Unknown date | Unknown author

Boolean Algebra

History

George Boole, born November 2, 1815, Lincoln, Lincolnshire, England, died December 8, 1864, Ballintemple, County Cork, Ireland

He was an English mathematician who helped establish modern symbolic logic and whose *algebra of logic*, now called *Boolean algebra*, is basic to the design of digital computer circuits.

(Enciclopaedia Britannica)

Basic Concepts

In Boolean algebra, an expression can only have one of two possible values: **TRUE** or **FALSE**. No other values exist. Therefore, in addition to many other uses, Boolean Algebra is used to describe digital circuits that also only have two states for each input or output.

The value **TRUE** is often represented by **1**, while **FALSE** is represented by **0**.

The core operators for Boolean Algebra are just three: **AND**, **OR**, and **NOT**.

Notation

Various different notations are used to express AND, OR, and NOT. The following variants are used in FIT1047 and are accepted for exam and assignments:

A AND B can be written as

A ∧ **B** or **AB**

A OR B can be written as

A ∨ **B** or **A+B**

NOT A can be written as

A or ¬ **A**

Although looking quite similar, the 0 and 1 in Boolean Algebra should not be confused with 0 and 1 in binary numbers. In particular as the short notation for AND and OR looks like mathematical operations for addition and multiplication, the behaviour in Boolean Algebra is different. To help distinguishing Boolean algebra terms from other mathematical terms, we use capital letters (A.B,C,...).

	Binary	Boolean
0+0	0	0 (because it is FALSE OR FALSE)
1+1	10 (equals 2)	1 (because it is TRUE OR TRUE)

Boolean Operators

The following paragraphs provide explanations of the concepts of AND, OR, and NOT. In principle, they are all very similar to the meaning of the words in natural language. However, there are some subtle differences that needs to be considered when using Boolean Algebra.

AND

A statement **A AND B** is only **TRUE** if both individual statements are TRUE.

AND in Boolean algebra matches our intuitive understanding of the word “and”.

Using 1 and 0, we can get a very compact representation as a **truth table**:

When you look at this truth table, what would be the value of TRUE AND FALSE? Reveal
FALSE, of course. Nothing can be true and false at the same time. Boolean logic is really very “black and white”.

OR

A OR B means that either A or B or both are TRUE

Note that OR in Boolean algebra is slightly different from our usual understanding of "or". Very often, the word "or" in natural language use means that either one or the other is true, but not both. For example, the sentence "I would like a toast with jam or honey." probably does mean that either a toast with jam or a toast with honey is okay, but not both. In Boolean algebra, the expression would mean that both together are okay as well.

Truth table for OR:

NOT

By just adding negation, Boolean algebra becomes a quite powerful tool that can be used to express complex logical statements, policies, or large digital circuits.

If a statement **A** is **TRUE**, then, the negation **NOT A** is **FALSE**. If a statement A is FALSE, then the negation NOT A is TRUE. Thus, negation just flips the value of a Boolean algebra statement from TRUE to FALSE or from FALSE to TRUE.

The truth table of NOT looks like this:

Other operators and gates

In principle, AND, OR, and NOT are sufficient to express everything we want. However, a few other operators are also frequently used:

XOR is the exclusive or that more resembles our intuitive understanding of "or". A XOR B is TRUE if either A is TRUE or B is TRUE, but not both.

NAND is the negated AND. A NAND B is TRUE only if at least one of A or B is FALSE.

NOR is the negated OR. A NOR B is TRUE only if both, A and B, are FALSE.

Can you write the truth table for NOT(A) AND NOT(B)? Which of the operators above has the same truth table? Reveal
Indeed, it is exactly the same truth table as NOR, the negated OR. The Boolean law for this is de Morgans law, which we will see a bit later.

From Boolean algebra to electrical (digital) circuits

In electrical circuits, AND, OR, NOT and additional operators (e.g. XOR, NAND, NOR) are realised as so-called logic gates.

A gate performs one (or several) logical operations on some logical input (i.e. bits) and produces a single logical output. We look at these electrical circuits on an abstract level as “logical circuits”. These logical circuits do not behave like electrical circuits. This can be a bit confusing, as a 0 input can result in a 1 output (e.g. when using NOT). This is the correct behaviour of a logical circuit, but it can contradict the intuitive understanding that in an electric circuit there can be no power at the output if there is no power coming in at the input side. Indeed, a logical gate for NOT only has one input and the negated output and a 0 input gets “magically” converted to a 1 output. The reason for this is that in an actual implementation, the NOT gate will have another input that provides power. Then, the NOT gate is actually

a switch that connects this additional power input to the output if the actual input is 0. In the idealised notion of logical circuits, these additional inputs are not shown, as they do not change state.

In logical circuits, the following symbols are used for the different types of gates.

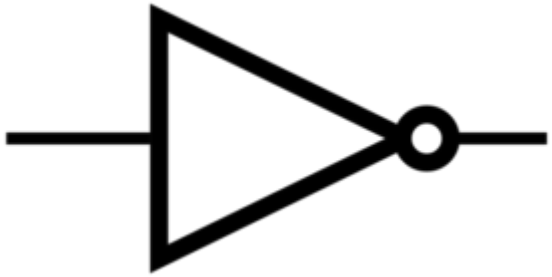
AND Gate

License: Public Domain

OR Gate

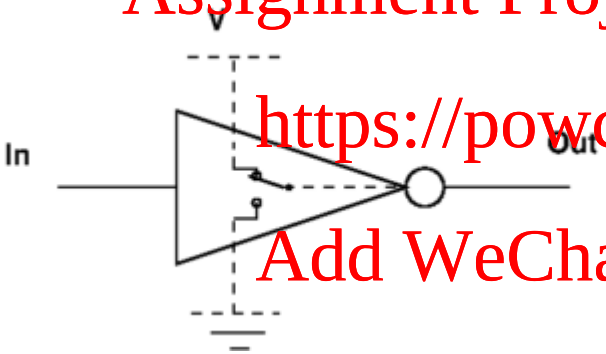
License: Public Domain

NOT Gate



License: Public Domain

A slightly less idealised version of the NOT gate would also show the additional connections that provide power to the gate:



License: Public Domain

These gates can be combined to create logical circuits for Boolean functions. The following video explains how to get from a truth table for a Boolean function to a logical circuit using the three basic gates for AND, OR, and NOT.

Boolean algebra rules

The term *Boolean algebra* implies that we might be able to do arithmetic on symbols. Indeed, there are a number of rules we can use to manipulate Boolean expressions in symbolic form, quite analogous to those rules of arithmetic. Some of these *laws* for Boolean algebra exist in versions for AND and OR.

Identity Law

Null Law (or Dominance Law)

Idempotent Law

Complement Law

Commutative Law

Associative Law

Distributive Law

Absorption Law

DeMorgans Law

Double Complement Law

Optimization of Boolean functions / K-maps

When realizing a function as circuit, one would like to minimize the number of gates used for the circuit. Obviously, the different Boolean laws can be used to derive smaller representations for Boolean functions. However, determining the correct order of applying the laws is sometimes difficult and trying different laws is not very efficient. In addition to having a smaller number of gates, one would also like to minimize the use of different types of gates. Therefore, normalized forms for Boolean functions can be useful.

One generic approach for minimizing (smaller) Boolean functions are **Karnaugh maps** or **K-maps**. The idea behind K-maps is to use a graphical representation to find cases where different terms in a Boolean formular can be combined to one simpler term. The simplification used in K-maps is based on the following observation. In

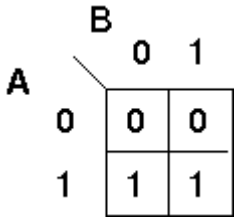
both cases, the function is independent from the value of B. Thus, B can be omitted.

K-maps are best understood by looking at a few examples.

A K-map with 2 variables

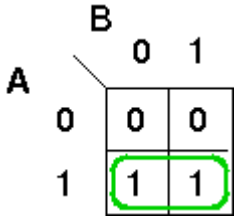
As described above, the following function is obviously independent from B. The example illustrates how this can be derived by using the graphical method of a K-map. The example uses the short notation (for and for).

In principle, a K-map is another type of truth table. The values of the two variables are noted on the top and the side of a table, while the function's value for a particular combination of inputs is noted within the table.



License: Public Domain

Now, the goal is to find groups of 1s in the map. Each group of ones that is not diagonal represents a group of terms that can be combined into one term. In the small example, there is only one group:



License: Public Domain

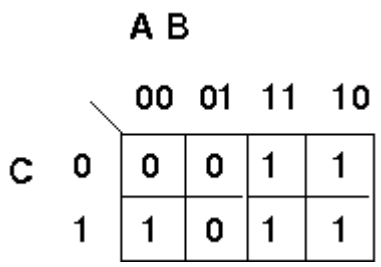
This group means that for the value of the function is 1 whatever the value of *B* is. Thus, the function can be minimized as follows.

A K-map with 3 variables

While the example with just 2 variables is quite obvious, it gets a bit less obvious with three variables. Now, the values of two of the variables are noted at one side and the other variable at the second side. Note that the order and choice of variables does not matter. The only important rule is that between two columns (and also two rows) there can only be one variable that is different. Thus, the order for two variables needs to be 00, 01, 11, 10. This is different to the order usually used in truth tables.

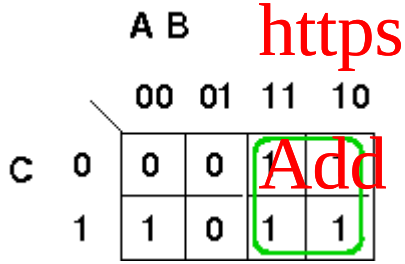
This example illustrates the use of K-maps for 3 variables.

We have 5 terms in this function. Thus, we have to place 5 1s into the K-map:

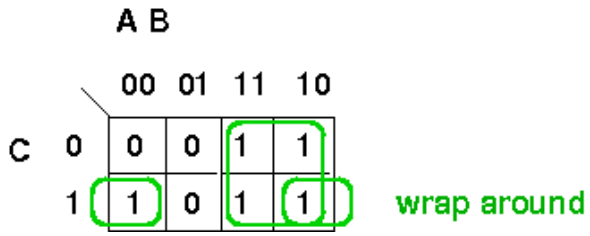


License: Public Domain

There is one large group with four 1s that is covering the complete space for A=1.



The remaining 1 seems to stand alone. Nevertheless, we can find a group by wrapping round to the other side of the table. This group covers two 1s for the area that is valid for C=1 and B=0. Thus, this group is independent from A.



License: Public Domain

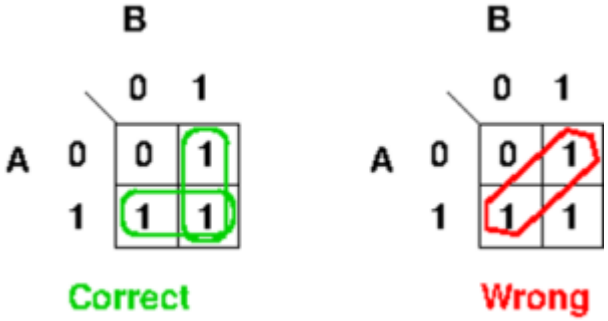
The minimized function now has only two terms representing the two groups in the K-map.

Rules for K-maps

Rule 1: No group can contain a **zero**.

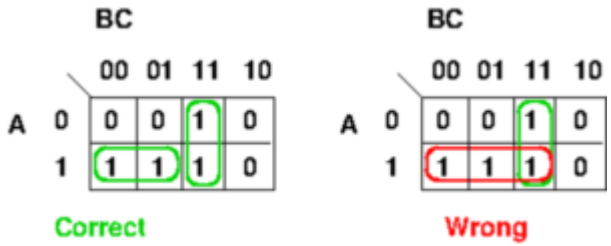
License: Public Domain

Rule 2: Groups may be horizontal/vertical/square, but **never diagonal**.



License: Public Domain

Rule 3: Groups must contain 1,2,4,8,16,32,... (**powers of 2**).

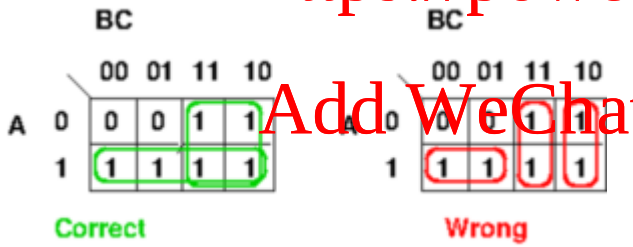


License: Public Domain

Rule 4: Each group must be as **large** as possible.

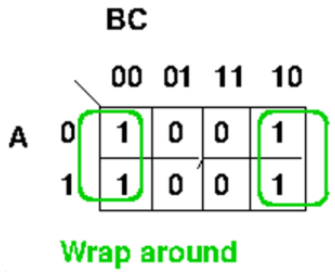
Rule 5: Groups can overlap.

Rule 6: Each **1** must be **part of at least one group**.



License: Public Domain

Rule 7: Groups may **wrap around** the map.



License: Public Domain

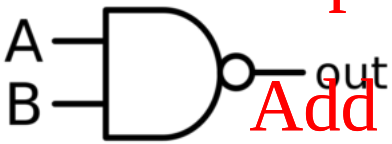
Universal gates

NAND has special properties:

> NAND can be physically implemented very efficiently.
> All other gates can be build only using NAND gates.

Thus, all logical circuits can be implemented using hardware with just a single type of gates. The same holds for NOR. Therefore, NAND and NOR are also called *universal gates*.

The symbol for a NAND gate is this:

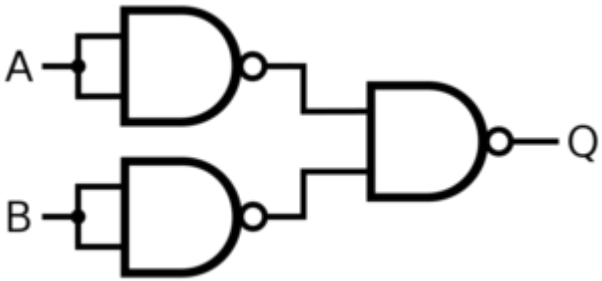


License: Public Domain

If NAND is negated, obviously the result is just AND. Thus, if we can realize NOT and OR with NAND, all three basic gates can be implemented just using NAND gates. The following circuits show how NOT and OR can be implemented just using NAND gates. Coorectness of these circuits is easily checked using truth tables.

Imlpementation of a NOT gate using NAND

License: Public Domain



Implementation of an OR gate using NAND