An introduction to Boolean Algebra and Logic in Computers

Logic is the business of propositions and propositions are statements that have a truth value. The logic algebra called Boolean algebra (after George Boole) is a two-valued (binary) algebra, a proposition can have the value yes or no, true or false, 1 or 0, high or low, there are no intermediate values. For example the statement "Today is Tuesday" is a proposition that is either true or false, it can't be partly true, today is either Tuesday or it isn't. Propositions can be combined by using operators, just like numbers can be combined by using operators. In arithmetic there are operations like 7 + 3 = 10, and, in logic there are equivalent operations like TRUE + TRUE = TRUE or A + B = C. In arithmetic there are laws and theorems which describe how numerical quantities behave and in logic there are laws and theorems which describe how logical quantities behave. The table below shows some of the propositions, laws and theorems which govern Boolean algebra.

a) A + A' = 1	b) A.A' = 0
c) A + 1 = 1	d) A.1 = A
e) A + A = A	f) A.A = A
g) A" = A	h) A.0 = 0
i) A + 0 = A	
j) (A + B) + C = A + (B + C)	k) (AB)C = A(BC)
l) A + B = B + A	m) A.B = B.A
n) A(B + C) = A.B + A.C	o) A + BC = (A + B)(A + C)
p) (A + B)' = A'B'	q) (AB)' = A' + B'
r) A + AB = A	s) A(A + B) = A

The items (a) to (i) are known as the elementary propositions, (j) and (k) are associative laws, (l) and (m) are commutative laws, (n) and (o) are distributive laws, (p) and (q) show DeMorgans theorem and (r) and (s) show the Absorption theorem.

The terms A and B and C stand for propositions, remember, something which can be true or false. The symbols + . and ' (plus dot and apostrophe) are logical operators. The + symbol means OR, the . symbol (dot) means AND, the ' symbol means NOT. The AND symbol is often left out (it is implied like the multiplication symbol in "normal" algebra), so AB means the same as A.B (A AND B). Expressions like A + A' = 1 read like "A OR NOT A equals 1". 1 means always TRUE, 0 means always FALSE, so the proposition A + A' = 1 simply states that something is either TRUE or FALSE.

The logical operations AND, OR, NOT, NAND and NOR are shown below (NOR means NOT OR and NAND means NOT AND). Each operation is shown in truth table form, ie all the possible states of the various propositions are enumerated and the results are tabulated.

AND Function

C = A.B	A	B	C
	0	0	0
	0	1	0
	1	0	0
	1	1	1

NOT Function

NOT A'	A	A'
	0	1
	1	0

NAND Function

C = (A.B)'	A	B	C
	0	0	1
	0	1	1
	1	0	1
	1	1	0

OR Function

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

NOR Function

C = (A + B)'	A	B	C
	0	0	1
	0	1	0
	1	0	0
	1	1	0

True and False, High and Low, Ones and Zeros

Usually when talking about propositions we use the terms true and false to represent the two possible values. There are other terms like high and low, one and zero which are also used. High and low are generally used when dealing with logic gates (a topic you meet soon), 1 and 0 are used just about anywhere but are particularly useful when using truth tables and Karnaugh maps.

TRUE	**FALSE**
HIGH	**LOW**
1	**0**

Using the Boolean Algebra

It is always a bit daunting learning an algebra, especially if it is many years since you have been to school. You should put algebra in it's place, it is just a general way of representing and manipulating values by using symbols. If you are told that the area of a room is it's length times it's breadth then this could be shown as A = LB, where A is area, L is length, B is breadth and by implication, L is multiplied by B. A = LB is a simple and concise way of expressing something. If you are told that a computer consists of a CPU, disk drive, memory, screen and keyboard then this could be written as C = PDMSK, where C represents Computer, P is CPU, D is disk drive, M is memory, S is screen and K is keyboard, as with the LB example, the AND operator is implied. In logic if P and D and M and S and K are all true then C is true.

The Absorption Theorem states that A + AB = A, ie the AB term is *absorbed* by the A term. Can it be proved algebraically?

A + AB = A	
A.1 + AB = A	by elementary proposition (d), since A.1 = A we can substitute A1 for A
A(1 + B) = A	the distributive law (n) states that A(B + C) = A.B + A.C so we can *undistribute* A from teh A1 and AB terms
A1 = A	elementary proposition (c) states that 1 + A is A, likewise 1 + B is B, 1 + ABCDEFG is ABCDEFG
A = A	elementary proposition (d) again

What about proving that (A' + B)(A + B')(A + B) = AB?

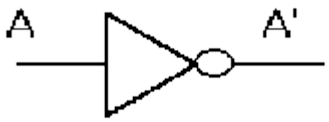
(A'+B)(A+B')(A+B) = (A'A + A'B + AB + B'B)(A + B)	use the distributive law to distribute (A'+B) over (A+B') ie multiply one by the other
(0 + A'B + AB + 0)(A+B) = (A'B+AB)(A+B)	elementary propositions (b) and (i) produce this line
AA'B + A'BB + AAB + ABB = 0 + 0 + AB + AB	apply the distributive law again and elementary proposition (e)
0 + 0 + AB + AB = AB	elementary propositions (i) and (e) simplify this line

Logic Gates

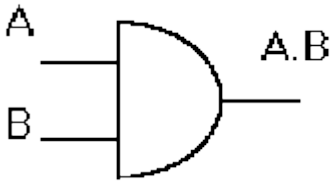
Logic gates are the building blocks of digital and computer circuits and implement the Boolean functions in silicon or some other semiconductor material. These gates can be bought in a variety of different packages and can range from single function gates similar to those shown here to the highly complex arrangements of gates found in computer integrated circuits. The devices you see soldered to the surfaces of mother boards and I/O cards contain logic gates, ie mechanisms for implementing Boolean functions. Many of these devices or chips are highly complex pieces of engineering built to exacting specifications. In recent years perhaps the single most important factor in the development has been the ability of engineers to package logic at increasingly higher densities.

Each boolean function has a logic gate and complex functions are built from the fundamental functions:

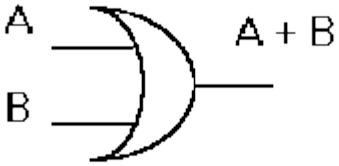
NOT (invertor). The single input is on the left of the gate, the output on the right.



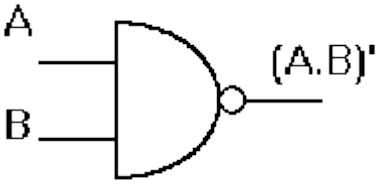
AND. This gate has two inputs, both on the left and a single output on the right.



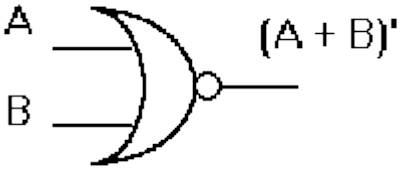
OR. This gate has two inputs, both on the left and a single output on the right.



NAND. This gate has two inputs, both on the left and a single output on the right.

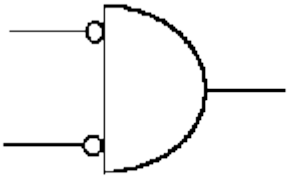


NOR. This gate has two inputs, both on the left and a single output on the right.

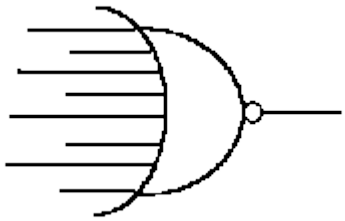


Each gate has one or more input connections and usually one output connection. As you will see below the connections are numbered so that they can be correlated to a physical package.

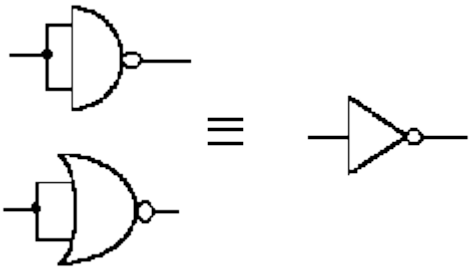
Note the use of an inversion symbol on the NAND and NOR gates. The bubble indicates inversion which could also appear on an input.

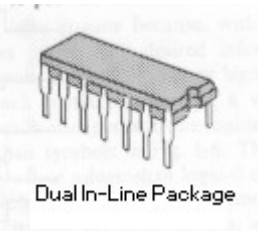


Gates are available with more than two inputs. The example here is an 8 input NOR gate

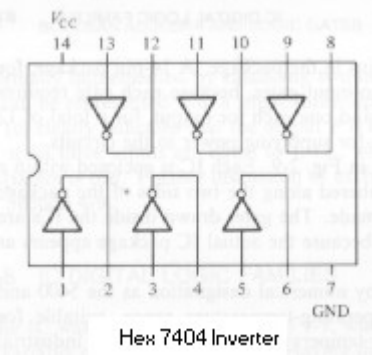


A NAND or NOR gate which has it's inputs connected together can function as a NOT gate

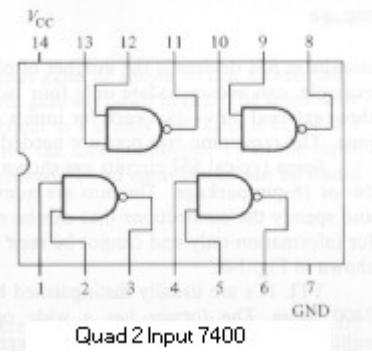




One way of packing the gates shown above is in small low-density 14/16/18 pin packages. Most electronic devices contain packages like these. Packages like those shown here have been around since the late '60's. More up-to-date packaging now achieves much higher densities, for example, 100,000's of thousands of gates per package.

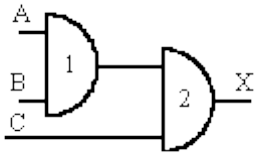


This diagram represents the contents of a specific logic package, a collection of NOT gates. It's described as a HEX inverter, ie 6 NOT gates.



Here is a package which contains 4 (quad) 2 input (A and B) NAND gates. The NAND gate is probably the most commonly used logic gate.

Logic Expressions in the Solid

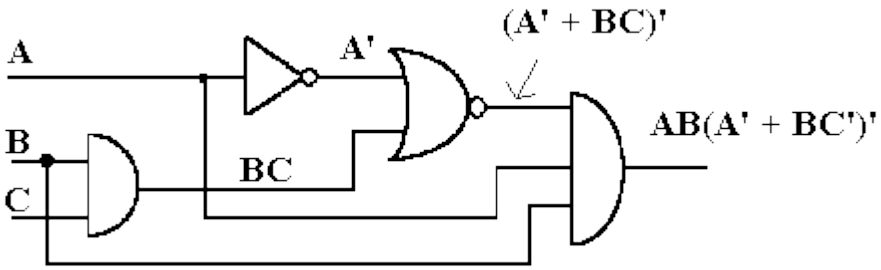


The symbols used in digital logic are just another way of writing logical expressions. The circuit shown here can be written as X = ABC since gates 1 and 2 are AND gates. The two variables A and B are both applied to the inputs of gate 1, the output of gate 1 (ie the term AB) is applied to one input of gate 2, C, another variable, is applied to the other input of gate 2. X is true when A is true and B is true and C is true.

Simplifying logical expressions

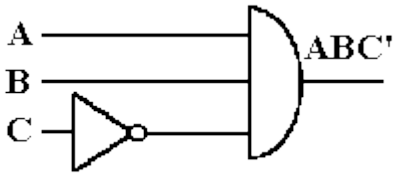
In the process of designing a logic circuit we may develop a series of logical expressions. To arrive at the optimum circuit we simplify the expressions and the objective of simplification may be lower cost or improved performance or both.

(Example 1) simplify the circuit:

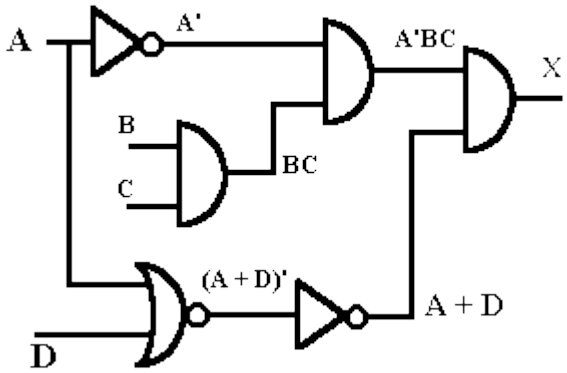


AB(A' + BC)' = ABA"(BC)' using DeMorgan's theorem
= ABA(BC)'
= AB(BC)'
= ABB' + ABC'
= A.0 + ABC

ABC' i.e



Evaluating a logic circuit output



If A = 0, B = 1, C = 1, D = 1, what is X?

To determine what value X has, first find the expression that represents X, ie $X = A'BC(A + D)$ then substitute the relevant values for the variables:

X = A'BC(A + D) = 0'.1.1.(0 + 1) = 1.1.1.(1) = 1.1.1.1 = 1

The Rules of Evaluation

1. NOT
2. Parentheses
3. AND before OR unless parentheses indicate otherwise

4. If expression is inverted, perform operations of the expression first and then invert the result.

Logic Conventions

If a function or variable is true when high we call that active high or positive logic. If it is true when low we call that active low or negative logic.

A	B	F	
0	0	1	What logic function
0	1	1	does this table
1	0	1	represent
1	1	0	

If you said NAND you are right for positive logic; but what if we assumed that the inputs and outputs are true when 0, what is it now? What about the case for negative logic inputs and positive logic outputs (ie input 0 is true but output 0 is false)? and for positive logic input and negative output (input 0 false, output 0 true)?

The logic conventions for the NAND gate can be summarised as:

Using a NAND gate as the functor

A	B	NAND	Input Convention	Output Convention	Gate Function
0	0	1	+ve	+ve	NAND
0	1	1	-ve	-ve	NOR
1	0	1	+ve	-ve	AND
1	1	0	-ve	+ve	OR

What do we get if a NOR gate is used as the functor?

Using a NOR gate as the functor

A	B	NOR	Input Convention	Output Convention	Gate function
0	0	1	+ve	+ve	NOR
0	1	0	-ve	-ve	NAND
1	0	0	+ve	-ve	OR
1	1	0	-ve	+ve	AND

Try building similar tables similar to the above for both AND and OR.

Minimising or Simplifying Boolean Expressions

In mathematics expressions are simplified for a number of reasons, for instance simpler expressions are easier to understand (and easier to write down), they are also less prone to error in interpretation but, most important, simplified expressions are usually more efficient and effective when implemented in practice. For example imagine you have devised some long and complex formula which describes how the efficiency of internal combustion engines can be improved by 50%. The formula indicates to an automotive engineer what he should do in practical terms. It may be that, as it stands, the formula may lead to a piece of gadgetry that is too expensive to build; if though the formula is simplified the gadget itself might also be simplified and the cost reduced. A Boolean expression, like any other algebraic expression, is composed of variables and terms, A variable is something like A or B, P or Q etc which represents a truth value and term is a collection of variables e.g. A'BC. The simplification of Boolean expressions can lead to more effective computer programs, algorithms and computer circuits.

Minimisation can be achieved by a number of methods, four well-known methods are:

- Algebraic manipulation
- Tree reduction
- Quine-McCluskey reduction
- Karnaugh maps

We will restrict ourselves to Algebraic manipulation and Karnaugh maps.

Algebraic Manipulation: Minterms and Maxterms and Duality

Minterms and maxterms are canonical expressions of a truth table line, and this means that the minterm (or maxterm) is a term which contains all the variables pertaining to that line of the truth table.

A minterm depicts variables in their true state, a maxterm depicts the complement of the minterm, for example:

A	B	C	Minterm	Maxterm
0	1	0	A'.B.C'	(A'.B.C')'

By DeMorgans theorem: (A'.B.C')' = A'' + B' + C'' = A + B' + C. A + B' + C is the usual form of a maxterm.

A	B	C	Minterm	Maxterm
0	1	0	A'.B.C'	A + B' + C

- **A minterm is a product of all the variables in their true state.**
- **A maxterm is a sum of all the variables in their complement state.**

Look at the truth table below:

A	B	F	Minterm	Maxterm
0	0	0	A'B'	A+B
0	1	1	A'B	A+B'
1	0	1	AB'	A'+B
1	1	1	AB	A'+B'

It is an OR function and any output which is 1 will satisfy the function F = A + B, ie any minterm which is 1 will satisfy the function F = A + B:

F = A'B + AB' + AB

In this form the expression is known as the **SUM OF PRODUCTS**. We can minimise the SOP, i.e. prove that:

A'B + AB' + AB = A + B

A'B+AB'+AB = A'B+A(B+B')	**. distributive over +**
A'B+A(B+B') = A'B + A.1	**B+B' = 1**
A'B+A = A+A'B	**commutative law**
A+A'B=(A+A')(A+B)	**+ distributive over .**
(A+A')(A+B) = 1.(A+B)	**A + A' = 1, A.1 = A**

And now with maxterms:

If a function is satisfied by any output which is 1 (i.e. any line in the truth table which is 1), then it is not satisfied by any output which is 0, or it is satisfied by the complement of any of the outputs which do not satisfy it. Remember a maxterm is the sum of all the variables in their complement state. The maxterms can be used to minimise an expression. In the OR case we can see that maxterm 1 (A + B) is the only maxterm which applies (i.e the only 0 output).

Here is an example using maxterms:

A	B	F	Minterm	Maxterm
0	0	0	A'B'	A+B
0	1	0	A'B	A+B'
1	0	0	AB'	A'+B
1	1	1	AB	A'+B'

We can state the **product of sums** as F = (A+B).(A+B').(A'+B)

(A+B).(A+B'). (A'+B)	= (AA + AB' + AB + BB'). (A'+B)	**distribute (A+B) over (A+B')**
	= (A + AB' + 0)(A'+B)	**AA = A, BB' = 0, A + AB + AB' = A (absorption)**
	= A(A' + B)	**distribute A over (A'+B)**
	= AA' + AB	**AA'=0, 0 + AB = AB**
	= AB	

Look at the function:

F = A'B'C' + A'BC + ABC (ie minterms 0, 3 and 7)

It is a sum of minterms form. A truth table can be constructed for it:

A	B	C	F	minterms	maxterms
0	0	0	1	A'B'C'	A+B+C
0	0	1	0	A'B'C	A+B+C'
0	1	0	0	A'BC'	A+B'+C
0	1	1	1	A'BC	A+B'+C'
1	0	0	0	AB'C'	A'+B+C
1	0	1	0	AB'C	A'+B+C'
1	1	0	0	ABC'	A'+B'+C
1	1	1	1	ABC	A'+B'+C'

The complement of the function can be obtained by the sum of products when F is false:

F' = A'B'C + A'BC' + AB'C' + AB'C + ABC'

So why is this interesting? There is a **principle of duality** that states **"the dual of a function can be found by exchanging operators and identity elements"**, i.e. ANDs become ORs (and vice-versa) and 1's become 0's (and vice-versa).

The dual of F' is then: F = (A+B+C')(A+B'+C)(A'+B+C)(A'+B+C')(A'+B'+C) which tells us that the Sum of Minterms 0,3 and 7 is equal to the Product of Maxterms 1,2,4,5 and 6, i.e.

F = A'B'C' + A'BC + ABC
(A+B+C')(A+B'+C)(A'+B+C)(A'+B+C')(A'+B'+C)

The truth table also indicates the simplest way in which the function can be implemented.

In this example:

- SOP requires three terms
- POS requires 5 terms

More on the canonical forms:

A canonical form is with all terms and variables included.

- A Boolean expression involving 3 variables has a maximum of 8 3-variable-terms.
- A Boolean expression involving 4 variables has a maximum of 16 4-variable-terms.
- A Boolean expression involving n variables has a maximum of 2n n-variable-terms.

Minimising a Sum of Products (minterms)

Given F=A'B'C+A'BC'+AB'C+ABC

Apply commutative law to re-arrange the terms:	F=A'B'C+AB'C+ABC+A'BC'
Use A + A = A to introduce a term twice:	F=A'B'C+AB'C+AB'C+ABC+A'BC'
	i.e. AB'C + AB'C = AB'C
Apply distributive law :	F=A'B'C+AB'C+AB'C+ABC+A'BC'
	=B'C(A'+A) + AC(B'+B) + A'BC'
Use A+A'=1 and A.1 = A	F =B'C(A'+A) + AC(B'+B) + A'BC'
	=B'C.1 + AC.1 + A'BC'
	= B'C + AC + A'BC'

Minimising a Product of Sums (maxterms)

Given F=(A+B+C)(A+B'+C')(A'+B+C)(A'+B'+C)

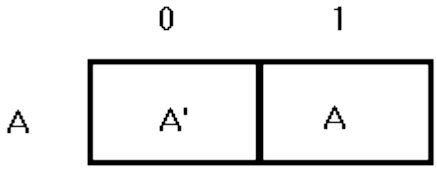
Use distributive law on the first two terms:	F = (AA+AB'+AC'+AB+BB'+BC'+AC+B'C+CC') (A'+B+C)(A'+B'+C)
Use A.A = A and A.A' = 0	F = (A+AB'+AC'+AB+0+BC'+B'C+0)(A'+B+C) (A'+B'+C)
	=(A+AB'+AC'+AB+BC'+B'C)(A'+B+C)(A'+B'+C)
	=(A+AB'+AB+AC'+BC'+B'C)(A'+B+C)(A'+B'+C)
Use absorption theorem to absorb those terms containing A or "undistribute" :	F = (A+A(B'+B)+A(C'+C)+BC'+B'C)(A'+B+C) (A'+B'+C)
	F = (A+BC'+B'C)(A'+B+C)(A'+B'+C)
Apply step 1 again to the first 2 terms of result 3:	F = (AA'+AB+AC+A'BC'+BBC'+BC'C+A'B'C+B'B'C+ B'CC)(A'+B'+C)
Use A.A' = 0 and A.A = A and absorption:	F = (AB+AC+A'BC'+BC'+A'B'C+B'C)(A'+B'+C)
Use distribution and A + 1 = 1:	F = (AB+AC+A'BC'+BC'+A'B'C+B'C)(A'+B'+C)
	=(AB + AC + BC'(A'+1) + B'C(A'+1))(A'+B'+C)
	=(AB + AC + BC'+ B'C)(A'+B'+C)
Step 1 again for the remaining terms:	F = (A'AB + ABB' + ABC + A'AC + AB'C + ACC + A'BC' + B'BC' + BC'C + A'B'C + B'B'C + B'CC)
Use A.A' = 0	F = (0 + 0 + ABC + 0 + AB'C + AC + A'BC' + 0 + 0 + A'B'C + B'C + B'C)
	= (ABC + AB'C + AC + A'BC' + A'B'C + B'C)
Absorb A'B'C + B'C and distribute:	F = (ABC + AB'C + AC + A'BC' + A'B'C + B'C)
	= (AC(B + B') + AC + A'BC' + B'C)
	= AC + B'C + A'BC'

Karnaugh Maps - Introduction

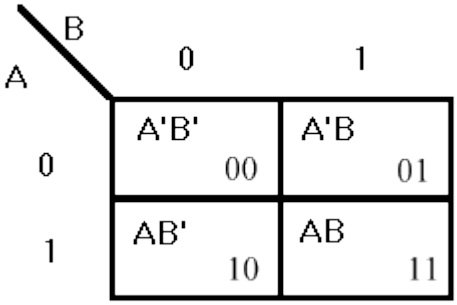
The process of minimising boolean expressions using algebraic manipulation can be quite difficult since we have to determine which of the various laws and propositions to use. The Karnaugh Map (KMAP) provides a simpler method of minimising expressions. It was first proposed by E.W. Veitch and later modified by M.Karnaugh.

The KMAP is a pictorial representation of a truth table and displays a number of cells, each of which is separated from it's neighbours by the unit distance (1 bit) between boolean terms:

Single variable KMAP



Two variable KMAP



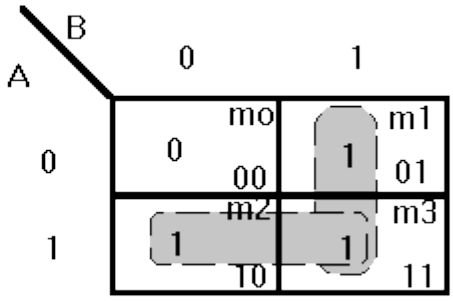
The two variable KMAP shows the variables and values for each cell. A'B' is value 00 or minterm 0 (m0).

Using a two variable KMAP

Given the Sum Of Products F = A'B + AB' + AB

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

≡



The KMAP on the right is equivalent to the truth table on the left. Note in the KMAP that only 1 bit changes between adjacent cells. This 1 bit difference is called a **unit distance**.

To use the KMAP:

To use the KMAP:

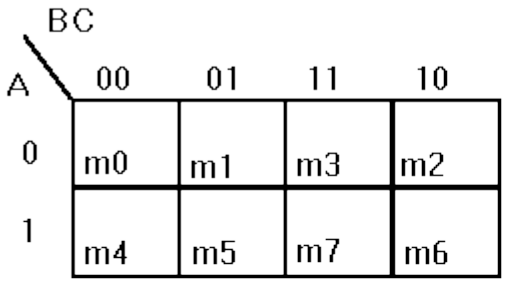
1. Insert a 1 in each cell where required (i.e. where each minterm is true).
2. Simplify cells separated by a unit distance, e.g. cells m1 and m3 are a unit distance apart. These simplify as A'B + AB = B, and cells m2 and m3 are also a unit distance apart, so AB'

+ AB = A

When comparing cells separated by a unit distance we can eliminate any variable which changes value between the adjacent cells.

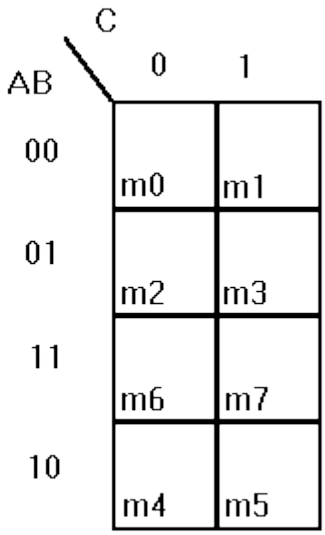
Three variable KMAP

The three variable truth table can be drawn as a 3 variable KMAP. The table below shows where each minterm appears in the KMAP. The rule that each cell differs from the neighbouring cells by a unit distance still applies. Note also that the KMAP is continuous at it's edges so (in the right-hand KMAP) m4 differs from m0 by a unit distance and m5 differs from m1 by a unit distance.



this way

or



this way

A	B	C	minterms
0	0	0	m0
0	0	1	m1
0	1	0	m2
0	1	1	m3
1	0	0	m4
1	0	1	m5
1	1	0	m6
1	1	1	m7

Four variable KMAP

AB \ CD	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

4 variable KMAPS are minimised in the same way as 3 variable maps, the same unit distance rule applies:

- One cell represents one minterm, giving a term of four variables.
- Two adjacent cells represent a term of three variables.
- Four adjacent cells represent a term of two variables.
- Eight adjacent cells represent a term of one variable.
- Sixteen adjacent cells represent a function which is always true.

Example - Simplify, using a KMAP, the expression:

$$F(A, B, C, D) = \sum(0, 1, 2, 4, 5, 6, 7, 9, 12, 13, 14)$$

First, sketch the KMAP and insert a 1 for each minterm that is true:

AB \ CD	00	01	11	10
00	1	1		1
01	1	1		1
11	1	1		1
10	1	1		

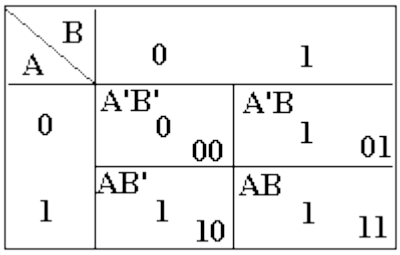
next, examine the KMAP and combine those cells which contain a 1 bit and which are separated by a unit distance:

- There are 8 adjacent cells (m0,m1,m4,5,m12,m13,m8,m9) which will simplify to a single

literal, C', ie the only variable which is always true in this area is C".

- There are two groups of 4 adjacent cells (m0,m4,m2,m6) and (m4,m12,m6,m14) which will combine to give two terms of two literals, A'D' and BD'. Since the KMAP "wraps" around at it's edges then cell pairs like (m4,m6) are a unit distance apart.
- All cells have been combined and the final expression is: F = C' + A'D' + BD'

Karnaugh Maps and non-canonical expressions



Assume you need to simplify the expression A + A'B + AB. You can see that one of the terms A is not in canonical from (ie the term doesn't contain all the variables, it's just a bit of a minterm or maxterm). How do you draw a KMAP with only partial terms? The KMAP here shows one way. Here you can see that each cell which contains A (ie A is true) has a 1 bit. In this simple case that means cells AB and AB'. Since the term AB' is already in the expression then a 1 appears in this cell anyway, you don't need to enter another 1 bit.

The Boolean Algebra trail finishes here. I hope you found it useful.