Combinational circuit
&
Simplification using KARNAUGH MAPS

The combinational circuit consist of logic gates whose outputs at any time is determined directly from the present combination of input without any regard to the previous input. A combinational circuit performs a specific information processing operation fully specified logically by a set of Boolean functions.

A *combinatorial circuit* is a generalized gate. In general such a circuit has *m* inputs and *n* outputs. Such a circuit can always be constructed as *n* separate combinatorial circuits, each with exactly one output. For that reason, some texts only discuss combinatorial circuits with exactly one output. In reality, however, some important *sharing of intermediate signals* may take place if the entire *n*-output circuit is constructed at once. Such sharing can significantly reduce the number of gates required to build the circuit.

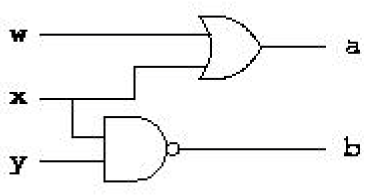
When we build a combinatorial circuit from some kind of specification, we always try to make it *as good as possible*. The only problem is that the definition of "as good as possible" may vary greatly. In some applications, we simply want to minimize the number of gates (or the number of transistors, really). In other, we might be interested in as short a *delay* (the time it takes a signal to traverse the circuit) as possible, or in as low *power consumption* as possible. In general, a mixture of such criteria must be applied.

Describing existing circuits using Truth tables

To specify the exact way in which a combinatorial circuit works, we might use different methods, such as logical expressions or *truth tables*.

A truth table is a complete enumeration of all possible combinations of input values, each one with its associated output value.

When used to describe an existing circuit, output values are (of course) either 0 or 1. Suppose for instance that we wish to make a truth table for the following circuit:



All we need to do to establish a truth table for this circuit is to compute the output value for the circuit for each possible combination of input values. We obtain the following truth table:

w	x	y	a	b
0	0	0	0	1
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

Specifying circuits to build

When used as a *specification* for a circuit, a table may have some output values that are not specified, perhaps because the corresponding combination of input values can never occur in the particular application. We can indicate such unspecified output values with a dash -.

For instance, let us suppose we want a circuit of four inputs, interpreted as two nonnegative binary integers of two binary digits each, and two outputs, interpreted as the nonnegative binary integer giving the quotient between the two input numbers. Since division is not defined when the denominator is zero, we do not care what the output value is in this case. Of the sixteen entries in the truth table, four have a zero denominator. Here is the truth table:

x1	x0	y1	y0	z1	z0
0	0	0	0	-	-
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	-	-
0	1	0	1	0	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	-	-
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	-	-
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

Unspecified output values like this can greatly decrease the number of circuits necessary to build the circuit. The reason is simple: when we are free to choose the output value in a particular situation, we choose the one that gives the fewest total number of gates.

Circuit minimization is a difficult problem from complexity point of view. Computer programs that try to optimize circuit design apply a number of *heuristics* to improve speed. In this course, we are not concerned with optimality. We are therefore only going to discuss a simple method that works for all possible combinatorial circuits (but that can waste large numbers of gates).

A separate single-output circuit is built for each output of the combinatorial circuit.

Our simple method starts with the truth table (or rather *one of the acceptable truth tables*, in case we have a choice). Our circuit is going to be a two-layer circuit. The first layer of the circuit will have at most 2^n *AND*-gates, each with *n* inputs (where *n* is the number of inputs of the combinatorial circuit). The second layer will have a single *OR*-gate with as many inputs as there are gates in the first layer. For each line of the truth table with an output value of 1, we put down a *AND*-gate with *n* inputs. For each input entry in the table with a 1 in it, we connect an input of the *AND*-gate to the corresponding input. For each entry in the table with a 0 in it, we connect an input of the *AND*-gate to the corresponding input *inverted*.

The output of each AND-gate of the fist layer is then connected to an input of the OR-gate of the second layer.

As an example of our general method, consider the following truth table (where a - indicates that we don't care what value is chosen):

x	y	z	a	b
0	0	0	-	0
0	0	1	1	1
0	1	0	1	-
0	1	1	0	0
1	0	0	0	1
1	0	1	0	-
1	1	0	-	-
1	1	1	1	0

The first step is to arbitrarily choose values for the undefined outputs. With out simple method, the best solution is to choose a 0 for each such undefined output. We get this table:

x	y	z	a	b
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	0

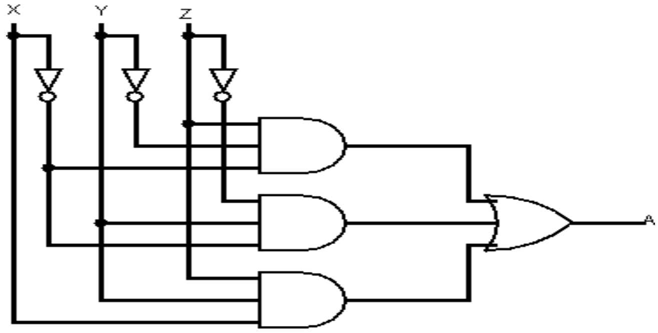
Now, we have to build two separate single-output circuits, one for the a column and one for the b column.

A=x′y′z+x′yz′+xyz

B=x′y′z+xy′z′

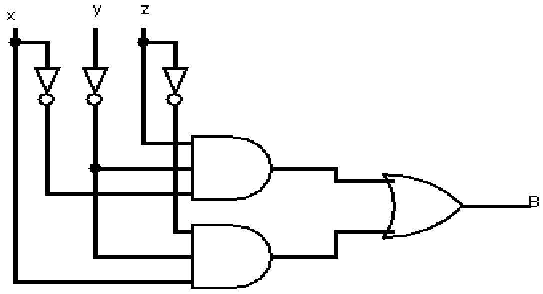
For the first column, we get three 3-input *AND*-gates in the first layer, and a 3-input *OR*-gate in the second layer. We get three *AND* -gates since there are three rows in the a column with a value of 1. Each one has 3-inputs since there are three inputs, x, y, and z of the circuit. We get a 3-input *OR*-gate in the second layer since there are three *AND* -gates in the first layer.

Here is the complete circuit for the first column:

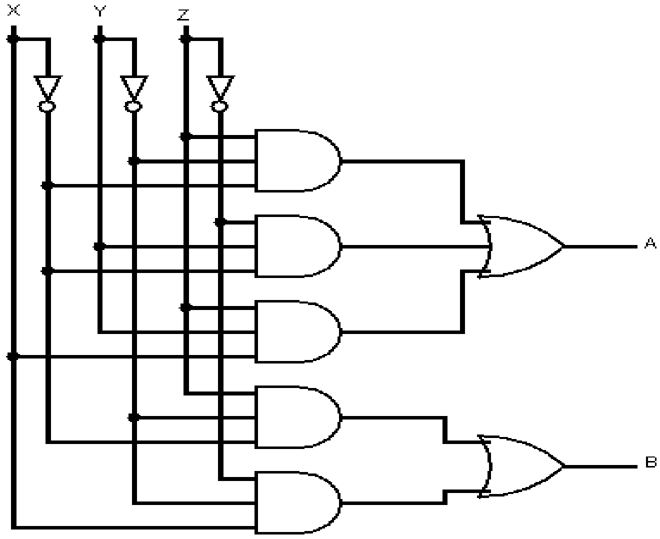


For the second column, we get two 3-input *AND* -gates in the first layer, and a 2-input *OR*-gate in the second layer. We get two *AND*-gates since there are two rows in the b column with a value of 1. Each one has 3-inputs since again there are three inputs, x, y, and z of the circuit. We get a 2-input *AND*-gate in the second layer since there are two *AND*-gates in the first layer.

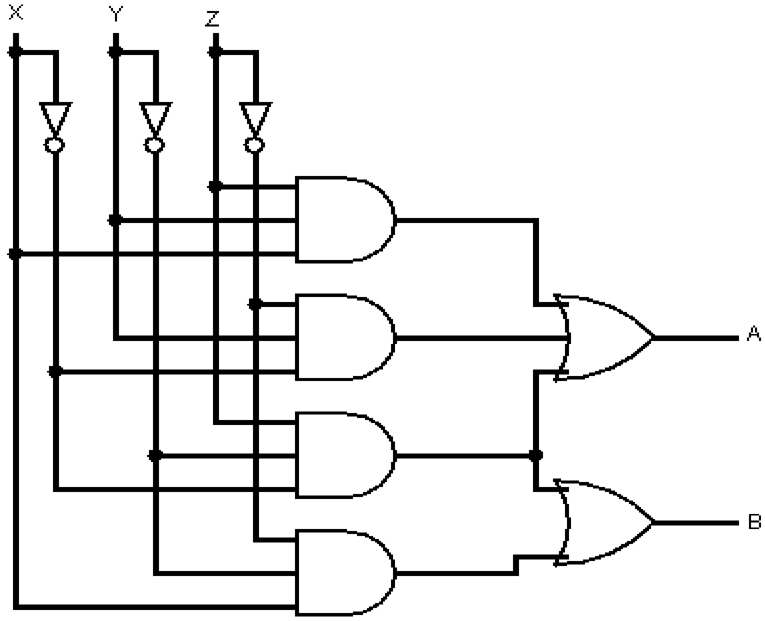
Here is the complete circuit for the second column:



Now, all we have to do is to combine the two circuits into a single one:



While this circuit works, it is not the one with the fewest number of gates. In fact, since both output columns have a 1 in the row correspoding to the inputs 0 0 1, it is clear that the gate for that row can be shared between the two subcircuits:



In some cases, even smaller circuits can be obtained, if one is willing to accept more layers (and thus a higher circuit delay).

Boolean functions

Operations of binary variables can be described by mean of appropriate mathematical function called Boolean function. A Boolean function define a mapping from a set of binary input values into a set of output values. A Boolean function is formed with binary variables, the binary operators AND and OR and the unary operator NOT.

For example , a Boolean function $f(x_{1,}x_{2,}x_{3,}\ldots\ldots,x_n)$ =y defines a mapping from an arbitrary combination of binary input values $(x_{1,}x_{2,}x_{3,}\ldots\ldots,x_n)$ into a binary value y. a binary function with n input variable can operate on 2^n distincts values. Any such function can be described by using a truth table consisting of 2^n rows and n columns. The content of this table are the values produced by that function when applied to all the possible combination of the n input variable.

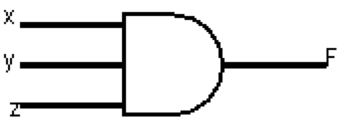
Example

x	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

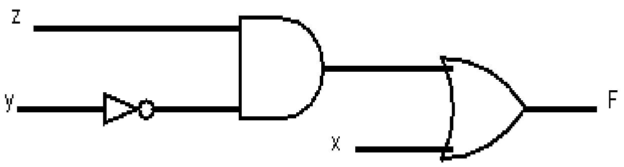
The function f, representing x.y, that is f(x,y)=xy. Which mean that f=1 if x=1 and y=1 and f=0 otherwise.

For each rows of the table, there is a value of the function equal to 1 or 0. The function f is equal to the sum of all rows that gives a value of 1.

A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gate. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate and each term is implemented with a logic gate . e.g.



F=xyz



F=x+y′z

Complement of a function

The complement of a function F is F′ and is obtained from an interchange of 0’s to 1’s and 1’s to 0’s in the value of F. the complement of a function may be derived algebraically trough De Morgan’s theorem

(A+B+C+….)′= A′B′C′….
 (ABC….)′= A′+ B′+C′……

The generalized form of de Morgan’s theorem state that the complement of function is obtained by interchanging AND and OR and complementing each literal.

F=X′YZ′+X′Y′Z′
F′=(X′YZ′+X′Y′Z′)′
=(X′YZ′)′.(X′Y′Z′)′
=(X″+Y′+Z″)(X″+Y″+Z″)
=(X+Y′+Z)(X+Y+Z)

Canonical form(Minterns and Maxterms)

A binary variable may appear either in it normal form or in it complement form . consider two binary variables x and y combined with AND operation. Since each variable may appears in either form there are four possible combinations: x′y′, x′y, xy′,xy. Each of the term represent one distinct area in the Venn diagram and is called minterm or a standard product. With n variable, 2^n minterms can be formed.

In a similar fashion, n variables forming an OR term provide 2^n possible combinations called maxterms or standard sum. Each maxterm is obtained from an OR term of the n variables, with each variable being primed if the corresponding bit is 1 and un-primed if the corresponding bit is 0. Note that each maxterm is the complement of its corresponding minterm and vice versa.

X	Y	Z	Minterm	maxterm
0	0	0	x′y′z′	X+y+z
0	0	1	X′y′z	X+y+z′
0	1	0	X′yz′	X+y′+z
0	1	1	X′yz	X+y′+z′
1	0	0	Xy′z′	X′+y+z
1	0	1	Xy′z	X′+y+z′
1	1	0	Xyz′	X′+y′+z
1	1	1	Xyz	X′+y′+z′

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of variable that produce a 1 and taken the OR of those terms.

Similarly, the same function can be obtained by forming the maxterm for each combination of variable that produces 0 and then taken the AND of those term.

It is sometime the following notation:

F(X,Y,Z)=∑(1,4,5,6,7) . the summation symbol∑ stands for the ORing of the terms; the number follow ing it are the minterms of the function. The letters in the parenthesis following F form list of the variables in the order taken when the minterm is converted to an AND term.

So, F(X,Y,Z)=∑(1,4,5,6,7) = X’Y’Z+XY’Z’+XY’Z+XYZ’+XYZ

Sometime it is convenient to express a Boolean function in its sum of minterm. If it is not in that case, the expression is expanded into the sum of AND term and if there is any missing variable, it is ANDed with an expression such as x+x′ where x is one of the missing variable.

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This can be done by using distributive law x+xz=(x+y)(x+z). then if there is any missing variable, say x in each OR term is ORded with xx′.

e.g. represent F=xy+x′z as a product of maxterm

=(xy +x′)(xy+z)
(x+x′)(y+x′)(x+z)(y+z)
(y+x′)(x+z)(y+z)
Adding missing variable in each term
(y+x′)= x′+y+zz′ =(x′+y+z)(x′+y+z′)
(x+z)= x+z+yy′ =(x+y+z)(x+y′+z)

(y+z)= y+z+xx′ =(x+y+z)(x′+y+z)
F= (x+y+z)(x+y′+z) (x′+y+z)(x′+y+z′)

A convenient way to express this function is as follow :

F(x,y,z)= ∏ (0,2,4,5)

Standard form

Another way to express a boolean function is in satndard form. Here the term that form the function may contains one, two or nay number of literals. There are two types of standard form. The sum of product and the product of sum.

The sum of product(SOP) is a Boolean expression containing AND terms called product term of one or more literals each. The sum denotes the ORing of these terms

e.g. F=x+xy′+x′yz

the product of sum (POS)is a Boolean expression containing OR terms called SUM terms. Each term may have any number of literals. The product denotes the ANDing of these terms

e.g. F= x(x+y′)(x′+y+z)

a boolean function may also be expressed in a non standard form. In that case, distributive law can be used to remove the parenthesis

F=(xy+zw)(x′y′+z′w′)

= xy(x′y′+z′w′)+zw(x′y′+z′w′)

=Xyx′y +xyz′w′ +zwx′y′ +zwz′w′

=xyz′w′+zwx′y′

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only methods is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar

Describing existing circuits using Logic expressions

To define what a combinatorial circuit does, we can use a *logic expression* or an *expression* for short. Such an expression uses the two constants 0 and 1, variables such as *x*, *y*, and *z* (sometimes with suffixes) as names of inputs and outputs, and the operators +, **.** and a horizontal bar or a prime (which stands for *not*). As usual, multiplication is considered to have higher priority than addition. Parentheses are used to modify the priority.

If Boolean functions in either Sum of Product or Product of Sum forms can be implemented using 2-Level implementations.

For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

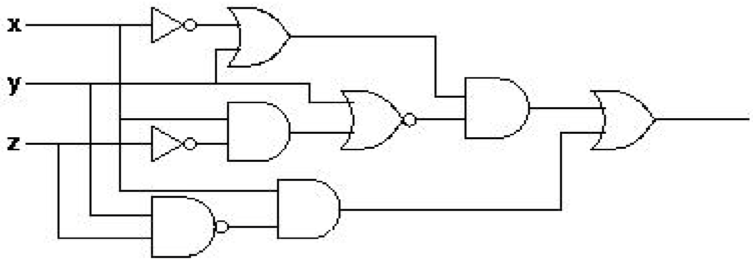
For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Note that using inverters to complement input variables is not counted as a level.

Examples:

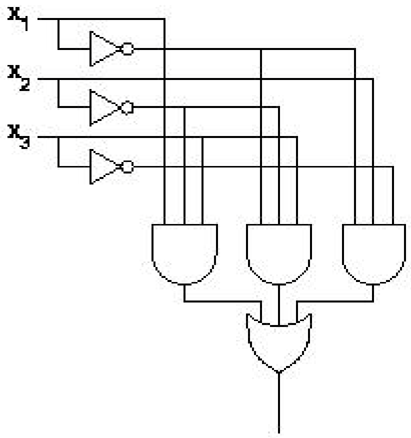
$(X'+Y)(Y+XZ')'+X(YZ)'$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



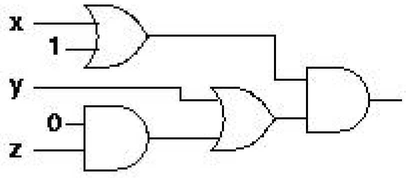
$X_1X_2'X_3+X_1'X_2'X_2+X_1'X_2X'_3$

The equation is in sum of product. The implementation is in 2-Levels. AND gates form the first level and a single OR gate the second level.



$(X+1)(Y+0Z)$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



Power of logic expressions

A valid question is: *can logic expressions describe all possible combinatorial circuits?*. The answer is *yes* and here is why:

You can trivially convert the truth table for an arbitrary circuit into an expression. The expression will be in the form of a sum of products of variables and there inverses. Each row with output value of 1 of the truth table corresponds to one term in the sum. In such a term, a variable having a 1 in the truth table will be uninverted, and a variable having a 0 in the truth table will be inverted.

Take the following truth table for example:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding expression is:

$X'Y'Z+XY'Z'+XYZ$

Since you can describe any combinatorial circuit with a truth table, and you can describe any truth table with an expression, you can describe any combinatorial circuit with an expression.

Simplicity of logic expressions

There are many logic expressions (and therefore many circuits) that correspond to a certain truth table, and therefore to a certain function computed. For instance, the following two expressions compute the same function:

$X(Y+Z)$ $XY+XZ$

The left one requires two gates, one *and*-gate and one *or*-gate. The second expression requires two *and*-gates and one *or*-gate. It seems obvious that the first one is preferable to the second one. However, this is not always the case. It is not always true that the number of gates is the only way, nor even the best way, to determine simplicity.

We have, for instance, assumed that gates are ideal. In reality, the signal takes some time to propagate through a gate. We call this time the gate *delay*. We might be interested in circuits that minimize the total gate delay, in other words, circuits that make the signal traverse the fewest possible gates from input to output. Such circuits are not necessarily the same ones that require the smallest number of gates.

Circuit minimization

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, it can appear in many different forms when expressed algebraically.

Simplification through algebraic manipulation

A Boolean equation can be reduced to a minimal number of literal by algebraic manipulation as stated above. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only methods is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar

e.g. simplify $x+x'y$

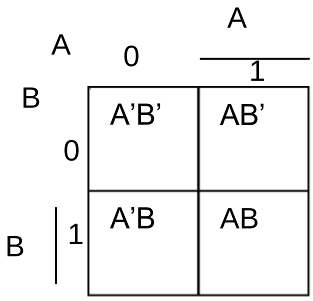
$x+x'y=(x+x')(x+y)=x+y$
simplify $x'y'z+x'yz+xy'$
$x'y'z+x'yz+xy'=x'z(y+y')+xy'$
$=x'z+xy'$

Simplify xy +x′z+yz
xy +x′z+yz= xy +x′z+yz(x+x′)
xy +x′z+yzx+yzx′
xy(1+z) +x′z(1+y)
=xy+x′z

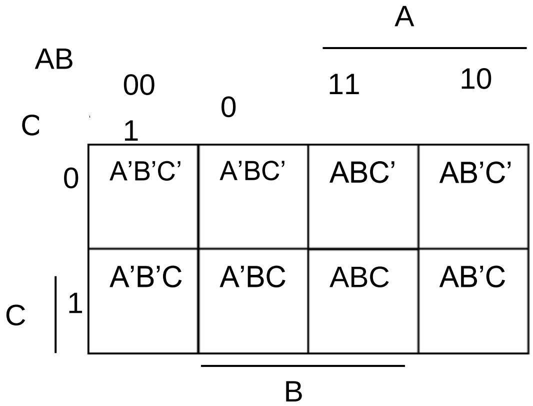
Karnaugh map

The Karnaugh map also known as Veitch diagram or simply as K map is a two dimensional form of the truth table, drawn in such a way that the simplification of Boolean expression can be immediately be seen from the location of 1’s in the map. The map is a diagram made up of squares , each sqare represent one minterm. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognised graphically in the map from the area enclosed by those squares whose minterms are included in the function.

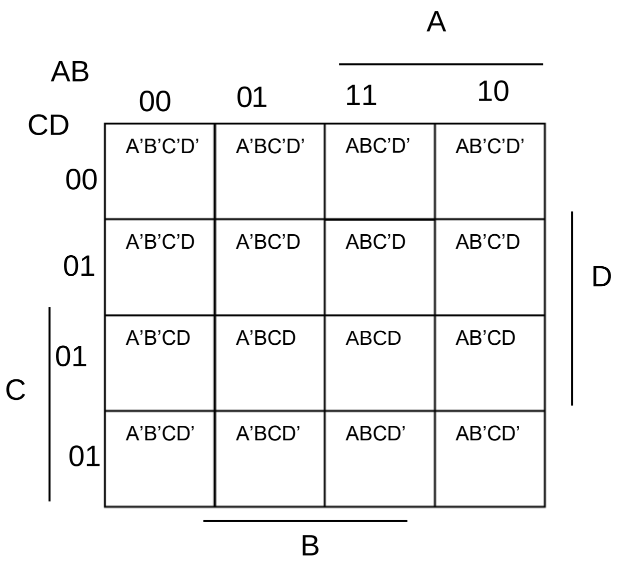
A two variable Boolean function can be represented as follow



A three variable



A four variable Boolean function can be represented in the map bellow



CD \ AB	00	01	11	10
00	A’B’C’D’	A’BC’D’	ABC’D’	AB’C’D’
01	A’B’C’D	A’BC’D	ABC’D	AB’C’D
01	A’B’CD	A’BCD	ABCD	AB’CD
01	A’B’CD’	A’BCD’	ABCD’	AB’CD’

To simplify a Boolean function using karnaugh map, the first step is to plot all ones in the function truth table on the map. The next step is to combine adjacent 1’s into a group of one, two, four, eight, sixteen. The group of minterm should be as large as possible. A single group of four minterm yields a simpler expression than two groups of two minterms.

In a four variable karnaugh map,

1 variable product term is obtained if 8 adjacent squares are covered
2 variable product term is obtained if 4 adjacent squares are covered
3 variable product term is obtained if 2 adjacent squares are covered
1 variable product term is obtained if 1 square is covered
A square having a 1 may belong to more than one term in the sum of product expression

The final stage is reached when each of the group of minterms are ORded together to form the simplified sum of product expression

The karnaugh map is not a square or rectangle as it may appear in the diagram. The top edge is adjacent to the bottom edge and the left hand edge adjacent to the right hand edge. Consequent, two squares in karnaugh map are said to be adjacent if they differ by only one variable

Implicant

In Boolean logic, an implicant is a "covering" (sum term or product term) of one or more minterms in a sum of products (or maxterms in a product of sums) of a boolean function. Formally, a product term P in a sum of products is an implicant of the Boolean function F if P implies F. More precisely:

> *P* implies *F* (and thus is an implicant of *F*) if *F* also takes the value 1 whenever *P* equals 1.

where

- *F* is a Boolean of *n* variables.
- *P* is a product term

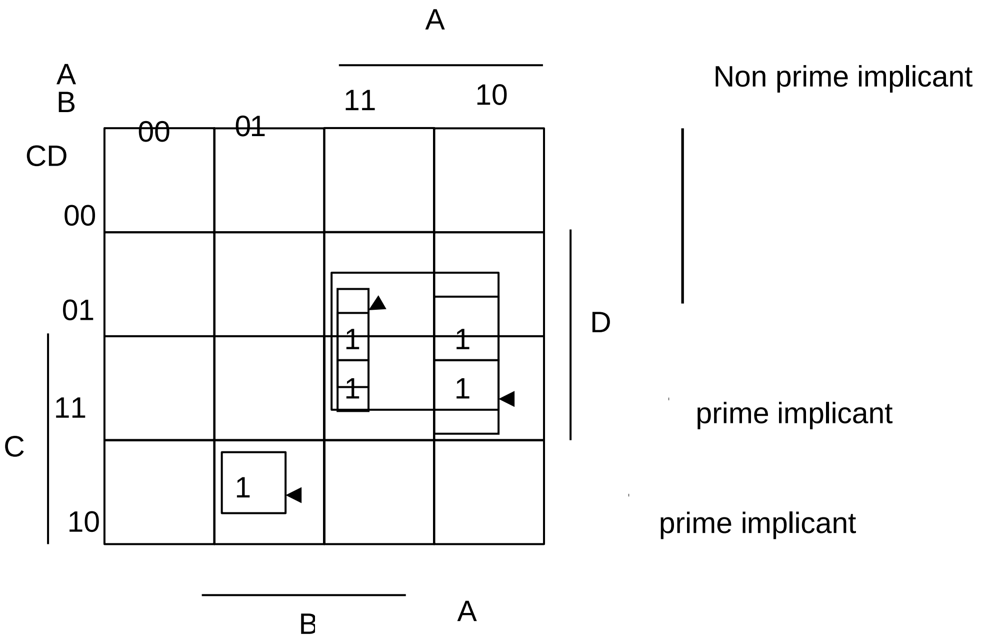
This means that P < = F with respect to the natural ordering of the Boolean space. For instance, the function

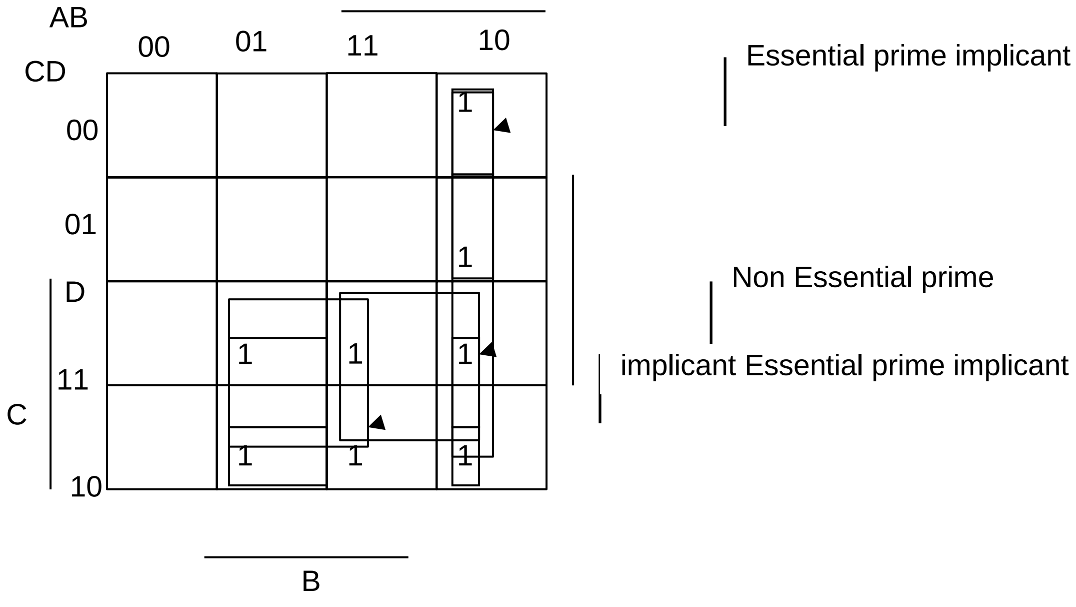
$$f(x,y,z,w) = xy + yz + w$$

is implied by *xy*, by *xyz*, by *xyzw*, by *w* and many others; these are the implicants of *f*.

Prime implicant

A prime implicant of a function is an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant. W.V. Quine defined a prime implicant of F to be an implicant that is minimal - that is, if the removal of any literal from P results in a non-implicant for F. Essential prime implicants are prime implicants that cover an output of the function that no combination of other prime implicants is able to cover.




A
AB
CD
00
01
11
10
00
01
11
10
1
1
1
1
1
1
1
1
1
D
C
B
Essential prime implicant
Non Essential prime
implicant Essential prime implicant

In simplifying a Boolean function using karnaugh map, non essential prime implicant are not needed

Minimization of Boolean expressions using Karnaugh maps.

Given the following truth table for the majority function.

a	b	C	M(output)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The Boolean algebraic expression is

$m = a'bc + ab'c + abc' + abc.$

the minimization using algebraic manipulation can be done as follows.

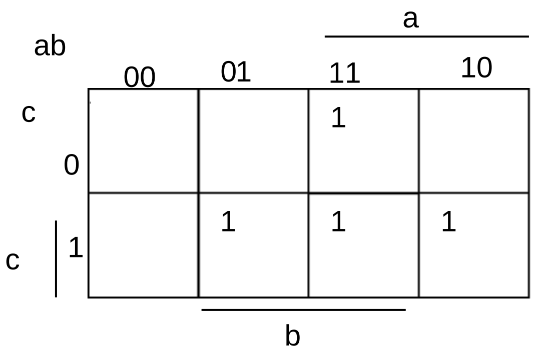
$m = a'bc + abc + ab'c + abc + abc' + abc$

$= (a' + a)bc + a(b' + b)c + ab(c' + c)$

$= bc + ac + ab$

The **abc** term was replicated and combined with the other terms.

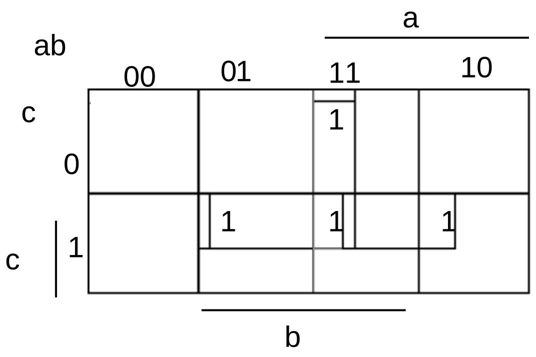
To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 101.



c \ ab	00	01	11	10
0			1	
1		1	1	1

The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 110 (**a** = 1, **b** = 1, **c** = 0).

The minimization is done by drawing circles around sets of adjacent 1s. Adjacency is horizontal, vertical, or both. The circles must always contain 2^n 1s where n is an integer.



We have circled two 1s. The fact that the circle spans the two possible values of **a**

(0 and 1) means that the **a** term is eliminated from the Boolean expression corresponding to this circle.

Now we have drawn circles around all the 1s. Thus the expression reduces to

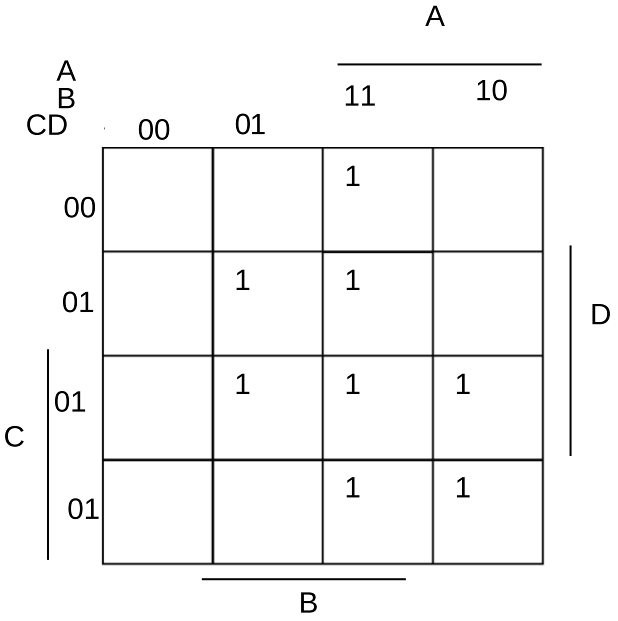
bc + ac + ab

as we saw before.

What is happening? What does adjacency and grouping the 1s together have to do with minimization? Notice that the 1 at position 111 was used by all 3 circles. This 1 corresponds to the abc term that was replicated in the original algebraic minimization. Adjacency of 2 1s means that the terms corresponding to those 1s differ in one variable only. In one case that variable is negated and in the other it is not.

The map is easier than algebraic minimization because we just have to recognize patterns of 1s in the map instead of using the algebraic manipulations. Adjacency also applies to the edges of the map.

Now for 4 Boolean variables. The Karnaugh map is drawn as shown below.

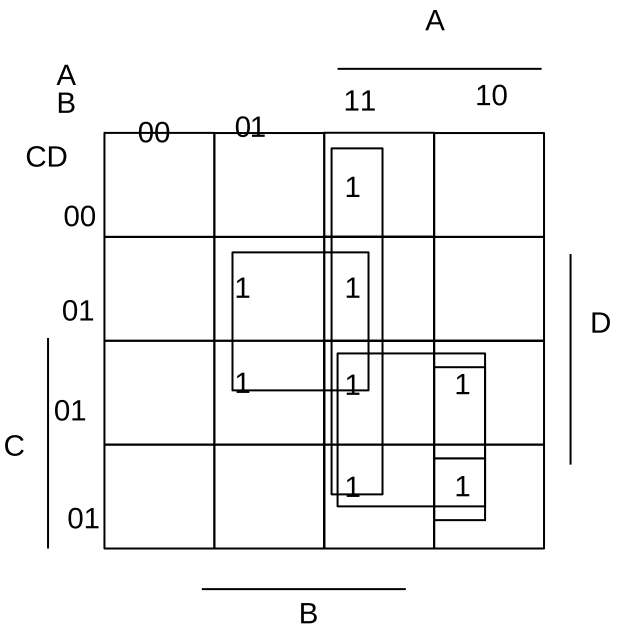


The following corresponds to the Boolean expression

Q = A′BC′D + A′BCD + ABC′D′ + ABC′D + ABCD + ABCD′ + AB′CD + AB′CD′

RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s.

Grouping the 1s together results in the following.



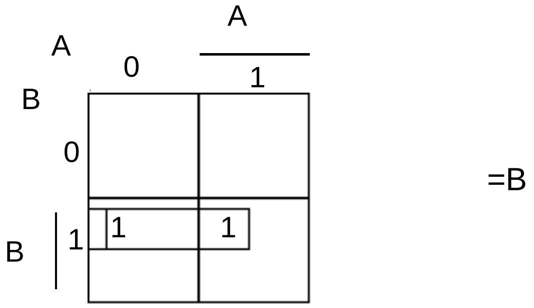
The expression for the groupings above is

Q = BD + AC + AB

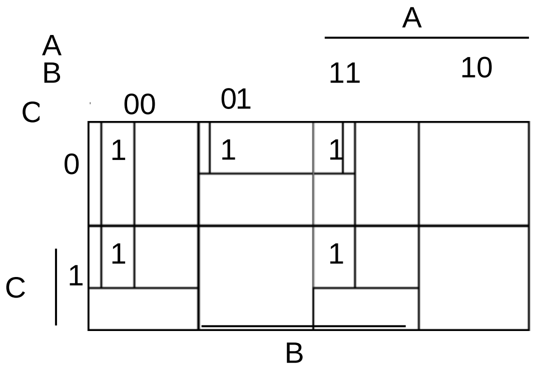
This expression requires 3 2-input **AND** gates and 1 3-input **OR** gate.

Other examples

1. **F=A′B+AB**

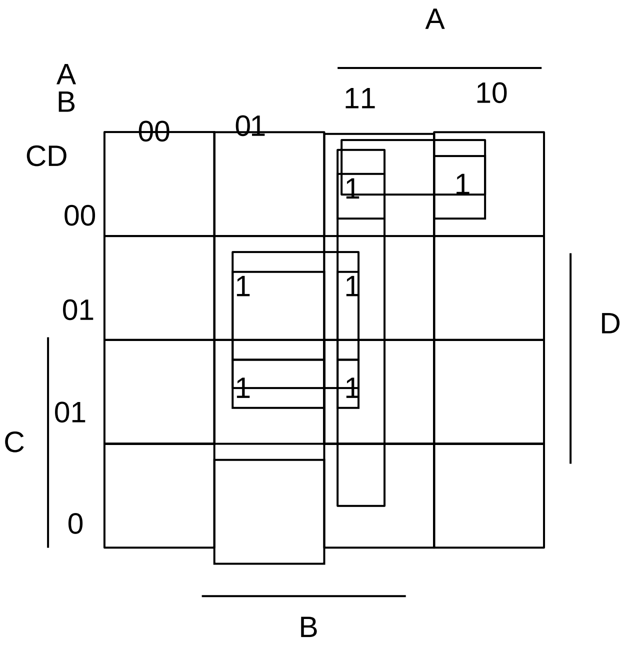


2. F=A′B′C′+A′B′C+A′BC′+ABC′+ABC



=A’B’+BC’+AB

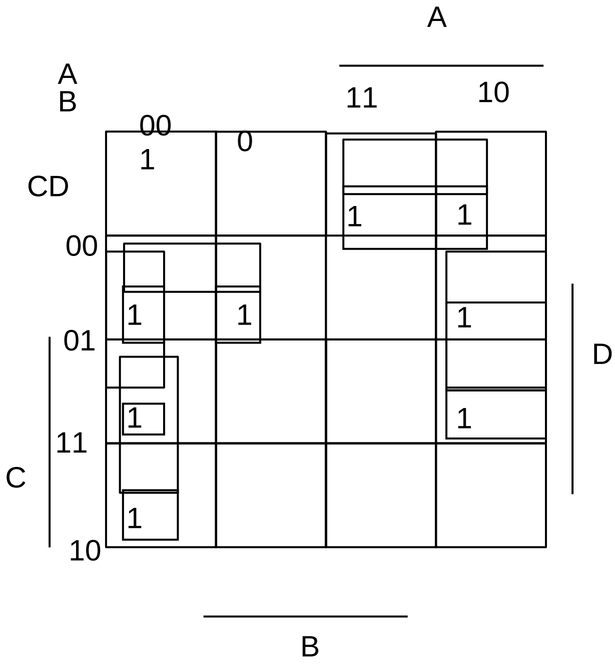
3. F=AB+A′BC′D+A′BCD+AB′C′D′



=BD+AB+AC’D’

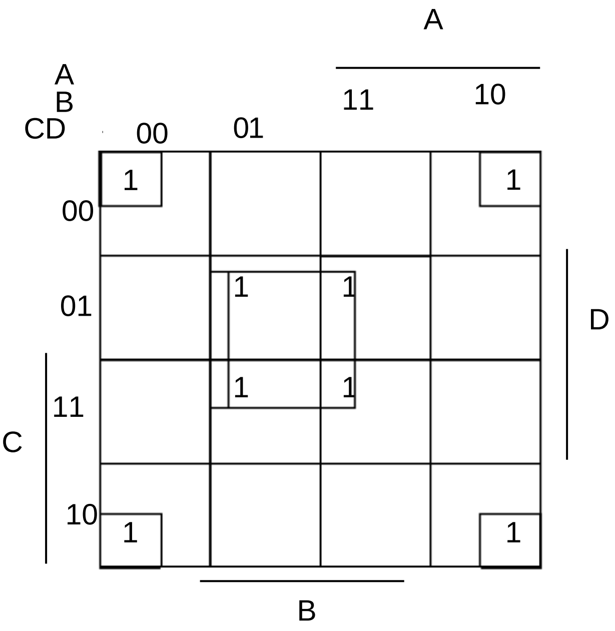
1

4. F=AC′D′+A′B′C+A′C′D+AB′D



=B’D+AC’D’+A’C’D+A’B’C

5. F=A′B′C′D′+AB′C′D′+A′BC′D+ABC′D+A′BCD+ABCD

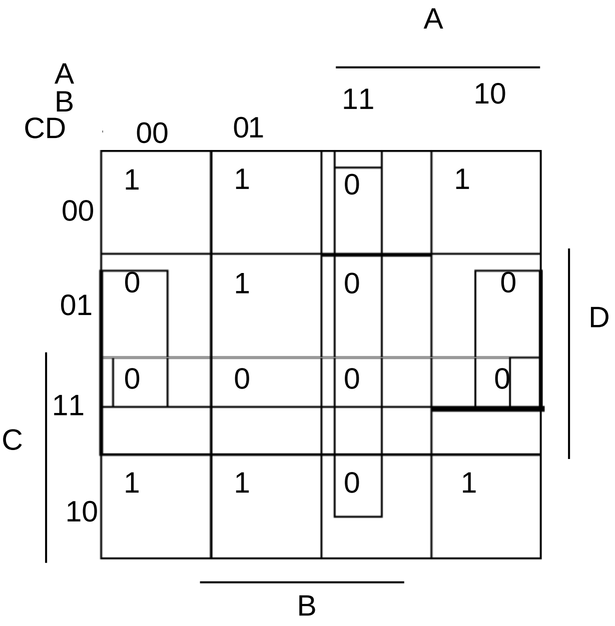


=BD+D’B’

Obtaining a Simplified product of sum using Karnaugh map

The simplification of the product of sum follows the same rule as the product of sum. However, adjacent cells to be combined are the cells containing 0. In this approach, the obtained simplified function is F′. since F is represented by the square marked with 1. The function F can be obtained in product of sum by applying de morgan’s rule on F′.

F=A′B′C′D′+A′BC′D′+AB′C′D′+A′BC′D+A′B′CD′+A′BCD′+AB′CD′



The obtained simplified F′=AB+CD+BD′. Since F″=F, By applying de morgan’s rule to F′, we obtain

F″=(AB+CD+BD′)′

=(A′+B′)(C′+D′)(B′+D) which is he simplified F in product of sum.

Don't Care condition

Sometimes we do not care whether a 1 or 0 occurs for a certain set of inputs. It may be that those inputs will never occur so it makes no difference what the output is. For example, we might have a BCD (binary coded decimal) code which consists of 4 bits to encode the digits 0 (0000) through 9 (1001). The remaining codes (1010 through 1111) are not used. If we had a truth table for the prime numbers 0 through 9, it would be

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$F=A'B'CD'+A'B'CD+A'BC'D+A'BCD$

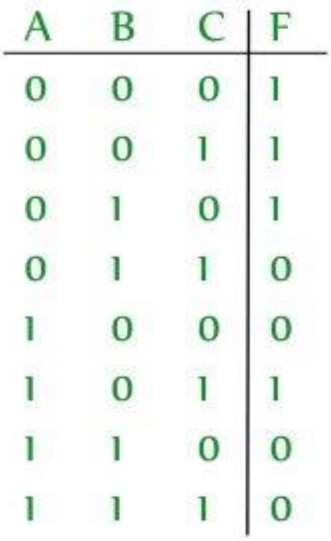
The X in the above stand for "don’t care", we don't care whether a 1 or 0 is the value for that combination of inputs because (in this case) the inputs will never occur.

Variable Entered Map

K-map is the best manual technique to solve Boolean equations, but it becomes difficult to manage when number of variables exceed 5 or 6. So, a technique called Variable Entrant Map (VEM) is used to increase the effective size of k-map. It allows a smaller map to handle large number of variables. This is done by writing output in terms of input.

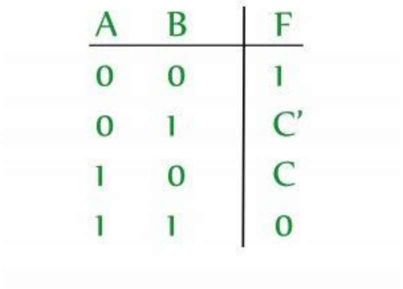
Example – A 3-variable function can be defined as a function of 2-variables if the output is written in terms of third variable.

Consider a function **F(A,B,C)** = **(0,1,2,5)**



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

If we define F in terms of ‘C’, then this function can be written as:



A	B	F
0	0	1
0	1	C’
1	0	C
1	1	0

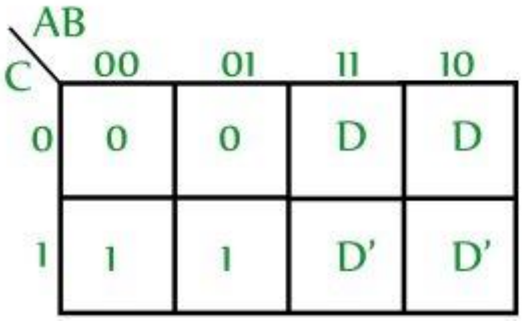
Advantages of using VEM –

- A VEM can be used to plot more than ‘n’ variables using an ‘n’ variable K-map.
- It is commonly used to solve problems involving multiplexers.

Minimization procedure for VEM – Now, let’s see how to find SOP expression if a VEM is given.

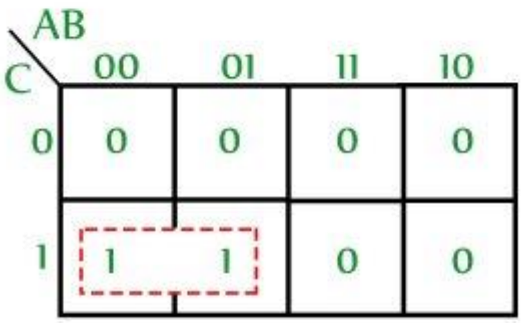
1. Write all the variables(original and complimented forms are treated as two different variables) in the map as 0, leave 0’s, minterms and don’t cares as it is and obtain the SOP expression.
2. (a) Select one variable and make all occurrences of that variable as 1, write minterms (1’s) as don’t cares, leave 0’s and don’t cares as it is. Now, obtain the SOP expression.
   (b) Multiply the obtained SOP expression with the concerned variable.
3. Repeat step 2 for all the variables in the k-map.
4. SOP of VEM is obtained by ORing all the obtained SOP expressions.

Let’s apply the above procedure on a sample VEM (X is used to represent don’t care):



C \ AB	00	01	11	10
0	0	0	D	D
1	1	1	D’	D’

Step 1: Write all the variables as 0 (D and D’ are considered as two different variables), leave minterms, 0’s and don’t cares as it is and obtain the SOP expression.

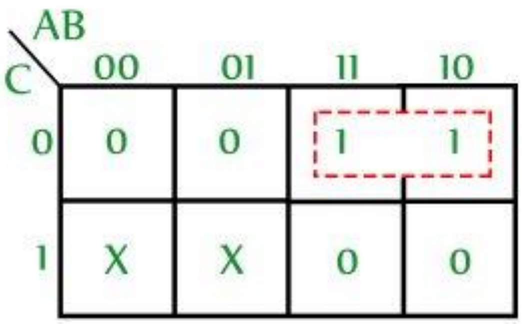


C \ AB	00	01	11	10
0	0	0	0	0
1	1	1	0	0

SOP expression: A’C

```
SOP obtained: A'C
```

Step 2:
(a) Replace all occurances of D with 1, all occurrences of D’ with 0 and all 1’s with don’t care. Leave 0’s and don’t cares as it is.



C \ AB	00	01	11	10
0	0	0	1	1
1	X	X	0	0

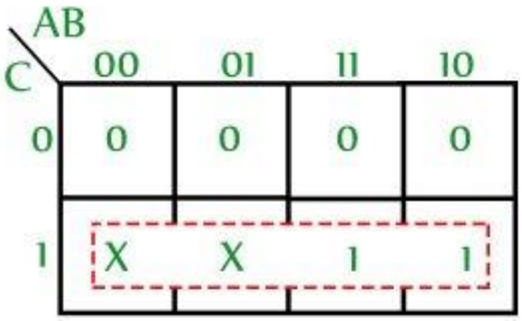
SOP expression: AC’

(b) Multiply the obtained SOP with the concerned variable.

```
SOP obtained: AC'D
```

Step 3: Repeat step 2 for D’

(a) Replace all occurrences of D’ with 1, all occurrences of D with 0 and all 1’s with don’t care. Leave 0’s and don’t cares as it is.



C \ AB	00	01	11	10
0	0	0	0	0
1	X	X	1	1

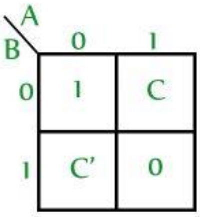
SOP expression: C

(b) Multiply the obtained SOP with the concerned variable.

```
SOP obtained: CD'
```

Step 4: SOP of VEM is obtained by **ORing** all the obtained SOP expressions. Therefore, the SOP expression for the given VEM is:

```
A'C + AC'D + CD'
```



B \ A	0	1
0	1	C
1	C’	0

Designing Combinatorial Circuits

The design of a combinational circuit starts from the verbal outline of the problem and ends with a logic circuit diagram or a set of Boolean functions from which the Boolean function can be easily obtained. The procedure involves the following steps:

- The problem is stated
- The number of available input variables and required output variables is determined.
- The input and output variable are assigned their letter symbol
- The truth table that defines the required relationship between the inputs and the outputs is derived.
- The simplified Boolean function for each output is obtained
- The logic diagram is drawn.

Example of combinational circuit

Adders

In electronics, an adder or summer is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where twos complement or ones complement is being used to represent negative numbers, it is trivial to modify an adder into an adder-subtracter. Other signed number representations require a more complex adder.

-Half Adder

A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

A half adder has two inputs, generally labelled A and B, and two outputs, the sum S and carry C. S is the two-bit XOR of A and B, and C is the AND of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs.

The drawback of this circuit is that in case of a multibit addition, it cannot include a carry.

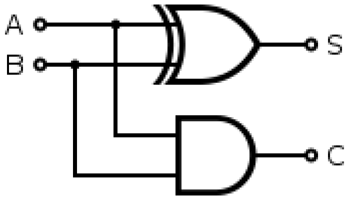
Following is the truth table for a half adder:

A	***B***	***Carry***	***Sum***
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Equation of the Sum and Carry.

$\text{Sum}=A'B+AB'$ $\text{Carry}=AB$

One can see that Sum can also be implemented using XOR gate as $A \oplus B$



-Full Adder.

A full adder has three inputs *A*, *B*, and a carry in *C*, such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled C_i or C_{in} while the carry out is labelled C_o or C_{out}.

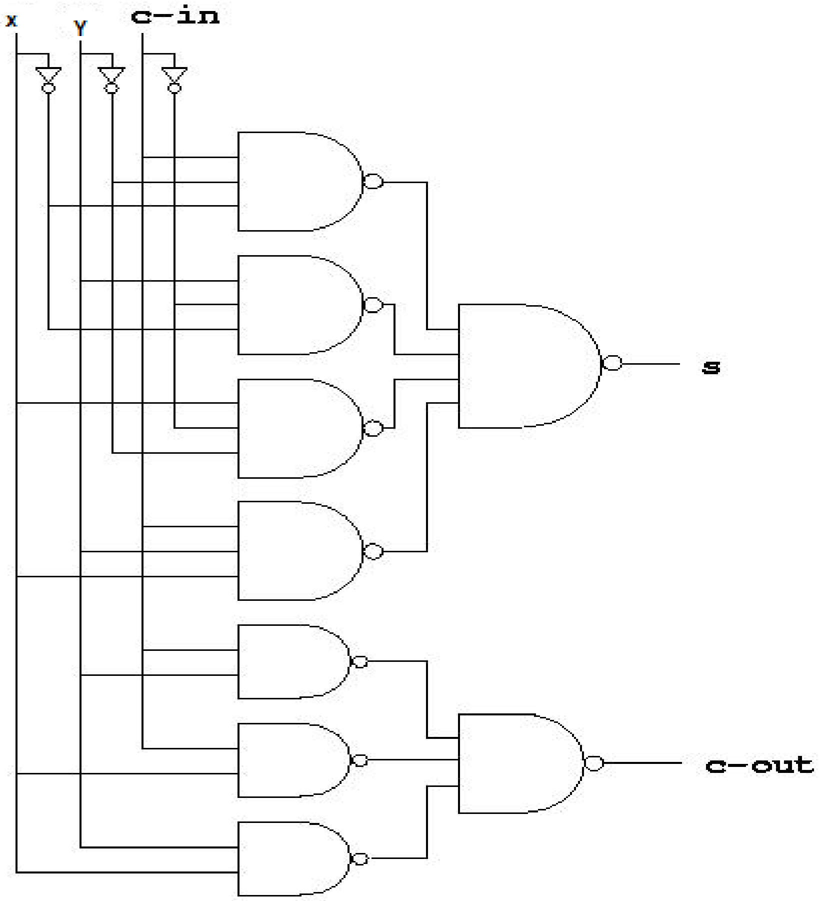
A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits. It can be combined with other full adders or work on its own.

Input			**Output**	
A	**B**	***C_i***	***C_o***	***S***
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$C_o=A'BC_i+AB'C_i+ABC_i'+ABC_i$

$S=A'B'C_i +A'BC_i'+ABC_i'+ABC_i$

A full adder can be trivially built using our ordinary design methods for combinatorial circuits. Here is the resulting



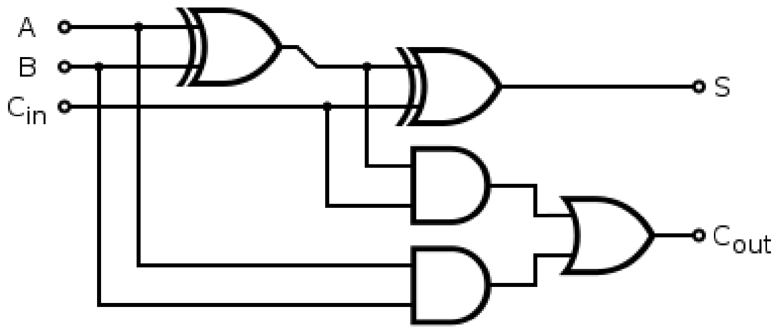
circuit diagram using NAND gates only:

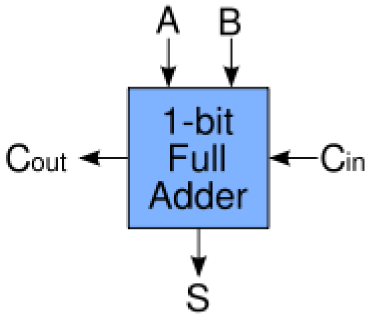
$C_o = A'BC_i + AB'C_i + ABC_i' + ABC_i$ by manipulating C_o, we can see that$C_o = C_i(A \oplus B) + AB$

$S = A'B'C_i + A'BC_i' + ABC_i' + ABC_i$ By manipulating S, we can see that $S = C_i \oplus (A \oplus B)$

Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and XOR gates occurs when both inputs are 1; for the adder shown here, this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips.

A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, connecting the sum from that to an input to the second adder, connecting Ci to the other input and OR the two carry outputs. Equivalently, S could be made the three-bit xor of A, B, and Ci and Co could be made the three-bit majority function of A, B, and Ci. The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.





Subtractor

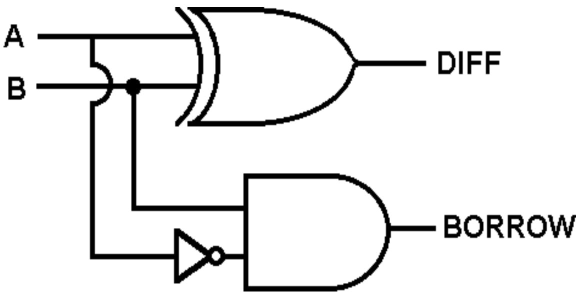
In electronics, a subtractor can be designed using the same approach as that of an adder. The binary subtraction process is summarized below. As with an adder, in the general case of calculations on multi-bit numbers, three bits are involved in performing the subtraction for each bit: the minuend (Xi), subtrahend (Yi), and a borrow in from the previous (less significant) bit order position (Bi). The outputs are the difference bit (Di) and borrow bit Bi + 1.

Half subtractor

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow). Such a circuit is called a half-subtractor because it enables a borrow out of the current arithmetic operation but no borrow in from a previous arithmetic operation.

The truth table for the half subtractor is given below.

X	**Y**	**D**	**B**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



$D=X'Y+XY'$ or $D= X\oplus Y$

$B=X'Y$

Full Subtractor

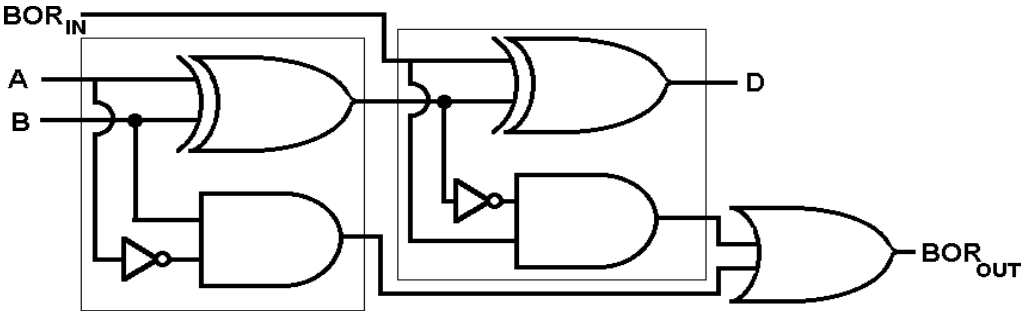
As in the case of the addition using logic gates , a *full subtractor* is made by combining two half-subtractors and an additional OR-gate. A full subtractor has the borrow in capability (denoted as **BOR$_{IN}$** in the diagram below) and so allows *cascading* which results in the possibility of **multi-bit subtraction**.

The final truth table for a full subtractor looks like:

A	B	BOR$_{IN}$	D	BOR$_{OUT}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Find out the equations of the borrow and the difference

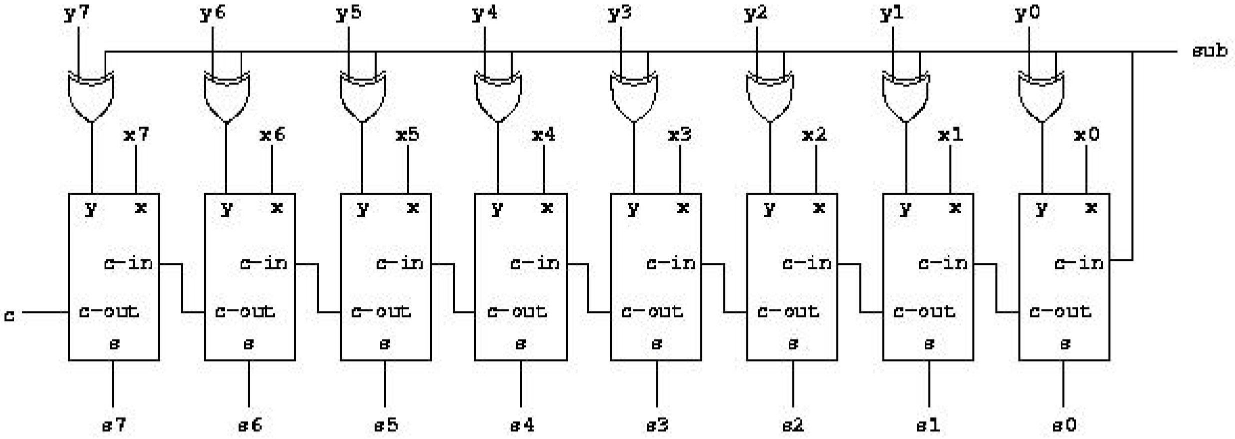
The circuit diagram for a full subtractor is given below.



For a wide range of operations many circuit elements will be required. A neater solution will be to use subtraction via addition using *complementing* as was discussed in the binary arithmetic topic. In this case only adders are needed as shown bellow.

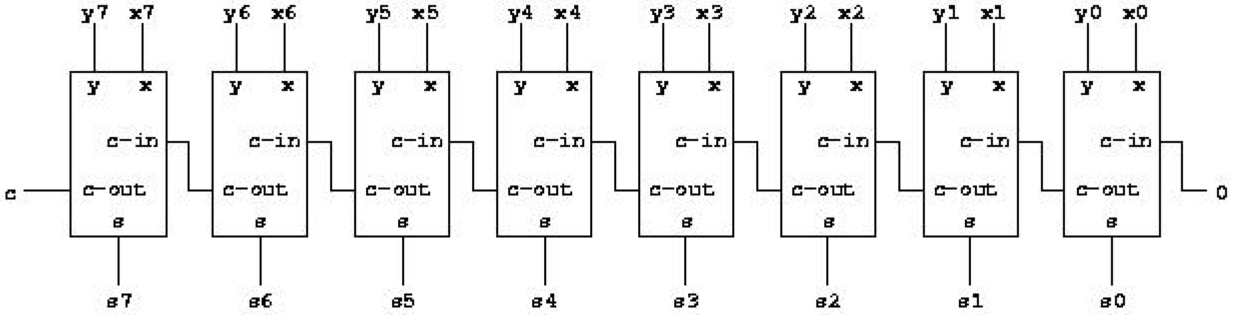
Binary subtraction using adders

Our binary adder can already handle negative numbers as indicated in the section on binary arithmetic But we have not discussed how we can get it to handle subtraction. To see how this can be done, notice that in order to compute the expression *x* - *y*, we can compute the expression *x* + -*y* instead. We know from the section on binary arithmetic how to negate a number by inverting all the bits and adding 1. Thus, we can compute the expression as *x* + *inv(y)* + *1*. It suffices to invert all the inputs of the second operand before they reach the adder, but how do we add the 1. That seems to require another adder just for that. Luckily, we have an unused carry-in signal to position 0 that we can use. Giving a 1 on this input in effect adds one to the result. The complete circuit with addition and subtraction looks like this:



Ripple carry adder

It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a *ripple carry adder*, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder.



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Following the path from C_{in} to C_{out} shows 2 gates that must be passed through. Therefore, a 32-bit adder requires 31 carry computations and the final sum calculation for a total of 31 * 2 + 1 = 63 gate delays.

Single decade BCD adder

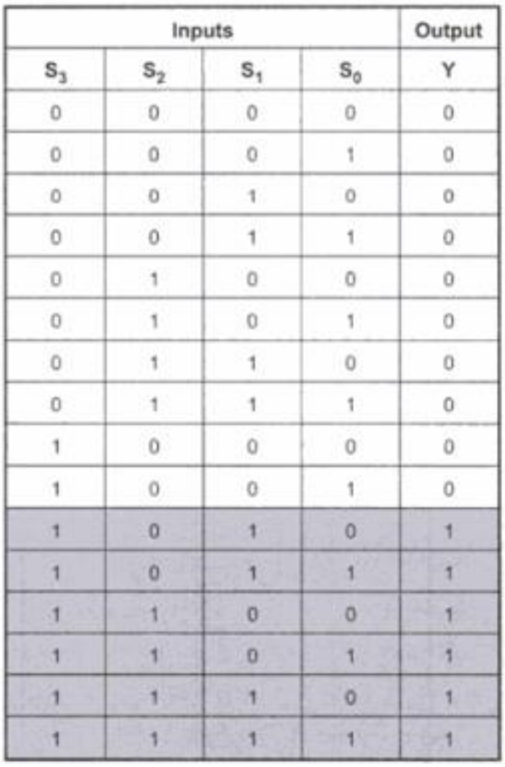
1. The digital systems handle the decimal number in the form of binary coded decimal numbers (BCD).
2. A BCD adder is a circuit that adds two BCD digits and produces a sum digit also in BCD.
3. To implement BCD adder we require:

   • 4-bit binary adder for initial addition

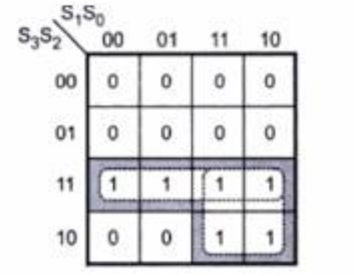
   • Logic circuit to detect sum greater than 9

   • One more 4-bit adder to add $0110_2$01102 in the sum if sum is greater than 9 or carry is 1

4. The logic circuit to detect sum greater than 9 can be determined by simplifying the Boolean expression of given truth Table.



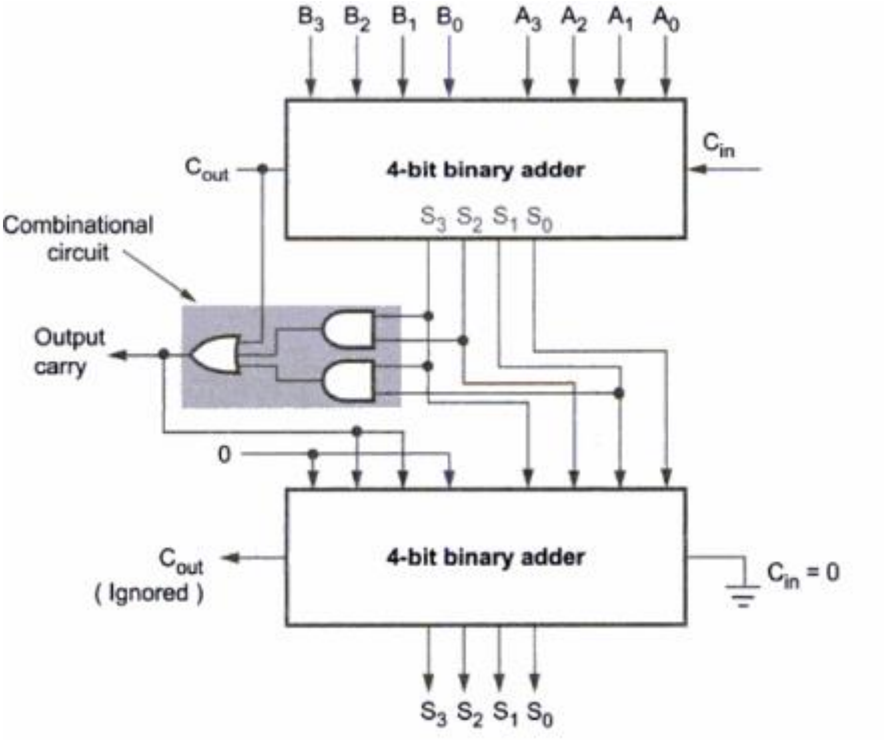
Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



S_3S_2 \ S_1S_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$Y = S_3S_2 + S_3S_1$

1. Y=1 indicates sum is greater than 9. We can put one more term, C_out in the above expression to check whether carry is one.
2. If any one condition is satisfied we add 6(0110) in the sum.
3. With this design information we can draw the block diagram of BCD adder, as shown in figure below.



1. As shown in the Fig, the two BCD numbers, together with input carry, are first added in the top 4-bit binary adder to produce a binary sum.
2. When the output carry is equal to zero (i.e. when sum <= 9 and C_out = 0) nothing (zero) is added to the binary sum.

   10.When it is equal to one (i.e. when sum > 9 or C_out = 1), binary 0110 is added to the binary sum through the bottom 4-bit binary adder.
3. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output carry terminal.

Carry Look-ahead Adder

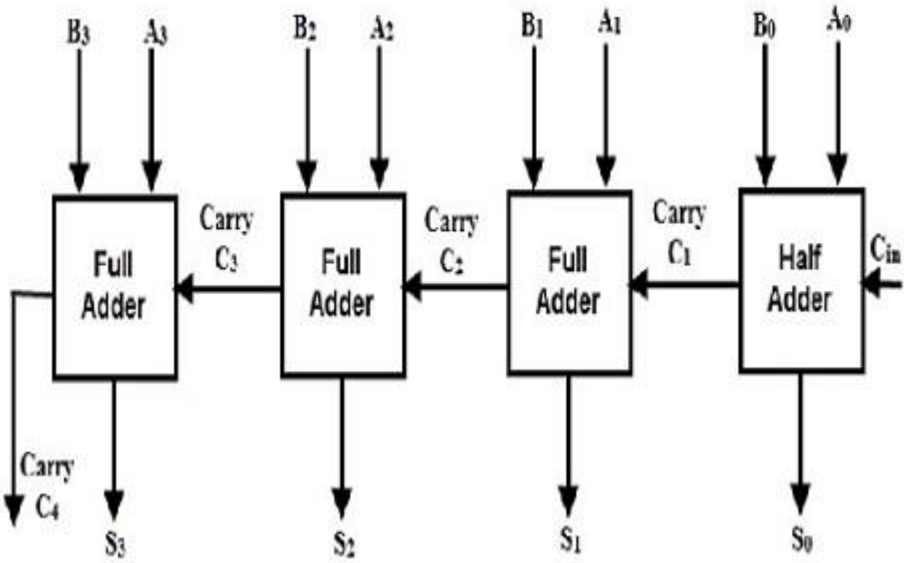
A digital computer must contain circuits which can perform arithmetic operations such as addition, subtraction, multiplication, and division. Among these, addition and subtraction are the basic operations whereas multiplication and division are the repeated addition and subtraction respectively.

To perform these operations 'Adder circuits' are implemented using basic logic gates. Adder circuits are evolved as Half-adder, Full-adder, Ripple-carry Adder, and Carry Look-ahead Adder.
Among these Carry Look-ahead Adder is the faster adder circuit. It reduces the propagation delay, which occurs during addition, by using more complex hardware circuitry. It is designed by transforming the ripple-carry Adder circuit such that the carry logic of the adder is changed into two-level logic.

4-Bit Carry Look-ahead Adder

In parallel adders, carry output of each full adder is given as a carry input to the next higher-order state. Hence, these adders it is not possible to produce carry and sum outputs of any state unless a carry input is available for that state.

So, for computation to occur, the circuit has to wait until the carry bit propagated to all states. This induces carry propagation delay in the circuit.

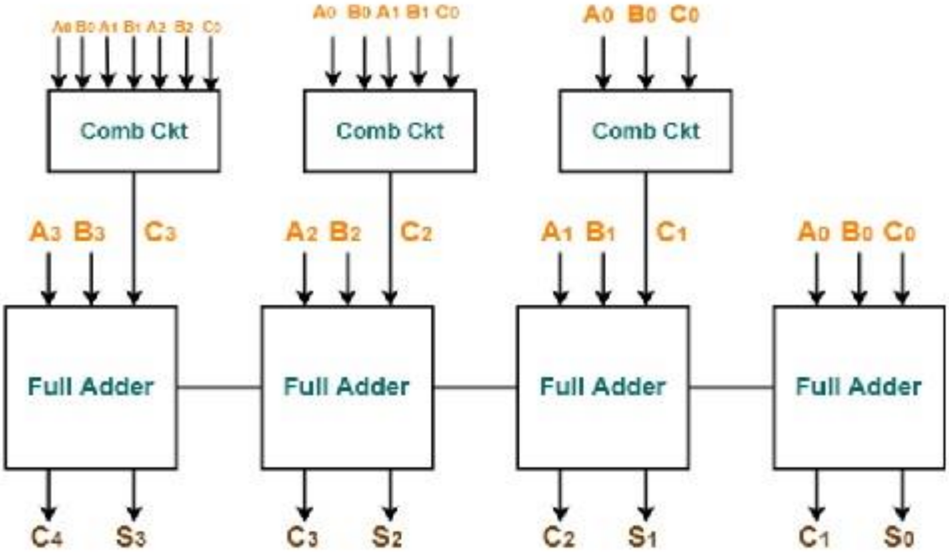


4-bit-Ripple-Carry-Adder

Consider the 4-bit ripple carry adder circuit above. Here the sum S3 can be produced as soon as the inputs A3 and B3 are given. But carry C3 cannot be computed until the carry bit C2 is applied whereas C2 depends on C1. Therefore to produce final steady-state results, carry must propagate through all the states. This increases the carry propagation delay of the circuit.

The propagation delay of the adder is calculated as “the propagation delay of each gate times the number of stages in the circuit”. For the computation of a large number of bits, more stages have to be added, which makes the delay much worse. Hence, to solve this situation, Carry Look-ahead Adder was introduced.

To understand the functioning of a Carry Look-ahead Adder, a 4-bit Carry Look-ahead Adder is described below.

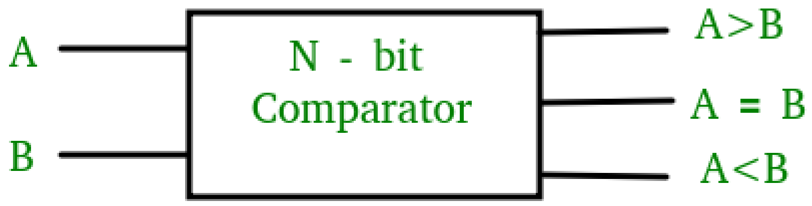


4-bit-Carry-Look-ahead-Adder-Logic-Diagram

In this adder, the carry input at any stage of the adder is independent of the carry bits generated at the independent stages. Here the output of any stage is dependent only on the bits which are added in the previous stages and the carry input provided at the beginning stage. Hence, the circuit at any stage does not have to wait for the generation of carry-bit from the previous stage and carry bit can be evaluated at any instant of time.

Magnitude Comparator in Digital Logic

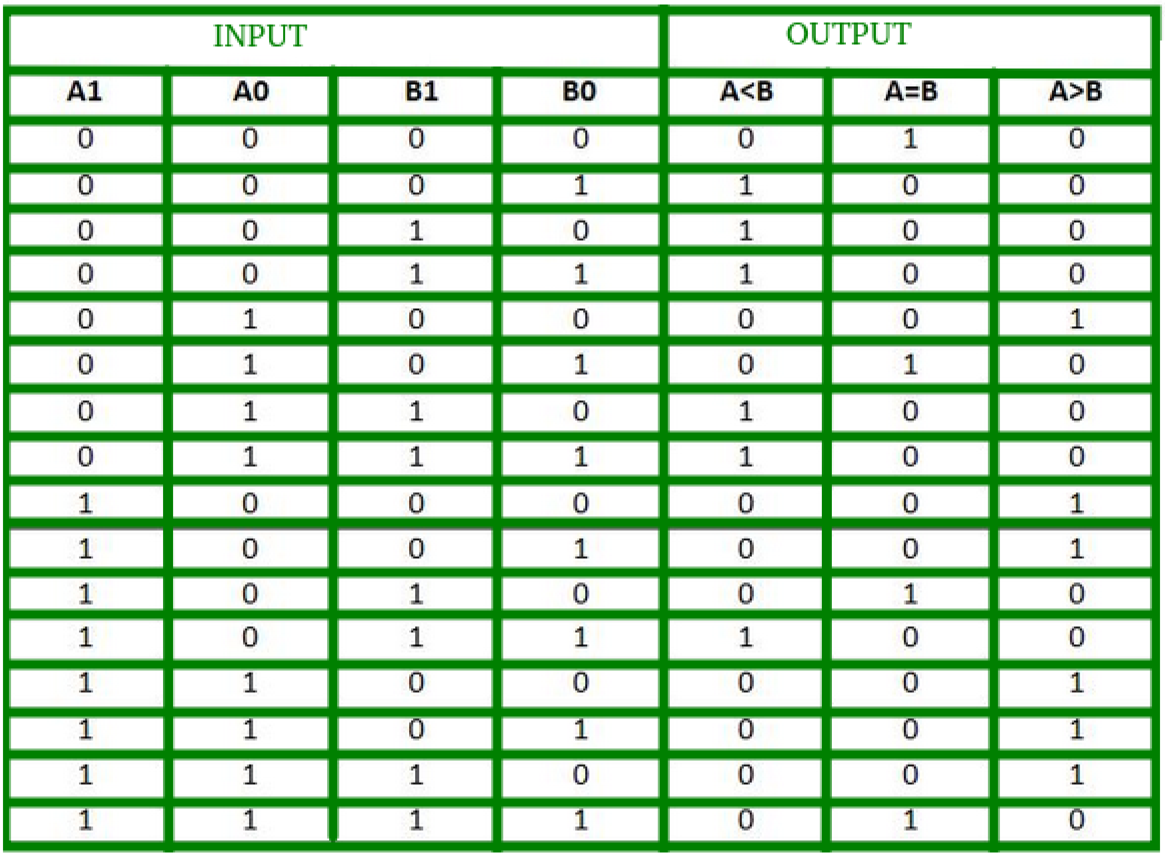
A magnitude digital Comparator is a combinational circuit that **compares two digital or binary numbers** in order to find out whether one binary number is equal, less than or greater than the other binary number. We logically design a circuit for which we will have two inputs one for A and other for B and have three output terminals, one for A > B condition, one for A = B condition and one for A < B condition.



2-Bit Magnitude Comparator –

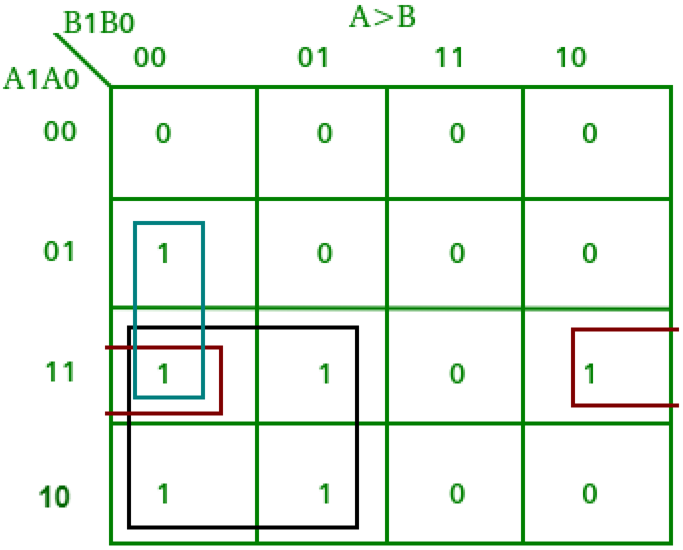
A comparator used to compare two binary numbers each of two bits is called a 2-bit Magnitude comparator. It consists of four inputs and three outputs to generate less than, equal to and greater than between two binary numbers.

The truth table for a 2-bit comparator is given below:



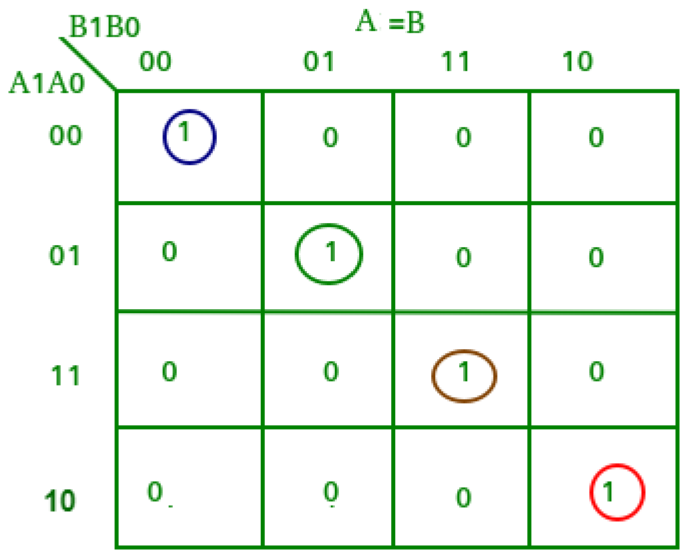
INPUT				OUTPUT		
A1	A0	B1	B0	A<B	A=B	A>B
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

From the above truth table K-map for each output can be drawn as follows:



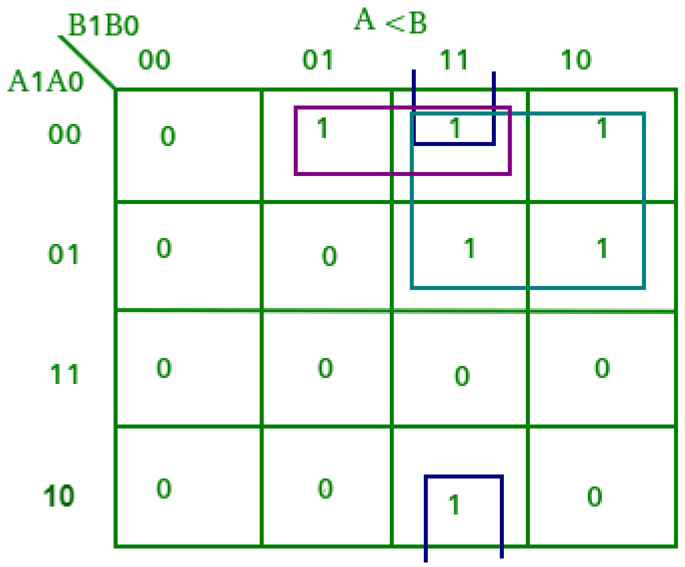
A>B

A1A0 \ B1B0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0



A =B

A1A0 \ B1B0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1



A <B

A1A0 \ B1B0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

From the above K-maps logical expressions for each output can be expressed as follows:

A>B:A1B1’ + A0B1’B0’ + A1A0B0’

A=B: A1’A0’B1’B0’ + A1’A0B1’B0 + A1A0B1B0 + A1A0’B1B0’

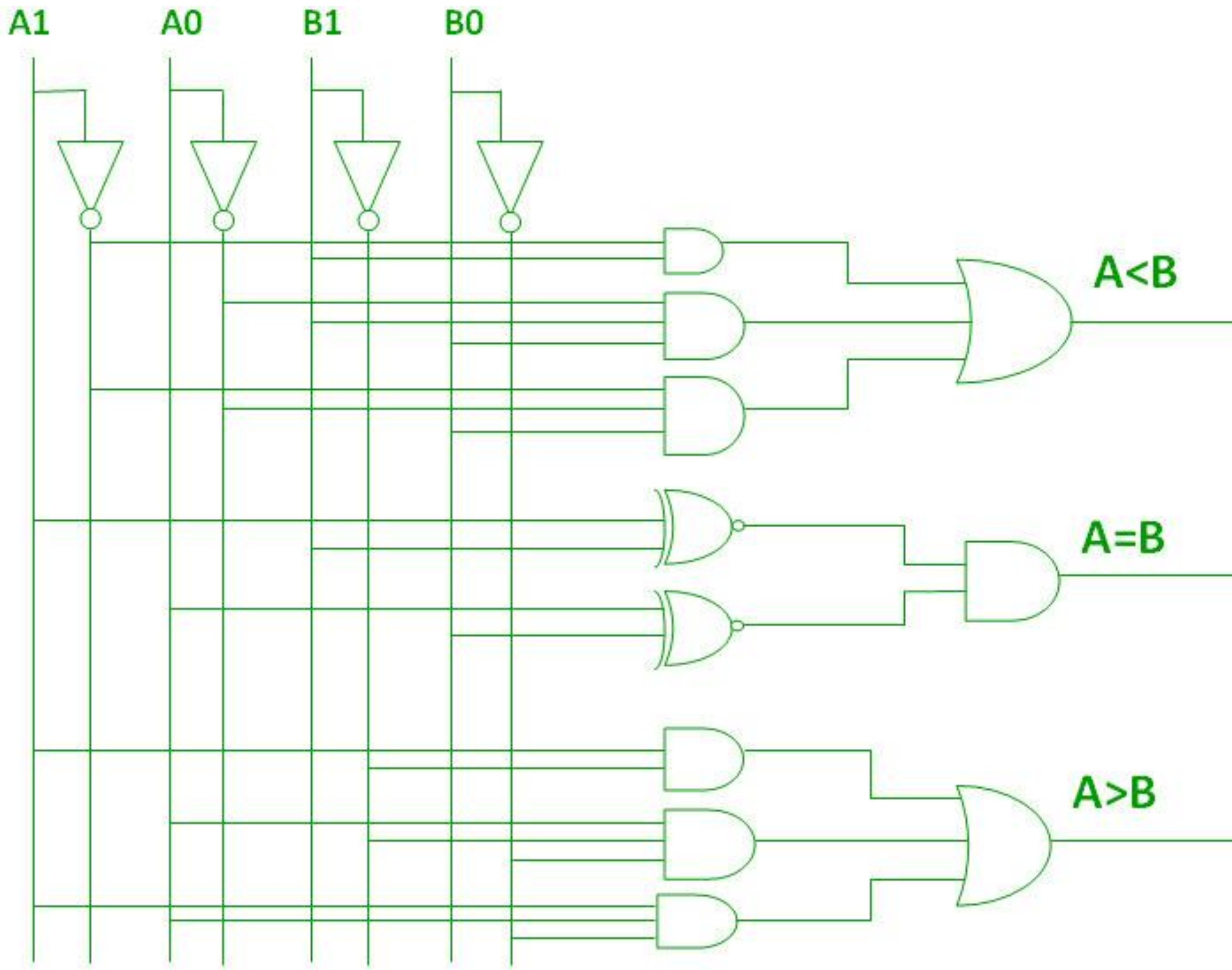
: A1’B1’ (A0’B0’ + A0B0) + A1B1 (A0B0 + A0’B0’)

: (A0B0 + A0’B0’) (A1B1 + A1’B1’)

: (A0 Ex-Nor B0) (A1 Ex-Nor B1)

A<B:A1’B1 + A0’B1B0 + A1’A0’B0

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below:



Applications of Comparators –

1. Comparators are used in central processing units (CPUs) and microcontrollers (MCUs).
2. These are used in control applications in which the binary numbers representing physical variables such as temperature, position, etc. are compared with a reference value.
3. Comparators are also used as process controllers and for Servo motor control.
4. Used in password verification and biometric applications.

Medium Scale integration component

The purpose of circuit minimization is to obtain an algebraic expression that, when implemented results in a low cost circuit. Digital circuit are constructed with integrated circuit(IC). An IC is a small silicon semiconductor crystal called chip containing the electronic component for digital gates. The various gates are interconnected inside the chip to form the required circuit. Digital IC are categorized according to their circuit complexity as measured by the number of logic gates in a single packages.

- Small scale integration (SSI). SSi devices contain fewer than 10 gates. The input and output of the gates are connected directly to the pins in the package.
- Medium Scale Integration. MSI devices have the complexity of approximately 10 to 100 gates in a single package
- Large Scale Integration (LSI). LSI devices contain between 100 and a few thousand gates in a single package
- Very Large Scale Integration(VLSI). VLSI devices contain thousand of gates within a single package. VLSI

devices have revolutionized the computer system design technology giving the designer the capabilities to create structures that previously were uneconomical.

Multiplexer

A multiplexer is a combinatorial circuit that is given a certain number (usually a power of two) *data inputs*, let us say 2^n, and n *address inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the selected data input.

In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input.

The truth table for a multiplexer is huge for all but the smallest values of n. We therefore use an abbreviated version of the truth table in which some inputs are replaced by `-' to indicate that the input value does not matter.

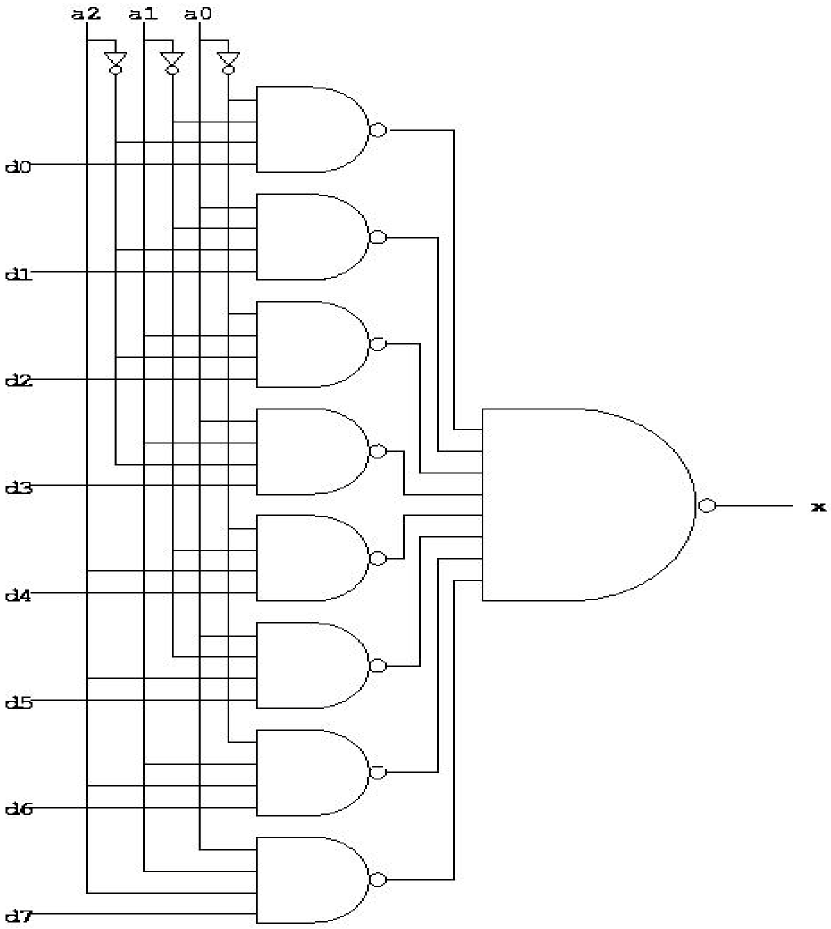
Here is such an abbreviated truth table for n = 3. The full truth table would have $2^{(3 + 23)}$ = 2048 rows.

SELECT			INPUT								
a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	0	0
0	0	0	-	-	-	-	-	-	-	1	1
0	0	1	-	-	-	-	-	-	0	-	0
0	0	1	-	-	-	-	-	-	1	-	1
0	1	0	-	-	-	-	-	0	-	-	0
0	1	0	-	-	-	-	-	1	-	-	1
0	1	1	-	-	-	-	0	-	-	-	0
0	1	1	-	-	-	-	1	-	-	-	1
1	0	0	-	-	-	0	-	-	-	-	0
1	0	0	-	-	-	1	-	-	-	-	1
1	0	1	-	-	0	-	-	-	-	-	0
1	0	1	-	-	1	-	-	-	-	-	1
1	1	0	-	0	-	-	-	-	-	-	0
1	1	0	-	1	-	-	-	-	-	-	1
1	1	1	0	-	-	-	-	-	-	-	0
1	1	1	1	-	-	-	-	-	-	-	1

We can abbreviate this table even more by using a letter to indicate the value of the selected input, like this:

a_2	a_1	a_0	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	x
0	0	0	-	-	-	-	-	-	-	c	c
0	0	1	-	-	-	-	-	-	c	-	c
0	1	0	-	-	-	-	-	c	-	-	c
0	1	1	-	-	-	-	c	-	-	-	c
1	0	0	-	-	-	c	-	-	-	-	c
1	0	1	-	-	c	-	-	-	-	-	c
1	1	0	-	c	-	-	-	-	-	-	c
1	1	1	c	-	-	-	-	-	-	-	c

The same way we can simplify the truth table for the multiplexer, we can also simplify the corresponding circuit. Indeed, our simple design method would yield a very large circuit. The simplified circuit looks like this:



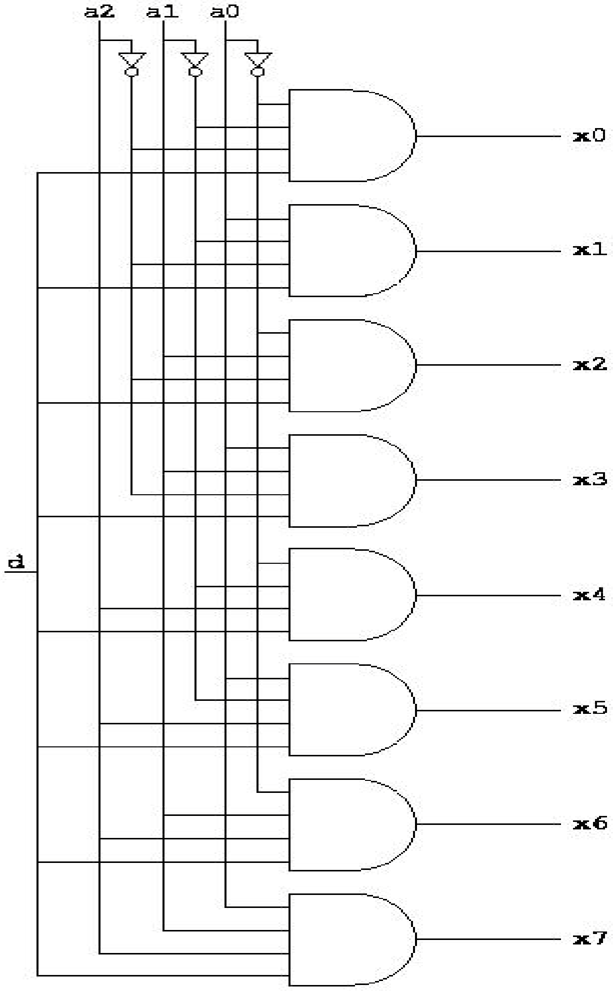
Demultiplexer

The demultiplexer is the inverse of the multiplexer, in that it takes a single data input and n address inputs. It has 2^n outputs. The address input determine which data output is going to have the same value as the data input. The other data outputs will have the value 0.

Here is an abbreviated truth table for the demultiplexer. We could have given the full table since it has only 16 rows, but we will use the same convention as for the multiplexer where we abbreviated the values of the data inputs.

a2	a1	a0	d	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	c	0	0	0	0	0	0	0	c
0	0	1	c	0	0	0	0	0	0	c	0
0	1	0	c	0	0	0	0	0	c	0	0
0	1	1	c	0	0	0	0	c	0	0	0
1	0	0	c	0	0	0	c	0	0	0	0
1	0	1	c	0	0	c	0	0	0	0	0
1	1	0	c	0	c	0	0	0	0	0	0
1	1	1	c	c	0	0	0	0	0	0	0

Here is one possible circuit diagram for the demultiplexer:



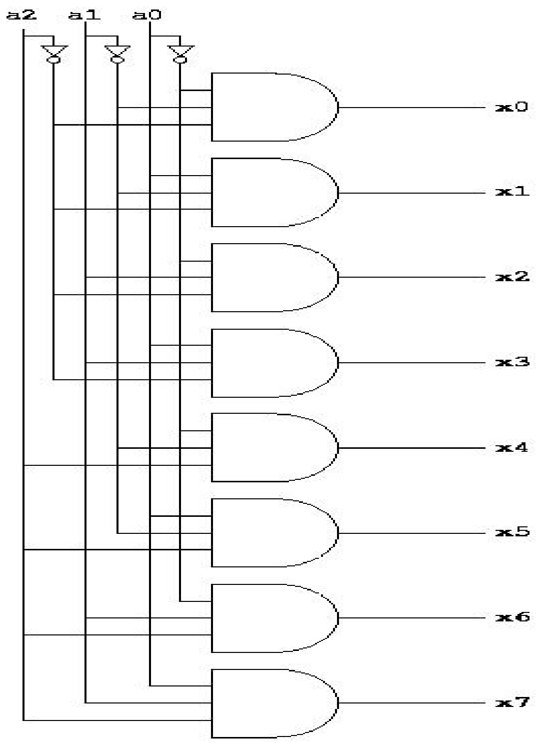
Decoder

In both the multiplexer and the demultiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of n digits to 2^n outputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0.

It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*. It has the following truth table (for n = 3):

a2	a1	a0	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Here is the circuit diagram for the decoder:



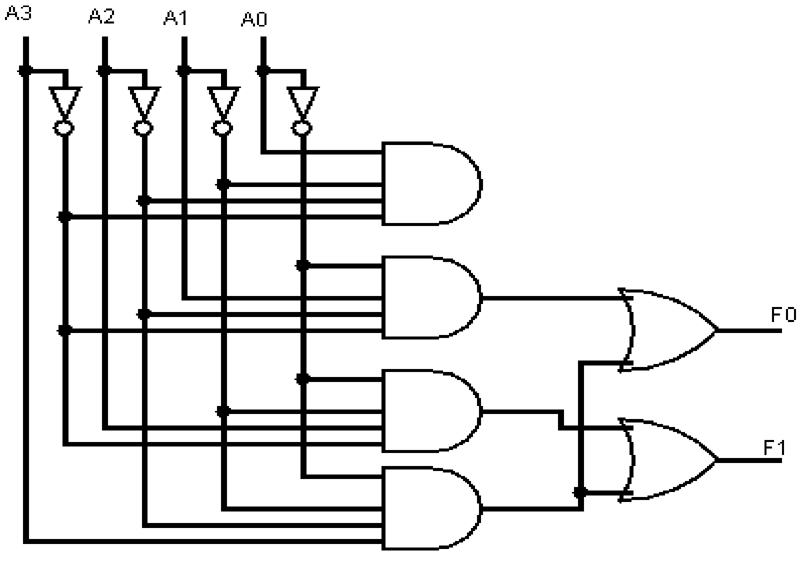
Encoder

An encoder has 2^n input lines and *n* output lines. The output lines generate a binary code corresponding to the input value. For example a single bit 4 to 2 encoder takes in 4 bits and outputs 2 bits. It is assumed that there are only 4 types of input signals these are : 0001, 0010, 0100, 1000.

I_3	**I_2**	**I_1**	**I_0**	**F_1**	F0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

F

4 to 2 encoder



The encoder has the limitation that only one input can be active at any given time. If two inputs are simultaneously active, the output produces an undefined combination. To prevent this we make use of the priority encoder.

A priority encoder is such that if two or more inputs are given at the same time, the input having the highest priority will take precedence. An example of a single bit 4 to 2 encoder is shown.

I_3	I_2	I_1	I_0	F_1	F_0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

4 to 2 priority encoder

The X's designate the don't care condition designating that fact that the binary value may be equal either to 0 or 1. For example, the input I_3has the highest priority so regarded the value of other inputs, if the value of I3 is 1, the output for F1F0=11(binary 3)

Exercise

1 By using algebraic manipulation, Show that:
- A'B'C+A'BC'+AB'C'+ABC= $A\odot(B\odot C)$ - AB+C'D = (A+B+C)(A+B'+C)(A'+B+C)(A'+B+C')

2. A circuit has four inputs D,C,B,A encoded in natural binary form where A is the least significant bit. The inputs in the range 0000=0 to 1011=11 represents the months of the year from January (0) to December (11). Input in the range 1100-1111(i.e.12 to 15) cannot occur. The output of the circuit is true if the month represented by the input has 31 days. Otherwise the output is false. The output for inputs in the range 1100 to 1111 is undefined.

- Draw the truth table to represent the problem and obtain the function F as a Sum of minterm.
- Use the Karnaugh map to obtain a simplified expression for the function F.
- Construct the circuit to implements the function using NOR gates only.

3. A circuit has four inputs P,Q,R,S, representing the natural binary number 0000=0, to 1111=15. P is the most significant bit. The circuit has one output, X, which is true if the input to the circuit represents is a prime number and false otherwise (A prime number is a number which is only divisible by 1 and by itself. Note that zero(0000) and one(0001) are not considered as prime numbers)

i. Design a true table for this circuit, and hence obtain an expression for X in terms of P,Q,R,S.
ii. Design a circuit diagram to implement this function using NOR gate only

4. A combinational circuit is defined by the following three Boolean functions: F1=x'y'z'+xz F2=xy'z'+x'y F3=x'y'z+xy Design the circuit that implements the functions

5. A circuit implements the Boolean function F=A'B'C'D'+A'BCD'+AB'C'D'+ABC'D It is found that the circuit input combinations A'B'CD', A'BC'D', AB'CD' can never occur.

i. Find a simpler expression for F using the proper don't care condition.
ii. Design the circuit implementing the simplified expression of F

6. A combinational circuit is defined by the following three Boolean functions: F1=x'y'z'+xz F2=xy'z'+x'y F3=x'y'z+xy Design the circuit with a decoder and external gates.

7. A circuit has four inputs P,Q,R,S, representing the natural binary number 0000=0, to 1111=15. P is the most significant bit. The circuit has one output, X, which is true if the number represented is divisible by three (Regard zero as being indivisible by three.)
Design a true table for this circuit, and hence obtain an expression for X in terms of P,Q,R,S as a product of maxterms and also as a sum of minterms
Design a circuit diagram to implement this function

8. Plot the following function on K map and use the K map to simplify the expression.

$$F = ABC + \overline{A}BC + A\overline{B}C + A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}\overline{C} \qquad F = A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C} + A\overline{B}C + A\overline{B}\overline{C}$$

9. Simplify the following expressions by means of Boolean algebra

$$F = \bar{A}\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}D + \bar{A}BCD + ABCD + \bar{A}\bar{B}C\bar{D} + A\bar{B}C\bar{D}$$

$$F = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + AB\bar{C} + \bar{A}BC + A\bar{B}C$$

10. 10. Design a 32 to 1 mux using lower order mux.

11. 11. Design a 3 to 8 line decoder using only 2 to 4 line decoders.