4

BOOLEAN FUNCTIONS

What To Expect

In 1854, George Boole introduced an algebra system designed to work with binary (or base 2) numbers, but he could not have foreseen the immense impact his work would have on systems like telephony and computer science. This chapter includes the following topics.

- Developing and using the primary Logic Functions: AND, OR, NOT
- Developing and using the secondary Logic Functions: XOR, XNOR, NAND, NOR, Buffer
- Developing and using the univariate Boolean Algebra Properties: Identity, Idempotence, Annihilator, Complement, Involution
- Developing and using the multivariate Boolean Algebra Properties: Commutative, Associative, Distributive, Absorption, Adjacency
- Analyzing circuits using DeMorgan's Theorem

4.1 INTRODUCTION TO BOOLEAN FUNCTIONS

Before starting a study of Boolean functions, it is important to keep in mind that this mathematical system concerns electronic components that are capable of only two states: *True* and *False* (sometimes called *High-Low* or 1 - 0). Boolean functions are based on evaluating a series of True-False statements to determine the output of a circuit.

For example, a Boolean expression could be created that would describe "If the floor is dirty OR company is coming THEN I will mop." (The things that I do for visiting company!) These types of logic statements are often represented symbolically using the symbols 1 and 0, where 1 stands for *True* and 0 stands for *False*. So, let "Floor is dirty" equal 1 and "Floor is not dirty" equal 0. Also, let "Company is coming" equal 1 and "Company is not coming" equal 0. Then, "Floor is dirty OR company is coming" can be symbolically represented by 1 OR 1. Within the discipline of Boolean algebra, common mathematical symbols are

used to represent Boolean expressions; for example, Boolean OR is frequently represented by a mathematics plus sign, as shown below.

$$\begin{aligned} 0+0&=0 \\ 0+1&=1 \\ 1+0&=1 \\ 1+1&=1 \end{aligned} \tag{4.1}$$

These look like addition problems, but they are not (as evidenced by the last line). It is essential to keep in mind that these are merely symbolic representations of *True-False* statements. The first three lines make perfect sense and look like elementary addition. The last line, though, violates the principles of addition for real numbers; but it is a perfectly valid Boolean expression. Remember, in the world of Boolean algebra, there are only two possible values for any quantity: 1 or 0; and that last line is actually saying *True OR True is True*. To use the dirty floor example from above, "The floor is dirty" (*True*) OR "Company is coming" (*True*) SO "I will mop the floor" (*True*) is symbolized by: $1+1=1$. This could be expressed as *T OR T SO T*; but the convention is to use common mathematical symbols; thus: $1+1=1$.

Moreover, it does not matter how many or few terms are OR ed together; if just one is *True*, then the output is *True*, as illustrated below:

$$\begin{aligned} 1+1+1&=1 \\ 1+0+0+0+0&=1 \\ 1+1+1+1+1+1&=1 \end{aligned} \tag{4.2}$$

Next, consider a very simple electronic sensor in an automobile: IF the headlights are on AND the driver's door is open THEN a buzzer will sound. In the same way that the plus sign is used to mathematically represent OR , a times sign is used to represent AND . Therefore, using common mathematical symbols, this automobile alarm circuit would be represented by $1X1=1$. The following list shows all possible states of the headlights and door:

$$\begin{aligned} 0X0&=0 \\ 0X1&=0 \\ 1X0&=0 \\ 1X1&=1 \end{aligned} \tag{4.3}$$

The first row above shows "*False* (the lights are not on) AND *False* (the door is not open) results in *False* (the alarm does not sound)". For AND logic, the only time the output is *True* is when all inputs are also

True; therefore: 1X1X1X1X0 = 0. In this way, Boolean AND behaves somewhat like algebraic multiplication.

Within Boolean algebra's simple *True-False* system, there is no equivalent for subtraction or division, so those mathematical symbols are not used. Like real-number algebra, Boolean algebra uses alphabetical letters to denote variables; however, Boolean variables are always CAPITAL letters, never lower-case, and normally only a single letter. Thus, a Boolean equation would look something like this:

$$A + B = Y \tag{4.4}$$

As Boolean expressions are realized (that is, turned into a real, or physical, circuit), the various operators become *gates*. For example, the above equation would be realized using an OR gate with two inputs (labeled A and B) and one output (labeled Y).

Boolean algebra includes three primary and four secondary logic operations (plus a Buffer, which has no logic value), six univariate, and six multivariate properties. All of these will be explored in this chapter.

4.2 PRIMARY LOGIC OPERATIONS

4.2.1 *AND*

An AND gate is a Boolean operation that will output a logical one, or *True*, only if all of the inputs are *True*. As an example, consider this statement: "If I have ten bucks AND there is a good movie at the cinema, then I will go see the movie." In this statement, "If I have ten bucks" is one variable and "there is a good movie at the cinema" is another variable. If both of these inputs are *True*, then the output variable ("I will go see the movie") will also be *True*. However, if either of the two inputs is *False*, then the output will also be *False* (or, "I will not go see the movie"). Of course, if I want popcorn, I would need another ten spot, but that is not germane to this example. When written in an equation, the Boolean AND term is represented a number of different ways. One method is to use the logic AND symbol as found in Equation **??**.

$$A \wedge B = Y \tag{4.5}$$

One other method is to use the same symbols that are used for multiplication in traditional algebra; that is, by writing the variables next to each other, with parenthesis, or, sometimes, with an asterisk between them, as in Equation 4.6.

$$AB = Y \tag{4.6}$$
$$(A)(B) = Y$$
$$A * B = Y$$

The multiplication symbols X and • (dot) are not commonly used in digital logic equations.

Logic AND is normally represented in equations by using an algebra multiplication symbol since it is easy to type; however, if there is any chance for ambiguity, then the Logic AND symbol ($\wedge$) can be used to differentiate between multiplication and a logic AND function.

Following is the truth table for the AND operator.

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 4.1: Truth Table for AND

A truth table is used to record all possible inputs and the output for each combination of inputs. For example, in the first line of Table 4.1, if input A is 0 and input B is 0 then the output, Y, will be 0. All possible input combinations are normally formed in a truth table by counting in binary starting with all variables having a 0 value to all variables having a 1 value. Thus, in Table 4.1, the inputs are 00, 01, 10, 11. Notice that the output for the AND operator is *False* (that is, 0) until the last row, when both inputs are *True*. Therefore, it could be said that just one *False* input would inactivate a physical AND gate. For that reason, an AND operation is sometimes called an *inhibitor*.

AND Gate Switches

Because a single *False* input can turn an AND gate off, these types of gates are frequently used as a switch in a logic circuit. As a simple example, imagine an assembly line where there are four different safety sensors of some sort. The sensor outputs could be routed to a single four-input AND gate and then as long as all sensors are *True* the assembly line motor will run. If, however, any one of those sensors goes *False* due to some unsafe condition, then the AND gate would also output a *False* and cause the motor to stop.

Logic gates are realized (or created) in electronic circuits by using transistors, resistors, and other components. These components are

normally packaged into a single IC "chip," so the logic circuit designer does not need to know all of the details of the electronics in order to use the gate. In logic diagrams, an AND gate is represented by a shape that looks like a capital D. In Figure 4.1, the input variables A and B are wired to an AND gate and the output from that gate goes to Y.

Figure 4.1: AND Gate

Notice that each input and output is named in order to make it easier to describe the circuit algebraically. In reality, AND gates are not packaged or sold one at a time; rather, several would be placed on a single IC, like the *7408 Quad AND Gate*. The designer would design the circuit card to use whichever of the four gates are needed while leaving the unused gates unconnected.

There are two common sets of symbols used to represent the various elements in logic diagrams, and whichever is used is of little consequence since the logic is the same. Shaped symbols, as used in Figure 4.1, are more common in the United States; but the IEEE has its own symbols which are sometimes used, especially in Europe. Figure 4.2 illustrates the circuit in Figure 4.1 using IEEE symbols:

In this book, only shaped symbols will be used.

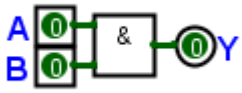

Figure 4.2: AND Gate Using IEEE Symbols

As an example of an AND gate at work, consider an elevator: if the door is closed (logic 1) AND someone in the elevator car presses a floor button (logic 1), THEN the elevator will move (logic 1). If both sensors (door and button) are input to an AND gate, then the elevator motor will only operate if the door is closed AND someone presses a floor button.

4.2.2 *OR*

An OR gate is a Boolean operation that will output a logical one, or *True*, if any or all of the inputs are *True*. As an example, consider this statement: "If my dog needs a bath OR I am going swimming, then I will put on a bathing suit." In this statement, "if my dog needs a bath" is one input variable and "I am going swimming" is another input

variable. If either of these is *True*, then the output variable, "I will put on a bathing suit," will also be *True*. However, if both of the inputs are *False*, then the output will also be *False* (or, "I will not put on a bathing suit"). If you think it odd that I would wear a bathing suit to bathe my dog then you have obviously never met my dog.

When written in an equation, the Boolean OR term is represented a number of different ways. One method is to use the logic OR symbol, as found in Equation 4.7.

$$A \vee B = Y \tag{4.7}$$

A more common method is to use the *plus* sign that is used for addition in traditional algebra, as in Equation 4.8.

$$A + B = Y \tag{4.8}$$

For simplicity, the mathematical *plus* symbol is normally used to indicate OR in printed material since it is easy to enter with a keyboard; however, if there is any chance for ambiguity, then the logic OR symbol ($\vee$) is used to differentiate between addition and logic OR.

Table 4.2 is the truth table for an OR operation.

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.2: Truth Table for OR

Notice for the OR truth table that the output is *True* (1) whenever at least one input is *True*. Therefore, it could be said that one *True* input would activate an OR Gate. In the following diagram, the input variables A and B are wired to an OR gate, and the output from that gate goes to Y.

Figure 4.3: OR Gate

As an example of an OR gate at work, consider a traffic signal. Suppose an intersection is set up such that the light for the main road

is normally green; however, if a car pulls up to the intersection from the crossroad, or if a pedestrian presses the "cross" button, then the main light is changed to red to stop traffic. This could be done with a simple OR gate. An automobile sensor on the crossroad would be one input and the pedestrian "cross" button would be the other input; the output of the OR gate connecting these two inputs would change the light to red when either input is activated.

4.2.3 *NOT*

NOT (or *inverter*) is a Boolean operation that inverts the input. That is, if the input is *True* then the output will be *False* or if the input is *False* then the output will be *True*. When written in an equation, the Boolean NOT operator is represented in many ways, though two are most popular. The older method is to overline (that is, a line above) a term, or group of terms, that are to be inverted, as in Equation 4.9.

Equation 4.9 is read A OR B NOT = Q *(notice that when spoken, the word* not *follows the term that is inverted).*

$$A + \overline{B} = Y \tag{4.9}$$

Another method of indicating NOT is to use the algebra *prime* indicator, an apostrophe, as in Equation 4.10.

$$A + B' = Y \tag{4.10}$$

The reason that NOT is most commonly indicated with an apostrophe is because that is easier to enter on a computer keyboard. There are many other ways authors use to represent NOT in a formula, but none are considered standardized. For example, some authors use an exclamation point: A+!B = Q, others use a broken line: $A + \neg B = Q$, others use a backslash: $A + \backslash B = Q$, and still others use a tilde: $A + \sim B = Q$. However, only the apostrophe and overline are consistently used to indicate NOT. Table 4.3 is the truth table for NOT:

Input	Output
0	1
1	0

Table 4.3: Truth Table for NOT

In a logic diagram, NOT is represented by a small triangle with a "bubble" on the output. In Figure 4.4, the input variable A is inverted by a NOT gate and then sent to output Y.

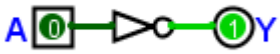

Figure 4.4: NOT Gate

4.3 SECONDARY LOGIC FUNCTIONS

4.3.1 *NAND*

NAND is a Boolean operation that outputs the opposite of AND, that is, NOT AND; thus, it will output a logic *False* only if all of the inputs are *True*. The NAND operation is not often used in Boolean equations, but when necessary it is represented by a vertical line. Equation 4.11 shows a NAND operation.

$$A|B = Y \tag{4.11}$$

Table 4.4 is the Truth Table for a NAND gate.

Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Table 4.4: Truth Table for NAND Gate

In Figure 4.5, the input variables A and B are wired to a NAND gate, and the output from that gate goes to Y.

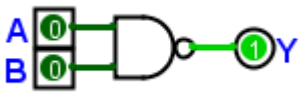

Figure 4.5: NAND Gate

Inverting bubbles are never found by themselves on a wire; they are always associated with either the inputs or output of a logic gate. To invert a signal on a wire, a NOT gate is used.

The logic diagram symbol for a NAND gate looks like an AND gate, but with a small bubble on the output port. A bubble in a logic diagram always represents some sort of signal inversion, and it can appear at the inputs or outputs of nearly any logic gate. For example, the bubble on a NAND gate could be interpreted as "take whatever the output would be generated by an AND gate—then invert it."

4.3.2 *NOR*

NOR is a Boolean operation that is the opposite of OR, that is, NOT OR; thus, it will output a logic *True* only if all of the inputs are *False*. The NOR operation is not often used in Boolean equations, but when necessary it is represented by a downward-pointing arrow. Equation 4.12 shows a NOR operation.

$$A \downarrow B = Y \tag{4.12}$$

Table 4.5 is the truth table for NOR .

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Table 4.5: Truth Table for NOR

In Figure 4.6, the input variables A and B are wired to a NOR gate, and the output from that gate goes to Y.

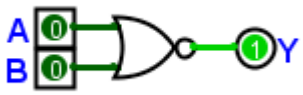

Figure 4.6: NOR Gate

4.3.3 *XOR*

XOR (*Exclusive OR*) is a Boolean operation that outputs a logical one, or *True*, only if the two inputs are different. This is useful for circuits that compare inputs; if they are different then the output is *True*, otherwise it is *False*. Because of this, an XOR gate is sometimes referred to as a *Difference Gate*. The XOR operation is not often used in Boolean equations, but when necessary it is represented by a plus sign (like the OR function) inside a circle. Equation 4.13 shows an XOR operation.

$$A \oplus B = Y \tag{4.13}$$

Table 4.6 is the truth table for an XOR gate.

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.6: Truth Table for XOR

In Figure 4.7, the input variables A and B are wired to an XOR gate, and the output from that gate goes to Y.

Figure 4.7: XOR Gate

There is some debate about the proper behavior of an XOR gate that has more than two inputs. Some experts believe that an XOR gate should output a *True* if one, and only one, input is *True* regardless of the number of inputs. This would seem to be in keeping with the rules of digital logic developed by George Boole and other early logisticians and is the strict definition of XOR promulgated by the IEEE. This is also the behavior of the XOR gate found in *Logisim-evolution*, the digital logic simulator used in the lab manual accompanying this text. Others believe, though, that an XOR gate should output a *True* if an odd number of inputs is *True*. in *Logisim-evolution* this type of behavior is found in a device called a "parity gate" and is covered in more detail elsewhere in this book.

4.3.4 *XNOR*

XNOR is a Boolean operation that will output a logical one, or *True*, only if the two inputs are the same; thus, an XNOR gate is often referred to as an *Equivalence Gate*. The XNOR operation is not often used in Boolean equations, but when necessary it is represented by a dot inside a circle. Equation 4.14 shows an XNOR operation.

$$A \odot B = Y \tag{4.14}$$

Table 4.7 is the truth table for XNOR .

Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Table 4.7: Truth Table for XNOR

In Figure 4.8, the input variables A and B are wired to an XNOR gate, and the output from that gate goes to Y.

Figure 4.8: XNOR Gate

4.3.5 *Buffer*

A buffer (sometimes called *Transfer*) is a Boolean operation that transfers the input to the output without change. If the input is *True*, then the output will be *True* and if the input is *False*, then the output will be *False*. It may seem to be an odd function since this operation does not change anything, but it has an important use in a circuit. As logic circuits become more complex, the signal from input to output may become weak and no longer able to drive (or activate) additional gates. A buffer is used to boost (and stabilize) a logic level so it is more dependable. Another important function for a buffer is to clean up an input signal. As an example, when an electronic circuit interacts with the physical world (such as a user pushing a button), there is often a very brief period when the signal from that physical device waivers between high and low unpredictably. A buffer can smooth out that signal so it is a constant high or low without voltage spikes in between.

Table 4.8 is the truth table for buffer.

Input	Output
0	0
1	1

Table 4.8: Truth Table for a Buffer

Buffers are rarely used in schematic diagrams since they do not actually change a signal; however, Figure 4.9, illustrates a buffer.

Figure 4.9: Buffer

4.4 UNIVARIATE BOOLEAN ALGEBRA PROPERTIES

4.4.1 *Introduction*

Boolean Algebra, like real number algebra, includes a number of properties. This unit introduces the univariate Boolean properties, or those properties that involve only one input variable. These properties permit Boolean expressions to be simplified, and circuit designers are interested in simplifying circuits to reduce construction expense, power consumption, heat loss (wasted energy), and troubleshooting time.

4.4.2 *Identity*

In mathematics, an identity is an equality where the left and right members are the same regardless of the values of the variables present. As an example, Equation 4.15 is an identity since the two members are identical regardless of the value of α:

$$\frac{\alpha}{2} = 0.5\alpha \tag{4.15}$$

An *Identity Element* is a special member of a set such that when that element is used in a binary operation the other element in that operation is not changed. This is sometimes called the *Neutral Element* since it has no effect on binary operations. As an example, in Equation 4.16 the two members of the equation are always identical. Therefore, zero is the identity element for addition since anything added to zero remains unchanged.

$$a + 0 = a \tag{4.16}$$

In a logic circuit, combining any logic input with a logic zero through an OR gate yields the original input. Logic zero, then, is considered the OR identity element because it causes the input of the gate to be copied to the output unchanged. Because OR is represented by a plus sign when written in a Boolean equation, and the identity element for OR is zero, Equation 4.17 is *True*.

$$A + 0 = A \tag{4.17}$$

The bottom input to the OR gate in 4.10 is a constant logic zero, or *False*. The output for this circuit, Y, will be the same as input A; therefore, the identity element for OR is zero.

Figure 4.10: OR Identity Element

In the same way, combining any logic input with a logic one through an AND gate yields the original input. Logic one, then, is considered the AND identity element because it causes the input of the gate to be copied to the output unchanged. Because AND is represented by a multiplication sign when written in a Boolean equation, and the identity element for AND is one, Equation 4.18 is *True*.

$$A * 1 = A \tag{4.18}$$

The bottom input to the AND gate in 4.11 is a constant logic one, or *True*. The output for this circuit, Y, will be the same as input A; therefore, the identity element for AND is one.

Figure 4.11: AND Identity Element

4.4.3 *Idempotence*

If the two inputs of either an OR or AND gate are tied together, then the same signal will be applied to both inputs. This results in the output of either of those gates being the same as the input; and this is called the idempotence property. An electronic gate wired in this manner performs the same function as a buffer.

$$A + A = A \tag{4.19}$$

Remember that in Boolean expressions a plus sign represents an OR gate, not mathematical addition.

Figure 4.12: Idempotence Property for OR Gate

Remember that in a Boolean expression a multiplication sign represents an AND gate, not mathematical multiplying.

Figure 4.12 illustrates the idempotence property for an AND gate.

$$A * A = A \tag{4.20}$$

Figure 4.13: Idempotence Property for AND Gate

4.4.4 *Annihilator*

Combining any data and a logic one through an OR gate yields a constant output of one. This property is called the annihilator since the OR gate outputs a constant one; in other words, whatever other data were input are lost. Because OR is represented by a plus sign when written in a Boolean equation, and the annihilator for OR is one, the following is true:

$$A + 1 = 1 \tag{4.21}$$

Figure 4.14: Annihilator For OR Gate

The bottom input for the OR gate in Figure 4.14 is a constant logic one, or *True*. The output for this circuit will be *True* (or 1) no matter whether input A is *True* or *False* (1 or 0).

Combining any data and a logic zero with an AND gate yields a constant output of zero. This property is called the annihilator since the AND gate outputs a constant zero; in other words, whatever logic data were input are lost. Because AND is represented by a multiplication sign when written in a Boolean equation, and the annihilator for AND is zero, the following is true:

$$A * 0 = 0 \tag{4.22}$$

Figure 4.15: Annihilator For AND Gate

The bottom input for the AND gate in Figure 4.15 is a constant logic zero, or *False*. The output for this circuit will be *False* (or 0) no matter whether input A is *True* or *False* (1 or 0).

4.4.5 *Complement*

In Boolean logic there are only two possible values for variables: 0 and 1. Since either a variable or its complement must be one, and since combining any data with one through an OR gate yields one (see the Annihilator in Equation 4.21), then the following is true:

$$A + A' = 1 \tag{4.23}$$

Figure 4.16: OR Complement

In Figure 4.16, the output (Y) will always equal one, regardless of the value of input A. This leads to the general property that when a variable and its complement are combined through an OR gate the output will always be one.

In the same way, since either a variable or its complement must be zero, and since combining any data with zero through an AND gate yields zero (see the Annihilator in Equation 4.22), then the following is true:

$$A * A' = 0 \tag{4.24}$$

Figure 4.17: AND Complement

4.4.6 *Involution*

The Involution Property is sometimes called the "Double Complement" Property.

Another law having to do with complementation is that of Involution. Complementing a Boolean variable two times (or any even number of times) results in the original Boolean value.

$$(A')' = A \tag{4.25}$$

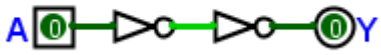

Figure 4.18: Involution Property

In the circuit illustrated in Figure 4.18, the output (Y) will always be the same as the input (A).

Propagation Delay

It takes the two NOT gates a short period of time to pass a signal from input to output, which is known as "propagation delay." A designer occasionally needs to build an intentional signal delay into a circuit for some reason and two (or any even number of) consecutive NOT gates would be one option.

4.5 MULTIVARIATE BOOLEAN ALGEBRA PROPERTIES

4.5.1 *Introduction*

Boolean Algebra, like real number algebra, includes a number of properties. This unit introduces the multivariate Boolean properties, or those properties that involve more than one input variable. These properties permit Boolean expressions to be simplified, and circuit designers are interested in simplifying circuits to reduce construction expense, power consumption, heat loss (wasted energy), and troubleshooting time.

4.5.2 *Commutative*

In essence, the commutative property indicates that the order of the input variables can be reversed in either OR or AND gates without changing the truth of the expression. Equation 4.26 expresses this property algebraically.

The examples here show only two variables, but this property is true for any number of variables.

$$A + B = B + A \qquad (4.26)$$
$$A * B = B * A$$

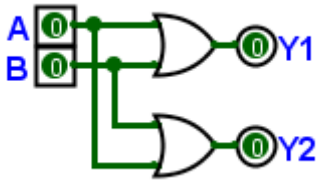

Figure 4.19: Commutative Property for OR

XOR and XNOR are also commutative; but for only two variables, not three or more.

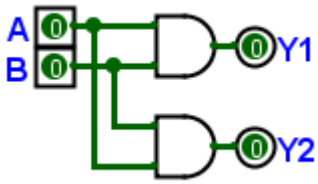

Figure 4.20: Commutative Property for AND

In Figures 4.19 and 4.20 the inputs are reversed for the two gates, but the outputs are the same. For example, A is entering the top input for the upper OR gate, but the bottom input for the lower gate; however, Y1 is always equal to Y2.

4.5.3 *Associative*

This property indicates that groups of variables in an OR or AND gate can be associated in various ways without altering the truth of the equations. Equation 4.27 expresses this property algebraically:

The examples here show only three variables, but this property is true for any number of variables.

$$(A + B) + C = A + (B + C) \qquad (4.27)$$
$$(A * B) * C = A * (B * C)$$

XOR and XNOR are also associative; but for only two variables, not three or more.

In the circuits in Figure 4.21 and 4.22 , notice that A and B are associated together in the first gate, and then C is associated with the

output of that gate. Then, in the lower half of the circuit, B and C are associated together in the first gate, and then A is associated with the output of that gate. Since Y1 is always equal to Y2 for any combination of inputs, it does not matter which of the two variables are associated together in a group of gates.

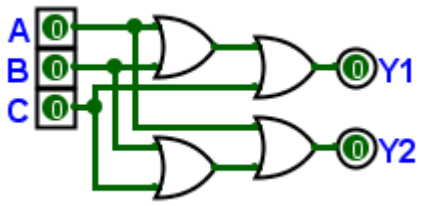

Figure 4.21: Associative Property for OR

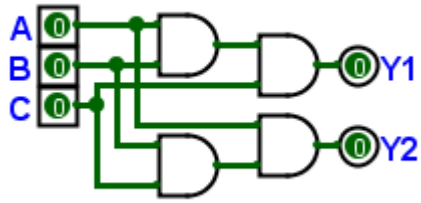

Figure 4.22: Associative Property for AND

4.5.4 *Distributive*

The distributive property of real number algebra permits certain variables to be "distributed" to other variables. This operation is frequently used to create groups of variables that can be simplified; thus, simplifying the entire expression. Boolean algebra also includes a distributive property, and that can be used to combine OR or AND gates in various ways that make it easier to simplify the circuit. Equation 4.28 expresses this property algebraically:

$$A(B+C) = AB + AC \tag{4.28}$$
$$A + (BC) = (A+B)(A+C)$$

In the circuits illustrated in Figures 4.23 and 4.24, notice that input A in the top half of the circuit is distributed to inputs B and C in the bottom half. However, output Y1 is always equal to output Y2 regardless of how the inputs are set. These two circuits illustrate Distributive of AND over OR and Distributive of OR over AND .

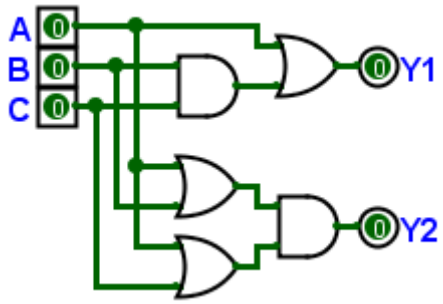

Figure 4.23: Distributive Property for AND over OR

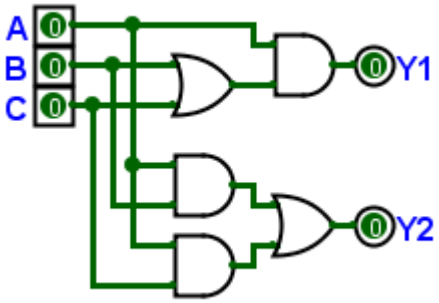

Figure 4.24: Distributive Property for OR over AND

4.5.5 *Absorption*

The absorption property is used to remove logic gates from a circuit if those gates have no effect on the output. In essence, a gate is "absorbed" if it is not needed. There are two different absorption properties:

$$\begin{aligned} A + (AB) &= A \\ A(A + B) &= A \end{aligned} \tag{4.29}$$

The best way to think about why these properties are true is to imagine a circuit that contains them. The first circuit below illustrates the top equation.

Figure 4.25: Absorption Property (Version 1)

Table 4.9 is the truth table for the circuit in Figure 4.25.

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	1
1	1	1

Table 4.9: Truth Table for Absorption Property

Notice that the output, Y, is always the same as input A. This means that input B has no bearing on the output of the circuit; therefore, the circuit could be replaced by a piece of wire from input A to output Y. Another way to state that is to say that input B is absorbed by the circuit.

The circuit illustrated in Figure 4.26 is the second version of the Absorption Property. Like the first Absorption Property circuit, a truth table would demonstrate that input B is absorbed by the circuit.

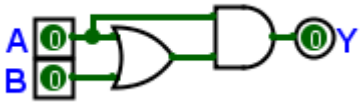

Figure 4.26: Absorption Property (Version 2)

4.5.6 *Adjacency*

The adjacency property simplifies a circuit by removing unnecessary gates.

$$AB + AB' = A \tag{4.30}$$

This property can be proven by simple algebraic manipulation:

$$\begin{aligned} AB + AB' &\quad \text{Original Expression} \\ A(B + B') &\quad \text{Distributive Property} \\ A1 &\quad \text{Complement Property} \\ A &\quad \text{Identity Element} \end{aligned} \tag{4.31}$$

The circuit in Figure 4.27 illustrates the adjacency property. If this circuit were constructed it would be seen that the output, Y, is always the same as input A; therefore, this entire circuit could be replaced by a single wire from input A to output Y.

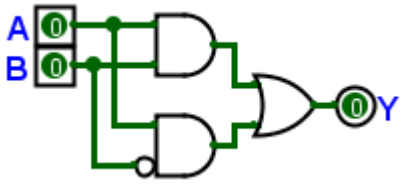

Figure 4.27: Adjacency Property

4.6 DEMORGAN'S THEOREM

4.6.1 *Introduction*

A mathematician named Augustus DeMorgan developed a pair of important theorems regarding the complementation of groups in Boolean algebra. DeMorgan found that an OR gate with all inputs inverted (a Negative-OR gate) behaves the same as a NAND gate with non-inverted inputs; and an AND gate with all inputs inverted (a Negative-AND gate) behaves the same as a NOR gate with non-inverted inputs. DeMorgan's theorem states that inverting the output of any gate is the same as using the opposite type of gate with inverted inputs. Figure 4.28 illustrates this in circuit terms: the NAND gate with normal inputs and the OR gate with inverted inputs are functionally equivalent; that is, Y1 will always equal Y2, regardless of the values of input A or B.

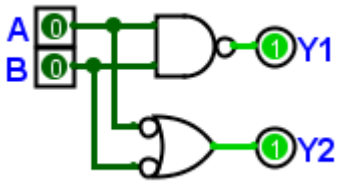

Figure 4.28: DeMorgan's Theorem Defined

The NOT function is commonly represented in an equation as an apostrophe because it is easy to enter with a keyboard, like: $(AB)'$ for *A AND B NOT*. However, it is easiest to work with DeMorgan's theorem if NOT is represented by an overline rather than an apostrophe, so it would be written as $\overline{AB}$ rather than $(AB)'$. Remember that an overline is a grouping symbol (like parenthesis) and it means that everything under that bar would first be combined (using an AND or OR gate) and then the output of the combination would be complemented.

4.6.2 *Applying DeMorgan's Theorem*

Applying DeMorgan's theorem to a Boolean expression may be thought of in terms of *breaking the bar*. When applying DeMorgan's theorem to a Boolean expression:

1. A complement bar is broken over a group of variables.
2. The operation (AND or OR) directly underneath the broken bar changes.
3. Pieces of the broken bar remain over the individual variables.

To illustrate:

$$\overline{A * B} \leftrightarrow \overline{A} + \overline{B} \tag{4.32}$$

$$\overline{A + B} \leftrightarrow \overline{A} * \overline{B} \tag{4.33}$$

Equation 4.32 shows how a two-input NAND gate is "broken" to form an OR gate with two inverted inputs and equation 4.33 shows how a two-input NOR gate is "broken" to form an AND gate with two complemented inputs.

4.6.3 *Simple Example*

When multiple "layers" of bars exist in an expression, only one bar is broken at a time, and the longest, or uppermost, bar is broken first. As an example, consider the circuit in Figure 4.29:

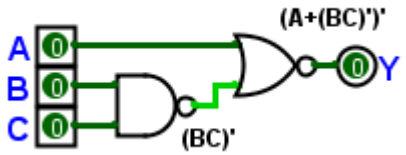

Figure 4.29: DeMorgan's Theorem Example 1

By writing the output at each gate (as illustrated in Figure 4.29), it is easy to determine the Boolean expression for the circuit. Note: all circuit diagrams in this book are generated with *Logisim-evolution* and the text tool in that software does not permit drawing overbars. Therefore, the circuit diagram will use the apostrophe method of indicating NOT but overbars will be used in the text.

$$\overline{A + \overline{BC}} \tag{4.34}$$

To simplify the circuit, break the bar covering the entire expression (the "longest bar"), and then simplify the resulting expression.

$$\begin{aligned} \overline{A + \overline{BC}} &\quad \text{Original Expression} \\ \overline{A}\,\overline{\overline{BC}} &\quad \text{"Break" the longer bar} \\ \overline{A}BC &\quad \text{Involution Property} \end{aligned} \tag{4.35}$$

As a result, the original circuit is reduced to a three-input AND gate with one inverted input.

4.6.4 *Incorrect Application of DeMorgan's Theorem*

More than one bar is never broken in a single step, as illustrated in Equation 4.36:

$$\begin{aligned} \overline{A + \overline{BC}} &\quad \text{Original Expression} \\ \overline{A\overline{B}} + \overline{\overline{C}} &\quad \text{Improperly Breaking Two Bars} \\ \overline{A}B + C &\quad \text{Incorrect Solution} \end{aligned} \tag{4.36}$$

Thus, as tempting as it may be to take a shortcut and break more than one bar at a time, it often leads to an incorrect result. Also, while it is possible to properly reduce an expression by breaking the short bar first; more steps are usually required and that process is not recommended.

4.6.5 *About Grouping*

An important, but easily neglected, aspect of DeMorgan's theorem concerns grouping. Since a bar functions as a grouping symbol, the variables formerly grouped by a broken bar must remain grouped or else proper precedence (order of operation) will be lost. Therefore, after simplifying a large grouping of variables, it is a good practice to place them in parentheses in order to keep the order of operation the same.

Consider the circuit in Figure 4.30.

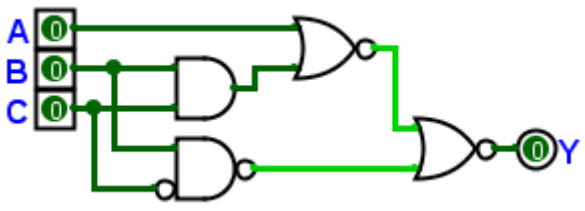

Figure 4.30: DeMorgan's Theorem Example 2

As always, the first step in simplifying this circuit is to generate the Boolean expression for the circuit, which is done by writing the sub-expression at the output of each gate. That results in Expression 4.37, which is then simplified.

$$
\begin{aligned}
&\overline{\overline{A+BC}+\overline{A\overline{B}}} && \text{Original Expression} && (4.37)\\
&(\overline{\overline{A+BC}})(\overline{\overline{A\overline{B}}}) && \text{Breaking the Longest Bar}\\
&(A+BC)(A\overline{\overline{B}}) && \text{Involution}\\
&(AA\overline{B})(BCA\overline{B}) && \text{Distribute } A\overline{B} \text{ to } (A+BC)\\
&(A\overline{B})+(BCA\overline{B}) && \text{Idempotence: } AA=A\\
&(A\overline{B})+(0CA)) && \text{Complement: } B\overline{B}=0\\
&(A\overline{B})+0 && \text{Annihilator: } 0CA=0\\
&A\overline{B} && \text{Identity: } A+0=A
\end{aligned}
$$

The equivalent gate circuit for this much-simplified expression is as follows:

Figure 4.31: DeMorgan's Theorem Example 2 Simplified

4.6.6 *Summary*

Here are the important points to remember about DeMorgan's Theorem:

- It describes the equivalence between gates with inverted inputs and gates with inverted outputs.
- When "breaking" a complementation (or NOT) bar in a Boolean expression, the operation directly underneath the break (AND or OR) reverses and the broken bar pieces remain over the respective terms.
- It is normally easiest to approach a problem by breaking the longest (uppermost) bar before breaking any bars under it.
- Two complementation bars are never broken in one step.
- Complementation bars function as grouping symbols. Therefore, when a bar is broken, the terms underneath it must remain grouped. Parentheses may be placed around these grouped terms as a help to avoid changing precedence.

4.6.7 *Example Problems*

The following examples use DeMorgan's Theorem to simplify a Boolean expression.

	Original Expression	Simplified
1	$(\overline{A+B})(\overline{ABC})(\overline{\overline{A}C})$	$\overline{A}\,\overline{B}\,\overline{C}$
2	$\overline{(AB+\overline{B}C)+(B\overline{C}+\overline{A}B)}$	$\overline{B}\,\overline{C}$
3	$(AB+\overline{B}C)(AC+\overline{A}\,\overline{C})$	$\overline{A}+\overline{C}$

4.7 BOOLEAN FUNCTIONS

Consider Figure 4.32, which is a generic circuit with two inputs and one output.

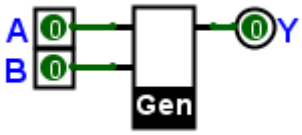

Figure 4.32: Generic Function

Without knowing anything about what is in the unlabeled box at the center of the circuit, there are a number of possible truth tables which could describe the circuit's output. Two possibilities are shown in Truth Table 4.10 and Truth Table 4.11.

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	0
1	1	1

Table 4.10: Truth Table for Generic Circuit One

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.11: Truth Table for Generic Circuit Two

In fact, there are 16 possible truth tables for this circuit. Each of those truth tables reflect a single potential function of the circuit by setting various combinations of input/output. Therefore, any two-input, one-output circuit has 16 possible functions. It is easiest to visualize all 16 combinations of inputs/outputs by using an odd-looking truth table. Consider only one of those 16 functions, the one for the generic circuit described by Truth Table 4.11. That function is also found in the Boolean Functions Table 4.13 and one row from that table is reproduced in Table 4.12.

A	0	0	1	1	
B	0	1	0	1	
F_6	0	1	1	0	Exclusive Or (XOR): $A \oplus B$

Table 4.12: Boolean Function Six

The line shown in Table 4.12 is for *Function 6*, or F_6 (note that the pattern of the outputs is 0110, which is binary 6). Inputs A and B are listed at the top of the table. For example, the highlighted column of the table shows that when A is zero and B is one the output is one. Therefore, on the line that defines F_6, the output is *True* when $[(A = 0$ AND $B = 1)$ OR $(A = 1$ AND $B = 0)]$ This is an XOR function, and the last column of the table verbally describes that function.

Table 4.13 is the complete Boolean Function table.

A	0	0	1	1	
B	0	1	0	1	
F_0	0	0	0	0	Zero or Clear. Always zero (Annihilation)
F_1	0	0	0	1	Logical AND: $A * B$
F_2	0	0	1	0	Inhibition: AB' or $A > B$
F_3	0	0	1	1	Transfer A to Output, Ignore B
F_4	0	1	0	0	Inhibition: $A'B$ or $B > A$
F_5	0	1	0	1	Transfer B to Output, Ignore A
F_6	0	1	1	0	Difference, XOR: $A \oplus B$
F_7	0	1	1	1	Logical OR: $A + B$
F_8	1	0	0	0	Logical NOR: $(A + B)'$
F_9	1	0	0	1	Equivalence, XNOR: $(A = B)'$
F_{10}	1	0	1	0	Not B and ignore A, B Complement
F_{11}	1	0	1	1	Implication, $A + B'$, $B >= A$
F_{12}	1	1	0	0	Not A and ignore B, A Complement
F_{13}	1	1	0	1	Implication, $A' + B$, $A >= B$
F_{14}	1	1	1	0	Logical NAND: $(A * B)'$
F_{15}	1	1	1	1	One or Set. Always one (Identity)

Table 4.13: Boolean Functions

4.8 FUNCTIONAL COMPLETENESS

A set of Boolean operations is said to be *functionally complete* if every possible Boolean function can be derived from that set. The Primary Logic Operations (page **??**) are functionally complete since the Secondary Logic Functions (page 72) can be derived from them. As an example, Equation 4.38 and Figure 4.33 shows how an XOR function can be derived from only AND, OR, and NOT gates.

$$((A * B)' * (A + B)) = A \oplus B \tag{4.38}$$

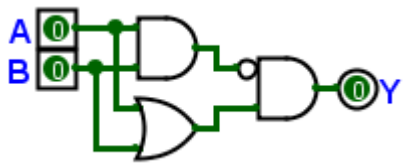

Figure 4.33: XOR Derived From AND/OR/NOT

While the Primary Operations are functionally complete, it is possible to define other functionally complete sets of operations. For example, using DeMorgan's Theorem, the set of {AND, NOT} is also functionally complete since the OR operation can be defined as $(A'B')'$. In fact, both {NAND} and {NOR} operations are functionally complete by themselves. As an example, the NOT operation can be derived using only NAND gates: $(A|A)$. Because all Boolean functions can be derived from either NAND or NOR operations, these are sometimes considered *universal* operations and it is a common challenge for students to create some complex Boolean function using only one of these two types of operations.