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

—Claude Shannon

CHAPTER 3 Boolean Algebra and Digital Logic

3.1 INTRODUCTION

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

Nearly 85 years later, John Vincent Atanasoff applied Boolean algebra to computing. He recounted the moment of his insight to Linda Null. At the time, Atanasoff was attempting to build a calculating machine based on the same technology used by Pascal and Babbage. His aim was to use this machine to solve systems of linear equations. After struggling with repeated failures, Atanasoff was so frustrated he decided to take a drive. He was living in Ames, Iowa, at the time, but found himself 200 miles away in Illinois before he suddenly realized how far he had driven.

Atanasoff had not intended to drive that far, but because he was in Illinois where it was legal to buy a drink in a tavern, he sat down and ordered a bourbon. He chuckled to himself when he realized that he had driven such a distance for a drink! Even more ironic is the fact that he never touched the drink. He felt he

needed a clear head to write down the revelations that came to him during his long, aimless journey. Exercising his physics and mathematics backgrounds and focusing on the failures of his previous computing machine, he made four critical breakthroughs necessary in the machine's new design.

He would use electricity instead of mechanical movements (vacuum tubes would allow him to do this).

Because he was using electricity, he would use base 2 numbers instead of base 10 (this correlated directly with switches that were either "on" or "off"), resulting in a digital, rather than an analog, machine.

He would use capacitors (condensers) for memory because they store electrical charges with a regenerative process to avoid power leakage.

Computations would be done by what Atanasoff termed "direct logical action" (which is essentially equivalent to Boolean algebra) and not by enumeration as all previous computing machines had done.

It should be noted that, at the time, Atanasoff did not recognize the application of Boolean algebra to his problem and that he devised his own direct logical action by trial and error. He was unaware that in 1938 Claude Shannon proved that two-valued Boolean algebra could describe the operation of two-valued electrical switching circuits. Today, we see the significance of Boolean algebra's application in the design of modern computing systems. It is for this reason that we include a chapter on Boolean logic and its relationship to digital computers.

This chapter contains a brief introduction to the basics of logic design. It provides minimal coverage of Boolean algebra and this algebra's relationship to logic gates and basic digital circuits. You may already be familiar with the basic Boolean operators from your previous programming experience. It is a fair question, then, to ask why you must study this material in more detail. The relationship between Boolean logic and the actual physical components of any computer system is very strong, as you will see in this chapter. As a computer scientist, you may never have to design digital circuits or other physical components—in fact, this chapter will not prepare you to design such items. Rather, it provides sufficient background for you to understand the basic motivation underlying computer design and implementation. Understanding how Boolean logic affects the design of various computer system components will allow you to use, from a programming perspective, any computer system more effectively. If you are interested in delving deeper, there are many resources listed at the end of the chapter to allow further investigation into these topics.

3.2 BOOLEAN ALGEBRA

Boolean algebra is an algebra for the manipulation of objects that can take on only two values, typically true and false, although it can be any pair of values. Because computers are built as collections of switches that are either "on" or "off," Boolean algebra is a very natural way to represent digital information. In reality, digital circuits use low and high voltages, but for our level of understanding, 0 and 1 will suffice. It is common to interpret the digital value 0 as false and the digital value 1 as true.

Inputs		Outputs
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 3.1 Truth Table for AND

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

TABLE 3.2 Truth Table for OR

3.2.1 Boolean Expressions

In addition to binary objects, Boolean algebra also has operations that can be performed on these objects, or variables. Combining the variables and operators yields **Boolean expressions**. A **Boolean function** typically has one or more input values and yields a result, based on these input values, in the set $\{0,1\}$.

Three common Boolean operators are **AND**, **OR**, and **NOT**. To better understand these operators, we need a mechanism to allow us to examine their behaviors. A Boolean operator can be completely described using a table that lists the inputs, all possible values for these inputs, and the resulting values of the operation for all possible combinations of these inputs. This table is called a **truth table**. A truth table shows the relationship, in tabular form, between the input values and the result of a specific Boolean operator or function on the input variables. Let's look at the Boolean operators AND, OR, and NOT to see how each is represented, using both Boolean algebra and truth tables.

The logical operator AND is typically represented by either a dot or no symbol at all. For example, the Boolean expression xy is equivalent to the expression $x \cdot y$ and is read "x and y." The expression xy is often referred to as a **Boolean product**. The behavior of this operator is characterized by the truth table shown in Table 3.1.

The result of the expression xy is 1 only when both inputs are 1, and 0 otherwise. Each row in the table represents a different Boolean expression, and all possible combinations of values for x and y are represented by the rows in the table.

The Boolean operator OR is typically represented by a plus sign. Therefore, the expression $x + y$ is read "x or y." The result of $x + y$ is 0 only when both of its input values are 0. The expression $x + y$ is often referred to as a **Boolean sum**. The truth table for OR is shown in Table 3.2.

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

We now understand that Boolean algebra deals with binary variables and logical operations on those variables. Combining these two concepts, we can examine

Inputs	Outputs
x	$\bar{x}$
0	1
1	0

TABLE 3.3 Truth Table for NOT

Inputs					Outputs
x	y	z	$\bar{y}$	$\bar{y}z$	$x + \bar{y}z = F$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

TABLE 3.4 The Truth Table for $F(x,y,z) = x + \bar{y}z$

Boolean expressions composed of Boolean variables and multiple logic operators. For example, the Boolean function:

$$F(x, y, z) = x + \bar{y}z$$

is represented by a Boolean expression involving the three Boolean variables x, y, and z and the logical operators OR, NOT, and AND. How do we know which operator to apply first? The rules of precedence for Boolean operators give NOT top priority, followed by AND, and then OR. For our previous function F, we would negate y first, then perform the AND of $\bar{y}$ and z, and last OR this result with x.

We can also use a truth table to represent this expression. It is often helpful, when creating a truth table for a more complex function such as this, to build the table representing different pieces of the function, one column at a time, until the final function can be evaluated. The truth table for our function F is shown in Table 3.4.

The last column in the truth table indicates the values of the function for all possible combinations of x, y, and z. We note that the real truth table for our function F consists of only the first three columns and the last column. The shaded columns show the intermediate steps necessary to arrive at our final answer. Creating truth tables in this manner makes it easier to evaluate the function for all possible combinations of the input values.

3.2.2 Boolean Identities

Frequently, a Boolean expression is not in its simplest form. Recall from algebra that an expression such as $2x + 6x$ is not in its simplest form; it can be reduced (represented by fewer or simpler terms) to $8x$. Boolean expressions can also be simplified, but we need new **identities**, or laws, that apply to Boolean algebra instead of regular algebra. These identities, which apply to single Boolean variables as well as Boolean expressions, are listed in Table 3.5. Note that each relationship (with the exception of the last one) has both an AND (or product) form and an OR (or sum) form. This is known as the **duality principle**.

The Identity Law states that any Boolean variable ANDed with 1 or ORed with 0 simply results in the original variable (1 is the identity element for AND; 0 is the identity element for OR). The Null Law states that any Boolean variable

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)\ (x+z)$	$x(y+z) = xy + xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

TABLE 3.5 Basic Identities of Boolean Algebra

ANDed with 0 is 0, and a variable ORed with 1 is always 1. The Idempotent Law states that ANDing or ORing a variable with itself produces the original variable. The Inverse Law states that ANDing or ORing a variable with its complement produces the identity for that given operation. You should recognize these as the Commutative and Associative Laws from algebra. Boolean variables can be reordered (commuted) and regrouped (associated) without affecting the final result. The Distributive Law shows how OR distributes over AND and vice versa.

The Absorption Law and DeMorgan's Law are not so obvious, but we can prove these identities by creating a truth table for the various expressions: If the right-hand side is equal to the left-hand side, the expressions represent the same function and result in identical truth tables. Table 3.6 depicts the truth table for both the left-hand side and the right-hand side of DeMorgan's Law for AND. It is left as an exercise to prove the validity of the remaining laws, in particular, the OR form of DeMorgan's Law and both forms of the Absorption Law.

The Double Complement Law formalizes the idea of the double negative, which evokes rebuke from high school teachers. The Double Complement Law can be useful in digital circuits as well as in your life. For example, let x be the amount of cash you have (assume a positive quantity). If you have no cash, you have $\bar{x}$. When an untrustworthy acquaintance asks to borrow some cash, you can truthfully say that you don't have no money. That is, $x = (\bar{\bar{x}})$ even if you just got paid.

One of the most common errors that beginners make when working with Boolean logic is to assume the following:

$(\overline{xy}) = \bar{x}\,\bar{y}$ Please note that this is not a valid equality!

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

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

DeMorgan's Law clearly indicates that this statement is incorrect; however, it is a very easy mistake to make, and one that should be avoided.

3.2.3 Simplification of Boolean Expressions

The algebraic identities we studied in algebra class allow us to reduce algebraic expressions (such as $10x + 2y - x + 3y$) to their simplest forms ($9x + 5y$). The Boolean identities can be used to simplify Boolean expressions in a similar fashion. We apply these identities in the following examples.

EXAMPLE 3.1 Suppose we have the function $F(x, y) = xy + xy$. Using the OR form of the Idempotent Law and treating the expression xy as a Boolean variable, we simplify the original expression to xy. Therefore, $F(x,y) = xy + xy = xy$.

EXAMPLE 3.2 Given the function $F(x, y, z) = \bar{x}yz + \bar{x}y\bar{z} + xz$, we simplify as follows:

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

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

EXAMPLE 3.3 Given the function $F(x,y,z) = xy + \bar{x}z + yz$, we simplify as follows:

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

Example 3.3 illustrates what is commonly known as the **Consensus Theorem**.

How did we know to insert additional terms to simplify the function? Unfortunately, there is no defined set of rules for using these identities to minimize a Boolean expression; it is simply something that comes with experience. There are other methods that can be used to simplify Boolean expressions; we mention these later in this section.

We can also use these identities to prove Boolean equalities. Suppose we want to prove that $(x + y)(\bar{x} + y) = y$. We can do so as follows:

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

To prove the equality of two Boolean expressions, you can also create the truth tables for each and compare. If the truth tables are identical, the expressions are equal. We leave it as an exercise to find the truth tables for the equality just proven.

3.2.4 Complements

As you saw in Example 3.1, the Boolean identities can be applied to Boolean expressions, not simply Boolean variables (we treated xy as a Boolean variable and then applied the Idempotent Law). The same is true for the Boolean operators. The most common Boolean operator applied to more complex Boolean expressions is the NOT operator, resulting in the **complement** of the expression. Quite often, it is cheaper and less complicated to implement the complement of a function rather than the function itself. If we implement the complement, we must invert the final output to yield the original function; this is accomplished with one simple NOT operation. Therefore, complements are quite useful.

To find the complement of a Boolean function, we use DeMorgan's Law. The OR form of this law states that $\overline{(x + y)} = \bar{x}\bar{y}$. We can easily extend this to three or more variables as follows:

Given the function:

$F(x, y, z) = (x + y + z)$. Then $\bar{F}(x, y, z) = \overline{(x + y + z)}$.

Let $w = (x + y)$. Then $\bar{F}(x, y, z) = \overline{(w + z)} = \bar{w}\,\bar{z}$.

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

$\bar{w}\,\bar{z} = \overline{(x + y)}\bar{z} = \bar{x}\,\bar{y}\,\bar{z} = \bar{F}(x, y, z)$.