CHAPTER 3

Boolean Algebra and Digital Logic

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

—Claude Shannon

3.1 INTRODUCTION

George Boole lived in England during the first half of the nineteenth century. The firstborn son of a cobbler, Boole taught himself Greek, Latin, French, German, and the language of mathematics. Just before he turned 16, Boole accepted a position teaching at a small Methodist school, providing his family with much-needed income. At the age of 19, Boole returned home to Lincoln, England, and founded his own boarding school to better provide support for his family. He operated this school for 15 years, until he became professor of mathematics at Queen's College in Cork, Ireland. His social status as the son of a tradesman prevented Boole's appointment to a more prestigious university, despite his authoring of more than a dozen highly esteemed papers and treatises. His most famous monograph, The Laws of Thought, published in 1854, created a branch of mathematics known as symbolic logic or Boolean algebra.

Nearly 85 years later, John Vincent Atanasoff applied Boolean algebra to computing. He recounted the moment of his insight to Linda Null. At the time, Atanasoff was attempting to build a calculating machine based on the same technology used by Pascal and Babbage. His aim was to use this machine to solve systems of linear equations. After struggling with repeated failures, Atanasoff was so frustrated that 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 because he was in Illinois, where it was legal to buy a drink in a

tavern, he sat down and ordered a bourbon. He chuckled to himself when he realized that he had driven such a distance for a drink! Even more ironic is the fact that he never touched the drink. He felt he needed a clear head to write down the revelations that came to him during his long, aimless journey. Exercising his physics and mathematics backgrounds and focusing on the failures of his previous computing machine, he made four critical breakthroughs necessary for the machine's new design:

1. He would use electricity instead of mechanical movements (vacuum tubes would allow him to do this).
2. 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.
3. He would use capacitors (condensers) for memory because they store electrical charges with a regenerative process to avoid power leakage.
4. 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 your previous programming experience. 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 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. If you are interested in delving deeper, 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 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 the input values, in the set {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 behaviors. 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.

TABLE 3.1 Truth Table for AND

Inputs	Outputs

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

TABLE 3.2 Truth Table for OR

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

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

TABLE 3.3 Truth Table for NOT

Inputs	Outputs
x	x’
0	1
1	0

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 + 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 y' and z, and finally 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.

TABLE 3.4 Truth Table for F(x,y,z) = x + y'z

Inputs					Outputs
x	y	z	y'	y'z	X + 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

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.

TABLE 3.5 Basic Identities of Boolean Algebra

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	xx’ = 0	x + 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	(xy)’ = x’ + y’	(x + y)’ = x’y’
Double Complement Law	x” = x	

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 produces the

identity for that given operation. Boolean variables can be reordered (commuted) and regrouped (associated) without affecting the final result. You should recognize these as the Commutative and Associative Laws from algebra. 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- and right-hand sides of DeMorgan's Law for AND. It is left as exercises to prove the validity of the remaining laws, in particular, the OR form of DeMorgan's Law and both forms of the Absorption Law.

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

x	y	(xy)	(xy)′	x′	y′	x′ + 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

The Double Complement Law formalizes the idea of the double negative, which evokes rebuke from high school English teachers. The Double Complement Law can be useful in digital circuits as well as in your life. For example, let x? = 1 represent the idea that you have a positive quantity of cash. If you have no cash, you have x ′. When an untrustworthy acquaintance asks to borrow some cash, you can truthfully say that you "don't have no money." That is, x = (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: (xy)′ = x′y′ . Please note that this is not a valid equality! DeMorgan's Law clearly indicates that this statement is incorrect. Instead, (xy)′ = x′ + y′. This is a very easy mistake to make, and one that should be avoided. Care must be taken with other expressions involving negation as well.

NULL POINTERS: TIPS AND HINTS

DeMorgan's Law is very useful from a programming perspective. It gives us a convenient way to convert a positive statement into a negative one and vice versa. This is very common when dealing with conditionals and loops in programs. For example, with loops, sometimes we need an ending condition, and sometimes we need a condition that specifies when the loop should continue to iterate. For example, suppose you have a loop that you want to terminate when either a counter COUNT is greater than 10 or when a sum SUM is greater than 100. (It is often much easier to identify when a loop should terminate versus when it should continue.) This condition can be represented as follows:

- terminate the loop if (COUNT > 10 or SUM > 100)

Suppose you only have a WHILE loop available and must specify the condition that causes the loop to continue iterating. You need the negative of the condition specified above. DeMorgan's Law explains exactly how to do this: We need to negate an OR. According to this law, the negative of (X OR Y) is (NOT X and NOT Y). Therefore, to continue in the loop, COUNT must be less than or equal to 10 (not greater than 10), and SUM must be less than or equal to 100 (not greater than 100). The condition for a WHILE loop becomes:

- WHILE (COUNT < = 10 AND SUM < = 100) DO. . .

DeMorgan's Law can also help you get rid of unwanted NOTs in your program. For example, suppose you have the following IF statement:

- IF (NOT (X > 2 AND Y <= 12) THEN. . .

You can apply DeMorgan's Law by negating the NOT (to remove it), resulting in:

- IF (X <= 2 OR Y > 12) THEN. . .

where NOT (X > 2) becomes X <= 2 and NOT(Y <= 12) becomes Y > 12. You may have been using DeMorgan's

Law in programs that you have already written, but now you should understand the logic behind your code.

Instead, (xy)′ = x′ + y′. This is a very easy mistake to make, and one that should be avoided. Care must be taken with other expressions involving negation as well.

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 manner. 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) = x′yz + x′yz′ + xz, we simplify as follows:

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

EXAMPLE 3.3

Given the function F (x, y) = y + (xy)′, we simplify as follows:

$$
\begin{aligned}
F(x, y) &= y + (xy)' \\
&= y + (x' + y') && \text{(DeMorgan's)} \\
&= y + (y' + x') && \text{(Commutative)} \\
&= (y + y') + x' && \text{(Associative)} \\
&= 1 + x' && \text{(Inverse)} \\
&= 1 && \text{(Null)}
\end{aligned}
$$

EXAMPLE 3.4

Given the function F (x, y) = (xy)′(x′ + y)(y′ + y), we simplify as follows:

$$
\begin{aligned}
F(x,y) &= (xy)'(x' + y)(y' + y) \\
&= (xy)'(x' + y)(1) && \text{(Inverse)} \\
&= (xy)'(x' + y) && \text{(Identity)} \\
&= (x' + y')(x' + y) && \text{(DeMorgan's)} \\
&= x' + y'y && \text{(Distributive over AND)} \\
&= x' + 0 && \text{(Inverse)} \\
&= x' && \text{(Idempotent)}
\end{aligned}
$$

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

EXAMPLE 3.5

Given the function F(x, y) = x′(x + y) + (y + x)(x + y′), we simplify as follows:

$$
\begin{aligned}
F(x,y) &= x'(x + y) + (y + x)(x + y') \\
&= x'(x + y) + (x + y)(x + y') && \text{(Commutative)} \\
&= x'(x + y) + (x + yy') && \text{(Distributive over AND)} \\
&= x'(x + y) + (x + 0) && \text{(Inverse)} \\
&= x'(x + y) + x && \text{(Identity)} \\
&= x'x + x'y + x && \text{(Distributive)} \\
&= 0 + x'y + x && \text{(Inverse)} \\
&= x'y + x && \text{(Identity)} \\
&= x + x'y && \text{(Commutative)} \\
&= (x + x')(x + y) && \text{(Distributive over AND)} \\
&= 1(x + y) && \text{(Inverse)} \\
&= x + y && \text{(Identity)}
\end{aligned}
$$

EXAMPLE 3.6

Given the function F(x, y, z) = xy + x′z + yz, we simplify

as follows:

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

Example 3.6 illustrates what is commonly known as the Consensus Theorem.

How did we know to insert additional terms to simplify the function in Example 3.6? 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.

We can also use these identities to prove Boolean equalities, as we see in Example 3.7.

EXAMPLE 3.7

Prove that (x + y)(x′ + y) = y.

$$
\begin{aligned}
(x + y)(x' + y) &= xx' + xy + yx' + yy && \text{(Distributive)} \\
&= 0 + xy + yx' + yy && \text{(Inverse)} \\
&= 0 + xy + yx' + y && \text{(Idempotent)} \\
&= xy + yx' + y && \text{(Identity)} \\
&= yx + yx' + y && \text{(Commutative)} \\
&= y(x + x') + y && \text{(Distributive)} \\
&= y(1) + y && \text{(Inverse)} \\
&= y + y && \text{(Identity)} \\
&= y && \text{(Idempotent)}
\end{aligned}
$$

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 proven in Example 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. 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 $(x + y)' = x'y'$. We can easily extend this to three or more variables as follows.

Given the function:

- $F(x, y, z) = (x + y + z)$. Then $F'(x, y, z) = (x + y + z)'$.

Let $w = (x + y)$. Then $F'(x, y, z) = (w + z)' = w'z'$.

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

- $w' z' = (x + y)' z' = x'y'z' = F'(x, y, z)$

Therefore, if $F(x,y,z) = (x + y + z)$, then $F'(x, y, z) = x'y'z'$. Applying the principle of duality, we see that $(xyz)' = x' + y' + z'$.

It appears that, to find the complement of a Boolean expression, we simply replace each variable by its complement (x is replaced by x') and interchange ANDs and ORs. In fact, this is exactly what DeMorgan's Law tells us to do. For example, the complement of $x' + yz'$ is $x(y' + z)$. 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 3.7

depicts the truth tables for F(x,y,z) = x′ + yz′ and its complement, F′(x, y, z) = x(y′ + z). The shaded portions indicate the final results for F and F′.

TABLE 3.7 Truth Table Representation for a Function and Its Complement

x	y	z	yz′	x′ + yz′	y′ + z	x(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

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 3.8).

EXAMPLE 3.8

Suppose F(x, y, z) = x + xy′. We can also express F as F(x,y,z) = x + x + xy′ because the Idempotent Law tells us these two expressions are the same. We can also express F as F(x, y, z) = x(1 + 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 prod-uct-of-sums form.

The sum-of-products form requires that the expression

be a collection of ANDed variables (or product terms) that are ORed together. The function $F_1(x, y, z) = xy + yz' + xyz$ is in sum-of-products form. The function $F_2(x, y, z) = xy' + x(y + z')$ is not in sum-of-products form. We apply the Distributive Law to distribute the x variable in F_2, resulting in the expression $xy' + xy + xz'$, 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 $F_1(x, y, z) = (x + y)(x + z')(y + z')(y + z)$ 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; we therefore 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. Example 3.9 shows us that it is a simple matter to convert a truth table into sum-of-products form.

EXAMPLE 3.9

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 3.8 depicts the truth table for this majority function over three variables.

TABLE 3.8 Truth Table Representation for the Majority Function

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

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 our majority function, we see that the function must output a 1 when x = 0, y = 1, and z = 1. The product term that satisfies this is 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 xy′z. The third product term we need is xyz′, and the last is xyz. In summary, to generate a sum-of-products expression using the truth table for any Boolean expression, we 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, we 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) = x′yz + xy′z + xyz′ + xyz.

Please note that the expression for the majority function in Example 3.9 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 shown in the next section. Unnecessary 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.

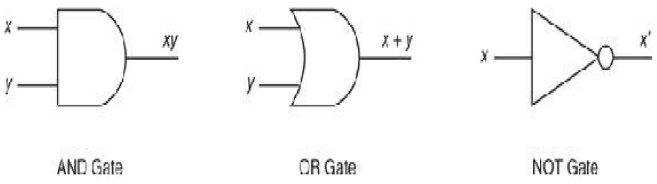
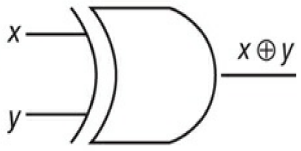


FIGURE 3.1 The Three Basic Gates

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.

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



(a) (b)

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

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

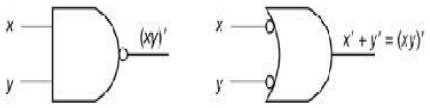


FIGURE 3.3 Truth Table and Logic Symbols for NAND

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

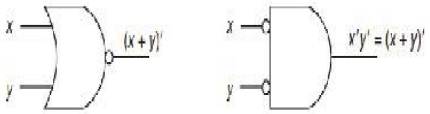


FIGURE 3.4 Truth Table and Logic Symbols for NOR

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.

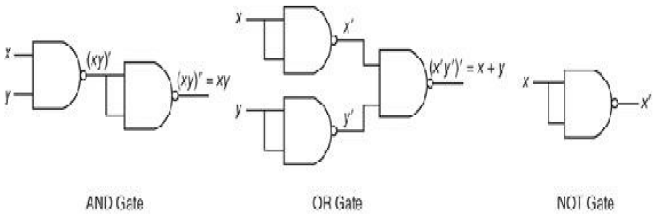


FIGURE 3.5 Three Circuits Constructed Using Only NAND Gates

Why not simply use the AND, OR, and NOT gates we already know exist? There are two reasons for 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. 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 xy′z.

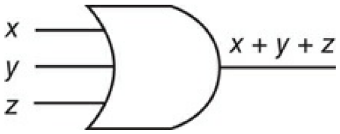


FIGURE 3.6 A Three-Input OR Gate Representing x + y + z

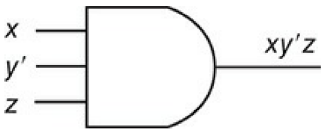


FIGURE 3.7 A Three-Input AND Gate Representing xy′z

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 Q, as shown in Figure 3.8.

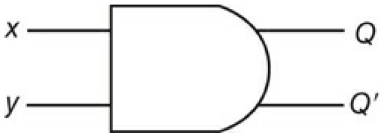


FIGURE 3.8 AND Gate with Two Inputs and Two Outputs

Note that Q always represents the actual output.

3.4 KARNAUGH MAPS

3.4.1 Introduction

In this chapter, so far we have 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 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 frustrating and time consuming. In this section, we introduce a systematic approach for reducing Boolean expressions.

3.4.2 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, x ′y′, x′y, xy′, 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: x ′y′z′ , x′y′z, x′yz′, x′yz, xy′z′ , xy′z, xyz′, and xyz.

As an example, consider the Boolean function F(x, y) = xy + x′y. Possible inputs for x and y are shown in Figure 3.9. The minterm x′ y represents the input pair (0, 0). Similarly, the minterm x′y represents (0, 1), the minterm

xy′ represents (1, 0), and xy represents (1, 1).

Minterm	x	y
x′y′	0	0
x′y	0	1
xy′	1	0
xy	1	1

FIGURE 3.9 Minterms for Two Variables

The minterms for three variables, along with the input values they represent, are shown in Figure 3.10.

Minterm	x	y	z
x′y′z′	0	0	0
x′y′z	0	0	1
x′yz′	0	1	0
x′yz	0	1	1
xy′z′	1	0	0
xy′z	1	0	1
xyz′	1	1	0
xyz	1	1	1

FIGURE 3.10 Minterms for Three Variables

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 3.10.

EXAMPLE 3.10

F(x, y) = xy

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

The corresponding Kmap is

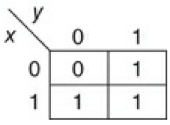
x \ y	0	1
0	0	0
1	1	1

Notice that the only cell in the map with a value of 1 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 3.11

F(x, y) = x + y

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



Three of the minterms in Example 3.11 have a value of 1, exactly the min-terms 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 min-terms for which the minterm has a value of 1. Then F(x,y) can be represented by the expression x′y + xy′ + xy. Obviously, this expression is not minimized (we already know this function is simply x + y). We can minimize using Boolean identities.

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

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

3.4.3 Kmap Simplification for Two Variables

In the previous reduction for the function F(x,y), the goal was to group the terms so we could factor out variables. We added the xy to give us a term to combine with the x

′y. This allowed us to factor out the y, leaving 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 3.11.

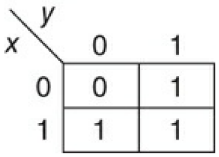


FIGURE 3.11 Kmap for F(x,y) = x + y

To use this map to reduce a Boolean function, we simply need to group 1s. Technically, if a rectangular grouping of 1s contains 1, 2, 4, 8, … (essentially any power of 2), it is called an implicant. A prime implicant is a group of 1s that isn't contained in any other group of 1s (i.e., it is the result of combining the maximum possible number of adjacent squares that results in a power of 2). An essential prime impli-cant is a group of 1s that contains at least one 1 that is not part of any prime impli-cant. Essential prime implicants are important because they must be part of the final result. Let's look at the rules for grouping first; we will return to implicants shortly.

This grouping is similar to how we grouped terms when we reduced using Boolean identities, except we must follow specific rules. First, we group only 1s. Second, we can group 1s in the Kmap if the 1s 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 1s if the total number in the group is a power of 2. The fourth rule specifies that we must make the groups as large as possible. As a fifth and final rule, all 1s 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 3.12 through 3.15.

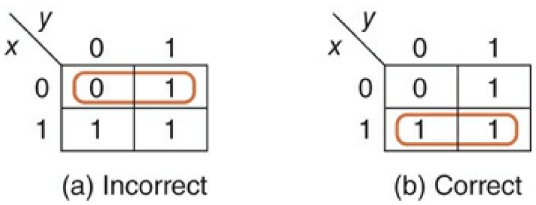


FIGURE 3.12 Groups Contain Only 1s

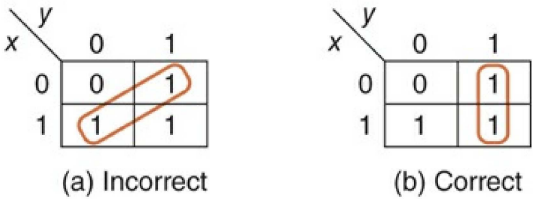


FIGURE 3.13 Groups Cannot Be Diagonal

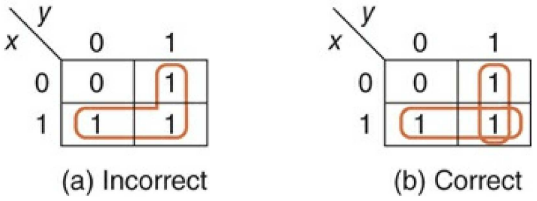


FIGURE 3.14 Groups Must Be Powers of 2

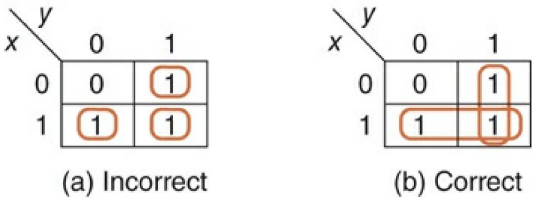


FIGURE 3.15 Groups Must Be as Large as Possible

Notice in Figures 3.14b and 3.15b 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 is 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 3.15b 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 xy′ and xy. This group represents the logical OR of these two terms, or xy′ + xy. These terms differ in y, so y is discarded, 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 xy + 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.

3.4.4 Kmap Simplification for Three Variables

Kmaps can be applied to expressions of more than two variables. In this section, we show three- 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 3.16.

x \ yz	00	01	11	10
0	X′Y′Z′	X′Y′Z	X′YZ	X′YZ′
1	XY′Z′	XY′Z	XYZ	XYZ′

FIGURE 3.16 Minterms and Kmap Format for Three Variables

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 corresponding minterms, x′y′z and 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.

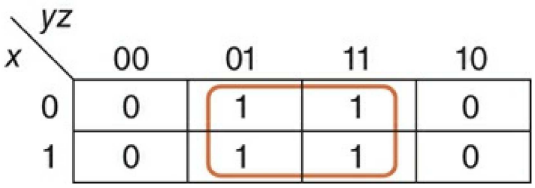
The largest groups we found in our two-variable examples were composed of two 1s. It is possible to have groups of four or even eight 1s, depending on the function. Let's look at a couple of examples of map simplification for expressions of three variables.

EXAMPLE 3.12

F(x, y, z) = x′y′z + x′yz + xy′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 that we must create the largest groups whose sizes are powers of two. There is one group of four, so we group these as follows:



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: x′y′z, x′yz, xy′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.

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

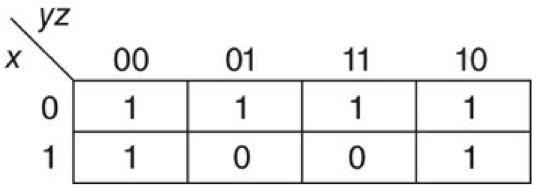
The end result using Boolean identities is exactly the

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.

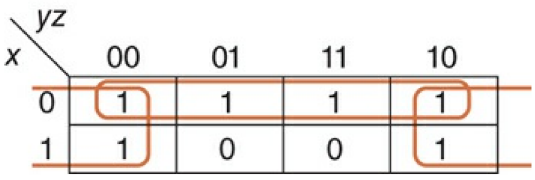
EXAMPLE 3.13

F(x, y, z) = x′y′z′ + x′ y′ z + x′ yz + x′ yz′ + xy′ z′ + xyz′



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:



We first choose all groups represented by essential prime implicants; then we pick from the remaining prime implicants such that all 1s are eventually covered. In this figure, we have two essential prime implicants (because the middle 1s in the top row belong to only one group, and the two 1s in the bottom row belong to only one group. The first group reduces to x′ (this is the only term the four have in common), and the second group reduces to z′, so the final minimized function is F(x, y, z) = x′ + z′.

EXAMPLE 3.14

Suppose we have a Kmap with all 1s:

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 see that we do indeed have a correct simplification. When using Kmaps to simplify Boolean expressions, just remember that the goal is to minimally cover (include in a group) all of the 1s. The resulting sum-of-products result is a sum of prime implicants.

3.4.5 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 3.17. Notice that the special order of 11 followed by 10 applies for the rows as well as the columns.

wx \ yz	00	01	11	10
00	W′X′Y′Z′	W′X′Y′Z	W′X′YZ	W′X′YZ′
01	W′XY′Z′	W′XY′Z	W′XYZ	W′XYZ′
11	WXY′Z′	WXY′Z	WXYZ	WXYZ′
10	WX′Y′Z′	WX′Y′Z	WX′YZ	WX′YZ′

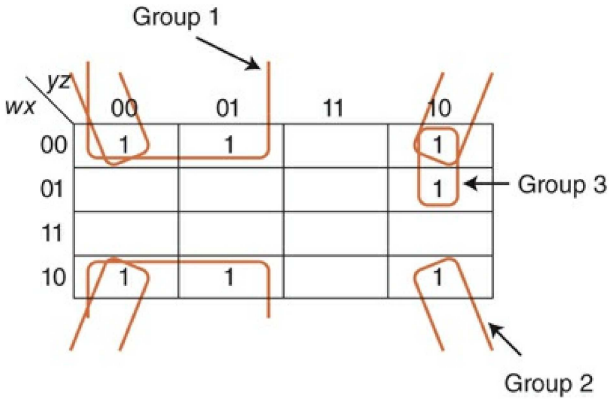
FIGURE 3.17 Minterms and Kmap Format for Four Variables

Example 3.15 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 3.15

F(w, x, y, z) = w′x′y′z′ + w′x′y′z + w′x′yz′ + w′xyz′ + wx

′y′z′ + wx′y′z + wx′yz′

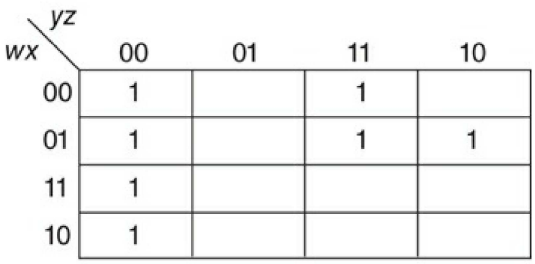


Group 1 is a “wraparound” group, as we saw previously. Group 3 is easy to find as well. Group 2 represents the ultimate wraparound group: It consists of the 1s in the four corners. Remember, these corners are logically adjacent. The final result is that F reduces to three terms, one from each group: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

Occasionally, there are choices to make when performing map simplification. Consider Example 3.16.

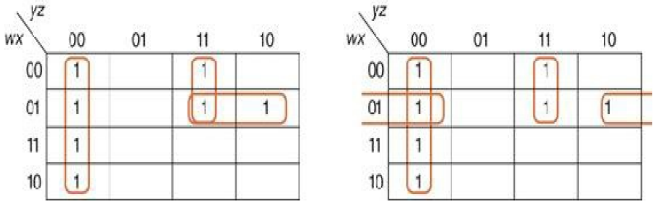
EXAMPLE 3.16

This example illustrates different choices when combining 1s



The first column should clearly be grouped because this is an essential prime implicant. (There is no other possible group that contains the w’x′y′z′ term, for example.) Also, the w’x′yz and w′xyz terms should be grouped, because the w’x′yz term is not part of any other possible group, making this an essential prime implicant. However, we have a choice as to how to group the w′xyz′ term. It could be grouped with w′xyz (this group is a

prime implicant) or with w′xy′z (as a wraparound, giving a different prime implicant). These two solutions are as follows:



The first map simplifies to F(w, x, y, z) = F_1 = y′z′ + w′yz + w′xy. The second map simplifies to F(w, x, y, z) = F_2 = y′z′ + w′yz + w′xz′. 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.

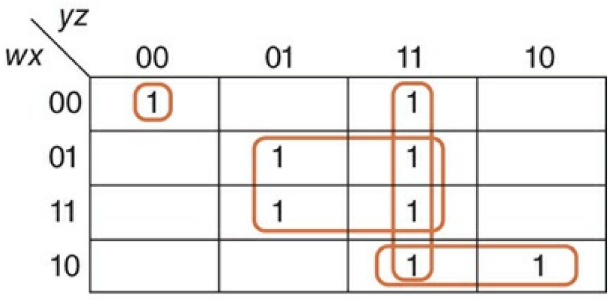
Before we move on to the next section, here are the rules for Kmap simplification:

1. The groups can contain only 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. Wraparounds are allowed.
8. Use the fewest number of groups possible.

Using these rules, let's complete one more example for a four-variable function. Example 3.17 shows several applications of the various rules.

EXAMPLE 3.17

F(w, x, y, z) = w′ x′ y′ z′ + w′ x′ yz + w′xy′ z + w′xyz + wxy′z + wxyz + wx′yz + wx′yz′



In this example, we have one group with a single element. Note that there is no way to group this term with any others (this is an essential prime impli-cant) if we follow the rules. The function represented by this Kmap simplifies to F(w, x, y, z) = yz + xz + w′x′y′z′ + wx ′y.

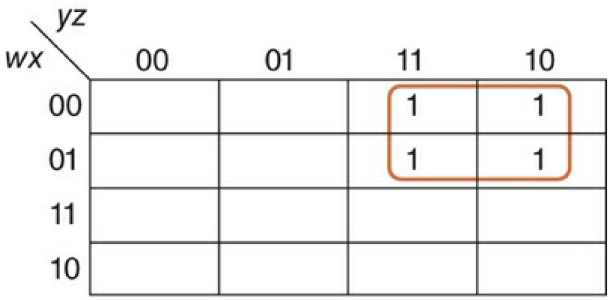
To summarize which groups to use to give us the resulting sum-of-products result, use these rules:

1. First, find all essential prime implicants (those groups that contain at least one 1 not covered by any other possible group). It's a good idea to start with 1s that have the fewest adjacent 1s.
2. Find prime implicants to cover the remaining 1s. Each new prime implicants should cover as many 1s as possible.
3. Write down the function represented by the groups that you have formed.

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 3.18 illustrates this procedure.

EXAMPLE 3.18

Suppose you are given the function F(w, x, y, z) = w′xy + w′x′yz + w′x′yz′. The last two terms are minterms, and we can easily place 1s in the appropriate positions in the Kmap. However, the term 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 w′xyz′ + w′xyz. You can now use these two terms in the Kmap, because they are both minterms. We now get the following Kmap:



So we know the function F(w, x, y, z) = w′xy + w′x′yz + w′x′yz′ simplifies to F(w, x, y, z) = w′y.

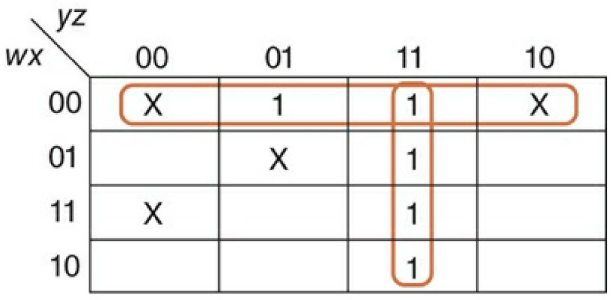
3.4.6 Don't Care Conditions

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 four 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.

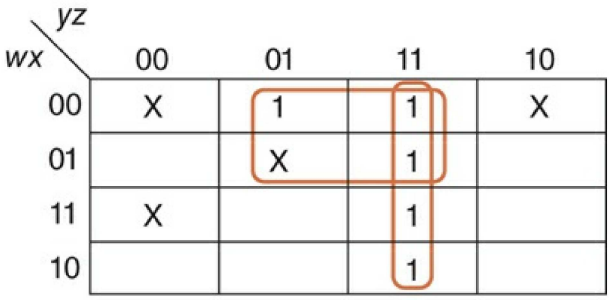
We can use these don't care inputs to our advantage when simplifying Kmaps. Because they are input values that should not matter (and should never occur), we can let them have values of either 0 or 1, depending on which helps us the most. The basic idea is to set these don't care values in such a way that they either contribute to making a larger group or don't contribute at all. Example 3.19 illustrates this concept.

EXAMPLE 3.19

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 $F_1(w, x, y, z) = w' x' + yz$.



There is another way these values can be grouped:



Using these groupings, we end up with a simplification of $F_2(w, x, y, z) = w' z + yz$. 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."

3.4.7 Summary

In this section, we have provided 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 the function represents.

3.5 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 pathways. 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.

3.5.1 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 + y′z (which we looked at earlier). Figure 3.18 represents a logic diagram that implements this function.

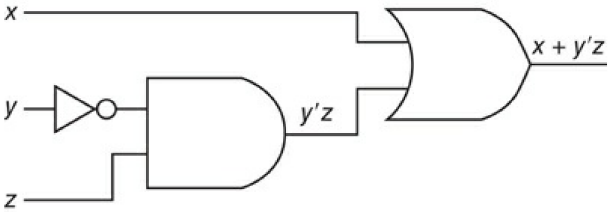
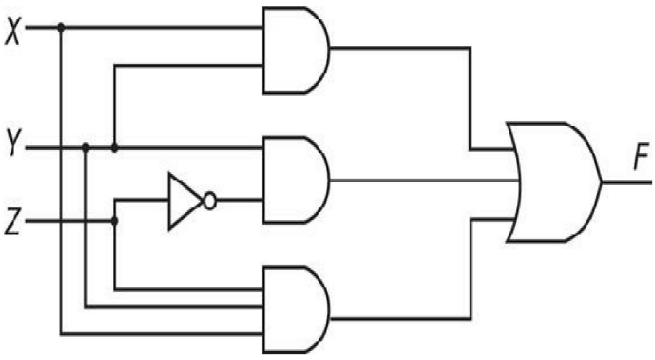


FIGURE 3.18 Logic Diagram for F(x,y,z) = x + y′z

Recall our discussion of sum-of-products form. This form lends itself well to implementation using digital circuits. For example, consider the function F(x, y, z) = xy + yz′ + xyz. Each term corresponds to an AND gate, and the sum is implemented by a single OR gate,

resulting in the following circuit:



We can build logic diagrams (which in turn lead to digital circuits) for any Boolean expression. At some level, every operation carried out by a computer is an implementation of a Boolean expression. This may not be obvious to high-level language programmers because the semantic gap between the high-level programming level and the Boolean logic level is so wide. Assembly language programmers, being much closer to the hardware, use Boolean tricks to accelerate program performance. A good example is the use of the XOR operator to clear a storage location, as in A XOR A. The XOR operator can also be used to exchange the values of two storage locations. The same XOR statement applied three times to two variables, say A and B, swaps their values:

- A = A XOR B
- B = A XOR B
- A = A XOR B

One operation that is nearly impossible to perform at the high-level language level is bit masking, where individual bits in a byte are stripped off (set to 0) according to a specified pattern. Boolean bit masking operations are indispensable for processing individual bits in a byte. For example, if we want to find out whether the 4's position of a byte is set, we AND the byte with 04_{16}. If the result is nonzero, the bit is equal to 1. Bit masking can strip off any pattern of bits. Place a 1 in the position of each bit that you want to keep, and set the others to 0. The AND operation leaves behind only the bits that are of interest.

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 effect 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, using the Kmaps from Section 3.4.

3.5.2 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, you are referred to the reading list at the end of this chapter for more information on this topic.

Typically, gates are not sold individually; they are sold in units called integrated circuits (ICs). A chip (a silicon semiconductor crystal) is a small electronic device consisting of the necessary electronic components (transistors, resistors, and capacitors) to implement various gates. As already explained, 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, the first ICs were called SSI chips and contained up to 100 electronic components per chip. We now have ultra-large-scale integration (ULSI) with more than 1 million electronic components per chip. Figure 3.19 illustrates a

simple SSI IC.

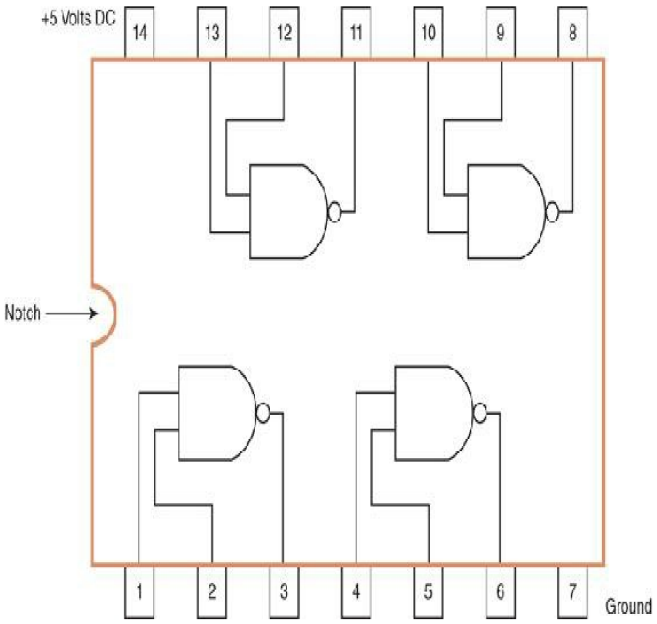
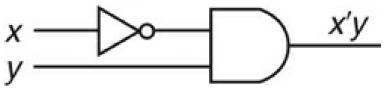


FIGURE 3.19 Simple SSI Integrated Circuit

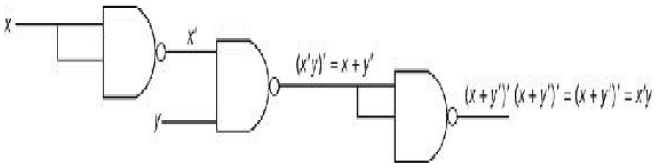
We have seen that we can represent any Boolean function as (1) a truth table, (2) a Boolean expression (in sum-of-products form), or (3) a logic diagram using gate symbols. Consider the function represented by the following truth table:

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

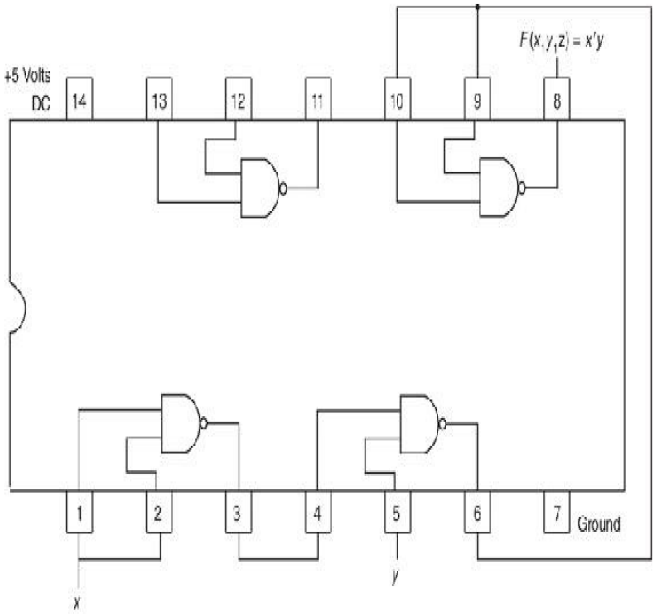
This function is expressed in sum-of-products form as F(x, y, z) = x′yz′ + x′yz. This simplifies to F(x,y,z) = x′y (the simplification is left as an exercise). We can now express this using a logic diagram as follows:



Using only NAND gates, we can redraw the logic diagram as follows:



We can implement this in hardware using the SSI circuit from Figure 3.19 as follows:



3.5.3 Putting It All Together: From Problem Description to Circuit

We now understand how to represent a function by a Boolean expression, how to simplify a Boolean expression, and how to represent a Boolean expression using a logic diagram. Let's combine these skills to design a circuit from beginning to end.

EXAMPLE 3.20

Suppose we are given the task of designing a logic circuit to help us determine the best time to plant our garden. We investigate three possible factors: (1) time, where 0 represents day and 1 represents evening; (2) moon phase, where 0 represents not full and 1 represents full; and (3) temperature, where 0 represents 45 °F and below, and 1 represents over 45 °F. These three items represent our inputs. After significant research, we

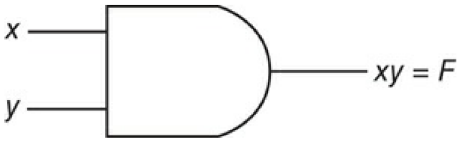
determine that the best time to plant a garden is during the evening with a full moon (temperature does not appear to matter). This results in the following truth table:

Time (x)	Moon (y)	Temperature (z)	Plant?
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

We have placed 1s in the output column when the inputs indicated “evening” and “full moon,” and 0s everywhere else. By converting our truth table to a Boolean function F, we see that F(x, y, z) = xyz′ + xyz (we use a process similar to that presented in Example 3.9: We include terms where the function evaluates to 1). We now simplify F:

$$F(x,y,z) = xyz' + xyz$$
$$= xy\,(\text{using the absorption law})$$

Therefore, this function evaluates to one AND gate using x and y as input.



The steps for designing a Boolean circuit are as follows: (1) read the problem carefully to determine the input and output values; (2) establish a truth table that shows the output for all possible inputs; (3) convert the truth table into a Boolean expression; and (4) simplify the Boolean expression.

EXAMPLE 3.21

Suppose you are responsible for designing a circuit that will allow the president of your college to determine whether to close the campus due to weather conditions. If the highway department has not salted the area roads, and there is ice on the roads, the campus should be closed. Regardless of whether there is ice or salt on the roads, if there is more than 8 in. of snow, the campus should be closed. In all other situations, the campus should remain open.

There are three inputs: ice (or no ice), salt (or not salt), and snow of more than 8 in. on the roads (or not), resulting in the following truth table:

Ice (x)	Salt (y)	Snow (z)	Close?
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

The truth table yields the Boolean expression F(x, y, z) = x′y′z + x′ yz + xy′ z′ + xy′ z + xyz. We can simplify this expression using Boolean identities as follows:

$$
\begin{aligned}
F(x,y,z) &= x'y'z + x'yz + xy'z' + xy'z + xyz \\
&= x'y'z + x'yz + xy'z + xyz + xy'z' && \text{(Commutative)} \\
&= x'(y'z + yz) + x(y'z + yz) + xy'z' && \text{(Distributive} \times 2\text{)} \\
&= (x' + x)(y'z' + yz) + xy'z' && \text{(Distributive)} \\
&= (y'z' + yz) + xy'z' && \text{(Inverse/Identity)} \\
&= (y' + y)z + xy'z' && \text{(Distributive)} \\
&= z + xy'z' && \text{(Inverse/Identity)}
\end{aligned}
$$

We leave it to the reader to draw the logic diagram corresponding to z + xy′ z′. Once the circuit has been implemented in hardware, all the college president has to do is set the inputs to indicate the current conditions,

and the output will tell her whether to close campus.

3.6 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 3.7.

3.6.1 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 (as we saw in Examples 3.20 and 3.21). 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.

3.6.2 Examples of Typical Combinational Circuits

Let's begin with a very simple combinational circuit called a half-adder.

EXAMPLE 3.22

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 3.9.

TABLE 3.9 Truth Table for a Half-Adder

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

1	1	0	1

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 3.20.

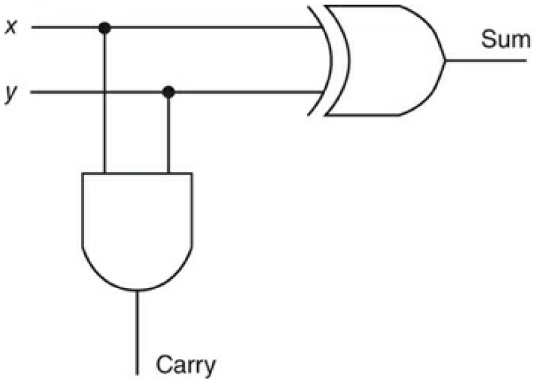


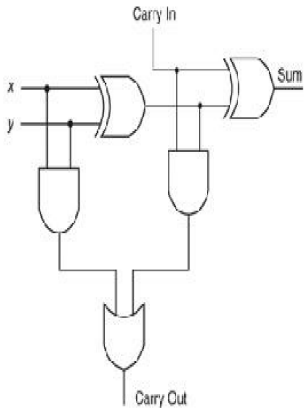
FIGURE 3.20 Logic Diagram for a Half-Adder

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 manner. We can add binary numbers in the same way. However, we need a circuit that allows three inputs (x, y, and Carry In) and two outputs (Sum and Carry Out).

Figure 3.21 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.

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

(a)



(b)

FIGURE 3.21 (a) Truth Table for a Full Adder
(b) Logic Diagram for a Full Adder

Given this full adder, you may be wondering how this circuit can add binary numbers; 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 3.22 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.

FIGURE 3.22 Logic Diagram for a Ripple-Carry Adder

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 attempt to shorten the delay required adding two binary numbers. In fact, these newer adders achieve speeds of 40-90% 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 one specific

output line"? It simply means that one unique output line is asserted, or set to 1, whereas the other output lines are set to 0. 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 first has to determine the actual address. This is done using a decoder. Example 3.23 should clarify any questions you may have about how a decoder works and what it might be used for.

NULL POINTERS: TIPS AND HINTS

Computer science students are required to take many classes; one such class typically includes discrete math. It is important for you to realize that the concepts you learn in a class like this can be applied to many different areas, including computer architecture. For example, consider the ripple-carry adder in Figure 3.22 extended to / bits. It you were asked to express the i^{th} carry bit, c_i, as a logical function, by reviewing the full-adder in Figure 3.21, we see that c_i can be expressed as the function:

$$c_i = x_{i-1}y_{i-1} + (x_{i-1} + y_{i-1})c_{i-1}$$

Note that this is a recurrence (c_i is expressed in terms of c_{i-1}), which is something studied in a discrete math class. If we could find an expression for c_i that removes this dependency on c_{i-1}, we could design a better, faster adder (because each stage would not have to wait for the previous one to produce a carry). If we represent the product term xy by p and the sum term x + y by s, we can

solve the recurrence as follows:

$$
\begin{aligned}
c_i &= x_{i-1}y_{i-1} + (x_{i-1} + y_{i-1})c_{i-1} \\
&= p_{i-1} + s_{i-1}c_{i-1} \\
&= p_{i-1} + s_{i-1}(p_{i-2} + s_{i-2}c_{i-2}) \\
&= p_{i-1} + s_{i-1}p_{i-2} + s_{i-1}s_{i-2}c_{i-2} \\
&= p_{i-1} + s_{i-1}p_{i-2} + s_{i-1}s_{i-2}(p_{i-3} + s_{i-3}c_{i-4}) \\
&= p_{i-1} + s_{i-1}p_{i-2} + s_{i-1}s_{i-2}p_{i-3} + s_{i-1}s_{i-2}s_{i-3}c_{i-4} \\
&\cdots \\
&= p_{i-1} + s_{i-1}p_{i-2} + s_{i-1}s_{i-2}p_{i-3} + \cdots + s_{i-1}s_{i-2} \cdots s_i c_0
\end{aligned}
$$

As you can see, the equation for the adder no longer contains a dependence of $carry_i$ on $carry_{i-1}$. We can now design the combinational logic for an adder that needs only c_0 and the regular inputs. (This is left as an exercise for the reader.)

EXAMPLE 3.23

Here's an example of a 3-to-8 decoder circuit. Imagine memory consisting of 8 chips, each containing 8K bytes. Let's assume that chip 0 contains memory addresses 0-8191 (or 1FFF in hex), chip 1 contains memory addresses 8192-16,383 (or 2000-3FFF in hex), and so on. We have a total of 8K χ 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 $64K = 2^6 \chi 2^{10} = 2^{16}$, which indicates that 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 four addresses— addresses 0, 1, 2, and 3, the binary equivalent of these addresses is 00, 01, 10, and 11, requiring two bits. We know that $2^2 = 4$. 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. (We will see this idea again later in this chapter, as well as in Chapters 4 and 6.)

All addresses on chip 0 have the format 000xxxxxxxxxxxxx. 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 that 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 3.23 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 3.7.

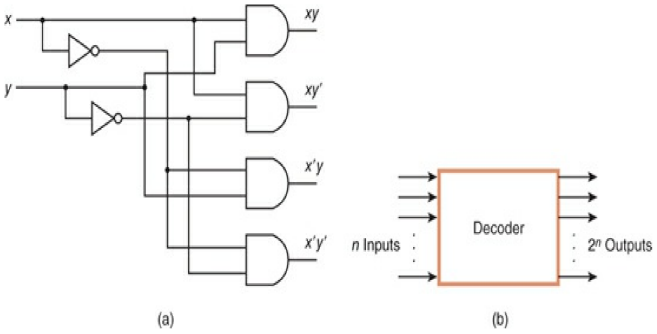


FIGURE 3.23 (a) A Look Inside a Decoder
(b) A Decoder Symbol

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 variables, 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 3.24 illustrates the physical components in a multiplexer and the symbol often used to represent a multiplexer. S_0 and S_1 are the control lines; I_0 - I_3 are the input values.

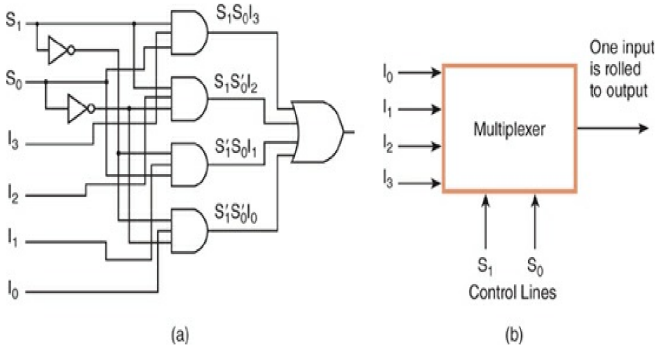


FIGURE 3.24 (a) A Look Inside a Multiplexer
(b) A Multiplexer Symbol

Another useful set of combinational circuits to study includes a parity generator and a parity checker (recall that 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 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 3.10. 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 3.11. 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.

TABLE 3.10 Parity Generator

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 Checker

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

Bit shifting, moving the bits of a word or byte one position to the left or right, is a very useful operation. Shifting a bit to the left takes it to the position of the next higher power of two. When the bits of an unsigned integer are shifted to the left by one position, it has the same effect as multiplying that integer by 2, but using significantly fewer machine cycles to do so. The leftmost or rightmost bit is lost after a left or right shift (respectively). Left-shifting the nibble, 1101, changes it to

1010, and right-shifting it produces 0110. Some buffers and encoders rely on shifters to produce a bit stream from a byte so that each bit can be processed in sequence. A 4-bit shifter is illustrated in Figure 3.25. When the control line, S, is low (i.e., zero), each bit of the input (labeled I_0 through I_3) is shifted left by one position into the outputs (labeled O_0 through O_3). When the control line is high, a right shift occurs. This shifter can easily be expanded to any number of bits, or combined with memory elements to create a shift register.

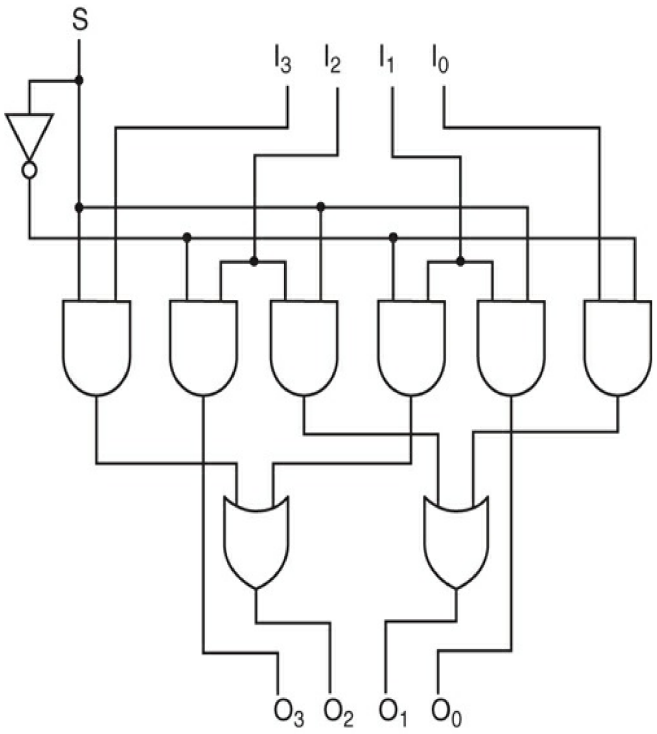


FIGURE 3.25 4-Bit Shifter

There are far too many combinational circuits for us to be able to cover them all in this brief chapter. The references at the end of this chapter provide much more information on combinational circuits than we can give here. 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 3.26 illustrates a 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 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, and B_0 and B_1 indicate the second word. C_0 and C_1 represent the output lines.

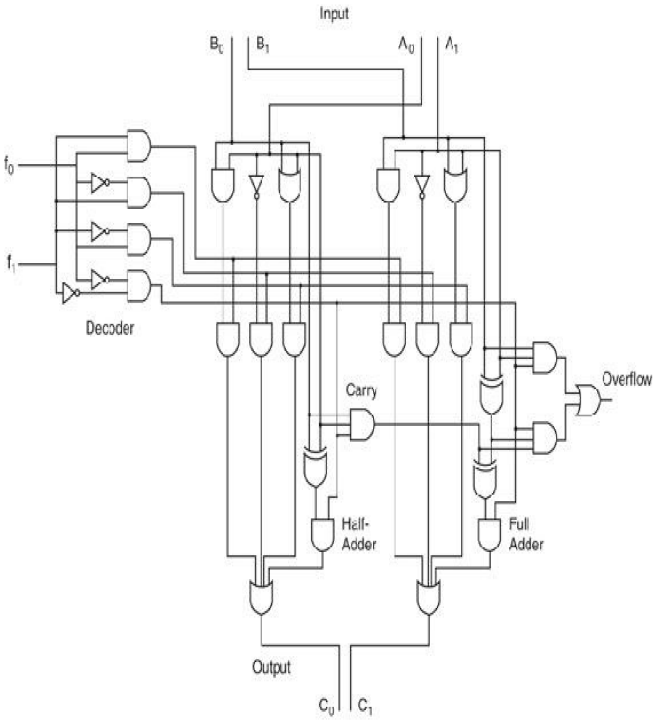


FIGURE 3.26 A Simple Two-Bit ALU

Let's look at one more circuit—a circuit to multiply two binary numbers together. This particular multiplier assumes two 2-bit numbers; however, it can be extended for larger numbers. In order to multiply two numbers, we use the halfadder from Figure 3.20. The two numbers, x and y, are represented by x_1x_0 and y_1 y_0. We multiply them together as follows:

$$\begin{array}{cccc} & & y_1 & y_0 \\ & & \times & x_1 x_0 \\ \hline & & x_0 y_1 & x_0 y_0 \\ + & x_1 y_1 & x_1 y_0 & \\ \hline z_3 & z_2 & z_1 & z_0 \end{array}$$

Each of the z¿'s represents a bit in the product resulting from the multiplication. In Figure 3.27, we show where each of the z terms comes from. The z_0 is simply the

result of x_0y_0, and z_1 is produced by adding the x_0y_1 term to the x_1y_0 term. The sum becomes z_1; any carry produced from this step is added to x_1y_1 to produce z_2. A carry from this last step becomes z_3. In general, we would use full adders for larger numbers.

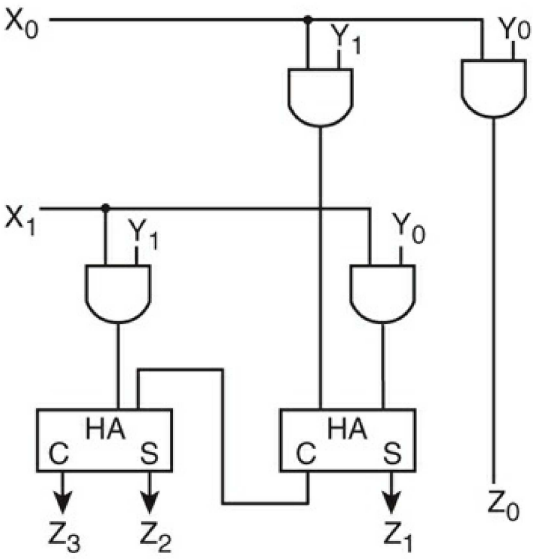


FIGURE 3.27 A Simple Two-Bit Multiplier

NULL POINTERS: TIPS AND HINTS

Let's review what we have learned so tar. Boolean algebra is a very natural way to represent digital information, making it an important concept it we want to understand how a computer makes decisions. Combinational circuits represent a large portion of the actual components used in today's machines. Adders, decoders, multiplexers, and parity checkers are just a few examples. Understanding these simple circuits goes a long way toward understanding the overall system. Although Boolean algebra allows us to represent expressions in abstract form, the actual building blocks we use for digital systems are logic gates. It is important that you understand how Boolean algebra relates to digital components—how Boolean expressions are physically implemented. Remember, there is a one-to-one correspondence between a Boolean expression and its digital representation. The complexity of a Boolean

expression has a direct impact on the complexity of the resulting digital circuit. Although Boolean expressions can be used to reduce these expressions, resulting in minimized circuits, Kmaps can simply that process, because when using Boolean identities, there is no way to know if the answer is indeed the simplest form of the expression; Kmaps, when used correctly, guarantee the minimum standard form of a Boolean function.

One of the most frequently asked questions is how to define minimal circuit. Using Kmaps, as mentioned previously, we are guaranteed to find the minimal form of a Boolean expression; this is one that implements the expression with as few literals and product terms as possible. (Remember, there may be more than one minimal form.) However, minimal circuit is usually defined as the circuit requiring the fewest gates to implement. (We note that the number of levels, the stages from input to output, is also very important, because the more levels there are, the more propagational delay the circuit has.) The minimal form resulting from a Kmap (in sum-of-products form) may not result in the fewest number of gates! For example, suppose that you use a Kmap and the resulting expression is xy + xz. This is implemented using three gates and two levels: two AND gates to get the xy and xz terms, and one OR gate to combine these two terms. However, the expression x (y + z), which we know to be equivalent to xy + xz, requires only two gates with two levels: an OR gate with inputs of y and z resulting in y + z, and then one AND gate to combine this result with x. Kmaps will always guarantee a sum-of-products form, which means the circuit will always be two levels, but it may not guarantee the minimal number of gates.

Another thing to consider when minimizing the circuit is the negation of an expression, because sometimes the negation of an expression results in fewer terms, and often fewer gates, than the expression itself Consider the Boolean function F(x, y, z) that is true for all values except when x, y, and z are all equal to 1. Using Kmaps, this reduces to x′ + y′ + z′′, which requires four gates (a NOT gate for each variable and an AND gate to combine the terms) and two levels. However, the negative of this expression, F′, is true only when all three variables are 1, resulting in one 1 in the Kmap. To implement this, we

need only two gates: an AND gate to get xyz and a NOT gate to negate the result.

Lastly, a circuit often implements multiple Boolean expressions (as in the simple ALU circuit in Figure 3.26). It may be that if some expressions are negated, they may share terms with other expressions, resulting in fewer gates overall. What this means is that minimizing a circuit may require looking at both the expressions and the negation of the expressions, and figuring out which combination results in the fewest number of gates.

3.7 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 effect 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 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 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.

3.7.1 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.

3.7.2 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 that 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 gigahertz (GHz).

A clock is used by a sequential circuit to decide when to update the state of the circuit (i.e., 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 3.28.

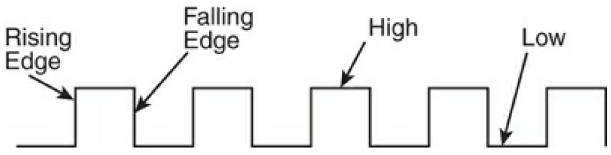


FIGURE 3.28 A Clock Signal Indicating Discrete Instances of Time

3.7.3 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 text, we use the term flip-flop. William Eccles and F. W. Jordan invented the first flip-flop (from vacuum tubes) in 1918, so these circuits have been around for some time. However, they have not always been called flip-flops. Like so many other inventions, they were originally named after the inventors and were called Eccles-Jordan trigger circuits. So where did flip-flop come from? Some say it was the sound the circuit made (as produced on a speaker connected to one of the components in the original circuit) when it was triggered; others believe it came from the circuit's ability to flip from one state to another and back again.

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 3.29. 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.

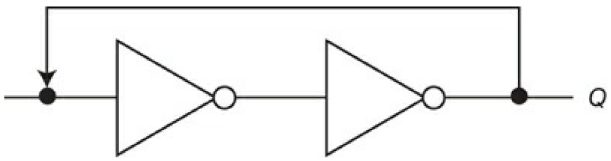


FIGURE 3.29 Example of Simple Feedback

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 3.30.

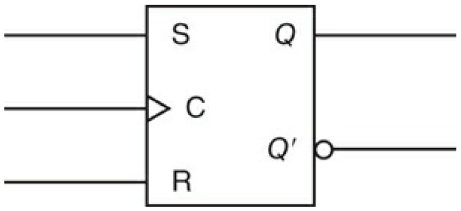
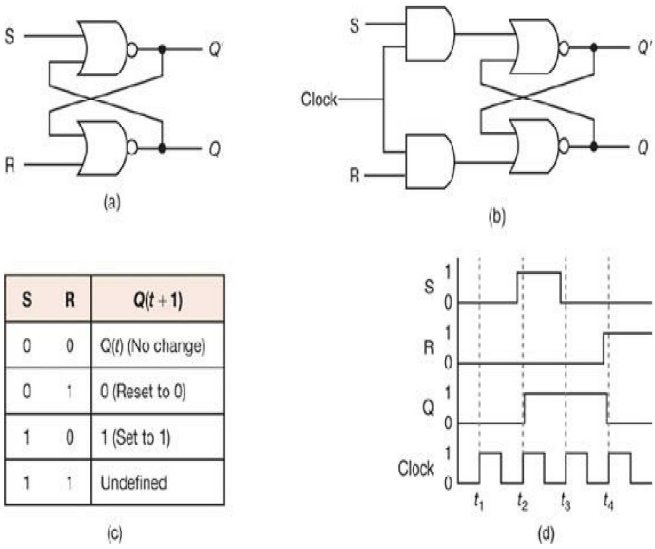


FIGURE 3.30 SR Flip-Flop Logic Diagram

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. We can also specify a timing diagram, which indicates the relationship of signals from the clock to changes in a flip-flop's output. Figure 3.31a shows the actual implementation of the SR sequential circuit; Figure 3.31b adds a clock to the flip-flop; Figure 3.31c specifies its characteristic table; and Figure 3.31d shows an example timing diagram. We are interested in exploring only clocked flip-flops.



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

(c)

FIGURE 3.31 (a) SR Flip-Flop
(b) Clocked SR Flip-Flop
(c) Characteristic Table for the SR Flip-Flop
(d) Timing Diagram for the SR Flip-Flop (assuming the initial state of Q is 0)

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 3.12 to further illustrate how this circuit works.

TABLE 3.12 Truth Table for 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

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 set to 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.

Looking at the example timing diagram in Figure 3.31d, we see that at time t_1, the clock ticks, but because S = R = 0, Q does not change. At t_2, S has changed to 1, and R is still 0, so when the clock ticks, Q is set to 1. At t_3, S = R = 0, so Q does not change. By t_4, because R has changed to 1, when the clock ticks, S = 0, R = 1, and Q is reset to 0.

There is one oddity with this particular flip-flop. What happens if both S and R are set to 1 at the same time? If we examine the unclocked flip-flop in Figure 3.31a, this forces a final state in which both Q and Q′ are 0, but how can Q = 0 = Q′ ? Let's look at what happens when S = R = 1 using the clocked flipflop in Figure 3.31b. "When the clock pulses, the S and R values are input into the flip-flop. This forces both Q and Q′ to 0. When the clock pulse is removed, the final state of the flip-flop cannot be determined, as once the clock pulse ends, both the S and R inputs are killed, and the resulting state depends on which one actually terminated first (this situation is often called a race conditiom). 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 3.32. This results in a JK flip-flop. A JK flip-flop is basically the same thing as an SR flip-flop except when both inputs are 1, this circuit negates the current state. The timing diagram in Figure 3.32d illustrates how this circuit works. At time t_1, J = K = 0, resulting in no change to Q. At t_2, J = 1 and K = 0, so Q is set to 1. At t_3, K = J = 1, which causes Q to be negated, changing it from 1 to 0. At t_4, K = 0 and J = 1, forcing Q to be set to 1.

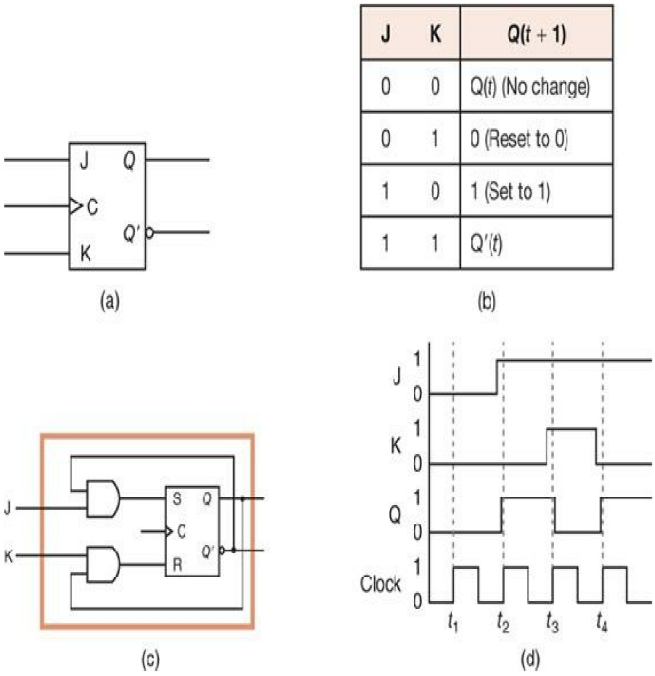


FIGURE 3.32 (a) JK Flip-Flop
(b) JK Characteristic Table
(c) JK Flip-Flop as a Modified SR Flip-Flop
(d) Timing Diagram for JK Flip-Flop (assuming the initial state of Q is 0)

There appears to be significant disagreement regarding where the JK came from. Some believe it was named after Jack Kilby, inventor of the integrated circuit. Others believe it is named after John Kardash, who is often credited as its inventor (as specified in his biographical data on his current company's website). Still others believe it was coined by workers at Hughes Aircraft who labeled circuits input using letters, and J and K just happened to be next on the list (as detailed in a letter submitted to the electronics magazine EDN in 1968).

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 3.33 illustrates the D flip-flop, lists its characteristic table and timing diagram, and reveals that

the D flip-flop is actually a modified SR flip-flop.

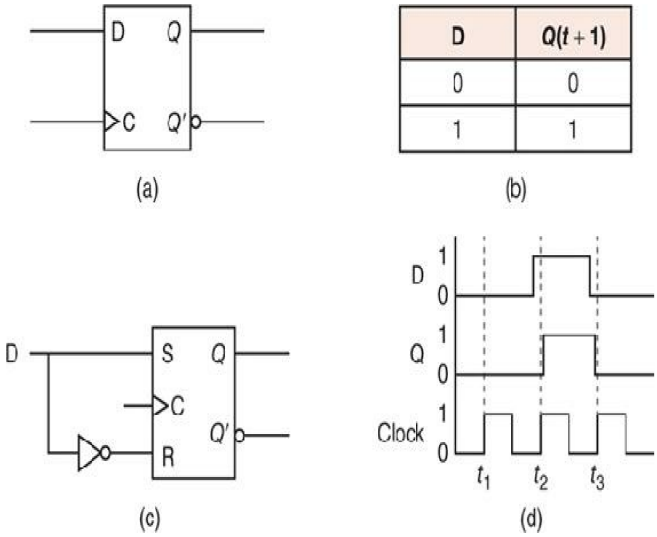


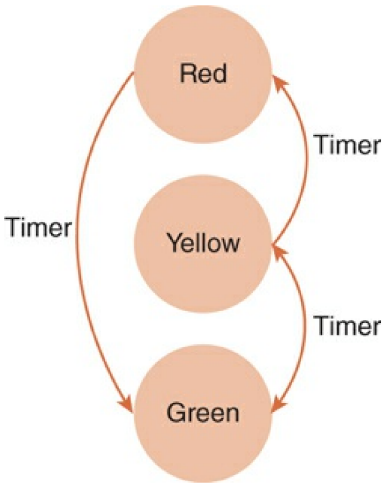
FIGURE 3.33 (a) D Flip-Flop
(b) D Flip-Flop Characteristic Table
(c) D Flip-Flop as a Modified SR Flip-Flop
(d) Timing Diagram for D Flip-Flop

3.7.4 Finite-State Machines

Characteristic tables and timing diagrams allow us to describe the behavior of flip-flops and sequential circuits. An equivalent graphical depiction is provided by a finite-state machine (FSM). Finite-state machines are very important when discussing the control and decision logic in digital systems. FSMs are “finite” because there are only a fixed number of states, and the output and next state are functions of the input and present state (which is why they are critical in realizing sequential logic functions). FSMs typically use circles to represent a set of machine states and directed arcs to represent transitions from one state to another. Each circle is labeled with the state it represents, and each arc is labeled with the input and/or output for that state transition. FSMs can be in only one state at a time. We are interested in synchronous FSMs (those allowing state transitions only when the clock ticks); on each clock cycle, the machine uses an input and generates a new state (it could remain in the same state) and a new output.

A real-world example that can be modeled with state machines is a common traffic light. It has three states: green, yellow, and red. The light can change from a green state to a yellow state to a red state, but not from a red

state to a yellow state. Thus, the FSM tells us precisely how a traffic light operates. Transitions among states occur as timers in the hardware expire. An FSM for a traffic light appears below:



There are a number of different kinds of finite-state machines, each suitable for a different purpose. Figure 3.34 shows a Moore machine representation of a JK flip-flop. With Moore machines, the outputs depend only on the present state. The circles represent the two states of the flip-flop, which we have labeled A and B. The output, Q, is indicated in brackets, and the arcs illustrate the transitions between the states. We can see in this figure exactly how a JK flip-flop goes from state 0 to state 1 when J = 1 and S = 0, or when J = K = 1, and how it goes from state 1 to state 0 when J = K = 1, or when J = 1 and K = 0. This finite-state machine is a Moore-type machine because each of the states is associated with the output of the machine. In fact, the reflexive arcs shown in the figure are not required because the output of the machine changes only when the state changes, and the state does not change through a reflexive arc. We can therefore draw a simplified Moore machine (Figure 3.35). Moore machines are named for Edward F. Moore, who invented this type of FSM in 1956.

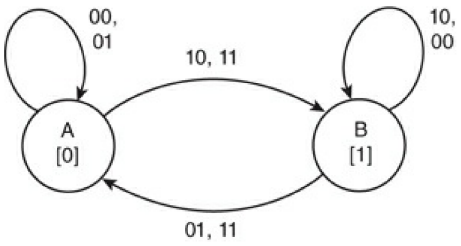


FIGURE 3.34 JK Flip-Flop Represented as a Moore Machine

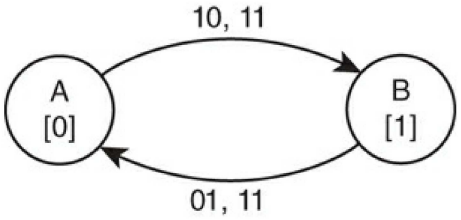


FIGURE 3.35 Simplified Moore Machine for the JK Flip-Flop

A contemporary of Edward Moore, George H. Mealy, independently invented another type of FSM that has also been named after its inventor. Like a Moore machine, a Mealy machine consists of a circle for each state, and the circles are connected by arcs for each transition. Unlike a Moore machine, which associates an output with each state (indicated in the Moore machine example by putting a 0 or 1 in square brackets), a Mealy machine associates an output with each transition. This implies that a Mealy machine's outputs are a function of its current state and its input, and a Moore machine's output is a function only of its current state. Each transition arc is labeled with its input and output separated by a slash. Reflexive arcs cannot be removed from Mealy machines because they depict an output of the machine. A Mealy machine for our JK flip-flop is shown in Figure 3.36.

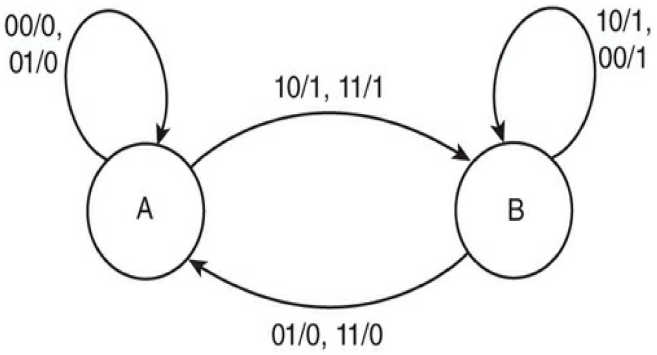


FIGURE 3.36 JK Flip-Flop Represented as a Mealy Machine

In the actual implementation of either a Moore or Mealy machine, two things are required: a memory (register) to store the current state and combinational logic components that control the output and transitions from

one state to another. Figure 3.37 illustrates this idea for both machines.

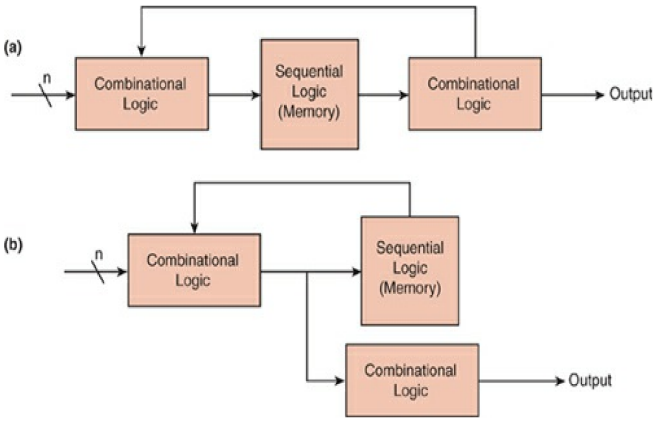
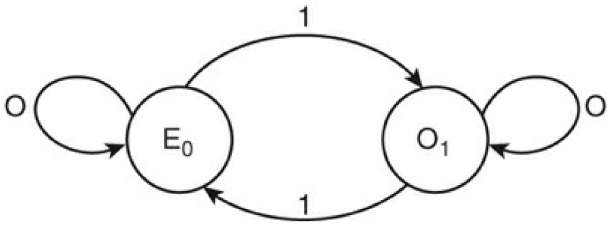


FIGURE 3.37 (a) Block Diagram for Moore Machines (b) Block Diagram for Mealy Machines

The general approach when using FSMs for design is: (1) Read the problem specification carefully—it is important to understand exactly what the circuit is supposed to do; (2) draw the state diagram that expresses the behavior of the circuit being implemented; (3) find the truth table resulting from the state diagram (you could also use a characteristic table); and (4) write the circuit in sum-of-products form. Example 3.24 illustrates this process.

EXAMPLE 3.24

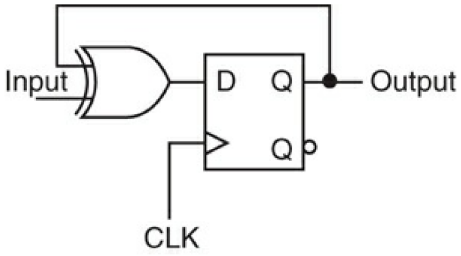
Suppose we want to design an even parity checker, a circuit that will detect an even number of 1s in the input. The following state diagram captures the circuit's behavior, where state E represents an even number of 1s (in our truth table, we represent this state using 0, hence the E_0 label), and O_1 represents an odd number of 1s:



If you trace this machine, you can see that it correctly identifies whether or not it has seen an odd number of 1s or an even number of 1s. Next, we need to find the truth table for this machine:

Input	Present State	Next State	Present Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

The Next State column identifies the behavior of the circuit. You should recognize this function as the XOR function, which means the next state is just the present state XORed with the present input. Implemented with circuits, we have:



EXAMPLE 3.25

Suppose we want to design a vending machine that requires a nickel and a dime to dispense a widget. The circuit must keep track of how much money has been inserted. A few notes: We are not going to worry about making change; we represent 0 cents by 00, 5 cents by 01, 10 cents by 10, and 15 cents by 11; and the output determines whether the vending machine “triggers” or not, but must wait one clock cycle before triggering.

The state table for this vending machine is:

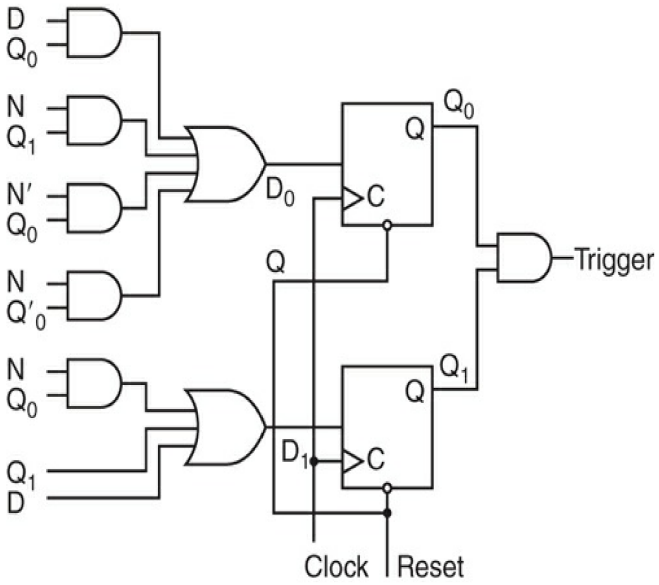
Present State	Inputs N	D	Next State	Present Output
0¢ = 00	0	0	0¢	0
	0	1	10¢	0
	1	0	5¢	0
	1	1	-	-
5¢ = 01	0	0	5¢	0
	0	1	15¢	0
	1	0	10¢	0
	1	1	-	-

10¢ = 10		0	0	10¢	0
		0	1	15¢	0
		1	0	15¢	0
		1	1	-	-
15¢ = 11		-	-	15¢	1

We are going to use D flip-flops to store the current states; because we have four states, we need two flip-flops, D_0 and D_1, with outputs Q_0 and Q_1, respectively, resulting in the following:

Present State		Inputs		Next State		Present Output (Trigger)
Q_1	Q_0	N	D	D_1	D_0	
0	0	0	0	0	0	0
		0	1	1	0	0
		1	0	0	1	0
		1	1	-		-
0	1	0	0	0	1	0
		0	1	1	1	0
		1	0	1	0	0
		1	1	-		-
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	-		-
1	1	-	-	1	1	1

We now need to find the function for D_1, D_0, and Trigger. We leave this as an exercise. (Hint: Try using Kmaps to find a minimized function for D_1, then again for D_0, and then again for Trigger.) The resulting circuit is shown below:



The graphical models and the block diagrams that we have presented for the Moore and Mealy machines are useful for high-level conceptual modeling of the behavior of circuits. However, once a circuit reaches a certain level of complexity, Moore and Mealy machines become unwieldy and only with great difficulty capture the details required for implementation. Consider, for example, a microwave oven. The oven will be in the “on” state only when the door is closed, the control panel is set to “cook” or “defrost,” and there is time on the timer. The “on” state means that the magnetron is producing microwaves, the light in the oven compartment is lit, and the carousel is rotating. If the time expires, the door opens, or the control is changed from “cook” to “off,” the oven moves to the “off” state. The dimension provided by the timer, along with the numerous signals that define a state, is hard to capture in the Moore and Mealy models. For this reason, Christopher R. Clare invented the algorithmic state machine (ASM). As its name implies, an algorithmic state machine is directed at expressing the algorithms that advance an FSM from one state to another.

An algorithmic state machine consists of blocks that contain a state box, a label, and optionally condition and output boxes (Figure 3.38). Each ASM block has exactly one entry point and at least one exit point. Moore-type outputs (the circuit signals) are indicated inside the state block; Mealy-type outputs are indicated in the oval output “box.” If a signal is asserted when “high,” it is

prefixed with an H; otherwise, it is prefixed with an L. If the signal is asserted immediately, it is also prefixed with an I; otherwise, the signal asserts at the next clock cycle. The input conditions that cause changes in state (this is the algorithmic part) are expressed by elongated, six-sided polygons called condition boxes. Any number of condition boxes can be placed inside an ASM block, and the order in which they are shown is unimportant. An ASM for our microwave oven example is shown in Figure 3.39.

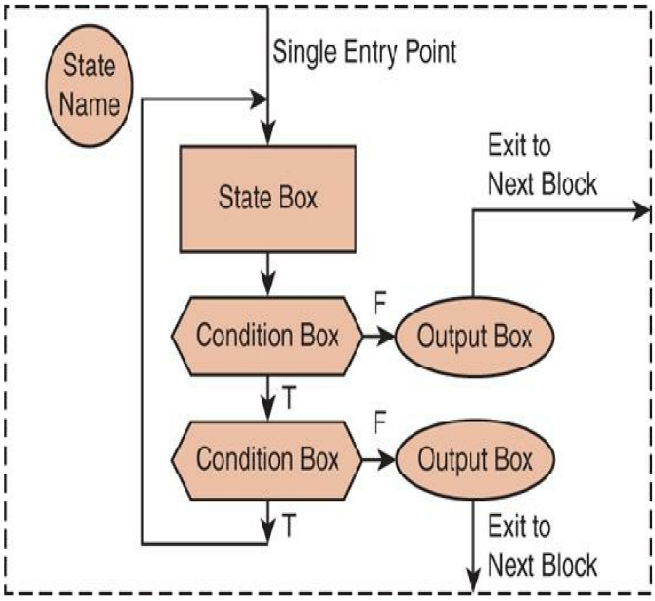


FIGURE 3.38 Components of an Algorithmic State Machine

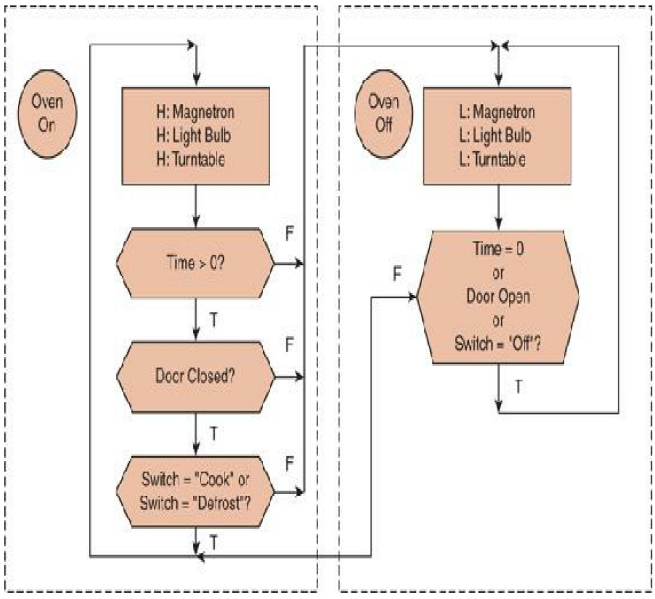


FIGURE 3.39 Algorithmic State Machine for a Microwave Oven

As implied, ASMs can express the behavior of either a Moore or Mealy machine. Moore and Mealy machines are probably equivalent and can be used interchangeably. However, it is sometimes easier to use one rather than the other, depending on the application. In most cases, Moore machines require more states (memory) but result in simpler implementations than Mealy machines, because there are fewer transitions to account for in Moore machines.

HARDWARE-FREE MACHINES

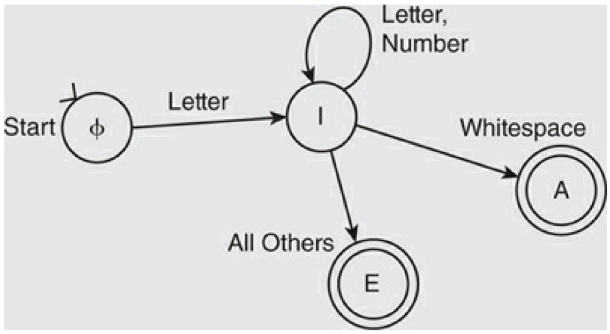
Moore and Mealy machines are only two of many different types of finite-state machines that you will encounter in computer science literature. An understanding of FSMs is essential in the study of programming languages, compilers, the theory of computation, and automata theory. We refer to these abstractions as machines because machines are devices that respond to a set of stimuli (events) by generating predictable responses (actions) based on a history of prior events (current state). One of the most important of these is the deterministic finite automata (DFA) computational model. Formally speaking, a DFA, M, is completely described by the quintuple $M = (Q, S, \Sigma, \delta, F)$, where:

- Q is a finite set of states that represents every configuration the machine can assume;
- S is an element of Q that represents the start state, which is the initial state of the machine before it receives any inputs;
- Σ is the input alphabet or set of events that the machine will recognize;
- δ is a transition function that maps a state in Q and a letter from the input alphabet to another (possibly the same) state in Q; and
- F is a set of states (elements of Q) designated as the final (or accepting) states.

DFAs are particularly important in the study of programming languages; they are used to recognize grammars or languages. To use a DFA, you begin in the Start state and process an input string, one character at a time, changing states as you go. Upon processing the entire string, if you are in a final accepting state, a legal

string is “accepted” by that DFA. Otherwise, the string is rejected.

We can use this DFA definition to describe a machine—as in a compiler—that extracts variable names (character strings) from a source code file. Suppose our computer language accepts variable names that must start with a letter, can contain an infinite stream of letters or numbers following the initial letter, and is terminated by a whitespace character (tab, space, linefeed, etc.). The initial state of the variable name is the null string, because no input has been read. We indicate this starting state in the figure below with an exaggerated arrowhead (there are several other notations). When the machine recognizes an alphabetic character, it transitions to State /, where it stays as long as a letter or number is input. Upon accepting a whitespace character, the machine transitions to State A, its final accepting state, which we indicate with a double circle. If a character other than a number, letter, or whitespace is entered, the machine enters its error state, which is a final state that rejects the string.



Finite State Machine for Accepting a Variable Name

Of more interest to us (because we are discussing hardware) are Moore and Mealy FSMs that have output states. The basic difference between these FSMs and DFAs is that—in addition to the transition function moving us from state to state—Moore and Mealy machines also generate an output symbol. Furthermore, no set of final states is defined because circuits have no concept of halting or accepting strings; they instead generate output. Both the Moore and Mealy machines, M, can be completely described by the quintuple M = (Q, S, Σ, Γ, δ), where

- Q is a finite set of states that represents each configuration of the

machine;

- S is an element of Q that represents the Start state, the state of the machine before it has received any inputs;
- Σ is the input alphabet or set of events that the machine will recognize;
- Γ is the finite output alphabet; and
- δ is a transition function that maps a state from Q and a letter from the input alphabet to a state from Q.

We note that the input and output alphabets are usually identical, but they don't have to be. The way in which output is produced is the distinguishing element between the Moore and Mealy machines. Hence, the output function of the Moore machine is embedded in its definition of S, and the output function for the Mealy machine is embedded in the transition function, δ.

If any of this seems too abstract, just remember that a computer can be thought of as a universal finite-state machine. It takes the description of one machine plus its input and then produces output that is (usually) as expected. Finite-state machines are just a different way of thinking about the computer and computation.

3.7.5 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.

EXAMPLE 3.26

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 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 3.40 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 0s). 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.

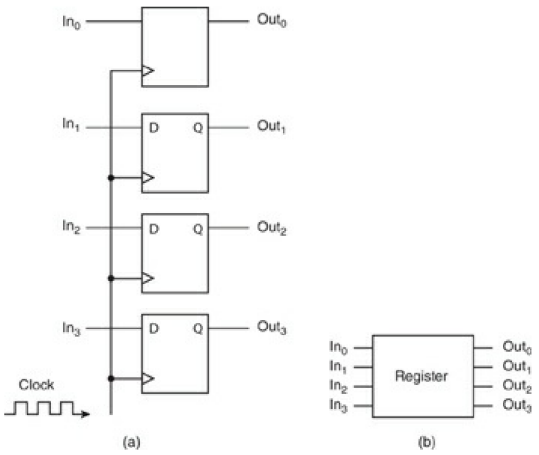


FIGURE 3.40 (a) 4-Bit Register
(b) Block Diagram for a 4-Bit Register

EXAMPLE 3.27

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 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 the count enable line is set to 1. If the count enable line is set to 0 and the clock pulses, the circuit does not change state. Examine Figure 3.41 very carefully, tracing the circuit with various inputs to make sure you understand how this circuit outputs the binary numbers from 0000 to 1111. Note: B_0, B_1, B_2, and B_3 are the outputs of this circuit, and they are always available

regardless of the values of the count enable and clock signals. Also check to see which state the circuit enters if the current state is 1111 and the clock is pulsed.

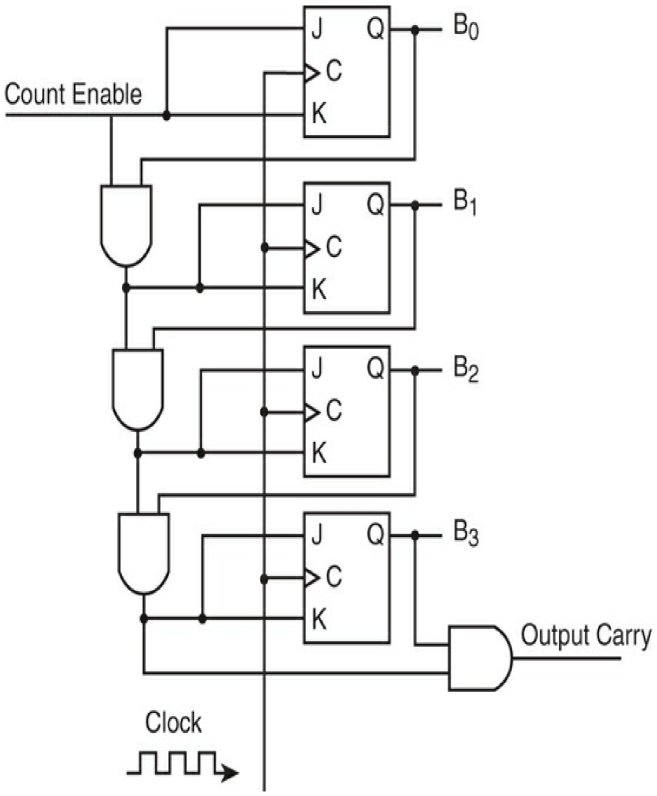


FIGURE 3.41 4-Bit Synchronous Counter Using JK Flip-Flops

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

EXAMPLE 3.28

The memory depicted in Figure 3.42 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 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_0,

Out_1, and Out_2) are used when reading words from memory.

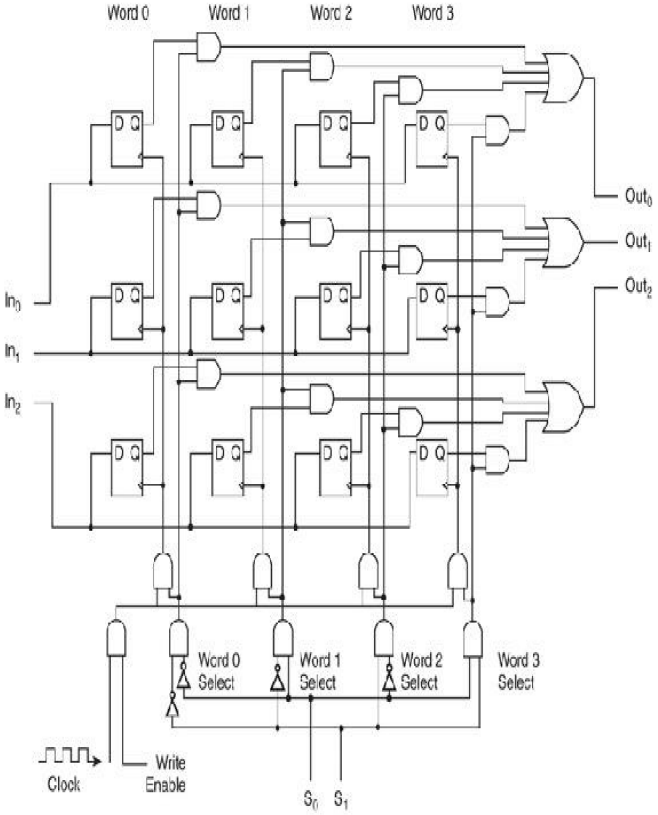


FIGURE 3.42 4 × 3 Memory

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.

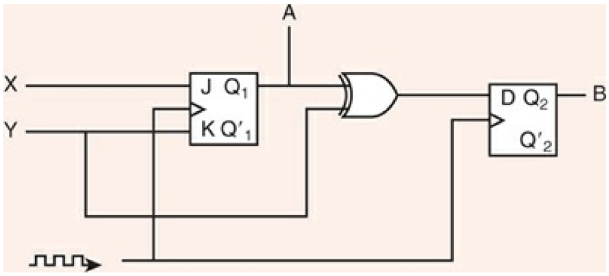
To summarize our discussion of this memory circuit, here are the steps necessary to write a word to memory:

1. An address is asserted on S_0 and S_1.
2. Write enable (WE) 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 combines with the clock and Write Enable 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 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.

NULL POINTERS: TIPS AND HINTS

Perhaps one of the most challenging concepts regarding sequential circuits is tracing them because the output, which can also be an input, changes, and it is often difficult to keep track of precisely where you are in the trace. Consider the following sequential circuit, for example:



Suppose you were asked to complete a truth table for this circuit (based on the inputs A, X, and Y, and the outputs A and B). You should follow these steps to analyze the circuit:

1. Determine the input equations for the flip-flops.
2. Determine the output equations for the outputs.
3. Find the next state equations for each flip-flop (using the input equations).
4. Use Kmaps to simplify the next state equations.
5. Construct the truth table.

The input equations for the flip-flops above are D = A XOR Q_1, J = X, and K = Y. The output equations are A = Q_1 and B = Q_2. The next state equations for the flip-flops are $Q_1^+ = JQ_1' + K'Q_1 = XQ_1' + Y'Q_1 = \boldsymbol{XA' + Y'A = A^+}$ and $Q_2^+ = D = \boldsymbol{A \text{ XOR } Y = B^+}$, where Q_1^+ (A^+) represents the next state for the JK flip-flip and Q_2^+ (B^+) represents the next state for the D flip-flop. We could

create Kmaps for the next states to minimize the expressions, but because the expressions are already simplified, we won't do that here. The truth table then becomes:

X	Y	A	Next State A^+	B^+
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

LOGICALLY SPEAKING, HOW'D THEY DO THAT?

In this chapter, we introduced logic gates. But exactly what goes on inside these gates to carry out the logic functions? How do these gates physically work? It's time to open the hood and take a peek at the internal composition of digital logic gates.

The implementation of the logic gates is accomplished using different types of logic devices belonging to different production technologies. These devices are often classified into logic families. Each family has its advantages and disadvantages, and each differs from the others in its capabilities and limitations. The logic families currently of interest include TTL, NMOS/PMOS, CMOS, and ECL.

TTL (transistor-transistor logic) replaces all the diodes originally found in integrated circuits with bipolar transistors. (See the sidebar on transistors in Chapter 1 for more information.) TTL defines binary values as follows: 0 to 0.8 V is logic 0, and 2_5 V is logic 1. Virtually any gate can be implemented using TTL. Not only does TTL offer the largest number of logic gates (from the standard combinational and sequential logic gates to memory), but this technology also offers superior speed of operation. The problem with these

relatively inexpensive integrated circuits is that they draw considerable power.

TTL was used in the first integrated circuits that were widely marketed. However, the most commonly used type of transistor used in integrated circuits today is called a MOSFET (metal-oxide semiconductor field effect transistor). Field effect transistors (FETs) are simply transistors whose output fields are controlled by a variable electric field. The phrase metal-oxide semiconductor is actually a reference to the process used to make the chip, and even though polysilicon is used today instead of metal, the name continues to be used.

NMOS (N-type metal-oxide semiconductors) and PMOS (P-type metal-oxide semiconductors) are the two basic types of MOS transistors. NMOS transistors are faster than PMOS transistors, but the real advantage of NMOS over PMOS is that of higher component density (more NMOS transistors can be put on a single chip). NMOS circuits have lower power consumption than their bipolar relatives. The main disadvantage of NMOS technology is its sensitivity to damage from electrical discharge. In addition, not as many gate implementations are available with NMOS as with TTL. Despite NMOS circuits using less power than TTL, increased NMOs circuit densities caused a resurgence in power consumption problems.

CMOS (complementary metal-oxide semiconductor) chips were designed as low-power alternatives to TTL and NMOS circuits, providing more TTL equivalents than NMOS in addition to addressing the power issues. Instead of using bipolar transistors, this technology uses a complementary pair of FETs, an NMOS and a PMOS FET (hence the name complementary). CMOS differs from NMOS because when the gate is in a static state, CMOS uses virtually no power. Only when the gate switches states does the circuit draw power. Lower power consumption translates to reduced heat dissipation.

For this reason, CMOS is extensively used in a wide variety of computer systems. In addition to low power consumption, CMOS chips operate within a wide range of supply voltages (typically from 3 to 15 V)—unlike TTL, which requires a power supply voltage of plus or minus 0.5 V. However, CMOS technology is extremely sensitive

to static electricity, so extreme care must be taken when handling circuits. Although CMOS technology provides a larger selection of gates than NMOS, it still does not match that of its bipolar relative, TTL.

ECL (emitter-coupled logic) gates are used in situations that require extremely high speeds. Whereas TTL and MOS use transistors as digital switches (the transistor is either saturated or cut off), ECL uses transistors to guide current through gates, resulting in transistors that are never completely turned off and never completely saturated. Because they are always in an active status, the transistors can change states very quickly. However, the trade-off for this high speed is substantial power requirements. Therefore, ECL is used only rarely, in very specialized applications.

A newcomer to the logic family scene, BiCMOS (bipolar CMOS) integrated circuits use both the bipolar and CMOS technologies. Despite the fact that BiCMOS logic consumes more power than TTL, it is considerably faster. Although not currently used in manufacturing, BiCMOS appears to have great potential.

3.7.6 An Application of Sequential Logic: Convolutional Coding and Viterbi Detection

Several coding methods are employed in data storage and communication. One of them is the partial response maximum likelihood (PRML) encoding method. This method allows conversion from an analog signal to a digital signal. The “maximum likelihood” component derives from the way that bits are encoded and decoded and is based on a method used in statistics for estimation based on observation. The relevant feature of the decoding process is that only certain bit patterns are valid. These patterns are produced using a convolutional code, an error-correcting code that uses parity bits in such a way that only the parity bits, not the parity bits plus the message, need to be transmitted. A Viterbi decoder reads the bits that have been output by a convolutional encoder and compares the symbol stream read with a set of “probable” symbol streams. The one with the least error is selected for output. We present this discussion because it brings together a number of concepts from this chapter as well as from Chapter 2. We begin with the encoding process.

The Hamming code introduced in Chapter 2 is a type of forward error correction that uses blocks of data (or block coding) to compute the necessary redundant bits. Some applications require a coding technique suitable for a continuous stream of data, such as that from a satellite television transmitter. Convolutional coding is a method that operates on an incoming serial bit stream, generating an encoded serial output stream (including redundant bits) that enables it to correct errors continuously. A convolutional code is an encoding process whereby the output is a function of the input and some number of bits previously received. Thus, the input is overlapped, or convoluted, over itself to form a stream of output symbols. In a sense, a convolutional code builds a context for accurate decoding of its output. Convolutional encoding combined with Viterbi decoding has become an accepted industry standard for encoding and decoding data stored or transmitted over imperfect (noisy) media.

The convolutional coding mechanism used in PRML is illustrated in Figure 3.43. Careful examination of this circuit reveals that two output bits are written for each input bit. The first output bit is a function of the input bit and the second previous input bit: A XOR C. The second bit is a function of the input bit and the two previous bits: A XOR C XOR B. The two AND gates at the right-hand side of the diagram alternatively select one of these functions during each pulse of the clock. The input is shifted through the D flip-flops on every second clock pulse. We note that the leftmost flip-flop serves only as a buffer for the input and isn't strictly necessary.

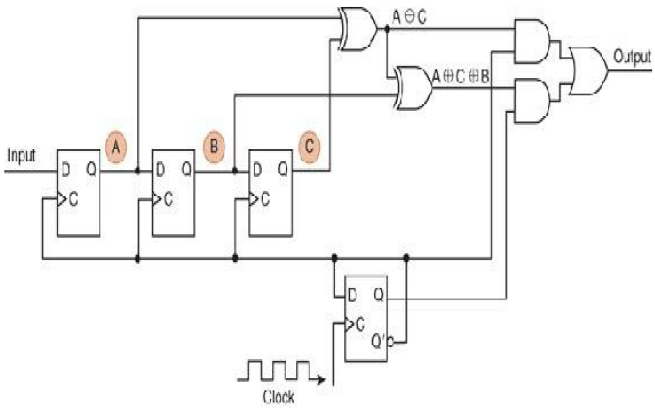


FIGURE 3.43 Convolutional Encoder for PRML

At first glance, it may not be easy to see how the encoder produces two output bits for every input bit. The trick has to do with the flip-flop situated between the clock and the other components of the circuit. When the complemented output of this flip-flop is fed back to its input, the flip-flop alternately stores 0s and 1s. Thus, the output goes high on every other clock cycle, enabling and disabling the correct AND gate with each cycle.

We step through a series of clock cycles in Figure 3.44. The initial state of the encoder is assumed to contain all 0s in the flip-flops labeled A, B, and C. A couple of clock cycles are required to move the first input into the A flip-flop (buffer), and the encoder outputs two 0s. Figure 3.44a shows the encoder with the first input (1) after it has passed to the output of flip-flop A. We see that the clock on flip-flops A, B, and C is enabled, as is the upper AND gate. Thus, the function A XOR C is routed to the output. At the next clock cycle (Figure 3.44b), the lower AND gate is enabled, which routes the function A XOR C XOR B to the output. However, because the clock on flip-flops A, B, and C is disabled, the input bit does not propagate from flip-flop A to flip-flop B. This prevents the next input bit from being consumed while the second output bit is written. At clock cycle 3 (Figure 3.44c), the input has propagated through flip-flop A, and the bit that was in flip-flop A has propagated to flip-flop B. The upper AND gate on the output is enabled and the function A XOR C is routed to the output.

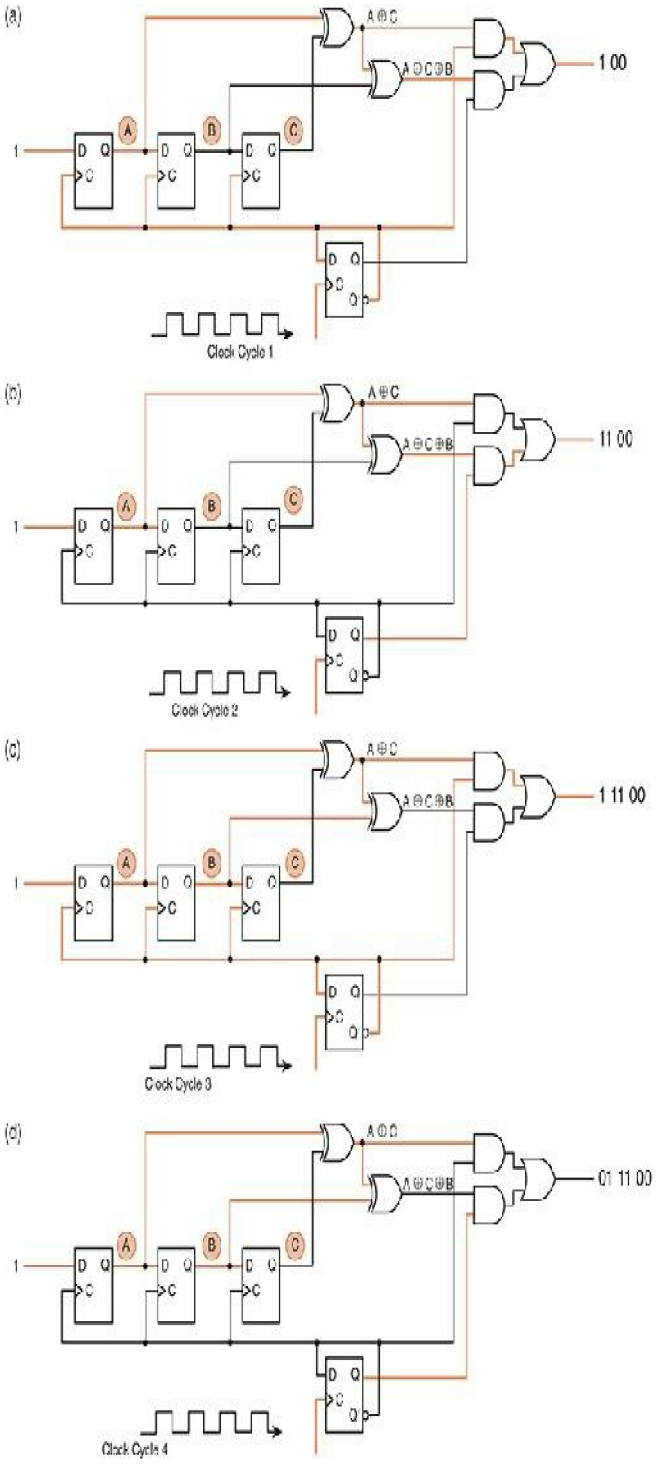


FIGURE 3.44 Stepping Through Four Clock Cycles of a Convolutional Encoder

The characteristic table for this circuit is given in Table 3.13. As an example, consider the stream of input bits, 11010010. The encoder initially contains all 0s, so B = 0 and C = 0. We say that the encoder is in State 0 (00_2).

When the leading 1 of the input stream exits the buffer, A, B = 0 and C = 0, giving (A XOR C XOR B) = 1 and (A XOR C) = 1. The output is 11 and the encoder transitions to State 2 (10$_2$). The next input bit is 1, and we have B = 1 and C = 0 (in State 2), giving (A XOR C XOR B) = 0 and (A XOR C) = 1. The output is 01 and the encoder transitions to State 1 (01$_2$). Following this process over the remaining 6 bits, the completed function is:

- F (1101 0010) = 11 01 01 00 10 11 11 10

TABLE 3.13 Characteristic Table for the Convolutional Encoder in Figure 3.43

Input A	Current State B C	Next State B C	Output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	10
1	10	11	01
0	11	01	01
1	11	11	10

The encoding process is made a little clearer using the Mealy machine (Figure 3.45). This diagram informs us at a glance as to which transitions are possible and which are not. You can see the correspondence between the Figure 3.45 machine and the characteristic table by reading the table and tracing the arcs, or vice versa. The fact that there is a limited set of allowable transitions is crucial to the error-correcting properties of this code and to the operation of the Viterbi decoder, which is responsible for decoding the stream of bits correctly. By reversing the inputs with the outputs on the transition arcs, as shown in Figure 3.46, we place bounds around the set of possible decoding inputs.

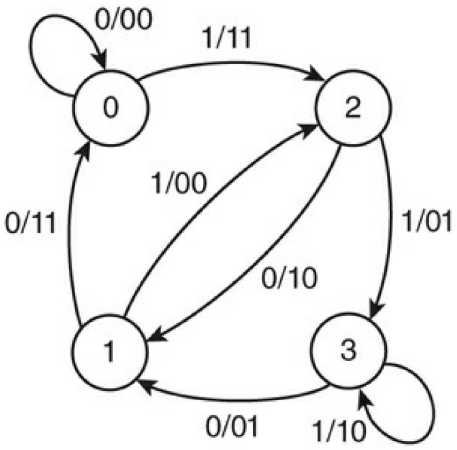


FIGURE 3.45 Mealy Machine for the Convolutional Encoder in Figure 3.43

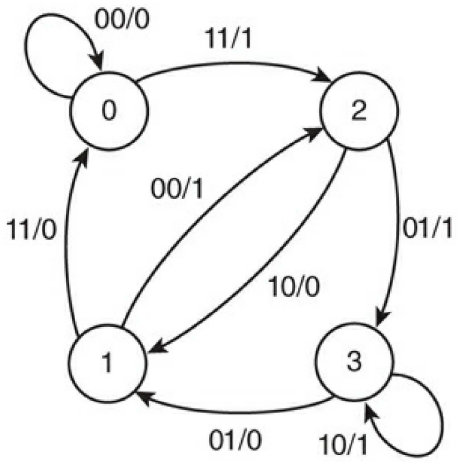


FIGURE 3.46 Mealy Machine for a Convolutional Decoder

For example, suppose the decoder is in State 1 and sees the pattern 00 01. The decoded bit values returned are 11, and the decoder ends up in State 3. (The path traversed is 1 → 2 → 3.) If, on the other hand, the decoder is in State 2 and sees the pattern 00 11, an error has occurred because there is no outbound transition on State 2 for 00. The outbound transitions on State 2 are 01 and 10. Both of these have a Hamming distance of 1 from 00. If we follow both (equally likely) paths out of State 2, the decoder ends up in either State 1 or State 3. We see that there is no outbound transition on State 3 for the next pair of bits, 11. Each outbound transition from State 3 has a Hamming distance of 1 from 11. This gives an accumulated Hamming distance of 2 for both paths: 2 → 3 → 1 and 2 → 3 → 2. However, State 1 has a

valid transition on 11. By taking the path 2 → 1 → 0, the accumulated error is only 1, so this is the most likely sequence. The input therefore decodes to 00 with maximum likelihood.

An equivalent (and probably clearer) way of expressing this idea is through the trellis diagram, shown in Figure 3.46. The four states are indicated on the left side of the diagram. The transition (or time) component reads from left to right. Every code word in a convolutional code is associated with a unique path in the trellis diagram. A Viterbi detector uses the logical equivalent of paths through this diagram to determine the most likely bit pattern. In Figure 3.47, we show the state transitions that occur when the input sequence 00 10 11 11 is encountered with the decoder starting in State 1. You can compare the transitions in the trellis diagram with the transitions in the Mealy diagram in Figure 3.46.

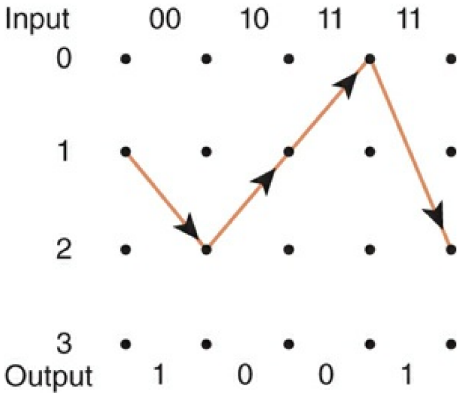


FIGURE 3.47 Trellis Diagram Illustrating State Transitions for the Sequence 00 10 11 11

Suppose we introduce an error in the first pair of bits in our input, giving the erroneous string 10 10 11 11. With our decoder starting in State 1 as before, Figure 3.48 traces the possible paths through the trellis. The accumulated Hamming distance is shown on each of the transition arcs. The correct path that correctly assumes that the string should be 00 10 11 11 is the one having the smallest accumulated error, so it is accepted as the correct sequence.

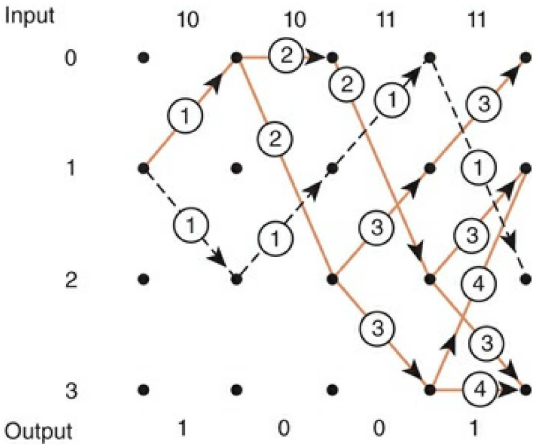


FIGURE 3.48 Trellis Diagram Illustrating Hamming Errors for the Sequence 10 10 11 11

In most cases where it is applied, the Viterbi decoder provides only one level of error correction. Additional error-correction mechanisms such as cyclic redundancy checking and Reed-Solomon coding (discussed in Chapter 2) are applied after the Viterbi algorithm has done what it can to produce a clean stream of symbols. All these algorithms are usually implemented in hardware for utmost speed using the digital building blocks described in this chapter.

We hope that our discussion in this section has helped you to see how digital logic and error-correction algorithms fit together. The same can be done with any algorithm that can be represented using one of the finite-state machines described. In fact, the convolutional code just described is also referred to as a (2, 1) convolutional code because two symbols are output for every one symbol input. Other convolutional codes provide somewhat deeper error correction, but they are too complex for economical hardware implementation.

3.8 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 computer-aided design (CAD) 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 finding 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.

There are many tools available to help with the design of digital logic circuits. Among the most popular are hardware description languages (HDLs). These are computer languages that allow for a top-down methodology resulting in a precise and formal description of digital circuits. HDLs not only increase productivity and flexibility, but also allow for quick changes and design reuse. In addition, the designer can use an HDL to simulate the circuit before actually building it. Unlike regular programs that describe an algorithm, HDLs describe a hardware structure. Code is typically written at the register transfer language (RTL) level (see Chapter 4 for more info on RTLs).

Two very popular HDLs are VHDL and Verilog. VHDL (VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuit) was developed in the early 1980s. Every VHDL program

contains at least one entity/architecture pair; these pairs are typically connected to form complete circuits. In addition, each program should contain an architecture declaration describing the actual function of the entity. The fundamental unit in VHDL is a signal. As an example, consider the following VHDL code to specify the behavior of an AND gate:

```
signal and_gate : std_logic;
and_gate <= input1 and input2;
entity andGate is
  port (
       input1   : in std_logic;
       input2   : in std_logic
       res_out  : out std_logic
  )
architecture func of andGate is
  signal and_gate : std_logic;
begin
  res_out <= input1 and input2;
end func;
```

This code first defines a signal and_gate of type std_logic that is assigned two inputs. The inputs are defined in the entity andGate, which contains three signals: input1, input2, and res_out. The architecture func describes the functionality of the entity andGate.

Verilog performs a similar task, but has a completely different structure from VHDL. Verilog allows you to design in a more modular fashion, allowing the programmer to use system blocks called modules. Each module specifies the input signals, the output signals, and the internal structure and behavior of the circuit being designed. The following is Verilog code to define a half-adder:

```
module hlfaddr (x,y,sum,car):
  input x;
  input y;
  output sum;
  output car;
  assign sum=x^y;
  assign car=x&y;
endmodule;
```

The Verilog ^ operator performs an exclusive OR, while the & operator performs an AND. If you compare this to the half-adder in Figure 3.20, you can see how the code corresponds to the actual circuit.

Each language has its advantages and disadvantages, but both can help with design and verification because they represent an easy way to abstract the details of electronic circuits.

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 (e.g., 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 view, the two most important factors are speed and cost; minimizing 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.

You have seen that sequential circuits can be represented in a number of different ways, depending on the particular behavior that we want to emphasize. Clear pictures can be rendered by Moore, Mealy, and algorithmic state machines. A lattice diagram expresses transitions as a function of time. These finite state machines differ from DFAs in that, unlike DFAs, they have no final state because circuits produce output rather than accept strings.

These logic circuits are the building blocks necessary for computer systems. In Chapter 4, we put these blocks together and take a closer, more detailed look at how a computer actually functions.

FURTHER READING

Most computer organization and architecture books have a brief discussion of digital logic and Boolean algebra. The books by Stallings (2013), Tanenbaum (2012), and Patterson and Hennessy (2011) contain good synopses of digital logic. Mano (1993) presents a good discussion on using Kmaps for circuit simplification 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.

Davis (2000) traces the history of computer theory, including biographies of all the seminal thinkers, in Universal Computer. This book is a joy to read. 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 straightforward and easy-to-read book on gates and flip-flops (as well as a terrific explanation of what computers are and how they work), see Petzold (1989). Davidson (1979) presents a method of decomposing NAND-based circuits (of interest because NAND is a universal gate).

Moore, Mealy, and algorithmic state machines were first proposed in papers by Moore (1956), Mealy (1955), and Clare (1973). Cohen's (1991) book on computer theory is one of the most easily understandable on this topic. In it you will find excellent presentations of Moore, Mealy, and finite state machines in general, including DFAs. Forney's (1973) well-written tutorial on the Viterbi algorithm in a paper by that same name explains the concept and the mathematics behind this con-volutional decoder. Fisher's (1996) article explains how PRML is used in disk drives.

If you are interested in actually designing some circuits, there are several nice simulators freely available. One 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).

Another nice package is Multimedia Logic (MMLogic) by Softronix, but it is currently available for Windows platforms only. This fully functional package has a nice GUI with drag-and-drop components and comprehensive online help. It includes not only the standard complement of devices (such as ANDs, ORs, NANDs, NORs, adders, and counters), but also special multimedia devices (including bitmap, robot, network, and buzzer devices). You can create logic circuits and interface them to real devices (keyboards, screens, serial ports, etc.) or other computers. The package is advertised for use by beginners but allows users to build quite complex applications (such as games that run over the internet). MMLogic can be found at www.softronix.com/logic.html, and the distribution includes not only the executable package but also the source code, so users can modify or extend its capabilities.

A third digital logic simulator is Logisim, an open-source software package available at http://ozark.hendrix.edu/~burch/logisim/. This software is compact, easy to install, and easy to use, and it requires only that Java 5 or later be installed; therefore, it is available for Windows, Mac, and Linux platforms. The interface is intuitive, and unlike most simulators, Logisim allows the user to modify a circuit during simulation. The application allows the user to build larger circuits from smaller 1s, draw bundles of wires (with multi-bit width) in one mouse action, and use a tree view to see the library of components that can be utilized for building circuits. Like MMLogic, the package was designed as an educational tool to help beginners experiment with digital logic circuits, but it also allows the user to build fairly complex circuits.

Any of these simulators can be used to build the MARIE architecture discussed next in Chapter 4.

REFERENCES

1. Clare, C. R. Designing Logic Systems Using State Machines. New York: McGraw-Hill, 1973.
2. Cohen, D. I. A. Introduction to Computer Theory, 2nd ed. New York: John Wiley & Sons, 1991.
3. Davidson, E. S. “An Algorithm for NAND Decomposition under Network Constraints.” IEEE Transactions on Computing C-18, 1979, p. 1098.
4. Davis, M. The Universal Computer: The Road from Leibniz to Turing. New York: W. W. Norton, 2000.
5. Fisher, K. D., Abbott, W. L., Sonntag, J. L., & Nesin, R. “PRML Detection Boosts Hard-Disk Drive Capacity.” IEEE Spectrum, November 1996, pp. 70-76.
6. Forney, G. D. “The Viterbi Algorithm.” Proceedings of the IEEE 61, March 1973, pp. 268-278.
7. Gregg, J. Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets. New York: IEEE Press, 1998.
8. Hayes, J. P. Digital Logic Design. Reading, MA: Addison-Wesley, 1993.
9. Katz, R. H. Contemporary Logic Design. Redwood City, CA: Benjamin Cummings, 1994.
10. Mano, M. M. Computer System Architecture, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
11. Maxfield, C. Bebop to the Boolean Boogie. Solana Beach, CA: High Text Publications, 1995.
12. Mealy, G. H. “A Method for Synthesizing Sequential Circuits.” Bell System Technical Journal 34, September 1955, pp. 1045-1079.
13. Moore, E. F. “Gedanken Experiments on Sequential Machines,” in Automata Studies, edited by C. E. Shannon and John McCarthy. Princeton, NJ: Princeton University Press, 1956, pp. 129-153.
14. Patterson, D. A., & Hennessy, J. L. Computer Organization and Design, The Hardware/Software Interface, 4th ed. San Mateo, CA: Morgan Kaufmann, 2011.
15. Petzold, C. Code: The Hidden Language of Computer Hardware and Software. Redmond, WA: Microsoft Press, 1989.
16. Stallings, W. Computer Organization and Architecture, 9th ed. Upper Saddle River, MJ: Prentice Hall, 2013.
17. Tanenbaum, A. Structured Computer Organization, 6th ed. Upper Saddle River, NJ: Prentice Hall, 2012.
18. 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. What is the difference between a gate and a circuit?
9. Name the four basic logic gates.
10. What are the two universal gates described in this chapter? Why are these universal gates important?
11. Describe the basic construction of a digital logic chip.
12. Describe the operation of a ripple-carry adder. Why are ripple-carry adders not used in most computers today?
13. What are the three methods we can use to express the logical behavior of Boolean functions?
14. What are the necessary steps one must take when designing a logic circuit from a description of the problem?
15. What is the difference between a half-adder and a full adder?
16. 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.
17. What kind of circuit selects binary information from one of many input lines and directs it to a single output line?
18. How are sequential circuits different from combinational circuits?
19. What is the basic element of a sequential circuit?
20. What do we mean when we say that a sequential circuit is edge triggered rather than level triggered?
21. In the context of digital circuits, what is feedback?
22. How is a JK flip-flop related to an SR flip-flop?
23. Why are JK flip-flops often preferred to SR flip-flops?
24. Which flip-flop gives a true representation of computer memory?
25. How is a Mealy machine different from a Moore machine?
26. What does an algorithmic state machine offer that is not provided by either a Moore or a Mealy machine?

EXERCISES

1. 1. Construct a truth table for the following:
    1. ◆a) yz + z(xy)′
    2. ◆b) x(y′ + z) + xyz
    3. c) (x + y)(x′ + y) (Hint: This is from Example 3.7.)
2. 2. Construct a truth table for the following:
    1. a) xyz + x(yz)′ + x′(y + z) + (xyz)′
    2. b) (x + y′)(x′ + z′)(y′ + z′)
3. ◆3. Using DeMorgan's Law, write an expression for the complement of F if F(x, y, z) = xy′(x + z).
4. 4. Using DeMorgan's Law, write an expression for the complement of F if F (x, y, z) = (x′ + y)(x + z)(y′ + z)′.
5. ◆5. Using DeMorgan's Law, write an expression for the complement of F if F (w, x, y, z) = xz′(x′yz + x) + y(w′z + x′).
6. 6. Using DeMorgan's Law, write an expression for the complement of F if F (x, y, z) = xz′(xy + xz) + xy′(wz + y).
7. 7. Prove that DeMorgan's Laws are valid.
8. ◆8. Is the following distributive law valid or invalid? Prove your answer.

    x XOR (y + z) = (x XOR y) + (x XOR z)
9. 9. Is the following true or false? Prove your answer.

    (x XOR y)′ = xy + (x + y)′
10. 10. Show that x = xy + xy′
    1. a) Using truth tables
    2. b) Using Boolean identities
11. 11. Use only the first seven Boolean identities to prove the Absorption Laws.
12. 12. Show that xz = (x + y)(x + y′)(x′ + z)
    1. a) Using truth tables
    2. b) Using Boolean identities
13. 13. Use any method to prove the following either true or false.

    xz + x′y′ + y′z′ = xz + y′
14. 14. Simplify the following functional expressions using Boolean algebra and its identities. List the identity used at each step.
    1. a) F(x, y, z) = y (x′ + (x + y)′)
    2. b) F (x, y, z) = x ′yz + xz

3. c) F (x, y, z) = (x′ + y + z)′ + xy′z′ + yz + xyz

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

    1. a) x(yz + y′z) + xy + x′y + xz
    2. b) xyz′′ + (y + z)′ + x′yz
    3. c) z(xy′ + z)(x + y′)

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

    1. a) z (w + x)′ + w′xz + wxyz′ + wx′yz′
    2. b) y′(x′z′ + xz) + z (x + y)′
    3. c) x(yz′ + x)(y′ + z)

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

    1. ◆a) x(y + z)(x′ + z′)
    2. b) xy + xyz + xy′z + x′y′z
    3. c) xy′z + x(y + z′)′ + xy′z′

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

    1. a) y(xz′ + x′z) + y′(xz′ + x′z)
    2. b) x(y′z + y) + x′(y + z′)′
    3. c) x[y′z + (y + z′)′](x′y + z)

19. ◆19. Using the basic identities of Boolean algebra, show that

    x(x′ + y) = xy

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

    x + x′y = x + y

21. 21. Using the basic identities of Boolean algebra, show that

    xy + x ′z + yz = xy + x ′z

22. ◆22. 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	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1

1	1	1	1

23. 23. 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	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

24. 24. Which of the following Boolean expressions is not logically equivalent to all the rest?

1. a) wx′ + wy′ + wz
2. b) w + x′ + y′ + z
3. c) w(x′ + y′ + z)
4. c) wx′yz′ + wx′y′ + wy′z′ + wz

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

xy′ + x′y + xz + y′z

26. 26. Given the Boolean function, F(x, y, z) = x′y + xyz′

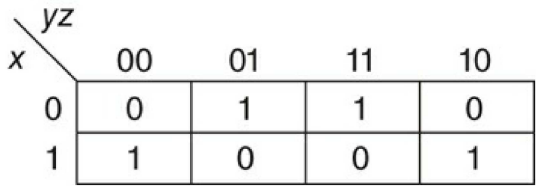
1. a) Derive an algebraic expression for the complement of F. Express in sum-of-products form.
2. b) Show that FF′. = 0.
3. c) Show that F + F′ = 1.

27. 27. Given the function, F (x, y, z) = y(x′z + xz′) + x(yz + yz′)

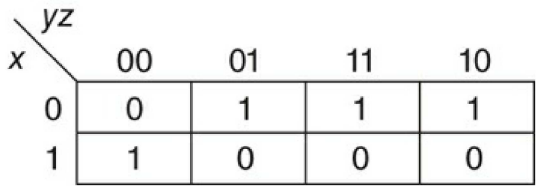
1. a) List the truth table for F.
2. b) Draw the logic diagram using the original Boolean expression.
3. c) Simplify the expression using Boolean algebra and identities.
4. d) List the truth table for your answer in Part c.
5. e) Draw the logic diagram for the simplified expression in Part c.

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

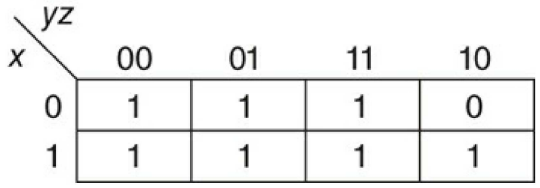
1. ◆a)



2. ◆b)

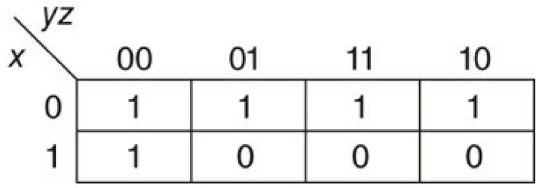


3. c)

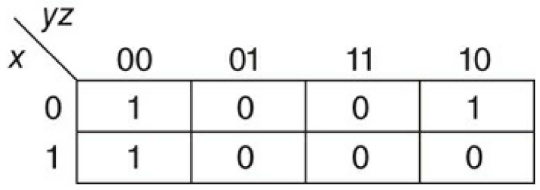


29. 29. Write a simplified expression for the Boolean function defined by each of the following Kmaps:

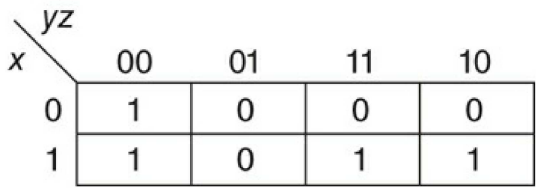
1. a)



2. b)



3. c)



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

1. a) F(x, y, z) = x′y′z′ + x′yz + x′yz′
2. b) F(x, y, z) = x′y′z′ + x′yz′ + xy′z′ + xyz′
3. c) F(x, y, z) = y′z′ + y′z + xyz′

31. 31. Write a simplified expression for the Boolean function defined

by each of the following Kmaps:

1. ◆a)

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

2. ◆b)

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

3. c)

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

32. 32. Write a simplified expression for the Boolean function defined by each of the following Kmaps (leave in sum-of-products form):

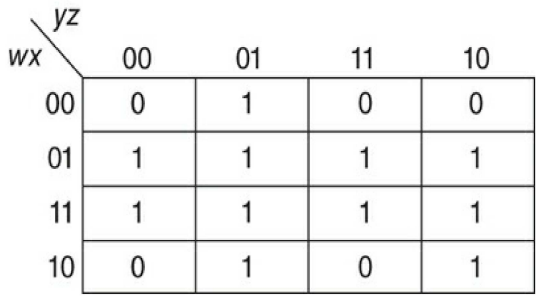
1. a)

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

2. b)

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

3. c)



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

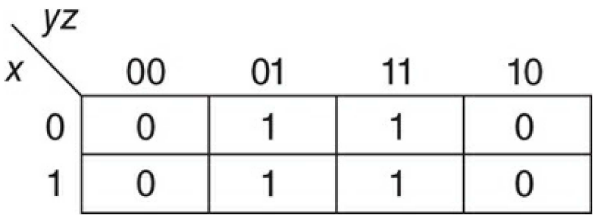
33. 33. Create the Kmaps and then simplify for the following functions (leave in sum-of-products form):

1. ◆a) F(w, x, y, z) = w′x′y′z′ + w′x′yz′ + w′xy′z + w′xyz + w′xyz′ + wx′y′z′ + wx′yz′
2. ◆b) F(w, x, y, z) = w′x′y′z′ + w′x′y′z + wx′y′z + wx′yz′ + wx′y′z′
3. c) F(w, x, y, z) = y′z + wy′ + w′xy + w′x′yz′ + wx′yz′

34. 34. Create the Kmaps and then simplify for the following functions (leave in sum-of-products form):

1. a) F(w, x, y, z) = w′x′y′z + w′x′yz′ + w′xy′z + w′xyz + w′xyz′ + wxy′z + wxyz + wx′y′z
2. b) F(w, x, y, z) = w′x′y′z′ + w′z + w′x′yz′ + w′xy′z′ + wx′y
3. c) F(w, x, y, z) = w′x′y′ + w′xz + wxz + wx′y′z′

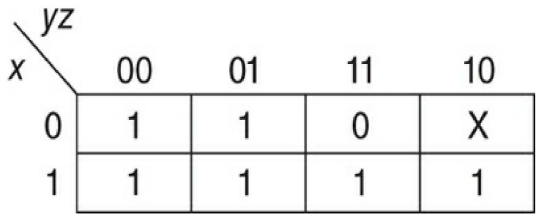
35. ◆35. Given the following Kmap, show algebraically (using Boolean identities) how the four terms reduce to one term.



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

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

1. a)



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

2. b)

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

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

1. a)

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

2. b)

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

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

1. a)

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

2. b)

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

39. 39. Find the minimized Boolean expression for the functions defined by each of the following truth tables:

1. a)

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

2. b)

w	x	y	z	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	X
0	1	0	1	0
0	1	1	0	X
0	1	1	1	0
1	0	0	0	1
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	1
1	1	1	0	X
1	1	1	1	X

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

41. 41. Construct the XOR operator using only NAND gates. Hint:x XOR y = ((x′y)′(xy′)′)′

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

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

44. 44. 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, 6, or 7, the binary output will be the complement of the input. When the binary input is 2, 3, 4, or 5, the output is the input shifted left with rotate. (For example, $3 = 011_2$ outputs 110; $4 = 100_2$ outputs 001.) Show your truth table, all computations for simplification, and the final circuit.

45. ◆45. Draw the combinational circuit that directly implements the Boolean expression:

F(x, y, z) = xyz + (y′ + z)

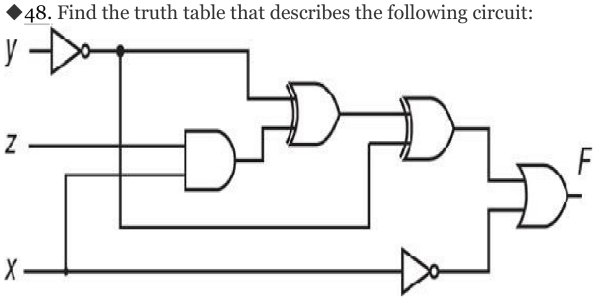
46. 46. Draw the combinational circuit that directly implements the following Boolean expression:

F(x, y, z) = x + xy + y′z

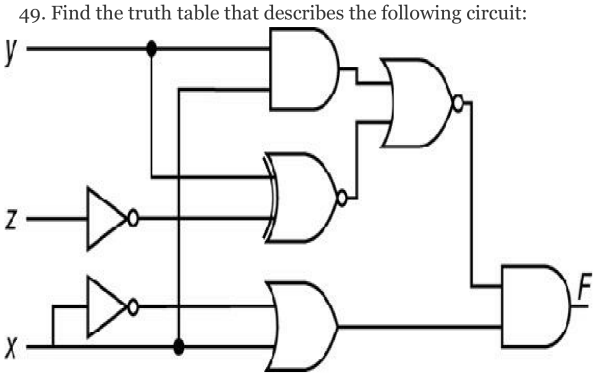
47. 47. Draw the combinational circuit that directly implements the Boolean expression:

F(x, y, z) = (x(yXORz)) + (xz)′

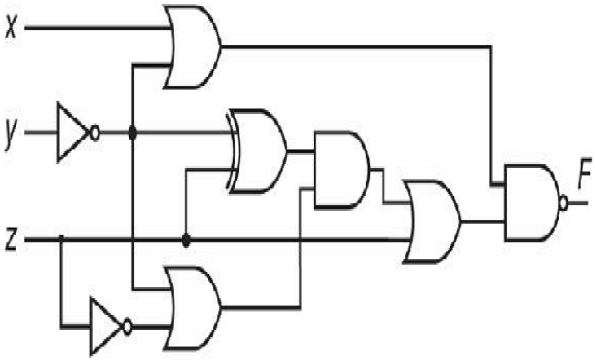
48. ◆48. Find the truth table that describes the following circuit:



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



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



51. 51. How many inputs does a decoder have if it has 64 outputs?

52. 52. How many control lines does a multiplexer have if it has 32 inputs?

53. 53. Draw circuits to implement the parity generator and parity checker shown in Tables 3.10 and 3.11, respectively.

54. 54. Assume you have the following truth tables for functions $F_1(x, y, z)$ and $F_2(x, y, z)$:

x	y	z	F_1	F_2
0	0	0	1	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

    1. a) Express F_1 and F_2 in sum-of-products form.
    2. b) Simplify each function.
    3. c) Draw one logic circuit to implement the above two functions.

55. 55. Assume you have the following truth tables for functions $F_1(w, x, y, z)$ and $F_2(w, x, y, z)$:

w	x	y	z	F_1	F_2
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

    1. a) Express F_1 and F_2 in sum-of-products form.
    2. b) Simplify each function.

3. c) Draw one logic circuit to implement the above two functions.

56. 56. Design a truth table for a combinational circuit that detects an error in the representation of a decimal digit encoded in BCD. (This circuit should output a 1 when the input is one of the six unused combinations for BCD code.)

57. 57. Simplify the function from exercise 56 and draw the logic circuit.

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

    1. a) Decoder
    2. b) Multiplexer

59. 59. Little Susie is trying to train her new puppy. She is trying to figure out when the puppy should get a dog biscuit as a reward. She has concluded the following:

    1. 1. Give the puppy a biscuit if it sits and wiggles but does not bark.
    2. 2. Give the puppy a biscuit if it barks and wiggles but does not sit.
    3. 3. Give the puppy a biscuit if it sits but does not wiggle or bark.
    4. 4. Give the puppy a biscuit if it sits, wiggles, and barks.
    5. 5. Don’t give the puppy a treat otherwise.

        1. Use the following:
        2. S: Sit (0 for not sitting; 1 for sitting)
        3. W: Wiggles (0 for not wiggling; 1 for wiggling)
        4. B: Barking (0 for not barking; 1 for barking)
        5. F: Biscuit function (0, don’t give the puppy a biscuit; 1, give the puppy a biscuit)

    Construct a truth table and find the minimized Boolean function to implement the logic telling Susie when to give her dog a biscuit.

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

    1. a) Buy if Pain and Meg both say Yes and the psychic says No.
    2. b) Buy if the psychic says Yes.
    3. c) Don’t buy otherwise.

    Construct a truth table and find the minimized Boolean function to

implement the logic telling Tyrone when to buy.

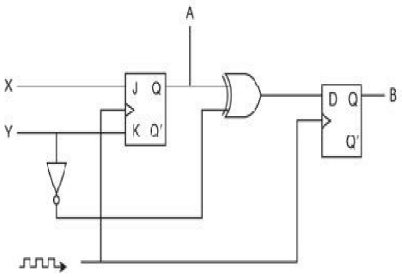
61. ◆*61. A very small company has hired you to install a security system. The brand of system that you install is priced by the number of bits encoded on the proximity cards that allow access to certain locations in a facility. Of course, this small company wants to use the fewest bits possible (spending the least amount of money possible) yet have all of its security needs met. The first thing you need to do is to 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:

    1. a) The Big Boss needs access to the executive lounge and the executive washroom.
    2. b) The Big Boss's secretary needs access to the supply closet, employee lounge, and executive lounge.
    3. c) Computer room employees need access to the server room and the employee lounge.
    4. d) The janitor needs access to all areas in the workplace.

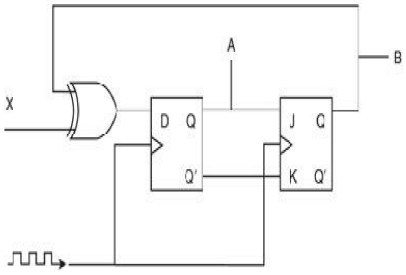
    Determine how each class of employee will be encoded on the cards and construct logic diagrams for the card readers in each of the five restricted areas.

62. ◆62. Complete the truth table for the following sequential circuit:



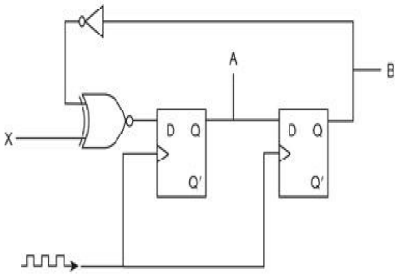
			Next State	
X	Y	A	A	B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

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



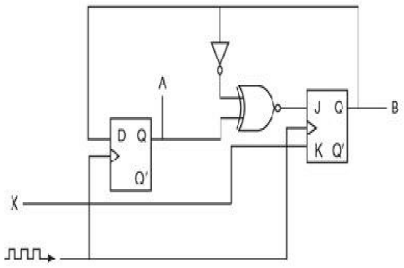
			Next State	
A	B	X	A	B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

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



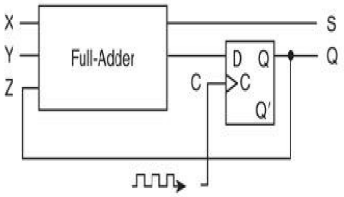
			Next State	
A	B	X	A	B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

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



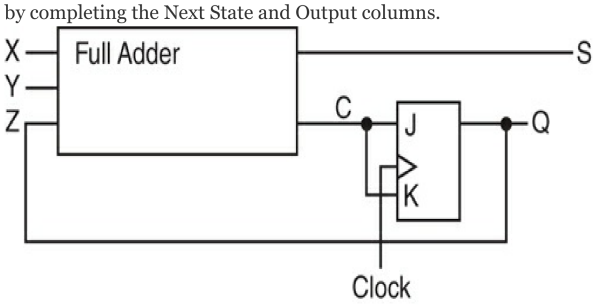
			Next State	
A	B	X	A	B
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

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



			Next State	
X	Y	Z	S	Q
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

67. 67. 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 JK flip-flop, as shown. Fill in the truth table for this sequential circuit by completing the Next State and Output columns.

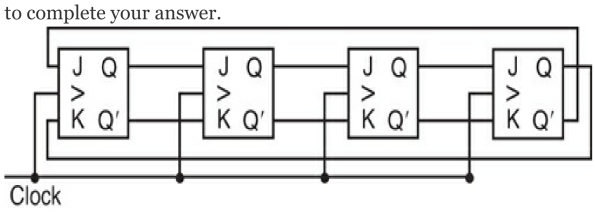


Present State Q(t)	Inputs X	Y	Next State Q(t+1)	Output S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

68. 68. True or false: When a JK flip-flop is constructed from an SR flip-flop, S = JQ′ and R = KQ.

69. ◆*69. 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.



70. 70. A Null-Lobur flip-flop (NL flip-flop) behaves as follows: If N = 0, the flip-flop does not change state. If N = 1, the next state of the flip-flop is equal to the value of L.

    1. a) Derive the characteristic table for the NL flip-flop.
    2. b) Show how an SR flip-flop can be converted to an NL flip-flop by adding gate(s) and inverter(s). (Hint: What values must S and R have so that the flip-flop will be set and reset at the proper time when N = 1? How can you prevent the flip-flop from changing state when N = 0?)

71. *71. 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.

    1. a) Derive the characteristic table for the flip-flop.
    2. b) Show how a JK flip-flop can be converted to an MN flip-flop by adding gate(s) and and inverter(s).

72. 72. List the steps necessary to read a word from memory in the 4 × 3 memory circuit shown in Figure 3.41.

73. 73. Construct Moore and Mealy machines that complement their input.

74. 74. Construct a Moore machine that counts modulo 5.

75. 75. Construct two parity checkers using a Moore machine for one and a Mealy machine for the other.

76. 76. Using the lemma that two FSMs are equivalent if and only if they produce the same output from the same input strings, show that Moore and Mealy machines are equivalent.

77. 77. Using the convolutional code and Viterbi algorithm described in this chapter, assuming that the encoder and decoder always start in State 0, determine the following:

    1. a) The output string generated for the input: 10010110.
    2. b) In which state is the encoder after the sequence in Part a is read?
    3. c) Which bit is in error in the string, 11 01 10 11 11 11 10? What is the probable value of the string?

78. 78. Repeat question 65 to determine the following:

    1. a) The output string generated for the input: 00101101.
    2. b) In which state is the encoder after the sequence in Part a is written?

3. c) Which bit is in error in the string, 00 01 10 11 00 11 00? What is the probable value of the string?

79. 79. Repeat question 65 to determine the following:

    1. a) The output string generated for the input: 10101010.
    2. b) In which state is the encoder after the sequence in Part a is written?
    3. c) Which bit is in error in the string, 11 10 01 00 00 11 01? What is the probable value of the string?

80. 80. Repeat question 65 to determine the following:

    1. a) The output string generated for the input: 01000111.
    2. b) In which state is the encoder after the sequence in Part a is written?
    3. c) Which bit is in error in the string, 11 01 10 11 01 00 01? What is the probable value of the string?

Null Pointer image: © iStockphoto/Getty Images