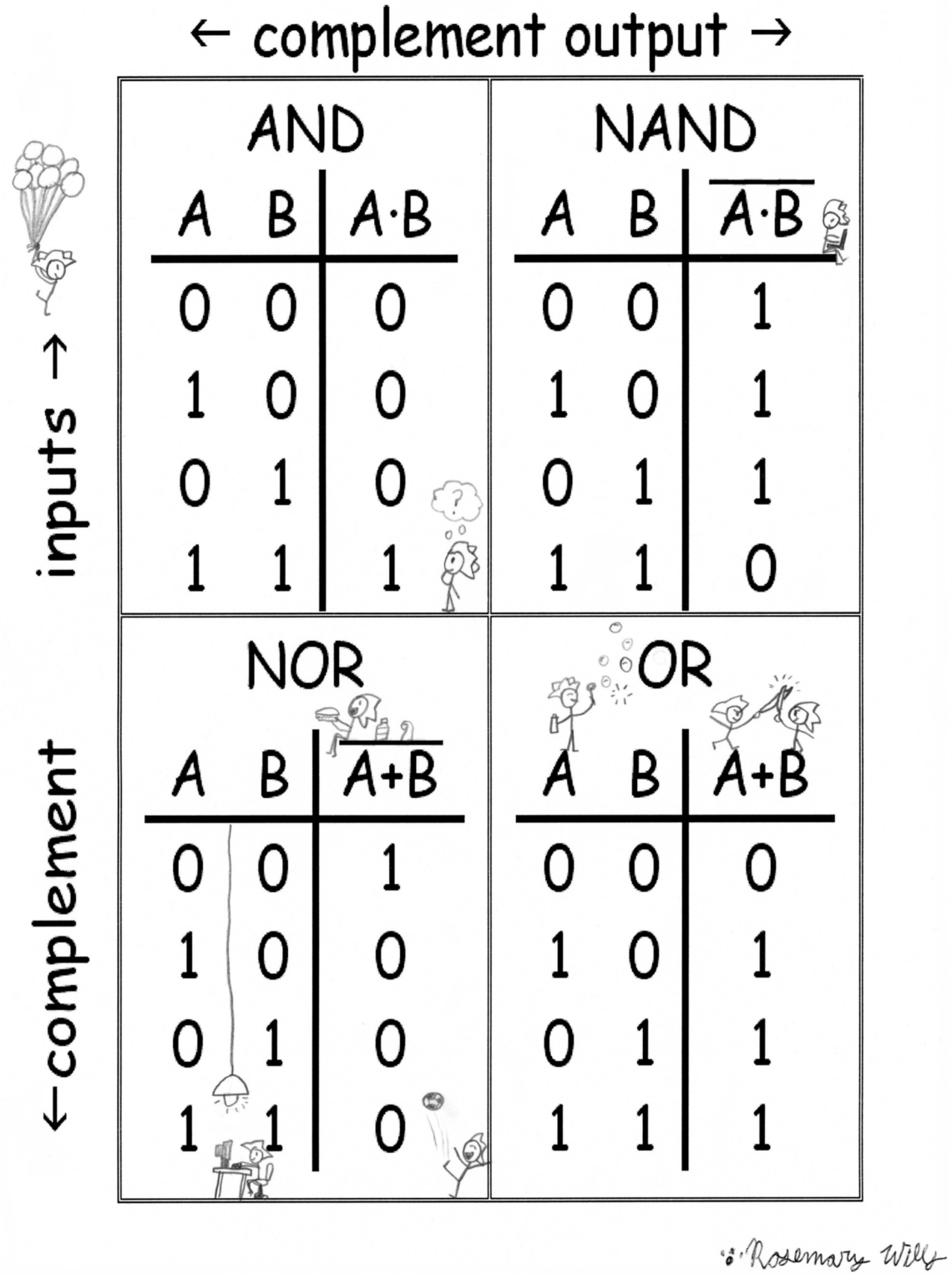
Boolean Algebra



Designing Computer Systems

Boolean Algebra

Programmable computers can exhibit amazing complexity and generality. And they do it all with simple operations on binary data. This is surprising since our world is full of quantitative computation. How can a computer complete complex tasks with simple skills?

A Little Logic: Computers use logic to solve problems. Computation is built from combinations of three logical operations: AND, OR, and NOT. Lucky for us, these operations have intuitive meanings.

AND	In order to get a good grade in ECE 2030, a student should come to class **AND** take good notes **AND** work study problems.
OR	Today's computers run Microsoft Windows 7 **OR** Mac OS X **OR** Linux.
NOT	Campus food is **NOT** a good value.

Surprisingly, these three functions underlie every operation performed by today's computers. To achieve usefulness and generality, we must be able to express them precisely and compactly. From an early age, we have used arithmetic expressions to represent equations with multi-valued variables and values.

$$Cost = X \cdot \$2.00 + Y \cdot \$1.50$$

In the world of logic, all variables have one of two values: *true* or *false*. But expressions can be written in the otherwise familiar form of an arithmetic expression. We'll use the "+" operator to represent OR and the "·" operator to represent AND. The following is a simple example of a Boolean expression:

$Out = A \cdot B + C$ Out is true if A AND B are true OR C is true

Just like in arithmetic expressions, operation precedence determines the order of evaluation. AND has higher precedence than OR just as multiplication has higher precedence than addition. Parentheses can be used to specify precise operation evaluation order if precedence is not right. Note that the expression below closely resembles the previous example. But it has a different behavior (e.g., consider each when A is false and C is true.)

$Out = A \cdot (B + C)$ Out is true if A is true AND (B OR C is true)

Is NOT enough?: NOT (also known as complement) is represented by a bar over a variable or expression. So $\overline{A}$ is the opposite of A (i.e., if A is true, $\overline{A}$ is false and vise versa). When a bar extends over an expression, (e.g., $\overline{A+B}$) the result of the

expression is complemented. When a bar extends over a subexpression, it implies that the subexpression is evaluated first and then complemented. It's like parentheses around the subexpression.

Many years ago in the 1800s, the mathematics of these binary variables and logical functions was described by a man named George Boole and a few of his colleagues. Now we call this mathematics *Boolean Algebra*.

Operation Behavior: These logical functions have intuitive behaviors. An AND expression is true if *all* of its variables are true. An OR expression is true if *any* of its variables are true. A NOT expression is true if its single variable is *false*.

Sometimes a table is used to specify the behavior of a Boolean expression. The table lists all possible input combinations of the right side and the resulting outputs on the left side. This behavior specification is called a truth table. Because "true" and "false" are hard to right compactly, we'll use 1 and 0 to represent these values. Here is a summary of AND, OR, and NOT behaviors using true tables.

A	B	A · B
0	0	0
1	0	0
0	1	0
1	1	1

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

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

Truth tables can have more than two inputs; just so long as all combinations of inputs values are included. If a combination was left out, then the behavior would not be fully specified. If there are i inputs, then there are 2^i combinations. It is also possible to have multiple outputs in a table, so long as all results are functions of the same inputs. Here are several Boolean expressions with three variables:

A	B	C	A · B · C	A + B + C	A · B + C	A · (B + C)
0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	1	0	0
1	1	0	0	1	1	1
0	0	1	0	1	1	0
1	0	1	0	1	1	1
0	1	1	0	1	1	0
1	1	1	1	1	1	1

Three variable AND and OR functions have expected behaviors. The AND output is true if all of the inputs are true. The OR output is true if any of the inputs are true. In the third expression, AND is higher precedence than OR. So the output is

true if either A AND B are true OR C is true. In the last expression, A must be true AND either B OR C (or both B and C) must be true for the output to be true.

There are a few basic properties of Boolean algebra that make it both familiar and convenient (plus a few new, not-so-familiar properties).

property	AND	OR
identity	$A \cdot 1 = A$	$A + 0 = A$
commutativity	$A \cdot B = B \cdot A$	$A + B = B + A$
associativity	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
distributivity	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
absorption	$A \cdot (A + B) = A$	$A + (A \cdot B) = A$

The identity, commutative, and associative properties are intuitive. Distributivity of AND over OR makes sense. OR over AND is new (don't try this with arithmetic addition over multiplication; it doesn't work!). Absorption is a new property of Boolean algebra. It comes in handy for simplifying expressions.

Generally, working with Boolean expressions is a lot like working with arithmetic expressions, with a few notable differences.

And that's NOT all: The complement (NOT) function adds an interesting dimension to the math. Where quantitative expressions have a rich range and domain for inputs and outputs, binary expression are decidedly limited. Any operation in a Boolean expression can have its inputs and/or its output complemented. But results will still be either true or false.

In fact most Boolean expression design extends the set of logical functions with NOTed AND (NAND) and NOTed OR (NOR). These functions are computed by complementing the result of the core operation.

AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1

NAND

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

OR

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

NOR

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

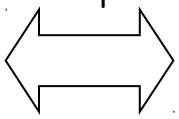
Here's where limited variable values and a small collection of basic operations leads to one of the most significant relationships in computation … DeMorgan's Theorem!

Sometimes the most amazing concepts are easy to see, when you look in the right way. In the table above, it's clear that NAND is just AND with its output complemented. All the zeros become ones and the one becomes zero. It's also clear that OR resembles NAND but for it being upside down. If all inputs to OR are complemented, the table flips and it matches NAND.

Complementing the inputs or the output of a NAND reverses this transformation. If inputs or an output is complemented twice, the function returns to its original behavior, leaving it unchanged. This supports reversible transformations between NAND and its left and right neighbors.

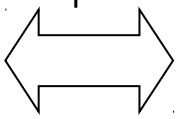
AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1



NAND

A	B	
0	0	1
1	0	1
0	1	1
1	1	0



OR

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

Note that the transformations to obtain the NAND function can be employed for any of the four logical functions. To determine the necessary neighbor functions, consider cutting out the **four** function table above and wrapping it into a cylinder where AND and NOR are now neighbors. Or better still, let's draw the four functions in a two dimensional table, shown below. This is DeMorgan's square and it shows how any logical function can be transformed into any other logical function using NOT gates.

← complement output →

inputs →

AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1

NAND

A	B	
0	0	1
1	0	1
0	1	1
1	1	0

←complement

NOR

A	B	
0	0	1
1	0	0
0	1	0
1	1	0

OR

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

You can start with a logical function, and by complementing all its inputs and/or its output, you can arrive at any other logical function. This has a profound effect on digital system design. Let's hear it for DeMorgan!

This principle can be applied to Boolean expressions as well. If you want to transform an OR into an AND, just complement all the OR inputs and its output. Let's try this process on a few expressions.

original expression	$A \cdot B$	$A \cdot (B + C)$	$A \cdot \overline{B} \cdot C$
AND becomes OR	$A + B$	$A + (B + C)$	$A + \overline{B} + C$
complement inputs	$\overline{A} + \overline{B}$	$\overline{A} + \overline{(B + C)}$	$\overline{A} + \overline{\overline{B}} + \overline{C}$
complement output	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A} + \overline{(B + C)}}$	$\overline{\overline{A} + \overline{\overline{B}} + \overline{C}}$
equivalent expression	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A} + \overline{(B + C)}}$	$\overline{\overline{A} + \overline{\overline{B}} + \overline{C}}$

In the first example, an AND function is turned into an OR function by complementing the inputs and the output. The second example has the same change, but one of the inputs to the AND is a subexpression. Note that when inputs are complemented, this subexpression receives a bar, but is otherwise unchanged. Just like this first input A, the subexpression is the input to the original AND function. The third example has a three input AND, so all three inputs must be complemented. Note also that the second input is already complemented. When it is complemented again, it has double bars. But when any variable or subexpression is complemented twice, the bars cancel out.

This DeMorgan transformation allows transformation of an OR to an AND using the same steps. It can be applied to the last evaluated function, the first evaluated function, or anything in between. It can even be applied to an entire expression (or subexpression) all at once … although some care must be exercised.

original expression	$A + (\overline{B} \cdot C)$	$\overline{(\overline{A} + B) \cdot (\overline{C} + D)}$	$\overline{A \cdot B + C} + \overline{D}$
swap AND and OR	$A \cdot (\overline{B} + C)$	$\overline{(\overline{A} \cdot B) + (\overline{C} \cdot D)}$	$(\overline{A + B}) \cdot \overline{C} \cdot \overline{D}$
complement inputs	$\overline{A} \cdot (\overline{\overline{B}} + \overline{C})$	$\overline{\overline{\overline{A}} \cdot \overline{B} + \overline{\overline{C}} \cdot \overline{D}}$	$(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot \overline{\overline{D}}$
complement output	$\overline{\overline{A} \cdot (\overline{\overline{B}} + \overline{C})}$	$\overline{\overline{\overline{A}} \cdot \overline{B} + \overline{\overline{C}} \cdot \overline{D}}$	$\overline{(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot D}$
equivalent expression	$\overline{\overline{A} \cdot (\overline{\overline{B}} + \overline{C})}$	$A \cdot \overline{B} + C \cdot \overline{D}$	$\overline{(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot D}$

In the first example, both AND and OR functions are swapped. Then all inputs and the output are complemented. One might ask why no bars are added on subexpressions (e.g., over $(\overline{B} \cdot C)$). The reason is that each subexpression is both an output for one function and an input for another. Since both are complemented,

the two bars cancel out. Only the input variables (e.g., A, B, and C) and the last function to be executed (the outermost function) will be complemented.

Note also that the function evaluation order is invariant throughout this process. In the first example, $\overline{B}$ is first ANDed with C. Then the result is ORed with A. After the transformation is complete, this is still the order. Often parentheses must be added to preserve this order since AND and OR have different precedence. Sometime parentheses can be dropped (like in the second example) since the new function precedence implies the correct (original) evaluation order.

In the second example, an initial bar over the outermost function (AND) is canceled when the entire expression is complemented. Note also that the bars over inputs are reversed. In the third example, a bar over the earlier OR function ($\overline{A \bullet B+C}$) remains unchanged through the transformation.

Eliminating Big Bars: Often implementation of Boolean expressions requires transforming them to a required form. For example, switch implementation needs a Boolean expression with complements (bars) only over the input variables (literals). If an expression has complements over larger subexpressions (big bars), DeMorgan's theorem must be applied to eliminate them. Here's an example.

	$Out=\overline{\overline{A+\overline{B}+\overline{C}\cdot D}}$	expression with many big bars
1	$\overline{\overline{A+\overline{B}}\cdot\overline{\overline{C}\cdot D}}$	replace final AND with OR, and
2	$Out=\overline{\overline{\overline{A+\overline{B}}}\cdot\overline{\overline{\overline{C}\cdot D}}}$	complement inputs and output
3	$Out=(A+\overline{B})\cdot\overline{\overline{\overline{C}\cdot D}}$	remove double bars
4	$(A+\overline{B})\cdot\overline{\overline{C}+D}$	replace first AND with OR, and
5	$Out=(A+\overline{B})\cdot\overline{\overline{\overline{C}+\overline{D}}}$	complement inputs and output
6	$Out=(A+\overline{B})\cdot(C+\overline{D})$	remove double bars

When eliminating big bars, one should start with the outermost complemented function. In this case, the OR in the center of the expression comes first. In step 1, it is replaced by an AND. The function's inputs and outputs are then complemented. Then double bars are removed. Note that parentheses must be added to maintain the same evaluation order. These first steps remove the big bars from the initial expression; but a new big bar is created over $\overline{C}$+D. So in step 4, this OR is replaced by an AND. Then its inputs and outputs are complemented. Again parentheses must be added to preserve the original evaluation order. The final expression (step 6) has an equivalent expression without big bars.

DeMorgan's Theorem allows us to transform a Boolean expression into many equivalent expressions. But which one is right? That depends on the situation. If we are designing an implementation with switches, eliminating big bars is an important step in the process. For gate design, we might want to use logical operations that better match the implementation technology. Regardless of implementation, we might just want to use a form of the expression that most clearly expresses (to a fellow engineer) the function we require.

In most cases, we can choose the equivalent expression that fits our needs. But how can we evaluate expressions for equivalence?

Standard Forms: There are two standard forms that offer a canonical representation of the expression. Let's explore these forms starting with a function's behavior in a truth table.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

To correctly express this function, we must show where its output is true (1) and where its output is false (0). We can accomplish this in two ways. Let's start with the "easy" one, expressing when the output is true. There are four cases.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Consider the first case, when A is true and B is false and C is false. We can create an expression to cover the case: $A \cdot \overline{B} \cdot \overline{C}$. If this were the *only* case where the output is true, this would accurately describe the function. It is an AND expression that contains all the inputs in their true (e.g., A) or complemented (e.g., $\overline{B}$) form. This is called a *minterm*. But there are three other cases. The output is true when $A \cdot \overline{B} \cdot \overline{C}$ is

true or when its the second case $A \cdot B \cdot \overline{C}$ or the third case $\overline{A} \cdot B \cdot C$ or the fourth case $A \cdot B \cdot C$. This behavior forms an OR expression.

$$\text{Out} = A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

Since this is an OR function applied to AND expressions, it's called a *Sum Of Products* (SOP). All inputs are included in each product term (minterms). So this becomes a canonical expression for the function's behavior: *a sum of products using minterms*. Everyone starting with this behavior will arrive at the identical Boolean expression.

If one works from bottom to top in the truth table, a different order of inputs can be derived.

$$\text{Out} = C \cdot B \cdot A + C \cdot B \cdot \overline{A} + \overline{C} \cdot B \cdot A + \overline{C} \cdot \overline{B} \cdot A$$

This is an *identical* expression (but for commutative ambiguity). It has the same logical operations applied to the same forms of the inputs.

You might notice that when B and C are true, the output is true, independent of A. The resulting expression becomes: Out = $A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + B \cdot C$. This is simpler, but not canonical since it is not composed of minterms.

There's another way to express this function behavior that is rooted in the binary world.

Popeye Logic: In the 1980 movie "Popeye", the title character is in denial about his father being the oppressive "Commodore" in their town, Sweet Haven ("My Papa ain't the Commodore!"). This denial is present when he asks directions to the Commodore's location ("Where ain't he?"). In our multivalued world, this is not so easy. While "north" is unambiguous, "not north" could be any direction except north. But in binary, things are different. We can state when something is true. Or we can use "Popeye Logic" and state when it is NOT false. Let's try Popeye logic on this behavior.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Suppose the only false value was the second one, when A is false, B is true, and C is false. If all the other outputs were true, we could express the function by stating when its *not* this case ("when it ain't $\overline{A}\cdot B\cdot\overline{C}$ "). As in the real world, this is more than one thing; it is all the truth table entries except for $\overline{A}\cdot B\cdot\overline{C}$. But binary makes expressing all the cases easier. In the expression $\overline{A}\cdot B\cdot\overline{C}$, A is false. So whenever A is true, the output is true. Or whenever B is false, the output is true. Or whenever C is true, the output is true. In fact, the function behavior is true whenever A is true *or* B is false *or* C is true ($A+\overline{B}+C$). This expression does not describe when $\overline{A}\cdot B\cdot\overline{C}$ is true, rather it covers all other cases, when "$\overline{A}\cdot B\cdot\overline{C}$ ain't true". The term $A+\overline{B}+C$ is an OR function of all inputs in their true or compliment form. This is a *maxterm*. But this only works if the second case was the only case when the output is false. What about when more than one case is false?

In this example, the output is false when $\overline{A}\cdot\overline{B}\cdot\overline{C}$ or $\overline{A}\cdot B\cdot\overline{C}$ or $\overline{A}\cdot\overline{B}\cdot C$ or $A\cdot\overline{B}\cdot C$. So showing when the output is true requires expressing when it is not any of these cases. It is not $\overline{A}\cdot\overline{B}\cdot\overline{C}$ when A is true or B is true or C is true ($A+B+C$). It is not $\overline{A}\cdot B\cdot\overline{C}$ when A is true or B is false or C is true ($A+\overline{B}+C$). It is not $\overline{A}\cdot\overline{B}\cdot C$ when A is true or B is true or C is false ($A+B+\overline{C}$). It is not $A\cdot\overline{B}\cdot C$ when A is false or B is true or C is false ($\overline{A}+B+\overline{C}$). But since the function output is only true when it is none of these cases, $A+B+C$, $A+\overline{B}+C$, $A+B+\overline{C}$, and $\overline{A}+B+\overline{C}$ must all be true for the function output to be true. So we can express the function:

$$\text{Out} = (A+B+C)\cdot(A+\overline{B}+C)\cdot(A+B+\overline{C})\cdot(\overline{A}+B+\overline{C})$$

Since this is an AND expression of OR terms, it is called a *Product of Sums* (POS). Using maxterms makes this canonical, but different from the sum of products using minterms. There is no direct way to transform a SOP using minterms expression into a POS expression using maxterms or vice versa. Standard forms provide a good way to clearly express a behavior.

Summary: Boolean algebra is the mathematics of digital computers. Here are the key points:

- Variables have one of two values (0 or 1).
- Functions include AND, OR, and NOT.
- A Boolean expression containing these functions can be used to specify a more complex behavior. Truth tables can also define this behavior.
- Boolean algebra exhibits many familiar and useful properties (plus some new ones).

- DeMorgan's square shows how any logical operation can be transformed into any other logical function by complementing the inputs and/or output.
- DeMorgan's Theorem allows Boolean expressions to be transformed into an equivalent expression that employs different logical functions.
- Standard forms provide a canonical expression in SOP and POS forms.

