Chapter 9

Boolean Algebra

One of the most significant mathematical tools available to electronics designers was actually invented for quite a different purpose. Around the 1850s, a British mathematician, George Boole (1815–1864), developed a new form of mathematics that is now known as *Boolean Algebra*. Boole's intention was to use mathematical techniques to represent and rigorously test logical and philosophical arguments. His work was based on the following: a statement is a sentence that asserts or denies an attribute about an object or group of objects:

> **Statement:** *Your face resembles a cabbage.*
>
> Depending on how carefully you choose your friends, they may either agree or disagree with the sentiment expressed; therefore, this statement cannot be proved to be either true or false.

By comparison, a proposition is a statement that is either true or false with no ambiguity:

> **Proposition:** *I just tipped a bucket of burning oil into your lap.*
>
> This proposition may be true or it may be false, but it is definitely one or the other and there is no ambiguity about it.

Propositions can be combined together in several ways; propositions combined with an AND operator are known as a conjunction:

Combining a Single Variable with Logic 0 or Logic 1

A set of simple but highly useful rules can be derived from the combination of a single variable with a logic 0 or logic 1:[1,2]

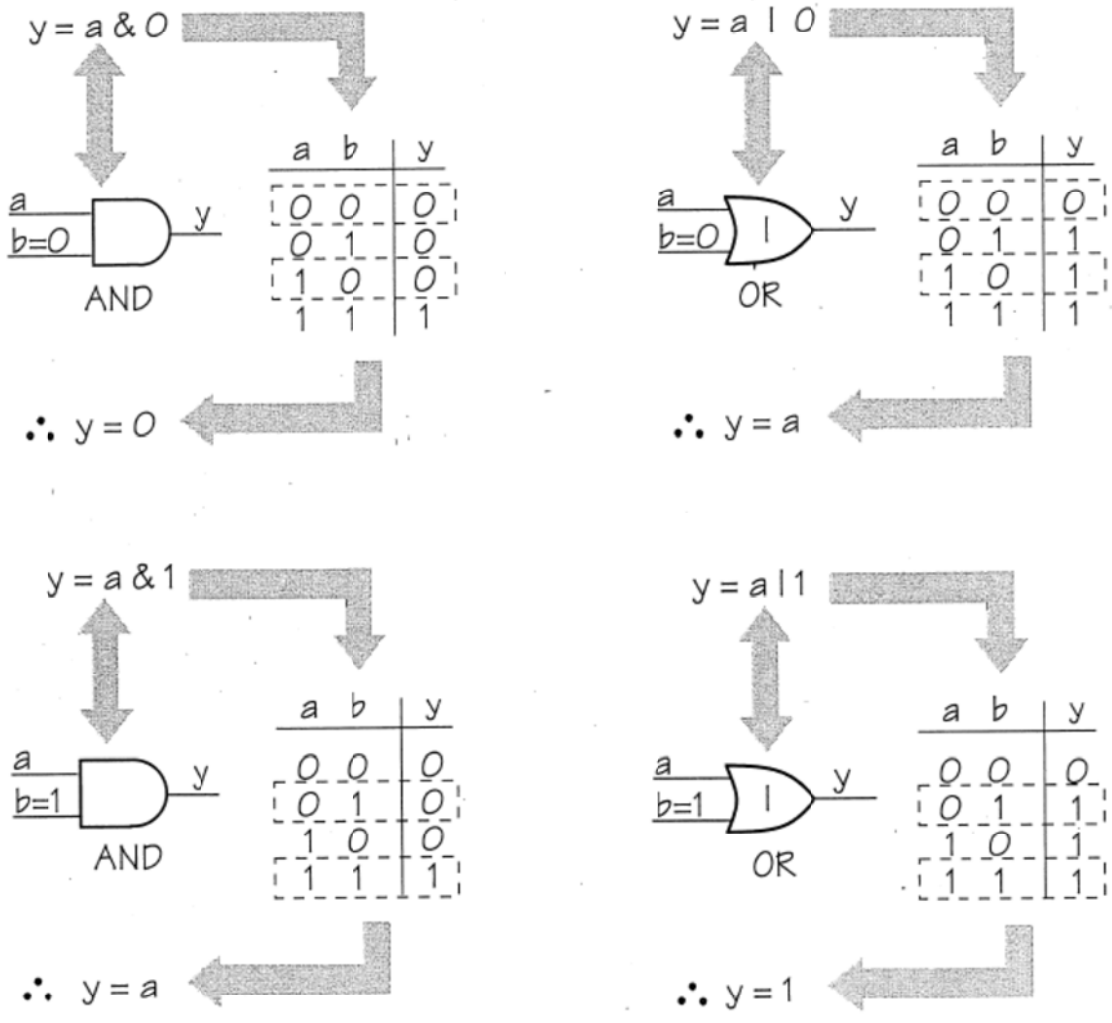


Figure 9.2: Combining a single variable with logic 0 or logic 1

[1] Note that, throughout these discussions, the results from the NAND and NOR functions would be the inverse of those from the AND and OR functions, respectively.

[2] Note that the symbol ∴ shown in the equations in Figure 9.2 means *"therefore."*

The Idempotent Rules

The rules derived from the combination of a single variable with itself are known as the idempotent rules:

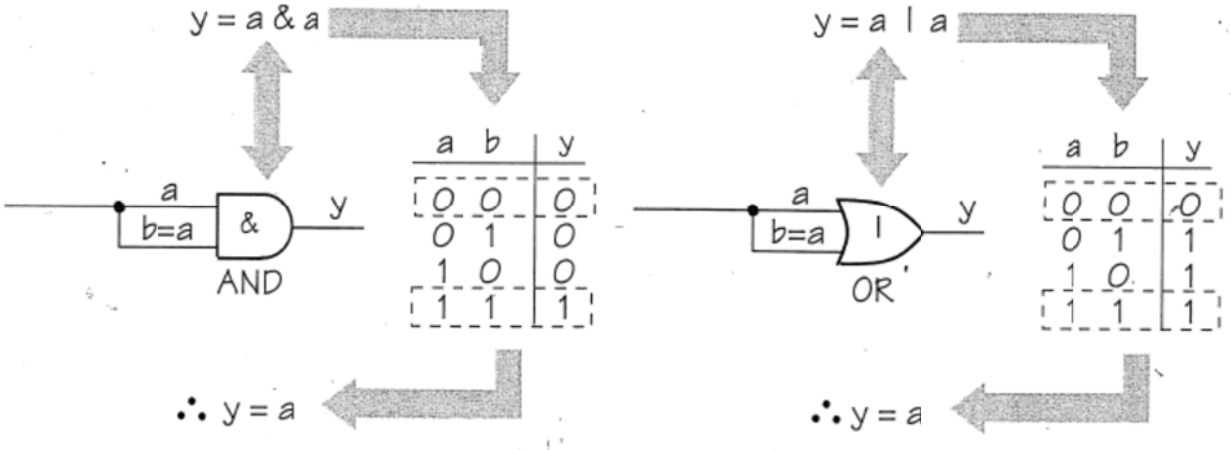


Figure 9.3: The idempotent rules

The Complementary Rules

The rules derived from the combination of a single variable with the inverse of itself are known as the complementary rules:

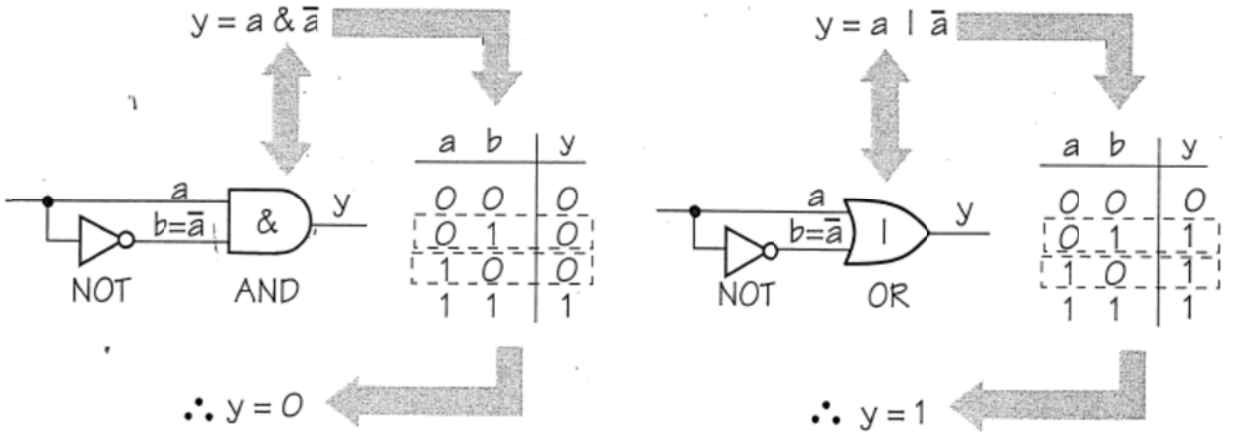


Figure 9.4: The complementary rules

The Involution Rule

The involution rule states that an even number of inversions cancel each other out; for example, two NOT functions connected in series generate an identical result to that of a BUF function:

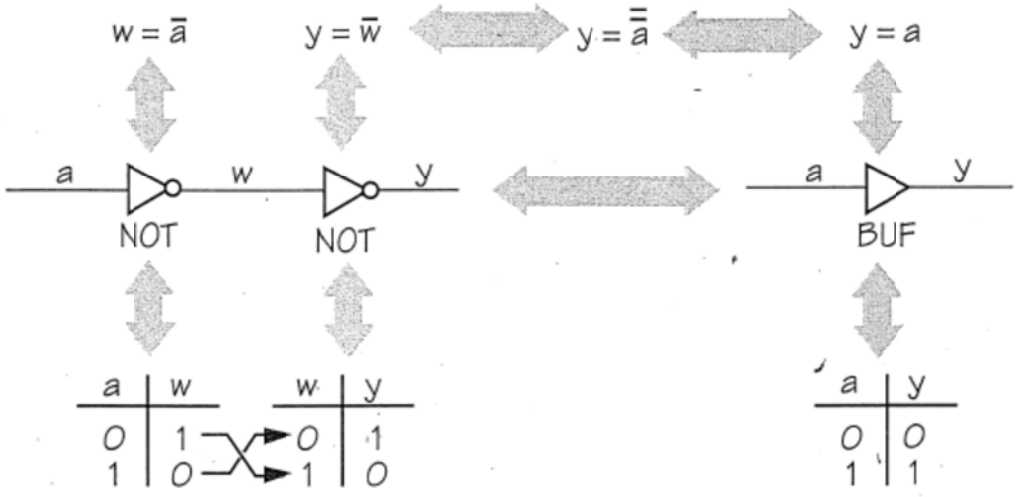


Figure 9.5: The involution rule

The Commutative Rules

The commutative rules state that the order in which variables are specified will not affect the result of an AND or OR operation:

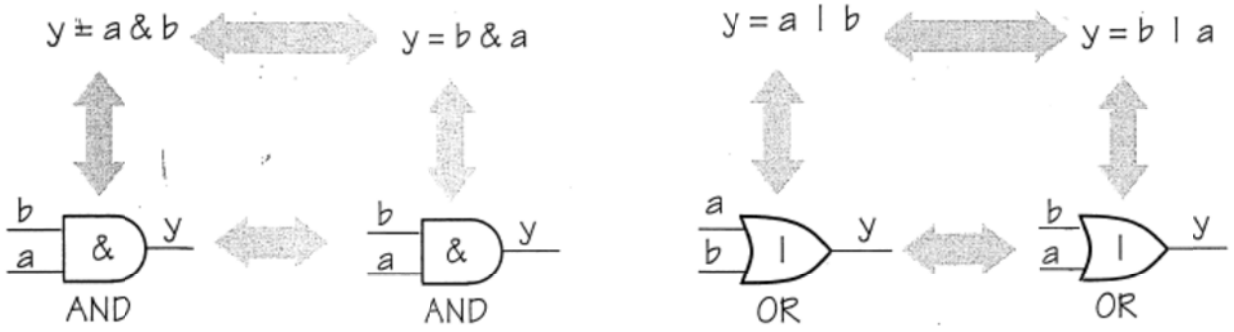


Figure 9.6: The commutative rules

The Associative Rules

The associative rules state that the order in which pairs of variables are associated together will not affect the result of multiple AND or OR operations:

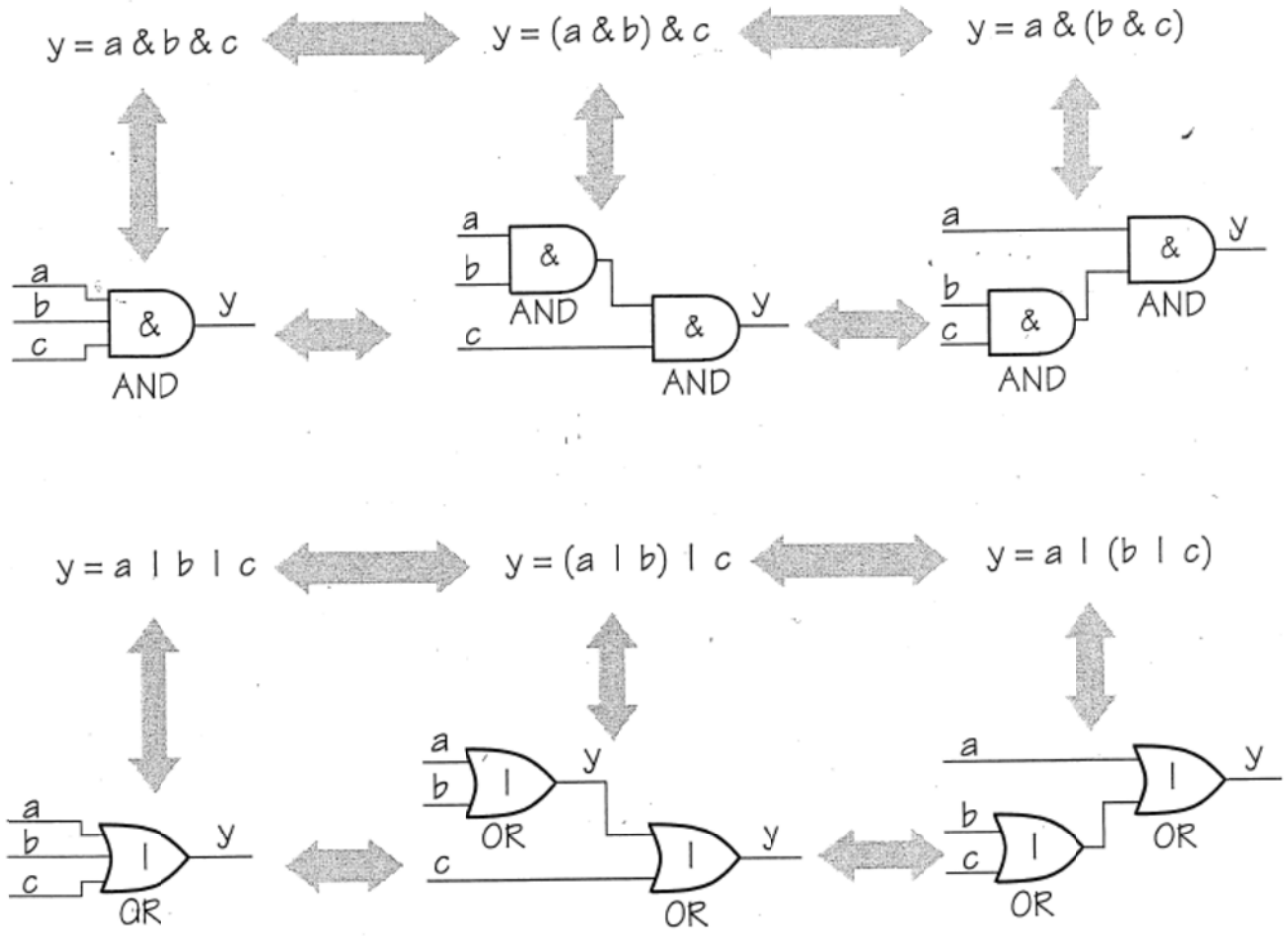


Figure 9.7: The associative rules

In addition to recognizing the application to Boolean algebra to electronic design, Shannon is also credited with the invention of the rocket-powered frisbee, and is famous for riding down the corridors at Bell Laboratories on a unicycle while simultaneously juggling four balls.

Precedence of Operators

In standard arithmetic, the multiplication operator is said to have a higher precedence than the addition operator. This means that, if an equation contains both multiplication and addition operators without parenthesis, then the multiplication is performed before the addition; for example:[3]

$$6 + 2 \times 4 \equiv 6 + (2 \times 4)$$

Similarly, in Boolean Algebra, the & operator has a higher precedence than the | operator:

$$a \mid b \,\&\, c \equiv a \mid (b \,\&\, c)$$

Due to the similarities between these arithmetic and logical operators, the & operator is known as a logical multiplication or *product*, and the | operator is known as a logical addition or *sum*. To avoid any confusion as to the order in which logical operations will be performed, this book will always make use of parentheses.

The first true electronic computer, ENIAC (Electronic Numerical Integrator and Calculator), was constructed at the University of Pennsylvania between 1943 and 1946. In many ways ENIAC was a monster; it occupied 30 feet by 50 feet of floor space, weighed approximately 30 tons, and used more than 18,000 vacuum tubes which required 150 kilowatts of power—enough to light a small town. One of the big problems with computers built from vacuum tubes was reliability; 90% of ENIAC's down-time was attributed to locating and replacing burnt-out tubes. Records from 1952 show that approximately 19,000 vacuum tubes had to be replaced in that year alone; that averages out to about 50 tubes a day!

[3] Note that the symbol ≡ shown in the equations indicates *"is equivalent to"* or *"is the same as."*

The First Distributive Rule

In standard arithmetic, the multiplication operator will distribute over the addition operator because it has a higher precedence; for example:

$$6 \times (5 + 2) \equiv (6 \times 5) + (6 \times 2)$$

Similarly, in Boolean Algebra, the & operator will distribute over an | operator because it has a higher precedence; this is known as the first distributive rule:

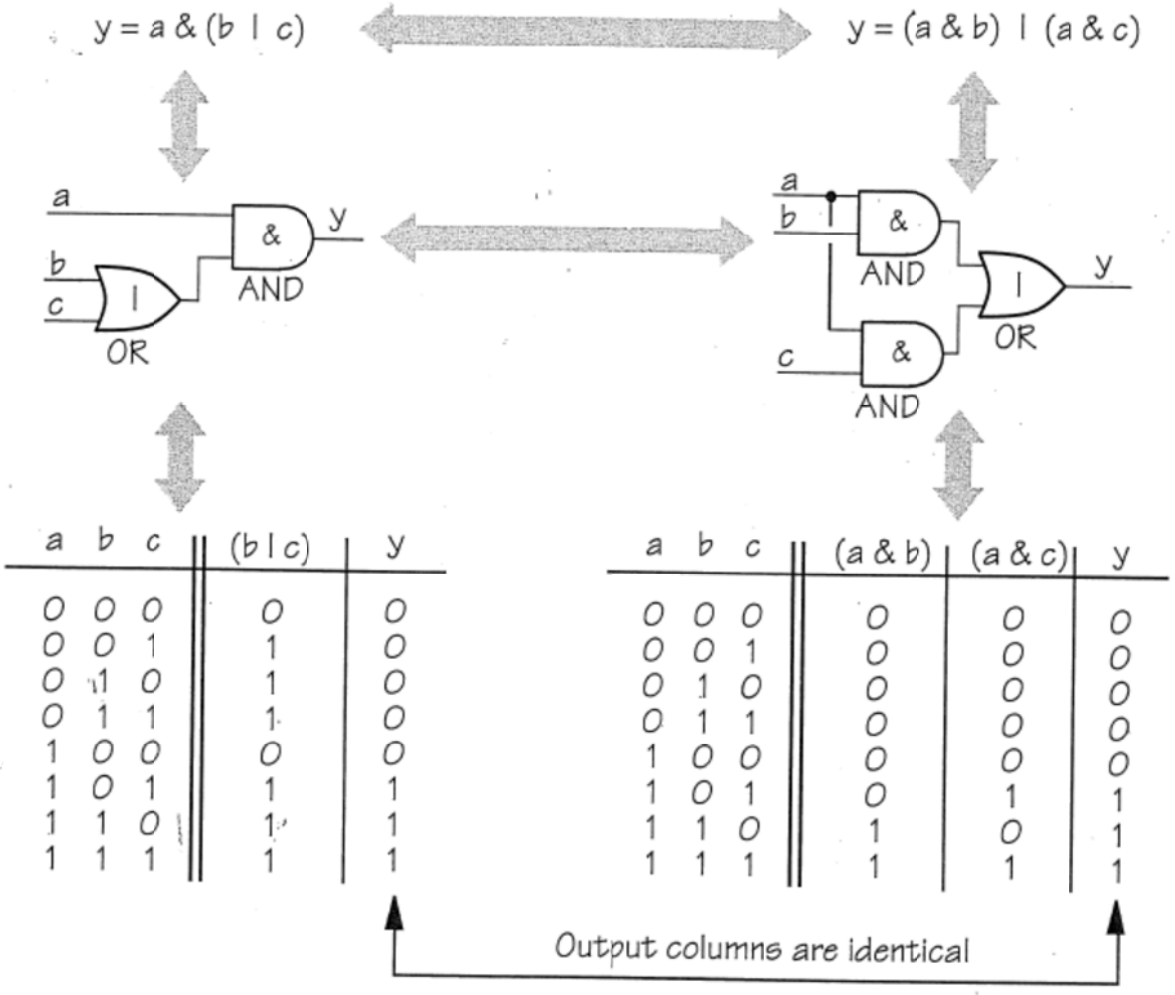


Figure 9.8: The first distributive rule

The Second Distributive Rule

In standard arithmetic, the addition operator will not distribute over the multiplication operator because it has a lower precedence:[4]

$$6 + (5 \times 2) \neq (6 + 5) \times (6 + 2)$$

However, Boolean Algebra is special in this case. Even though the | operator has lower precedence than the & operator, it will still distribute over the & operator; this is known as the second distributive rule:

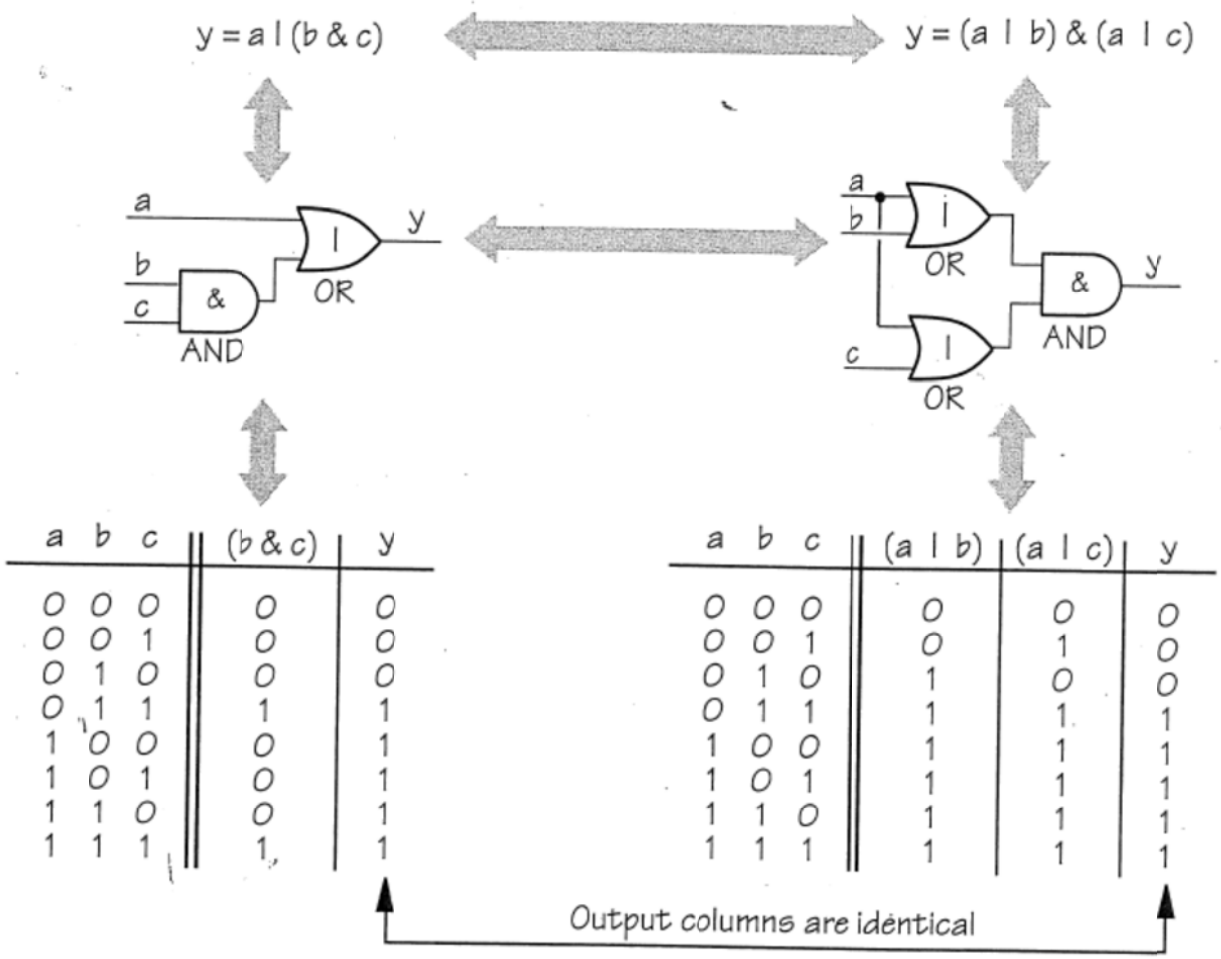


Figure 9.9: The second distributive rule

[4] Note that the symbol ≠ shown in the equation indicates *"is not equal to."*

The Simplification Rules

There are a number of simplification rules which can be used to reduce the complexity of Boolean expressions. As the end result is to reduce the number of logic gates required to implement the expression, the process of simplification is also known as *minimization*:

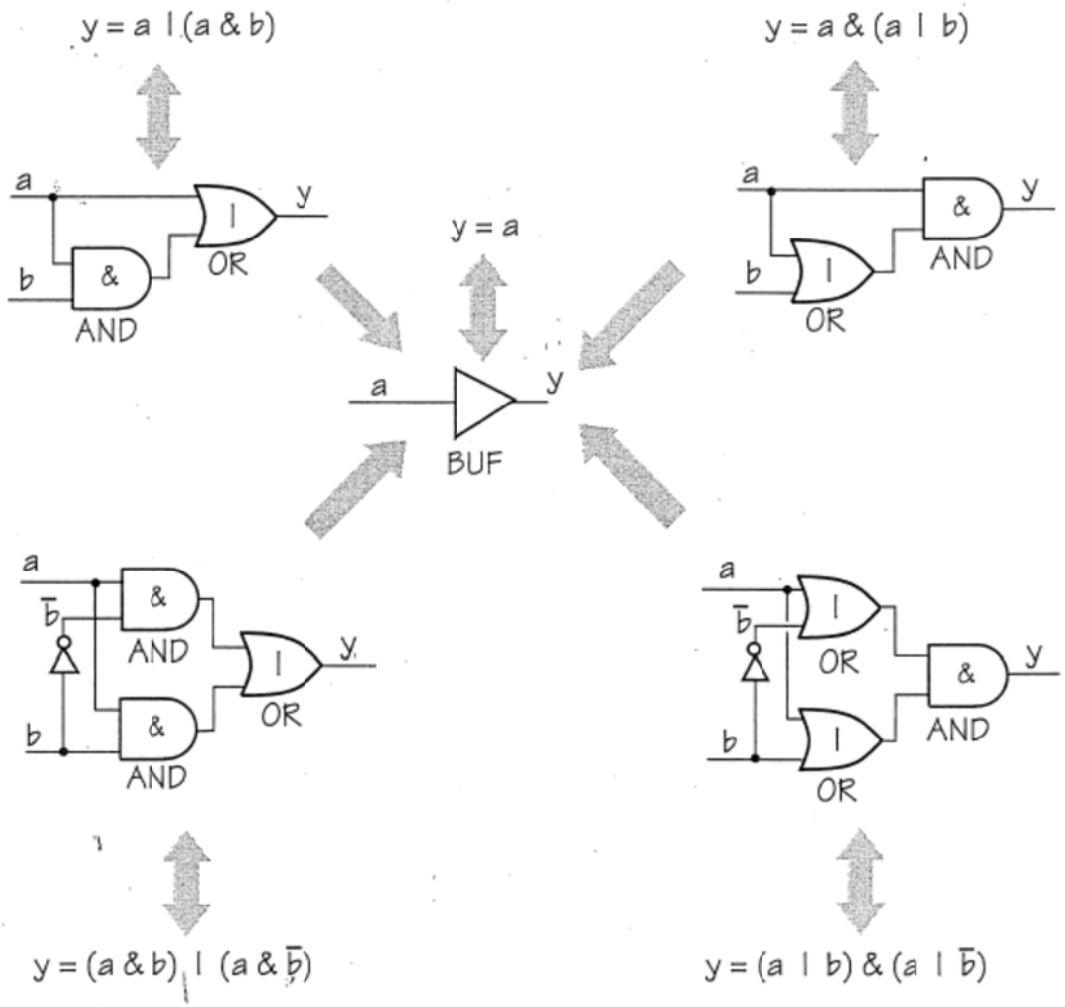


Figure 9.10: The simplification rules

– continued on next page

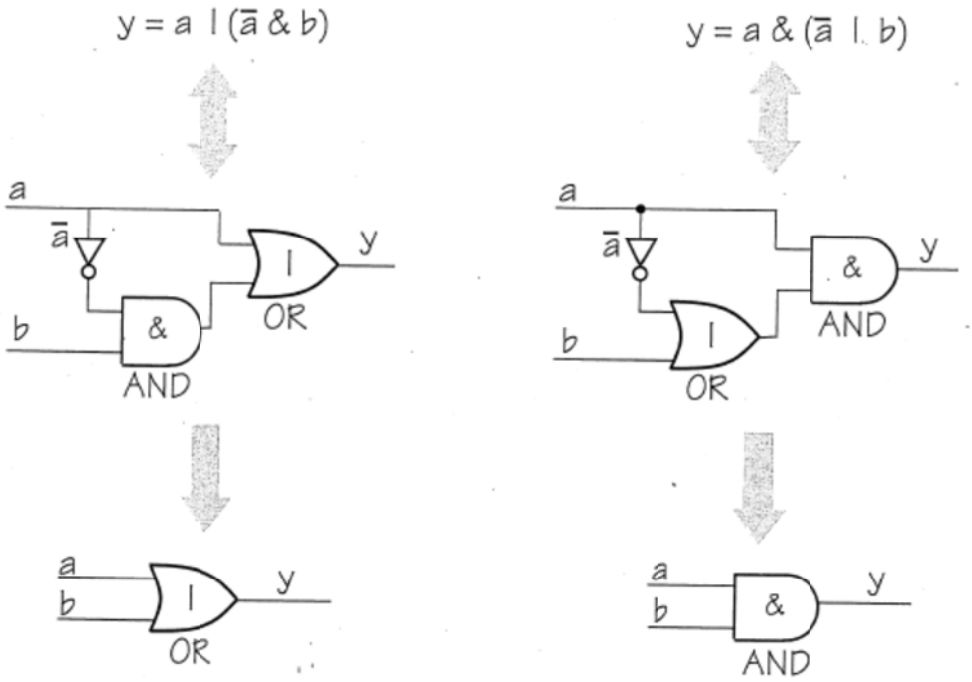


Figure 9.10 *(continued):* The simplification rules

Few now know that, before elbowing his way up the ladder of corporate success, the author of this epic volume once aspired to be a world-famous explorer. Picture him, if you will, as he strode across the veldt (hair by Vidal Sasson™, eyelashes by Max Factor™, and loin cloths by Fruit of the Loom™). The name Tracker Max, as he then styled himself, was oft bandied by the denizens of the dim and dingy taverns in the deepest, direst domains of the dark continent.

Today his Arnold Schwarzenegger look-alike body is compressed into an executive-style business suit (in a fashion reminiscent of years gone by) and he passes his twilight years quaffing the odd fermented coconut and penning trail-blazing books such as this one. Now that time has healed the deeper wounds, some in unusual and interesting places that he is only too happy to display (for a nominal fee), he feels more able to tell the tale of those far off times . . . unfortunately we don't have enough space here!

DeMorgan Transformations

A contemporary of Boole's, Augustus DeMorgan (1806–1871), also made significant contributions to the field of symbolic logic, most notably a set of rules which facilitate the conversion of Boolean expressions into alternate and often more convenient forms. A DeMorgan Transformation comprises four steps:

1. Exchange the & operators for | operators and vice versa.
2. Invert all the variables; also exchange 0s for 1s and vice versa.
3. Invert the entire function.
4. Reduce all multiple inversions.

Consider the DeMorgan Transformation of a 2-input AND function (Figure 9.11). Note that the NOT gate on the output of the new function can be combined with the OR to form a NOR.

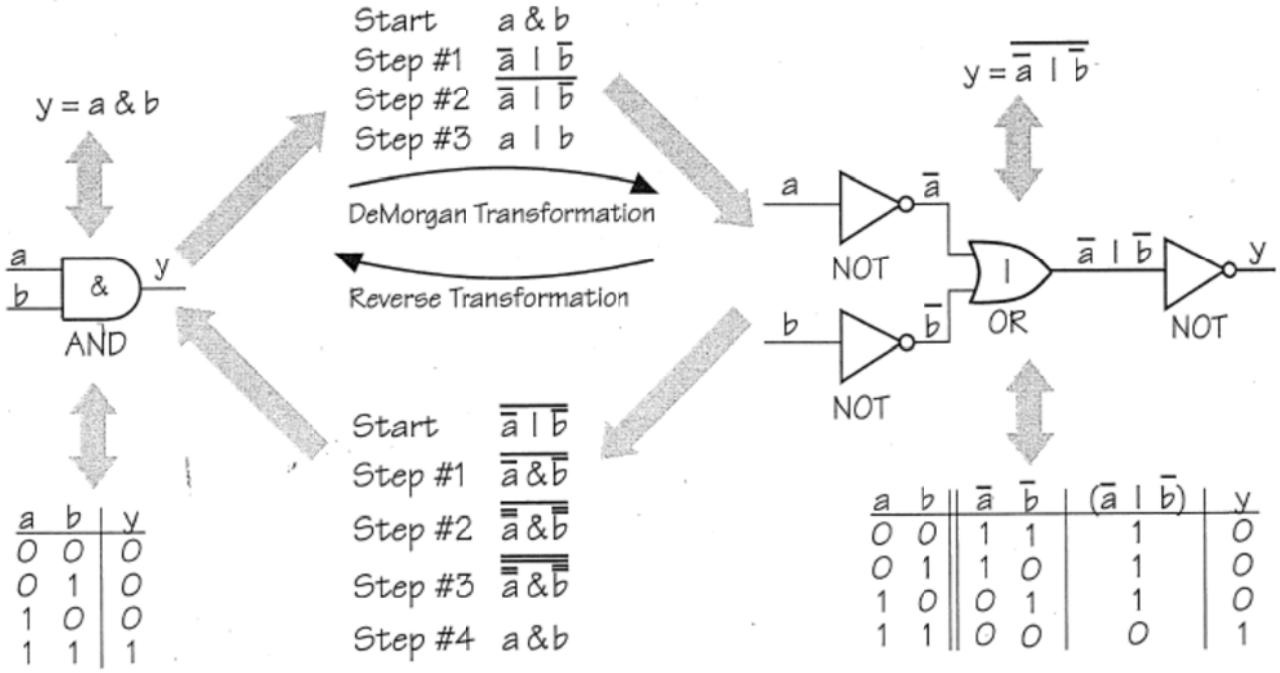


Figure 9.11: DeMorgan Transformation of an AND

Similar transformations can be performed on the other primitive functions:

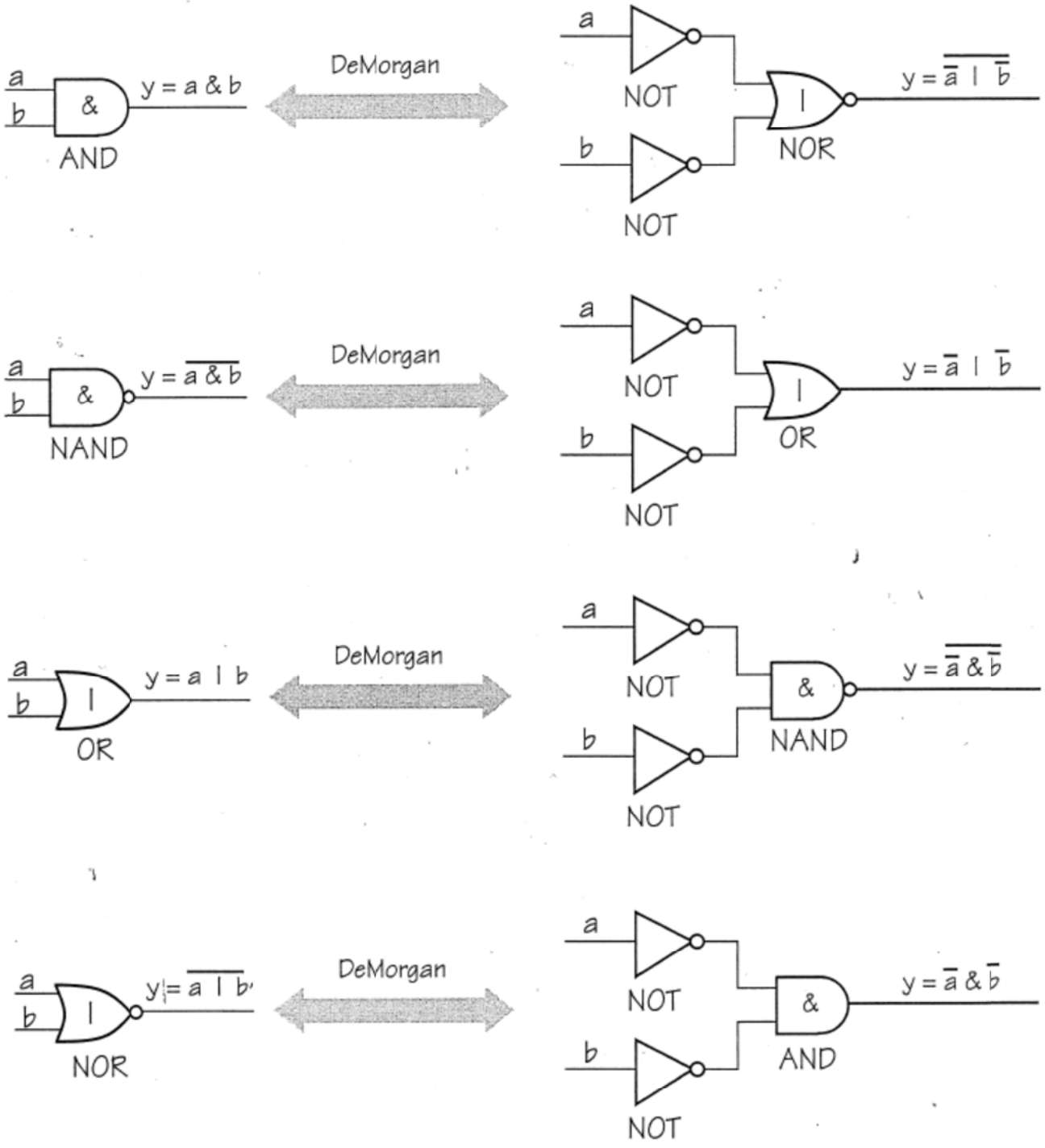


Figure 9.12: DeMorgan Transformations of AND, NAND, OR and NOR

Minterms and Maxterms

For each combination of inputs to a logical function, there is an associated *minterm* and an associated *maxterm*. Consider a truth table with three inputs a, b, and c (Figure 9.13.)

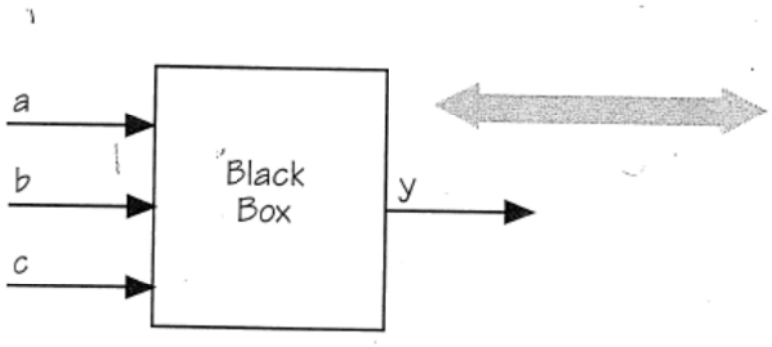
The minterm associated with each input combination is the &, or product, of the input variables; the maxterm is the |, or sum, of the inverted input variables. Minterms and maxterms are useful for deriving Boolean equations from truth tables as discussed below.

a	b	c	minterms	maxterms
0	0	0	$(\bar{a} \,\&\, \bar{b} \,\&\, \bar{c})$	$(a \mid b \mid c)$
0	0	1	$(\bar{a} \,\&\, \bar{b} \,\&\, c)$	$(a \mid b \mid \bar{c})$
0	1	0	$(\bar{a} \,\&\, b \,\&\, \bar{c})$	$(a \mid \bar{b} \mid c)$
0	1	1	$(\bar{a} \,\&\, b \,\&\, c)$	$(a \mid \bar{b} \mid \bar{c})$
1	0	0	$(a \,\&\, \bar{b} \,\&\, \bar{c})$	$(\bar{a} \mid b \mid c)$
1	0	1	$(a \,\&\, \bar{b} \,\&\, c)$	$(\bar{a} \mid b \mid \bar{c})$
1	1	0	$(a \,\&\, b \,\&\, \bar{c})$	$(\bar{a} \mid \bar{b} \mid c)$
1	1	1	$(a \,\&\, b \,\&\, c)$	$(\bar{a} \mid \bar{b} \mid \bar{c})$

Figure 9.13: Minterms and maxterms

Sum-of-Products and Product-of-Sums

A designer will often specify portions of a design using truth tables, and determine how to implement these functions as logic gates later. The designer may start by representing a function as a *black box* with an associated truth table. Note that the values assigned to output y in the truth table shown below were selected randomly, and have no significance beyond the purposes of this example.



a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 9.14: Black box with associated truth table

There are two commonly used techniques for deriving Boolean equations from a truth table. In the first technique, the minterms corresponding to each line in the truth table for which the output is a logic 1 are extracted and combined using | operators; this method results in an equation said to be in *sum-of-products* form. In the second technique, the maxterms corresponding to each line in the truth table for which the output is a logic 0 are combined using & operators; this method results in an equation said to be in *product-of-sums* form:

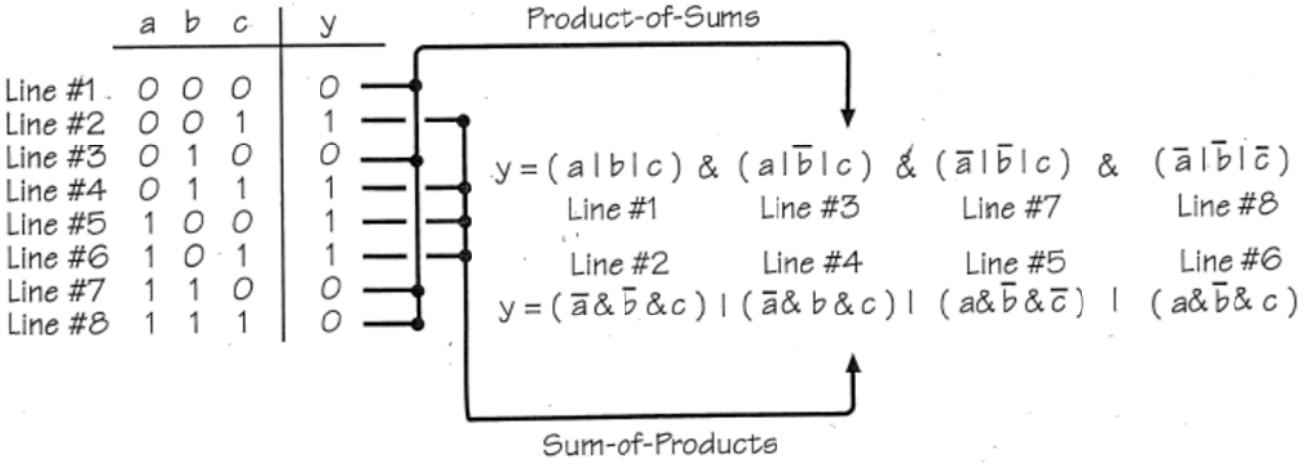


Figure 9.15: Sum-of-products versus product-of-sums forms

For a function whose output is logic 1 fewer times than it is logic 0, it is generally easier to extract a sum-of-products equation. Similarly, if the output is logic 0 fewer times than it is logic 1, it is generally easier to extract a product-of-sums equation. The sum-of-products and product-of-sums forms complement each other and return identical results. An equation in either form can be transformed into its alternative form by means of the appropriate DeMorgan Transformation.

Once an equation has been obtained in the required form, the designer would typically make use of the appropriate simplification rules to minimize the number of logic gates required to implement the function. However, neglecting any potential minimization, the equations above could be translated directly into their logic gate equivalents (Figure 9.16).

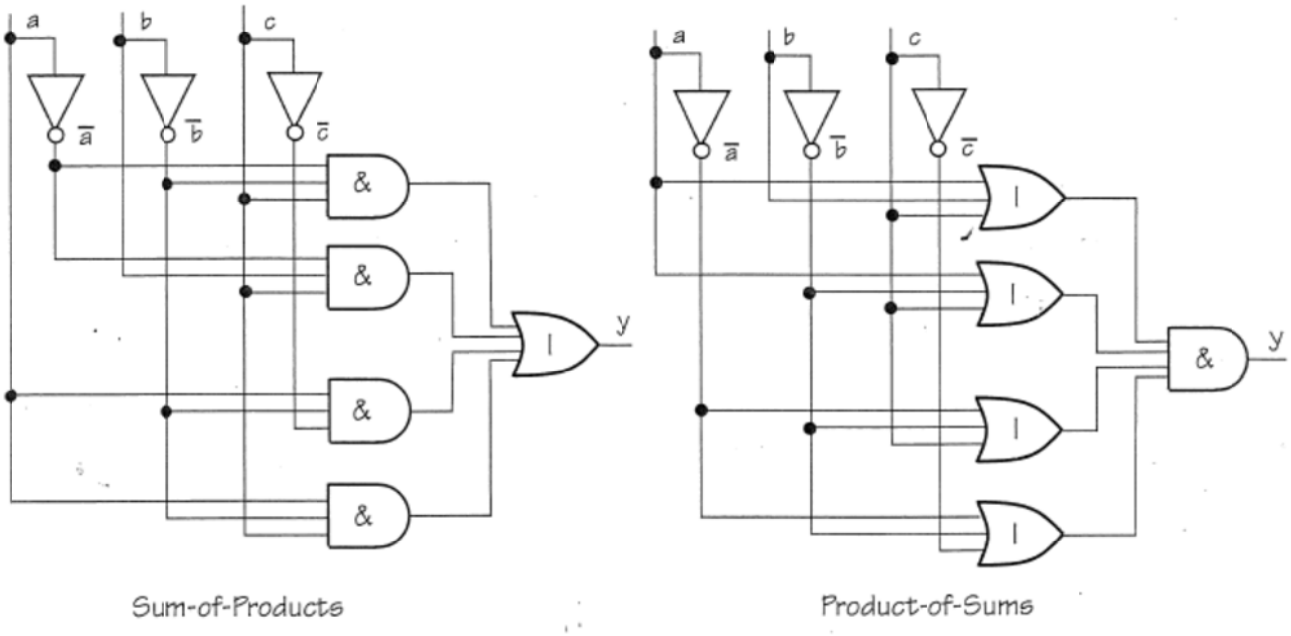


Figure 9.16: Sum-of-products versus product-of-sums implementations

Canonical Forms

In a mathematical context, the term *canonical form* is taken to mean a generic or basic representation. Canonical forms provide the means to compare two expressions without falling into the trap of trying to compare "apples" with "oranges." The sum-of-products and product-of-sums representations are both canonical forms. Thus, to compare two Boolean equations, both must first be coerced into the same canonical form; either sum-of-products or product-of-sums.