# *DIGITAL LOGIC DESIGN*

3RD sem CSE
Faculty Name: MITALI PANDA

# Content

Module-II:
Minimization of Boolean Functions: Karnaugh Map, Don’t care conditions, Prime Implicants, Quine-McCluskey technique, Logic gates, NAND/NOR gates, Universal gates.

# KARNAUGH MAPS ( K- MAP)

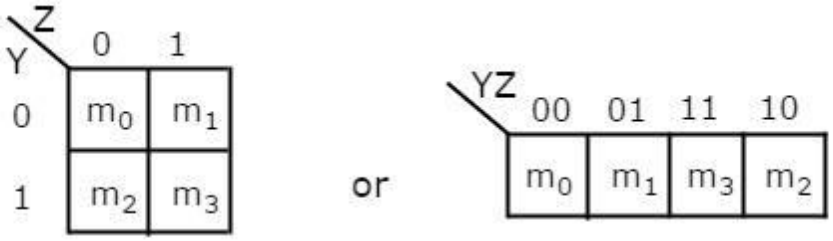
A method for graphically determining implicants and implicates of a Boolean function was developed by Veitch and modified by Karnaugh . The method involves a diagrammatic representation of a Boolean algebra. This graphic representation is called map.
It is seen that the truth table can be used to represent complete function of n-variables. Since each variable can have value of 0 or 1. The truth table has 2n rows. Each rows of the truth table consist of two parts (1) an n-tuple which corresponds to an assignment to the n-variables and (2) a functional value.
A Karnaugh map (K-map) is a geometrical configuration of 2n cells such that each of the n-tuples corresponds to a row of a truth table uniquely locates a cell on the map. The functional values assigned to the n-tuples are placed as entries in the cells, i.e. 0 or 1 are placed in the associated cell.
An significant about the construction of K-map is the arrangement of the cells. Two cells are physically adjacent within the configuration if and only if their respective n-tuples differ in exactly by one element. So that the Boolean law x+x=1 cab be applied to adjacent cells. Ex. Two 3- tuples (0,1,1) and (0,1,0) are physically adjacent since these tuples vary by one element.

**One variable** :

One variable needs a map of $2^1$= 2 cells map.

**2 Variable K-Map**

The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.
Two variable needs a map of $2^2$ = 4 cells

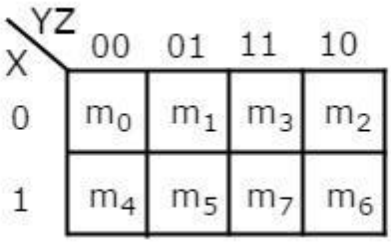



There is only one possibility of grouping 4 adjacent min terms.

The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m2, m3), (m0, m2) and (m1, m3)}.

**3 Variable K-Map**

The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.




There is only one possibility of grouping 8 adjacent min terms.

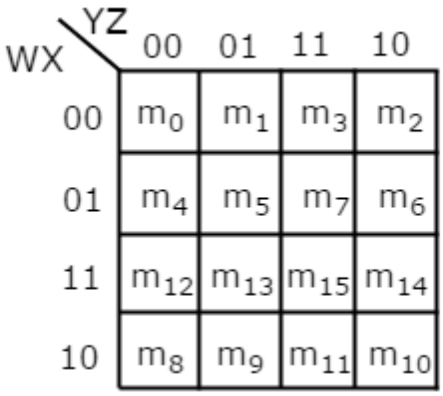
The possible combinations of grouping 4 adjacent min terms are {(m0, m1, m3, m2), (m4, m5, m7, m6), (m0, m1, m4, m5), (m1, m3, m5, m7), (m3, m2, m7, m6) and (m2, m0, m6, m4)}.

The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m1, m3), (m3, m2), (m2, m0), (m4, m5), (m5, m7), (m7, m6), (m6, m4), (m0, m4), (m1, m5), (m3, m7) and (m2, m6)}.

If x=0, then 3 variable K-map becomes 2 variable K-map.

**4 Variable K-Map**

The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.



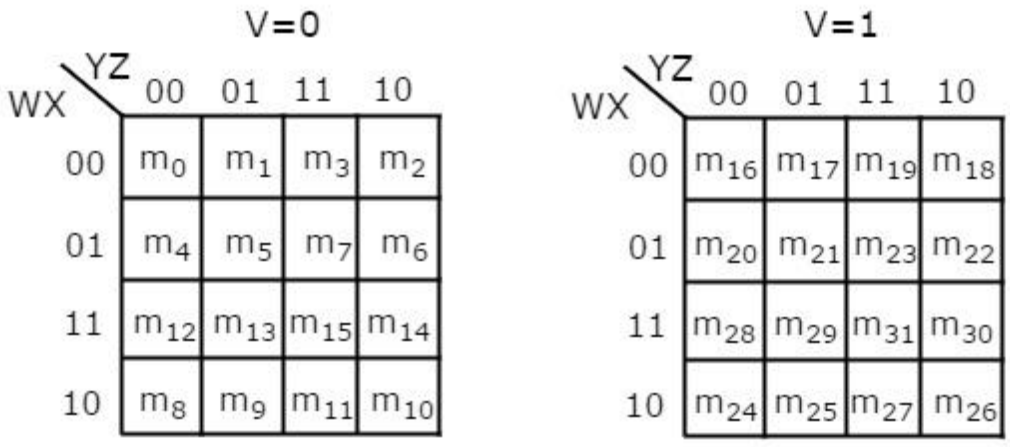

There is only one possibility of grouping 16 adjacent min terms.

Let R1, R2, R3 and R4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C1, C2, C3 and C4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R1, R2), (R2, R3), (R3, R4), (R4, R1), (C1, C2), (C2, C3), (C3, C4), (C4, C1)}.

If w=0, then 4 variable K-map becomes 3 variable K-map.

**5 Variable K-Map**

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.




There is only one possibility of grouping 32 adjacent min terms.

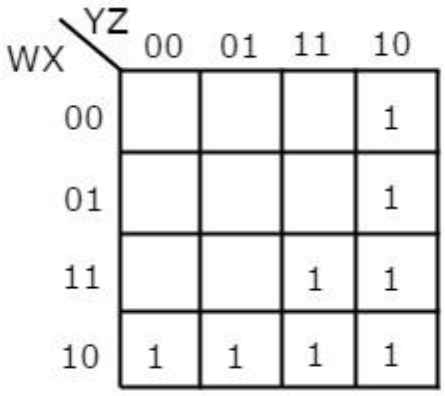
There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m0 to m15 and m16 to m31.

If v=0, then 5 variable K-map becomes 4 variable K-map.
In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

**Minimization of Boolean Functions using K-Maps**

If we consider the combination of inputs for which the Boolean function is „1", then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.
Similarly, if we consider the combination of inputs for which the Boolean function is „0", then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

**Example**
Let us **simplify** the following Boolean function, **f(W, X, Y, Z)= WX'Y' + WY + W'YZ'** using K-map.
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.

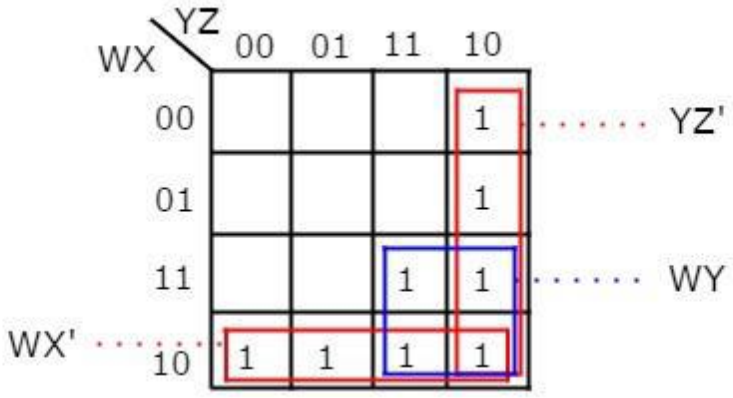



Here, 1s are placed in the following cells of K-map.
The cells, which are common to the intersection of Row 4 and columns 1 & 2 are corresponding to the product term, **WX'Y'**.

The cells, which are common to the intersection of Rows 3 & 4 and columns 3 & 4 are corresponding to the product term, **WY**.

The cells, which are common to the intersection of Rows 1 & 2 and column 4 are corresponding to the product term, **W'YZ'**.

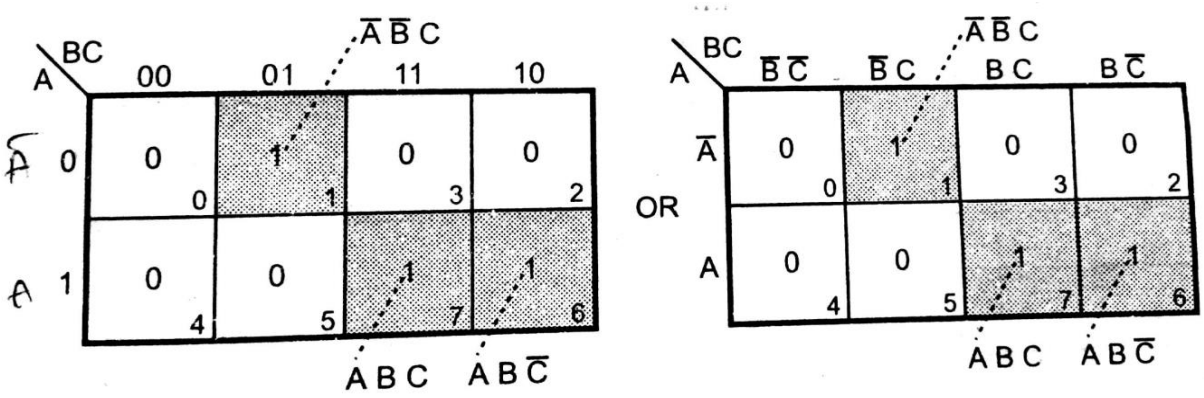
There are no possibilities of grouping either 16 adjacent ones or 8 adjacent ones. There are three possibilities of grouping 4 adjacent ones. After these three groupings, there is no single one left as ungrouped. So, we no need to check for grouping of 2 adjacent ones. The **4 variable K-map** with these three **groupings** is shown in the following figure.




Here, we got three prime implicants WX‟, WY & YZ‟
Therefore, the **simplified Boolean function** is **f= WX' + WY + YZ'**

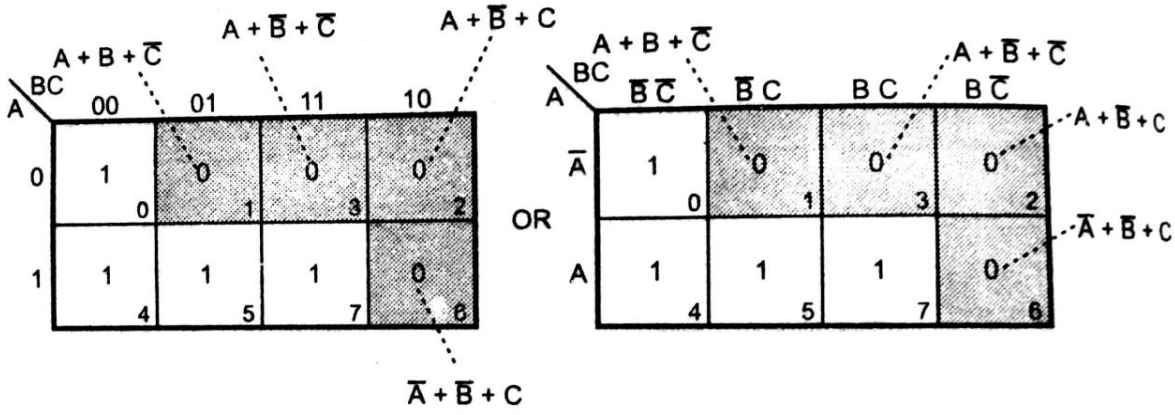
**Standard SOP Representation on k-map**

- A boolean expression with product terms can be plotted on the k-map by placing a 1 in each cell corresponding to a term in the SOP expression.
- Ex: **Y=ABC'+ABC+A'B'C**



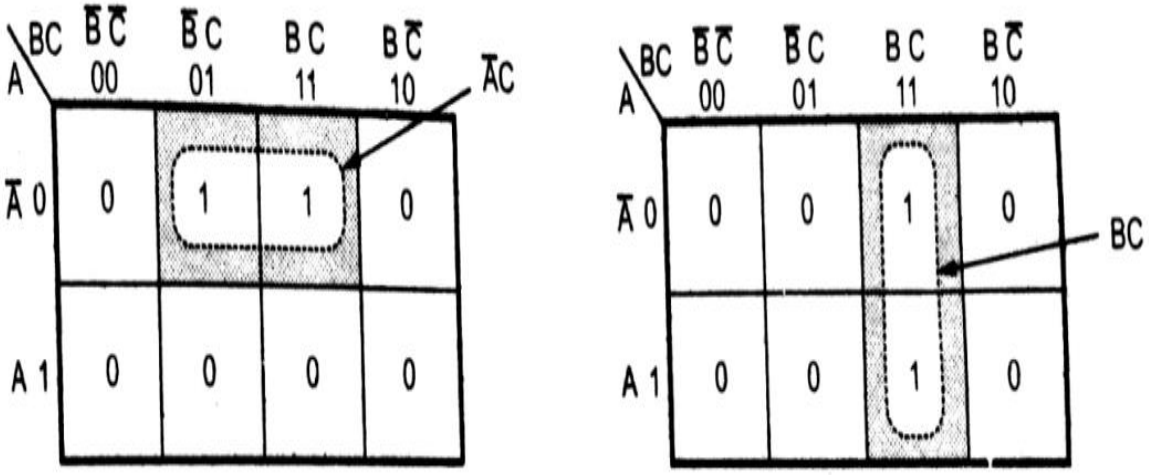

**Standard POS Representation on K-map**

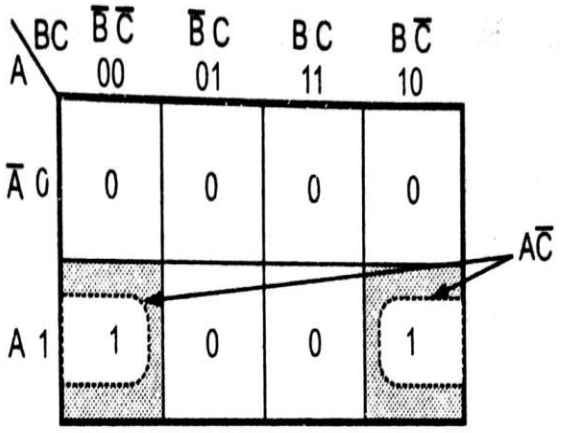
- A boolean expression with sum terms can be plotted on the k-map by placing a 0 in each cell corresponding to a term in the POS expression.
- Ex: **Y=(A+B'+C)(A+B'+C')(A'+B'+C)(A+B+C')**

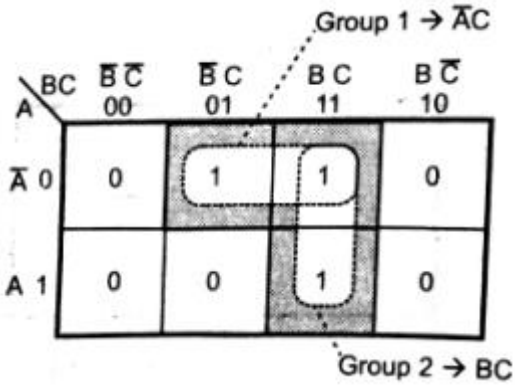


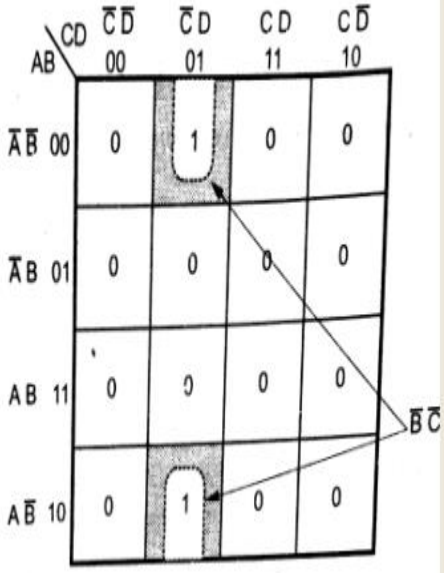

**Obtain minimal expression using K-map**

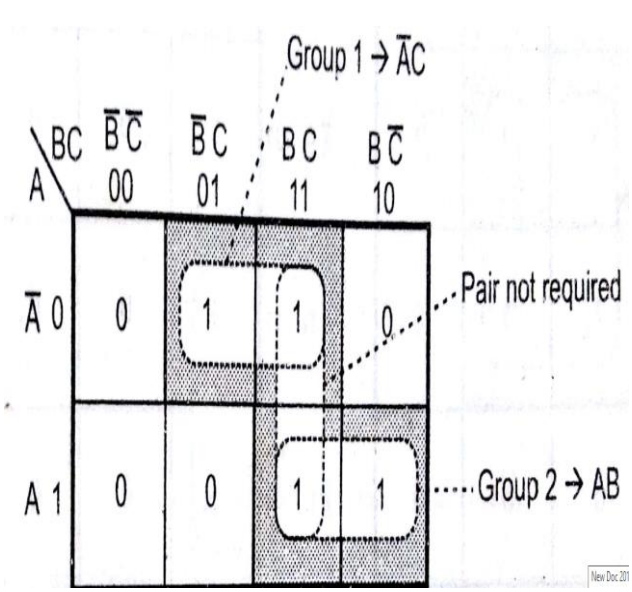
- Grouping Two Adjacent Ones(Pair)



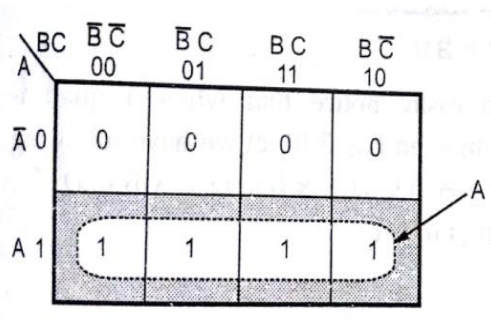


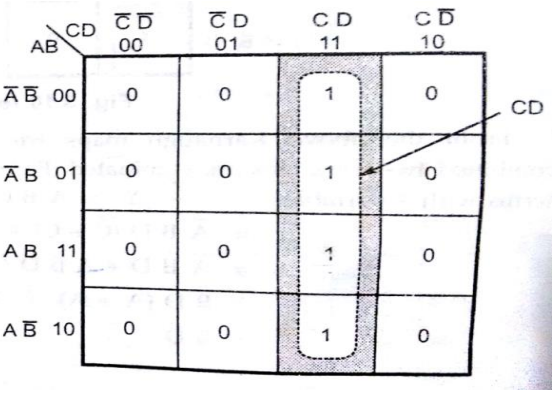



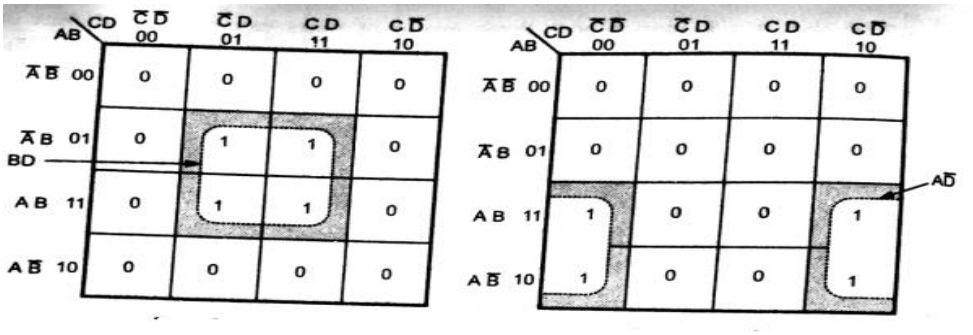



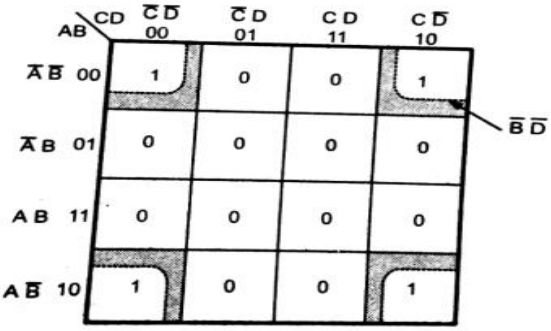



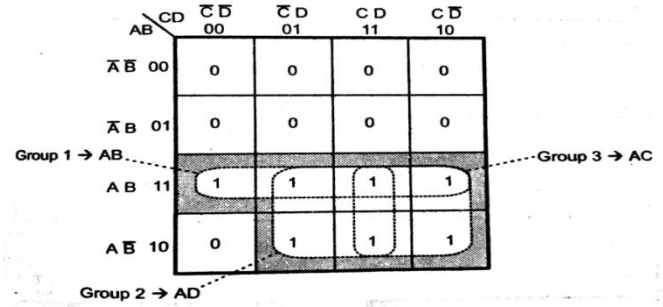

- Grouping Four Adjacent ones(Quad):



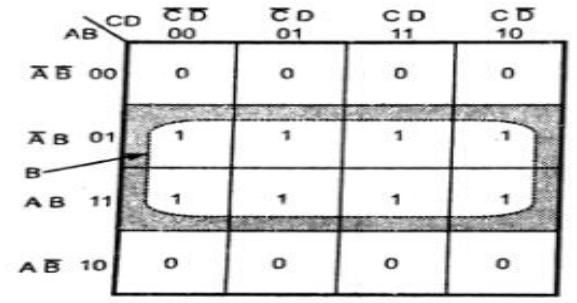



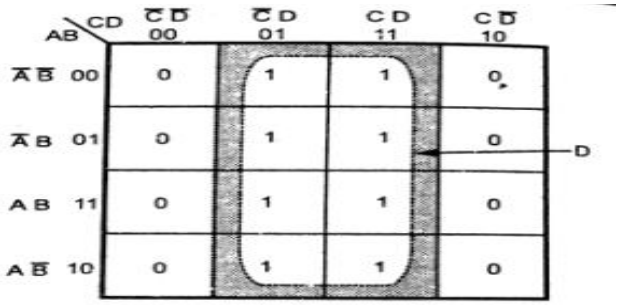


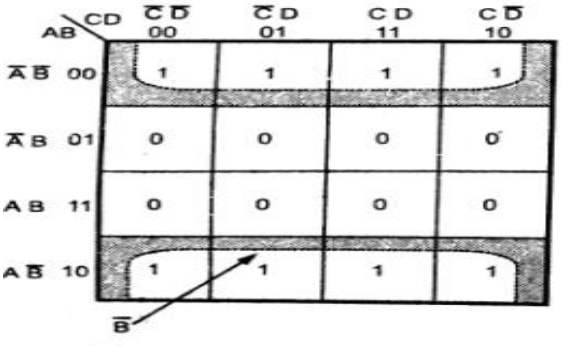



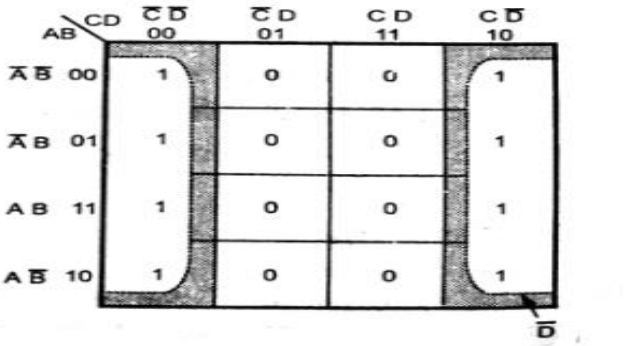




- Grouping Eight Adjacent ones(Octet):



# Don’t Care (X) Conditions in K-Maps

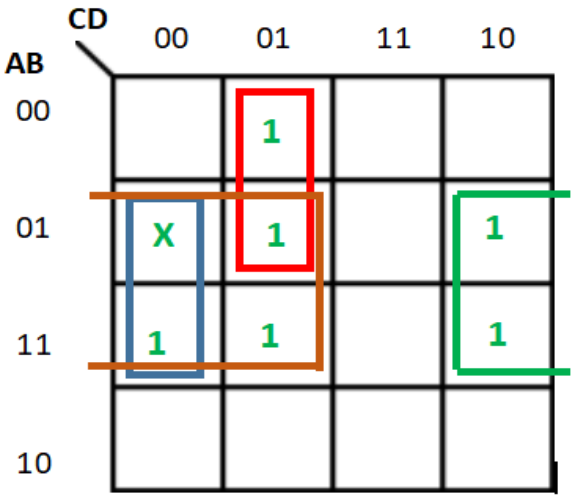
The “Don’t Care” conditions allow us to replace the empty cell of a K-Map to form a grouping of the variables. While forming groups of cells, we can consider a “Don’t Care” cell as either 1 or 0 or we can simply ignore that cell. Therefore, “Don’t Care” condition can help us to form a larger group of cells.
A Don’t Care cell can be represented by a cross(X) in K-Maps representing a invalid combination. For example, in Excess-3 code system, the states 0000, 0001, 0010, 1101, 1110 and 1111 are invalid or unspecified. These are called don’t cares. Also, in design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don’t cares.

A standard SOP function having don’t cares can be converted into a POS expression by keeping don’t cares as they are, and writing the missing minterms of the SOP form as the maxterm of POS form. Similarly, a POS function having don’t cares can be converted to SOP form keeping the don’t cares as they are and write the missing maxterms of the POS expression as the minterms of SOP expression.

**Example-1:**
Minimise the following function in SOP minimal form using K-Maps:

f = m(1, 5, 6, 12, 13, 14) + d(4)

**Explanation:**
The SOP K-map for the given expression is:




Therefore, SOP minimal is,

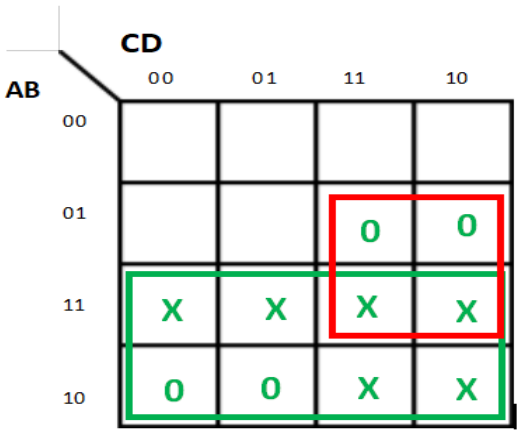
f = BC' + BD' + A'C'D

**Example-2:**
Minimise the following function in SOP minimal form using K-Maps:

F(A, B, C, D) = m(0, 1, 2, 3, 4, 5) + d(10, 11, 12, 13, 14, 15)

**Explanation:**
Writing the given expression in POS form:

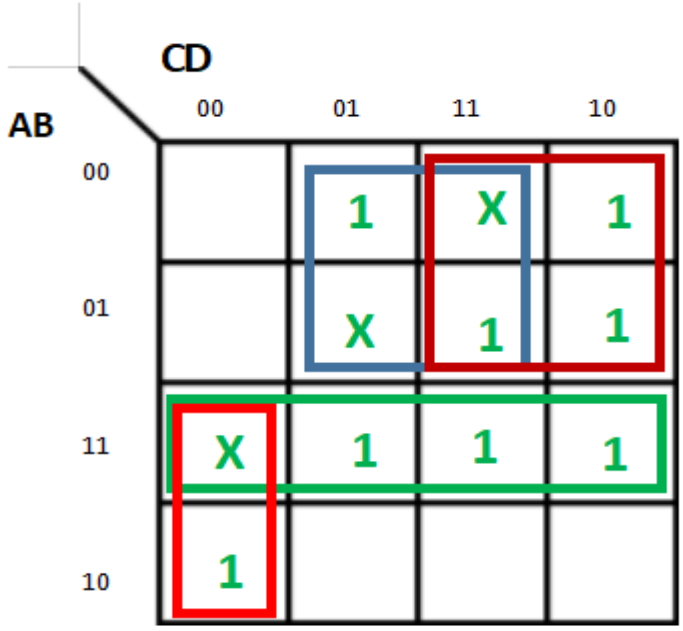
F(A, B, C, D) = M(6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)




Therefore, POS minimal is,

F = A'(B' + C')

**Example-3:**
Minimise the following function in SOP minimal form using K-Maps: F(A, B, C, D) = m(1, 2, 6, 7, 8, 13, 14, 15) + d(3, 5, 12)
**Explanation:**
The SOP K-map for the given expression is:




Therefore,

f = AC'D' + A'D + A'C + AB

**Significance of "Don't Care" Conditions:**

Don't Care conditions has the following significance with respect to the digital circuit design:

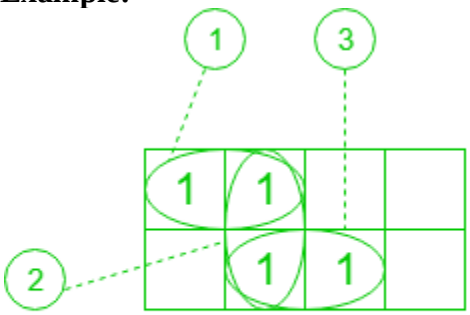
1. **Simplification:**
   These conditions denotes the set of inputs which never occurs for a given digital circuits. Thus, they are being used to further simplify the boolean output expression.
2. **Lesser number of gates:**
   Simplification reduces the number of gates to be used for implementing the given expression. Therefore, don't cares make the digital circuit design more economical.
3. **Reduced Power Consumption:**
   While grouping the terms long with don't cares reduces switching of the states. This decreases the required memory space which in turn results in less power consumption.
4. **States in Code Converters:**
   These are used in code converters. For example- In design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don't cares.
5. **Prevention of Hazards:**
   Don't cares also prevents hazards in digital systems.

# Prime implicants, and essensial prime implicants

**Prime Implicants –**

A group of square or rectangle made up of bunch of adjacent minterms which is allowed by definition of K-Map are called **prime implicants(PI)** i.e. all possible groups formed in K-Map.

**Example:**

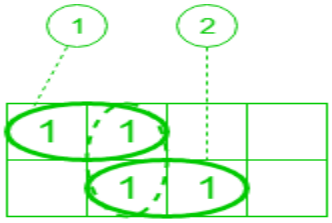



No. of Prime Implicants = 3

**Essential Prime Implicants –**

These are those subcubes(groups) which cover atleast one minterm that can't be covered by any other prime implicant. **Essential prime implicants(EPI)** are those prime implicants which always appear in final solution.

**Example:**




No. of Essential Prime Implicants = 2

## Limitations of k-map

- The map method of simplification is convenient as long as the number of variables does not exceed five or six.
- In case of complex problem with 7,8 or 10 variables it is almost impossible task to simplify expression by the map method.
- K-map simplification is a manual technique and simplification process is heavily depends on human abilities.

# Tabular Method or Quine-McCluskey Method of Minimization

- The tabular method which is also known as the Quine-Mc Cluskey method is particularly useful when minimising functions having a large number of variables, e.g. The six-variable functions. Computer programs have been developed employing this algorithm.
- The method reduces a function in standard sum of products form to a set of ***prime implicants*** from which as many variables are eliminated as possible.
- These prime implicants are then examined to see if some are redundant.

**Algorithm for Quine-McCluskey Method**

**Step-1:**

- Arrange all minterms in groups, such that all terms in the same group have the same number of 1's in their binary representation.
- The number of 1's in a term is called the **index** of that term.
- EX: F(W,X,Y,Z) = $\sum(0,1,2,5,7,8,9,10,13,15)$

Step 2:

- Compare every term of the lowest index group with each term in the sucessive group.
- Combine two terms being compared by the means of combining theorem AB+AB'=A.
- Repeat this by comparing each term in a group of index i with every term in the group of i+1.

**Step-3:**

- Now compare the terms generated in step2.
- A new term generated by combining two terms that differ by only a single 1 and whose dashes are in same position.
- The process continues until no further combinations are possible.

Step 1

| | w | x | y | z | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | ✓ |
| 1 | 0 | 0 | 0 | 1 | ✓ |
| 2 | 0 | 0 | 1 | 0 | ✓ |
| 8 | 1 | 0 | 0 | 0 | ✓ |
| 5 | 0 | 1 | 0 | 1 | ✓ |
| 9 | 1 | 0 | 0 | 1 | ✓ |
| 10 | 1 | 0 | 1 | 0 | ✓ |
| 7 | 0 | 1 | 1 | 1 | ✓ |
| 13 | 1 | 1 | 0 | 1 | ✓ |
| 15 | 1 | 1 | 1 | 1 | ✓ |

Step 2

| | w | x | y | z | |
|---|---|---|---|---|---|
| 0,1 | 0 | 0 | 0 | – | ✓ |
| 0,2 | 0 | 0 | – | 0 | ✓ |
| 0,8 | – | 0 | 0 | 0 | ✓ |
| 1,5 | 0 | – | 0 | 1 | ✓ |
| 1,9 | – | 0 | 0 | 1 | ✓ |
| 2,10 | – | 0 | 1 | 0 | ✓ |
| 8,9 | 1 | 0 | 0 | – | ✓ |
| 8,10 | 1 | 0 | – | 0 | ✓ |
| 5,7 | 0 | 1 | – | 1 | ✓ |
| 5,13 | – | 1 | 0 | 1 | ✓ |
| 9,13 | 1 | – | 0 | 1 | ✓ |
| 7,15 | – | 1 | 1 | 1 | ✓ |
| 13,15 | 1 | 1 | – | 1 | ✓ |

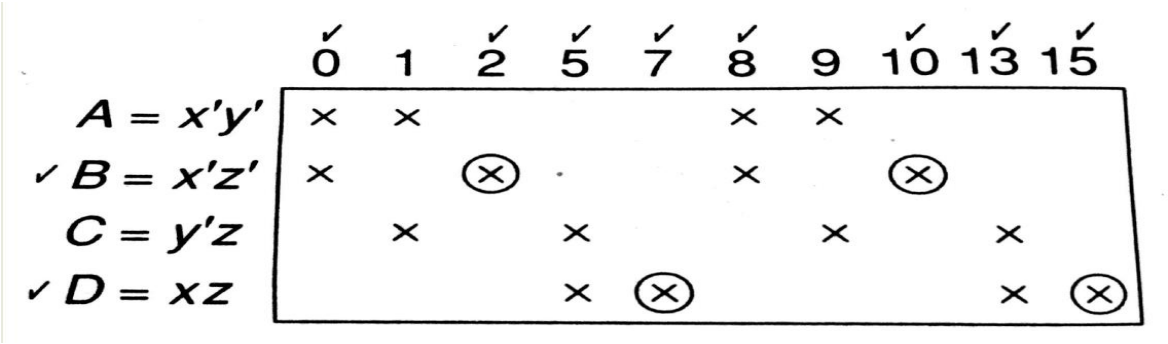
Step 3

| | w | x | y | z | |
|---|---|---|---|---|---|
| 0,1,8,9 | – | 0 | 0 | – | A |
| 0,2,8,10 | – | 0 | – | 0 | B |
| 1,5,9,13 | – | – | 0 | 1 | C |
| 5,7,13,15 | – | 1 | – | 1 | D |

Prime implicants are F=A+B+C+D

# Prime Implicant Chart

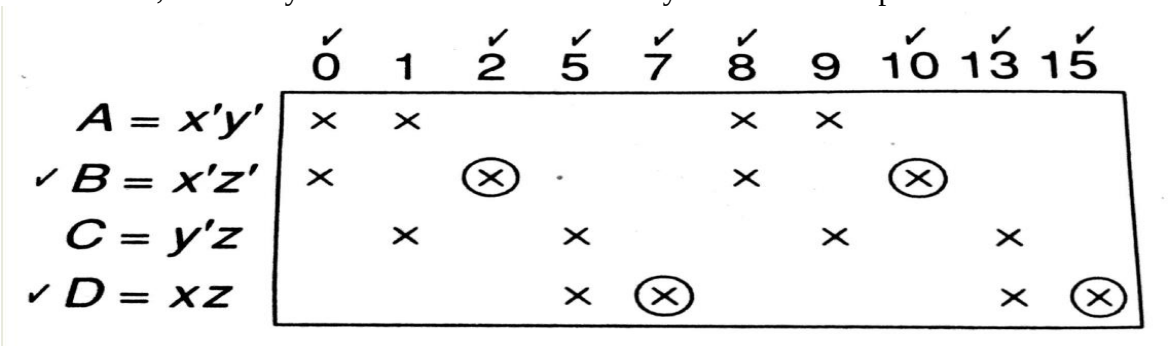
- The prime implicant chart displays pictorially the covering relationships between the prime implicants and minterms of a function.
- It consists of an array of u columns and v rows, where u and v designate the number of minterms in the function and number of prime implicants respectively.
- The entries of $i^{th}$ row in the chart consist of X's placed at its intersections with the coloumns corresponding to minterms covered by the $i^{th}$ prime implicant.




**Essential Rows:**
If any column contains just a single X then the prime implicant corresponding to the row in which this X appears is essential and consequently must be included

**Don't-care Combinations:**

- Don't-care minterms need not be listed as column headings in the prime implicant chart, since they do not have to be covered by the minimal expression.




EX 1:
Reduce the given function using Tabular method
F(W,X,Y,Z)= ∑ (0,5,7,8,9,10,11,14,15)

| | Decimal Number | Binary Representation of Each Term | Decimal Numbers | First Reduction | Decimal Numbers | Second Reduction |
|---|---|---|---|---|---|---|
| Index 0 | 0 | 0000.✓ | 0, 8 | _000 *E* | 8, 9, 10, 11 | 10__ *B* |
| Index 1 | 8 | 1000.✓ | 8, 9 | 100_✓ | 10, 11, 14, 15 | 1_1_ *A* |
| | 5 | 0101.✓ | 8, 10 | 10_0✓ | | |
| Index 2 | 9 | 1001.✓ | 5, 7 | 01_1 *D* | | |
| | 10 | 1010.✓ | 9, 11 | 10_1✓ | | |
| | 7 | 0111.✓ | 10, 11 | 101_✓ | | |
| Index 3 | 11 | 1011.✓ | 10, 14 | 1_10✓ | | |
| | 14 | 1110.✓ | 7, 15 | _111 *C* | | |
| Index 4 | 15 | 1111.✓ | 11, 15 | 1_11✓ | | |
| | | | 14, 15 | 111_✓ | | |

A,B,C,D,E are prime implicants.

*Table of Essential Prime Implicants*

| Prime Implications | Minterms | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 15 |
| *A | | | | | | × | × | ⊗ | × |
| *B | | | | × | ⊗ | × | × | | |
| C | | | × | | | | | | |
| *D | | ⊗ | × | | | | | | |
| *E | ⊗ | | | × | | | | | |

F(W,X,Y,Z)=A+B+D+E=WY+WX'+W'XZ+X'Y'Z'

F(V,W,X,Y,Z)= ∑(0,2,4,5,6,7,8,10,14,17,18,21,29,31) ∑d(11,20,22)

| | Decimal Number | Representation of Each Term | Decimal Numbers | First Reduction | Decimal Numbers | Second Reduction |
|---|---|---|---|---|---|---|
| Index 0 | 0 | 00000✓ | 0, 2 | 000_0✓ | 0, 2, 4, 6 | 00_0 *E* |
| | 2 | 00010✓ | 0, 4 | 00_00✓ | 0, 2, 8, 10 | 0_0_0 *F* |
| Index 1 | 4 | 00100✓ | 0, 8 | 0_000✓ | 2, 6, 10, 14 | 0_ _10 *G* |
| | 8 | 01000✓ | 2, 6 | 00_10✓ | 2, 6, 18, 22 | _0_10 *H* |
| | 5 | 00101✓ | 2, 10 | 0_010✓ | 4, 5, 6, 7 | 001_ _ *I* |
| | 6 | 00110✓ | 2, 18 | _0010✓ | 4, 5, 20, 21 | _010_ *J* |
| Index 2 | 10 | 01010✓ | 4, 5 | 0010_✓ | 4, 20, 6, 22 | _01_0 *K* |
| | 17 | 10001✓ | 4, 6 | 001_0✓ | | |
| | 18 | 10010✓ | 4, 20 | _0100✓ | | |
| | 20 | 10100✓ | 8, 10 | 010_0✓ | | |
| | 7 | 00111✓ | 5, 7 | 001_1✓ | | |
| | 11 | 01011✓ | 5, 21 | _0101✓ | | |
| Index 3 | 14 | 01110✓ | 6, 7 | 0011_✓ | | |
| | 21 | 10101✓ | 6, 14 | 0_110✓ | | |
| | 22 | 10110✓ | 6, 22 | _0110✓ | | |
| Index 4 | 29 | 11101✓ | 10, 14 | 01_10✓ | | |
| Index 5 | 31 | 11111✓ | 10, 11 | 0101_ *A*✓ | | |
| | | | 17, 21 | 10_01 *B*✓ | | |
| | | | 18, 22 | 10_10✓ | | |
| | | | 20, 21 | 1010_✓ | | |
| | | | 20, 22 | 101_0✓ | | |
| | | | 21, 29 | 1_101 *C* | | |
| | | | 29, 31 | 111_1 *D* | | |

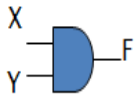
*Table for Finding a Minimal Cover (Example 3.22)*

| | 0 | 2 | 4 | 5 | 6 | 7 | 8 | 10 | 14 | 17 | 18 | 21 | 29 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *A* | | | | | | | | × | | | | | | |
| **B* | | | | | | | | | | ⊗ | | × | | |
| *C* | | | | | | | | | | | | × | × | |
| **D* | | | | | | | | | | | | | × | × |
| *E* | × | × | × | | × | | | | | | | | | |
| **F* | × | × | | | | | ⊗ | × | | | | | | |
| **G* | | × | | | × | | | × | ⊗ | | | | | |
| **H* | | × | | | × | | | | | | ⊗ | | | |
| **I* | | | × | × | | ⊗ | | | | | | | | |
| *J* | | | × | × | | | | | | | | × | | |
| *K* | | | × | | × | | | | | | | | | |

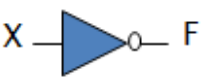
Note: The *don't-care* minterms 11, 20, and 22 do not appear in this table.

F(V,W,X,Y,Z)=VW'Y'Z+VWXZ+V'X'Z'+V'YZ+W'YZ'+V'W'X

**LOGIC GATES:**

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| AND |   | F=XY | X Y F<br>0 0 0<br>0 1 0<br>1 0 0<br>1 1 1 |

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| OR |   | F=X+Y | X Y F<br>0 0 0<br>0 1 1<br>1 0 1<br>1 1 1 |

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| Inverter |   | F=X’ | X F<br>0 1<br>1 0 |

## Universal Gates:

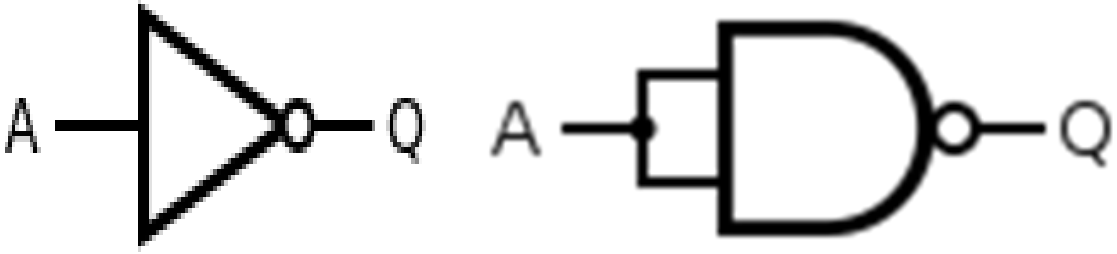
- A gate which can be use to implement any logic gate or any switching functions is called Universal Gate
- NAND and NOR gates are called Universal Gates
- All the other gates can be implemented by using Universal gates

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| NAND |   | F=(XY)' | X Y F<br>0 0 1<br>0 1 1<br>1 0 1<br>1 1 0 |

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| NOR |   | F=(X+Y)' | X Y F<br>0 0 1<br>0 1 0<br>1 0 0<br>1 1 0 |

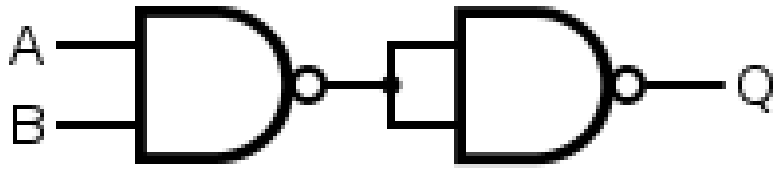
**Implementation of NOT using NAND**

- A NOT gate is made by joining the inputs of a NAND gate together.

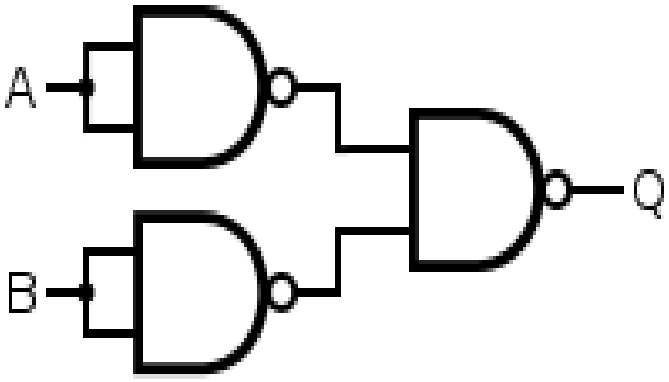



**Implementation of AND using NAND**

- A NAND gate is an inverted AND gate.
- An AND gate is made by following a NAND gate with a NOT gate

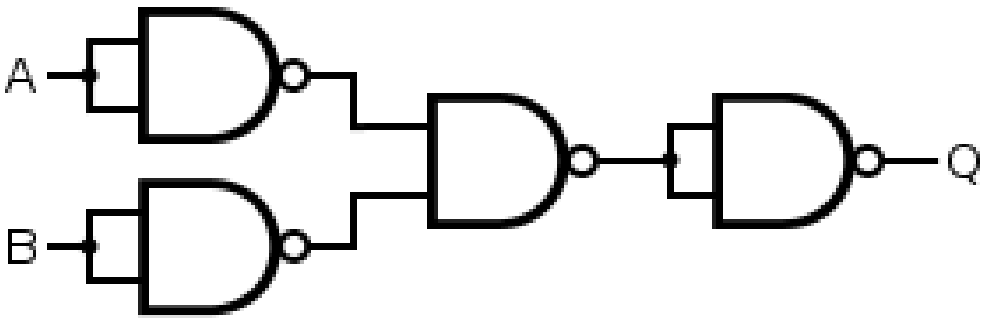



**Implementation of OR gate using NAND**



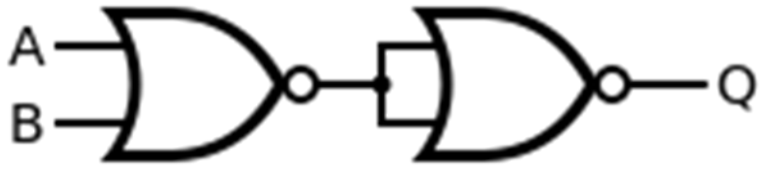

**Implementation of NOR gate using NAND**

- A NOR gate is simply an inverted OR gate. Output is high when neither input A nor input B is high:



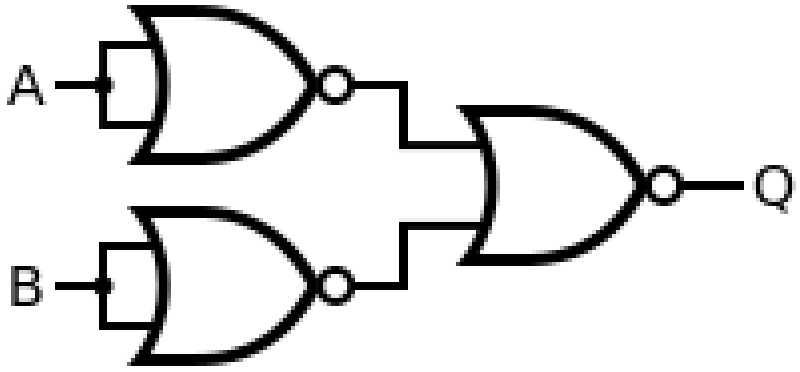

**Implementation of NOT gate using NOR**


**Implementation of OR gate using NOR**

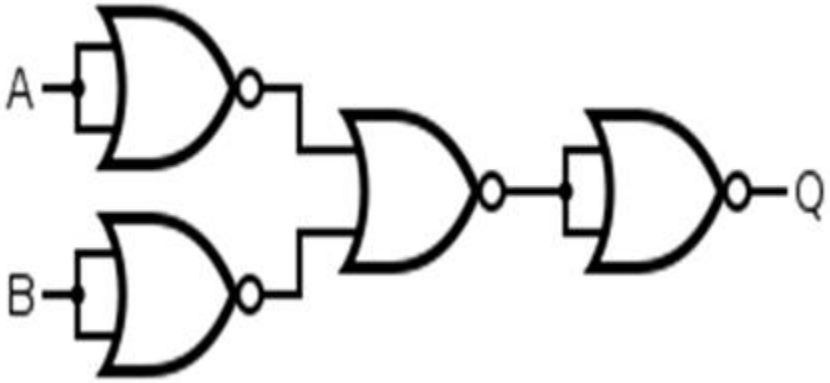



**Implementation of AND gate using NOR**

- An AND gate is made by inverting the inputs to a NOR gate.




**Implementation of NAND gate using NOR**



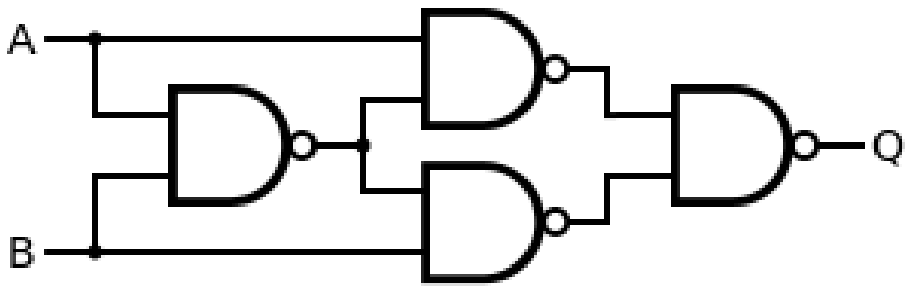
**Special Gates**

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| Exclusive-OR (XOR) |  | F=XY′+X′Y<br>= X ⊕ Y | X Y F<br>0 0 0<br>0 1 1<br>1 0 1<br>1 1 0 |

| NAME | GRAPHIC SYMBOL | ALGEBRIC FUNCTION | TRUTH TABLE |
|---|---|---|---|
| Exclusive-NOR |  | F=XY+X'Y<br>,<br>=X ⊙ Y | X Y F<br>0 0 1<br>0 1 0<br>1 0 0<br>1 1 1 |

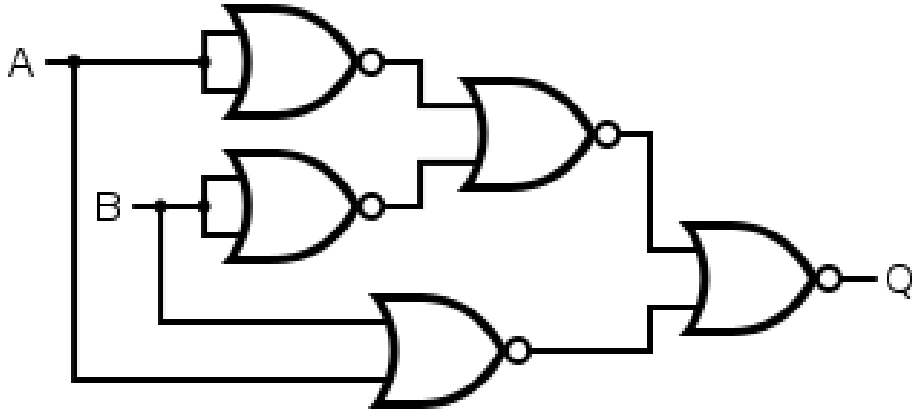
**Implementation of XOR gate using NAND**

- An XOR gate is constructed similarly to an OR gate, except with an additional NAND gate inserted such that if both inputs are high, the inputs to the final NAND gate will also be high.

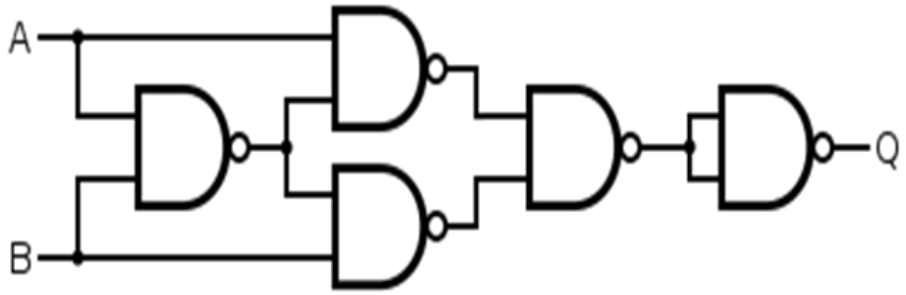



**Implementation of XOR gate using NOR**

- An XOR gate is made by connecting the output of 3 NOR gates (connected as an AND gate) and the output of a NOR gate to the respective inputs of a NOR gate.




**Implementation of XNOR gate using NAND**




**Implementation of XNOR gate using NOR**

- An XNOR gate can be constructed from four NOR gates implementing the expression "(A NOR N) NOR (B NOR N) where N = A NOR B". This construction has a propagation delay three times that of a single NOR gate, and uses more gates.