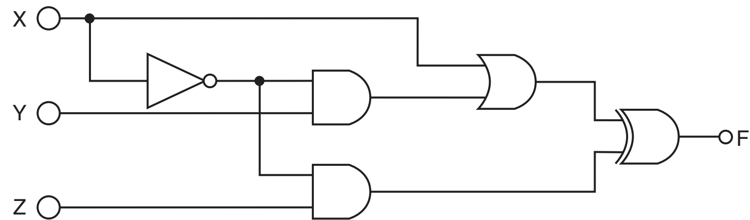
$F(x,y,z) = xz + (xy + \bar{z})$

◆ **27.** Draw the combinational circuit that directly implements the following Boolean expression:

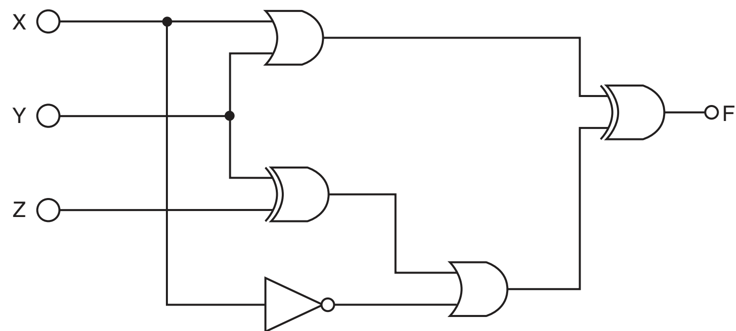
$F(x,y,z) = (xy \text{ XOR } (\overline{y + \bar{z}})) + \bar{x}z$

Null03 1/7/03 4:35 PM Page 127

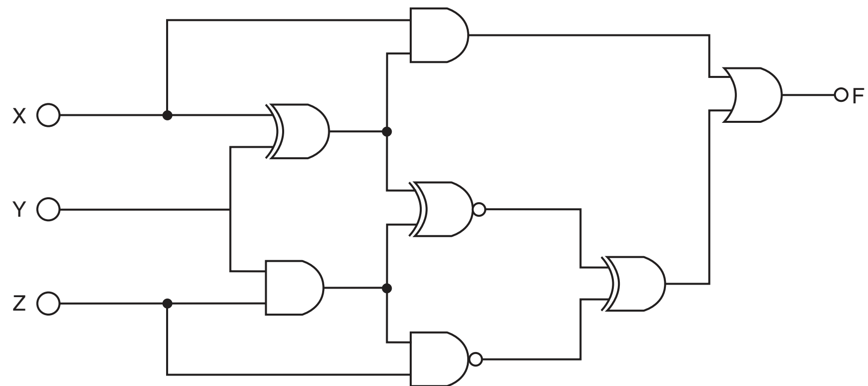
28. Find the truth table that describes the following circuit:



◆ **29.** Find the truth table that describes the following circuit:



30. Find the truth table that describes the following circuit:



31. Draw circuits to implement the parity generator and parity checker shown in Tables 3.11 and 3.12, respectively.

32. Draw a half-adder using only NAND gates.

33. Draw a full-adder using only NAND gates.

34. Tyrone Shoelaces has invested a huge amount of money into the stock market and doesn't trust just anyone to give him buying and selling information. Before he will buy a certain stock, he must get input from three sources. His first source is Pain

Null03 1/7/03 4:35 PM Page 128

Webster, a famous stock broker. His second source is Meg A. Cash, a self-made millionaire in the stock market, and his third source is Madame LaZora, a world-famous psychic. After several months of receiving advice from all three, he has come to the following conclusions:

a) Buy if Pain and Meg both say yes and the psychic says no.

b) Buy if the psychic says yes.

c) Don't buy otherwise.

Construct a truth table and find the minimized Boolean function to implement the logic telling Tyrone when to buy.

◆ ***35.** A very small company has hired you to install a security system. The brand of system that you install is priced by the number of bits encoded on the proximity cards that allow access to certain locations in a facility. Of course, this small company wants to use the fewest bits possible (spending the least amount of money as possible) yet have all of their security needs met. The first thing you need to do is determine how many bits each card requires. Next, you have to program card readers in each secured location so that they respond appropriately to a scanned card.

This company has four types of employees and five areas that they wish to restrict to certain employees. The employees and their restrictions are as follows:

a) The Big Boss needs access to the executive lounge and the executive washroom.

b) The Big Boss's secretary needs access to the supply closet, employee lounge, and executive lounge.

c) Computer room employees need access to the server room and the employee lounge.

d) The janitor needs access to all areas in the workplace.

Determine how each class of employee will be encoded on the cards and construct logic diagrams for the card readers in each of the five restricted areas.

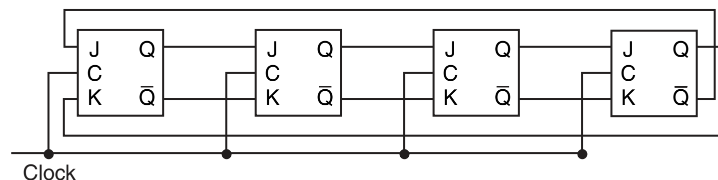
36. How many 256 × 8 RAM chips are needed to provide a memory capacity of 4096 bytes?

a) How many bits will each memory address contain?

b) How many address lines must go to each chip?

c) How many lines must be decoded for the chip select inputs? Specify the size of the decoder.

◆ ***37.** Investigate the operation of the following circuit. Assume an initial state of 0000. Trace the outputs (the Qs) as the clock ticks and determine the purpose of the circuit. You must show the trace to complete your answer.

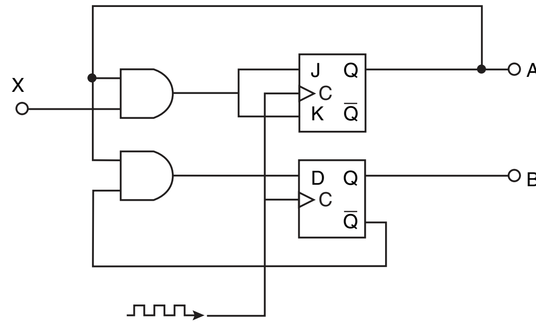


38. Describe how each of the following circuits works and indicate typical inputs and outputs. Also provide a carefully labeled *black box* diagram for each.

Null03 1/7/03 4:35 PM Page 129

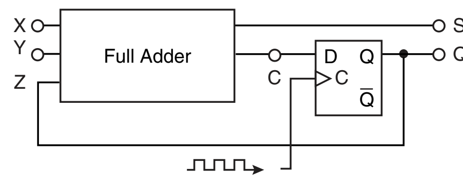
a) Decoder **b)** Multiplexer

◆ **39.** Complete the truth table for the following sequential circuit:



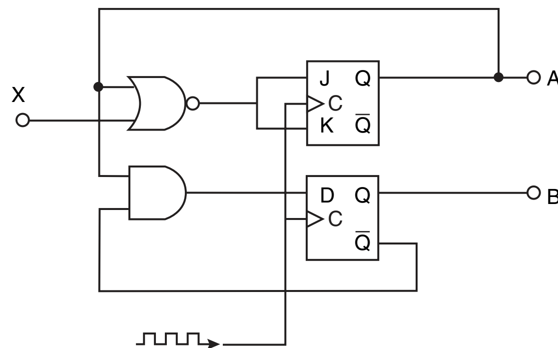
			Next State	
A	*B*	*X*	*A*	*B*
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

40. Complete the truth table for the following sequential circuit:



			Next State	
A	*B*	*X*	*A*	*B*
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

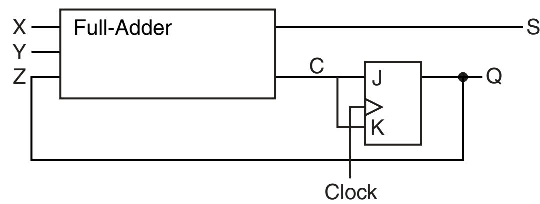
◆ **41.** Complete the truth table for the following sequential circuit:



			Next State	
A	*B*	*X*	*A*	*B*
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

42. A sequential circuit has one flip-flop; two inputs, X and Y; and one output, S. It consists of a full-adder circuit connected to a D flip-flop, as shown below. Fill in the characteristic table for this sequential circuit by completing the *Next State* and *Output* columns.

Null03 1/7/03 4:36 PM Page 130



Present State Q (t)	Inputs X	Y	Next State Q (t + 1)	Output S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

◆ ***43.** A Mux-Not flip-flop (MN flip-flop) behaves as follows: If M = 1, the flip-flop complements the current state. If M = 0, the next state of the flip-flop is equal to the value of N.

a) Derive the characteristic table for the flip-flop.

b) Show how a JK flip-flop can be converted to an MN flip-flop by adding gate(s) and inverter(s).

◆ **44.** List the steps necessary to read a word from memory in the 4 × 3 memory circuit shown in Figure 3.25.

FOCUS ON KARNAUGH MAPS

3A.1 INTRODUCTION

In this chapter, we focused on Boolean expressions and their relationship to digital circuits. Minimizing these circuits helps reduce the number of components in the actual physical implementation. Having fewer components allows the circuitry to operate faster.

Reducing Boolean expressions can be done using Boolean identities; however, using identities can be very difficult because no rules are given on how or when to use the identities, and there is no well-defined set of steps to follow. In one respect, minimizing Boolean expressions is very much like doing a proof: You know when you are on the right track, but getting there can sometimes be

Null03 1/7/03 4:36 PM Page 131

frustrating and time-consuming. In this appendix, we introduce a systematic approach for reducing Boolean expressions.

3A.2 DESCRIPTION OF KMAPS AND TERMINOLOGY

Karnaugh maps, or *Kmaps*, are a graphical way to represent Boolean functions. A map is simply a table used to enumerate the values of a given Boolean expression for different input values. The rows and columns correspond to the possible values of the function's inputs. Each cell represents the outputs of the function for those possible inputs.

If a product term includes all of the variables exactly once, either complemented or not complemented, this product term is called a *minterm*. For example, if there are two input values, x and y, there are four minterms, $\bar{x}\bar{y}$, $\bar{x}y$, $x\bar{y}$, and xy, which represent all of the possible input combinations for the function. If the input variables are x, y, and z, then there are eight minterms: $\bar{x}\bar{y}\bar{z}$, $\bar{x}\bar{y}z$, $\bar{x}y\bar{z}$, $\bar{x}yz$, $x\bar{y}\bar{z}$, $x\bar{y}z$, $xy\bar{z}$, and xyz.

As an example, consider the Boolean function $F(x,y) = xy + \bar{x}y$. Possible inputs for x and y are shown in Figure 3A.1.

The minterm $\bar{x}\bar{y}$ represents the input pair (0,0). Similarly, the minterm $\bar{x}y$ represents (0,1), the minterm $x\bar{y}$ represents (1,0), and xy represents (1,1).

The minterms for three variables, along with the input values they represent, are shown in Figure 3A.2.

Minterm	x	y
$\bar{X}\bar{Y}$	0	0
$\bar{X}Y$	0	1
$X\bar{Y}$	1	0
XY	1	1

FIGURE 3A.1 **Minterms for Two Variables**

Minterm	x	y	z
$\bar{X}\bar{Y}\bar{Z}$	0	0	0
$\bar{X}\bar{Y}Z$	0	0	1
$\bar{X}Y\bar{Z}$	0	1	0
$\bar{X}YZ$	0	1	1
$X\bar{Y}\bar{Z}$	1	0	0
$X\bar{Y}Z$	1	0	1
$XY\bar{Z}$	1	1	0
XYZ	1	1	1

FIGURE 3A.2 **Minterms for Three Variables**

Null03 1/7/03 4:36 PM Page 132

A Kmap is a table with a cell for each minterm, which means it has a cell for each line of the truth table for the function. Consider the function $F(x,y) = xy$ and its truth table, as seen in Example 3A.1.

EXAMPLE 3A.1 $F(x,y) = xy$

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

The corresponding Kmap is:

x \ y	0	1
0	0	0
1	0	1

Notice that the only cell in the map with a value of one occurs when $x = 1$ and $y = 1$, the same values for which $xy = 1$. Let's look at another example, $F(x,y) = x + y$.

EXAMPLE 3A.2 $F(x,y) = x + y$

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x \ y	0	1
0	0	1
1	1	1

Three of the minterms in Example 3A.2 have a value of 1, exactly the minterms for which the input to the function gives us a 1 for the output. To assign 1s in the Kmap, we simply place 1s where we find corresponding 1s in the truth table. We can express the function $F(x,y) = x + y$ as the logical OR of all minterms for which the minterm has a value of 1. Then $F(x,y)$ can be represented by the expression $\overline{x}y + x\overline{y} + xy$ Obviously, this expression is not minimized (we already know this function is simply $x + y$). We can minimize using Boolean identities:

Null03 1/7/03 4:36 PM Page 133

$$
\begin{aligned}
F(x,y) &= \overline{x}y + x\overline{y} + xy \\
&= \overline{x}y + xy + x\overline{y} + xy \qquad \text{(remember, } xy + xy = xy\text{)} \\
&= y(\overline{x} + x) + x(\overline{y} + y) \\
&= y + x \\
&= x + y
\end{aligned}
$$

How did we know to add in an extra xy term? Algebraic simplification using Boolean identities can be very tricky. This is where Kmaps can help.

3A.3 KMAP SIMPLIFICATION FOR TWO VARIABLES

In the previous reduction for the function $F(x,y)$, the goal was to group terms so we could factor out variables. We added the xy to give us a term to combine with the $\overline{x}y$. This allowed us to factor out the y, leaving $\overline{x} + x$, which reduces to 1. However, if we use Kmap simplification, we won't have to worry about which terms to add or which Boolean identity to use. The maps take care of that for us.

Let's look at the Kmap for $F(x,y) = x + y$ again in Figure 3A.3.

To use this map to reduce a Boolean function, we simply need to group ones. This grouping is very similar to how we grouped terms when we reduced using Boolean identities, except we must follow specific rules. First, we group only ones. Second, we can group ones in the Kmap if the ones are in the same row or in the same column, but they cannot be on the diagonal (i.e., they must be adjacent cells). Third, we can group ones if the total number in the group is a power of 2. The fourth rule specifies we must make the groups as large as possible. As a fifth and final rule, all ones must be in a group (even if some are in a group of one). Let's examine some correct and incorrect groupings, as shown in Figures 3A.4 through 3A.7.

Notice in Figure 3A.6(b) and 3A.7(b) that one 1 belongs to two groups. This is the map equivalent of adding the term xy to the Boolean function, as we did when we were performing simplification using identities. The xy term in the map will be used twice in the simplification procedure.

To simplify using Kmaps, first create the groups as specified by the rules above. After you have found all groups, examine each group and discard the variable that differs within each group. For example, Figure 3A.7(b) shows the correct grouping for $F(x,y) = x + y$. Let's begin with the group represented by the second row (where $x = 1$). The two minterms are $x\overline{y}$ and xy. This group represents the logical OR of these two terms, or $x\overline{y} + xy$. These terms differ in y, so y is discarded,

x \ y	0	1
0	0	1
1	1	1

FIGURE 3A.3 Kmap for $F(x,y) = x + y$

Null03 1/7/03 4:36 PM Page 134

x \ y	0	1
0	0	1
1	1	1

a) Incorrect

x \ y	0	1
0	0	1
1	1	1

b) Correct

FIGURE 3A.4 Groups Contain Only 1s

x \ y	0	1
0	0	1
1	1	1

a) Incorrect

x \ y	0	1
0	0	1
1	1	1

b) Correct

FIGURE 3A.5 Groups Cannot Be Diagonal

x \ y	0	1
0	0	1
1	1	1

a) Incorrect

x \ y	0	1
0	0	1
1	1	1

b) Correct

FIGURE 3A.6 Groups Must Be Powers of 2

x \ y	0	1
0	0	1
1	1	1

a) Incorrect

x \ y	0	1
0	0	1
1	1	1

b) Correct

FIGURE 3A.7 Groups Must Be as Large as Possible

leaving only x. (We can see that if we use Boolean identities, this would reduce to the same value. The Kmap allows us to take a shortcut, helping us to automatically discard the correct variable.) The second group represents $\overline{x}y + xy$. These differ in x, so x is discarded, leaving y. If we OR the results of the first group and the second group, we have $x + y$, which is the correct reduction of the original function, F.

3A.4 KMAP SIMPLIFICATION FOR THREE VARIABLES

Kmaps can be applied to expressions of more than two variables. In this focus section, we show three-variable and four-variable Kmaps. These can be extended for situations that have five or more variables. We refer you to Maxfield (1995) in the "Further Reading" section of this chapter for thorough and enjoyable coverage of Kmaps.

You already know how to set up Kmaps for expressions involving two variables. We simply extend this idea to three variables, as indicated by Figure 3A.8.

The first difference you should notice is that two variables, y and z, are grouped together in the table. The second difference is that the numbering for the columns is not sequential. Instead of labeling the columns as 00, 01, 10, 11 (a normal binary progression), we have labeled them 00, 01, 11, 10. The input values for the Kmap must be ordered so that each minterm differs in only one variable from each neighbor. By using this order (for example 01 followed by 11), the

x \ yz	00	01	11	10
0	$\overline{X}\overline{Y}\overline{Z}$	$\overline{X}\overline{Y}Z$	$\overline{X}YZ$	$\overline{X}Y\overline{Z}$
1	$X\overline{Y}\overline{Z}$	$X\overline{Y}Z$	XYZ	$XY\overline{Z}$

FIGURE 3A.8 Minterms and Kmap Format for Three Variables

Null03 1/7/03 4:36 PM Page 135

corresponding minterms, $\bar{x}\bar{y}z$ and $\bar{x}yz$, differ only in the y variable. Remember, to reduce, we need to discard the variable that is different. Therefore, we must ensure that each group of two minterms differs in only one variable.

The largest groups we found in our two-variable examples were composed of two 1s. It is possible to have groups of four or even eight 1s, depending on the function. Let's look at a couple of examples of map simplification for expressions of three variables.

EXAMPLE 3A.3 $F(x, y, z) = \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z + xyz$

x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

We again follow the rules for making groups. You should see that you can make groups of two in several ways. However, the rules stipulate we must create the largest groups whose sizes are powers of two. There is one group of four, so we group these as follows:

x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

It is not necessary to create additional groups of two. The fewer groups you have, the fewer terms there will be. Remember, we want to simplify the expression, and all we have to do is guarantee that every 1 is in some group.

How, exactly, do we simplify when we have a group of four 1s? Two 1s in a group allowed us to discard one variable. Four 1s in a group allows us to discard two variables: The two variables in which all four terms differ. In the group of four from the preceding example, we have the following minterms: $\bar{x}\bar{y}z$, $\bar{x}yz$, $x\bar{y}z$ and xyz. These all have z in common, but the x and y variables differ. So we discard x and y, leaving us with $F(x,y,z) = z$ as the final reduction. To see how this parallels simplification using Boolean identities, consider the same reduction using identities. Note that the function is represented originally as the logical OR of the minterms with a value of 1.

$$\begin{aligned}
F(x. y, z) &= \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z + xyz \\
&= \bar{x}(\bar{y}z + yz) + x(\bar{y}z + yz) \\
&= (\bar{x} + x)(\bar{y}z + yz) \\
&= \bar{y}z + yz \\
&= (\bar{y} + y)z \\
&= z
\end{aligned}$$

Null03 1/7/03 4:36 PM Page 136

The end result using Boolean identities is exactly the same as the result using map simplification.

From time to time, the grouping process can be a little tricky. Let's look at an example that requires more scrutiny.

EXAMPLE 3A.4 $F(x, y, z) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}yz + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$

x \ yz	00	01	11	10
0	1	1	1	1
1	1	0	0	1

This is a tricky problem for two reasons: We have overlapping groups, and we have a group that "wraps around." The leftmost 1s in the first column can be grouped with the rightmost 1s in the last column, because the first and last columns are logically adjacent (envision the map as being drawn on a cylinder). The first and last rows of a Kmap are also logically adjacent, which becomes apparent when we look at four-variable maps in the next section.

The correct groupings are as follows:

x \ yz	00	01	11	10
0	1	1	1	1
1	1	0	0	1

The first group reduces to $\bar{x}$ (this is the only term the four have in common), and the second group reduces to $\bar{z}$, so the final minimized function is $F(x,y,z) = \bar{x} + \bar{z}$.

EXAMPLE 3A.5 A Kmap with all 1s

Suppose we have the following Kmap:

x \ yz	00	01	11	10
0	1	1	1	1
1	1	1	1	1

The largest group of 1s we can find is a group of eight, which puts all of the 1s in the same group. How do we simplify this? We follow the same rules we have been following. Remember, groups of two allowed us to discard one variable, and groups of four allowed us to discard two variables; therefore, groups of eight should allow us to discard three variables. But that's all we have! If we discard all the variables, we are left with $F(x,y,z) = 1$. If you examine the truth table for this function, you will see that we do indeed have a correct simplification.

Null03 1/7/03 4:36 PM Page 137

wx \ yz	00	01	11	10
00	$\overline{W}\overline{X}\overline{Y}\overline{Z}$	$\overline{W}\overline{X}\overline{Y}Z$	$\overline{W}\overline{X}YZ$	$\overline{W}\overline{X}Y\overline{Z}$
01	$\overline{W}X\overline{Y}\overline{Z}$	$\overline{W}X\overline{Y}Z$	$\overline{W}XYZ$	$\overline{W}XY\overline{Z}$
11	$WX\overline{Y}\overline{Z}$	$WX\overline{Y}Z$	$WXYZ$	$WXY\overline{Z}$
10	$W\overline{X}\overline{Y}\overline{Z}$	$W\overline{X}\overline{Y}Z$	$W\overline{X}YZ$	$W\overline{X}Y\overline{Z}$

FIGURE 3A.8 Minterms and Kmap Format for Four Variables

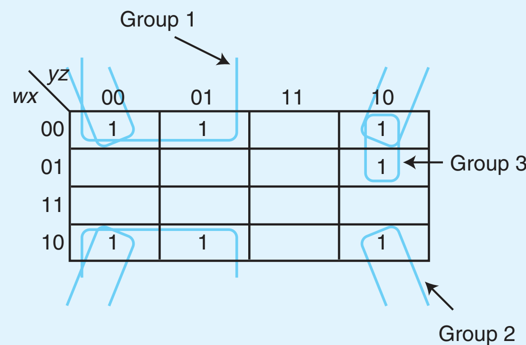
3A.5 KMAP SIMPLIFICATION FOR FOUR VARIABLES

We now extend the map simplification techniques to four variables. Four variables give us 16 minterms, as shown in Figure 3A.9. Notice the special order of 11 followed by 10 applies for the rows as well as the columns.

Example 3A.6 illustrates the representation and simplification of a function with four variables. We are only concerned with the terms that are 1s, so we omit entering the 0s into the map.

EXAMPLE 3A.6

$$F(w, x, y, z) = \overline{w}\,\overline{x}\,\overline{y}\,\overline{z} + \overline{w}\,\overline{x}\,\overline{y}z + \overline{w}\,\overline{x}y\overline{z} + \overline{w}xy\overline{z} + w\overline{x}\,\overline{y}\,\overline{z} + w\overline{x}\,\overline{y}x + w\overline{x}y\overline{z}$$



Group 1 is a "wrap-around" group, as we saw previously. Group 3 is easy to find as well. Group 2 represents the ultimate wrap-around group: It consists of the 1s in the four corners. Remember, these corners are logically adjacent. The final result is that F reduces to three terms, one from each group: $\overline{x}\,\overline{y}$ (from Group 1), $\overline{x}\,\overline{z}$ (from Group 2), and $\overline{w}y\overline{z}$ (from Group 3). The final reduction for F is then $F(w, x, y, z) = \overline{x}\,\overline{y} + \overline{x}\,\overline{z} + \overline{w}y\overline{z}$.

Occasionally, there are choices to make when performing map simplification. Consider Example 3A.7.

Null03 1/7/03 4:36 PM Page 138

EXAMPLE 3A.7 A Choice of Groups

wx \ yz	00	01	11	10
00	1		1	
01	1		1	1
11	1			
10	1			

The first column should clearly be grouped. Also, the $\overline{w}\,\overline{x}yz$ and $\overline{w}xyz$ terms should be grouped. However, we have a choice as to how to group the $\overline{w}xy\overline{z}$ term. It could be grouped with $\overline{w}xyz$ or with $\overline{w}x\overline{y}\,\overline{z}$ (as a wrap-around). These two solutions are indicated below.

wx \ yz	00	01	11	10
00	1		1	
01	1		1	1
11	1			
10	1			

wx \ yz	00	01	11	10
00	1		1	
01	1		1	1
11	1			
10	1			

The first map simplifies to $F(w, x, y, z) = F_1 = \overline{y}\,\overline{z} + \overline{w}yz + \overline{w}xy$. The second map simplifies to $F(w, x, y, z) = F_2 = \overline{y}\,\overline{z} + \overline{w}yz + \overline{w}x\overline{z}$. The last terms are different. F_1 and F_2, however, are equivalent. We leave it up to you to produce the truth tables for F_1 and F_2 to check for equality. They both have the same number of terms and variables as well. If we follow the rules, Kmap minimization results in a minimized function (and thus a minimal circuit), but these minimized functions need not be unique in representation.

Before we move on to the next section, here are the rules for Kmap simplification.

1. The groups can only contain 1s; no 0s.

2. Only 1s in adjacent cells can be grouped; diagonal grouping is not allowed.

3. The number of 1s in a group must be a power of 2.

4. The groups must be as large as possible while still following all rules.

5. All 1s must belong to a group, even if it is a group of one.

6. Overlapping groups are allowed.

7. Wrap around is allowed.

8. Use the fewest number of groups possible.

Null03 1/7/03 4:36 PM Page 139

Using these rules, let's complete one more example for a four-variable function. Example 3A.8 shows several applications of the various rules.

EXAMPLE 3A.8

$$F(w, x, y, z) = \overline{w}\,\overline{x}\,\overline{y}\,\overline{z} + \overline{w}\,\overline{x}yz + \overline{w}x\overline{y}z + \overline{w}xyz + wx\overline{y}z + wxyz + w\overline{x}yz + w\overline{x}y\overline{z}$$

wx \ yz	00	01	11	10
00	1		1	
01		1	1	
11		1	1	
10			1	1

In this example, we have one group with a single element. Note there is no way to group this term with any others if we follow the rules. The function represented by this Kmap simplifies to $F(w, x, y, z) = yz + xz + \overline{w}\,\overline{x}\,\overline{y}\,\overline{z}$.

If you are given a function that is not written as a sum of minterms, you can still use Kmaps to help minimize the function. However, you have to use a procedure that is somewhat the reverse of what we have been doing to set up the Kmap before reduction can occur. Example 3A.9 illustrates this procedure.

EXAMPLE 3A.9 A Function Not Represented as a Sum of Minterms

Suppose you are given the function $F(w, x, y, z) = \overline{w}xy + \overline{w}\,\overline{x}yz + \overline{w}\,\overline{x}y\overline{z}$. The last two terms are minterms, and we can easily place 1s in the appropriate positions in the Kmap. However the term $\overline{w}xy$ is not a minterm. Suppose this term were the *result* of a grouping you had performed on a Kmap. The term that was discarded was the z term, which means this term is equivalent to the two terms $\overline{w}xy\overline{z}$ + $\overline{w}xyz$. You can now use these two terms in the Kmap, because they are both minterms. We now get the following Kmap:

wx \ yz	00	01	11	10
00			1	1
01			1	1
11				
10				

Nul103 1/7/03 4:36 PM Page 140

So we know the function $F(w, x, y, z) = \overline{w}xy + \overline{w}\overline{x}yz + \overline{w}\overline{x}y\overline{z}$ simplifies to $F(w, x, y, z) = \overline{w}y$.

3A.6 DON'T CARE CONDITIONS

There are certain situations where a function may not be completely specified, meaning there may be some inputs that are undefined for the function. For example, consider a function with 4 inputs that act as bits to count, in binary, from 0 to 10 (decimal). We use the bit combinations 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, and 1010. However, we do not use the combinations 1011, 1100, 1101, 1110, and 1111. These latter inputs would be invalid, which means if we look at the truth table, these values wouldn't be either 0 or 1. They should not be in the truth table at all.

We can use these *don't care* inputs to our advantage when simplifying Kmaps. Because they are input values that should not matter (and should never occur), we can let them have values of either 0 or 1, depending on which helps us the most. The basic idea is to set these don't care values in such a way that they either contribute to make a larger group, or they don't contribute at all. Example 3A.10 illustrates this concept.

EXAMPLE 3A.10 Don't Care Conditions

Don't care values are typically indicated with an "X" in the appropriate cell. The following Kmap shows how to use these values to help with minimization. We treat the don't care values in the first row as 1s to help form a group of four. The don't care values in rows 01 and 11 are treated as 0s. This reduces to $F_1(w, x, y, z) = \overline{w}\overline{x} + yz$.

wx \ yz	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

There is another way these values can be grouped:

wx \ yz	00	01	11	10
00	X	1	1	X
01		X	1	
11	X		1	
10			1	

Nul103 1/7/03 4:36 PM Page 141

Using the above groupings, we end up with a simplification of $F_2(w, x, y, z) = \overline{w}z + yz$. Notice that in this case, F_1 and F_2 are not equal. However, if you create the truth tables for both functions, you should see that they are not equal only in those values for which we "don't care."

3A.7 SUMMARY

In this section we have given a brief introduction to Kmaps and map simplification. Using Boolean identities for reduction is awkward and can be very difficult. Kmaps, on the other hand, provide a precise set of steps to follow to find the minimal representation of a function, and thus the minimal circuit that function represents.

EXERCISES

1. Write a simplified expression for the Boolean function defined by each of the following Kmaps:

◆ a)

x \ yz	00	01	11	10
0	0	1	1	0
1	1	0	0	1

◆ b)

x \ yz	00	01	11	10
0	0	1	1	1
1	1	0	0	0

c)

x \ yz	00	01	11	10
0	1	1	1	0
1	1	1	1	1

2. Create the Kmaps and then simplify for the following functions:

a) $F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}yz + \overline{x}y\overline{z}$

b) $F(x, y, z) = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}y\overline{z} + x\overline{y}\,\overline{z} + xy\overline{z}$

c) $F(x, y, z) = \overline{y}\,\overline{z} + \overline{y}z + xy\overline{z}$

Null03 1/7/03 4:36 PM Page 142

3. Write a simplified expression for the Boolean function defined by each of the following Kmaps:

a)

wx \ yz	00	01	10	11
00	1			1
01	1			1
11			1	
10	1		1	

b)

wx \ yz	00	01	11	10
00	1	1	1	1
01			1	1
11	1	1	1	1
10	1			1

c)

wx \ yz	00	01	11	10
00		1		1
01		1	1	1
11	1	1		
10	1	1		1

4. Create the Kmaps and then simplify for the following functions:

a) $F(w, x, y, z) = \overline{w}\,\overline{x}\,\overline{y}\,\overline{z} + \overline{w}\,\overline{x}\,y\,\overline{z} + \overline{w}\,x\,\overline{y}\,z + \overline{w}\,x\,y\,z + \overline{w}\,x\,y\,\overline{z} +$
$w\,\overline{x}\,\overline{y}\,\overline{z} + w\,\overline{x}\,y\,\overline{z}$

b) $F(w. x. y, z) = \overline{w}\,\overline{x}\,\overline{y}\,\overline{z} + \overline{w}\,\overline{x}\,\overline{y}\,z + w\,\overline{x}\,\overline{y}\,z + w\,\overline{x}\,y\,\overline{z} + w\,\overline{x}\,\overline{y}\,\overline{z}$

c) $F(w. x. y, z) = \overline{y}\,z + w\,\overline{y} + \overline{w}\,x\,y + \overline{w}\,\overline{x}\,y\,\overline{z} + w\,\overline{x}\,y\,\overline{z}$

◆ **5.** Given the following Kmap, show algebraically (using Boolean identities) how the four terms reduce to one term.

x \ yz	00	01	11	10
0	0	1	1	0
1	0	1	1	0

Null03 1/7/03 4:36 PM Page 143

6. Write a simplified expression for the Boolean function defined by each of the following Kmaps:

◆ **a)**

x \ yz	00	01	11	10
0	1	1	0	X
1	1	1	1	1

b)

wx \ yx	00	01	11	10
00	1	1	1	1
01		X	1	X
11			X	
10	1		X	1