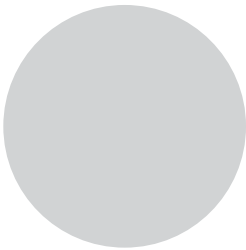# 4

BOOLEAN ALGEBRA

Boolean algebra was developed in the 19th century by an English mathematician, George Boole, who was working on ways to use mathematical rigor to solve logic problems. He formalized a mathematical system for manipulating logical values in which the only possible values for the variables are *true* and *false*, usually designated 1 and 0, respectively. The basic operations in Boolean algebra are *conjunction* (AND), *disjunction* (OR), and *negation* (NOT). This distinguishes it from elementary algebra, which includes the infinite set of real numbers and uses the arithmetic operations addition, subtraction, multiplication, and division. (Exponentiation is a simplified notation for repeated multiplication.)

As mathematicians and logicians were expanding the field of Boolean algebra in increasingly complex and abstract ways, engineers were learning to harness electrical flows using switches in circuits to perform logic operations. The two fields developed in parallel until the mid-1930s, when a graduate student named Claude Shannon proved that electrical switches could be used to implement the full range of Boolean algebraic expressions. (When used to describe switching circuits, Boolean algebra is sometimes called *switching algebra*.) With Shannon's discovery a world of possibilities was opened, and Boolean algebra became the mathematical foundation of the computer.

This chapter will start with descriptions of the basic Boolean operators. Then you'll learn about their logical rules, which form the basis of Boolean algebra. Next, I'll explain ways to combine Boolean variables and operators into algebraic expressions to form Boolean logic functions. Finally, I'll discuss techniques for simplifying Boolean functions. In subsequent chapters, you'll learn how electronic on/off switches can be used to implement logic functions that can be connected together in logic circuits to perform the primary functions of a computer—arithmetic and logic operations and memory storage.

Basic Boolean Operators

There are several symbols used to denote each Boolean operator, which I'll include in the description of each of the operators. In this book, I'll present the symbols used by logicians. A Boolean operator acts on a value, or pair of values, called the *operands*.

I'll use *truth tables* to show the results of each operation. A truth table shows the results for all possible combinations of the operands. For example, consider the addition of two bits, x and y. There are four possible combinations of the values. Addition will give a sum and a possible carry. Table 4-1 shows how to express this in a truth table.

Table 4-1: Truth Table Showing Addition of Two Bits

x	y	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

I'll also provide the electronic circuit representations for the *gates*, the electronic devices that implement the Boolean operators. You'll learn more

about these devices in Chapters 5 through 8, where you'll also see that the real-world behavior of the physical devices varies slightly from the ideal mathematical behavior shown in the truth tables.

As with elementary algebra, you can combine these basic operators to define secondary operators. You'll see an example of this when we define the XOR operator near the end of this chapter.

AND

AND is a *binary operator*, meaning it acts on two operands. The result of AND is 1 if and only if *both* operands are 1; otherwise, the result is 0. In logic, the operation is known as *conjunction*. I'll use ∧ to designate the AND operation. It's also common to use the · symbol or simply AND. Figure 4-1 shows the circuit symbol for an AND gate and a truth table defining the output, with operands *x* and *y*.

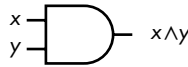| *x*	*y*	*x* ∧ *y*
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4-1: The AND gate acting on two variables, x and y

As you can see from the truth table, the AND operator has properties similar to multiplication in elementary algebra, which is why some use the · symbol to represent it.

OR

OR is also a binary operator. The result of OR is 1 if at least one of the operands is 1; otherwise, the result is 0. In logic, the operation is known as *disjunction*. I'll use ∨ to designate the OR operation. It's also common to use the + symbol or simply OR. Figure 4-2 shows the circuit symbol for an OR gate and a truth table defining the output, with operands *x* and *y*.

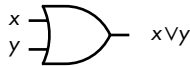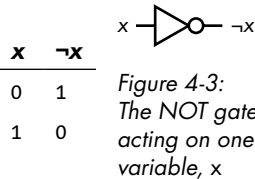| *x*	*y*	*x* ∨ *y*
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4-2: The OR gate acting on two variables, x and y

The truth table shows that the OR operator follows rules somewhat similar to addition in elementary algebra, which is why some use the + symbol to represent it.

NOT

NOT is a *unary operator*, which acts on only one operand. The result of NOT is 1 if the operand is 0, and it is 0 if the operand is 1. Other names for the NOT operation are *complement* and *invert*. I'll use ¬ to designate the NOT operation. It's also common to use the ' symbol, an overscore above the variable, or simply NOT. Figure 4-3 shows the circuit symbol for a NOT gate, and a truth table defining the output, with the operand *x*.

x	*¬x*
0	1
1	0

Figure 4-3:
The NOT gate
acting on one
variable, x

As you'll see, NOT has some properties of the arithmetic negation used in elementary algebra, but there are some significant differences.

It's no accident that AND is multiplicative and OR additive. When George Boole was developing his algebra, he was looking for a way to apply mathematical rigor to logic and use addition and multiplication to manipulate logical statements. Boole developed the rules for his algebra based on using AND for multiplication and OR for addition. In the next section, you'll see how to use these operators, together with NOT, to represent logical statements.

Boolean Expressions

Just as you can use elementary algebra operators to combine variables into expressions like $(x + y)$, you can use Boolean operators to combine variables into expressions.

There is a significant difference, though. A Boolean expression is created from values (0 and 1) and literals. In Boolean algebra, a *literal* is a single instance of a variable or its complement that's being used in an expression. In the expression

$$x \wedge y \vee \neg x \wedge z \vee \neg x \wedge \neg y \wedge \neg z$$

there are three variables (*x*, *y*, and *z*) and seven literals. In a Boolean expression, you can see a variable in both its complemented form and its uncomplemented form because each form is a separate literal.

We can combine literals using either the ∧ or ∨ operator. Like in elementary algebra, Boolean algebra expressions are made up of *terms*, groups of literals that are acted upon by operators, like $(x \vee y)$ or $(a \wedge b)$. And just like elementary algebra, *operation precedence* (or *order of operations*) specifies how these operators are applied when evaluating the expression. Table 4-2 lists the precedence rules for the Boolean operators. As with elementary algebra, expressions in parentheses are evaluated first, following the precedence rules.

Table 4-2: Precedence Rules of Boolean Algebra Operators

Operation	Notation	Precedence
NOT	$\neg x$	Highest
AND	$x \wedge y$	Middle
OR	$x \vee y$	Lowest

Now that you know how the three fundamental Boolean operators work, we'll look at some of the rules they obey when used in algebraic expressions. As you'll see later in the chapter, we can use the rules to simplify Boolean expressions, which will allow us, in turn, to simplify the way we implement those expressions in the hardware.

Knowing how to simplify Boolean expressions is an important tool for both those making hardware and those writing software. A computer is just a physical manifestation of Boolean logic. Even if your only interest is in programming, every programming statement you write is ultimately carried out by hardware that is completely described by the system of Boolean algebra. Our programming languages tend to hide much of this through abstraction, but they still use Boolean expressions to implement programming logic.

Boolean Algebra Rules

When comparing AND and OR in Boolean algebra to multiplication and addition in elementary algebra, you'll find that some of the rules of Boolean algebra are familiar, but some are significantly different. Let's start with the rules that are the same, followed by the rules that differ.

Boolean Algebra Rules That Are the Same as Elementary Algebra

AND and OR are associative.

We say that an operator is *associative* if when there are two or more occurrences of the operator in an expression, the order of applying the operator does not change the value of the expression. Mathematically:

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

$$x \vee (y \vee z) = (x \vee y) \vee z$$

To prove the associative rule for AND and OR, let's use exhaustive truth tables, as shown in Tables 4-3 and 4-4. Table 4-3 lists all possible values of the three variables *x*, *y*, and *z*, as well as the intermediate computations of the terms $(y \wedge z)$ and $(x \wedge y)$. In the last two columns, we can compute the values of each expression on both sides of the previous equations, which shows that the two equalities hold.

Table 4-3: Associativity of the AND Operation

x	y	z	(y ∧ z)	(x ∧ y)	x ∧ (y ∧ z)	(x ∧ y) ∧ z
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	1	0	0
1	1	1	1	1	1	1

Table 4-4 lists all possible values of the three variables x, y, and z, as well as the intermediate computations of the terms $(y \vee z)$ and $(x \vee y)$. In the last two columns, we can compute the values of each expression on both sides of the previous equations, which shows that the two equalities hold.

Table 4-4: Associativity of the OR Operation

x	y	z	(y ∨ z)	(x ∨ y)	x ∨ (y ∨ z)	(x ∨ y) ∨ z
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

This strategy will work for each of the rules shown in this section, but I'll only go through the truth table for the associative rule here. You'll get to do this for the other rules when it's Your Turn at the end of this section.

AND and OR have identity values.

An *identity value* is a value specific to an operation such that using that operation on a quantity with the identity value yields the value of the original quantity. For AND and OR, the identity values are 1 and 0, respectively:

$$x \wedge 1 = x$$

$$x \vee 0 = x$$

AND and OR are commutative.

We can say that an operator is *commutative* if we can reverse the order of its operands:

$$x \wedge y = y \wedge x$$

$$x \vee y = y \vee x$$

AND is distributive over OR.

The AND operator applied to quantities OR-ed together can be *distributed* to apply to each of the OR-ed quantities, like so:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

Unlike in elementary algebra, the additive OR *is* distributive over the multiplicative AND. You'll see this in the next section.

AND has an annulment (also called annihilation) value.

Operating on a value with the operator's *annulment value* yields the annulment value. The annulment value for AND is 0:

$$x \wedge 0 = 0$$

We're used to 0 being the annulment value for multiplication in elementary algebra, but addition has no concept of annulsment. You'll learn about the annulment value for OR in the next section.

NOT shows involution.

An operator shows *involution* if applying it to a quantity twice yields the original quantity:

$$\neg(\neg x) = x$$

Involution is simply the application of a double complement: NOT(NOT true) = true. This is similar to double negation in elementary algebra.

Boolean Algebra Rules That Differ from Elementary Algebra

Although AND is multiplicative and OR is additive, there are significant differences between these logical operations and the arithmetic ones. The differences stem from the fact that Boolean algebra deals with logic expressions that evaluate to either true or false, while elementary algebra deals with the infinite set of real numbers. In this section, you'll see expressions that might remind you of elementary algebra, but the Boolean algebra rules are different.

OR is distributive over AND.

The OR operator applied to quantities AND-ed together can be *distributed* to apply to each of the AND-ed quantities:

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

Because addition is not distributive over multiplication in elementary algebra, you may miss this way of manipulating Boolean expressions.

First, let's look at elementary algebra. Using addition for OR and multiplication for AND in the previous equation, we have this:

$$x + (y \cdot z) \neq (x+y) \cdot (x+z)$$

We can see this by plugging in the numbers $x = 1$, $y = 2$, and $z = 3$. The left-hand side gives

$$1 + (2 \cdot 3) = 7$$

and the right-hand side gives

$$(1+2) \cdot (1+3) = 12$$

Thus, addition is *not* distributive over multiplication in elementary algebra.

The best way to show that OR is distributive over AND in Boolean algebra is to use a truth table, as shown in Table 4-5.

Table 4-5: OR Distributes over AND

x	y	z	$x \vee (y \wedge z)$	$(x \vee y) \wedge (x \vee z)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Comparing the two right-hand columns, you can see that the variable that is common to the two OR terms, *x*, can be factored out, and thus the distributive property holds.

OR has an annulment (also called annihilation) value.

An *annulment value* is a value such that operating on a quantity with the annulment value yields the annulment value. There is no annulment value for addition in elementary algebra, but in Boolean algebra, the annulment value for OR is 1:

$$x \lor 1 = 1$$

AND and OR both have a complement value.

The *complement value* is the diminished radix complement of the variable. You saw in Chapter 3 that the sum of a quantity and that quantity's diminished radix complement is equal to (radix – 1). Since the radix in Boolean algebra is 2, the complement of 0 is 1, and the complement of 1 is 0. So, the complement of a Boolean quantity is simply the NOT of that quantity, which gives

$$x \land \neg x = 0$$

$$x \lor \neg x = 1$$

The complement value illustrates one of the differences between the AND and OR logical operations and the multiplication and addition arithmetic operations. In elementary algebra:

$$x \cdot (-x) = -x^2$$
$$x + (-x) = 0$$

Even if we restrict x to be 0 or 1, in elementary algebra $1 \cdot (-1) = -1$, and $1 + (-1) = 0$.

AND and OR are idempotent.

If an operator is *idempotent*, applying it to two of the same operands results in that operand. In other words:

$$x \land x = x$$

$$x \lor x = x$$

This looks different than in elementary algebra, where repeatedly multiplying a number by itself is exponentiation, and repeatedly adding a number to itself is equivalent to multiplication.

De Morgan's law applies.

In Boolean algebra, the special relationship between the AND and OR operations is captured by *De Morgan's law*, which states

$$\neg(x \land y) = \neg x \lor \neg y$$

$$\neg(x \lor y) = \neg x \land \neg y$$

The first equation states that the NOT of the AND of two Boolean quantities is equal to the OR of the NOT of the two quantities. Likewise, the second equation states that the NOT of the OR of two Boolean quantities is equal to the AND of the NOT of the two quantities.

This relationship is an example of the *principle of duality*, which in Boolean algebra states that if you replace every 0 with a 1, every 1 with a 0, every AND with an OR, and every OR with an AND, the equation is still true. Look back over the rules just given and you'll see that all of them except involution have dual operations. De Morgan's law is one of the best examples of duality. Please, when it's Your Turn, prove De Morgan's law so you can see the principle of duality in play.

YOUR TURN

Use truth tables to prove the Boolean algebra rules given in this section.
Prove De Morgan's law.

Boolean Functions

The functionality of a computer is based on Boolean logic, which means the various operations of a computer are specified by Boolean functions. A Boolean function looks somewhat like a function in elementary algebra, but the variables can appear in either uncomplemented or complemented form. The variables and constants are connected by Boolean operators. A Boolean function evaluates to either 1 or 0 (true or false).

In the section "Adding in the Binary Number System" in Chapter 3, you saw that when adding two bits, x and y, in a binary number, we have to include a possible carry into their bit position in the number. The conditions that cause carry to be 1 are

$x = 1$, $y = 1$, and there's no carry into the current bit position,
or
$x = 0$, $y = 1$, and there's carry into the current bit position,
or
$x = 1$, $y = 0$, and there's carry into the current bit position,
or
$x = 1$, $y = 1$, and there's carry into the current bit position.
We can express this more concisely with this Boolean function:

$$C_{out}(c_{in},x,y)=(\neg c_{in}\wedge x\wedge y)\vee(c_{in}\wedge\neg x\wedge y)\vee(c_{in}\wedge x\wedge\neg y)\vee(c_{in}\wedge x\wedge y)$$

where x is one bit, y is the other bit, c_{in} is the carry in from the next-lower-order bit position, and $C_{out}(c_{in},x,y)$ is the carry resulting from the addition in the current bit position. We'll use this equation throughout this section, but first, let's think a bit about the differences between Boolean and elementary functions.

Like an elementary algebra function, a Boolean algebra function can be manipulated mathematically, but the mathematical operations are different. Operations in elementary algebra are performed on the infinite set of real numbers, but Boolean functions work on only two possible values, 0 or 1. Elementary algebra functions can evaluate to any real number, but Boolean functions can evaluate only to 0 or 1.

This difference means we have to think differently about Boolean functions. For example, if you look at the elementary algebra function

$$F(x,y)=x\cdot(-y)$$

you probably read it as "if I multiply the value of x by the negative of the value of y, I'll get the value of $F(x, y)$." However, if you look at the Boolean function

$$F(x,y)=x\wedge(\neg y)$$

there are only four possibilities. If $x = 1$ and $y = 0$, then $F(x, y) = 1$. For the other three possibilities, $F(x, y) = 0$. Whereas you can plug in any numbers in an elementary algebra function, a Boolean algebra function shows you what the values of the variables are that cause the function to evaluate to 1. I think of elementary algebra functions as *asking* me to plug in values for the variables for evaluation, while Boolean algebra functions *tell* me what values of the variables cause the function to evaluate to 1.

There are simpler ways to express the conditions for carry. And those simplifications lead to being able to implement this function with fewer logic gates, thus lowering the cost and power usage. In this and the following sections, you'll learn how the mathematical nature of Boolean algebra makes function simplification easier and more concise.

Canonical Sum or Sum of Minterms

A *canonical form* of a Boolean function explicitly shows whether each variable in the problem is complemented or not in each term that defines the function, just as we did with our English statement of the conditions that produce a carry of 1 earlier. This ensures that you have taken all possible combinations into account in the function definition. The truth table, shown in Table 4-6, for carry equation we saw earlier

$$C_{out}(c_{in},x,y)=(\neg c_{in}\wedge x\wedge y)\vee(c_{in}\wedge\neg x\wedge y)\vee(c_{in}\wedge x\wedge\neg y)\vee(c_{in}\wedge x\wedge y)$$

should help to clarify this.

Table 4-6: Conditions That Cause Carry to Be 1

Minterm	c_{in}	x	y	$(\neg c_{in} \wedge x \wedge y)$	$(c_{in} \wedge \neg x \wedge y)$	$(c_{in} \wedge x \wedge \neg y)$	$(c_{in} \wedge x \wedge y)$	$C_{out}(c_{in}, x, y)$
m_0	0	0	0	0	0	0	0	0
m_1	0	0	1	0	0	0	0	0
m_2	0	1	0	0	0	0	0	0
m_3	0	1	1	1	0	0	0	1
m_4	1	0	0	0	0	0	0	0
m_5	1	0	1	0	1	0	0	1
m_6	1	1	0	0	0	1	0	1
m_7	1	1	1	0	0	0	1	1

Although the parentheses in the equation are not required, I've added them to help you see the form of the equation. The parentheses show four *product terms*, terms where all the literals are operated on only by AND. The four product terms are then OR-ed together. Since the OR operation is like addition, the right-hand side is called a *sum of products*. It's also said to be in *disjunctive normal form*.

Now let's look more closely at the product terms. Each of them includes all the variables in this equation in the form of a literal (uncomplemented or complemented). An equation that has n variables has $2n$ permutations of the values for the variables; a *minterm* is a product term that specifies exactly one of the permutations. Since there are four combinations of values for c_{in}, x, and y that produce a carry of 1, the previous equation has four out of the possible eight minterms. A Boolean function that is defined by summing (OR-ing) all the minterms that evaluate to 1 is said to be a *canonical sum*, a *sum of minterms*, or in *full disjunctive normal form*. A function defined by a sum of minterms evaluates to 1 when at least one of the minterms evaluates to 1.

For every minterm, exactly one set of values for the variables makes the minterm evaluate to 1. For example, the minterm $(c_{in} \wedge x \wedge \neg y)$ in the previous equation evaluates to 1 only when $c_{in} = 1$, $x = 1$, $y = 0$. A product term that does not contain all the variables in the problem, either in uncomplemented or in complemented form, will always evaluate to 1 for more sets of values for the variables than a minterm. For example, $(c_{in} \wedge x)$ evaluates to 1 for $c_{in} = 1$, $x = 1$, $y = 0$, and $c_{in} = 1$, $x = 1$, $y = 1$. Because they minimize the number of cases that evaluate to 1, we call them *min*terms.

Rather than write out all the literals in a function, logic designers commonly use the notation m_i to specify the ith minterm, where i is the integer represented by the values of the literals in the problem if the values are placed in order and treated as binary numbers. For example, $c_{in} = 1$, $x = 1$, $y = 0$ gives 110, which is the (base 10) number 6; thus, that minterm is m_6. Table 4-6 shows all eight possible minterms for a three-variable equation, and the minterm, m_6 $(c_{in} \wedge x \wedge \neg y)$, in the four-term equation shown earlier evaluates to 1 when $c_{in} = 1$, $x = 1$, $y = 0$.

Using this notation to write Boolean equations as a canonical sum and using the Σ symbol to denote summation, we can restate the function for carry:

$$\begin{aligned}
C_{out}(c_{in},x,y) &= (\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y) \\
&= m_3 \vee m_5 \vee m_6 \vee m_7 \\
&= \Sigma(3,5,6,7)
\end{aligned}$$

We are looking at a simple example here. For more complicated functions, writing all the minterms out is error-prone. The simplified notation is easier to work with and helps to avoid making errors.

Canonical Product or Product of Sums

Depending on factors like available components and personal choice, a designer may prefer to work with the cases where a function evaluates to 0 instead of 1. In our example, that means a design that specifies when carry is 0. To see how this works, let's take the complement of both sides of the equation for specifying carry, using De Morgan's law:

$$\neg C_{out}(c_{in},x,y) = (c_{in} \vee \neg x \vee \neg y) \wedge (\neg c_{in} \vee x \vee \neg y) \wedge (\neg c_{in} \vee \neg x \vee y) \wedge (\neg c_{in} \vee \neg x \vee \neg y)$$

Because we complemented both sides of the equation, we now have the Boolean equation for $\neg C_{out}$, the complement of carry. Thus, we are looking for conditions that cause $\neg C_{out}$ to evaluate to 0, not 1. These are shown in the truth table, Table 4-7.

Table 4-7: Conditions That Cause the Complement of Carry to Be 0

Maxterm	c_{in}	x	y	$(c_{in} \vee \neg x \vee \neg y)$	$(\neg c_{in} \vee x \vee \neg y)$	$(\neg c_{in} \vee \neg x \vee y)$	$(\neg c_{in} \vee \neg x \vee \neg y)$	$\neg C_{out}(c_{in}, x, y)$
M_0	0	0	0	1	1	1	1	1
M_1	0	0	1	1	1	1	1	1
M_2	0	1	0	1	1	1	1	1
M_3	0	1	1	0	1	1	1	0
M_4	1	0	0	1	1	1	1	1
M_5	1	0	1	1	0	1	1	0
M_6	1	1	0	1	1	0	1	0
M_7	1	1	1	1	1	1	0	0

In this equation, the parentheses are required due to the precedence rules of Boolean operators. The parentheses show four *sum terms*, terms where all the literals are operated on only by OR. The four sum terms are then AND-ed together. Since the AND operation is like multiplication, the right-hand side is called a *product of sums*. It's also said to be in *conjunctive normal form*.

Each of the sum terms includes all the variables in this equation in the form of literals (uncomplemented or complemented). Where a minterm was a *product* term that specified a single permutation of the $2n$ permutations of possible values for the variables, a *maxterm* is a *sum* term specifying exactly one of those permutations. A Boolean function that is defined by multiplying (AND-ing) all the maxterms that evaluate to 0 is said to be a *canonical product*, a *product of maxterms*, or in *full conjunctive normal form*.

Each maxterm identifies exactly one set of values for the variables in a function that evaluates to 0 when OR-ed together. For example, the maxterm $(\neg c_{in} \vee \neg x \vee y)$ in the previous equation evaluates to 0 only when $c_{in} = 1$, $x = 1$, $y = 0$. But a sum term that does not contain all the variables in the problem, either in uncomplemented or complemented form, will always evaluate to 0 for more than one set of values. For example, the sum term $(\neg c_{in} \vee \neg x)$ evaluates to 0 for two sets of values for the three variables in this example, $c_{in} = 1$, $x = 1$, $y = 0$ and $c_{in} = 1$, $x = 1$, and $y = 1$. Because they minimize the number of cases that evaluate to 0 and thus *maximize* the number of cases that evaluate to 1, we call them *max*terms.

Rather than write out all the literals in a function, logic designers commonly use the notation M_i to specify the ith maxterm, where i is the integer value of the base 2 number created by concatenating the values of the literals in the problem. For example, stringing together $c_{in} = 1$, $x = 1$, $y = 0$ gives 110, which is the maxterm M_6. The truth table, Table 4-7, shows the maxterms that cause carry = 0. Notice that maxterm M_6, $(\neg c_{in} \vee \neg x \vee y)$ evaluates to 0 when $c_{in} = 1$, $x = 1$, $y = 0$.

Using this notation to write Boolean equations as a canonical sum and using the $\prod$ symbol to denote multiplication, we can restate the function for the complement of carry as follows:

$$\begin{aligned} \neg C_{out}(c_{in}, x, y) &= (c_{in} \vee \neg x \vee \neg y) \wedge (\neg c_{in} \vee x \vee \neg y) \wedge (\neg c_{in} \vee \neg x \vee y) \wedge (\neg c_{in} \vee \neg x \vee \neg y) \\ &= M_3 \wedge M_5 \wedge M_6 \wedge M_7 \\ &= \prod (3,5,6,7) \end{aligned}$$

If you look back at Table 4-7, you'll see that these are the conditions that cause the complement of carry to be 0 and hence carry to be 1. This shows that using either minterms or maxterms is equivalent. The one you use can depend on factors such as what hardware components you have available to implement the function and personal preference.

Comparison of Canonical Boolean Forms

Table 4-8 shows all the minterms and maxterms for a three-variable problem. If you compare corresponding minterms and maxterms, you can see the duality of minterms and maxterms: one can be formed from the other using De Morgan's law by complementing each variable and interchanging OR and AND.

Table 4-8: Canonical Terms for a Three-Variable Problem

Minterm = 1		x	y	z	Maxterm = 0	
m_0	$\neg x \wedge \neg y \wedge \neg z$	0	0	0	M_0	$x \vee y \vee z$
m_1	$\neg x \wedge \neg y \wedge z$	0	0	1	M_1	$x \vee y \vee \neg z$
m_2	$\neg x \wedge y \wedge \neg z$	0	1	0	M_2	$x \vee \neg y \vee z$
m_3	$\neg x \wedge y \wedge z$	0	1	1	M_3	$x \vee \neg y \vee \neg z$
m_4	$x \wedge \neg y \wedge \neg z$	1	0	0	M_4	$\neg x \vee y \vee z$
m_5	$x \wedge \neg y \wedge z$	1	0	1	M_5	$\neg x \vee y \vee \neg z$
m_6	$x \wedge y \wedge \neg z$	1	1	0	M_6	$\neg x \vee \neg y \vee z$
m_7	$x \wedge y \wedge z$	1	1	1	M_7	$\neg x \vee \neg y \vee \neg z$
...						

The canonical forms give you a complete, and unique, statement of the function because they take all possible combinations of the values of the variables into account. However, there often are simpler solutions to the problem. The remainder of this chapter will be devoted to methods of simplifying Boolean functions.

Boolean Expression Minimization

When implementing a Boolean function in hardware, each $\wedge$ operator becomes an AND gate, each $\vee$ operator an OR gate, and each $\neg$ operator a NOT gate. In general, the complexity of the hardware is related to the number of AND and OR gates used (NOT gates are simple and tend not to contribute significantly to the complexity). Simpler hardware uses fewer components, thus saving cost and space, and uses less power. Cost, space, and power savings are especially important with handheld and wearable devices. In this section, you'll learn how you can manipulate Boolean expressions to reduce the number of ANDs and ORs, thus simplifying their hardware implementation.

Minimal Expressions

When simplifying a function, start with one of the canonical forms to ensure that you have taken all possible cases into account. To translate a problem into a canonical form, create a truth table that lists all possible combinations of the variables in the problem. From the truth table, it will be easy to list the minterms or maxterms that define the function.

Armed with a canonical statement, the next step is to look for a functionally equivalent *minimal expression*, an expression that does the same thing as the canonical one, but with a minimum number of literals and

Boolean operators. To minimize an expression, we apply the rules of Boolean algebra to reduce the number of terms and the number of literals in each term, without changing the logical meaning of the expression.

There are two types of minimal expressions, depending on whether you use minterms or maxterms:

Minimal Sum of Products

When starting with a minterms description of the problem, the minimal expression is called a *minimal sum of products*, which is a sum of products expression where all other mathematically equivalent sum of products expressions have at least as many product terms, and those with the same number of product terms have at least as many literals.

As an example of a minimal sum of products, consider these equations:

$$S(x,y,z) = (\neg x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y \wedge z)$$
$$S1(x,y,z) = (\neg x \wedge \neg y \wedge \neg z) \vee (x \wedge \neg y)$$
$$S2(x,y,z) = (x \wedge \neg y \wedge z) \vee (\neg y \wedge \neg z)$$
$$S3(x,y,z) = (x \wedge \neg y) \vee (\neg y \wedge \neg z)$$

S is in canonical form as each of the product terms explicitly shows the contribution of all three variables. The other three functions are simplifications of S. Although all three have the same number of product terms, $S3$ is a minimal sum of products for S because it has fewer literals in its product terms than $S1$ and $S2$.

Minimal Product of Sums

When starting with a maxterms description of the problem, the minimal expression is called a *minimal product of sums*, which is a product of sums expression where all other mathematically equivalent product of sums expressions have at least as many sum terms, and those with the same number of sum terms have at least as many literals.

For an example of a minimal product of sums, consider these equations:

$$P(x,y,z) = (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z)$$
$$P1(x,y,z) = (x \vee \neg y \vee z) \wedge (\neg x \vee z)$$
$$P2(x,y,z) = (\neg x \vee y \vee z) \wedge (\neg y \vee z)$$
$$P3(x,y,z) = (\neg x \vee z) \wedge (\neg y \vee z)$$

P is in canonical form, and the other three functions are simplifications of P. Although all three have the same number of sum terms as P, $P3$ is a minimal product of sums for P because it has fewer literals in its product terms than $P1$ and $P2$.

A problem may have more than one minimal solution. Good hardware design typically involves finding several minimal solutions and then

assessing each one within the context of the available hardware. This means more than using fewer gates. For example, as you'll learn when we discuss the actual hardware implementations, adding judiciously placed NOT gates can actually reduce hardware complexity.

In the following two sections, you'll see two ways to find minimal expressions.

Minimization Using Algebraic Manipulations

To illustrate the importance of reducing the complexity of a Boolean function, let's return to the function for carry:

$$C_{out}(c_{in},x,y)=(\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y)$$

The expression on the right-hand side of the equation is a sum of minterms. Figure 4-4 shows the circuit to implement this function. It requires four AND gates and one OR gate. The small circles at the inputs to the AND gates indicate a NOT gate at that input.
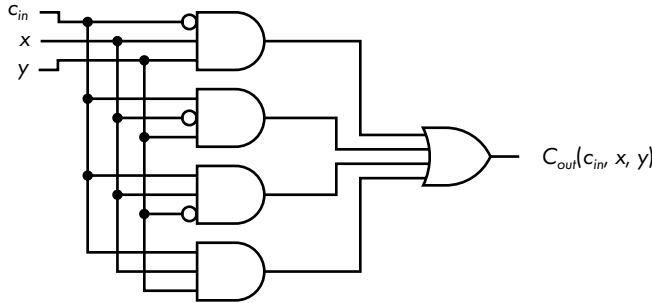

Figure 4-4: Hardware implementation of a function to generate the value of carry when adding two numbers

Now let's try to simplify the Boolean expression implemented in Figure 4-4 to see whether we can reduce the hardware requirements. Note that there may not be a single path to a solution, and there may be more than one correct solution. I'm presenting only one way here.

First, we'll do something that might look strange. We'll use the idempotency rule to duplicate the fourth term twice:

$$C_{out}(c_{in},x,y)=(\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y) \vee (c_{in} \wedge x \wedge y)\ \vee (c_{in} \wedge x \wedge y)$$

Next, rearrange the product terms a bit to OR each of the three original terms with $(c_{in} \wedge x \wedge y)$:

$$C_{out}(c_{in},x,y)=((\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge x \wedge y)) \vee ((c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y)) \vee ((c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge y))$$

Now we can use the rule for distribution of AND over OR to factor out terms that OR to 1:

$$\begin{aligned} C_{out}(c_{in},x,y) &= (x{\wedge}y{\wedge}(\neg c_{in}{\vee}c_{in})){\vee}(c_{in}{\wedge}x{\wedge}(\neg y{\vee}y)){\vee}(c_{in}{\wedge}y{\wedge}(\neg x{\vee}x)) \\ &= (x{\wedge}y{\wedge}1){\vee}(c_{in}{\wedge}x{\wedge}1){\vee}(c_{in}{\wedge}y{\wedge}1) \\ &= (x{\wedge}y){\vee}(c_{in}{\wedge}x){\vee}(c_{in}{\wedge}y) \end{aligned}$$

Figure 4-5 shows the circuit for this function. Not only have we eliminated an AND gate, all the AND gates and the OR gate have one fewer input.

Figure 4-5: Simplified hardware implementation generating carry when adding two numbers

Comparing the circuits in Figures 4-5 and 4-4, Boolean algebra has helped you to simplify the hardware implementation. You can see this simplification from stating the conditions that result in a carry of 1 in English: the original, canonical form of the equation stated that carry, $C_{out}(c_{in},x,y)$, will be 1 in any of these four cases:

if $c_{in} = 0$, $x = 1$, and $y = 1$,

if $c_{in} = 1$, $x = 0$, and $y = 1$,

if $c_{in} = 1$, $x = 1$, and $y = 0$,

if $c_{in} = 0$ $x = 1$, and $y = 1$.

The minimization can be stated much simpler: carry is 1 if at least two of c_{in}, x, and y are 1.

We arrived at the solution in Figure 4-5 by starting with the sum of minterms; in other words, we were working with the values of c_{in}, x, and y that generate a 1 for carry. As you saw in the section "Canonical Product or Product of Sums" (page XX), since carry must be either 1 or 0, it's equally as valid to start with the values of c_{in}, x, and y that generate a 0 for the complement of carry and to write the equation as a product of maxterms:

$$\neg C_{out}(c_{in},x,y) = (c_{in}{\vee}\neg x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}\neg y)$$

To simplify this equation, we'll take the same approach we took with the sum of minterms and start by duplicating the last term twice to give:

$$\neg C_{out}(c_{in},x,y) = (c_{in}{\vee}\neg x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}\neg y){\wedge}(\neg c_{in}{\vee}\neg x{\vee}\neg y)$$

Adding some parentheses helps to clarify the simplification process:

$$\neg C_{out}(c_{in},x,y)=((c_{in}\vee\neg x\vee\neg y)\wedge(\neg c_{in}\vee\neg x\vee\neg y))\wedge((\neg c_{in}\vee x\vee\neg y)\wedge(\neg c_{in}\vee\neg x\vee\neg y))\wedge((\neg c_{in}\vee\neg x\vee y)\wedge(\neg c_{in}\vee\neg x\vee\neg y))$$

Next, use the distribution of OR over AND. Because this is tricky, I'll go over the steps to simplify the first grouping of product terms in this equation—the steps for the other two groupings are similar to this one. Distribution of OR over AND has this generic form:

$$(X\vee Y)\wedge(X\vee Z)=X\vee(Y\wedge Z)$$

Looking at the sum terms in our first grouping, you can see they both share a $(\neg x\vee\neg y)$. So, we'll make the following substitutions into the generic form:

$$X=(\neg x\vee\neg y)$$
$$Y=c_{in}$$
$$Z=\neg c_{in}$$

Making the substitutions and using the complement rule for AND, we get

$$(c_{in}\vee\neg x\vee\neg y)\wedge(\neg c_{in}\vee\neg x\vee\neg y)=(\neg x\vee\neg y)\vee(c_{in}\wedge\neg c_{in})$$
$$=(\neg x\vee\neg y)$$

Applying these same manipulations to other two groupings, we get

$$\neg C_{out}(c_{in},x,y)=(\neg x\vee\neg y)\wedge(\neg c_{in}\vee\neg x)\wedge(\neg c_{in}\vee\neg y)$$

Figure 4-6 shows the circuit implementation of this function. This circuit produces the complement of carry. We would need to complement the output, $\neg C_{out}(c_{in},x,y)$, to get the value of carry.
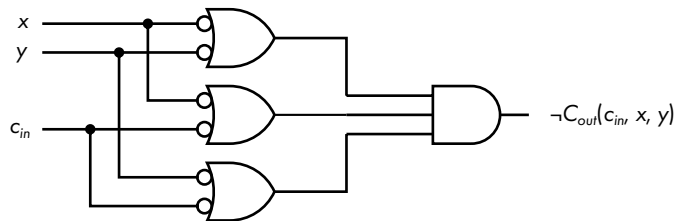

Figure 4-6: Simplified hardware implementation generating the complement of carry when adding two numbers

Compare Figure 4-6 with Figure 4-5, and you can graphically see De Morgan's law: the ORs have become ANDs with complemented values as inputs.

The circuit in Figure 4-5 might look simpler to you because the circuit in Figure 4-6 requires NOT gates at the six inputs to the OR gates.

But as you will see in the next chapter, this may not be the case due to the inherent electronic properties of the devices used to construct logic gates. The important point to understand here is that there is more than one way to solve the problem. One of the jobs of the hardware engineer is to decide which solution is best, based on things such as cost, availability of components, and so on.

Minimization Using Karnaugh Maps

The algebraic manipulations used to minimize Boolean functions may not always be obvious. You may find it easier to work with a graphic representation of the logical statements.

A commonly used graphic tool for working with Boolean functions is the *Karnaugh map*, also called a *K-map*. Invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs, the Karnaugh map gives a way to visually find the same simplifications you can find algebraically. They can be used either with a sum of products, using minterms, or a product of sums, using maxterms. To illustrate how they work, we'll start with minterms.

Simplifying Sums of Products Using Karnaugh Maps

The Karnaugh map is a rectangular grid with a cell for each minterm. There are $2n$ cells for n variables. Figure 4-7 is a Karnaugh map showing all four possible minterms for two variables, x and y. The vertical axis is used for plotting x and the horizontal for y. The value of x for each row is shown by the number (0 or 1) immediately to the left of the row, and the value of y for each column appears at the top of the column.

$F(x, y)$

	y	
	0	1
x 0	m_0	m_1
1	m_2	m_3

Figure 4-7: Mapping of two-variable minterms on a Karnaugh map

To illustrate how to use a Karnaugh map, let's look at an arbitrary function of two variables:

$$F(x,y) = (x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge y)$$

Start by placing a 1 in each cell corresponding to a minterm that appears in the equation, as shown in Figure 4-8.

$F(x, y)$

Figure 4-8: Karnaugh
map of the arbitrary
function, F(x, y)

By placing a `1` in the cell corresponding to each minterm that evaluates to `1`, we can see graphically when the equation evaluates to `1`. The two cells on the right side correspond to the minterms m_1 and m_3, $(\neg x \wedge y)$ and $(x \wedge y)$. Since these terms are OR-ed together, $F(x, y)$ evaluates to `1` if either of these minterms evaluates to `1`. Using the distributive and complement rules, we can see that

$$(\neg x \wedge y) \vee (x \wedge y) \quad = (\neg x \vee x) \wedge y$$
$$= y$$

This shows algebraically that $F(x, y)$ evaluates to `1` whenever y is `1`, which you'll see next by simplifying this Karnaugh map.

The only difference between the two minterms, $(\neg x \wedge y)$ and $(x \wedge y)$, is the change from x to $\neg x$. Karnaugh maps are arranged such that only one variable changes between two cells that share an edge, a requirement called the *adjacency rule.*

To use a Karnaugh map to perform simplification, you group two adjacent cells in a sum of products Karnaugh map that have `1`s in them. Then you eliminate the variable that differs between them and coalesce the two product terms. Repeating this process allows you to simplify the equation. Each grouping eliminates a product term in the final sum of products. This can be extended to equations with more than two variables, but the number of cells that are grouped together must be a multiple of 2, and you can only group adjacent cells. The adjacency wraps around from side to side and from top to bottom. You'll see an example of that in a few pages.

To see how all this works, consider the grouping in the Karnaugh map in Figure 4-9.

$F(x, y)$

Figure 4-9: Two of
the minterms in F(x, y)
grouped

This grouping is a graphical representation of the algebraic manipulation we did earlier. You can see that $F(x, y)$ evaluates to `1` whenever $y = 1$,

regardless of the value of x. Thus, the grouping coalesces two minterms into one product term by eliminating x.

From the last grouping, we know our final simplified function will have a y term. Let's do another grouping to find the next term. First, we'll simplify the equation algebraically. Returning to the original equation for $F(x, y)$, we can use idempotency to duplicate one of the minterms:

$$F(x,y)=(x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge y) \vee (x \wedge y)$$

Now we'll do some algebraic manipulation on the first product term and the one we just added:

$$(x \wedge \neg y) \vee (x \wedge y) = (\neg y \vee y) \wedge x$$
$$= x$$

Instead of using algebraic manipulations, we can do this directly on our Karnaugh map, as shown in Figure 4-10. This map shows that separate groups can include the same cell (minterm).
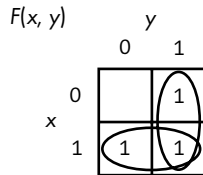

Figure 4-10: A Karnaugh map grouping showing that $(x \wedge \neg y) \vee (\neg x \wedge y)$ $\vee (x \wedge y) = (x \vee y)$

The group in the bottom row represents the product term x, and the one in the right-hand column represents y, giving us the following minimization:

$$F(x,y)=x \vee y$$

Note that the cell that is included in both groupings, $(x \wedge y)$, is the term that we duplicated using the idempotent rule in our algebraic solution previously. You can think of including a cell in more than one group as adding a duplicate copy of the cell, like using the idempotent rule in our algebraic manipulation earlier, and then coalescing it with the other cell(s) in the group, thus removing it.

The adjacency rule is automatically satisfied when there are only two variables in the function. But when we add another variable, we need to think about how to order the cells of a Karnaugh map such that we can use the adjacency rule to simplify Boolean expressions.

Karnaugh Map Cell Order

One of the problems with both the binary and BCD codes is that the difference between two adjacent values often involves more than one bit being

changed. In 1943 Frank Gray introduced a code, the *Gray code*, in which adjacent values differ by only one bit. The Gray code was invented because the switching technology of that time was more prone to errors. If one bit was in error, the value represented by a group of bits was off by only one in the Gray code. That's seldom a problem these days, but this property shows us how to order the cells in a Karnaugh map.

Constructing the Gray code is quite easy. Start with one bit:

Decimal	Gray Code
0	0
1	1

To add a bit, first write the mirror image of the existing pattern:

Gray Code
0
1
1
0

Then add a 0 to the beginning of each of the original bit patterns and add a 1 to the beginning of each of the mirror image set to give the Gray code for two bits, as shown in Table 4-9.

Table 4-9: Gray Code for Two Bits

Decimal	Gray Code
0	00
1	01
2	11
3	10

This is the reason the Gray code is sometimes called *reflected binary code* (*RBC*). Table 4-10 shows the Gray code for four bits.

Table 4-10: Gray Code for Four Bits

Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010

(continued)

Decimal	Gray Code	Binary
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Let's compare the binary codes with the Gray codes for the decimal values 7 and 8 in Table 4-10. The binary codes for 7 and 8 are 0111 and 1000, respectively; all four bits change when stepping only 1 in decimal value. But comparing the Gray codes for 7 and 8, 0100 and 1100, respectively, only one bit changes, thus satisfying the adjacency rule for a Karnaugh map.

Notice that the pattern of changing only one bit between adjacent values also holds when the bit pattern wraps around. That is, only one bit is changed when going from the highest value (15 for four bits) to the lowest (0).

Karnaugh Map for Three Variables

To see how the adjacency property is important, let's consider a more complicated function. We'll use a Karnaugh map to simplify our function for carry, which has three variables. Adding another variable means that we need to double the number of cells to hold the minterms. To keep the map two-dimensional, we add the new variable to an existing variable on one side of the map. We need a total of eight cells (2^3), so we'll draw it four cells wide and two high. We'll add z to the y-axis and draw our Karnaugh map with y and z on the horizontal axis, and x on the vertical, as shown in Figure 4-11.

$F(x, y, z)$

	yz			
	00	01	11	10
x 0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

Figure 4-11: Mapping of three-variable minterms on a Karnaugh map

The order of the bit patterns along the top of the three-variable Karnaugh map is 00, 01, 11, 10, as opposed to 00, 01, 10, 11, which is the Gray code order in Table 4-9. The adjacency rule also holds when wrapping around the edges of the Karnaugh map—that is, going from m_2 to m_0 or going from m_6 to m_4—which means that groups can wrap around the edges of the map. (Other axis labeling schemes will also work, as you'll see when it's Your Turn at the end of this section.)

You saw earlier in this chapter that carry can be expressed as the sum of four minterms:

$$C_{out}(c_{in},x,y) = (\neg c_{in} \wedge x \wedge y) \vee (c_{in} \wedge \neg x \wedge y) \vee (c_{in} \wedge x \wedge \neg y) \vee (c_{in} \wedge x \wedge y)$$
$$= m_3 \vee m_5 \vee m_6 \vee m_7$$
$$= \sum (3,5,6,7)$$

Figure 4-12 shows these four minterms on the Karnaugh map.
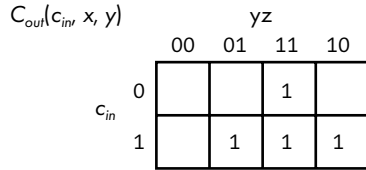

Figure 4-12: Karnaugh map of the function for carry

We look for adjacent cells that can be grouped together to eliminate one variable from the product term. As noted, the groups can overlap, giving the three groups shown in Figure 4-13.
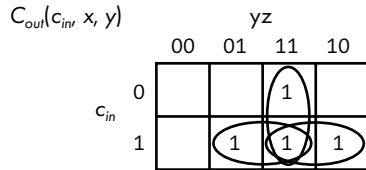

Figure 4-13: A minimum sum of products of the function for carry = 1

Using the three groups in the Karnaugh map in Figure 4-13, we end up with the same equation we got through algebraic manipulations:

$$C_{out}(c_{in},x,y) = (x \wedge y) \vee (c_{in} \wedge x) \vee (c_{in} \wedge y)$$

Simplifying Products of Sums Using Karnaugh Maps

It's equally valid to work with a function that shows when the complement of carry is 0. We did that using maxterms:

$$\neg C_{max}(c_{in},x,y) = (c_{in} \vee \neg x \vee \neg y) \wedge (\neg c_{in} \vee x \vee \neg y) \wedge (\neg c_{in} \vee \neg x \vee y) \wedge (\neg c_{in} \vee \neg x \vee \neg y)$$
$$= M_7 \wedge M_6 \wedge M_5 \wedge M_3$$
$$= \prod(3,5,6,7)$$

Figure 4-14 shows the arrangement of maxterms on a three-variable Karnaugh map.

¬F(x, y, z)

	yz			
	00	01	11	10
x 0	M_0	M_1	M_3	M_2
1	M_4	M_5	M_7	M_6

Figure 4-14: Mapping of three-variable maxterms on a Karnaugh map

When working with a maxterm statement of the solution, you mark the cells that evaluate to 0. The minimization process is the same as when working with minterms, except that you group the cells with 0s in them.

Figure 4-15 shows a minimization of $\neg C_{out}(c_{in},x,y)$, the complement of carry.

¬C_{out}(c_{in}, x, y)

	yz			
	00	01	11	10
c_{in} 0			0	
1		0	0	0

Figure 4-15: A minimum product of sums of the function for NOT carry = 0

The Karnaugh map in Figure 4-15 leads to the same product of sums we got algebraically for the complement of carry = 0:

$$\neg C_{out}(c_{in},x,y)=(\neg x \vee \neg y) \wedge (\neg c_{in} \vee \neg x) \wedge (\neg c_{in} \vee \neg y)$$

If you compare Figures 4-13 and 4-15, you can see a graphic view of De Morgan's law. When making this comparison, keep in mind that Figure 4-13 shows the product terms that get added, and Figure 4-15 shows the sum terms that get multiplied, and the result is complemented. Thus, we exchange 0 and 1 and exchange AND and OR to go from one Karnaugh map to the other.

To further emphasize the duality of minterm and maxterm, compare Figure 4-16(a) and Figure 4-16(b).

F(x, y, z)

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

(a)

¬F(x, y, z)

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

(b)

Figure 4-16: Comparison of (a) one minterm and (b) one maxterm

Figure 4-16(a) shows the function:

$$F(x,y,z)=\neg x \wedge \neg y \wedge \neg z$$

Although it's not necessary and usually not done, we have placed a 0 in each of the cells representing minterms not included in this function.

Similarly, in Figure 4-16(b), we have placed a 0 for the maxterm and a 1 in each of the cells representing the maxterms that are not included in the function:

$$\neg F = x \vee y \vee z$$

This comparison graphically shows how a minterm specifies the minimum number of 1s in a Karnaugh map, while a maxterm specifies the maximum number of 1s.

Larger Groupings on a Karnaugh Map

Thus far, we have grouped only two cells together on our Karnaugh maps. Let's look at an example of larger groups. Consider a function that outputs 1 when a three-bit number is even. Table 4-11 shows the truth table. It uses 1 to indicate that the number is even and uses 0 to indicate odd.

Table 4-11: Even Values of an Eight-Bit Number

Minterm	X	y	z	Number	Even(x, y, z)
m_0	0	0	0	0	1
m_1	0	0	1	1	0
m_2	0	1	0	2	1
m_3	0	1	1	3	0
m_4	1	0	0	4	1
m_5	1	0	1	5	0
m_6	1	1	0	6	1
m_7	1	1	1	7	0

The canonical sum of products for this function is

$$Even(x,y,z)=\sum(0,2,4,6)$$

Figure 4-17 shows these minterms on a Karnaugh map with these four terms grouped together. You can group all four together because they all have adjacent edges.

From the Karnaugh map in Figure 4-17, we can write the equation for showing when a three-bit number is even:

$$Even(x,y,z)=\neg z$$

Even(x, y, z)

Figure 4-17: Karnaugh map showing
even values of three-bit number

The Karnaugh map shows that it does not matter what the values of x and y are, only that $z = 0$.

Adding More Variables to a Karnaugh Map

Each time you add another variable to a Karnaugh map, you need to double the number of cells. The only requirement for the Karnaugh map to work is that you arrange the minterms, or maxterms, according to the adjacency rule. Figure 4-18 shows a four-variable Karnaugh map for minterms. The y and z variables are on the horizontal axis, and w and x are on the vertical.

$F(w, x, y, z)$

Figure 4-18: Mapping of four-variable
minterms on a Karnaugh map

So far we have assumed that every minterm (or maxterm) is accounted for in our functions. But design does not take place in a vacuum. We might have knowledge about other components of the overall design telling us that some combinations of variable values can never occur. Next, we'll see how to take this knowledge into account in your function simplification process. The Karnaugh map provides an especially clear way to visualize the situation.

Don't Care Cells

Sometimes, you have information about the values that the variables can have. If you know which combinations of values will never occur, then the minterms (or maxterms) that represent those combination are irrelevant. For example, you may want a function that indicates whether one of two possible events has occurred or not, but you know that the two events cannot occur simultaneously. Let's name the events x and y, and let 0 indicate

that the event has not occurred and 1 indicate that it has. Table 4-12 shows the truth table for our function, $F(x, y)$.

Table 4-12: Truth Table for x or y Occurring, but not Both

x	y	F(x, y)
0	0	0
0	1	1
1	0	1
1	1	X

We can show that both events cannot occur simultaneously by placing an × in that row. We can draw a Karnaugh map with an × for the minterm that can't exist in the system, as shown in Figure 4-19. The × represents a *don't care* cell—we don't care whether this cell is grouped with other cells or not.
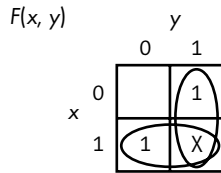

Figure 4-19: Karnaugh map for F(x, y), showing a "don't care" cell

Since the cell that represents the minterm $(x \wedge y)$ is a "don't care" cell, we can include it, or not, in our minimization groupings, leading to the two groupings shown. The Karnaugh map in Figure 4-19 leads us to the solution:

$$F(x,y) = x \vee y$$

which is a simple OR gate. You probably guessed this solution without having to use a Karnaugh map. You'll see a more interesting use of "don't care" cells when you learn about the design of two digital logic circuits at the end of Chapter 7.

Combining Basic Boolean Operators

As mentioned earlier in this chapter, we can combine basic Boolean operators to implement more complex Boolean operators. Now that you know how to work with Boolean functions, we'll design one of the more common operators, the *exclusive or*, often called *XOR*, using the three basic operators, AND, OR, and NOT. It's so commonly used that it has its own circuit symbol.

XOR

The XOR is a binary operator. The result is 1 if one, and only one, of the two operands is 1; otherwise, the result is 0. We'll use $\underline{\vee}$ to designate the XOR operation. It's also common to use the $\oplus$ symbol. Figure 4-20 shows the XOR gate operation with inputs *x* and *y*.

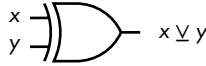| *x*	*y*	*x* $\underline{\vee}$ *y*
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4-20: The XOR gate acting on two variables, x and y

The minterm implementation of this operation is

$$x \underline{\vee} y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

The XOR operator can be implemented with two AND gates, two NOT gates, and one OR gate, as shown in Figure 4-21.
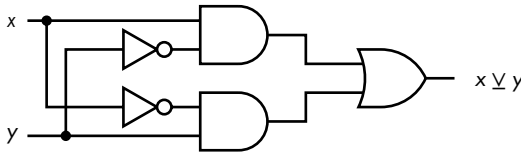

Figure 4-21: XOR gate made from AND, OR, and NOT gates

We can, of course, design many more Boolean operators. But we're going to move on in the next few chapters and see how these operators can be implemented in hardware. It's all done with simple on/off switches.

YOUR TURN

Design a function that will detect all the four-bit integers that are even.
Find a minimal sum of products expression for this function:

$$F(x,y,z) = (\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge y \wedge z)$$

Find a minimal product of sums expression for this function:

$$F(x,y,z) = (x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

The arrangement of the variables for a Karnaugh map is arbitrary, but the minterms (or maxterms) need to be consistent with the labeling. Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4-11.

$F(x, y, z)$

	xy			
	00	01	11	10

z 0
   1

The arrangement of the variables for a Karnaugh map is arbitrary, but the minterms (or maxterms) need to be consistent with the labeling. Show where each minterm is located with this Karnaugh map axis labeling using the notation of Figure 4-11.

$F(x, y, z)$

	xz			
	00	01	11	10

y 0
   1

Create a Karnaugh map for five variables. You'll probably need to review the Gray code in Table 4-10 and increase it to five bits.

Design a logic function that detects the single-digit prime numbers. Assume that the numbers are coded in four-bit BCD (see Table 2-7 in Chapter 2. The function is 1 for each prime number.

What You've Learned

Boolean operators   The basic Boolean operators are AND, OR, and NOT.

Rules of Boolean algebra   Boolean algebra provides a mathematical way to work with the rules of logic. AND works like multiplication and OR is similar to addition in elementary algebra.

Simplifying Boolean algebra expressions   Boolean functions specify the functionality of a computer. Simplifying these functions leads to a simpler hardware implementation.

Karnaugh maps   These provide a graphical way to simplify Boolean expressions.

Gray code   This shows how to order the cells in a Karnaugh map.

Combining basic Boolean operators   XOR can be created from AND, OR, and NOT.

The next chapter starts with an introduction to basic electronics that will provide a basis for understanding how transistors can be used to implement switches. From there, we'll look at how transistor switches are used to implement logic gates.