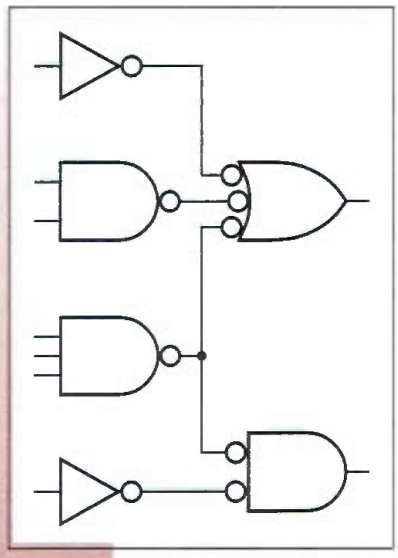

chapter 4

Combinational Logic Design Principles

Logic circuits are classified into two types, "combinational" and "sequential." A *combinational logic circuit* is one whose outputs depend only on its current inputs. The rotary channel selector knob on an old-fashioned television is like a combinational circuit—its "output" selects a channel based only on the current position of the knob ("input").

The outputs of a *sequential logic circuit* depend not only on the current inputs but also on the past sequence of inputs, possibly arbitrarily far back in time. The channel selector controlled by the up and down pushbuttons on a TV remote control is a sequential circuit—the channel selection depends on the past sequence of up/down pushes, at least since when you started viewing 10 hours before, and perhaps as far back as when you first powered up the device. Sequential circuits are discussed in Chapters 7 and 8.

A combinational circuit may contain an arbitrary number of logic gates and inverters but no feedback loops. A *feedback loop* is a signal path of a circuit that allows the output of a gate to propagate back to the input of that same gate; such a loop generally creates sequential circuit behavior.

In combinational circuit *analysis* we start with a logic diagram and proceed to a formal description of the function performed by that circuit, such as a truth table or a logic expression. In *synthesis* we do the reverse, starting with a formal description and proceeding to a logic diagram.

SYNTHESIS VS. DESIGN

Logic circuit design is a superset of synthesis, since in a real design problem we usually start out with an informal (word or thought) description of the circuit. Often the most challenging and creative part of design is to formalize the circuit description, defining the circuit's input and output signals and specifying its functional behavior by means of truth tables and equations. Once we've created the formal circuit description, we can usually follow a "turn-the-crank" synthesis procedure to obtain a logic diagram for a circuit with the required functional behavior. The material in the first four sections of this chapter is the basis for "turn-the-crank" procedures, whether the crank is turned by hand or by a computer.

The next chapter describes hardware description languages—ABEL, VHDL and Verilog. When we create a design using one of these languages, a computer program can perform the synthesis steps for us. In later chapters we'll encounter many examples of the real design process.

Combinational circuits may have one or more outputs. In this chapter, we'll discuss methods that apply to single-output circuits. Most analysis and synthesis techniques can be extended in an obvious way from single-output to multiple-output circuits (e.g., "Repeat these steps for each output"). Some techniques can be extended in a not-so-obvious way for improved effectiveness in the multiple-output case.

The purpose of this chapter is to give you a solid theoretical foundation for the analysis and synthesis of combinational logic circuits, a foundation that will be doubly important later when we move on to sequential circuits. Although most of the analysis and synthesis procedures in this chapter are automated nowadays by computer-aided design tools, you need a basic understanding of the fundamentals to use the tools and to figure out what's wrong when you get unexpected or undesirable results.

Before launching into a discussion of combinational logic circuits, we must introduce switching algebra, the fundamental mathematical tool for analyzing and synthesizing logic circuits of all types.

4.1 Switching Algebra

Boolean algebra

Formal analysis techniques for digital circuits have their roots in the work of an English mathematician, George Boole. In 1854, he invented a two-valued algebraic system, now called *Boolean algebra*, to "give expression ... to the fundamental laws of reasoning in the symbolic language of a Calculus." Using this system, a philosopher, logician, or inhabitant of the planet Vulcan can formulate propositions that are true or false, combine them to make new propositions, and determine the truth or falsehood of the new propositions. For

example, if we agree that "People who haven't studied this material are either failures or not nerds," and "No computer designer is a failure," then we can answer questions like "If you're a nerdy computer designer, then have you already studied this?"

Long after Boole, in 1938, Bell Labs researcher Claude E. Shannon showed how to adapt Boolean algebra to analyze and describe the behavior of circuits built from relays, the most commonly used digital logic elements of that time. In Shannon's *switching algebra*, the condition of a relay contact, open or closed, is represented by a variable X that can have one of two possible values, 0 or 1. In today's logic technologies, these values correspond to a wide variety of physical conditions—voltage HIGH or LOW, light off or on, capacitor discharged or charged, fuse blown or intact, and so on—as we detailed in Table 3-1 on page 81.

switching algebra

In the remainder of this section we develop the switching algebra directly, using "first principles" and what we already know about the behavior of logic elements (gates and inverters). For more historical and/or mathematical treatments of this material, consult the References section of this chapter.

4.1.1 Axioms

In switching algebra we use a symbolic variable, such as X, to represent the condition of a logic signal. A logic signal is in one of two possible conditions—low or high, off or on, and so on, depending on the technology. We say that X has the value "0" for one of these conditions and "1" for the other.

For example, with the CMOS and TTL logic circuits in Chapter 3, the *positive-logic convention* dictates that we associate the value "0" with a LOW voltage and "1" with a HIGH voltage. The *negative-logic convention* makes the opposite association: 0 = HIGH and 1 = LOW. However, the choice of positive or negative logic has no effect on our ability to develop a consistent algebraic description of circuit behavior; it only affects details of the physical-to-algebraic abstraction, as we'll explain later in our discussion of "duality." For the moment, we may ignore the physical realities of logic circuits and pretend that they operate directly on the logic symbols 0 and 1.

positive-logic convention

negative-logic convention

The *axioms* (or *postulates*) of a mathematical system are a minimal set of basic definitions that we assume to be true, from which all other information about the system can be derived. The first two axioms of switching algebra embody the "digital abstraction" by formally stating that a variable X can take on only one of two values:

axiom

postulate

$$\text{(A1)} \quad X = 0 \quad \text{if } X \neq 1 \qquad \text{(A1')} \quad X = 1 \quad \text{if } X \neq 0$$

Notice that we stated these axioms as a pair, the only difference between A1 and A1′ being the interchange of the symbols 0 and 1. This is a characteristic of all

the axioms of switching algebra and is the basis of the "duality" principle that we'll study later.

complement
prime (′)

In Section 3.3.3 we showed the design of an inverter, a logic circuit whose output signal level is the opposite (or *complement*) of its input signal level. We use a *prime* (′) to denote an inverter's function. That is, if a variable X denotes an inverter's input signal, then X′ denotes the value of a signal on the inverter's output. This notation is formally specified in the second pair of axioms:

(A2) If X = 0, then X′ = 1 (A2′) If X = 1, then X′ = 0

algebraic operator
expression
NOT operation

As shown in Figure 4-1, the output of an inverter with input signal X may have an arbitrary signal name, say Y. Algebraically, however, we write Y = X′ to say "signal Y always has the opposite value as signal X." The prime (′) is an *algebraic operator*, and X′ is an *expression*, which you can read as "X prime" or "NOT X." This usage is analogous to what you've learned in programming languages, where if J is an integer variable, then –J is an expression whose value is 0 – J. Although this may seem like a small point, you'll learn that the distinction between signal names (X, Y), expressions (X′), and equations (Y = X′) is very important when we study documentation standards and software tools for logic design. In the logic diagrams in this book we maintain this distinction by writing signal names in black and expressions in color.

Figure 4-1
Signal naming and algebraic notation for an inverter.

logical multiplication
multiplication dot (·)

In Section 3.3.6 we showed how to build a 2-input CMOS AND gate, a circuit whose output is 1 if both of its inputs are 1. The function of a 2-input AND gate is sometimes called *logical multiplication* and is symbolized algebraically by a *multiplication dot* (·). That is, an AND gate with inputs X and Y has an output signal whose value is X · Y, as shown in Figure 4-2(a). Some authors, especially mathematicians and logicians, denote logical multiplication with a wedge (X ∧ Y). We follow standard engineering practice by using the dot (X · Y).

NOTE ON NOTATION

The notations $\overline{X}$, ~X, and ¬X are also used by some authors to denote the complement of X. The overbar notation ($\overline{X}$) is probably the most widely used and the best looking typographically. However, we use the prime notation to get you used to writing logic expressions on a single text line without the more graphical overbar, and to force you to parenthesize complex complemented subexpressions—because that's what you'll have to do when you use HDLs and other tools.

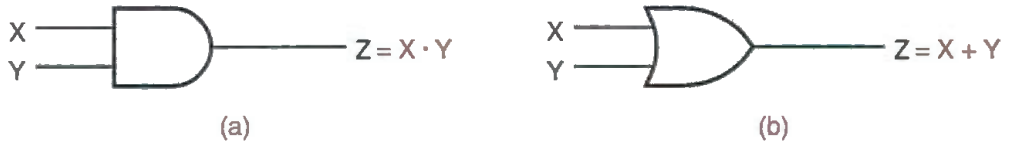

Figure 4-2
Signal naming and algebraic notation: (a) AND gate; (b) OR gate.

Among hardware description languages (HDLs), Verilog uses an ampersand (&) to denote the same thing, while VHDL forces you to write out "and".

We also described in Section 3.3.6 how to build a 2-input CMOS OR gate, a circuit whose output is 1 if either of its inputs is 1. The function of a 2-input OR gate is sometimes called *logical addition* and is symbolized algebraically by a plus sign (+). An OR gate with inputs X and Y has an output signal whose value is X + Y, as shown in Figure 4-2(b). Some authors denote logical addition with a vee ($X \vee Y$), but we follow the typical engineering practice of using the plus sign (X + Y). Once again, other notations may be used in HDLs, such as "|" in Verilog and "or" in VHDL. By convention, in a logic expression involving both multiplication and addition, multiplication has *precedence*, just as in integer expressions in conventional programming languages. That is, the expression $W \cdot X + Y \cdot Z$ is equivalent to $(W \cdot X) + (Y \cdot Z)$.

logical addition

precedence

The last three pairs of axioms state the formal definitions of the AND and OR operations by listing the output produced by each gate for each possible input combination:

AND operation
OR operation

(A3)	$0 \cdot 0 = 0$	(A3′)	$1 + 1 = 1$
(A4)	$1 \cdot 1 = 1$	(A4′)	$0 + 0 = 0$
(A5)	$0 \cdot 1 = 1 \cdot 0 = 0$	(A5′)	$1 + 0 = 0 + 1 = 1$

The five pairs of axioms, A1–A5 and A1′–A5′, completely define switching algebra. All other facts about the system can be proved using these axioms as a starting point.

JUXT A MINUTE... Older texts use simple *juxtaposition* (XY) to denote logical multiplication, but we don't. In general, juxtaposition is a clear notation only when signal names are limited to a single character. Otherwise, is XY a logical product or a two-character signal name? One-character variable names are common in algebra, but in real digital design problems we prefer to use multicharacter signal names that mean something. Thus, we need a separator between names, and the separator might just as well be a multiplication dot rather than a space. The HDL equivalent of the multiplication dot (such as "&" and " and " in Verilog and VHDL, respectively) is absolutely required when logic formulas are written in a hardware-description language.

Table 4-1 Switching-algebra theorems with one variable.

(T1)	$X + 0 = X$	(T1′)	$X \cdot 1 = X$	(Identities)
(T2)	$X + 1 = 1$	(T2′)	$X \cdot 0 = 0$	(Null elements)
(T3)	$X + X = X$	(T3′)	$X \cdot X = X$	(Idempotency)
(T4)	$(X')' = X$			(Involution)
(T5)	$X + X' = 1$	(T5′)	$X \cdot X' = 0$	(Complements)

4.1.2 Single-Variable Theorems

During the analysis or synthesis of logic circuits, we often write algebraic expressions that characterize a circuit's actual or desired behavior. Switching-algebra *theorems* are statements, known to be always true, that allow us to manipulate algebraic expressions to allow simpler analysis or more efficient synthesis of the corresponding circuits. For example, the theorem $X + 0 = X$ allows us to substitute every occurrence of $X + 0$ in an expression with X.

theorem

Table 4-1 lists switching-algebra theorems involving a single variable X. How do we know that these theorems are true? We can either prove them ourselves or take the word of someone who has. OK, we're in college now, let's learn how to prove them.

Most theorems in switching algebra are exceedingly simple to prove using a technique called *perfect induction*. Axiom A1 is the key to this technique—since a switching variable can take on only two different values, 0 and 1, we can prove a theorem involving a single variable X by proving that it is true for both $X = 0$ and $X = 1$. For example, to prove theorem T1, we make two substitutions:

perfect induction

$[X = 0]$	$0 + 0 = 0$	true, according to axiom A4′
$[X = 1]$	$1 + 0 = 1$	true, according to axiom A5′

All of the theorems in Table 4-1 can be proved using perfect induction, as you're asked to do in the Drills 4.2 and 4.3.

4.1.3 Two- and Three-Variable Theorems

Switching-algebra theorems with two or three variables are listed in Table 4-2. Each of these theorems is easily proved by perfect induction, by evaluating the theorem statement for the four possible combinations of X and Y, or the eight possible combinations of X, Y, and Z.

The first two theorem pairs concern commutativity and associativity of logical addition and multiplication and are identical to the commutative and associative laws for addition and multiplication of integers and reals. Taken together, they indicate that the parenthesization or order of terms in a logical sum or logical product is irrelevant. For example, from a strictly algebraic point of view, an expression such as $W \cdot X \cdot Y \cdot Z$ is ambiguous; it should be written as $(W \cdot (X \cdot (Y \cdot Z)))$ or $(((W \cdot X) \cdot Y) \cdot Z)$ or $(W \cdot X) \cdot (Y \cdot Z)$ (see Exercise 4.26).

Table 4-2 Switching-algebra theorems with two or three variables.

(T6)	$X + Y = Y + X$	(T6′)	$X \cdot Y = Y \cdot X$	(Commutativity)
(T7)	$(X + Y) + Z = X + (Y + Z)$	(T7′)	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	(Associativity)
(T8)	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$	(T8′)	$(X + Y) \cdot (X + Z) = X + Y \cdot Z$	(Distributivity)
(T9)	$X + X \cdot Y = X$	(T9′)	$X \cdot (X + Y) = X$	(Covering)
(T10)	$X \cdot Y + X \cdot Y' = X$	(T10′)	$(X + Y) \cdot (X + Y') = X$	(Combining)
(T11)	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$			(Consensus)
(T11′)	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$			

But the theorems tell us that the ambiguous form of the expression is OK because we get the same results in any case. We even could have rearranged the order of the variables (e.g., $X \cdot Z \cdot Y \cdot W$) and gotten the same results.

As trivial as this discussion may seem, it is very important, because it forms the theoretical basis for using logic gates with more than two inputs. We defined $\cdot$ and $+$ as *binary operators*—operators that combine *two* variables. Yet we use 3-input, 4-input, and larger AND and OR gates in practice. The theorems tell us we can connect gate inputs in any order; in fact, many printed-circuit-board and ASIC layout programs take advantage of this to optimize wiring. We can use either one n-input gate or $(n-1)$ 2-input gates interchangeably, though propagation delay and cost are likely to be higher with multiple 2-input gates.

binary operator

Theorem T8 is identical to the distributive law for integers and reals—that is, logical multiplication distributes over logical addition. Hence, we can "multiply out" an expression to obtain a sum-of-products form, as in the example below:

$$V \cdot (W + X) \cdot (Y + Z) = V \cdot W \cdot Y + V \cdot W \cdot Z + V \cdot X \cdot Y + V \cdot X \cdot Z$$

However, switching algebra also has the unfamiliar property that the reverse is true—logical addition distributes over logical multiplication—as demonstrated by theorem T8′. Thus, we can also "add out" an expression to obtain a product-of-sums form:

$$(V \cdot W \cdot X) + (Y \cdot Z) = (V + Y) \cdot (V + Z) \cdot (W + Y) \cdot (W + Z) \cdot (X + Y) \cdot (X + Z)$$

Theorems T9 and T10 are used extensively in the minimization of logic functions. For example, if the subexpression $X + X \cdot Y$ appears in a logic expression, the *covering theorem* T9 says that we need only include X in the expression; X is said to *cover* $X \cdot Y$. The *combining theorem* T10 says that if the subexpression $X \cdot Y + X \cdot Y'$ appears in an expression, we can replace it with X. Since Y must be 0 or 1, either way the original subexpression is 1 if and only if X is 1.

covering theorem
cover
combining theorem

Although we could easily prove T9 by perfect induction, the truth of T9 is more obvious if we prove it using the other theorems that we've proved so far:

$$\begin{aligned} X + X \cdot Y &= X \cdot 1 + X \cdot Y && \text{(according to T1′)} \\ &= X \cdot (1 + Y) && \text{(according to T8)} \\ &= X \cdot 1 && \text{(according to T2)} \\ &= X && \text{(according to T1′)} \end{aligned}$$

Likewise, the other theorems can be used to prove T10, where the key step is to use T8 to rewrite the lefthand side as $X \cdot (Y + Y')$.

consensus theorem
consensus

Theorem T11 is known as the *consensus theorem*. The $Y \cdot Z$ term is called the *consensus* of $X \cdot Y$ and $X' \cdot Z$. The idea is that if $Y \cdot Z$ is 1, then either $X \cdot Y$ or $X' \cdot Z$ must also be 1, since Y and Z are both 1 and either X or X′ must be 1. Thus, the $Y \cdot Z$ term is redundant and may be dropped from the righthand side of T11. The consensus theorem has two important applications. It can be used to eliminate certain timing hazards in combinational logic circuits, as we'll see in Section 4.4. And it also forms the basis of the iterative-consensus method of finding prime implicants (see Section ProgMin at DDPPonline).

In all of the theorems, it is possible to replace each variable with an arbitrary logic expression. A simple replacement is to complement one or more variables:

$$(X + Y') + Z' = X + (Y' + Z') \quad \text{(based on T7)}$$

But more complex expressions may be substituted as well:

$$(V' + X) \cdot (W \cdot (Y' + Z)) + (V' + X) \cdot (W \cdot (Y' + Z))' = V' + X \quad \text{(based on T10)}$$

4.1.4 *n*-Variable Theorems

Several important theorems, listed in Table 4-3, are true for an arbitrary number of variables, *n*. Most of these theorems can be proved using a two-step method called *finite induction*—first proving that the theorem is true for $n = 2$ (the *basis step*) and then proving that if the theorem is true for $n = i$, then it is also true for $n = i + 1$ (the *induction step*). For example, consider the generalized idempotency theorem T12. For $n = 2$, T12 is equivalent to T3 and is therefore true. If it is true for a logical sum of *i* X's, then it is also true for a sum of $i + 1$ X's, according to the following reasoning:

finite induction
basis step
induction step

$$\begin{aligned} X + X + X + \cdots + X &= X + (X + X + \cdots + X) && (i + 1 \text{ X's on either side}) \\ &= X + (X) && (\text{if T12 is true for } n = i) \\ &= X && \text{(according to T3)} \end{aligned}$$

Thus, the theorem is true for all finite values of *n*.

DeMorgan's theorems

DeMorgan's theorems (T13 and T13′) are probably the most commonly used of all the theorems of switching algebra. Theorem T13 says that an *n*-input