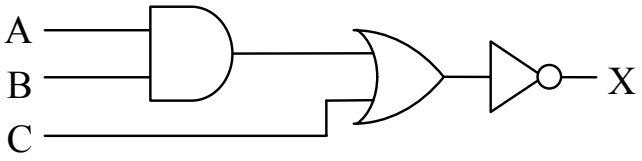
5.1 Need for Boolean Expressions

At this point in our study of digital circuits, we have two methods for representing combinational logic: schematics and truth tables.



A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Figure 5-1 Schematic and Truth Table of Combinational Logic

These two methods are inadequate for a number of reasons:

- Both schematics and truth tables take too much space to describe the operation of complex circuits with numerous inputs.
- The truth table "hides" circuit information.
- The schematic diagram is difficult to use when trying to determine output values for each input combination.

To overcome these problems, a discipline much like algebra is practiced that uses expressions to describe digital circuitry. These expressions, which are called ***boolean expressions***, use the input variable names, A, B, C, etc., and combine them using symbols

representing the AND, OR, and NOT gates. These boolean expressions can be used to describe or evaluate the output of a circuit.

There is an additional benefit. Just like algebra, a set of rules exist that when applied to boolean expressions can dramatically simplify them. A simpler expression that produces the same output can be realized with fewer logic gates. A lower gate count results in cheaper circuitry, smaller circuit boards, and lower power consumption.

If your software uses binary logic, the logic can be represented with boolean expressions. Applying the rules of simplification will make the software run faster or allow it to use less memory.

The next section describes the representation of the three primary logic functions, NOT, AND, and OR, and how to convert combinational logic to a boolean expression.

5.2 Symbols of Boolean Algebra

Analogous behavior can be shown between boolean algebra and mathematical algebra, and as a result, similar symbols and syntax can be used. For example, the following expressions hold true in math.

$$0 \cdot 0 = 0 \qquad 0 \cdot 1 = 0 \qquad 1 \cdot 0 = 0 \qquad 1 \cdot 1 = 1$$

This looks like the AND function allowing an analogy to be drawn between the mathematical multiply and the boolean AND functions. Therefore, in boolean algebra, A AND'ed with B is written $A \cdot B$.

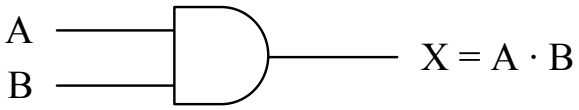


Figure 5-2 Boolean Expression for the AND Function

Mathematical addition has a similar parallel in boolean algebra, although it is not quite as flawless. The following four mathematical expressions hold true for addition.

$$0 + 0 = 0 \qquad 0 + 1 = 1 \qquad 1 + 0 = 1 \qquad 1 + 1 = 2$$

The first three operations match the OR function, and if the last operation is viewed as having a non-zero result instead of the decimal result of two, it too can be viewed as operating similar to the OR

function. Therefore, the boolean OR function is analogous to the mathematical function of addition.

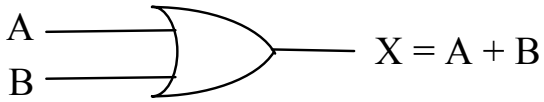


Figure 5-3 Boolean Expression for the OR Function

An analogy cannot be made between the boolean NOT and any mathematical operation. Later in this chapter we will see how the NOT function, unlike AND and OR, requires its own special theorems for algebraic manipulation. The NOT is represented with a bar across the inverted element.

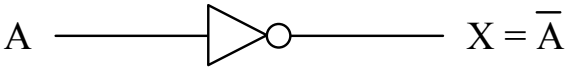


Figure 5-4 Boolean Expression for the NOT Function

The NOT operation may be used to invert the result of a larger expression. For example, the NAND function which places an inverter at the output of an AND gate is written as:

$$X = \overline{A \cdot B}$$

Since the bar goes across A · B, the NOT is performed after the AND.

Let's begin with some simple examples. Can you determine the output of the boolean expression 1 + 0 + 1? Since the plus-sign represents the OR circuit, the expression represents 1 or 0 or 1.

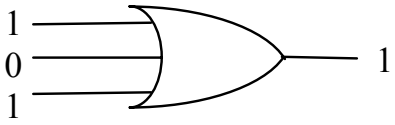


Figure 5-5 Circuit Representation of the Boolean Expression 1+0+1

Since an OR-gate outputs a 1 if any of its inputs equal 1, then 1 + 0 + 1 = 1.

The two-input XOR operation is represented using the symbol ⊕, but it can also be represented using a boolean expression. Basically, the

two-input XOR equals one if A = 0 and B = 1 or if A = 1 and B = 0. This gives us the following expression.

$$X = A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$$

The next section shows how the boolean operators ·, +, ⊕, and the NOT bar may be combined to represent complex combinational logic.

5.3 Boolean Expressions of Combinational Logic

Just as mathematical algebra combines multiplication and addition to create complex expressions, boolean algebra combines AND, OR, and NOT functions to represent complex combinational logic. Our experience with algebra allows us to understand the expression $Y = X \cdot (X + 5) + 3$. The decimal value 5 is added to a copy of X, the result of which is then multiplied by a second copy of X. Lastly, a decimal 3 is added and the final result is assigned to Y.

This example shows us two things. First, each mathematical operation has a priority, e.g., multiplication is performed before addition. This priority is referred to as precedence. Second, variables such X can appear multiple times in an expression, each appearance representing the current value of X.

Boolean algebra allows for the same operation. Take for example the circuit shown in Figure 5-6.

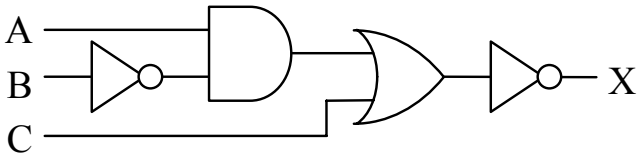
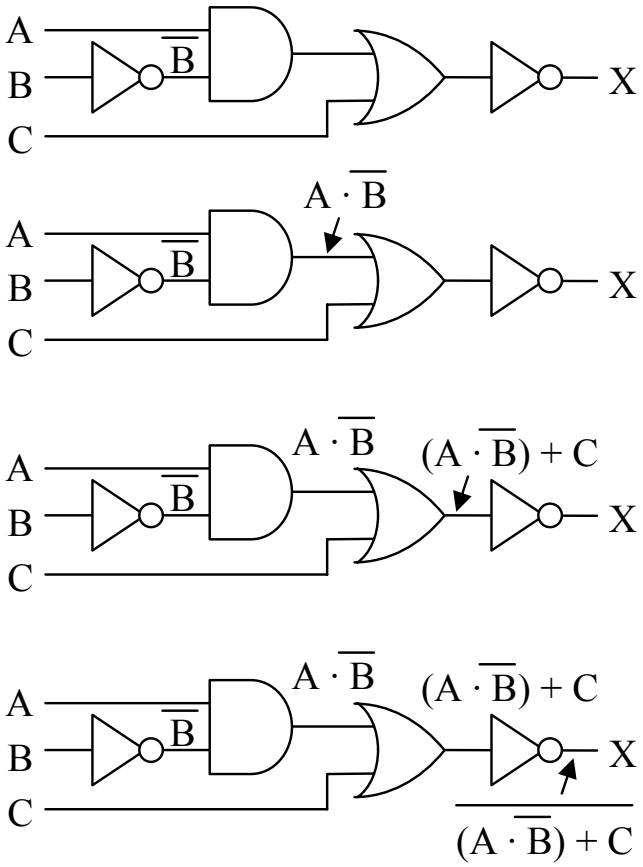


Figure 5-6 Sample of Multi-Level Combinational Logic

In Chapter 4, we determined the truth table for this circuit by taking the input signals A, B, and C from left to right through each gate. As shown in Figure 5-7, we can do the same thing to determine the boolean expression.

Notice the use of parenthesis in step c. Just as in mathematical algebra, parenthesis can be used to force the order in which operations are taken. In the absence of parenthesis, however, the AND, OR, and NOT functions have an order of precedence.



a) B goes through the first inverter which outputs a $\overline{B}$

b) A and $\overline{B}$ go through the AND gate which outputs $A \cdot \overline{B}$.

c) $A \cdot \overline{B}$ and C go through the OR gate which outputs $(A \cdot \overline{B}) + C$.

d) The output of the OR gate goes through a second inverter giving us our result.

Figure 5-7 Creating Boolean Expression from Combinational Logic

To begin with, AND takes precedence over OR unless overridden by parenthesis. NOT is a special case in that it can act like a set of parenthesis. If the bar indicating the NOT function spans a single variable, it takes precedence over AND and OR. If, however, the NOT bar spans an expression, the expression beneath the bar must be evaluated before the NOT is taken. Figure 5-8 presents two examples of handling precedence with the NOT function.

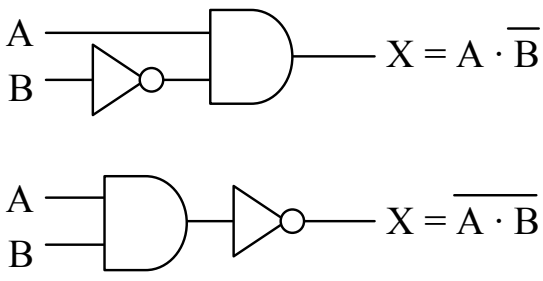


Figure 5-8 Examples of the Precedence of the NOT Function

Understanding this is vital because unlike the mathematical inverse, the two expressions below are *not* equivalent.

$$\overline{A} \cdot \overline{B} \neq \overline{A \cdot B}$$

Let's do an example addressing precedence with a more complex boolean expression. Using parenthesis and the order of precedence, the boolean expression below has a single interpretation.

$$X = A \cdot D + \overline{(A + \overline{B + C})}$$

The following steps show the order to evaluate the above expression.

1. OR B with C because the operation is contained under a single NOT bar and is contained within the lowest set of parenthesis
2. Invert the result of step 1 because NOT takes precedence over OR
3. OR A with the result of step 2 because of the parenthesis
4. Invert result of step 3
5. AND A and D because AND takes precedence over OR
6. OR the results of steps 4 and 5

We can use this order of operations to convert the expression to its schematic representation. By starting with a list of inputs to the circuit, then passing each input through the correct gates, we can develop the circuit. Figure 5-9 does just this for the previous boolean expression. We list the inputs for the expression, A, B, C, and D, on the left side of the figure. These inputs are then passed through the gates using the same order as the steps shown above. The number inside each gate of the figure corresponds to the order of the steps.

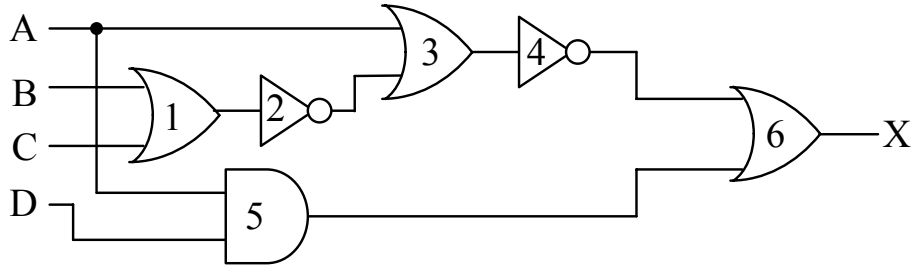


Figure 5-9 Example of a Conversion from a Boolean Expression

The following sections show how boolean expressions can be used to modify combinational logic in order to reduce complexity or otherwise modify its structure.

5.4 Laws of Boolean Algebra

The manipulation of algebraic expressions is based on fundamental laws. Some of these laws extend to the manipulation of boolean expressions. For example, the commutative law of algebra which states that the result of an operation is the same regardless of the order of operands holds true for boolean algebra too. This is shown for the OR function applied to two variables in the truth tables of Figure 5-10.

A	B	A + B
0	0	0+0 = 0
0	1	0+1 = 1
1	0	1+0 = 1
1	1	1+1 = 1

A	B	B + A
0	0	0+0 = 0
0	1	1+0 = 1
1	0	0+1 = 1
1	1	1+1 = 1

Figure 5-10 Commutative Law for Two Variables OR'ed Together

Not only does Figure 5-10 show how the commutative law applies to the OR function, it also shows how truth tables can be used in boolean algebra to prove laws and rules. If a rule states that two boolean expressions are equal, then by developing the truth table for each expression and showing that the output is equal for all combinations of ones and zeros at the input, then the rule is proven true.

Below, the three fundamental laws of boolean algebra are given along with examples.

Commutative Law: The results of the boolean operations AND and OR are the same regardless of the order of their operands.

$$A + B = B + A$$
$$A \cdot B = B \cdot A$$

Associative Law: The results of the boolean operations AND and OR with three or more operands are the same regardless of which pair of elements are operated on first.

$$A + (B + C) = (A + B) + C$$
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

Distributive Law: The AND'ing of an operand with an OR expression is equivalent to OR'ing the results of an AND between the first operand and each operand within the OR expression.

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

The next section uses truth tables and laws to prove twelve rules of boolean algebra.

5.5 Rules of Boolean Algebra

5.5.1 NOT Rule

In algebra, the negative of a negative is a positive and taking the inverse of an inverse returns the original value. Although the NOT gate does not have an equivalent in mathematical algebra, it operates in a similar manner. If the boolean inverse of a boolean inverse is taken, the original value results.

$$\overline{\overline{A}} = A$$

This is proven with a truth table.

A	$\overline{A}$	$\overline{\overline{A}}$
0	1	0
1	0	1

Since the first column and the third column have the same pattern of ones and zeros, they must be equivalent. Figure 5-11 shows this rule in schematic form.

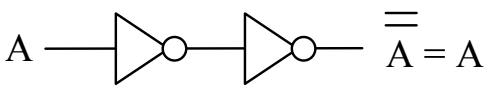


Figure 5-11 Schematic Form of NOT Rule

5.5.2 OR Rules

If an input to a logic gate is a constant 0 or 1 or if the same signal is connected to more than one input of a gate, a simplification of the

expression is almost always possible. This is true for the OR gate as is shown with the following four rules for simplifying the OR function.

First, what happens when one of the inputs to an OR gate is a constant logic 0? It turns out that the logic 0 input drops out leaving the remaining inputs to stand on their own. Notice that the two columns in the truth table below are equivalent thus proving this rule.

Rule: A + 0 = A

A	A + 0
0	0+0 = 0
1	1+0 = 1

What about inputting a logic 1 to an OR gate? In this case, a logic 1 forces the other operands into the OR gate to drop out. Notice that the output column (A + 1) is always equal to 1 regardless of what A equals. Therefore, the output of this gate will always be 1.

Rule: A + 1 = 1

A	A + 1
0	0+1 = 1
1	1+1 = 1

If the same operand is connected to all of the inputs of an OR gate, we find that the OR gate has no effect. Notice that the two columns in the truth table below are equivalent thus proving this rule.

Rule: A + A = A

A	A + A
0	0+0 = 0
1	1+1 = 1

Another case of simplification occurs when an operand is connected to one input of a two-input OR gate and its inverse is connected to the other. In this case, either the operand is equal to a one or its inverse is. There is no other possibility. Therefore, at least one logic 1 is connected to the inputs of the OR gate. This gives us an output of logic 1 regardless of the inputs.

Rule: A + $\overline{\mathbf{A}}$ = 1

A	A + $\overline{\text{A}}$
0	0+1 = 1
1	1+0 = 1

5.5.3 AND Rules

Just as with the OR gate, if either of the inputs to an AND gate is a constant (logic 0 or logic 1) or if the two inputs are the same or inverses

of each other, a simplification can be performed. Let's begin with the case where one of the inputs to the AND gate is a logic 0. Remember that an AND gate must have all ones at its inputs to output a one. In this case, one of the inputs will always be zero forcing this AND to always output zero. The truth table below shows this.

Rule: A · 0 = 0

A	A · 0
0	0 · 0 = 0
1	1 · 0 = 0

If one input of a two-input AND gate is connected to a logic 1, then it only takes the other input going to a one to get all ones on the inputs. If the other input goes to zero, the output becomes zero. This means that the output follows the input that is not connected to the logic 1.

Rule: A · 1 = A

A	A · 1
0	0 · 1 = 0
1	1 · 1 = 1

If the same operand is connected to all of the inputs of an AND gate, we get a simplification similar to that of the OR gate. Notice that the two columns in the truth table below are equivalent proving this rule.

Rule: A · A = A

A	A · A
0	0 · 0 = 0
1	1 · 1 = 1

Last of all, when an operand is connected to one input of a two-input AND gate and its inverse is connected to the other, either the operand is equal to a zero or its inverse is equal to zero. There is no other possibility. Therefore, at least one logic 0 is connected to the inputs of the AND gate giving us an output of logic 0 regardless of the inputs.

Rule: $A \cdot \overline{A} = 0$

A	$A \cdot \overline{A}$
0	0 · 1 = 0
1	1 · 0 = 0

5.5.4 XOR Rules

Now let's see what happens when we apply these same input conditions to a two-input XOR gate. Remember that a two-input XOR

gate outputs a 1 if its inputs are different and a zero if its inputs are the same.

If one of the inputs to a two-input XOR gate is connected to a logic 0, then the gate's output follows the value at the second input. In other words, if the second input is a zero, the inputs are the same forcing the output to be zero and if the second input is a one, the inputs are different and the output equals one.

Rule: $A \oplus 0 = A$

A	$A \oplus 0$
0	$0 \oplus 0 = 0$
1	$1 \oplus 0 = 1$

If one input of a two-input XOR gate is connected to a logic 1, then the XOR gate acts as an inverter as shown in the table below.

Rule: $A \oplus 1 = \overline{A}$

A	$A \oplus 1$
0	$0 \oplus 1 = 1$
1	$1 \oplus 1 = 0$

If the same operand is connected to both inputs of a two-input XOR gate, then the inputs are always the same and the gate outputs a 0.

Rule: $A \oplus A = 0$

A	$A \oplus A$
0	$0 \oplus 0 = 0$
1	$1 \oplus 1 = 0$

Lastly, if the inputs of a two-input XOR gate are inverses of each other, then the inputs are always different and the output is 1.

Rule: $A \oplus \overline{A} = 1$

A	$A \oplus \overline{A}$
0	$0 \oplus 1 = 1$
1	$1 \oplus 0 = 1$

5.5.5 Derivation of Other Rules

If we combine the NOT, OR, and AND rules with the commutative, associative, and distributive laws, we can derive other rules for boolean algebra. This can be shown with the following example.

Example

Prove that $A + A \cdot B = A$

Solution

A + A·B	= A·1 + A·B	Rule: A · 1 = A
	= A·(1 + B)	Distributive Law
	= A·(B + 1)	Commutative Law
	= A·1	Rule: A + 1 = 1
	= A	Rule: A · 1 = A

Remember also that rules of boolean algebra can be proven using a truth table. The example below uses a truth table to derive another rule.

Example

Prove $A + \overline{A} \cdot B = A + B$

Solution

The truth table below goes step-by-step through both sides of the expression to prove that $A + \overline{A} \cdot B = A + B$.

A	B	$\overline{A}$	$\overline{A}$·B	A + $\overline{A}$·B	A + B
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

The mathematical "F-O-I-L" principle, based on the distributive law, works in boolean algebra too. FOIL is a memory aid referring to the multiplication pattern for multiplying quadratic equations. It stands for:

F – AND the first terms from each OR expression
O – AND the outside terms (the first term from the first OR expression and the last term from the last OR expression)
I – AND the inside terms (the last term from the first OR expression and the first term from the last OR expression)
L – AND the last terms from each OR expression

Example

Prove *(A + B)·(A + C) = A + B·C*

Solution

(A + B)·(A + C)	= (A + B)·A + (A + B)·C	Distributive Law
	= A·A + B·A + A·C + B·C	Distributive Law
	= A + B·A + A·C + B·C	Rule: A·A = A
	= A + A·B + A·C + B·C	Commutative Law
	= A + A·C + B·C	Rule: A + A·B = A
	= A + B·C	Rule: A + A·B = A

Now that you have a taste for the manipulation of boolean expressions, the next section will show examples of how complex expressions can be simplified.

5.6 Simplification

Many students of algebra are frustrated by problems requiring simplification. Sometimes it feels as if extrasensory perception is required to see where the best path to simplification lies. Unfortunately, boolean algebra is no exception. There is no substitute for practice. Therefore, this section provides a number of examples of simplification in the hope that seeing them presented in detail will give you the tools you need to simplify the problems on your own.

The rules of the previous section are summarized in Figure 5-12.

1.	$\overline{\overline{A}} = A$	9.	$A \cdot \overline{A} = 0$
2.	$A + 0 = A$	10.	$A \oplus 0 = A$
3.	$A + 1 = 1$	11.	$A \oplus 1 = \overline{A}$
4.	$A + A = A$	12.	$A \oplus A = 0$
5.	$A + \overline{A} = 1$	13.	$A \oplus \overline{A} = 1$
6.	$A \cdot 0 = 0$	14.	$A + A \cdot B = A$
7.	$A \cdot 1 = A$	15.	$A + \overline{A} \cdot B = A + B$
8.	$A \cdot A = A$	16.	$(A + B) \cdot (A + C) = A + B \cdot C$

Figure 5-12 Rules of Boolean Algebra

Example

Simplify $(A \cdot B + C)(A \cdot B + D)$

Solution

From the rules of boolean algebra, we know that (A + B)(A + C) = A + BC. Substitute A·B for A, C for B, and D for C and we get:

$$(A \cdot B + C)(A \cdot B + D) = A \cdot B + C \cdot D$$

Example

Simplify $(A + B) \cdot (\overline{B} + B)$

Solution

$(A + B) \cdot 1$	Anything OR'ed with its inverse is 1
$(A + B)$	Anything AND'ed with 1 is itself

Example

Simplify $\overline{B} \cdot (A + \overline{A} \cdot B)$

Solution

$\overline{B} \cdot A + \overline{B} \cdot \overline{A} \cdot B$	Distributive Law
$\overline{B} \cdot A + \overline{A} \cdot \overline{B} \cdot B$	Associative Law
$\overline{B} \cdot A + \overline{A} \cdot 0$	Anything AND'ed with its inverse is 0
$\overline{B} \cdot A + 0$	Anything AND'ed with 0 is 0
$\overline{B} \cdot A$	Anything OR'ed with 0 is itself
$A \cdot \overline{B}$	Associative Law

Example

Simplify $(\overline{A} + B) \cdot (A + \overline{B})$

Solution

$\overline{A}\cdot A + \overline{A}\cdot\overline{B} + B\cdot A + B\cdot\overline{B}$ Use FOIL to distribute terms

$0 + \overline{A}\cdot\overline{B} + B\cdot A + 0$ Anything AND'ed with its inverse is 0

$\overline{A}\cdot\overline{B} + B\cdot A$ Anything OR'ed with 0 is itself

Example

Simplify $\overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + \overline{A}\cdot B\cdot C$

Solution

$\overline{A}\cdot(\overline{B}\cdot\overline{C} + \overline{B}\cdot C + B\cdot\overline{C} + B\cdot C)$ Distributive Law

$\overline{A}\cdot(\overline{B}\cdot(\overline{C} + C) + B\cdot(\overline{C} + C))$ Distributive Law

$\overline{A}\cdot(\overline{B}\cdot 1 + B\cdot 1)$ Anything OR'ed with its inverse is 1

$\overline{A}\cdot(\overline{B} + B)$ Anything AND'ed with 1 is itself

$\overline{A}\cdot 1$ Anything OR'ed with its inverse is 1

$\overline{A}$ Anything AND'ed with 1 is itself

5.7 DeMorgan's Theorem

Some of you may have noticed that the truth tables for the AND and OR gates are similar. Below is a comparison of the two operations.

AND

A	B	X = A·B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	X = A+B
0	0	0
0	1	1
1	0	1
1	1	1

Okay, so maybe they're not exactly the same, but notice that the output for each gate is the same for three rows and different for the

fourth. For the AND gate, the row that is different occurs when all of the inputs are ones, and for the OR gate, the different row occurs when all of the inputs are zeros. What would happen if we inverted the inputs of the AND truth table?

AND of inverted inputs

A	B	$X = \overline{A} \cdot \overline{B}$
0	0	1
0	1	0
1	0	0
1	1	0

OR

A	B	X = A+B
0	0	0
0	1	1
1	0	1
1	1	1

The two truth tables are still not quite the same, but they are quite close. The two truth tables are now inverses of one another. Let's take the inverse of the output of the OR gate and see what happens.

AND of inverted inputs

A	B	$X = \overline{A} \cdot \overline{B}$
0	0	1
0	1	0
1	0	0
1	1	0

OR with inverted output

A	B	$X = \overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0

So the output of an AND gate with inverted inputs is equal to the inverted output of an OR gate with non-inverted inputs. A similar proof can be used to show that the output of an OR gate with inverted inputs is equal to the inverted output of an AND gate with non-inverted inputs. This resolves our earlier discussion where we showed that the NOT gate cannot be distributed to the inputs of an AND or an OR gate.

This brings us to DeMorgan's Theorem, the last Boolean law presented in this chapter.

$$\overline{A + B} = \overline{A} \cdot \overline{B} \qquad \overline{A} + \overline{B} = \overline{A \cdot B}$$

The purpose of DeMorgan's Theorem is to allow us to distribute an inverter from the output of an AND or OR gate to the gate's inputs. In doing so, an AND gate is switched to an OR gate and an OR gate is

switched to an AND gate. Figure 5-13 shows how pushing the inverter from the output of a gate to its inputs.

a.) Pushing an inverter through an AND gate flips it to an OR gate

b.) Pushing an inverter through an OR gate flips it to an AND gate

Figure 5-13 Application of DeMorgan's Theorem

DeMorgan's Theorem applies to gates with three or more inputs too.

$$A + B + C + D = (A + B)\cdot(C + D)$$

$$= (\overline{A}\cdot\overline{B})\cdot(\overline{C}\cdot\overline{D})$$

$$= \overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\overline{D}$$

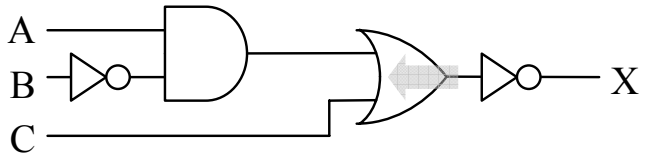
One of the main purposes of DeMorgan's Theorem is to distribute any inverters in a digital circuit back to the inputs. This typically improves the circuit's performance by removing layers of logic and can be done either with the boolean expression or the schematic. Either way, the inverters are pushed from the output side to the input side one gate at a time. The sequence of steps in Figure 5-14 shows this process using a schematic.

It is a little more difficult to apply DeMorgan's Theorem to a boolean expression. To guarantee that the inverters are being distributed properly, it is a good idea to apply DeMorgan's Theorem in the reverse order of precedence for the expression.

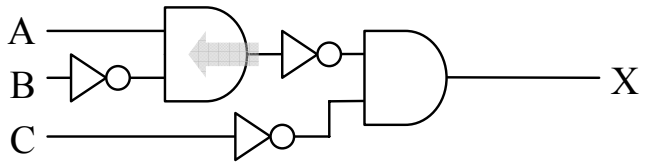
$\overline{A \cdot B + C}$	Step 1: The AND takes precedence over the OR, so distribute inverter across the OR gate first.

$\overline{A \cdot B \cdot C}$ Step 2: Now distribute the inverter across the A·B term.

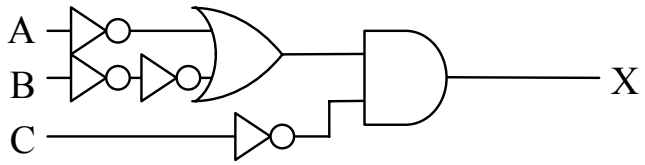
$\overline{(\overline{A} + \overline{B})} \cdot \overline{C}$ Step 3: In this final case, the use of parenthesis is vital.



a.) Push inverter through the OR gate distributing it to the inputs



b.) Push inverter through the AND gate distributing it to the inputs



c.) Two inverters at B input cancel each other

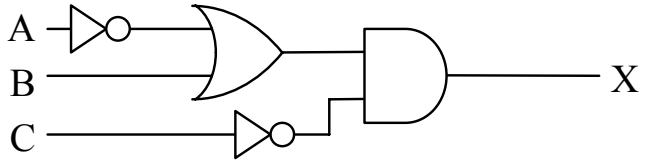


Figure 5-14 Schematic Application of DeMorgan's Theorem

5.8 What's Next?

Using the methods presented in this chapter, a boolean expression can be manipulated into whatever form best suits the needs of the computer system. As far as the binary operation is concerned, two circuits are the same if their truth tables are equivalent. The circuits, however, may not be the same when measuring performance or when counting the number of gates it took to implement the circuit. The

optimum circuit for a specific application can be designed using the tools presented in this chapter.

In Chapter 6, we will show how the rules presented in this chapter are used to take any boolean expression and put it into one of two standard formats. The standard formats allow for quicker operation and support the use of programmable hardware components. Chapter 6 also presents some methods to convert truth tables into circuitry. It will be our first foray into designing circuitry based on a system specification.

Problems

1. List three drawbacks of using truth tables or schematics for describing a digital circuit.
2. List three benefits of a digital circuit that uses fewer gates.
3. True or False: Boolean expressions can be used to optimize the logic functions in software.
4. Convert the following boolean expressions to their schematic equivalents. Do not modify the original expression

   a.) $\overline{A \cdot B} + C$

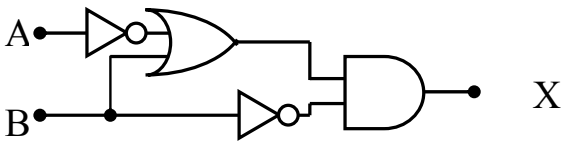
   b.) $A \cdot \overline{B} \cdot C + \overline{A} \cdot B + \overline{A \cdot C}$

   c.) $\overline{(A + B \cdot C)} + A \cdot \overline{D}$

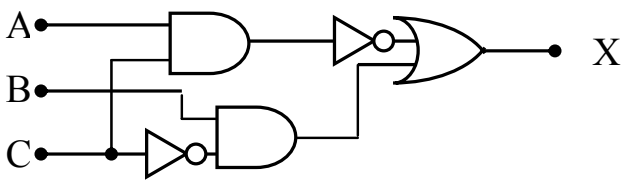
   d.) $A \cdot \overline{B} + \overline{A} \cdot B$

5. Convert each of the digital circuits shown below to their corresponding boolean expressions without simplification.

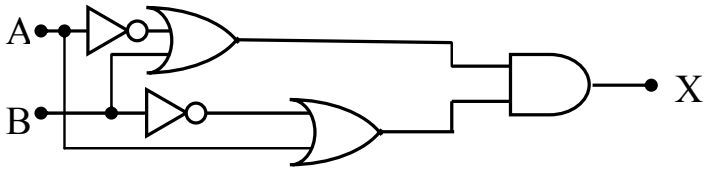
   a.)



b.)



c.)



6. Apply DeMorgan's Theorem to each of the following expressions so that the NOT bars do not span more than a single variable.

   a.) $\overline{A \cdot C + B}$

   b.) $\overline{D(C + B)(A + B)}$

   c.) $\overline{A + \overline{B} + C + \overline{(AB)}}$

7. Simplify each of the following expressions.

   a.) $B \cdot A + B \cdot \overline{A}$

   b.) $(A + B)(B + \overline{A})$

   c.) $A + \overline{B} + C + \overline{(AB)}$

   d.) $\overline{B}(A + A \cdot B)$

   e.) $(\overline{A} + B)(A + \overline{B})$

   f.) $\overline{B} + \overline{(AB)} + \overline{C}$