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

TABLE 3.7 Truth Table Representation for a Function and Its Complement

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

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

You can verify that this simple rule of thumb for finding the complement of a Boolean expression is correct by examining the truth tables for both the expression and its complement. The complement of any expression, when represented as a truth table, should have 0s for output everywhere the original function has 1s, and 1s in those places where the original function has 0s. Table 3.7 depicts the truth tables for $F(x, y, z) = \bar{x} + y\bar{z}$ and its complement, $\overline{F}(x, y, z) = x(\bar{y} + z)$. The shaded portions indicate the final results for F and $\overline{F}$.

3.2.5 Representing Boolean Functions

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

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

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

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

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

Any Boolean expression can be represented in sum-of-products form. Because any Boolean expression can also be represented as a truth table, we conclude that any truth table can also be represented in sum-of-products form. It is a simple matter to convert a truth table into sum-of-products form (see Example 3.5).

EXAMPLE 3.5 Consider a simple majority function. This is a function that, when given three inputs, outputs a 0 if less than half of its inputs are 1, and a 1 if at least half of its inputs are 1. Table 3.8 depicts the truth table for this majority function over three variables.

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

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

TABLE 3.8 Truth Table Representation for the Majority Function

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

3.3 LOGIC GATES

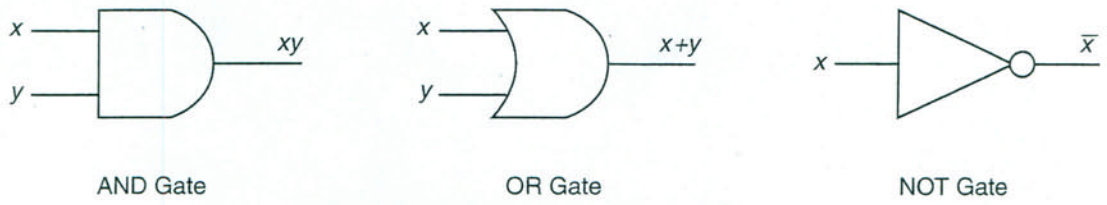
The logical operators AND, OR, and NOT that we have discussed have been represented thus far in an abstract sense using truth tables and Boolean expressions. The actual physical components, or **digital circuits**, such as those that perform arithmetic operations or make choices in a computer, are constructed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

3.3.1 Symbols for Logic Gates

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

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



AND Gate OR Gate NOT Gate

FIGURE 3.1 The Three Basic Gates

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

true otherwise. Figure 3.2 illustrates the truth table for XOR as well as the logic diagram that specifies its behavior.

3.3.2 Universal Gates

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

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.

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

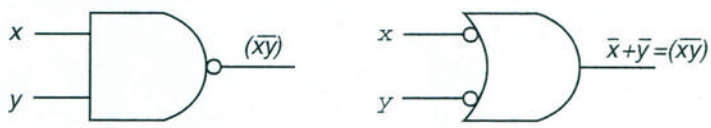


FIGURE 3.3 Truth Table and Logic Symbols for NAND

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

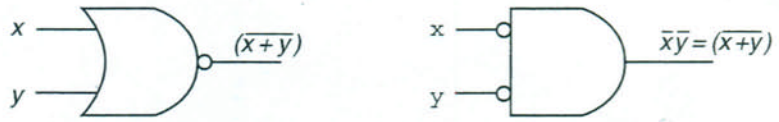
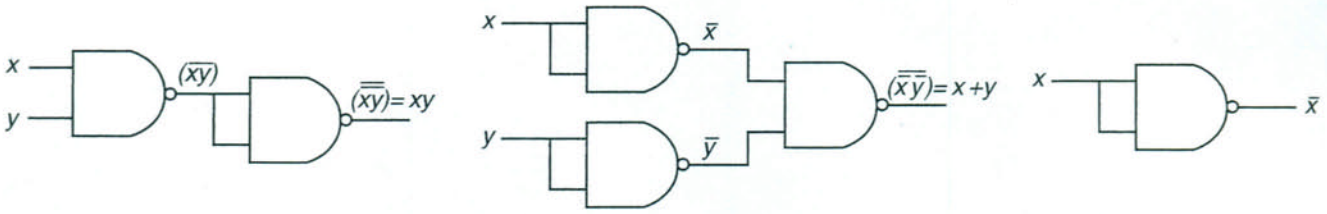


FIGURE 3.4 Truth Table and Logic Symbols for NOR



AND Gate OR Gate NOT Gate

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 $x\bar{y}z$.

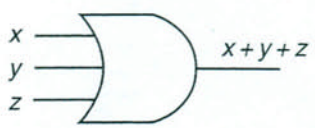


FIGURE 3.6 A Three-Input OR Gate Representing $x + y + z$

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

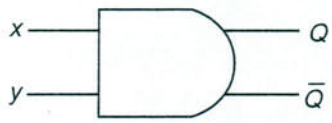


FIGURE 3.8 AND Gate with Two Inputs and Two Outputs

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

Note that Q always represents the actual output.

3.4 DIGITAL COMPONENTS

Upon opening a computer and looking inside, one would realize that there is a lot to know about all of the digital components that make up the system. Every computer is built using collections of gates that are all connected by way of wires acting as signal 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.

3.4.1 Digital Circuits and Their Relationship to Boolean Algebra

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

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

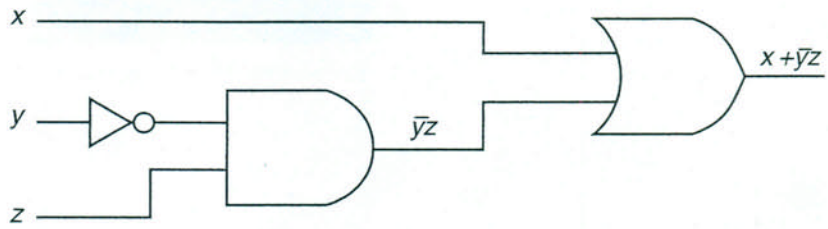


FIGURE 3.9 Logic Diagram for $F(x, y, z) = x + \bar{y}z$

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:

$A = A \text{ XOR } B$

$B = A \text{ XOR } B$

$A = A \text{ XOR } B$

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 04h. 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 impact on the complexity of the resulting digital circuit; the more complex the expression, the more complex the resulting circuit. We should point out that we do not typically simplify our circuits using Boolean identities; we have already seen that this can sometimes be quite difficult and time consuming. Instead, designers use a more automated method to do this. This method involves the use of **Karnaugh maps** (or **Kmaps**). Refer to the focus section following this chapter to learn how Kmaps are used to simplify digital circuits.

3.4.2 Integrated Circuits

Computers are composed of various digital components, connected by wires. Like a good program, the actual hardware of a computer uses collections of gates