12 | Boolean Functions + Logic Gates

We'll be looking at **boolean algebra**, or expressions involving true/false values. We'll also be looking at **digital logic**, or hardware components used to build digital devices, where their fundamental component is logic gates. We will continue to build upon them to create combinational logic circuits and components. Eventually, we'll get to the Arithmetic and Logic Unit (ALU), which does logical operations and comparisons.

A **boolean variable** can have a value which is 0 (false) and 1 (true).

Boolean operations include:

Boolean operations	**Meaning**
x + y	x OR y
x • y (dot may be omitted)	x AND y
x'	NOT x
x ⊕ y	x XOR y

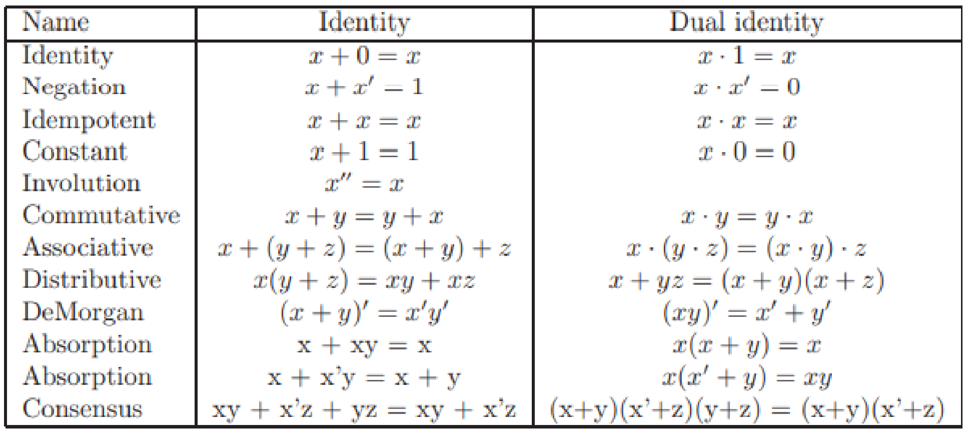
Boolean expressions involve variables and operations. Parentheses can be used to order precedence. Prime takes precedence over an AND and OR. Generally, inclusive and exclusive or have the same precedence. **Examples** include:

x + y'z	=	complement y, then and that result with z (dot omitted here), form inclusive or with x
(x + y')z	=	use inclusive or on both x and the complement of y, then and that result with z
x ⊕ y ⊕ z + 1	=	form the exclusive or of x and y, then take that result and exclusive or that with z, then inclusive or that with 1

Here are some **identities** from chapter 3 (the last column are deMorgan's laws):

x + 0 = x	x + 1 = 1	x + x' = 1	x ⊕ 0 = x	(x • y)' = x' + y'
x • 0 = 0	x • 1 = x	x • x' = 0	x ⊕ 1 = x'	(x + y)' = x' • y'

More and their names are shown below:



Name	Identity	Dual identity
Identity	$x + 0 = x$	$x \cdot 1 = x$
Negation	$x + x' = 1$	$x \cdot x' = 0$
Idempotent	$x + x = x$	$x \cdot x = x$
Constant	$x + 1 = 1$	$x \cdot 0 = 0$
Involution	$x'' = x$	
Commutative	$x + y = y + x$	$x \cdot y = y \cdot x$
Associative	$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Distributive	$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
DeMorgan	$(x + y)' = x'y'$	$(xy)' = x' + y'$
Absorption	x + xy = x	$x(x + y) = x$
Absorption	x + x'y = x + y	$x(x' + y) = xy$
Consensus	xy + x'z + yz = xy + x'z	(x+y)(x'+z)(y+z) = (x+y)(x'+z)

Figure 6.3: Boolean algebraic identites

Simplification of Boolean expressions, using Karnaugh Maps. In hardware, you'll be doing this to save money, since implementing a complicated boolean expression can involve lots of logic gates and hardware components, so if we simplify them, we will be able to use less components. This is also helpful in software; often, specifications for a large system are expressed with logical expressions, and it can get quite complicated. Often, these specifications can be simplified.

The three-variable K-map (essentially a truth table), where every possible combination of variables is represented here (the order is this way so that the first two are only changing 1 bit):

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

Example: Simplify x'y'z' + x'yz + x'yz' + xy'z'. When will this be 1? (Usage explained here.) If any of the 4 expressions, separated by ORs, is 1, then everything will be 1.

Thus, we will try to make the combinations 1 (ex. If x, y, and z were all 0's, the leftmost term would be 1, since complementing them all = 1 and AND-ing them all would give you 1, hence the 1 in the table). All the other values inside will be 0.

Well,   x'y'z' + x'yz + x'yz' + xy'z'...
        0 0 0   0 1 1   0 1 0   1 0 0

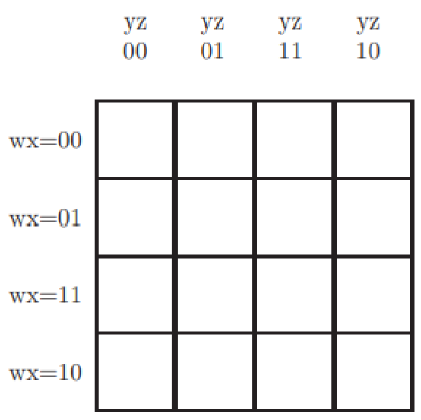
	yz 00	**yz 01**	**yz 11**	**yz 10**
x = 0	1	0	1	1
x = 1	1	0	0	0

What we'll do next is look for neighboring 1's. Neighbors are two 1's next to each other or directly above/below each other. In this example, we have 2 pairs. Notice we have a 1 as our result, and it doesn't matter what *x* is (row of 1's) or *z* is (column of 1's) for y.

Simplified expression: Leftmost pair says that y and z is 0. Hence, to get 1, y'z'.
Rightmost pair says that y is 1 and x is 0. Hence, to get 1, x'y.
Put them together to get **y'z' + x'y**.

How about 4-variable K-maps? With these, you can now have 4x1, 4x2, and 2x4 blocks of 1's. You can also group the first and last column 1's as long as they're in the same row. (See book for better explanation of simplification with an example.)

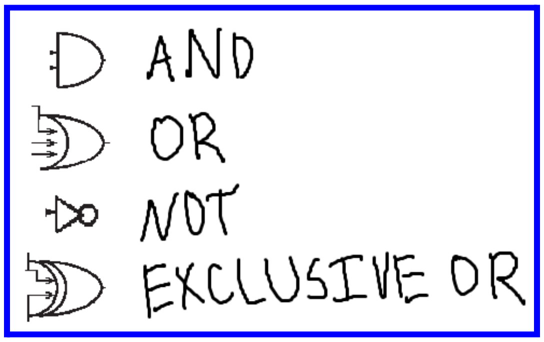
You can check the answer here: Boolean Algebra Calculator - eMathHelp.



	yz 00	yz 01	yz 11	yz 10
wx=00				
wx=01				
wx=11				
wx=10				

Figure 6.9: An empty K-map with four variables

A **gate** is a fundamental building block for digital devices; we will use them for AND, OR, NOT, and XOR. You draw arrows representing the elements going into them, and the arrow that comes out represents that result. Gates can be connected with other gates. Here are what the gates look like in drawings:



A **wire** connects gates and components and carries 1 bit of information. A **bus** is made up of parallel wires, carrying *n* bits of information, where *n* is the bus width. Note how the bus width is always written above the arrow.

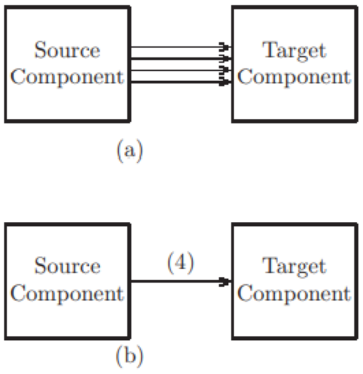
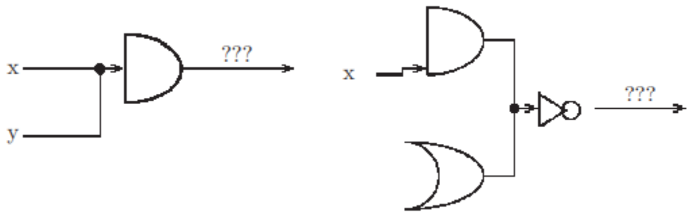


Figure 6.21: Two components connected by four wires, (a) The wires are shown explicitly, and (b) The wires are shown as a bus of width 4

A **contradiction** is an incorrect correction of wires. The wires should lead from one gate to another, not be combined unless there is a split or joining of buses. Some examples are shown below:



You can split and join buses, as long as they are properly split and joined with representative wires and arrows, like below (note that the bits specified are written there):

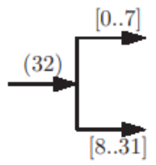


Figure 6.22: A 32-bit bus is split into an 8-bit bus and a 24-bit bus

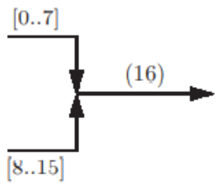


Figure 6.23: Two 8-bit buses are joined to form a 16-bit bus