2017

MODULE : 2

COMBINATIONAL LOGIC DESIGN

Boolean Algebra

- Algebra refers to rules of a no. system
- Operations: $\wedge$ $\vee$ $\neg$

Axioms → truth of a system

1. Axiom 1 : Logical values
   $a = 0$ if $a \neq 1$ and $a = 1$ if $a \neq 0$
2. Axiom 2 : Definition of logical negation
   if $a = 0$, then a compliment is 1 and
   if $a = 1$, then $a' = 0$
3. Axiom 3 : Definition of a logical product
   $a \cdot b = 1$ if $a = b = 1$ and $a \cdot b = 0$ otherwis
4. Axiom 4 : Definition of a logical sum
   $a + b = 1$ if $a = 1$ or $b = 1$ and $a + b = 0$
   otherwise.
5. Axiom 5 : Logical precedence
   NOT PRECEDES AND PRECEDES OR

   eg : $A' \cdot B + C$

   3rd, 2nd, 1st evaluates

Theorem

Boolean algebra accepts only two values. Therefore input possibilities are bounded. So a theorem can be proved using bounded input code. This is known as proof by exhaustion

1. De-Morgan's theorem of Duality

An algebraic equality will remain through if all zeroes and ones are interchanged and all

AND and OR functions are interchanged.

- Taking the dual of a positive logic function will produce equivalent function using -ve logic if the original overall precedence is maintained.

eg: original

$A \cdot 0 = 0$

proof: $0 \cdot 0 = 0$

$1 \cdot 0 = 0$

Dual

$A + 1 = 1$

$0 + 1 = 1$

$1 + 1 = 1$

- Through the above process of proof by exhaustion we have proved that $A \cdot 0 = 0$ is a true statement
- We have proved through proof by exhaustion that $A + 1 = 1$ is a true statement and that the theory of duality is held.

Two reasons to use duality

1. Doubles the impact of a theorem
2. It can be used to convert b/w +ve and -ve logic.

Eg: original : $F = A \cdot B$

A	B	F
L	L	L
L	H	L
H	L	L
H	H	H

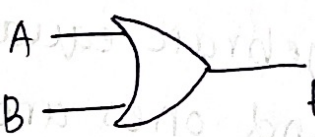
+ve logic ⇒ F

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

-ve logic ⇒ F_D

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

Dual : $F_D = A + B$



eg: ie, 2) original : $(A \cdot B)+C$ Dual: $(A+B) \cdot C$

2. Identity

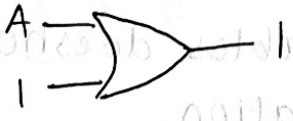
ORing any variable with a logic zero will give the original variable, ie $A+0=A$

Dual is $A \cdot 1 = A$:

3. Null element

OR ing any variable with a logic 1 will give logic 1.

ie, $A+1=1$

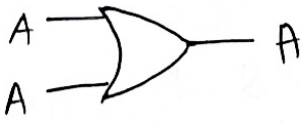


Dual: $A \cdot 0 = 0$

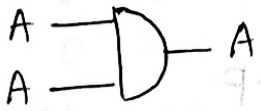
4. Idempotent

ORing a variable with itself will you the same variable.

ie, $A + A = A$



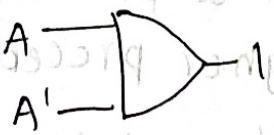
Dual : $A \cdot A = A$



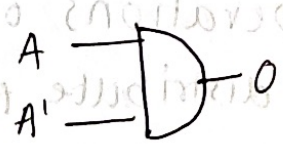
5. Compliments

ORing a variable with its compliment, we get 1.

ie, $A + A' = 1$



Dual: $A \cdot A' = 0$



6. Involution

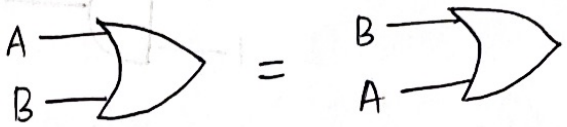
Taking double compliment of a variable will result in the original variable.

31/7

7. Commutative Property

Changing the order of variables in an OR operation doesnot change the end result

eg: $A+B = B+A$

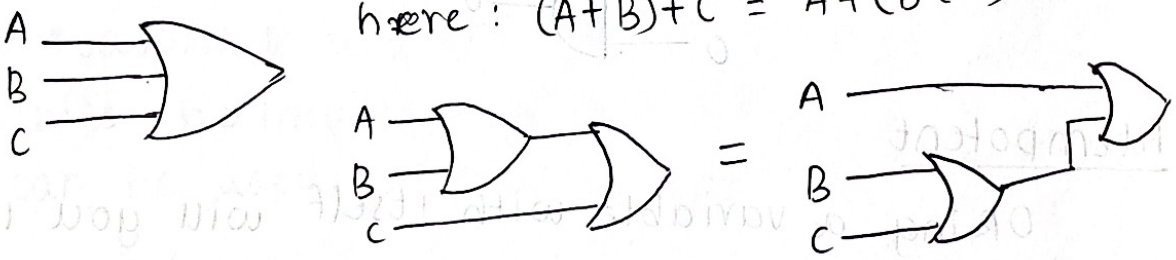


Dual: $A \cdot B = B \cdot A$

8. Associative Property

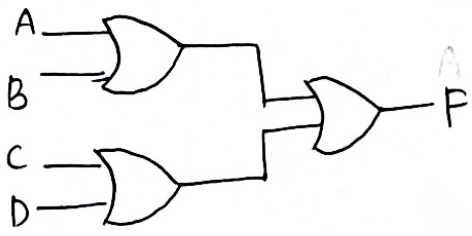
Grouping of variables doesnot impact the result of an OR operation.

- here: $(A+B)+C = A+(B+C)$



- $F = A+B+C+D$

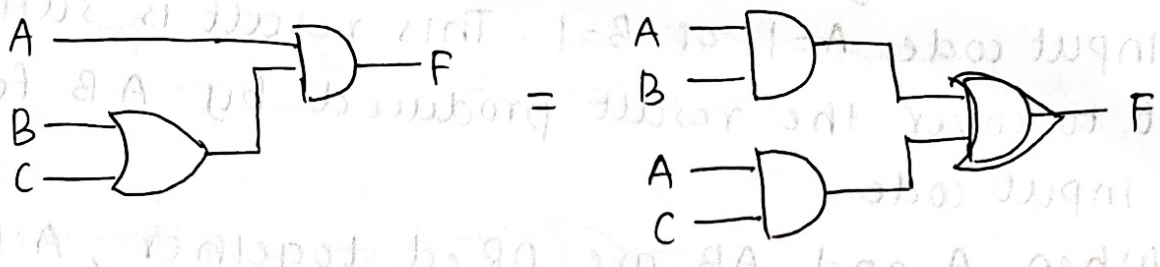
  ie, $(A+B)+(C+D)$



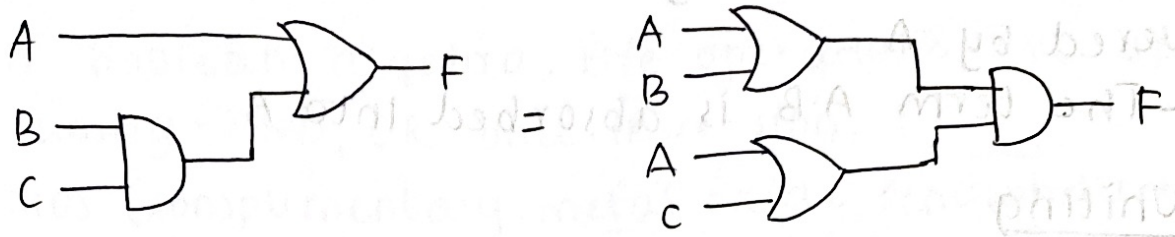
9. Distributive property

An operation on a paranthesized operation or operations or higher precedence operator will distribute through each term.

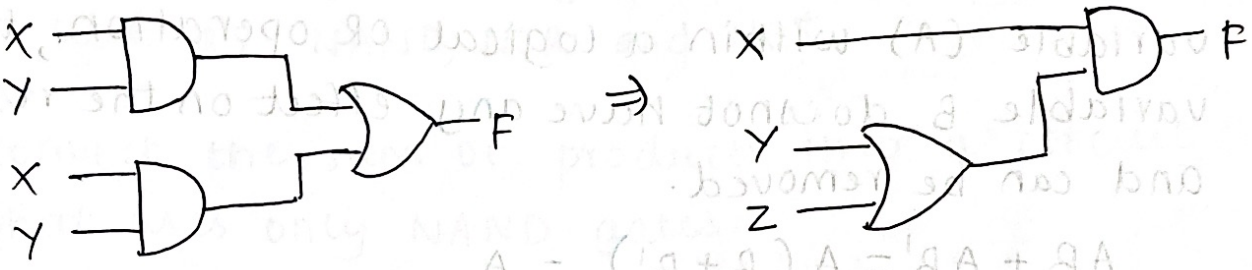
$A \cdot (B + C) = A \cdot B + A \cdot C$



Dual: $A + (B \cdot C) = (A + B) \cdot (A + C)$



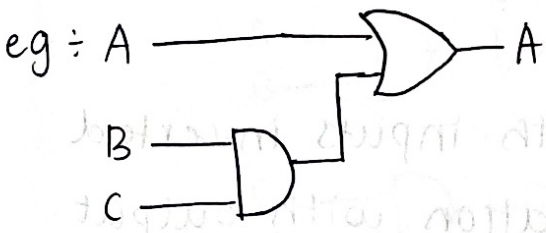
Eg: $F = X \cdot Y + X \cdot Z \Rightarrow F = X \cdot (Y + Z)$



10. Absorption

when a term within a logic expression produces the same output as another term, the second term can be removed without affecting the result

eg:



A	B	A + A·B	A	AB
0	0	0	0	0
0	1	0	0	0
1	0	1	1	0
1	1	1	1	1

same output

Hence, $A + A \cdot B = A$

Dual: $A \cdot (A + B)$

- The above ~~gat~~ table A will produce 1 for the input code A=1 & B=1. This result is sufficient to cover the result produced by A·B for this input code.

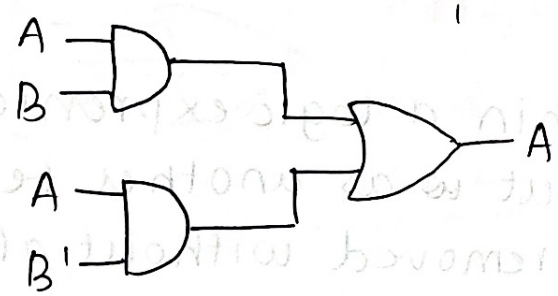
- When A and AB are ORed together, A·B becomes unneccesary becos the output is fully covered by A.

- The term A·B is absorbed into A

11. Uniting

when a variable B and its complement B' appears multiple product terms with a common variable (A) within a logical OR operation, the variable B doesnot have any effect on the result and can be removed.

$$AB + AB' = A(\underbrace{B + B'}_{1}) = A$$

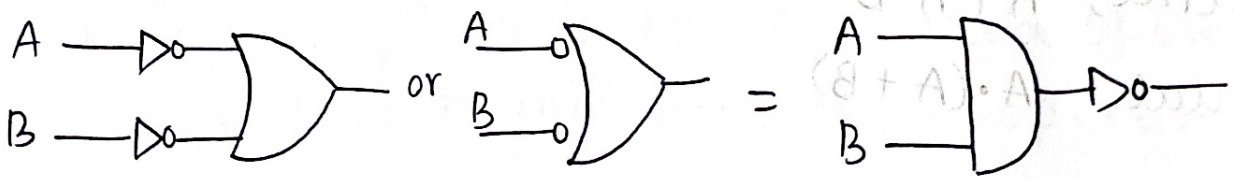


Dual: $(A+B)\cdot(A+B') = A+(\underbrace{B\cdot B'}_{0}) = A$

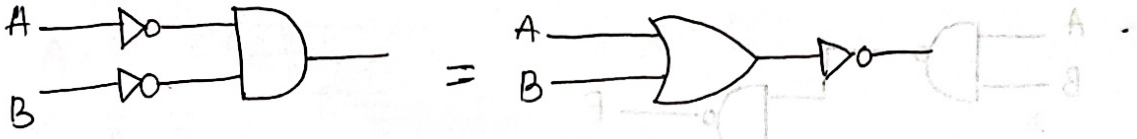
12. De morgan's theorem

An OR operation with both inputs inverted is equivalent to an AND operation with output inverted.

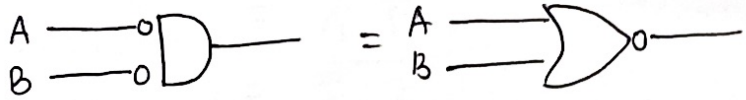
$$(A'+B') = (A\cdot B)'$$



5/8 Dual : $F = A' \cdot B' = (A + B)'$



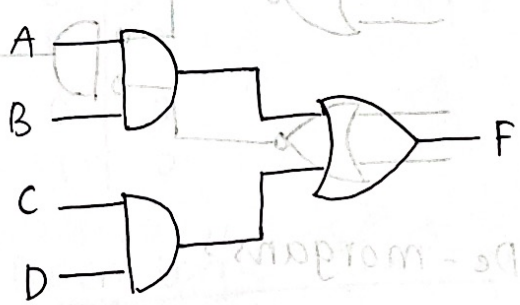
(OR)



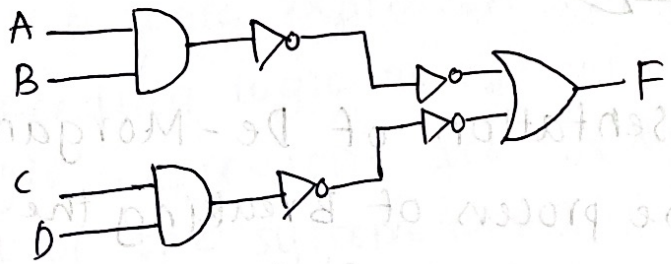
- In boolean algebra, the operations are defined using AND, OR and Inversion.
- CMOS (Complimentary metal oxide semiconductor) doesnot directly implement AND and OR gates but it can only directly implement -ve type gates such as NAND, NOR and NOT.

Ex: Convert the sum of products into a circuit that uses only NAND gates.

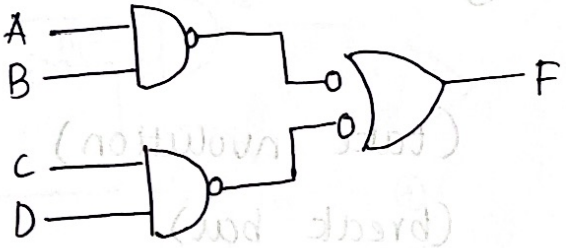
(a) $F = A \cdot B + C \cdot D$ ie,



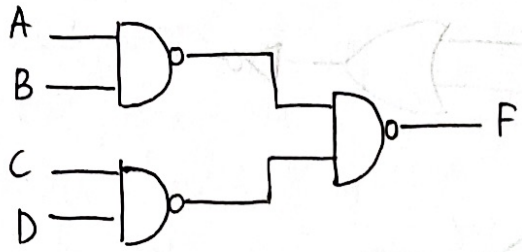
Take double compliment (Involution)



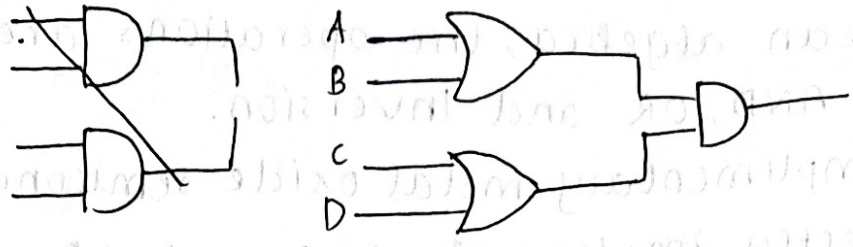
ie,

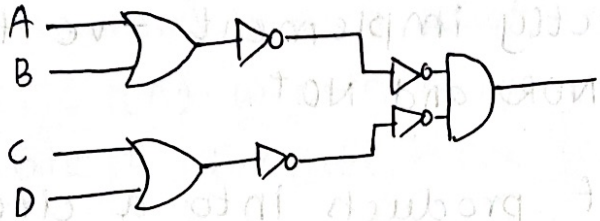


De-morgan's law:

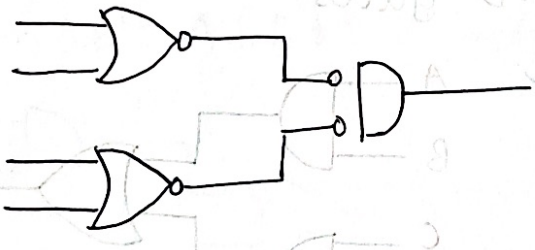


(b) $F = (A+B)\cdot(C+D)$

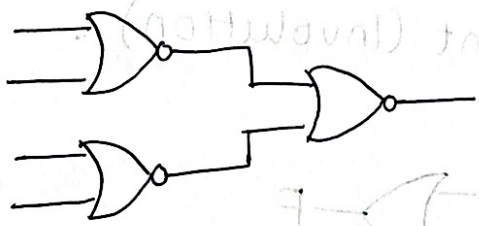




ie,



De-morgans's



Algebraic Representation of De-Morgan's Theorem using the procers of Breaking the Bar and Flipping the sign

1. $\bar{A} + \bar{B} = \overline{A \cdot B}$

$$\bar{A} + \bar{B} = \overline{\overline{\bar{A} + \bar{B}}} \quad \text{(take involution)}$$

$$= \overline{\bar{\bar{A}} + \bar{\bar{B}}} \quad \text{(break bar)}$$

$$= \overline{\bar{\bar{A}} \cdot \bar{\bar{B}}} \quad \text{(flip sign)}$$

$= \overline{A \cdot B}$

2. $\bar{A} \cdot \bar{B} = \overline{(A+B)}$

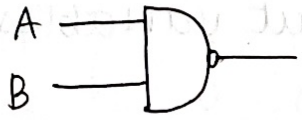
$\bar{A} \cdot \bar{B} = \overline{\bar{A} \cdot \bar{B}}$

$= \overline{\bar{A} \cdot \bar{B}}$

$= \overline{A+B}$

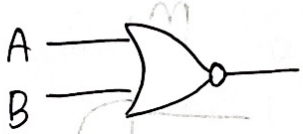
$= \overline{A+B}$

Configuring NAND gate as an Inverter



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

0	1
1	0

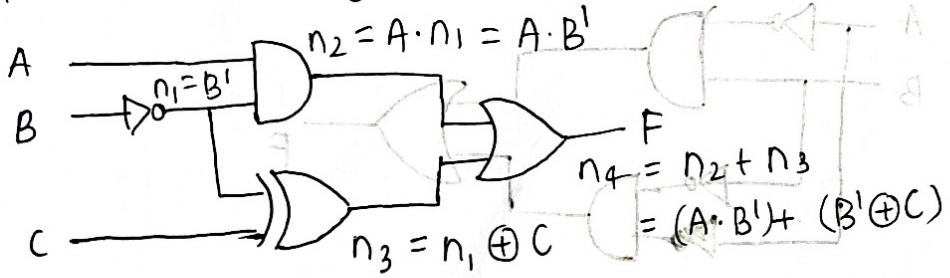


A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

0	1
1	0

Finding logic expression from logic diagram

1. Label each intermediate node in the system
2. write the logic expression for each node based on preceding logic operation(s)
3. The logic expressions are written until the output of the system is reached.

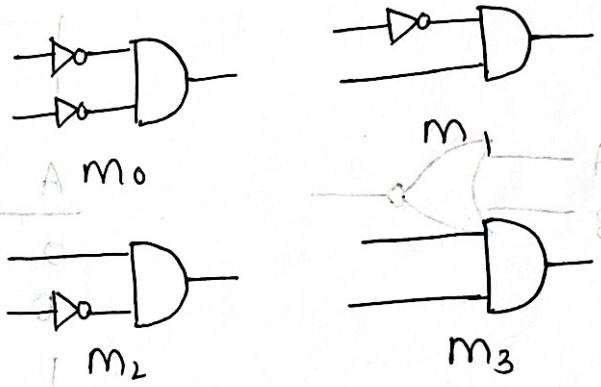


Combinational Logic Synthesis

Canonical sum of products.

- Based on minterms
- A minterm is a product term, ie an AND operation that will be true for one and only one input code.
- Each minterm must contain every literal (an input variable which may or may not be complimented).
- Compliments are applied to the input variables to create current logic.

Row	A	B	minterm
0	0	0	$m_0 = A' \cdot B'$
1	0	1	$m_1 = A' \cdot B$
2	1	0	$m_2 = A \cdot B'$
3	1	1	$m_3 = A \cdot B$



- Creating a canonical sum of products logic circuit using minterms:

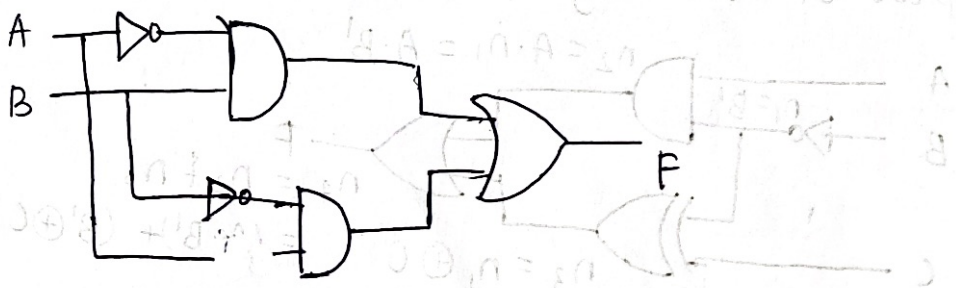
Row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

$m_1 = A' \cdot B$

$m_2 = A \cdot B'$

$m_1 + m_2 = A' \cdot B + A \cdot B'$

Gate:



Minterm list (Σ)

- Lists the row numbers that correspond to an output 1 in the truth table.

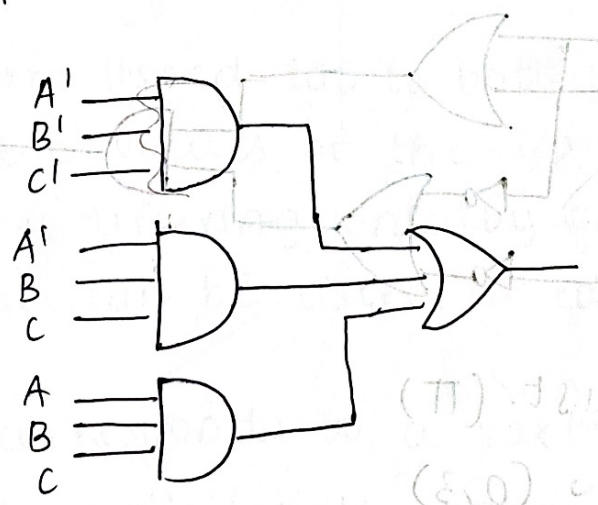
$F = \sum_{A,B} (1,2)$. (rows 1 and 2 in prev truth table)

Q.) Create equivalent functional representation from a minterm list $F = \sum_{A,B,C} (0,3,7)$

→

Row	A	B	C	F
0	0	0	0	$m_0 = A'B'C'$
1	0	0	1	
2	0	1	0	
3	0	1	1	$m_3 = A'BC$
4	1	0	0	
5	1	0	1	
6	1	1	0	
7	1	1	1	$m_7 = ABC$

Theoritically :

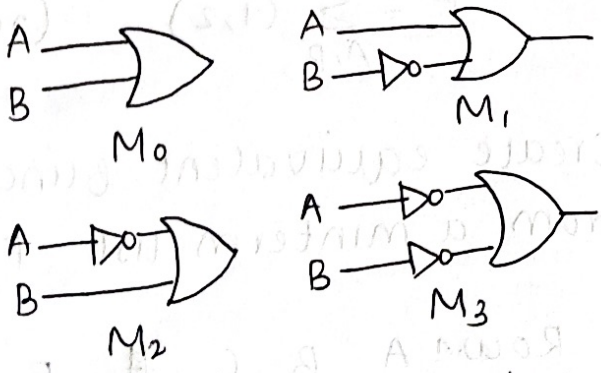


PTO →

Canonical product of sum : Maxterm

- Each maxterm is a sum term that produces a zero for one and only one input code.

Row	A	B	maxterm
0	0	0	$M_0 = A + B$
1	0	1	$M_1 = A + B'$
2	1	0	$M_2 = A' + B$
3	1	1	$M_3 = A' + B'$



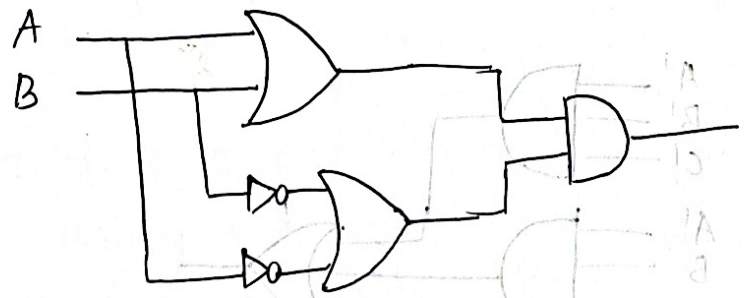
- Creating a canonical POS logic circuit using maxterms:

Row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

$M_0 = A + B$

$M_3 = A' + B'$

$M_0 \cdot M_3 = (A + B) \cdot (A' + B')$



Maxterm list (Π)

$F = \Pi_{A,B}(0,3)$

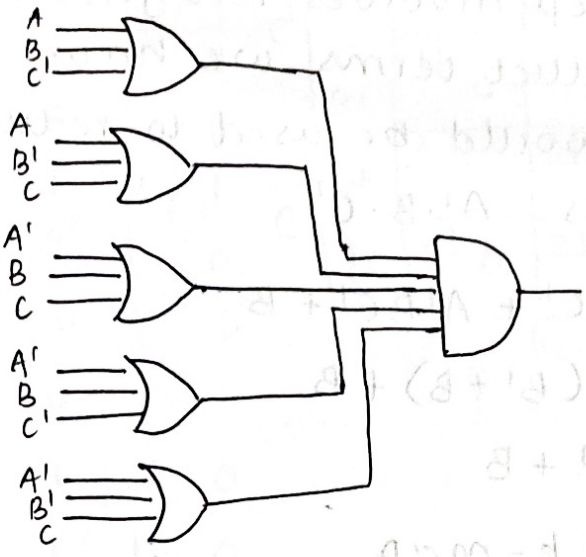
Q. Create equivalent functional representation from a maxterm list, $F = \Pi_{A,B,C}(1,2,4,5,6)$

→

Row	A	B	C	F
0	0	0	0	~~$M_0 = A+B+C$~~
1	0	0	1	$M_1 = A+B+C'$
2	0	1	0	$M_2 = A+B'+C$
3	0	1	1	

4	1	0	0	$M_4 = A' + B + C$
5	1	0	1	$M_5 = A' + B + C'$
6	1	1	0	$M_6 = A' + B' + C$
7	1	1	1	

Theoretically :



k-Map (karnaugh-Map)

In a k-map;

1. Each cell corresponds to a row in the truth table.
2. The variables are listed top to bottom
3. Lists all possible values of the i/p variables along the sides differing only by one bit
4. The row number can be listed in each cell for clarity.
5. Each i/p code corresponds to a particular colomn or row and the literals can be written outside the k-map.

7/8 ## Algebraic minimisation

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

12/8 **Rule 2**

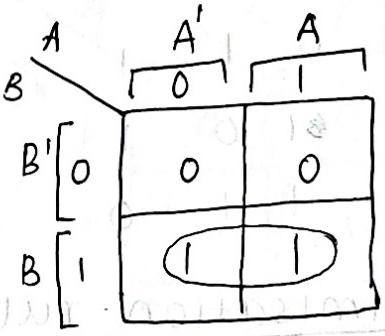
Create a product term for each prime implicant using the following rules.

1. If a circle covers a region where the input variable is 1, then include it in the product term as uncomplimented.
2. The circle covers a region where the input variable is 0, then include it in the product term as complimented.
3. If the circle covers an area where the input variable is both 0 and 1, then the variable is excluded from the product term.

K-map minimisation (SOP)

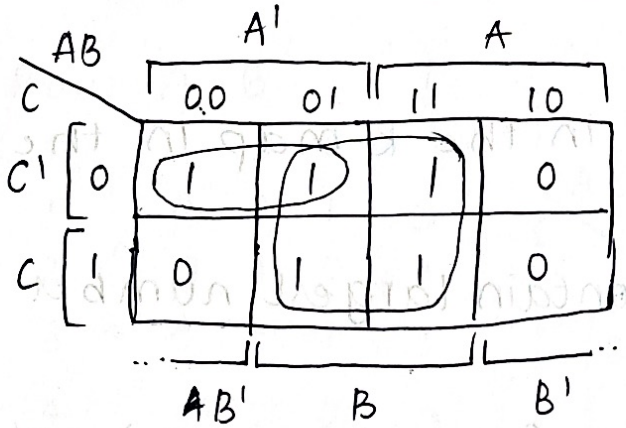
a)

Row	A	B	F
0	0	0	0
1	0	1	1
2	1	0	0
3	1	1	1



$F = B$

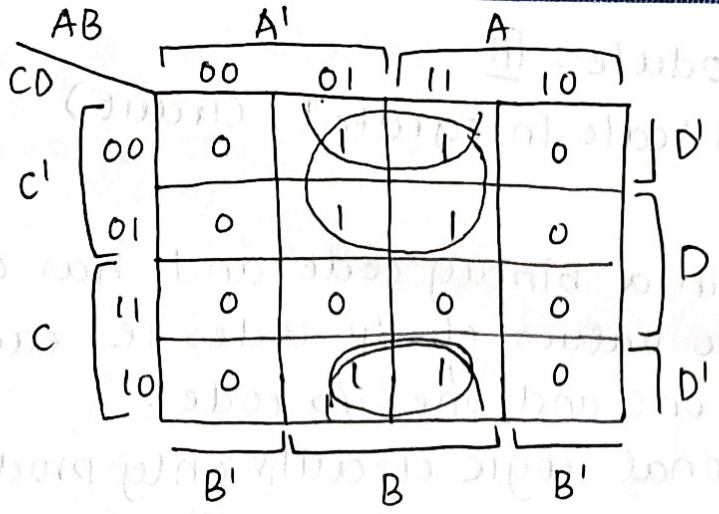
b)



- for circle with four 1s
  col : B row : —
- for circle with two 1s
  col : A' row : C'

$\therefore F = B + A'C'$

c)



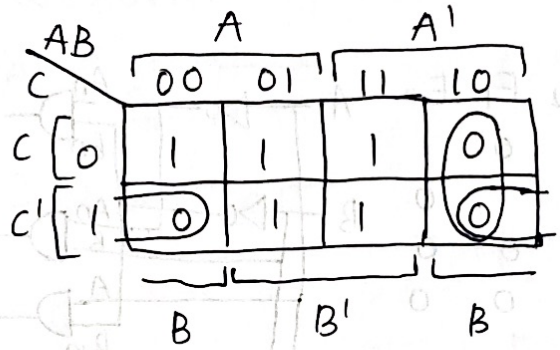
(edge coverage)

$F = BC' + BD'$

K-map minimisation (POS)

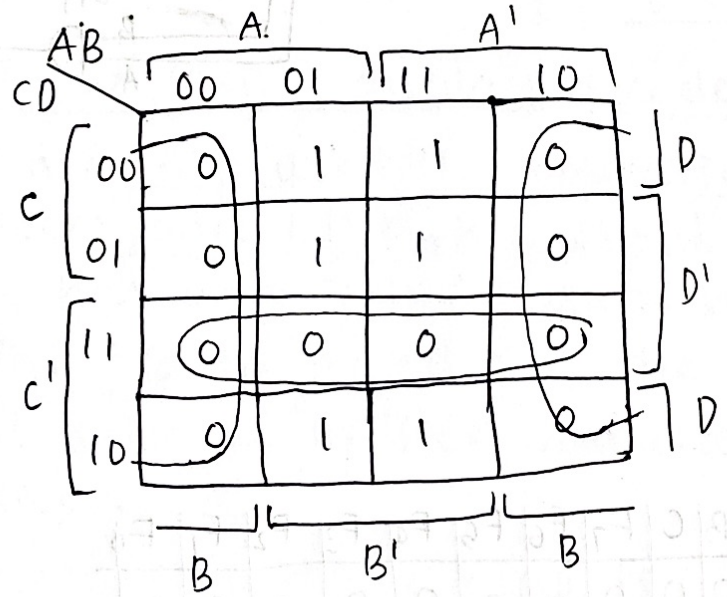
- (Reverse of SOP)

a)



$F = (B + A') \cdot (B + C')$

b)



$F = (C' + D') \cdot B$

Module: 2

Modelling Concurrent Functionality in Verilog

- In normal programming languages the lines of codes are executed sequentially as they appear in the source file.
- In verilog, the lines of codes represent behaviour of real hardware. Therefore assignments are executed concurrently.

Continuous Assignment

Verilog uses the keyword 'assign' to denote continuous assignment. After the keyword 'assign' an assignment is made using '=' symbol.

- The LHS of the assignment is the target signal and must be a net type and RHS of the assignment contains i/p arguments which can contain regs, nets, constants or operators.
- Continuous assignment models combinational logic.

Eg:-
```
assign F1 = A;
assign F2 = 1'b0;
assign F3 = 1'b1;
```

F_1 is updated any time A changes where A is a signal.

- assign X = A;
  assign Y = B;
  assign Z = C;

~~assig~~ The signal assignment in the above code happens at the same time ie, assign concurrently

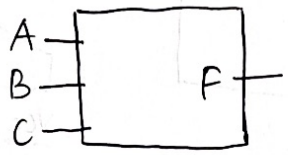
- assign A = B;
  assign B = C;

In verilog, the signal assignments are C to B and B to A will take place at the same time, ie during synthesis the signal B will be eliminated from the design.

Continuous Assignment with Logical Operator

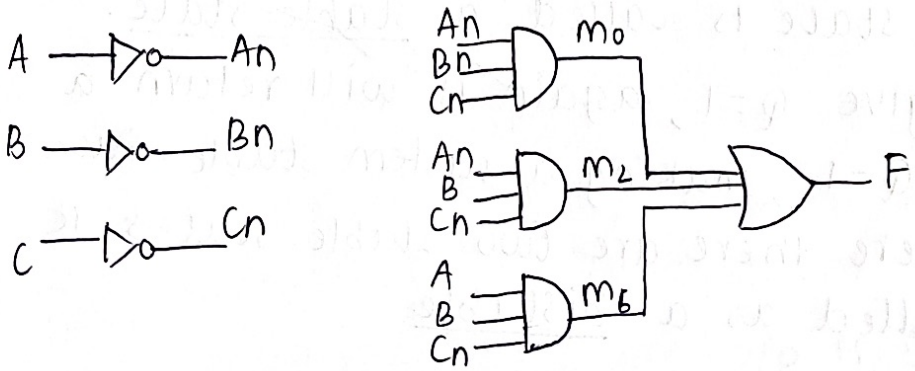
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

SystemX.v



$F = \sum_{A,B,C} (0,2,6)$

$= A'B'C' + A'BC' + ABC'$



module systemX (output wire F, input wire A,B,C);

wire An, Bn, Cn;

wire m0, m2, m6;

```
assign An = ~A;
assign Bn = ~B;
assign Cn = ~C;
assign m0 = An & Bn & Cn;
assign m2 = An & B & Cn;
assign m6 = A & B & Cn;
assign F = m0 | m2 | m6;
end module
```