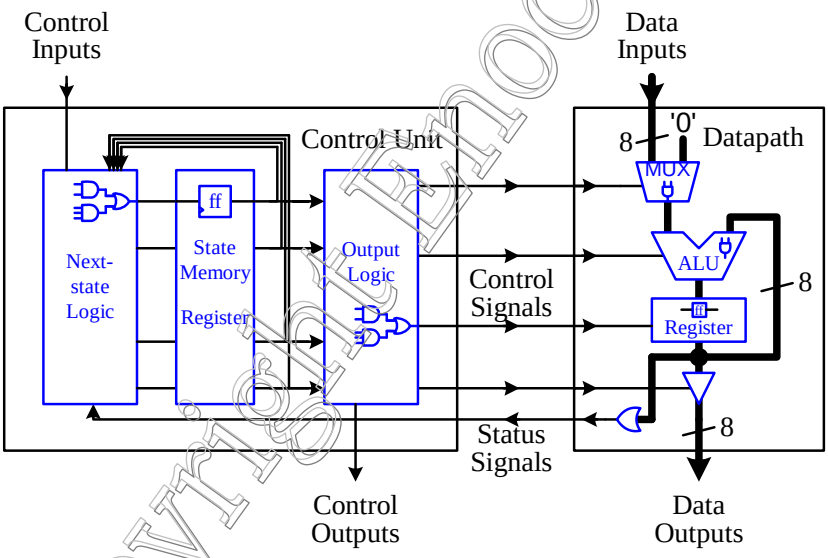
# Contents

## Chapter 2

# Digital Circuits



Our world is an analog world. Measurements that we make of the physical objects around us are never in discrete units, but rather in a continuous range. We talk about physical constants such as 2.718281828… or 3.141592…. To build analog devices that can process these values accurately is next to impossible. Even building a simple analog radio requires very accurate adjustments of frequencies, voltages, and currents at each part of the circuit. If we were to use voltages to represent the constant 3.14, we would have to build a component that will give us exactly 3.14 volts every time. This is again impossible; due to the imperfect manufacturing process, each component produced is slightly different from the others. Even if the manufacturing process can be made as perfect as perfect can get, we still would not be able to get 3.14 volts from this component every time we use it. The reason being that the physical elements used in producing the component behave differently in different environments, such as temperature, pressure, and gravitational force, just to name a few. Therefore, even if the manufacturing process is perfect, using this component in different environments will not give us exactly 3.14 volts every time.

To make things simpler, we work with a digital abstraction of our analog world. Instead of working with an infinite continuous range of values, we use just two values! Yes, just two values: 1 and 0, on and off, high and low, true and false, black and white, or however you want to call it. It certainly is much easier to control and work with two values rather than an infinite range. We call these two values a binary value for the reason that there are only two of them. A single 0 or a single 1 is then a **binary digit** or **bit**. This sounds great, but we have to remember that the underlining building block for our digital circuits is still based on an analog world.

This chapter provides the theoretical foundations for building digital logic circuits using logic gates, the basic building blocks for all digital circuits. In order to understand how logic gates are used to implement digital circuits, we need to have a good understanding of the basic theory of Boolean algebra, Boolean functions, and how to use and manipulate them. Most people may find Sections 2.5 and 2.6 on these theories to be boring, but let me encourage you to grind through it patiently, because if you do not understand it now, you quickly will get lost in the later chapters. The good news is that these two sections are the only sections in this book on theory, and I will try to keep them as short and simple as possible. You also will find that many of the Boolean theorems are very familiar, because they are similar to the algebra theorems that you have learned from your high school math class. As you can see from the microprocessor road map, this chapter affects all of the parts for building a microprocessor.

## 2.1 Binary Numbers

Since digital circuits deal with binary values, we will begin with a quick introduction to binary numbers. A bit, having either the value of 0 or 1, can represent only two things or two pieces of information. It is, therefore, necessary to group many bits together to represent more pieces of information. A string of $n$ bits can represent $2^n$ different pieces of information. For example, a string of two bits results in the four combinations: 00, 01, 10, and 11. By using different encoding techniques, a group of bits can be used to represent different information, such as a number, a letter of the alphabet, a character symbol, or a command for the microprocessor to execute.

The use of decimal numbers is quite familiar to us. However, since the binary digit is used to represent information within the computer, we also need to be familiar with **binary numbers**. Note that the use of binary numbers is just a form of representation for a string of bits. We can just as well use octal, decimal, or hexadecimal numbers to represent the string of bits. In fact, you will find that hexadecimal numbers are often used as a shorthand notation for binary numbers.

The decimal number system is a positional system. In other words, the value of the digit is dependent on the position of the digit within the number. For example, in the decimal number 48, the decimal digit 4 has a greater value than the decimal digit 8 because it is in the tenth position, whereas the digit 8 is in the unit position. The value of the number is calculated as $(4 \times 10^1) + (8 \times 10^0)$.

Like the decimal number system, the binary number system is also a positional system. The only difference between the two is that the binary system is a base-2 system, and so, it uses only two digits, 0 and 1, instead of ten. The binary numbers from 0 to 15 (decimal) are shown in Figure 2.1. The range from 0 to 15 has 16 different combinations. Since $2^4 = 16$, therefore, we need a 4-bit binary number (i.e., a string of four bits) to represent this range.

When we count in decimal, we count from 0 to 9. After 9, we go back to 0, and have a carry of a 1 to the next digit. When we count in binary numbers, we do the same thing, except that we only count from 0 to 1. After 1, we go back to 0 and have a carry of a 1 to the next bit.

The decimal value of a binary number can be found just like that for a decimal number, except that we raise the base number 2 to a power rather than the base number 10 to a power. For example, the value for the decimal number 658 is

$$658_{10} = (6 \times 10^2) + (5 \times 10^1) + (8 \times 10^0) = 600 + 50 + 8 = 658_{10}$$

Similaly, the decimal value for the binary number $1011011_2$ is

$$1011011_2 = (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

$$= 64 + 16 + 8 + 2 + 1 = 91_{10}$$

To get the decimal value, the least significant bit (in this case, the rightmost 1) is multiplied with $2^0$. The next bit to the left is multiplied with $2^1$, and so on. Finally, they are all added together to give the value $91_{10}$.

Notice the subscript 10 in the decimal number $658_{10}$, and the 2 in the binary number $1011011_2$. This subscript is used to denote the base of the number whenever there might be confusion as to what base the number is in.

| Decimal | Binary | Octal | Hexadecimal |
|---|---|---|---|
| 0 | 0000 | 0 | 0 |
| 1 | 0001 | 1 | 1 |
| 2 | 0010 | 2 | 2 |
| 3 | 0011 | 3 | 3 |
| 4 | 0100 | 4 | 4 |
| 5 | 0101 | 5 | 5 |
| 6 | 0110 | 6 | 6 |
| 7 | 0111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

**Figure 2.1** Numbers from 0 to 15 in binary, octal, and hexadecimal number systems.

Converting a decimal number to its binary equivalent can be done by successively dividing the decimal number by 2 and keeping track of the remainder at each step. Combining the remainders together (starting with the last one) forms the equivalent binary number. For example, using the decimal number 91, we divide it by 2 to get 45 with a remainder of 1. Then we divide 45 by 2 to get 22 with a remainder of 1. We continue in this fashion until the end as shown here.

```
2 | 91  1   <-- Least significant bit
2 | 45  1
2 | 22  0
2 | 11  1          = 1011011
2 |  5  1
2 |  2  0
    1   <-- Most significant bit
```

Concatenating the remainders together, starting with the last one (most significant bit) results in the binary number $1011011_2$.

Binary numbers usually consist of a long string of bits. A shorthand notation for writing out this lengthy string of bits is to use either octal or hexadecimal numbers. Since the octal system is base-8 and the hexadecimal system is base-16 (both of which are a power of 2), a binary number can be converted easily to an octal or hexadecimal number, or vice versa.

**Octal** numbers only use the digits from 0 to 7 for the eight different combinations. When counting in octals, the number after 7 is 10 as shown in Figure 2.1. To convert a binary number to octal, we simply group the bits into groups of threes, starting from the right (least significant bit). The reason for this is because $8 = 2^3$. For each group of three bits, we write the equivalent octal digit for it. For example, the conversion of the binary number $1110011_2$ to the octal number $163_8$ is shown here.

| 001 | 110 | 011 |
|---|---|---|
| 1 | 6 | 3 |

Since the original binary number has seven bits, we need to extend it with two leading 0's to get three bits for the leftmost group. Note that when we are dealing with negative numbers, we may require extending the number with leading 1's instead of 0's.

Converting an octal number to its binary equivalent is just as easy. For each octal number, we write down the equivalent three bits. These groups of three bits are concatenated together to form the final binary number. For example, the conversion of the octal number $5724_8$ to the binary number $101111010100_2$ is shown here.

| 5 | 7 | 2 | 4 |
|---|---|---|---|
| 101 | 111 | 010 | 100 |

The decimal value of an octal number can be found just like that for a binary or decimal number, except that we raise the base number 8 to a power instead of the base number 2 or 10 to a power. For example, the octal number $5724_8$ has the value:

$$5724_8 = (5 \times 8^3) + (7 \times 8^2) + (2 \times 8^1) + (4 \times 8^0) = 2560 + 448 + 16 + 4 = 3028_{10}$$

**Hexadecimal** numbers are treated basically the same way as octal numbers except with the appropriate changes to the base. Hexadecimal (or hex for short) numbers use base-16, and thus require 16 different digit symbols, as shown in Figure 2.1. Converting binary numbers to hexadecimal numbers involves grouping the bits into groups of fours since $16 = 2^4$. For example, the conversion of the binary number $11011011011_2$ to the hexadecimal number $6DB_{16}$ is shown below. Again, we need to extend it with a leading 0 to get four bits for the leftmost group.

| 0110 | 1101 | 1011 |
|---|---|---|
| 6 | D | B |

To convert a hex number to a binary number, we write down the equivalent four bits for each hex digit, and then concatenate them together to form the final binary number. For example, the conversion of the hexadecimal number $5C4A_{16}$ to the binary number $0101110001001010_2$ is shown here.

| 5 | C | 4 | A |
|---|---|---|---|
| 0101 | 1100 | 0100 | 1010 |

The following example shows how the decimal value of the hexadecimal number $C4A_{16}$ is evaluated.

$$C4A_{16} = (C \times 16^2) + (4 \times 16^1) + (A \times 16^0)$$

$$= (12 \times 16^2) + (4 \times 16^1) + (10 \times 16^0)$$

$$= 3072 + 64 + 10 = 3146_{10}$$

## 2.2 Binary Switch

Besides the fact that we are working only with binary values, digital circuits are easy to understand because they are based on one simple idea of turning a switch on or off to obtain either one of the two binary values. Since

the switch can be in either one of two states (on or off), we call it a **binary switch**, or just a **switch** for short. The switch has three connections: an input, an output, and a control for turning the switch on or off, as shown in Figure 2.2. When the switch is opened, as in Figure 2.2(a), it is turned off, and nothing gets through from the input to the output. When the switch is closed, as in Figure 2.2(b), it is turned on, and whatever is presented at the input is allowed to pass through to the output.


**Figure 2.2** Binary switch: (a) opened or off; (b) closed or on.

Uses of the binary switch idea can be found in many real-world devices. For example, the switch can be an electrical switch with the input connected to a power source and the output connected to a siren *S*, as shown in Figure 2.3.

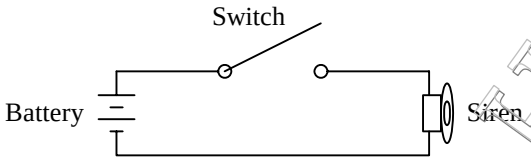



**Figure 2.3** A siren controlled by a switch.

When the switch is closed, the siren turns on. The usual convention is to use a 1 to mean "on" and a 0 to mean "off." Therefore, when the switch is closed, the output is a 1 and the siren will turn on. We can also use a variable, *x*, to denote the state of the switch. We can let $x = 1$ to mean the switch is closed and $x = 0$ to mean the switch is opened. Using this convention, we can describe the state of the siren *S* in terms of the variable *x* using a simple logic expression. Since $S = 1$ if $x = 1$ and $S = 0$ if $x = 0$, we can write

$$S = x$$

This logic expression describes the output *S* in terms of the input variable *x*.

## 2.3 Basic Logic Operators and Logic Expressions

Two binary switches can be connected together either in series or in parallel, as shown in Figure 2.4.

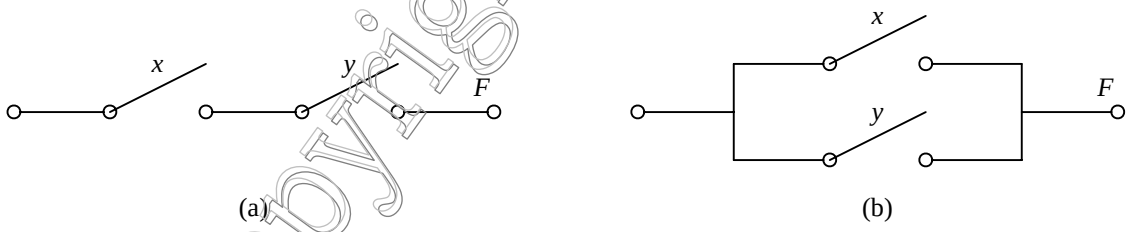



**Figure 2.4** Connection of two binary switches: (a) in series; (b) in parallel.

If two switches are connected in series, as in Figure 2.4(a), then both switches have to be on in order for the output *F* to be 1. In other words, $F = 1$ if $x = 1$ and $y = 1$. If either *x* or *y* is off, or both are off, then $F = 0$. Translating this into a logic expression, we get

$$F = x \text{ AND } y$$

Hence, two switches connected in series give rise to the logical **AND** operator. In a Boolean function (which we will explain in more detail in Section 2.5), the AND operator is denoted either with a dot ( • ) or no symbol at all. Thus, we can rewrite the above expression as

$$F = x \bullet y$$

or simply

$$F = xy$$

If we connect two switches in parallel, as in Figure 2.4(b), then only one switch needs to be on in order for the output $F$ to be 1. In other words, $F = 1$ if either $x = 1$, or $y = 1$, or both $x$ and $y$ are 1's. This means that $F = 0$ only if both $x$ and $y$ are 0's. Translating this into a logic expression, we get

$$F = x \text{ OR } y$$

and this gives rise to the logical **OR** operator. In a Boolean function, the OR operator is denoted with a "plus" symbol ( + ). Thus, we can rewrite the above expression as

$$F = x + y$$

In addition to the AND and OR operators, there is another basic logic operator—the **NOT** operator, also known as the **INVERTER**. Whereas, the AND and OR operators have multiple inputs; the NOT operator has only one input and one output. The NOT operator simply inverts its input, so a 0 input will produce a 1 output, and a 1 becomes a 0. In a Boolean function, the NOT operator is denoted either with an "apostrophe" symbol ( ′ ) or a "bar" on top ( ¯ ) as in

$$F = x'$$

or

$$F = \bar{x}$$

When several operators are used in the same expression, the precedence given to the operators are (from highest to lowest) NOT, AND, and OR. The order of evaluation can be changed by means of using parenthesis. For example, the expression

$$F = xy + z'$$

means ($x$ AND $y$) OR (NOT $z$), and the expression

$$F = x(y + z)'$$

means $x$ AND (NOT ($y$ OR $z$)).

## 2.4 Truth Tables

The operation of the AND, OR, and NOT logic operators can be described formally by using a **truth table**, as shown in Figure 2.5. A truth table is a two-dimensional array where there is one column for each input and one column for each output (a circuit may have more than one output). Since we are dealing with binary values, each input can be either a 0 or a 1. We simply enumerate all possible combinations of 0's and 1's for all the inputs. Usually, we want to write these input values in the normal binary counting order. With two inputs, there are $2^2$ combinations giving us the four rows in the table. The values in the output column are determined from applying the corresponding input values to the functional operator. For the AND truth table in Figure 2.5(a), $F = 1$ only when $x$ and $y$ are both 1, otherwise, $F = 0$. For the OR truth table in Figure 2.5(b), $F = 1$ when either $x$ or $y$ or both is a 1, otherwise $F = 0$. For the NOT truth table, the output $F$ is just the inverted value of the input $x$.

| *x* | *y* | *F* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)

| *x* | *y* | *F* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(b)

| *x* | *F* |
|---|---|
| 0 | 1 |
| 1 | 0 |

(c)

**Figure 2.5** Truth tables for the three basic logical operators: (a) AND; (b) OR; (c) NOT.

Using a truth table is one method to formally describe the operation of a circuit or function. The truth table for any given logic expression (no matter how complex it is) can always be derived. Examples on the use of truth tables to describe digital circuits are given in the following sections. Another method to formally describe the operation of a circuit is by using Boolean expressions or Boolean functions.

## *2.5 Boolean Algebra and Boolean Functions*

### 2.5.1 Boolean Algebra

George Boole, in 1854, developed a system of mathematical logic, which we now call **Boolean algebra**. Based on Boole's idea, Claude Shannon, in 1938, showed that circuits built with binary switches can easily be described using Boolean algebra. The abstraction from switches being on and off to the use of Boolean algebra is as follows. Let $B = \{0, 1\}$ be the Boolean algebra whose elements are one of the two values, 0 and 1. We define the operations AND (•), OR (+), and NOT ($'$) for the elements of $B$ by the axioms in Figure 2.6(a). These axioms are simply the definitions for the AND, OR, and NOT operators.

A variable $x$ is called a **Boolean variable** if $x$ takes on only values in $B$ (i.e., either 0 or 1). Consequently, we obtain the theorems in Figure 2.6(b) for single variable and Figure 2.6(c) for two and three variables.

The theorems in Figure 2.6(b) can be proven easily by substituting the binary values into the expressions and using the axioms. For example, to show that Theorem 6a is true, we substitute 0 into $x$ to get Axiom 3a, and substitute 1 into $x$ to get Axiom 2a.

To prove the theorems in Figure 2.6(c), we can use either one of two methods: (1) use a truth table, or (2) use axioms and theorems that have already been proven. We show these two methods in the following two examples.

| | | | |
|---|---|---|---|
| 1a. | $0 \bullet 0 = 0$ | 1b. | $1 + 1 = 1$ |
| 2a. | $1 \bullet 1 = 1$ | 2b. | $0 + 0 = 0$ |
| 3a. | $0 \bullet 1 = 1 \bullet 0 = 0$ | 3b. | $1 + 0 = 0 + 1 = 1$ |
| 4a. | $0' = 1$ | 4b. | $1' = 0$ |

(a)

| | | | | |
|---|---|---|---|---|
| 5a. | $x \bullet 0 = 0$ | 5b. | $x + 1 = 1$ | Null Element |
| 6a. | $x \bullet 1 = 1 \bullet x = x$ | 6b. | $x + 0 = 0 + x = x$ | Identity |
| 7a. | $x \bullet x = x$ | 7b. | $x + x = x$ | Idempotent |
| 8a. | $(x')' = x$ | | | Double Complement |
| 9a. | $x \bullet x' = 0$ | 9b. | $x + x' = 1$ | Inverse |

(b)

| | | | | |
|---|---|---|---|---|
| 10a. | $x \bullet y = y \bullet x$ | 10b. | $x + y = y + x$ | Commutative |
| 11a. | $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ | 11b. | $(x + y) + z = x + (y + z)$ | Associative |
| 12a. | $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$ | 12b. | $x + (y \bullet z) = (x + y) \bullet (x + z)$ | Distributive |
| 13a. | $x \bullet (x + y) = x$ | 13b. | $x + (x \bullet y) = x$ | Absorption |
| 14a. | $(x \bullet y) + (x \bullet y') = x$ | 14b. | $(x + y) \bullet (x + y') = x$ | Combining |

| | | | | |
|---|---|---|---|---|
| 15a. | $(x \bullet y)' = x' + y'$ | 15b. | $(x + y)' = x' \bullet y'$ | DeMorgan's |

(c)

**Figure 2.6** Boolean algebra axioms and theorems: (a) axioms; (b) single-variable theorems; (c) two- and three-variable theorems.

**Example 2.1**: Proof of theorem using a truth table.

Prove Theorem 12a from Figure 2.6(c), using a truth table.

Theorem 12a states that $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$. To prove that Theorem 12a is true using a truth table, we need to show that, for every combination of values for the three variables *x*, *y*, and *z*, the left-hand side of the expression is equal to the right-hand side.

We start with the first three columns labeled *x*, *y*, and *z*; and we enumerate all possible combinations of values for these three variables giving us the eight rows as shown next.

| *x* | *y* | *z* | $(y + z)$ | $(x \bullet y)$ | $(x \bullet z)$ | $x \bullet (y + z)$ | $(x \bullet y) + (x \bullet z)$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

For each combination (row), we evaluate the intermediate expressions $(y + z)$, $(x \bullet y)$, and $(x \bullet z)$ by substituting the values of *x*, *y*, and *z* into the expression. Finally, we obtain the values for the last two columns, which correspond to the left-hand side and right-hand side of Theorem 12a. The values in these two columns are identical for every combination of *x*, *y*, and *z*; therefore, we can say that Theorem 12a is true. ♦

**Example 2.2**: Proof of theorem using axioms and theorems.

Prove Theorem 13b from Figure 2.6(c), using other axioms and theorems from the figure.

Theorem 13b states that $x + (x \bullet y) = x$. To prove that Theorem 13b is true using other axioms and theorems, we can argue as follows:

$$
\begin{aligned}
x + (x \bullet y) &= (x \bullet 1) + (x \bullet y) && \text{by Identity Theorem 6a} \\
&= x \bullet (1 + y) && \text{by Distributive Theorem 12a} \\
&= x \bullet (1) && \text{by Null element Theorem 5b} \\
&= x && \text{by Identity Theorem 6a}
\end{aligned}
$$
♦

Example 2.2 shows that some theorems can be derived from others that already have been proven with the truth table. Full treatment of Boolean algebra is beyond the scope of this book and can be found in the references. For our purposes, we simply assume that all of the theorems are true and will use them just to show that two circuits are equivalent, as depicted in the next two examples.

**Example 2.3**: Using Boolean algebra to reduce an equation

Use Boolean algebra to reduce the equation $F(x,y,z) = (x' + y' + x'y' + xy)\ (x' + yz)$ as much as possible.

$$F = (x' + y' + x'y' + xy)\ (x' + yz)$$

$= (x' \bullet 1 + y' \bullet 1 + x'y' + xy)\ (x' + yz)$ by Identity Theorem 6a
$= (x'\ (y + y'\ ) + y'\ (x + x'\ ) + x'y' + xy)\ (x' + yz)$ by Inverse Theorem 9b
$= (x'y + x'y' + y'x + y'x' + x'y' + xy)\ (x' + yz)$ by Distributive Theorem 12a
$= (x'y + x'y' + y'x + \sout{y'x'} + \sout{x'y'} + xy)\ (x' + yz)$ by Idempotent Theorem 7b
$= (x'\ (y + y') + x\ (y + y'))\ (x' + yz)$ by Distributive Theorem 12a
$= (x' \bullet 1 + x \bullet 1)\ (x' + yz)$ by Inverse Theorem 9b
$= (x' + x)\ (x' + yz)$ by Identity Theorem 6a
$= 1\ (x' + yz)$ by Inverse Theorem 9b
$= (x' + yz)$ by Identity Theorem 6a

Since the expression $(x' + y' + x'y' + xy)\ (x' + yz)$ reduces down to $(x' + yz)$, therefore, we do want to implement the circuit for the latter expression rather than the former because the circuit size for the latter is much smaller. ♦

**Example 2.4**: Using Boolean algebra to show that two equations are equivalent

Show, using Boolean algebra, that the two equations

$F_1 = (xy' + x'y + x' + y' + z'\ )\ (x + y' + z)$ and

$F_2 = y' + x'z + xz'$

are equivalent.

$F_1 = (xy' + x'y + x' + y' + z'\ )\ (x + y' + z)$
$= xy'x + xy'y' + xy'z + x'yx + x'yy' + x'yz + x'x + x'y' + x'z + y'x + y'y' + y'z + z'x + z'y' + z'z$
$= xy' + xy' + xy'z + 0 + 0 + x'yz + 0 + x'y' + x'z + xy' + y' + y'z + xz' + y'z' + 0$
$= xy' + xy'z + x'yz + x'y' + x'z + y' + y'z + xz' + y'z'$
$= y'(x + xz + x' + 1 + z + z') + x'z(y + 1) + xz'$
$= y' + x'z + xz'$
$= F_2$ ♦

## 2.5.2 * Duality Principle

Notice in Figure 2.6 that we have listed the axioms and theorems in pairs. Specifically, we define the **dual** of a logic expression as one that is obtained by changing all + operators with • operators, and vice versa, and by changing all 0's with 1's, and vice versa. For example, the dual of the logic expression:

$(x \bullet y' \bullet z) + (x \bullet y \bullet z'\ ) + (y \bullet z) + 0$

is

$(x + y' + z) \bullet (x + y + z'\ ) \bullet (y + z) \bullet 1$

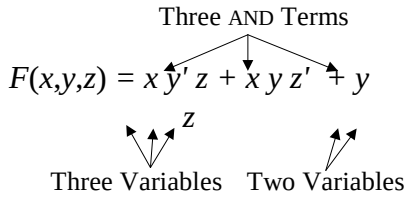
The **duality principle** states that if a Boolean expression is true, then its dual is also true. Be careful in that it does not say that a Boolean expression is equivalent to its dual. For example, Theorem 5a in Figure 2.6(b) says that $x \bullet 0 = 0$ is true, thus by the duality principle, its dual, $x + 1 = 1$ is also true. However, $x \bullet 0 = 0$ is not equal to $x + 1 = 1$, since 0 definitely is not equal to 1.

We will see in Section 2.5.3 that the inverse of a Boolean expression easily can be obtained by first taking the dual of that expression and then complementing each Boolean variable in the resulting dual expression. In this respect, the duality principle is often used in digital logic design. Whereas an expression might be complex to implement, its inverse might be simpler, thus resulting in a smaller circuit; inverting the final output of this circuit will produce the same result as from the original expression.

## 2.5.3 Boolean Functions and their Inverses

As we have seen, any digital circuit can be described by a logical expression, also known as a **Boolean function**. Any Boolean functions can be formed from binary variables and the Boolean operators •, +, and ′ (for AND, OR, and NOT, respectively). For example, the following Boolean function uses the three variables or literals *x*, *y*, and *z*. It has three **AND terms** (also referred to as **product terms**), and these AND terms are ORed (summed)

together. The first two AND terms each contain all three variables, while the last AND term contains only two variables. By definition, an AND (or product) term is either a single variable or two or more variables ANDed together. Quite often, we refer to functions that are in this format as a **sum-of-products** or **or-of-ands**.




The value of a function evaluates to either 0 or 1, depending on the given set of values for the variables. For example, the function in the above figure evaluates to 1 when any one of the three AND terms evaluate to a 1, since 1 OR *x* is 1. The first AND term, *xy'z*, equals 1 if

*x* = 1, *y* = 0, and *z* = 1

because, if we substitute these values for *x*, *y*, and *z* into the first AND term *xy'z*, we get a 1. Similarly, the second AND term, *xyz'*, equals 1 if

*x* = 1, *y* = 1, and *z* = 0

The last AND term, *yz*, has only two variables. What this means is that the value of this term is not dependent on the missing variable *x*. In other words, *x* can be either 0 or 1, but as long as *y* = 1 and *z* = 1, this term will be equal to 1.

Thus, we can summarize by saying that *F* evaluates to 1 if

*x* = 1, *y* = 0, and *z* = 1

or

*x* = 1, *y* = 1, and *z* = 0

or

*x* = 0, *y* = 1, and *z* = 1

or

*x* = 1, *y* = 1, and *z* = 1.

Otherwise, *F* evaluates to 0.

It is often more convenient to summarize this verbal description of a function with a truth table, as shown in Figure 2.7 under the column labeled *F*. Notice that the four rows in the table, where *F* = 1, match the four cases in the description.

| *x* | *y* | *z* | *F* | *F'* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

**Figure 2.7** Truth table for the function *F* = *xy'z* + *xyz'* + *yz*

The inverse of a function, denoted by *F'*, can be obtained easily from the truth table for *F* by simply changing all the 0's to 1's and 1's to 0's, as shown in the truth table in Figure 2.7 under the column labeled *F'*. Therefore, we can write the Boolean function for *F'* in the sum-of-products format, where the AND terms are obtained from those rows where *F'* = 1. Thus, we get

$$F' = x'y'z' + x'y'z + x'yz' + xy'z'$$

To deduce $F'$ algebraically from $F$ requires the use of DeMorgan's theorem (Theorem 15a in Figure 2.6(c)) twice. For example, using the same function

$$F = xy'z + xyz' + yz$$

we obtain $F'$ as follows

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

There are three things to notice about this equation for $F'$. First, $F'$ is just the dual of $F$ (as defined in Section 2.5.2) with all the variables inverted. Second, instead of being in a sum-of-products format, it is in a **product-of-sums** (**and-of-ors**) format where three OR terms (also referred to as "sum" terms) are ANDed together. Third, from the same original function $F$, we obtained two different equations for $F'$. From the truth table in Figure 2.7, we obtained

$$F' = x'y'z' + x'y'z + x'yz' + xy'z'$$

and from applying DeMorgan's Theorem to $F$, we obtained

$$F' = (x' + y + z' ) \bullet (x' + y' + z) \bullet (y' + z' )$$

Hence, we must conclude that these two expressions for $F'$, where one is in the sum-of-products format and the other is in the product-of-sums format, are equivalent. In general, all functions can be expressed in either the sum-of-products or product-of-sums format.

Thus, we should also be able to express the same function, $F = xy'z + xyz' + yz$, in the product-of-sums format. We can derive it using one of two methods. For method one, we can start with $F'$ and apply DeMorgan's theorem to it just like how we obtained $F'$ from $F$.

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

For the second method, we start with the original $F$ and convert it to the product-of-sums format using the Boolean theorems from Figure 2.6.

$$\begin{aligned} F &= xy'z + xyz' + yz \\ &= (x+x+y) \bullet (x+x+z) \bullet (x+y+y) \bullet (x+y+z) \bullet (x+z'+y) \bullet \cancel{(x+z'+z)} \bullet && \text{(Step 1)} \\ &\quad \cancel{(y'+x+y)} \bullet (y'+x+z) \bullet \cancel{(y'+y+y)} \bullet \cancel{(y'+y+z)} \bullet \cancel{(y'+z'+y)} \bullet \cancel{(y'+z'+z)} \bullet \\ &\quad (z+x+y) \bullet (z+x+z) \bullet (z+y+y) \bullet (z+y+z) \bullet \cancel{(z+z'+y)} \bullet \cancel{(z+z'+z)} \\ &= (x+y) \bullet (x+z) \bullet \cancel{(x+y)} \bullet (x+y+z) \bullet (x+z'+y) \bullet (y'+x+z) \bullet \cancel{(z+x+y)} \bullet \cancel{(z+x)} \bullet (z+y) \bullet \cancel{(z+y)} && \text{(Step 2)} \\ &= (x+y) \bullet (x+z) \bullet (x+y+z) \bullet (x+y+z' ) \bullet (x+y'+z) \bullet (y+z) && \text{(Step 3)} \\ &= (x+y+zz' ) \bullet (x+yy'+z) \bullet (x+y+z) \bullet (x+y+z' ) \bullet (x+y'+z) \bullet (xx'+y+z) && \text{(Step 4)} \\ &= (x+y+z) \bullet (x+y+z') \bullet \cancel{(x+y+z)} \bullet (x+y'+z) \bullet \cancel{(x+y+z)} \bullet \cancel{(x+y+z')} \bullet \cancel{(x+y'+z)} \bullet \cancel{(x+y+z)} \bullet (x'+y+z) && \text{(Step 5)} \\ &= (x+y+z) \bullet (x+y+z' ) \bullet (x+y'+z) \bullet (x'+y+z) \end{aligned}$$

In Step 1, we apply Theorem 12b (Distributive) to get every possible combination of the sum terms. For example, the first sum term $(x + x + y)$ is obtained from getting the first $x$ from $xy'z$, the second $x$ from $xyz'$, and the $y$ from $yz$. The second sum term $(x + x + z)$ is obtained from getting the first $x$ from $xy'z$, the second $x$ from $xyz'$, and the $z$ from $yz$. This is repeated for all combinations. In this step, the sum terms, such as $(x + z' + z)$, where it contains variables of the form $v + v'$, can be eliminated, since $v + v' = 1$, and $1 \bullet x = x$.

In Step 2 and Step 3, duplicate variables and terms are eliminated. For example, the term $(x + x + y)$ is equal to $(x + y + y)$, which is just $(x + y)$. The term $(x + z' + z)$ is equal to $(x + 1)$, which is equal to just 1, and therefore, can be eliminated completely from the expression.

In Step 4, every sum term with a missing variable will have that variable added back in by using Theorems 6b and 9a, which says that $x + 0 = x$ and $yy' = 0$, therefore, $x + yy' = x$.

Step 5 uses the Distributive Theorem, and the resulting duplicate terms are again eliminated to give us the format that we want.

Functions that are in the product-of-sums format (such as the one shown below) are more difficult to deduce when they evaluate to 1. For example, using

$$F' = (x' + y + z') \bullet (x' + y' + z) \bullet (y' + z')$$

$F'$ evaluates to 1 when all three terms evaluate to 1. For the first term to evaluate to 1, $x$ can be 0, $y$ can be 1, or $z$ can be 0. For the second term to evaluate to 1, $x$ can be 0, $y$ can be 0, or $z$ can be 1. Finally, for the last term, $y$ can be 0, $z$ can be 0, or $x$ can be either 0 or 1. As a result, we end up with many more combinations to consider, even though many of the combinations are duplicates.

However, it is easier to determine when a product-of-sums format expression evaluates to 0. For example, using the same expression:

$$F' = (x' + y + z') \bullet (x' + y' + z) \bullet (y' + z')$$

$F'$ evaluates to 0 when any one of the three OR terms is 0, since 0 AND $x$ is 0; and this happens when

$x = 1$, $y = 0$, and $z = 1$ for the first OR term,

or

$x = 1$, $y = 1$, and $z = 0$ for the second OR term,

or

$y = 1$, $z = 1$, and $x$ can be either 0 or 1 for the last OR term.

Similarly, for a sum-of-products format expression, it is easy to evaluate when it is a 1 but difficult to evaluate when it is a 0.

These four conditions in which $F'$ evaluates to 0 match exactly the four rows in the truth table shown in Figure 2.7 where $F' = 0$. Therefore, we see that, in general, the unique algebraic expression for any Boolean function can be specified by either (1) selecting the rows from the truth table where the function is a 1 and using the sum-of-products format, or (2) selecting the rows from the truth table where the function is a 0 and using the product-of-sums format. Whatever format we decide to use, the one thing to remember is that we are always interested in only when the function (or its inverse) is equal to a 1.

Figure 2.8 summarizes these two formats for the function $F = xy'z + xyz' + yz$ and its inverse $F'$. Notice that the sum-of-products format for $F$ is the dual with its variables inverted from the product-of-sums format for $F'$. Similarly, the product-of-sums format for $F$ is the dual with its variables inverted from the sum-of-products format for $F'$.

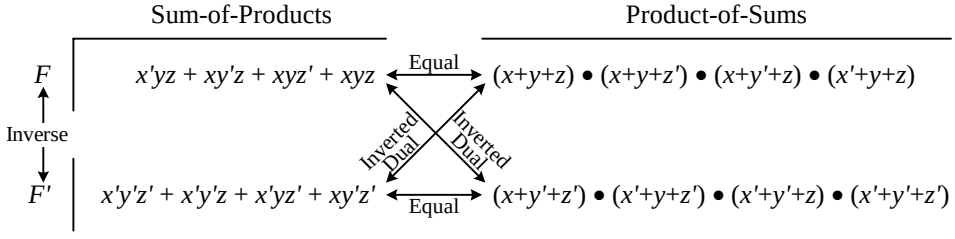



**Figure 2.8** Relationships between the function $F = xy'z + xyz' + yz$ and its inverse $F'$, and between the sum-of-products and product-of-sums formats. The label "Inverted Dual" means applying the duality principle and then inverting the variables.

## *2.6 Minterms and Maxterms*

As you recall, a product term is a term with either a single variable or two or more variables ANDed together, whereas, a sum term is a term with either a single variable or two or more variables ORed together. To differentiate

between a term that contains any number of variables with a term that contains *all* the variables used in the function, we use the words minterm and maxterm. We are not introducing new ideas here; rather, we are just introducing two new words and notations for defining what we have already learned.

## 2.6.1 Minterms

A **minterm** is a product term that contains all the variables used in a function. For a function with $n$ variables, the notation $m_i$, where $0 \le i < 2^n$, is used to denote the minterm whose index $i$ is the binary value of the $n$ variables, such that the variable is complemented if the value assigned to it is a 0 and uncomplemented if it is a 1.

For example, for a function with three variables $x$, $y$, and $z$, the notation $m_3$ is used to represent the term in which the values for the variables $xyz$ are 011 (for the subscript 3). Since we want to complement the variable whose value is a 0 and uncomplement it if it is a 1, therefore, $m_3$ is for the minterm $x'yz$. Figure 2.9(a) shows the eight minterms and their notations for $n = 3$ using the three variables $x$, $y$, and $z$.

When specifying a function, we usually start with product terms that contain all of the variables used in the function. In other words, we want the **sum of minterms**, and more specifically, the sum of the one-minterms (i.e., the minterms for which the function is a 1) as opposed to the zero-minterms (i.e., the minterms for which the function is a 0). We use the notation **1-minterm** to denote one-minterm, and **0-minterm** to denote zero-minterm.

| $x$ | $y$ | $z$ | Minterm | Notation |
|---|---|---|---|---|
| 0 | 0 | 0 | $x'\,y'\,z'$ | $m_0$ |
| 0 | 0 | 1 | $x'\,y'\,z$ | $m_1$ |
| 0 | 1 | 0 | $x'\,y\,z'$ | $m_2$ |
| 0 | 1 | 1 | $x'\,y\,z$ | $m_3$ |
| 1 | 0 | 0 | $x\,y'\,z'$ | $m_4$ |
| 1 | 0 | 1 | $x\,y'\,z$ | $m_5$ |
| 1 | 1 | 0 | $x\,y\,z'$ | $m_6$ |
| 1 | 1 | 1 | $x\,y\,z$ | $m_7$ |

(a)

| $x$ | $y$ | $z$ | Maxterm | Notation |
|---|---|---|---|---|
| 0 | 0 | 0 | $x + y + z$ | $M_0$ |
| 0 | 0 | 1 | $x + y + z'$ | $M_1$ |
| 0 | 1 | 0 | $x + y' + z$ | $M_2$ |
| 0 | 1 | 1 | $x + y' + z'$ | $M_3$ |
| 1 | 0 | 0 | $x' + y + z$ | $M_4$ |
| 1 | 0 | 1 | $x' + y + z'$ | $M_5$ |
| 1 | 1 | 0 | $x' + y' + z$ | $M_6$ |
| 1 | 1 | 1 | $x' + y' + z'$ | $M_7$ |

(b)

**Figure 2.9** (a) Minterms for three variables. (b) Maxterms for three variables.

The function from the previous section:

$$F = xy'z + xyz' + yz$$
$$= x'yz + xy'z + xyz' + xyz$$

and repeated in the following truth table has the 1-minterms: $m_3$, $m_5$, $m_6$, and $m_7$.

| $x$ | $y$ | $z$ | $F$ | $F'$ | Minterm | Notation |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | $x'\,y'\,z'$ | $m_0$ |
| 0 | 0 | 1 | 0 | 1 | $x'\,y'\,z$ | $m_1$ |
| 0 | 1 | 0 | 0 | 1 | $x'\,y\,z'$ | $m_2$ |
| 0 | 1 | 1 | 1 | 0 | $x'\,y\,z$ | $m_3$ |
| 1 | 0 | 0 | 0 | 1 | $x\,y'\,z'$ | $m_4$ |
| 1 | 0 | 1 | 1 | 0 | $x\,y'\,z$ | $m_5$ |
| 1 | 1 | 0 | 1 | 0 | $x\,y\,z'$ | $m_6$ |
| 1 | 1 | 1 | 1 | 0 | $x\,y\,z$ | $m_7$ |

Thus, a shorthand notation for the function is

$$F(x, y, z) = m_3 + m_5 + m_6 + m_7$$

By using just the minterm notations, we do not know how many variables are in the original function. Consequently, we need to explicitly specify the variables used by the function, as in $F(x, y, z)$. We can further

simplify the notation by using the standard algebraic symbol Σ for summation and listing out the minterm index numbers. Therefore, we have

$F(x, y, z) = \Sigma(3, 5, 6, 7)$

These are just different ways of expressing the same function.

Since a function is obtained from the sum of the 1-minterms, the inverse of the function, therefore, must be the sum of the 0-minterms. This can be obtained easily by replacing the set of indices with those that were excluded from the original set. Therefore,

$F'(x, y, z) = \Sigma(0, 1, 2, 4)$

**Example 2.5**: Converting a function to the sum-of-minterms format using Boolean algebra

Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the sum-of-minterms format.

This function has three variables. In a sum-of-minterms format, all product terms must have all variables. To do so, we need to expand each product term by ANDing it with $(v + v')$ for every missing variable $v$ in that term. Since $(v + v') = 1$, ANDing a product term with $(v + v')$ does not change the value of the term.

$F = y + x'z$
$= y(x+x')(z+z') + x'z(y+y')$ expand 1st term by ANDing it with $(x+x')(z+z')$, and 2nd term with $(y+y')$
$= xyz + xyz' + x'yz + x'yz' + \sout{x'yz} + x'y'z$
$= m_7 + m_6 + m_3 + m_2 + m_1$
$= \Sigma(1, 2, 3, 6, 7)$ sum of 1-minterms ♦

**Example 2.6**: Converting the inverse of a function to the sum-of-minterms format

Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the sum-of-minterms format.

$F' = (y + x'z)'$ inverse
$= y' \bullet (x'z)'$ use DeMorgan
$= y' \bullet (x+z')$ use DeMorgan
$= y'x + y'z'$ use Distributive Theorem to change to sum of products format
$= y'x(z+z') + y'z'(x+x')$ expand 1st term by ANDing it with $(z+z')$, and 2nd term with $(x+x')$
$= xy'z + xy'z' + \sout{xy'z'} + x'y'z'$
$= m_5 + m_4 + m_0$
$= \Sigma(0, 4, 5)$ sum of 0-minterms ♦

### 2.6.2 * Maxterms

Analogous to a minterm, a **maxterm** is a sum term that contains all of the variables used in the function. For a function with $n$ variables, the notation $M_i$, where $0 \le i < 2^n$, is used to denote the maxterm whose index $i$ is the binary value of the $n$ variables, such that the variable is complemented if the value assigned to it is a 1 and uncomplemented if it is a 0.

For example, for a function with three variables $x$, $y$, and $z$, the notation $M_3$ is used to represent the term in which the values for the variables $xyz$ are 011 (for the subscript 3). For maxterms, we want to complement the variable whose value is a 1 and uncomplement it if it is a 0, hence, $M_3$ is for the maxterm $x + y' + z'$. Figure 2.9(b) shows the eight maxterms and their notations for $n = 3$ using the three variables $x$, $y$, and $z$.

We have seen that a function can also be specified as a product-of-sums, or more specifically, a **product of 0-maxterms** (i.e., the maxterms for which the function is 0. Just like the minterms, we use the notation **1-maxterm** to denote one-maxterm, and **0-maxterm** to denote zero-maxterm. Thus, the function:

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

$= (x + y + z) \bullet (x + y + z') \bullet (x + y' + z) \bullet (x' + y + z)$

which is shown in the following table

| $x$ | $y$ | $z$ | $F$ | $F'$ | Maxterm | Notation |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | $x + y + z$ | $M_0$ |
| 0 | 0 | 1 | 0 | 1 | $x + y + z'$ | $M_1$ |
| 0 | 1 | 0 | 0 | 1 | $x + y' + z$ | $M_2$ |
| 0 | 1 | 1 | 1 | 0 | $x + y' + z'$ | $M_3$ |
| 1 | 0 | 0 | 0 | 1 | $x' + y + z$ | $M_4$ |
| 1 | 0 | 1 | 1 | 0 | $x' + y + z'$ | $M_5$ |
| 1 | 1 | 0 | 1 | 0 | $x' + y' + z$ | $M_6$ |
| 1 | 1 | 1 | 1 | 0 | $x' + y' + z'$ | $M_7$ |

can be specified as the product of the 0-maxterms $M_0$, $M_1$, $M_2$, and $M_4$. The shorthand notation for the function is

$F(x, y, z) = M_0 \bullet M_1 \bullet M_2 \bullet M_4$

By using the standard algebraic symbol $\Pi$ for product and listing out the maxterms index numbers, the notation is further simplified to

$F(x, y, z) = \Pi(0, 1, 2, 4)$

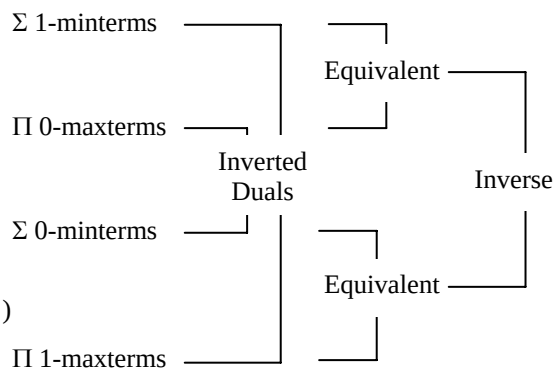
The following summarizes these relationships for the function $F = xy'z + xyz' + yz$ and its inverse. Comparing these equations with those in Figure 2.8, we see that they are identical.

$F(x, y, z) = x'\,y\,z + x\,y'\,z + x\,y\,z' + x\,y\,z$
$= m_3 + m_5 + m_6 + m_7$
$= \Sigma(3, 5, 6, 7)$

$= (x+y+z) \bullet (x+y+z') \bullet (x+y'+z) \bullet (x'+y+z)$
$= M_0 \bullet M_1 \bullet M_2 \bullet M_4$
$= \Pi(0, 1, 2, 4)$

$F'(x, y, z) = x'\,y'\,z' + x'\,y'\,z + x'\,y\,z' + x\,y'\,z'$
$= m_0 + m_1 + m_2 + m_4$
$= \Sigma(0, 1, 2, 4)$

$= (x+y'+z') \bullet (x'+y+z') \bullet (x'+y'+z) \bullet (x'+y'+z')$
$= M_3 \bullet M_5 \bullet M_6 \bullet M_7$
$= \Pi(3, 5, 6, 7)$




Notice that it is always the $\Sigma$ of minterms and $\Pi$ of maxterms; you never have $\Sigma$ of maxterms or $\Pi$ of minterms.

**Example 2.7**: Converting a function to the product-of-maxterms format

Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the function to the product-of-maxterms format.

To change a sum term to a maxterm, we expand each term by ORing it with $(vv')$ for every missing variable $v$ in that term. Since $(vv') = 0$, therefore, ORing a sum term with $(vv')$ does not change the value of the term.

$F = y + x'z$
$= y + (x'z)$
$= (y+x')(y+z)$ use Distributive Theorem to change to product of sums format
$= (y+x'+zz')(y+z+xx')$ expand 1st term by ORing it with $zz'$, and 2nd term with $xx'$
$= (x'+y+z)\,(x'+y+z')\,(x+y+z)\,$~~$(x'+y+z)$~~
$= M_4 \bullet M_5 \bullet M_0$
$= \Pi(0, 4, 5)$ product of 0-maxterms ♦

**Example 2.8**: Converting the inverse of a function to the product-of-maxterms format

Given the Boolean function $F(x, y, z) = y + x'z$, use Boolean algebra to convert the inverse of the function to the product-of-maxterms format.

$F' = (y + x'z)'$ inverse
$= y' \bullet (x'z)'$ use DeMorgan
$= y' \bullet (x+z')$ use DeMorgan
$= (y' + xx' + zz') \bullet (x+z'+yy')$ expand 1st term by ORing it with $xx' + zz'$, and 2nd term with $yy'$
$= (x+y'+z)\,(x+y'+z')\,(x'+y'+z)\,(x'+y'+z')\,(x+y+z')\,\cancel{(x+y'+z')}$
$= M_2 \bullet M_3 \bullet M_6 \bullet M_7 \bullet M_1$
$= \Pi(1, 2, 3, 6, 7)$ product of 1-maxterms ♦

## 2.7 Canonical, Standard, and non-Standard Forms

Any Boolean function that is expressed as a sum-of-minterms, or as a product-of-maxterms is said to be in its **canonical form**. For example, the following two expressions are in their canonical forms:

$$F = x'y\,z + x\,y'z + x\,y\,z' + x\,y\,z$$

$$F' = (x+y'+z') \bullet (x'+y+z') \bullet (x'+y'+z) \bullet (x'+y'+z')$$

As noted from the previous section, to convert a Boolean function from one canonical form to its other equivalent canonical form, simply interchange the symbols Σ with Π and list the index numbers that were excluded from the original form. For example, the following two expressions are equivalent.

$$F_1(x, y, z) = \Sigma(3, 5, 6, 7)$$

$$F_2(x, y, z) = \Pi(0, 1, 2, 4)$$

To convert a Boolean function from one canonical form to its inverse, simply interchange the symbols Σ with Π and list the same index numbers from the original form. For example, the following two expressions are inverses.

$$F_1(x, y, z) = \Sigma(3, 5, 6, 7)$$

$$F_2(x, y, z) = \Pi(3, 5, 6, 7)$$

A Boolean function is said to be in a **standard form** if a sum-of-products expression or a product-of-sums expression has at least one term that is not a minterm or a maxterm, respectively. In other words, at least one term in the expression is missing at least one variable. For example, the following expression is in a standard form because the last term is missing the variable *x*.

$$F = xy'z + xyz' + yz$$

Sometimes, common variables in a standard form expression can be factored out. The resulting expression is no longer in a sum-of-products or product-of-sums format. These expressions are in a **non-standard form**. For example, starting with the previous expression, if we factor out the common variable *x* from the first two terms, we get the following expression, which is in a non-standard form.

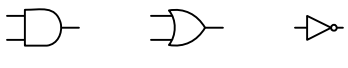
$$F = x(y'z + yz') + yz$$

## 2.8 Logic Gates and Circuit Diagrams

**Logic gates** are the actual physical implementations of the logical operators discussed in the previous sections. Transistors, acting as tiny electronic binary switches, are connected together to form these gates. Thus, we have the AND gate, the OR gate, and the NOT gate (also called the INVERTER ) for the corresponding AND, OR, and NOT logical operators. These gates form the basic building blocks for all digital logic circuits. The name "gate" comes from the fact that these devices operate like a door or gate to let or not to let things (in our case, current) through.

In drawing digital circuit diagrams (also called **schematic diagrams** or just **schematics**), we use special **logic symbols** to denote these gates, as shown in Figure 2.10. The AND gate (specifically, the 2-input AND gate) in Figure

2.10(a) has two input connections coming in from the left and one output connection going out on the right. Similarly, the 2-input OR gate in Figure 2.10(b) has two input connections and one output connection. The INVERTER in Figure 2.10(c) has one input coming from the left and one output going to the right. The outputs from these gates, of course, are dependent on their inputs and are defined by their logical functions.



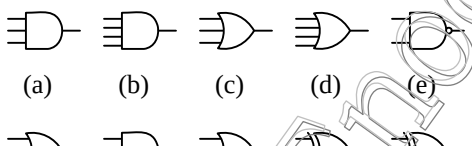
(a) (b) (c)

**Figure 2.10** Logic symbols for the three basic logic gates: (a) 2-input AND; (b) 2-input OR; (c) NOT.

Sometimes, an AND gate or an OR gate with more than two inputs are needed. Hence, in addition to the 2-input AND and OR gates, there are 3-input, 4-input, or as many inputs as are needed of the AND and OR gates. In practice, however, the number of inputs is limited to a small number, such as five. The logic symbols for some of these gates are shown in Figure 2.11(a) through (d).

There are several other gates that are variants of the three basic gates that also are used often in digital circuits. They are the NAND gate, the NOR gate, the XOR gate, and the XNOR gate. The NAND gate is derived from an AND gate and the INVERTER connected in series, so that the output of the AND gate is inverted. The name "NAND" comes from the description "NOT AND." Similarly, the NOR gate is the OR gate with its output inverted. The XOR or eXclusive OR gate is like the OR gate except that when both inputs are 1, the output is a 0 instead. The XNOR (or eXclusive NOR) gate is just the inverse of the XOR gate for when there are an even number of inputs (like two inputs). When there are an odd number of inputs (like three inputs), the XOR is the same as the XNOR. The logic symbols and their truth tables for some of these gates are shown in Figure 2.11(e) through (j) and Figure 2.12, respectively.

Notice, in Figure 2.11, the use of the little circle or bubble at the output of some of the logic symbols. This bubble is used to denote the inverted value of a signal. For example, the NAND gate is the inverse of the AND gate. Thus, the NAND gate logic symbol is the same as the AND gate logic symbol, except that it has the extra bubble at the output.



(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

**Figure 2.11** Logic symbols for: (a) 3-input AND; (b) 4-input AND; (c) 3-input OR; (d) 4-input OR; (e) 2-input NAND; (f) 2-input NOR; (g) 3-input NAND; (h) 3-input NOR; (i) 2-input XOR; (j) 2-input XNOR.

| $x$ | $y$ | 2-NAND $(x \bullet y)'$ | 2-NOR $(x+y)'$ | 2-XOR $x \oplus y$ | 2-XNOR $x \odot y$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

| $x$ | $y$ | $z$ | 3-AND $x \bullet y \bullet z$ | 3-OR $x + y + z$ | 3-NAND $(x \bullet y \bullet z)'$ | 3-NOR $(x + y + z)'$ | 3-XOR $x \oplus y \oplus z$ | 3-XNOR $x \odot y \odot z$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

**Figure 2.12** Truth tables for: 2-input NAND; 2-input NOR; 2-input XOR; 2-input XNOR; 3-input AND; 3-input OR; 3-input NAND; 3-input NOR; 3-input XOR; 3-input XNOR.

The notations used for these gates in a logical expression are $(xy)'$ for the 2-input NAND gate, $(x+y)'$ for the 2-input NOR gate, $x \oplus y$ for the 2-input XOR gate, and $x \odot y$ for the 2-input XNOR gate.

Looking at the truth table in Figure 2.12 for the 2-XOR gate, we can derive the equation for the 2-input XOR gate as

$$x \oplus y = x'y + xy'$$

Similarly, the equation for the 2-input XNOR gate as derived from the 2-XNOR truth table in Figure 2.12 is

$$x \odot y = x'y' + xy$$

The equation for the 3-input XOR gate is derived as follows

$$\begin{aligned} & x \oplus y \oplus z \\ &= (x \oplus y) \oplus z \\ &= (x'y + xy') \oplus z \\ &= (x'y + xy')z' + (x'y + xy')'z \\ &= x'yz' + xy'z' + (x'y)'(xy')'z \\ &= x'yz' + xy'z' + (x+y')(x'+y)z \\ &= x'yz' + xy'z' + \cancel{xx'z} + xyz + x'y'z + \cancel{y'yz} \\ &= x'y'z + x'yz' + xy'z' + xyz \end{aligned}$$

The last four product terms in this derivation are the four 1-minterms in the 3-input XOR truth table in Figure 2.12. For three or more inputs, the XOR gate has a value of 1 when there is an odd number of 1's in the inputs; otherwise, it is a 0.

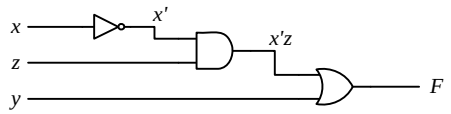
Notice also that the truth tables in Figure 2.12 for the 3-input XOR and XNOR gates are identical. It turns out that for an even number of inputs, XOR is the inverse of XNOR, but for an odd number of inputs, XOR is equal to XNOR.

All these gates can be interconnected together to form large, complex circuits which we call **networks**. These networks can be described graphically either using **circuit diagrams**, with Boolean expressions, or with truth tables.

**Example 2.9**: Drawing a circuit diagram

Draw the circuit diagram for the equation $F(x, y, z) = y + x'z$.

In the equation, we need to first invert $x$ and then AND it with $z$. Finally, we need to OR $y$ with the output of the AND. The resulting circuit is shown here. For easy reference, the internal nodes in the circuit are annotated with the two intermediate values $x'$ and $x'z$.

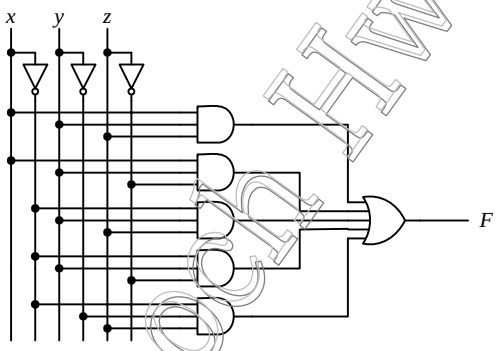



♦

**Example 2.10**: Drawing a circuit diagram

Draw the circuit diagram for the equation $F(x, y, z) = xyz + xyz' + x'yz + x'y'z'$.

The equation consists of five AND terms that are ORed together. Each AND term requires three inputs for the three variables. Hence, the circuit shown below has five 3-input AND gates, whose outputs are connected to a 5-input OR gate. The inputs to the AND gates come directly from the three variables *x*, *y*, and *z* (or their inverted values). Notice that in the equation, there are six inverted variables. However, in the circuit, we do not need six INVERTERs. Rather, only three INVERTERs are used; one for each variable.




♦

## 2.9 Designing a Car Security System

In a car security system, we usually want to connect the siren in such a way that the siren will activate when it is triggered by one or more sensors. In addition, there will be a master switch to turn the system on or off. Let us assume that there is a car door switch *D*, a vibration detector switch *V*, and a master switch *M*. We will use the convention that when the door is opened, $D = 1$, otherwise, $D = 0$. Similarly, when the car is being shaken, $V = 1$, otherwise, $V = 0$. Thus, we want the siren *S* to turn on (that is, set $S = 1$) when either $D = 1$ or $V = 1$ or when both $D = 1$ and $V = 1$, but only for when the system is turned on (that is, when $M = 1$). However, when we turn off the system and either enter or drive the car, we do not want the siren to turn on. Hence, when $M = 0$, it does not matter what values *D* and *V* have, the siren should remain off.

Given the above description of a car security system, we can build a digital circuit that meets our specifications. We start by constructing a truth table, which is basically a precise way of stating the operations for the device. The table will have three input columns *M*, *D*, and *V*, and an output column *S*, as shown in Figure 2.13(a).

Under the three input columns, we enumerate all possible binary values for the three inputs. The values under the *S* column are obtained from interpreting the description of when we want the siren to turn on. When $M = 0$, we don't want the siren to come on, regardless of what the values for *D* and *V* are. When $M = 1$, we want the siren to come on when either *D* or *V* is a 1, or both *D* and *V* are 1's.

The truth table in Figure 2.13(a) can be described formally with a logic expression written in words as

$$S = (M \text{ AND } (\text{NOT } D) \text{ AND } V) \text{ OR } (M \text{ AND } D \text{ AND } (\text{NOT } V)) \text{ OR } (M \text{ AND } D \text{ AND } V)$$

or preferably, using the simpler notation of a Boolean function:

$$S = (M\,D'\,V) + (M\,D\,V') + (M\,D\,V)$$

Again, what this equation is saying is that we want the siren to activate (i.e., $S = 1$) when:

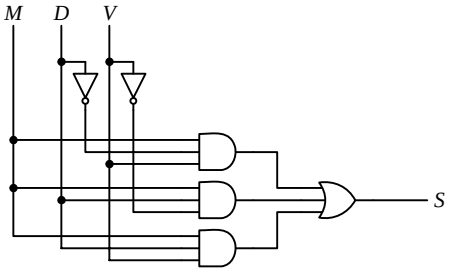
- the master switch is on and the door is not opened and the vibration switch is on, or
- the master switch is on and the door is opened and the vibration switch is not on, or
- the master switch is on and the door is opened and the vibration switch is on.

Notice that we are only interested in the situations when $S = 1$. We ignore the rows when $S = 0$. When we construct circuits from truth tables, we always use only the rows where the output is a 1.

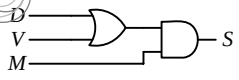
Finally, we can translate this equation into a circuit diagram. The translation is a simple one-to-one mapping of changing the AND operator into the AND gate, the OR operator into the OR gate, and the NOT operator into the INVERTER. Thus, we get the circuit diagram shown in Figure 2.13(b) for our car security system.

| *M* | *D* | *V* | *S* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

(a)




(b)




(c)

**Figure 2.13** Car security system: (a) truth table; (b) circuit diagram derived from the truth table; (c) simplified circuit diagram.

A careful reader might notice that the Boolean equation shown above for specifying when the siren is to be turned on can be simplified to

$$S = M\,(D + V)$$

This simplified equation says that the siren is to be turned on only when the master switch is on and either the door switch or vibration switch is on. The corresponding simplified circuit diagram is shown in Figure 2.13(c). Just by using simple reasoning, we can see that this simplified circuit will do exactly what the circuit in Figure 2.13(b) does. In other words, both circuits are functionally equivalent.

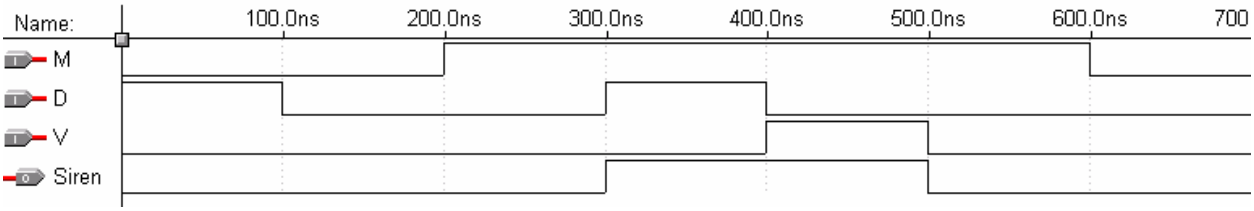
More formally, we can use the Boolean theorems from Section 2.5.1 to show that these two equations (and therefore, the two circuits) are indeed equivalent as follows:

$S = (M\,D'\,V) + (M\,D\,V') + (M\,D\,V)$
$= M\,(D'\,V + D\,V' + D\,V)$ by Distributive Theorem 12a
$= M\,(D'\,V + D\,V' + D\,V + DV)$ by Idempotent Theorem 7b
$= M\,(\,D(V' + V) + V(D' + D)\,)$ by Distributive Theorem 12a
$= M\,(\,D(1) + V(1))$ by Inverse Theorem 9b
$= M\,(D + V)$ by Identity Theorem 6a

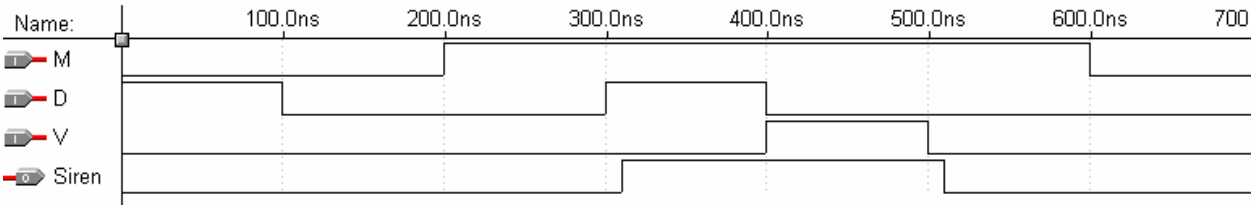
Figure 2.14(a) shows a sample simulation trace of the car security system circuit. Between times 0 and 200 ns, the master switch *M* is a 0, so regardless of the values of *D* and *V*, the siren is off (*Siren* = 0). Between times 200 ns and 600 ns, *M* = 1. During this time, whenever either *D* = 1 or *V* = 1, the siren is on.

Figure 2.14(a) is a **functional** trace of the circuit, so all the signal edges line up exactly, i.e., the output signal edge changes at exactly the same time (with no delay) as the input edge that caused it to change. For a **timing** trace,

on the other hand, the output signal edge will be delayed slightly after the causing input edge, as shown in Figure 2.14(b).




(a)




(b)

**Figure 2.14** Sample simulation trace of the car security system circuit: (a) functional trace; (b) timing trace.

When building circuits, besides having a functionally correct circuit, we also want to optimize it in terms of its size, speed, heat dissipation, and power consumption. We will see in later sections how circuits are optimized.

## *2.10 VHDL for Digital Circuits*

A digital circuit that is described with a Boolean function can easily be converted to VHDL code using the dataflow model. At the dataflow level, a circuit is defined using built-in VHDL logic operators such as AND, OR, and NOT. These operators are applied to signals using concurrent signal assignment statements.

### 2.10.1 VHDL code for a 2-input NAND gate

Figure 2.15 shows the VHDL code for a 2-input NAND gate. It also serves as a basic template for all VHDL codes. Lines starting with two hyphens are comments. The LIBRARY and USE statements specify that the IEEE library is needed and that all of the components in that library package can be used. These two statements are equivalent to the "#include" preprocessor line in C.

Every component defined in VHDL, whether it is a simple NAND gate or a complex microprocessor, has two parts: an ENTITY section and an ARCHITECTURE section. The ENTITY section is similar to a function declaration in C and serves as the interface between the component and the outside. It declares all of the input and output signals for a circuit. Every entity must have a unique name; in this example, the name *NAND2gate* is used. The entity contains a PORT list, which, like a parameter list, specifies the data to be passed in and out of the component. In this example, there are two input signals called *x* and *y* of type STD_LOGIC and an output signal called *f* of the same type. The STD_LOGIC type is like the BIT type, except that it contains additional values besides just 0's and 1's.

The ARCHITECTURE section defines the operation of the entity; it contains the code that realizes the operation of the component. For every architecture, you need to specify its name and which entity it is for. In this example, the name is *Dataflow*, and it is for the entity *NAND2gate*. It is possible for one entity to have more than one architecture, since an entity can be implemented in more than one way. Within the body of the architecture, we can have one or more concurrent statements. Unlike statements in C where they are executed in sequential order, concurrent statements in the architecture body are executed in parallel. Thus, the ordering of these statements is irrelevant. The symbol "<=" is used for a signal assignment statement. The expression on the right-hand side of the <= symbol is evaluated when either *x* or *y* changes values (either from 0 to 1 or from 1 to 0), and the result is assigned to the signal on the left-hand side. The NAND operator is a built-in VHDL operator.

```
-- this is a dataflow model of a 2-input NAND gate
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY NAND2gate IS PORT (
  x: IN STD_LOGIC;
  y: IN STD_LOGIC;
  f: OUT STD_LOGIC);
END NAND2gate;

ARCHITECTURE Dataflow OF NAND2gate IS
BEGIN
  f <= x NAND y;       -- signal assignment
END Dataflow;
```

**Figure 2.15** VHDL code for a 2-input NAND gate.

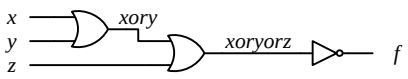
## 2.10.2 VHDL code for a 3-input NOR gate

Figure 2.16(a) shows the VHDL code for a 3-input NOR gate. In addition to the three input signals *x*, *y*, and *z*, and one output signal *f* declared in the entity section, this example has two internal signals, *xory* and *xoryorz*, both of which are of the STD_LOGIC type. The keyword SIGNAL in the architecture section is used to declare these two internal signals. Internal signals are used for naming connection points (or nodes) within a circuit. Three concurrent signal assignment statements are used. All the signal assignment statements are executed concurrently, so the ordering of the statements is irrelevant. The coding of these three signal assignment statements is based on the 3-input NOR gate circuit shown in Figure 2.16(b).

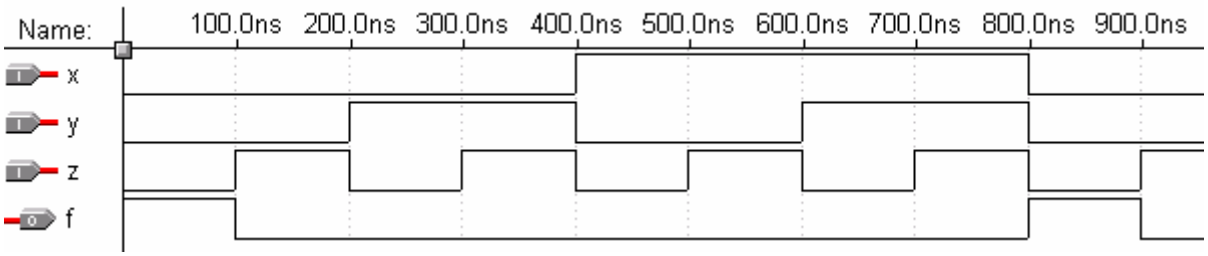
Figure 2.16(c) shows a sample simulation trace of the circuit. In the trace, we see that the output signal *f* is 1 only when all three inputs are 0's. This occurs twice: the first time between 0 and 100 ns, and the second time between 800 ns and 900 ns. For all of the other times, *f* is 0, since not all three inputs are 0's. Hence, the simulation trace shows the correct operation of this circuit for the 3-input NOR gate.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY NOR3gate IS PORT (
  x: IN STD_LOGIC;
  y: IN STD_LOGIC;
  z: IN STD_LOGIC;
  f: OUT STD_LOGIC);
END NOR3gate;

ARCHITECTURE Dataflow OF NOR3gate IS
  SIGNAL xory, xoryorz : STD_LOGIC;
BEGIN
  xory <= x OR y;         -- three concurrent signal assignments
  xoryorz <= xory OR z;
  f <= NOT xoryorz;
END Dataflow;
```

(a)



(b)




(c)

**Figure 2.16** 3-input NOR gate: (a) VHDL code; (b) circuit; (c) simulation trace.

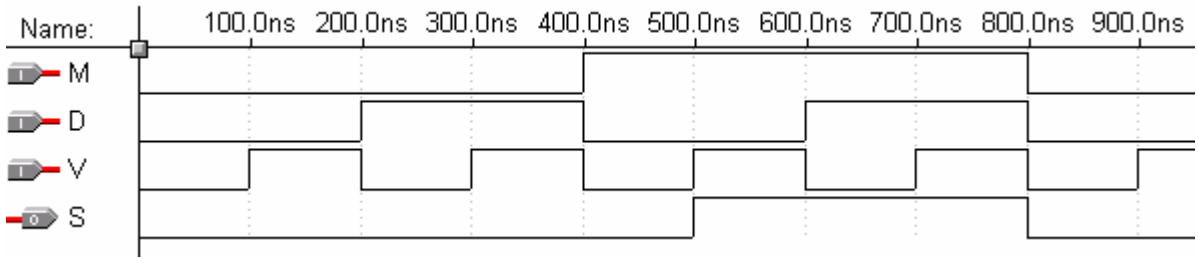
## 2.10.3 VHDL code for a function

Figure 2.17 shows the VHDL code and the simulation trace for the car security system circuit discussed in Section 2.9. The function implemented is $S = (M\,D'\,V) + (M\,D\,V') + (M\,D\,V)$.

This VHDL code (as well as the ones from the two previous sections) is written at the dataflow level. This is not because the name of the architecture is "Dataflow." Dataflow level coding uses logic equations to describe a circuit, and this is done by using the built-in VHDL operators such as AND, OR, and NOT in concurrent signal assignment statements.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY Siren IS PORT (
  M: IN STD_LOGIC;
  D: IN STD_LOGIC;
  V: IN STD_LOGIC;
  S: OUT STD_LOGIC);
END Siren;

ARCHITECTURE Dataflow OF Siren IS
  SIGNAL term_1, term_2, term_3: STD_LOGIC;
BEGIN
  term_1 <= M AND (NOT D) AND V;
  term_2 <= M AND D AND (NOT V);
  term_3 <= M AND D AND V;
  S <= term_1 OR term_2 OR term_3;
END Dataflow;
```

(a)



(b)

**Figure 2.17** The car security system circuit of Section 2.9: (a) VHDL code; (b) simulation trace.

## *2.11 Summary Checklist*

- Binary number
- Hexadecimal number
- Binary switch
- AND, OR, and NOT
- Truth table
- Boolean algebra axioms and theorems
- Duality principle
- Boolean function and the inverse
- Product term
- Sum term
- Sum-of-products (or-of-ands)
- Product of sums (and-of-ors)
- Minterm and maxterm
- Sum-of minterms
- Product-of-maxterms
- Canonical, standard, and non-standard form
- Logic gate, logic symbol
- Circuit diagram
- NAND, NOR, XOR, XNOR
- Network
- VHDL
- Be able to derive the Boolean equation from a truth table (or vice versa)
- Be able to derive the circuit diagram from a Boolean equation (or vice versa)
- Be able to derive the circuit diagram from a truth table (or vice versa)
- Be able to use Boolean algebra to reduce a Boolean equation

## 2.12 Problems

2.1 Convert the following decimal numbers to binary numbers.
a) 66
b) 49
c) 513
d) 864
e) 1897
f) 2004

2.2 Convert the following unsigned binary numbers to decimal, hexadecimal and octal numbers.
a) 11110
b) 11010
c) 100100011
d) 1011011
e) 1101101110
f) 101111010100

2.3 Convert the following hexadecimal numbers to binary numbers.
a) 66
b) E3
c) 2FE8
d) 7C2
e) 5A2D
f) E08B

2.4 Derive the truth table for the following Boolean functions.
a) $F(x,y,z) = x'y'z' + x'yz + xy'z' + xyz$
b) $F(x,y,z) = xy'z + x'yz' + xyz + xyz'$
c) $F(w,x,y,z) = w'xy'z + w'xyz + wxy'z + wxyz$
d) $F(w,x,y,z) = wxy'z + w'yz' + wxz + xyz'$
e) $F(x,y,z) = xy' + x'y'z + xyz'$
f) $F(w,x,y,z) = w'z' + w'xy + wx'z + wxyz$
g) $F(x,y,z) = [(x+y')\,(yz)']\,(xy' + x'y)$
h) $F(N_3,N_2,N_1,N_0) = N_3'N_2'N_1N_0' + N_3'N_2'N_1N_0 + N_3N_2'N_1N_0' + N_3N_2'N_1N_0 + N_3N_2N_1'N_0' + N_3N_2N_1N_0$

2.5 Derive the Boolean function for the following truth tables.

a)

| *a* | *b* | *c* | *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 | 1 |
| 1 | 1 | 1 | 0 |

b)

| *w* | *x* | *y* | *z* | *F* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 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 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

c)

| $w$ | $x$ | $y$ | $z$ | $F_1$ | $F_2$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

d)

| $N_3$ | $N_2$ | $N_1$ | $N_0$ | $F$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

2.6 Use a truth table to show that the following expressions are true.

a) $w'z' + w'xy + wx'z + wxyz = w'z' + xyz + wx'y'z + wyz$
b) $z + y' + yz' = 1$
c) $xy'z' + x' + xyz' = x' + z'$
d) $xy + x'z + yz = xy + x'z$
e) $w'x'yz' + w'x'yz + wx'yz' + wx'yz + wxyz = y(x' + wz)$
f) $w'xy'z + w'xyz + wxy'z + wxyz = xz$
g) $x_iy_i + c_i(x_i + y_i) = x_iy_ic_i + x_iy_ic_i' + x_iy_i'c_i + x_i'y_ic_i$
h) $x_iy_i + c_i(x_i + y_i) = x_iy_i + c_i(x_i \oplus y_i)$

2.7 Use Boolean algebra to show that the expressions in Problem 2.6 are true.

2.8 Use Boolean algebra to reduce the functions in Problem 2.4 as much as possible.

2.9 Any function can be implemented directly either as specified or as its inverted form with a NOT gate added at the final output. Assume that the circuit size is proportional to only the number of AND gates and OR gates (i.e., ignore the number of NOT gates in determining the circuit size). Determine which form of the function (the inverted or non-inverted) will result in a smaller circuit size for the following function. Give your reason, and specify how many AND and OR gates are needed to implement the smaller circuit.

$$F(x,y,z) = x'y'z' + x'y'z + xy'z + xy'z' + xyz$$

2.10 Derive the truth table for the following logic gates.

a) A 4-input AND gate.
b) A 4-input NAND gate.
c) A 4-input NOR gate.
d) A 4-input XOR gate.
e) A 4-input XNOR gate.
f) A 5-input XOR gate.
g) A 5-input XNOR gate.

2.11 Derive the truth table for the following Boolean functions.
- a) $F(w,x,y,z) = [(x \odot y)' + (xyz)'] (w' + x + z)$
- b) $F(x,y,z) = x \oplus y \oplus z$
- c) $F(w,x,y,z) = [w'xy'z + w'z (y \oplus x)]'$

2.12 Use Boolean algebra to convert the functions in Problem 2.11 to:
- a) The sum-of-minterms format
- b) The product-of-maxterms format

2.13 Use Boolean algebra to reduce the functions in Problem 2.11 as much as possible.

2.14 Use a truth table to show that the following expressions are true.
- a) $(x \oplus y) = (x \odot y)'$
- b) $x \oplus y' = x \odot y$
- c) $(w \oplus x) \odot (y \oplus z) = (w \odot x) \odot (y \odot z) = (((w \odot x) \odot y) \odot z)$.
- d) $[((xy)'x)' ((xy)'y)' ]' = x \oplus y$

2.15 Use Boolean algebra to show that the expressions in Question 2.14 are true.

2.16 Use Boolean algebra to show that XOR = XNOR for three inputs.

2.17 Express the Boolean functions in Problem 2.4 using:
- a) The Σ notation
- b) The Π notation

2.18 Write the following expression as a Boolean function in the canonical form.
- a) $F(x, y, z) = \Sigma(1, 3, 7)$
- b) $F(w, x, y, z) = \Sigma(1, 3, 7)$
- c) $F(x, y, z) = \Pi(1, 3, 7)$
- d) $F(w, x, y, z) = \Pi(1, 3, 7)$
- e) $F' (x, y, z) = \Sigma(1, 3, 7)$
- f) $F' (x, y, z) = \Pi(1, 3, 7)$

2.19 Given $F' (x, y, z) = \Sigma(1, 3, 7)$, express the function $F$ using a truth table.

2.20 Use Boolean algebra to convert the function $F(x, y, z) = \Sigma(3, 4, 5)$ to its equivalent product-of-sums canonical form.

2.21 Given $F = xy'z' + xy'z + xyz' + xyz$, write the expression for $F'$ using:
- a) The product-of-sums format
- b) The sum-of-products format

2.22 Use Boolean algebra to convert the equation $F = w \odot x \odot y \odot z$ to:
- a) The sum-of-minterms format
- b) The product-of-maxterms format

2.23 Write the complete dataflow VHDL code for the Boolean functions in Problem 2.4.

2.24 Write the complete dataflow VHDL code for the logic gates in Problem 2.10.

2.25 Write the complete dataflow VHDL code for the Boolean functions in Problem 2.11.

## *Index*