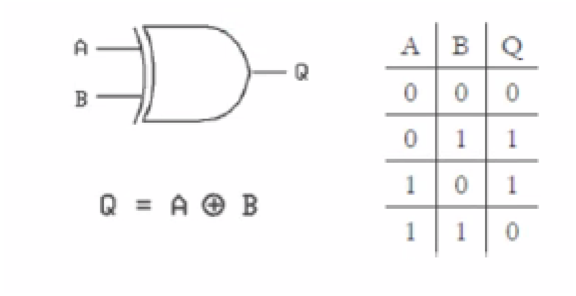
Analyzes basic digital logic gates in term of their unique functionalities.

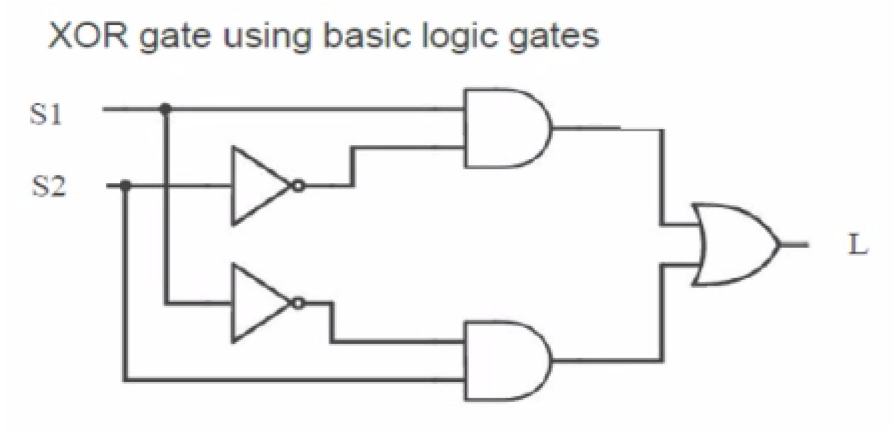
Basic Logic Gates

XOR

Exclusive OR, if inputs are different to each other, this returns 1, else 0



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



- Truth table with 3 inputs

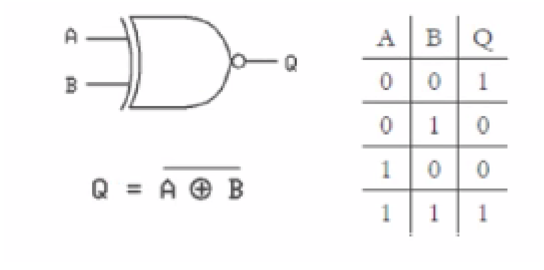
Here when you get 3 inputs (S1, S2,S3) the method you simply the inputs are as `(S1 xor S2) xor S3`
The output of `S1 xor S2` is then xored with `S3`. All 3 aren't xored at the same time.

Truth table for three input XOR gate

INPUTS				Final Output
S1	S2	S3	S1 ⊕ S2	S1 ⊕ S2 ⊕ S3
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

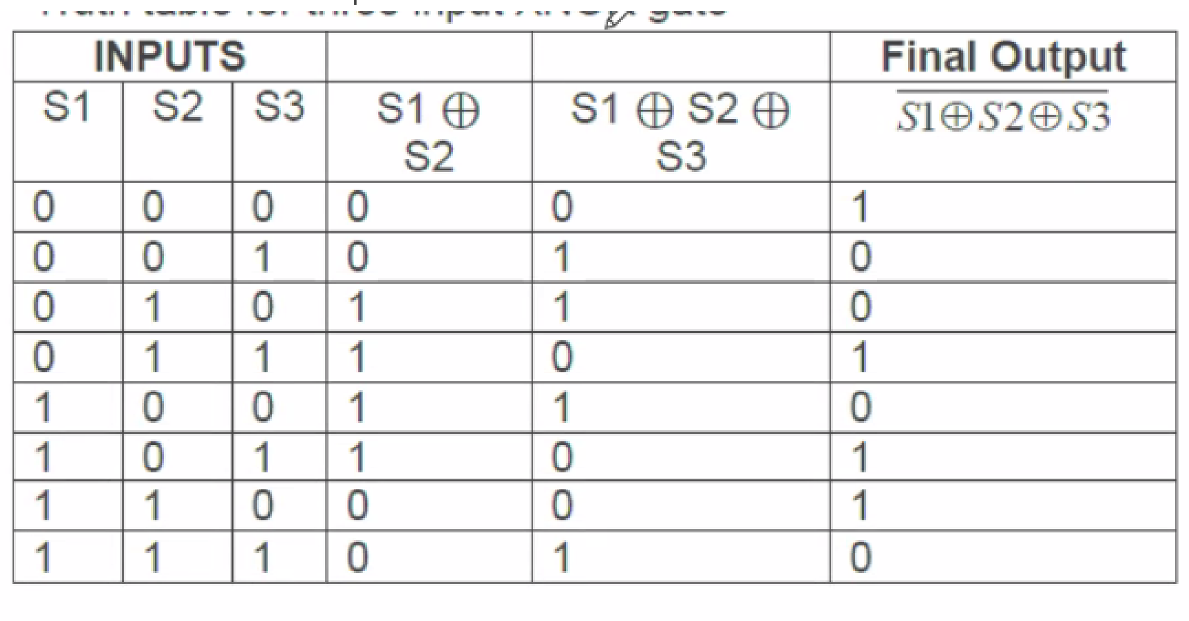
XNOR

Complement for XOR



A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

- Truth table with 3 inputs



INPUTS					Final Output
S1	S2	S3	$S1 \oplus S2$	$S1 \oplus S2 \oplus S3$	$\overline{S1 \oplus S2 \oplus S3}$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	1	0

Universal Gates

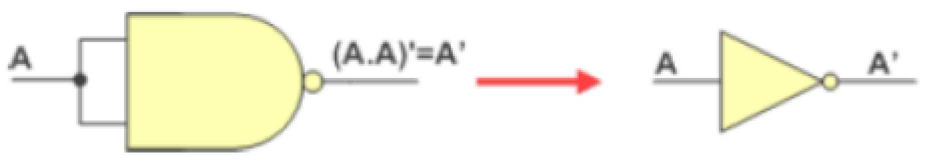
There are 2 universal gates
1. NAND
2. NOR

These 2 gates can be used to create all the other gates

NAND

1. Creating NOT from NAND

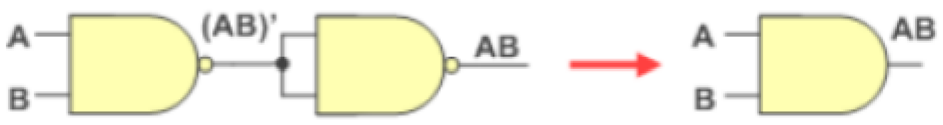
```
NOT(A.A) = NOT(A)
```



A	A	NOT(A.A)		A	NOT(A)
1	1	0		1	0
0	0	1		0	1

2. Creating AND from NAND

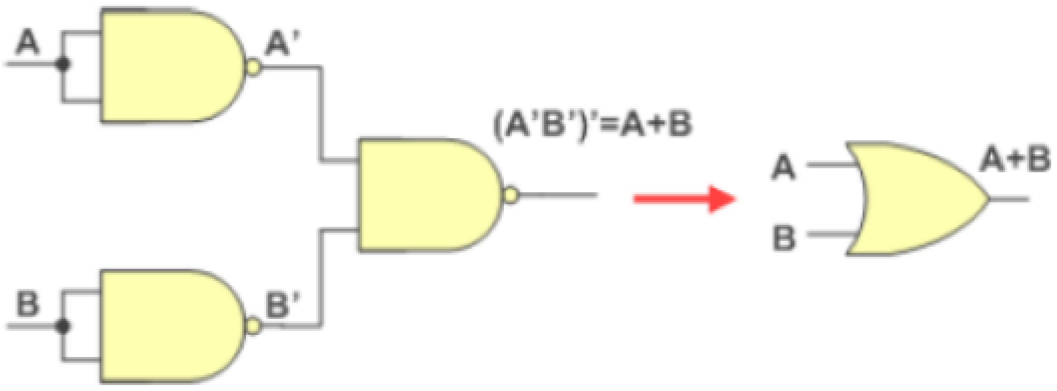
```
NOT( NOT(A.B) . NOT(A.B) ) = A.B
```



A	B	NOT(A.B)	NOT(NOT(A.B) . NOT(A.B))	=	A.B
1	0	1	0		0
0	1	1	0		0
1	1	0	1		1
0	0	1	0		0

3. Creating OR from NAND

```
NOT( NOT(A.A) . NOT(B.B)) = A+B
```

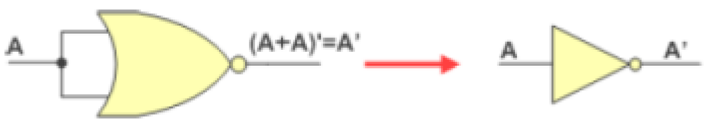


A	NOT(A)		B	NOT(B)		(NOT(A).NOT(B))	=	A+B
1	0		1	0		0		0
0	1		0	1		1		1
1	0		0	1		1		1
0	1		1	0		1		1

NOR

1. Creating NOT from NOR

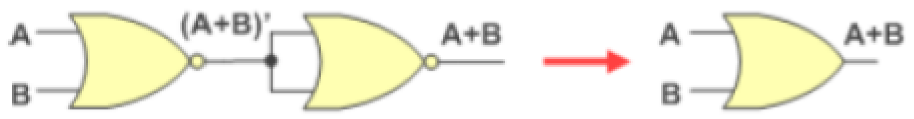
```
NOT(A+A) = NOT(A)
```



A	NOT(A+A)	=	NOT(A)
1	0		0
0	1		1

2. Creating OR from NOR

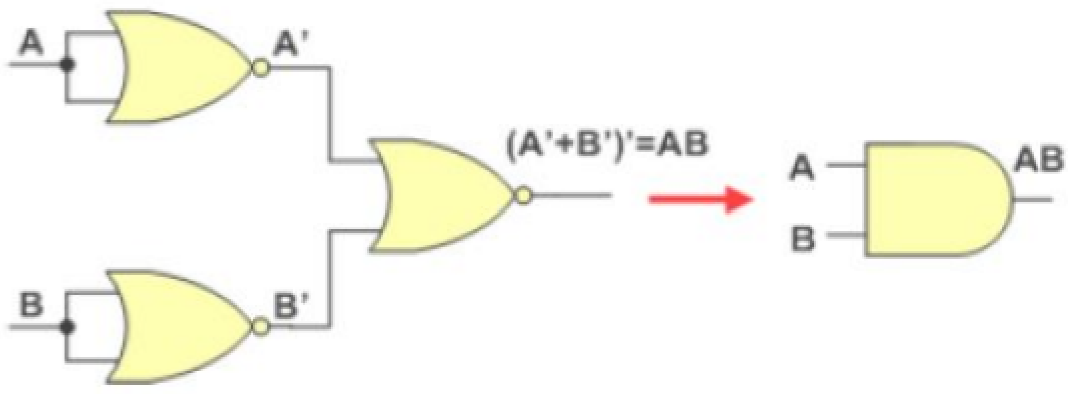
```
NOT(NOT(A+B)+NOT(A+B)) = A+B
```



A	B	NOT(A+B)	NOT(NOT(A+B)+NOT(A+B))	=	A+B
1	0	0	1		1
0	1	0	1		1
1	1	0	1		1
0	0	1	0		0

3. Creating AND from NOR

```
NOT(NOT(A)+NOT(B)) = A.B
```



A	B	NOT(A)	NOT(B)	NOT(NOT(A)+NOT(B))	=	A.B
1	0	0	1	0		0
1	1	0	0	1		1
0	0	1	1	0		0
0	1	1	0	0		0

Summary

1. NAND
   - NOT = NAND

- `NOT(A) = NOT(A.B)`
- AND = NAND(NAND . NAND)
  - `(A.B) = NOT( NOT(A.B) . NOT(A.B) )`
- OR = NAND(NAND . NAND)
  - `(A+B) = NOT( NOT(A.A) . NOT(B.B))`

2. NOR
   - NOT = NOR
     - `NOT(A) = NOT(A+B)`
   - OR = NOR(NOR + NOR)
     - `(A+B) = NOT( NOT(A+B) + NOT(A+B) )`
   - AND = NOR(NOR + NOR)
     - `(A.B) = NOT( NOT(A+A) + NOT(B+B) )`

When building circuits in real life, the following elements should be considered

1. Cost
2. Power Consumption
3. Heat generation
4. Physical size
5. Inter-communication speed - Between 2 ICs
6. Intra-communication speed - Inside one IC

Question:

1. When constructing logic circuits, why do industries prefer to use universal gates rather than using basic gates?

> Using universal gates instead of basic gates allows circuit designers to simplify the design process by reducing the number of gate types needed. This reduces the complexity of the circuit, making it easier to design, test, and debug.

02 - Standard forms in Boolean expressions

Standard forms in Boolean expressions

Boolean algebra

Just like normal algebra, in boolean algebra as well we can do different operations. There are 3 different operators we can use in boolean algebra
1. bar (`-`) - Not or negation (`A'`)
2. dot (`.`) - Anda
3. plus (`+`) - Or

Only these 3 basic operations are used with boolean algebra, if you want to make an operation like XOR we need to use these to make it. XOR like operations can't be applied directly.

Boolean Laws

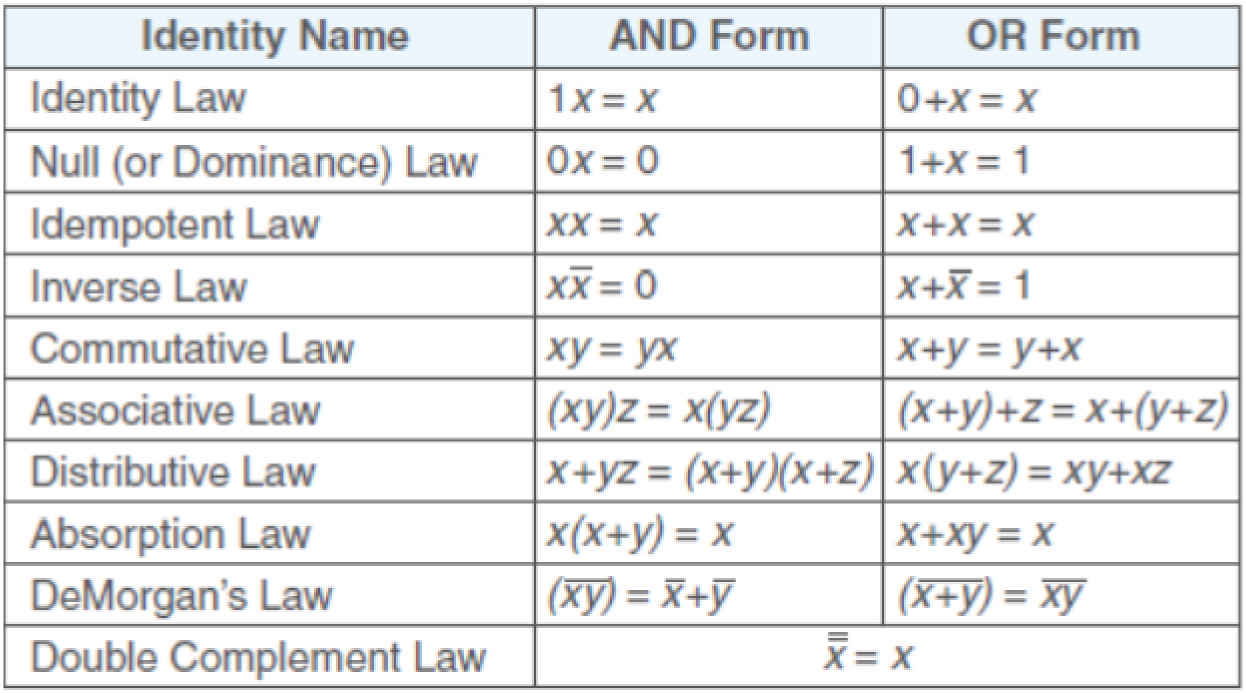
Frequently, a Boolean expression is not in its simplest form

```
2x + 6x
```

So we can simplify them to make them be in the simplest form

```
2x + 6x = 8x
```

Like we did with mathematical expression, we can simplify boolean expressions too. For that some laws should be used.



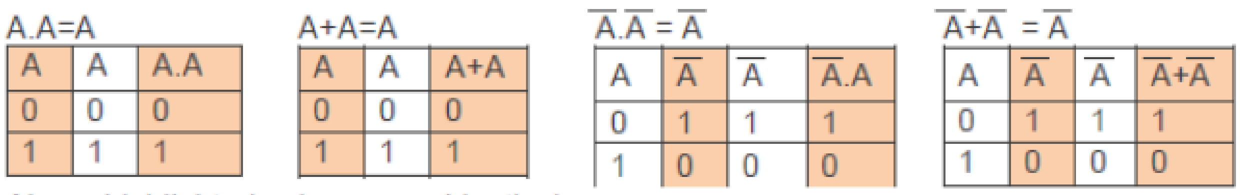
Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

Table is just to refer, not to study. Following laws should be studies.

Idempotent Law

If same input is put into an operation, the simplified answer is the input itself.

```
A . A  = A
A + A  = A
A' . A' = A'
A' + A' = A'
```



A.A=A

A	A	A.A
0	0	0
1	1	1

A+A=A

A	A	A+A
0	0	0
1	1	1

$\bar{A}.\bar{A} = \bar{A}$

A	$\bar{A}$	$\bar{A}$	$\bar{A}.\bar{A}$
0	1	1	1
1	0	0	0

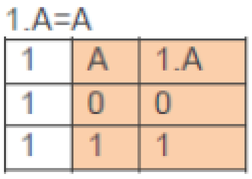
$\bar{A}+\bar{A} = \bar{A}$

A	$\bar{A}$	$\bar{A}$	$\bar{A}+\bar{A}$
0	1	1	1
1	0	0	0

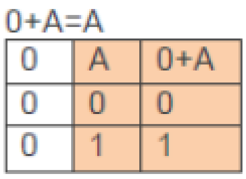
Above highlighted columns are identical.

Identity Law

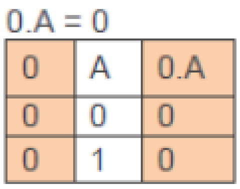
1	A	1.A =A		1	A	1+A=1
1	1	1		1	1	1
1	0	0		1	0	1
0	A	0.A =0		0	A	0+A=A
0	1	0		0	1	1
0	0	0		0	0	0


1.A=A

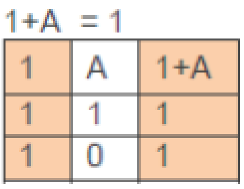
1	A	1.A
1	0	0
1	1	1


0+A=A

0	A	0+A
0	0	0
0	1	1


0.A = 0

0	A	0.A
0	0	0
0	1	0

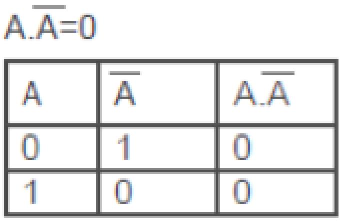

1+A = 1

1	A	1+A
1	1	1
1	0	1

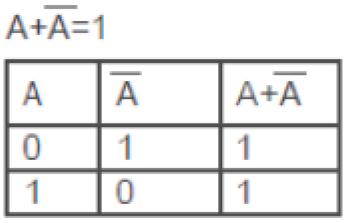
Above highlighted columns are identical.

Inverse/Complement Law

A	A'	A.A'=0	A+A'=1
1	0	0	1
0	1	0	1


$A.\overline{A}=0$

A	$\overline{A}$	$A.\overline{A}$
0	1	0
1	0	0

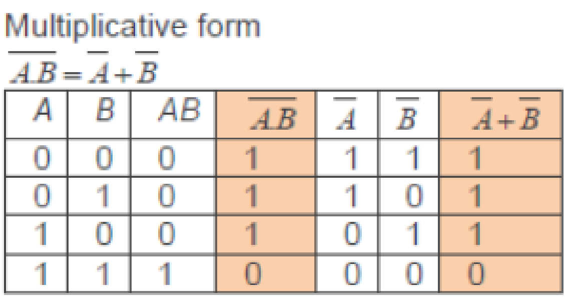

$A+\overline{A}=1$

A	$\overline{A}$	$A+\overline{A}$
0	1	1
1	0	1

De Morgan’s Law

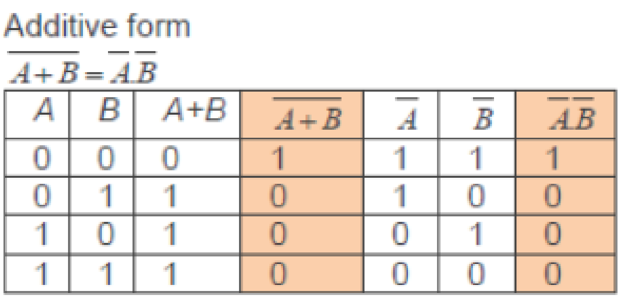
Break the bar and change the operator

```
(A.B)' = A' + B'
(A+B)' = A' . B'
```


Multiplicative form

$\overline{AB}=\overline{A}+\overline{B}$

A	B	AB	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0


Additive form

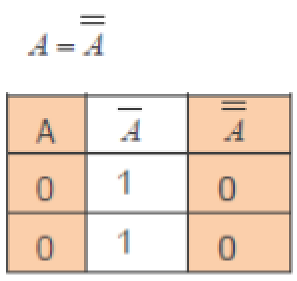
$\overline{A+B}=\overline{A}\,\overline{B}$

A	B	A+B	$\overline{A+B}$	$\overline{A}$	$\overline{B}$	$\overline{A}\,\overline{B}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Double Complement Law

A	A'	A''
1	0	1

A	A'	A''
0	1	0

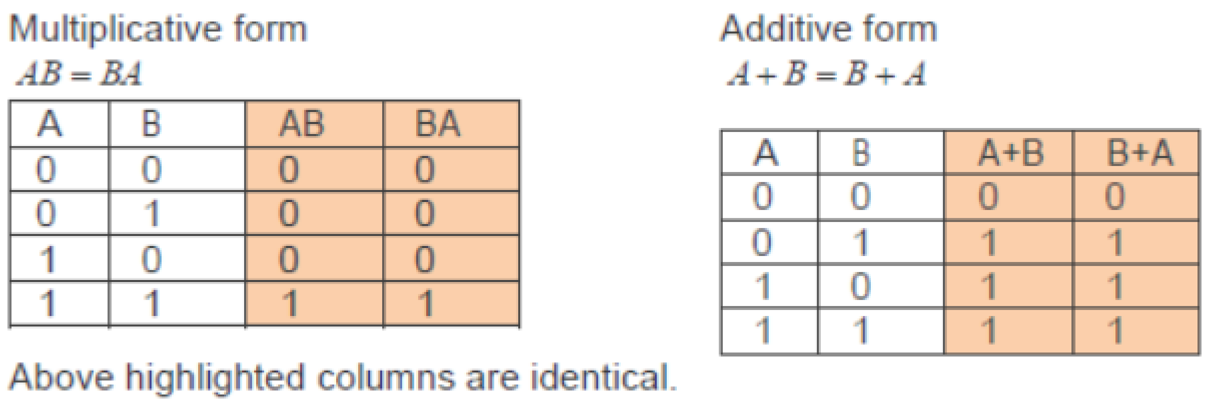


$A = \overline{\overline{A}}$

A	$\overline{A}$	$\overline{\overline{A}}$
0	1	0
0	1	0

Commutative Law

```
A.B = B.A
A + B = B + A
```



Multiplicative form

$AB = BA$

A	B	AB	BA
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Additive form

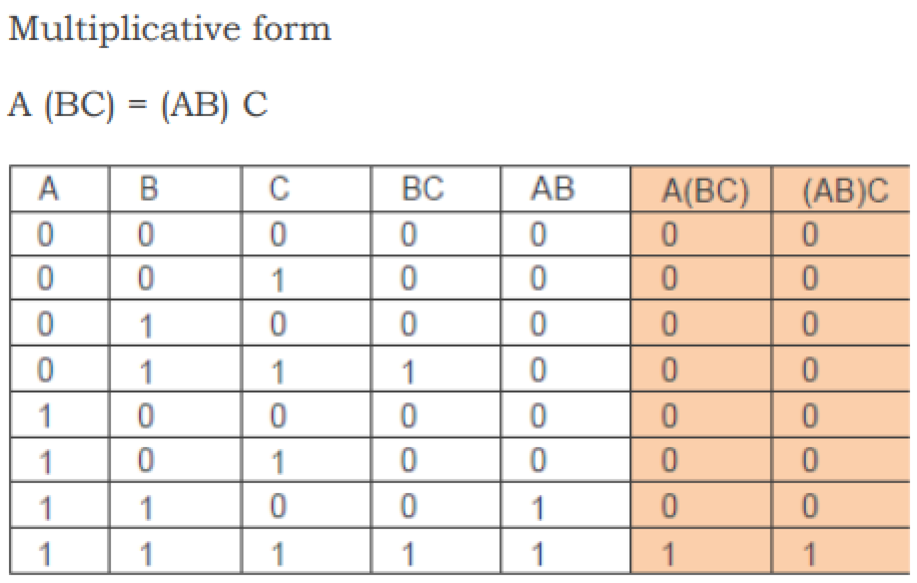
$A + B = B + A$

A	B	A+B	B+A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Above highlighted columns are identical.

Associative Law

```
A (B.C) = (A.B).C = (A.B.C)
```



Multiplicative form

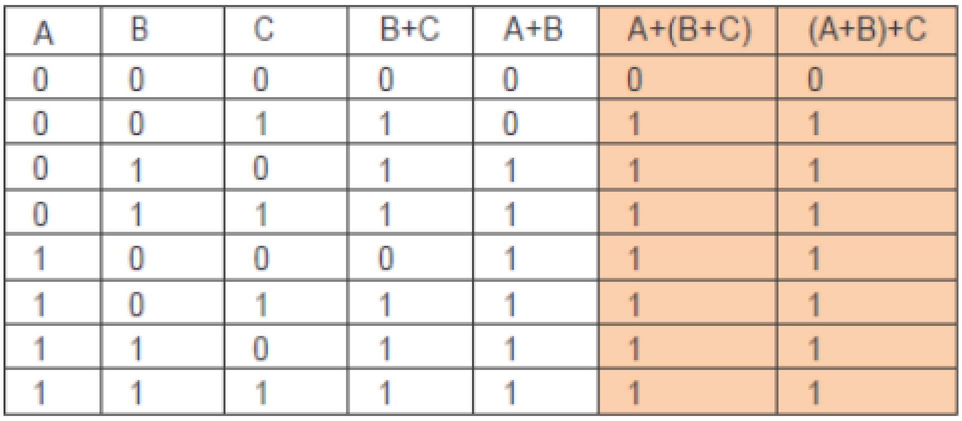
A (BC) = (AB) C

A	B	C	BC	AB	A(BC)	(AB)C
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	1	0	0
1	1	1	1	1	1	1

```
A + (B+C) = (A+B) + C = (A+B+C)
```

Additive form.

A + (B + C) = (A + B) + C

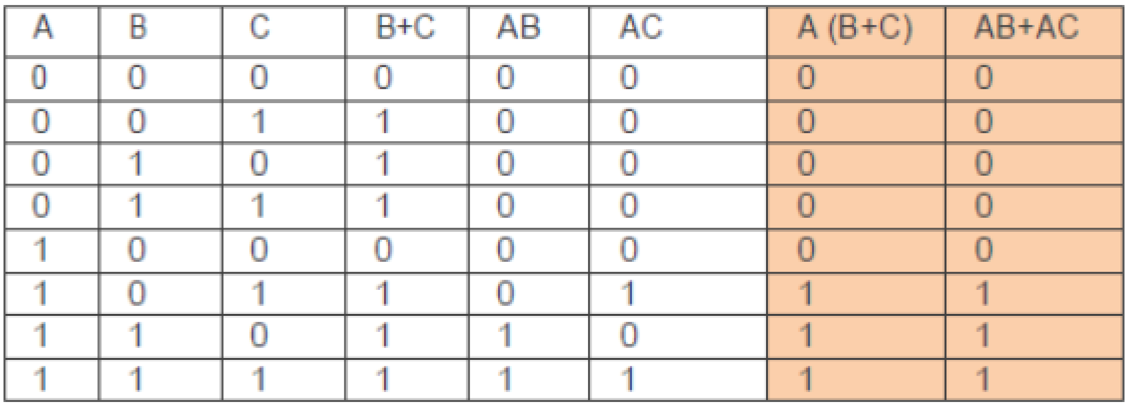


A	B	C	B+C	A+B	A+(B+C)	(A+B)+C
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Distributive Law

```
A (B+C) = AB + AC --> like bracket opening
```

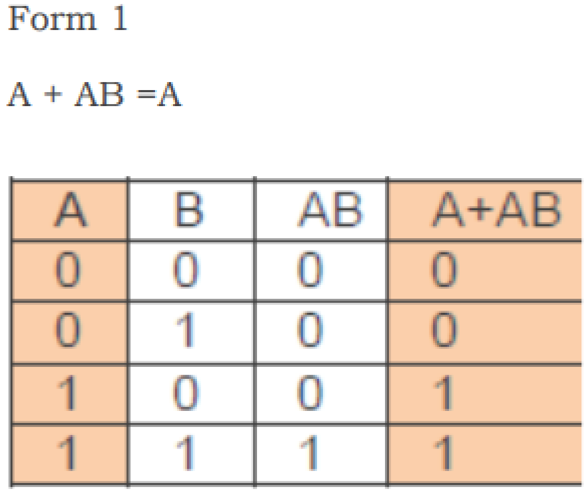
A (B+C) = AB+AC



A	B	C	B+C	AB	AC	A (B+C)	AB+AC
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

Redundancy Law/ Absorption Law

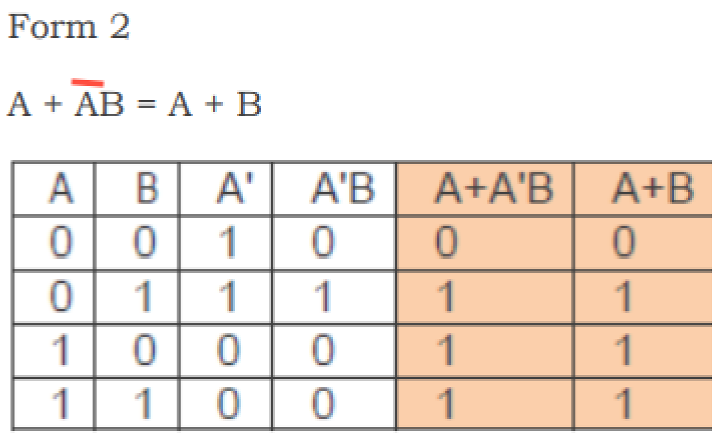
A	B	AB	A+AB=A
1	1	1	1
0	0	0	0
1	0	0	1
0	1	0	0


Form 1

$A + AB = A$

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

A	B	A'	A'B	A+A'B	A+B
1	0	0	0	1	1
0	1	1	1	1	1
1	1	0	0	1	1
0	0	1	0	0	0


Form 2

$A + \overline{A}B = A + B$

A	B	A'	A'B	A+A'B	A+B
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	0	1	1

Also,

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

Also,

```
AB' + ABC = A(B' + BC) = A(B' + C) = AB' + AC
```

03 - Simplifies logic expressions using law of Boolean algebra and Karnaugh map

Simplifies logic expressions using law of Boolean algebra and Karnaugh map

There exist two standard forms of Boolean expressions
1. SOP - Sum of Product - `AB + BC`
2. POS - Product of Sum - `(A+B).(B+C)`

SOP - Sum of products (`(x.y)+(x.y)`)

Two or more product terms are summed by Boolean summation.

When a truth table is made, if the output is 1 (also known as min terms) those terms are taken as products are added together

```
A + B = Z
```

A	B	Z
0	0	0
1	0	0
0	1	1
1	1	1

Here 3rd and 4th entries have the output 1, so these are taken for the SOP expression. When writing the expressions, 0 values should be turned to 1 used a bar (`'`)

```
x = (A'B) + (AB)
```

POS - Product of sums (`(x+y).(x+y)`)

When a truth table is made, if the output is 0 (also known as max terms) those terms are taken as sums and multiplied together

In the above table, 1s and 2nd entries have the output 0, so these are taken for the POS expression. When writing the expressions, 1 values should be turned to 0 used a bar (`'`)

```
x = (A + B).(A' + B)
```

Standardizing a boolean expression

In a standard boolean expression, all the terms in the expression should have all the variables

SOP standardizing with inverse law (A' + A)

```
A + BC              --> not standard
ABc + AB'C + A'BC'   --> standars
```

What we do is that for the missing variable (I.e `D`), we use the inverse law and get `D' + D` which results in 1 (cuz 1(ABC) = ABC so makes no difference) and use distributive law to the term which doesn't have `D`. Then that will result 2 terms with `D'` and `D`

$$A\overline{B}C + \overline{A}\overline{B} + AB\overline{C}D$$

$$\boxed{A\overline{B}C = A\overline{B}C(D+\overline{D}) = \boxed{A\overline{B}CD + A\overline{B}C\overline{D}}}$$

$$\overline{A}\overline{B} = \overline{A}\overline{B}(C+\overline{C}) = \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C}$$
$$\overline{A}\overline{B}C(D+\overline{D}) + \overline{A}\overline{B}\overline{C}(D+\overline{D}) = \boxed{\overline{A}\overline{B}CD + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}\overline{B}\overline{C}\overline{D}}$$

$$A\overline{B}C + \overline{A}\overline{B} + AB\overline{C}D = \boxed{A\overline{B}CD + A\overline{B}C\overline{D}} + \boxed{\overline{A}\overline{B}CD + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}D + \overline{A}\overline{B}\overline{C}\overline{D}} + AB\overline{C}D$$

Same method is done to all the terms until all variables are in all terms

This is done for some reasons
- It's easy to use a standardized expression to use the karnough map method
- When constructing truth tables its important to have the expression in standard format

POS standardizing with inverse law (A'A)

Similar to SOP standardization, here we do the same. But instead of `D' + D` we use `D'D` to get the variable to the terms

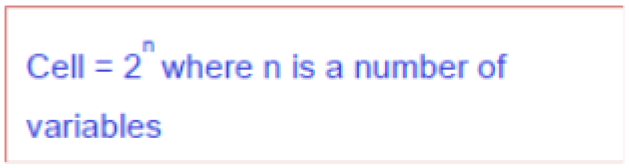
$$(A+\overline{B}+C)(\overline{B}+C+\overline{D})(A+\overline{B}+\overline{C}+D)$$

$$A+\overline{B}+C = A+\overline{B}+C+\boxed{D\overline{D}} = \boxed{(A+\overline{B}+C+D)(A+\overline{B}+C+\overline{D})}$$

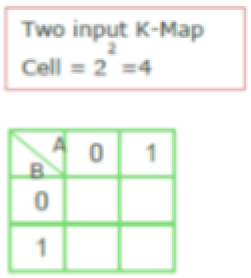
$$\overline{B}+C+\overline{D} = \overline{B}+C+\overline{D}+A\overline{A} = \boxed{(A+\overline{B}+C+\overline{D})(\overline{A}+\overline{B}+C+\overline{D})}$$

$$(A+\overline{B}+C)(\overline{B}+C+\overline{D})\boxed{(A+\overline{B}+\overline{C}+D)} =$$
$$\boxed{(A+\overline{B}+C+D)(A+\overline{B}+C+\overline{D})}\boxed{(A+\overline{B}+C+\overline{D})(\overline{A}+\overline{B}+C+\overline{D})}\boxed{(A+\overline{B}+\overline{C}+D)}$$

Simplifies logic expressions using Karnaugh map

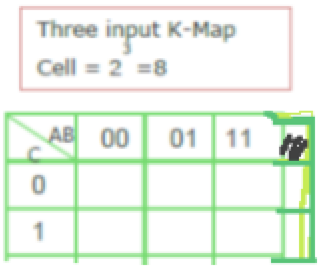


Cell = 2^n where n is a number of variables



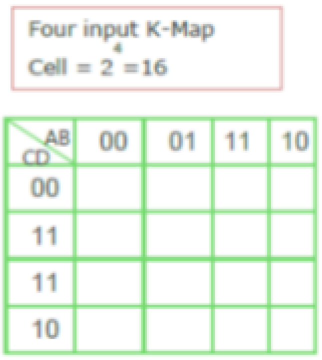
Two input K-Map
Cell = 2^2 =4

B \ A	0	1
0		
1		



Three input K-Map
Cell = 2^3 =8

C \ AB	00	01	11	10
0				
1				



Four input K-Map
Cell = 2^4 =16

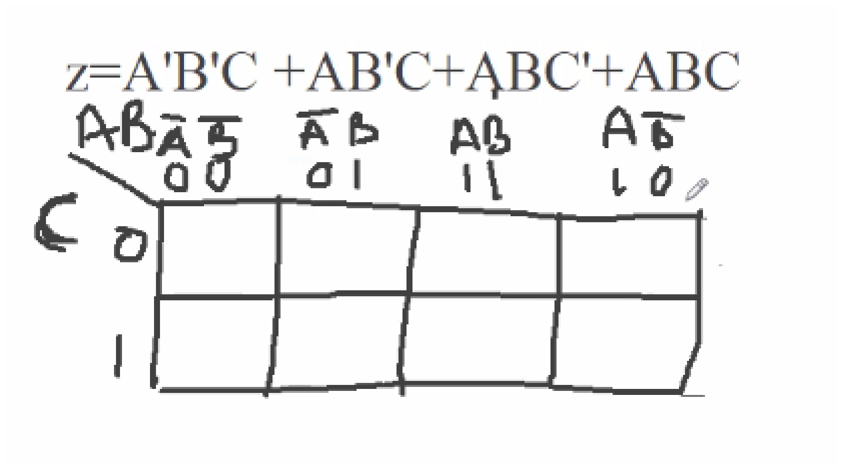
CD \ AB	00	01	11	10
00				
11				
11				
10				

Let's take a 2 bit example

1. Find the no of cells (`2**2 = 4`)
2. Add the gray code to the table cols and rows (here only in cols since 4 cols)

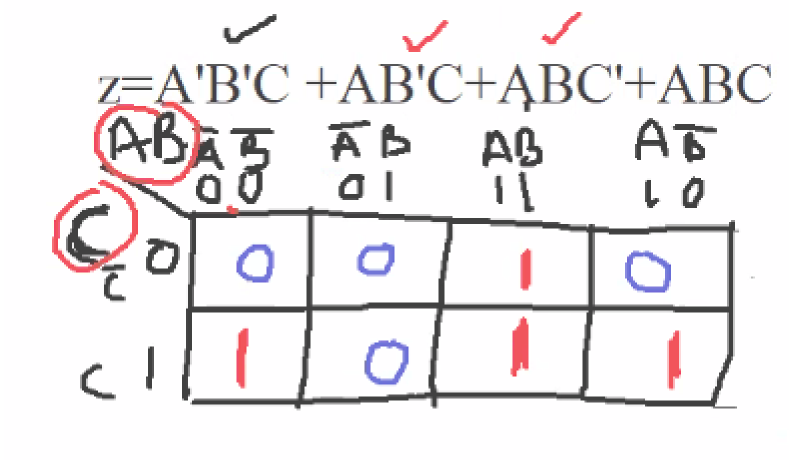
```
This is the gray code you need to remember

00
01
11
10
```

Here a 0 is an input with bar (`'`) since the input is in the format of `AB` in hte left up cell, `00` means `A'B'` , `01` means `A'B`



3. Then we cross-reference the terms we see in our SOP expression, I.e `A'B'C` we put a 1 (since its a SOP expression and it's made my min terms) in the cell under `A'B'` and `C` which is the 2nd row in 1st col

We put zeros to the remaining cells



4. Then we group the cells with of 1 with multiples of 2 or 1(1,2,4,8). There are some rules when grouping them

Rules of K- map simplification

1. No zeros allowed.	5. Every "one" must be in at least one group.
2. No diagonals.	6. Overlapping allowed.
3. Only power of 2 number of cells in each group.	7. Wrap around allowed.
4. Groups should be as large as possible.	8. Fewest number of groups possible.

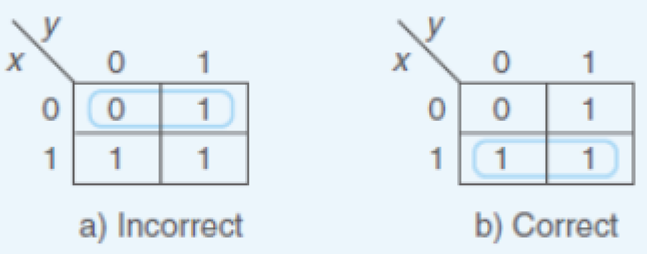


FIGURE 3A.4 **Groups Contain Only 1s**

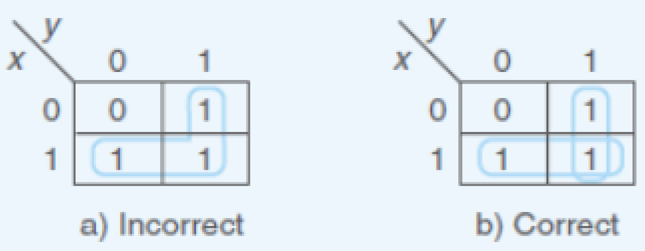


FIGURE 3A.6 **Groups Must Be Powers of 2**

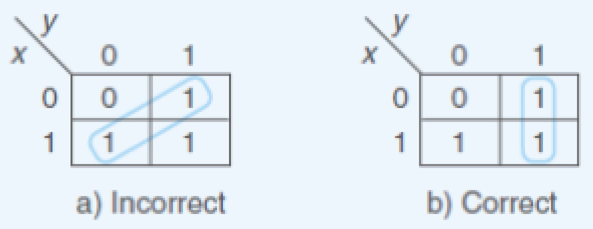


FIGURE 3A.5 **Groups Cannot Be Diagonal**

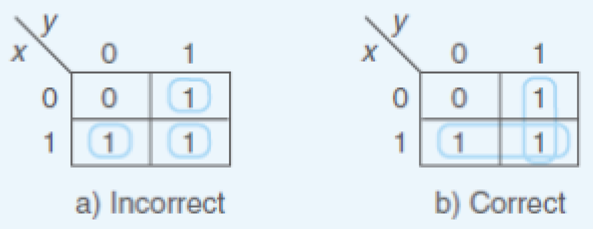
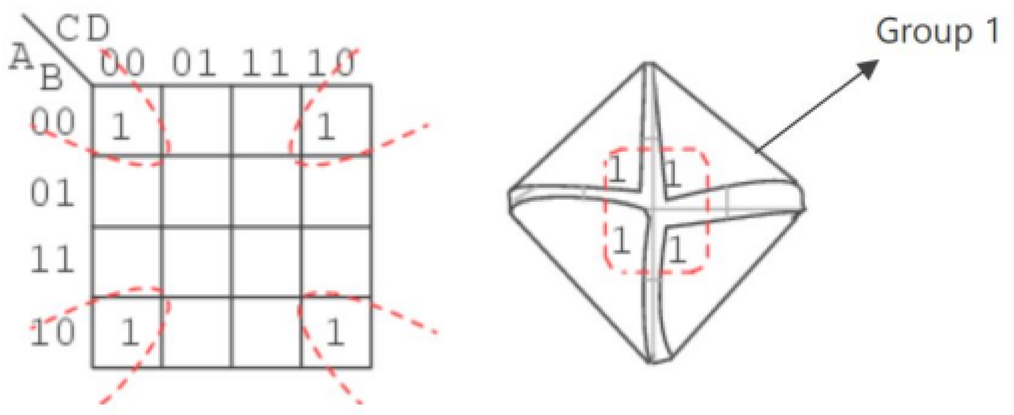
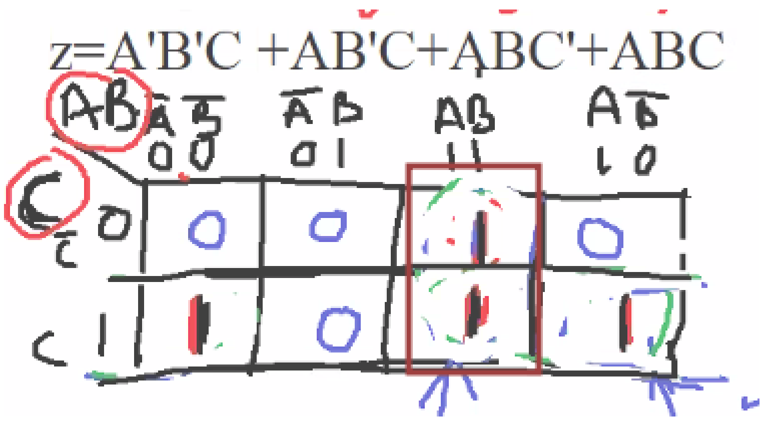


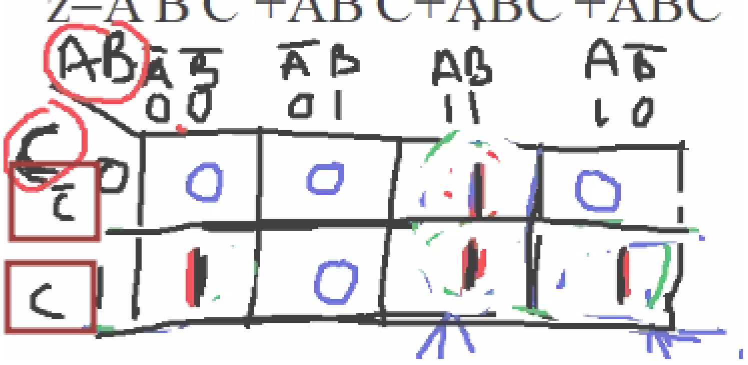
FIGURE 3A.7 **Groups Must Be as Large as Possible**



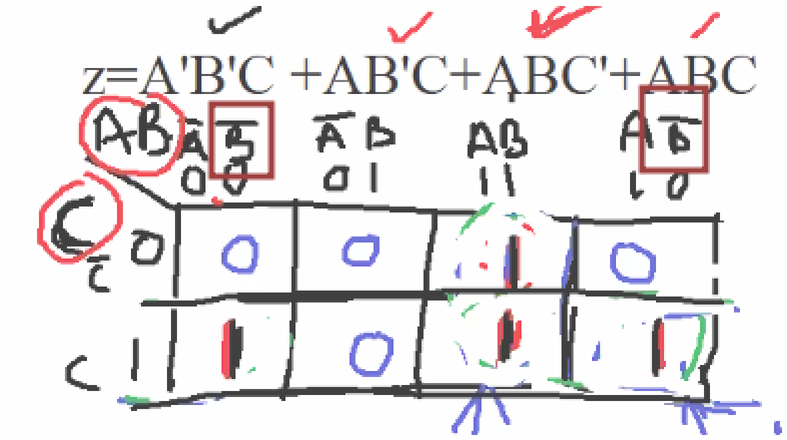
5. Once we have the table filled up and grouped, we need to look for non-changing variables through the cols and rows to the group. I.e 3rd col has two 1s.



And the `AB` value of the 3rd col in same to the 1 in the first row and the 2nd row. So that means it doesn't change. So we pick `AB` And then we go for the row, looking at the 1st row it's a `C` and in the 2ns row it's a `C'` So we can't pick that



In the 1st and the 4th col the value of A changes. Because in the 1st col its `A'` but its the 4th col its `A` But `B'` is same in both. So we pick `B`



Then we look for the row, in the 2nd row it's `C`. And since both the 1s are in the 2nd row, the `C` value doesn't change for them. So we pick `C` too

Now in the end we get our picked values and form the expression. Since this was a SOP, we separate these with a `+`

```
z = AB + B'C
```

- **With SOP, we grouped the cells with `1`. So when it comes to getting POS, we just group the cells with `0`** All the other steps are the same
- In a given question, normally when we fill the map, usually we use the SOP

04 - How combinational logic circuits are used in CPU and sequential circuits in physical memory

How combinational logic circuits are used in CPU and sequential circuits in physical memory

Major building blocks of CPU

- Half Adder
- Full Adder

An adder is a digital circuit that performs addition of numbers.

Half Adder

This adder is used to add 2 single bits.

When we add such 1 bit 2 numbers together, it can be shown as below.

0+0 = 00

0+1 = 01

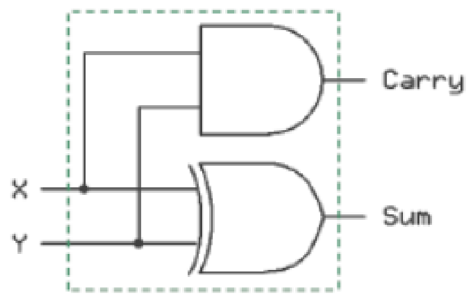
1+0 = 01

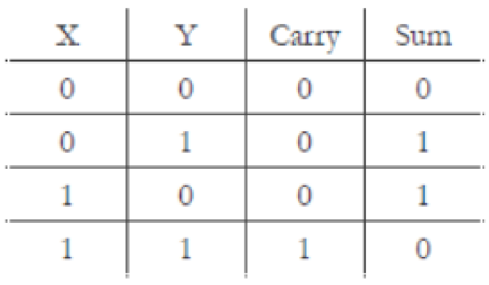
1+1 = 10

Since at the last one we have a carry bit of 1 (`10`) we write all the other answers as 2 bit numbers.

> Here the output '1'of '10' becomes the carry-out. The result is shown in a truth-table below. 'SUM' is the normal output and 'CARRY' is the carry-out.

We can represent this addition in a logic circuit using `AND` gate and `XOR` gate like below

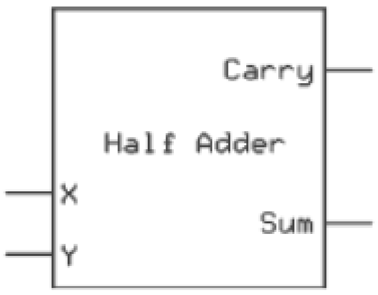




X	Y	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Here, when we give the inputs for x and y as in the truth table, we get 2 outputs as `Carry` and `Sum`. In combination, these two provide the answer of the addition we did.

This circuit is also represented as this



This is called the half adder.

If we wanted to increase the number of numbers, we have to increase the number of half adders we use. I.e

- 2 numbers --> 1 half adder

```
0 + 0 = 00
0 + 1 = 01
```

- 3 numbers --> 2 half adders = Full adder

```
0 + 0 + 0 = 0
0 + 1 + 1 = 10
```

and if we increase the number of bits, a Full adder is added for every increasing bit.

- 2 bit 2 numbers --> 2 Full Adders

```
01 + 01 = 010
11 + 01 = 100
```

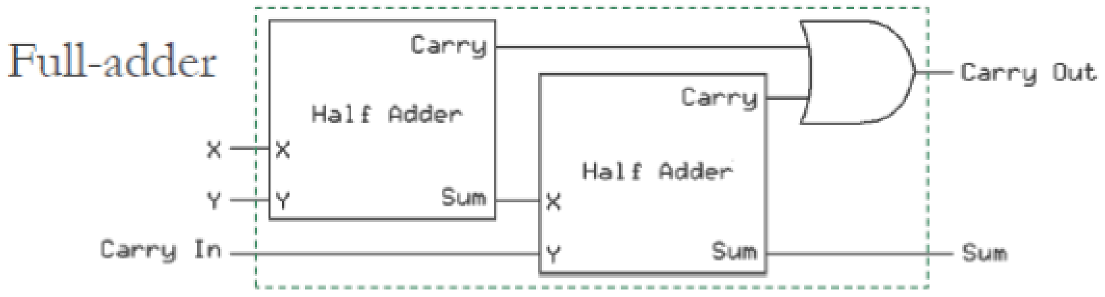
- 3 bit 2 numbers --> 2 Full Adders

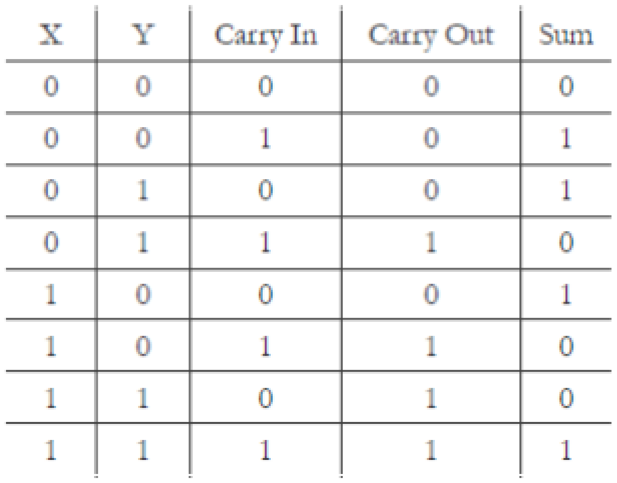
```
000 + 001 = 0001
010 + 100 = 0110
```

Full Adder

This Adder is made by using 2 half adders.

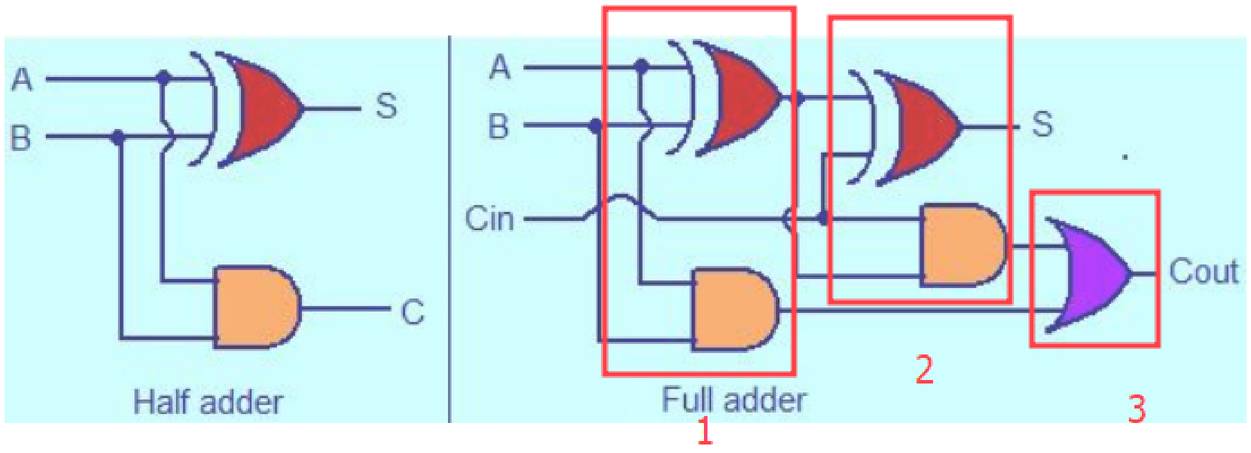
Half Adder	Full Adder
have 2 inputs	have 3 inputs
have 2 outputs	have 2 outputs





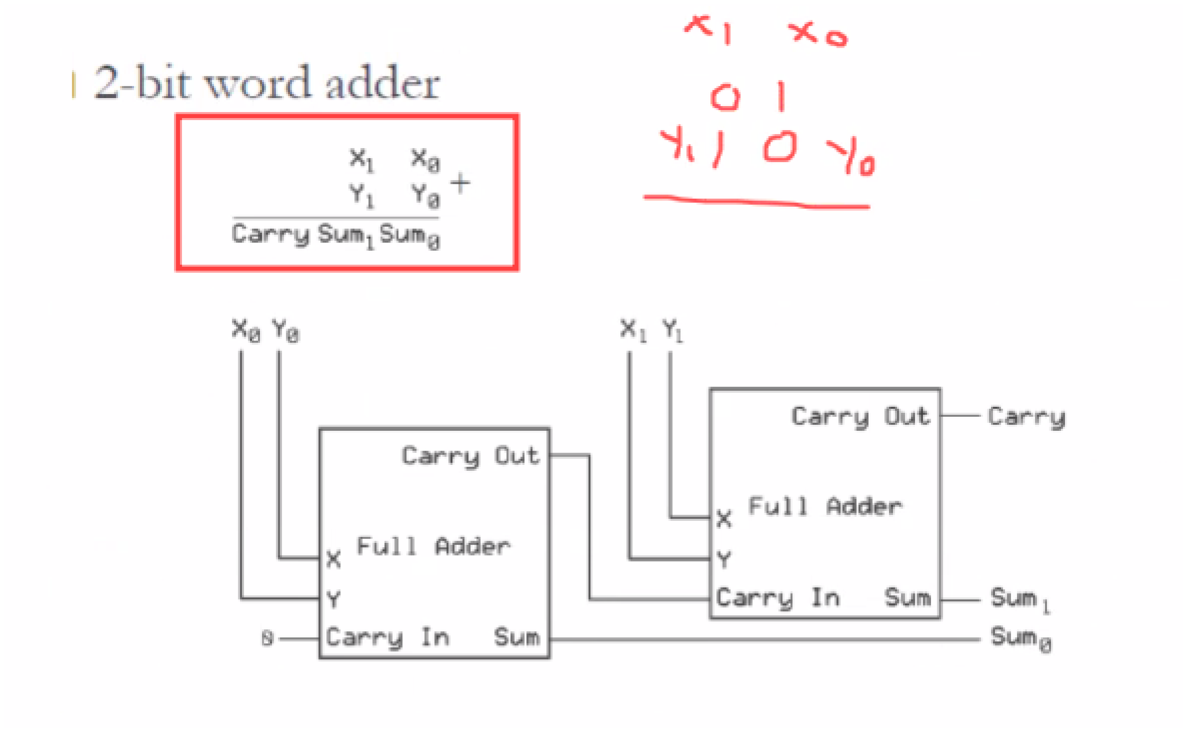
X	Y	Carry In	Carry Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

With 2 half adders an additional `OR` gate is added.



```
1 and 2 -> 2 half adders
3 -> additional OR gate
```

2-bit Word Adder

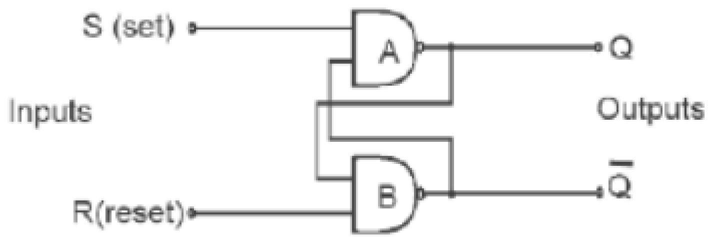


Flip Flops

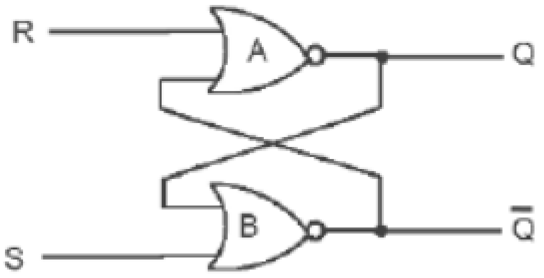
https://www.youtube.com/watch?v=Hi7rK0hZnfc

Flip Flops are logic gates too. Here we can create memory with them. Flip flops can also be considered as the **most basic idea of a Random-Access Memory**

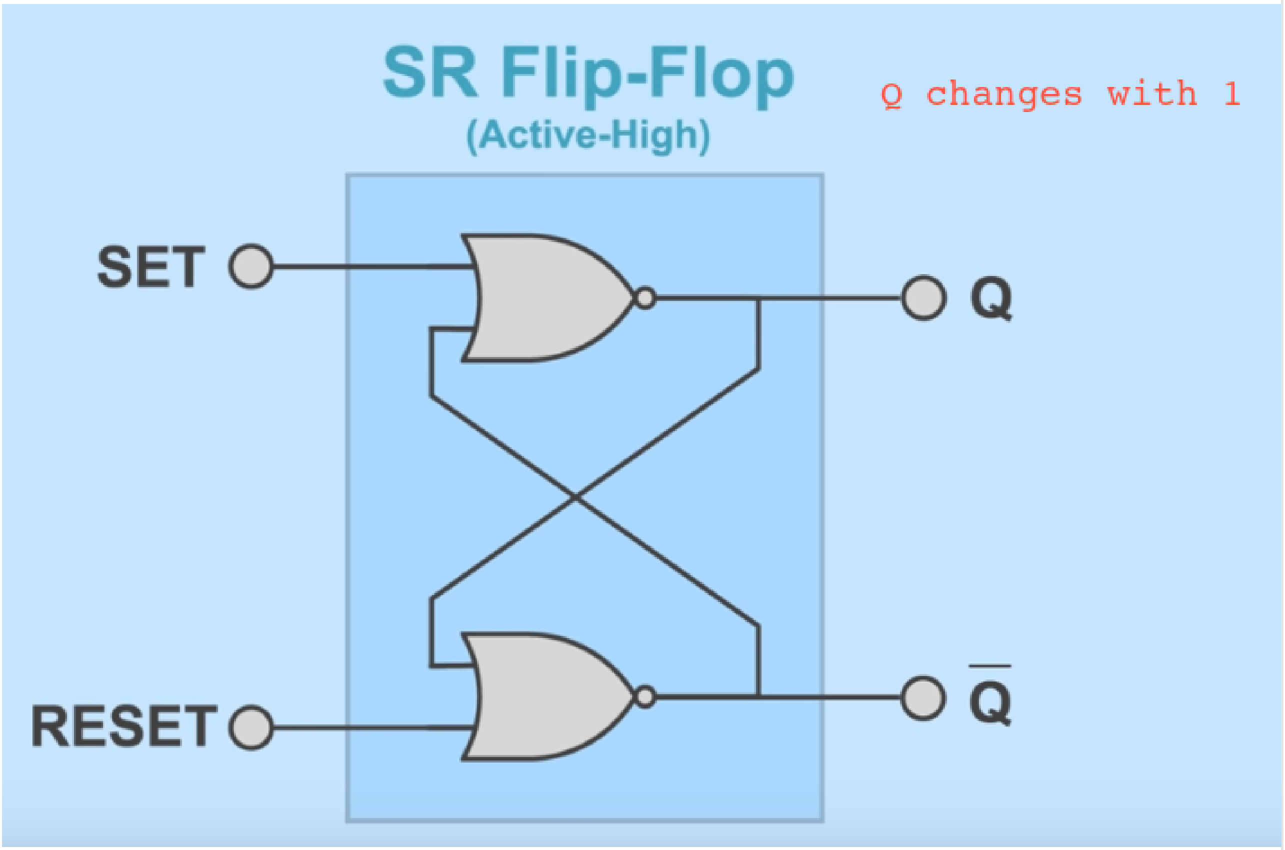
These flip flops are made by coupling either 2 `NAND` gates or `NOR` gates



(a) Latch Flip Flop NAND Gate

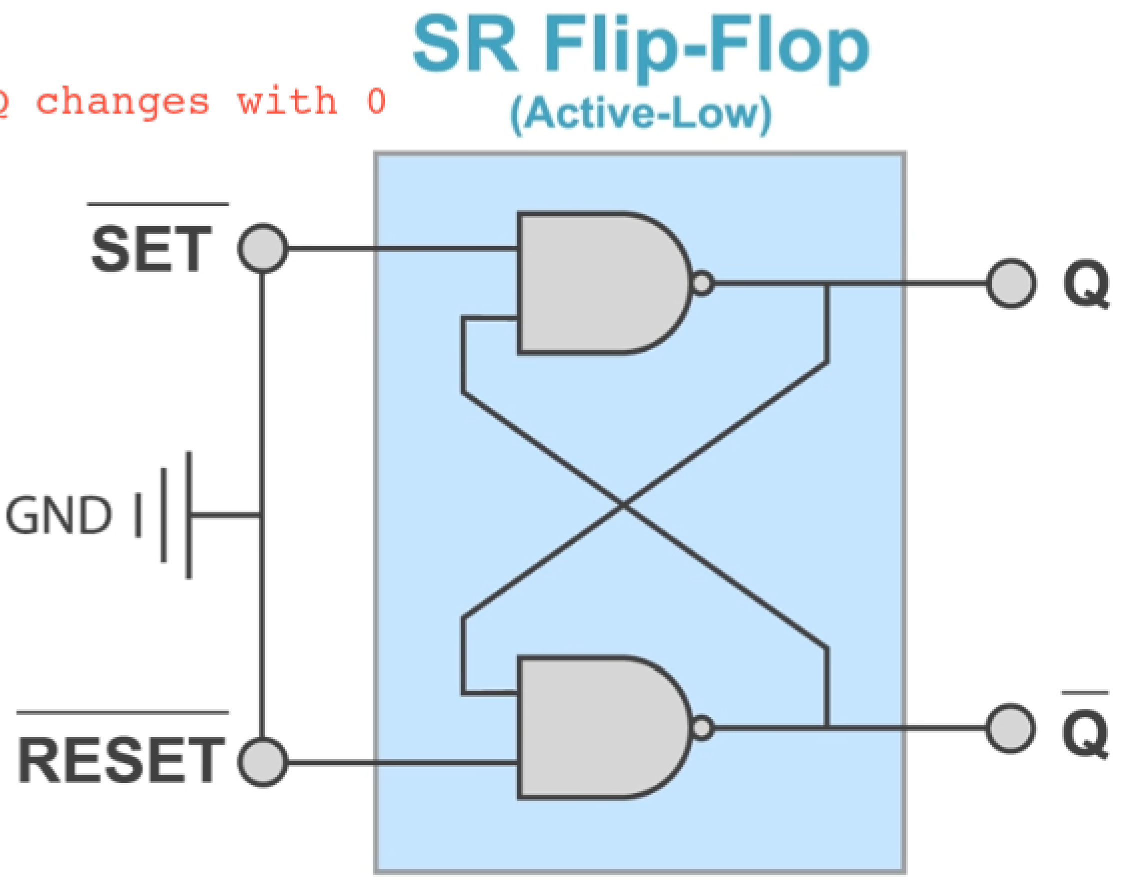


(b) RS Latch Flip Flop NOR Gate



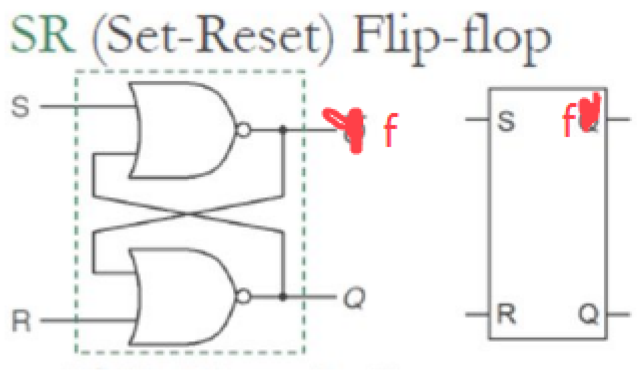
> When set or reset is 1, Q changes

Q changes with 0



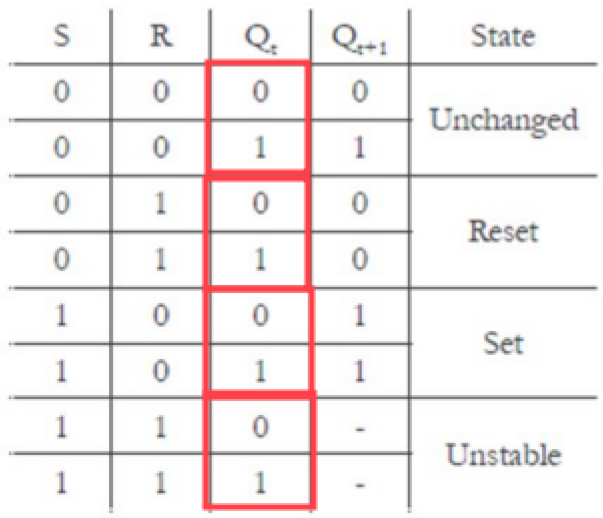
> When set or reset is 0, Q changes

Here, one thing is different, here the output of one gate is used as an input to the other gate.



the `Q` output is used as an input for the `f` (`S NOR Q = F`)output. The same goes for the `F` output. (therefore called sequential circuits)

Because of this behavior one problem occurs. Which is when you try to get the output of one gate, you can't accurately say the value of the other input (which is the output of the other gate). So we get all the possible values which are 0 and 1 and get the output.



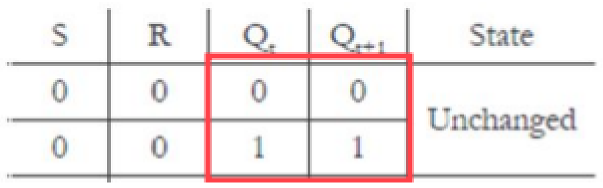
S	R	Q_t	Q_{t+1}	State
0	0	0	0	Unchanged
0	0	1	1	
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	-	Unstable
1	1	1	-	

S	R	State
0	0	Unchanged
0	1	Reset
1	0	Set
1	1	Unstable

Here, since we can't say the accurate value of `Q` to get as an input, we consider both possible values as 2 scenarios.

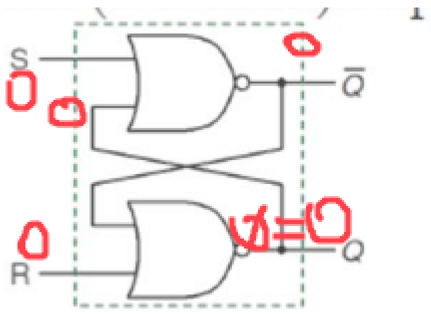
Once that's considered, after it goes through the 2nd cycle, if the original value of `Q` doesn't change and stays the same, we call that State as `Unchanged`

```
IF (S,R) = (0,0) output would remain unchanged
```

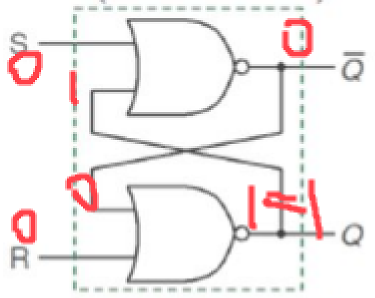


S	R	Q_t	Q_{t+1}	State
0	0	0	0	Unchanged
0	0	1	1	

- when `Q = 0`



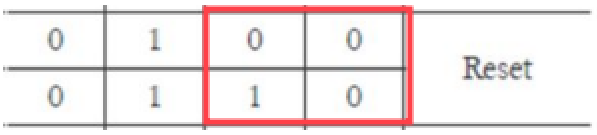
- when `Q = 1`



Here the value we considered stays the same. So the state is `Unchanged`

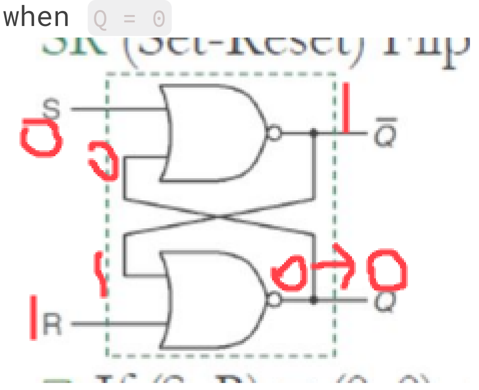
In the next stage it's reset.

```
If (S,R) = (0,1), no matter the 1st output, the next output would be 0
```



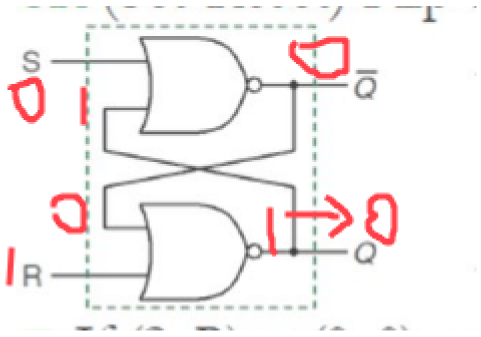
0	1	0	0	Reset
0	1	1	0	

- when `Q = 0`



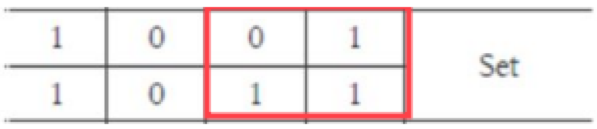
In the first instance, we consider Q as 0. So is the 2nd instance

- when `Q = 1`



In the first instance, we consider Q as 1. But in the 2nd instance it becomes 0

The 3rd stage is reset, this means regardless what the output is in the first time, the last output is going to be 1.



1	0	0	1	Set
1	0	1	1	

There are 2 types of logic gates

1. Combinational circuits
2. Sequential circuits

Combinational circuit

A combinational circuit is a circuit in which the output depends on the current input. I.e Encoders, Decoders

- In combinational circuits, the output depends on the levels present at inputs
- A combinational circuit can have an n number of inputs and m number of outputs.
- The **previous state of input does not have any effect** on the present state of the circuit.
- The combinational circuit do not use any memory.

Sequential circuit

A sequential circuit is a logical circuit, where the output depends on the present value of the input signal as well as the sequence of past inputs I.e Flip Flops

- Based on a concept called feedback
- Output depends not only on the current inputs but also on the previous inputs
- Used for storage (SRAM) and timing
- Generalization of Flip-flops

Combinational	Sequential
Output depends on the present inputs	Output depends on the present input and the past state
No memory	Pocesses memory