# UNIT II

## 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'

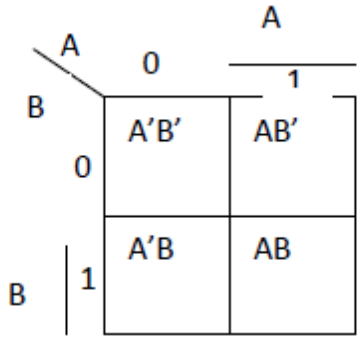
Simplify xy +x'z+yz xy +x'z+yz

= xy +x'z+yz(x+x') xy +x'z+yzx+yzx' xy(1+z) +x'z(1+y) =

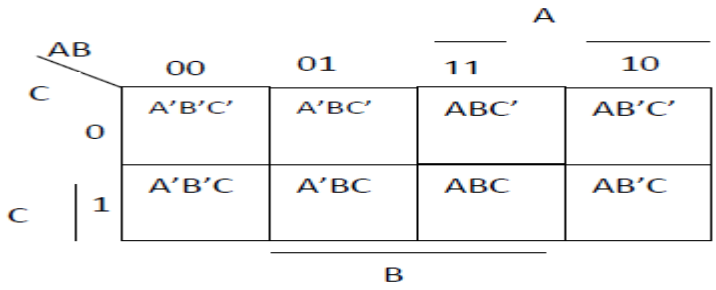
xy+x'z

Karnaugh map

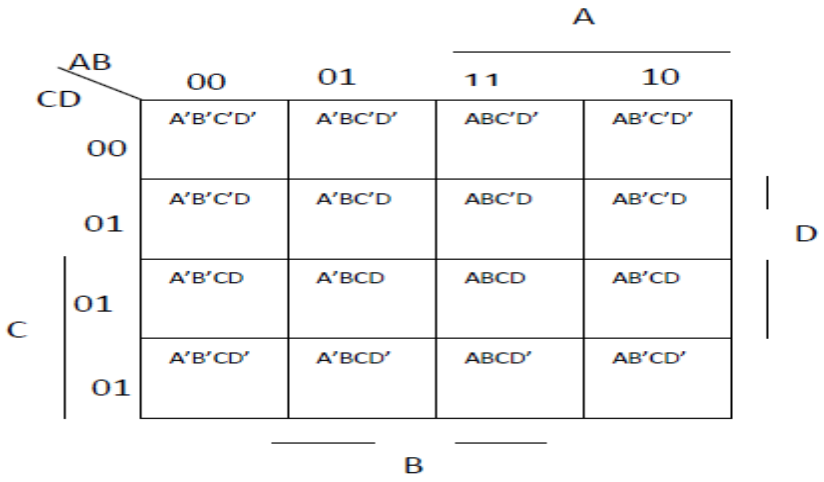
The Karnaugh map also known as Veitch diagram or simply as K map is a two dimensional form of the truth table, drawn in such a way that the simplification of Boolean expression can be immediately be seen from the location of 1's in the map. The map is a diagram made up of squares , each sqare represent 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 function can be represented as follow




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




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

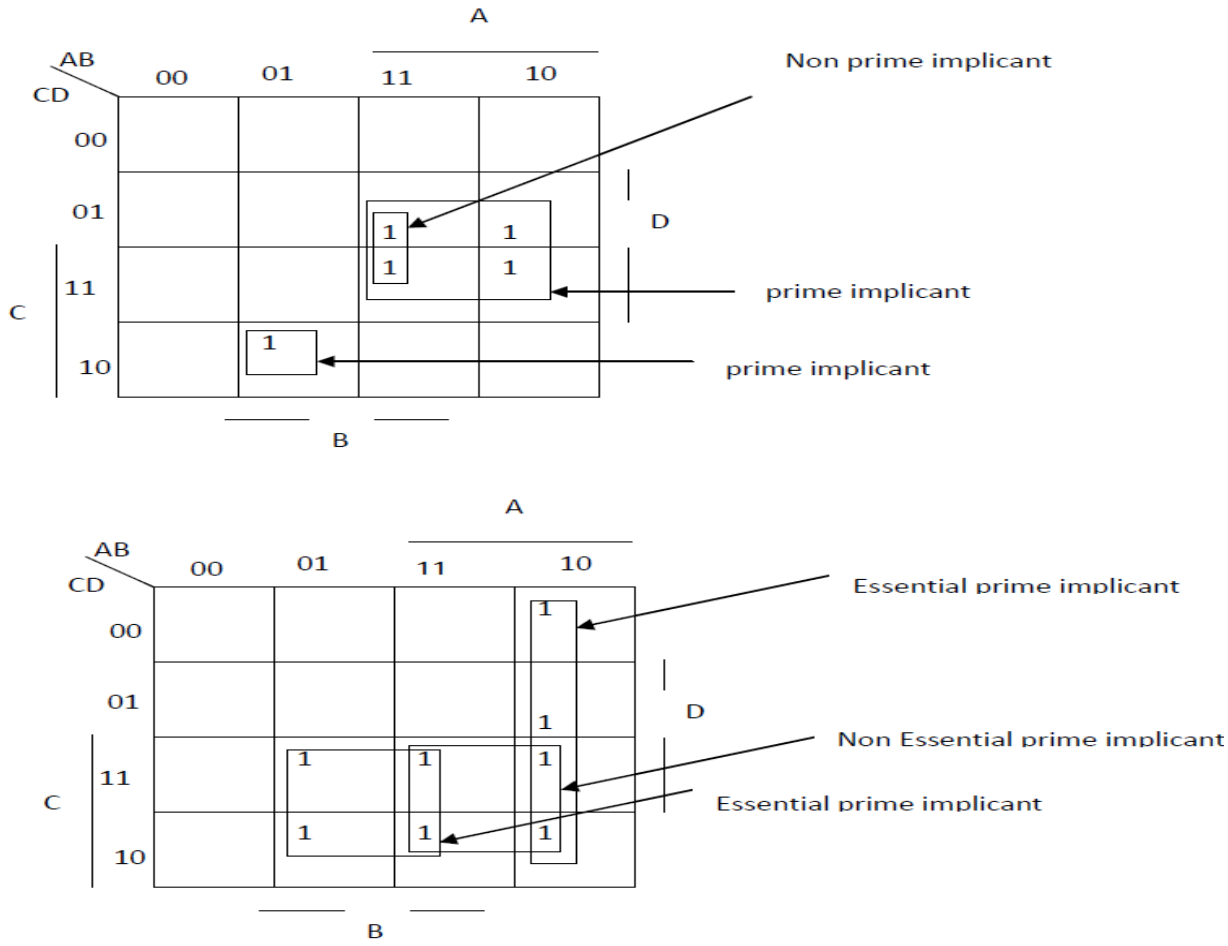
**Implicant**

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

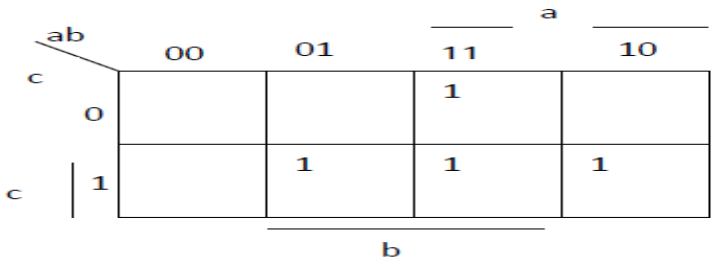
• *F* is a Boolean of *n* variables.
• *P* is a product term

**Prime implicant**

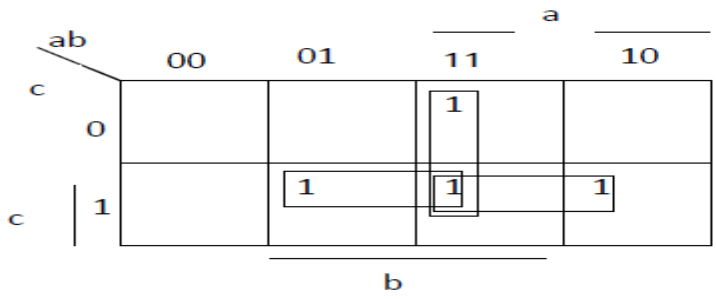
A prime implicant of a function is an implicant that cannot be covered by a more general (more reduced - meaning with fewer literals) implicant. W.V. Quine defined a prime implicant of F to be an implicant that is minimal - that is, if the removal of any literal from P results in a non-implicant for 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.



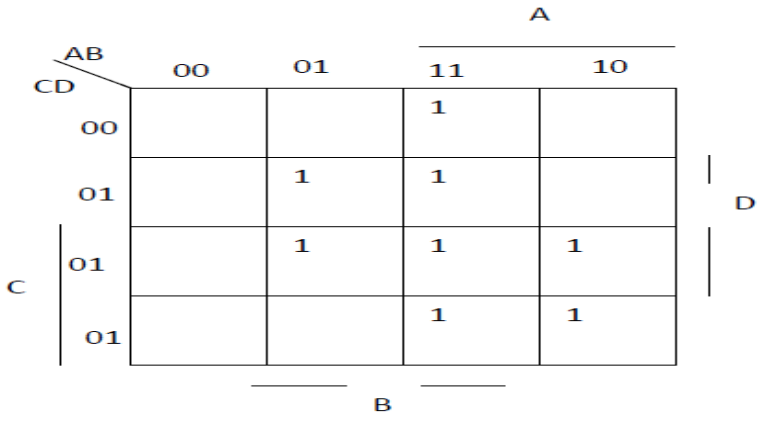

To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 101.



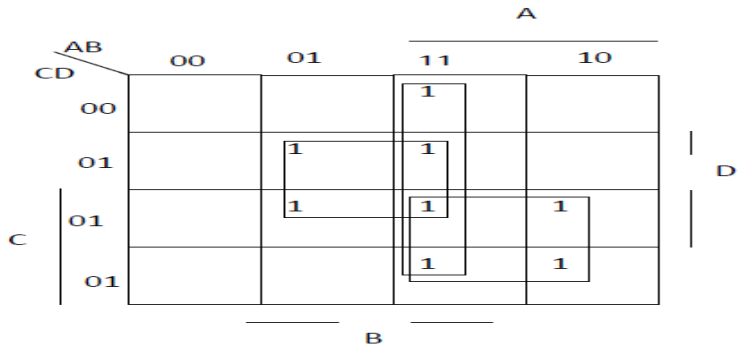

The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 110 (**a** = 1, **b** = 1, **c** = 0). The minimization is done by drawing circles around sets of adjacent 1s. Adjacency is horizontal, vertical, or both. The circles must always contain 2n 1s 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.

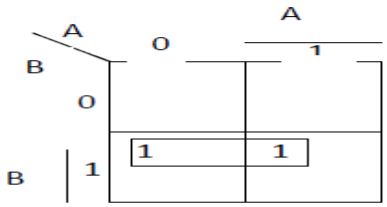


The following corresponds to the Boolean expression Q = A'BC'D + A'BCD + ABC'D' + ABC'D + ABCD + ABCD' + AB'CD + AB'CD' RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s. Grouping the 1s together results in the following.



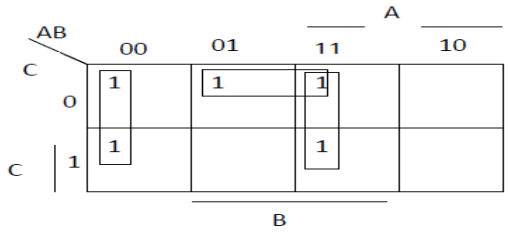

The expression for the groupings above is Q = BD + AC + AB This expression requires 3 2-input **AND** gates and 1 3-input **OR** gate. **Other examples**

**1. F=A'B+AB**



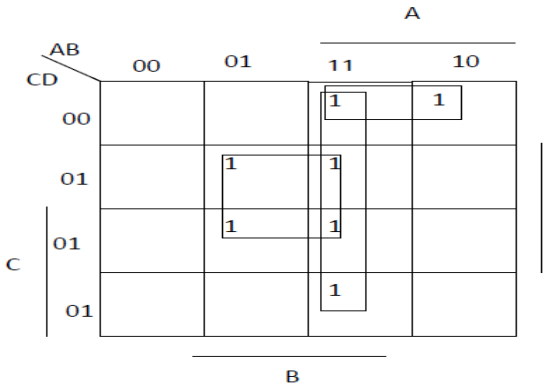

=B

2. F=A'B'C'+A'B'C+A'BC'+ABC'+ABC



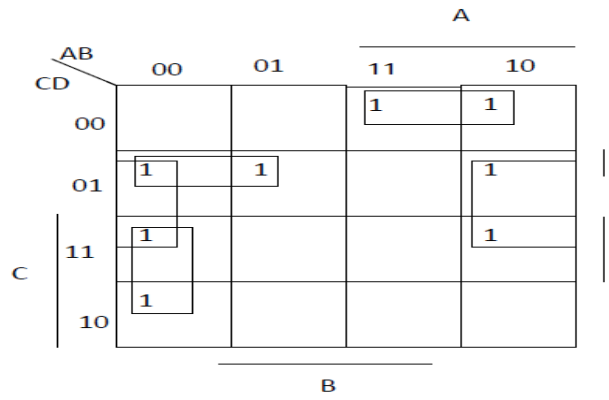

=A'B'+BC'+AB

3. F=AB+A'BC'D+A'BCD+AB'C'D'



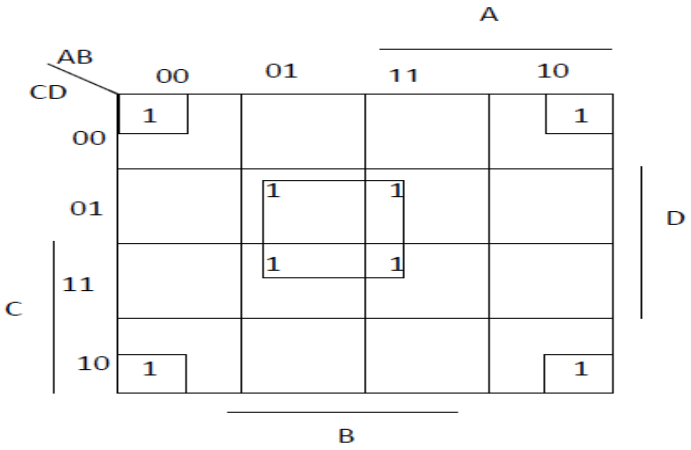

=BD+AB+AC'D'

4. F=AC'D'+A'B'C+A'C'D+AB'D




=B'D+AC'D'+A'C'D+A'B'C

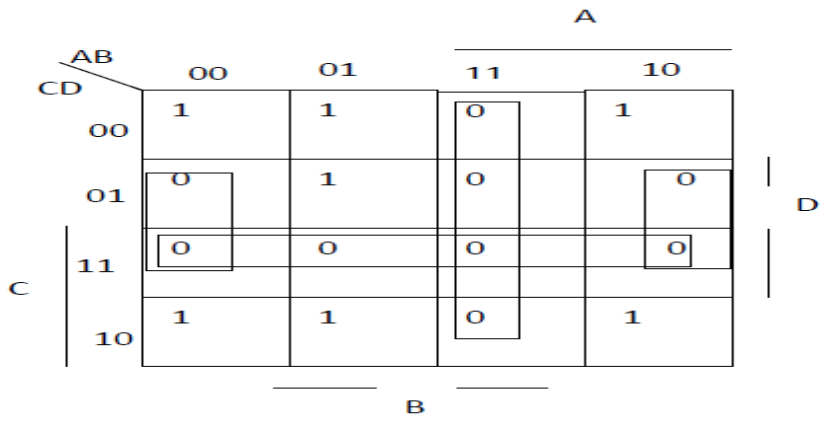
5. F=A'B'C'D'+AB'C'D'+A'BC'D+ABC'D+A'BCD+ABCD




=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.

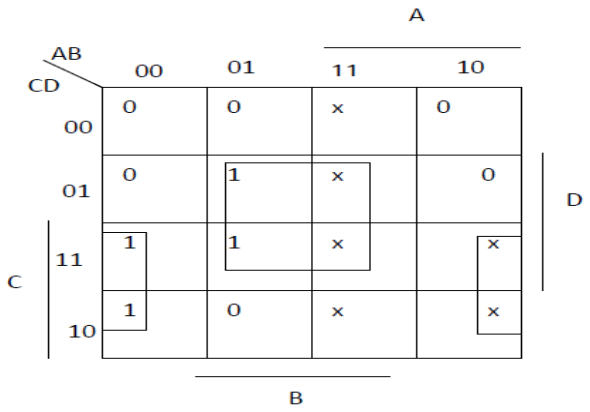
### Don't Care condition

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.




F=BD+B'C

### The tabulation method(Quine-McCluskey)

For function of five or more variables, it is difficult to be sure that the best selection is made. In such case, the tabulation method can be used to overcome such difficulty. The tabulation method was first formulated by Quine and later improved by McCluskey. It is also known as Quine-McCluskey method.

The Quine–McCluskey algorithm (or the method of prime implicants) is a method used for minimization of boolean functions. It is functionally identical to Karnaugh mapping, but the tabular form makes it

more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a Boolean function has been reached. The method involves two steps: Finding all prime implicants of the function. Use those prime implicants in a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function.

Step 1: finding prime implicants Minimizing an arbitrary function:

| | A | B | C | D | f |
|---|---|---|---|---|---|
| m0 | 0 | 0 | 0 | 0 | 0 |
| m1 | 0 | 0 | 0 | 1 | 0 |
| m2 | 0 | 0 | 1 | 0 | 0 |
| m3 | 0 | 0 | 1 | 1 | 0 |
| m4 | 0 | 1 | 0 | 0 | 1 |
| m5 | 0 | 1 | 0 | 1 | 0 |
| m6 | 0 | 1 | 1 | 0 | 0 |
| m7 | 0 | 1 | 1 | 1 | 0 |
| m8 | 1 | 0 | 0 | 0 | 1 |
| m9 | 1 | 0 | 0 | 1 | x |
| m10 | 1 | 0 | 1 | 0 | 1 |
| m11 | 1 | 0 | 1 | 1 | 1 |
| m12 | 1 | 1 | 0 | 0 | 1 |
| m13 | 1 | 1 | 0 | 1 | 0 |
| m14 | 1 | 1 | 1 | 0 | x |
| m15 | 1 | 1 | 1 | 1 | 1 |

One can easily form the canonical sum of products expression from this table, simply by summing the minterms (leaving out don't-care terms) where the function evaluates to one: F(A,B,C,D) = A'BC'D' + AB'C'D' + AB'CD' + AB'CD + ABC'D' + ABCD Of course, that's certainly not minimal. So to optimize, all minterms that evaluate to one are first placed in a minterm table. Don't-care terms are also added into this table, so they can be combined with minterms:

| Number of 1s | Minterm | Binary Representation |
|---|---|---|
| 1 | m4 | 0100 |
| | m8 | 1000 |
| 2 | m9 | 1001 |
| | m10 | 1010 |
| | m12 | 1100 |
| 3 | m11 | 1011 |
| | m14 | 1110 |
| 4 | m15 | 1111 |

At this point, one can start combining minterms with other minterms. If two terms vary by only a single digit changing, that digit can be replaced with a dash indicating that the digit doesn't matter. Terms that can't be combined any more are marked with a "*". When going from Size 2 to Size 4, treat '-' as a third bit value. Ex: -110 and -100 or -11- can be combined, but not -110 and 011-. (Trick: Match up the '-' first.)

| Number of 1s | Minterm | 0-Cube | Size 2 Implicants | Size 4 Implicants |
|---|---|---|---|---|
| 1 | m4 | 0100 | m(4,12) -100* | m(8,9,10,11) 10--* |
| | m8 | 1000 | m(8,9) 100- | m(8,10,12,14) 1--0* |
| 2 | m9 | 1001 | m(8,10) 10-0 | m(10,11,14,15) 1-1-* |
| | m10 | 1010 | m(8,12) 1-00 | |
| | m12 | 1100 | m(9,11) 10-1 | |
| 3 | m11 | 1011 | m(10,11) 101- | |
| | m14 | 1110 | m(10,14) 1-10 | |
| 4 | m15 | 1111 | m(12,14) 11-0 | |
| | | | m(11,15) 1-11 | |
| | | | m(14,15) 111- | |

At this point, the terms marked with * can be seen as a solution. That is the solution is F=AB'+AD'+AC+BC'D'

If the karnaugh map was used, we should have obtain an expression simplier than this. To obtain a minimal form, we need to use the prime implicant chart

Step 2: prime implicant chart

None of the terms can be combined any further than this, so at this point we construct an essential prime implicant table. Along the side goes the prime implicants that have just been generated, and along the top go the minterms specified earlier. The don't care terms are not placed on top - they are omitted from this section because they are not necessary inputs.

| | 4 | 8 | 10 | 11 | 12 | 15 | |
|---|---|---|---|---|---|---|---|
| m(4,12) | X | | | | X | | -100 (BC'D') |
| m(8,9,10,11) | | X | X | X | | | 10--(AB') |
| m(8,10,12,14) | | X | X | | X | | 1--0 (AD') |
| m(10,11,14,15) | | | X | X | | X | 1-1- (AC) |

In the prime implicant table shown above, there are 5 rows, one row for each of the prime implicant and 6 columns, each representing one minterm of the function. X is placed in each row to indicate the minterms contained in the prime implicant of that row. For example, the two X in the first row indicate that minterm 4 and 12 are contained in the prime implicant represented by (-100) i.e. BC'D'

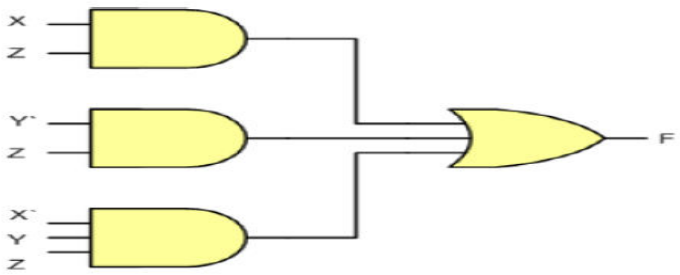
The completed prime implicant table is inspected for columns containing only a single x. in this example, there are two minterms whose column have a single x. 4,15. The minterm 4 is covered by prime implicant BC'D'. that is the selection of prime implicant BC'D' guarantee that minterm 4 is included in the selection. Similarly, for minterm 15 is covered by prime implicant AC. Prime implicants that cover

minterms with a single X in their column are called essential prime implicants. Those essential prime implicant must be selected. Now we find out each column whose minterm is covered by the selected essential prime implicant For this example, essential prime implicant BC'D' covers minterm 4 and 12. Essential prime implicant AC covers 10, 11 and 15. An inspection of the implicant table shows that, all the minterms are covered by the essential prime implicant except the minterms 8. The minterms not selected must be included by the selection of one or more prime implicants. From this example, we have only one minterm which is 8. It can be included in the selection either by including the prime implicant AB' or AD'. Since both of them have minterm 8 in their selection. We have thus found the minimum set of prime implicants whose sum gives the required minimized function: F=BC'D'+AD'+AC OR F= BC'D'+AB'+AC. Both of those final equations are functionally equivalent to this original (very area-expensive) equation: F(A,B,C,D) = A'BC'D' + AB'C'D' + AB'CD' + AB'CD + ABC'D' + ABCD

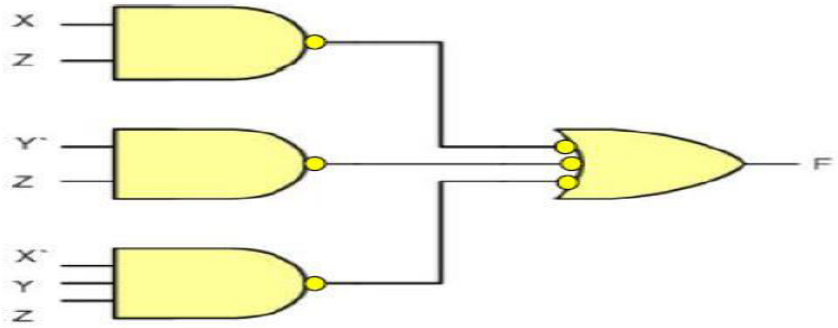
**Implimenting logical circuit using NAND and NOR gate only.**

In addition to AND, OR, and NOT gates, other logic gates like NAND and NOR are also used in the design of digital circuits. The NAND gate represents the complement of the AND operation. Its name is an abbreviation of NOT AND. The graphic symbol for the NAND gate consists of an AND symbol with a bubble on the output, denoting that a complement operation is performed on the output of the AND gate as shown earlier The NOR gate represents the complement of the OR operation. Its name is an abbreviation of NOT OR. The graphic symbol for the NOR gate consists of an OR symbol with a bubble on the output, denoting that a complement operation is performed on the output of the OR gate as shown earlier. A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The NAND and NOR gates are universal gates. In practice, this is advantageous since NAND and NOR gates are economical and easier to fabricate and are the basic gates used in all IC digital logic families. In fact, an AND gate is typically implemented as a NAND gate followed by an inverter not the other way around. Likewise, an OR gate is typically implemented as a NOR gate followed by an inverter not the other way around.

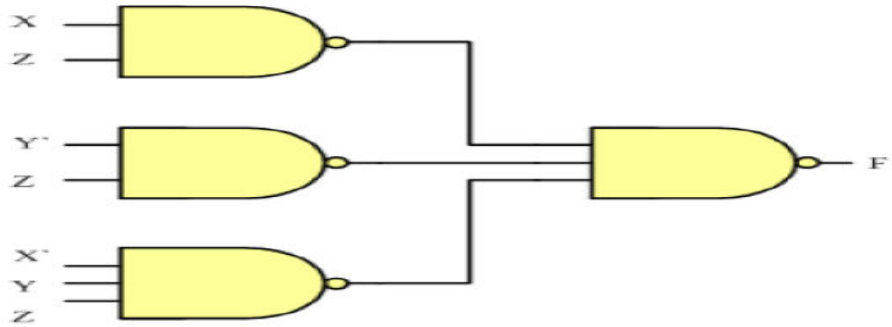
**Two-Level Implementations:** We have seen before that Boolean functions in either SOP or POS forms can be implemented using 2-Level implementations. For SOP forms AND gates will be in the first level and a single OR gate will be in the second level. For POS forms OR gates will be in the first level 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. To implement a function using NAND gates only, it must first be simplified to a sum of product and to implement a function using NOR gates only, it must first be simplified to a product of sum We will show that SOP forms can be implemented using only NAND gates, while POS forms can be implemented using only NOR gates through examples. **Example 1:** Implement the following SOP function using NAND gate only **F = XZ + Y'Z + X'YZ** Being an SOP expression, it is implemented in 2-levels as shown in the figure.



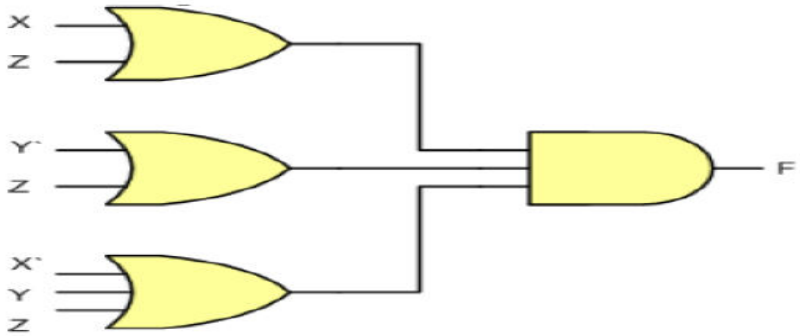
Introducing two successive inverters at the inputs of the OR gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.



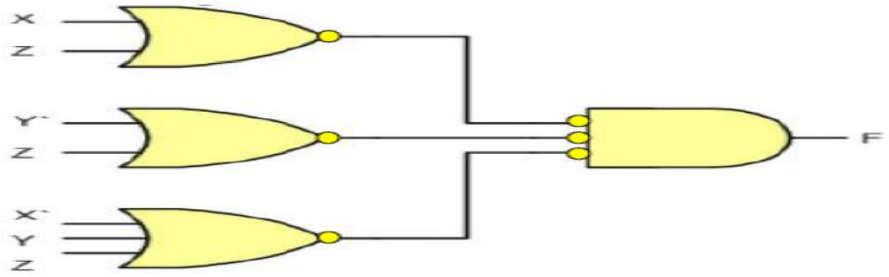

By associating one of the inverters with the output of the first level AND gate and the other with the input of the OR gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NAND gates as shown in Figure.



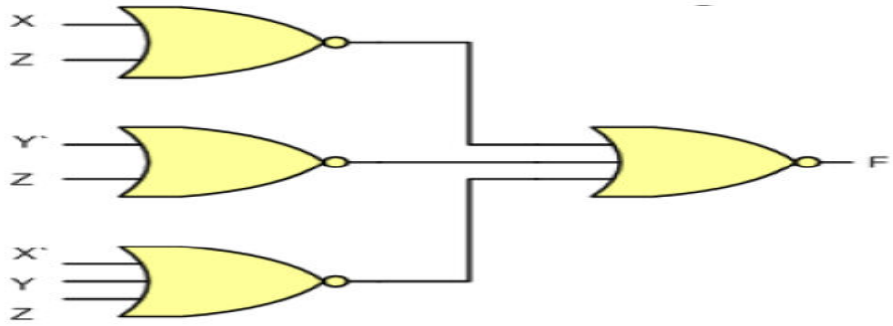

**Example 2:** Implement the following POS function using NOR gates only **F = (X+Z) (Y'+Z) (X'+Y+Z)** Being a POS expression, it is implemented in 2-levels as shown in the figure.



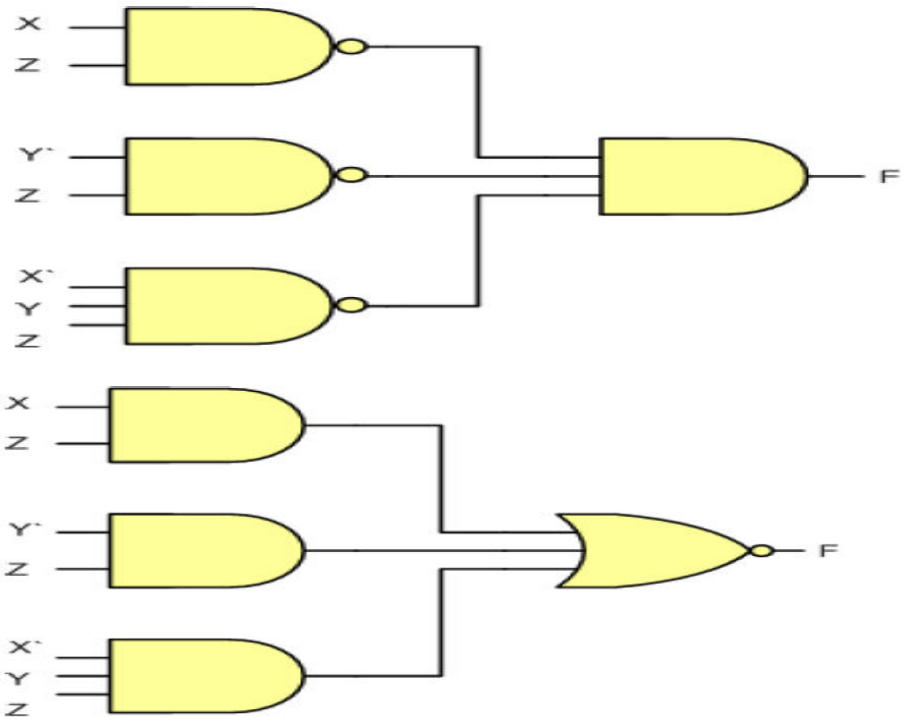

Introducing two successive inverters at the inputs of the AND gate results in the shown equivalent implementation. Since two successive inverters on the same line will not have an overall effect on the logic as it is shown before.



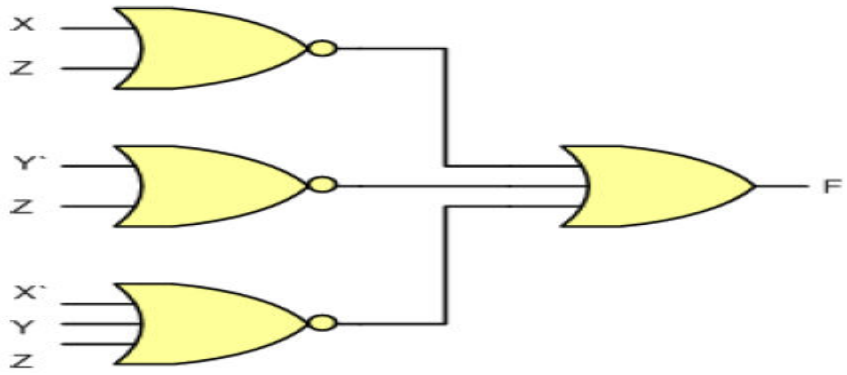

By associating one of the inverters with the output of the first level OR gates and the other with the input of the AND gate, it is clear that this implementation is reducible to 2-level implementation where both levels are NOR gates as shown in Figure.




There are some other types of 2-level combinational circuits which are • NAND-AND • AND-NOR, • NOR-OR, • OR-NAND These are explained by examples. **AND-NOR functions:** Example 3: Implement the following function *F=(XZ+Y′Z+X′YZ)′ OR F′=XZ+Y′Z+X′YZ* Since F′ is in SOP form, it can be implemented by using NAND-NAND circuit. By complementing the output we can get F, or by using *NAND-AND* circuit as shown in the figure



**OR-NAND functions:** Example 4: Implement the following function F=((X+Z)(Y**'+Z**)(X**'+Y+Z**)) **'** or**F'** (X+Z)(Y**'+Z**)(X**'+Y+Z**) Since **F'** is in POS form, it can be implemented by using NOR-NOR circuit. By complementing the output we can get **F**, **or** by using ***NOR****-OR* circuit as shown in the figure.




It can also be implemented using ***OR-NAND*** circuit as it is equivalent to NOR-OR circuit as shown in the figure