To define a binary number:

assign x = 4'b1011

Behavioural way:

```
module example(x,y,z,f);
input (x,y,z);
output f;
wire w1,w2;
assign w1 = x && y;
assign w2 = !z;
assign f = !(w1 && w2);
end module
```

A = 4'b0000 , B = 4'b1010

(1) A! // logic 1

B! // logic 0

A && B // logic 0

(2) Concatenation operator → replication operator

→ {A, B}
= 00001010

→ {A, 3'b101, B}
= 0000 101 1010

→ {B[1], A}
= 10000

B = 4'b1010

B<<1 → 0100

B<<<2 → 1000

>> logical right shift and arithmetic right shift

<< logical left shift and arithmetic left shift

A = 4'b1110

A>>>2

→ 1011

→ {8{1'b0}}
= 00000000

A = 4'b1110

A>>>2

→ 1011

→ A = 4'b11x2 , B = 4'b001x , C = 4'b11x2 , D = 4'b1011.

A===D // logic 0

AB!==D // logic 1

} ← A===D, A!==D case inequalities, we will get logic 0/1

=== → case inequalities

A==D // X

A>=D // X

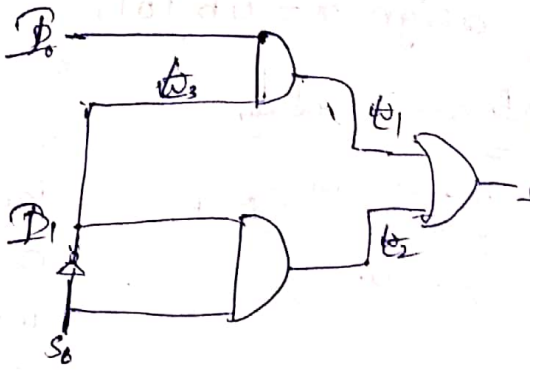
A!=D // X

A===C // logic 1

A==C // X

Verilog code of 2:1 MUX:

```
module MUX_2:1 (I0, I1, S0, f);
input (I0, I1, S0);
Output (f);
wire (t1, t2, t3);
not (t3, S0);
and (t1, I0, t3);
and (t2, I1, S0);
or (f, t1, t2);
end module;
```



$f = I_0 S_0' + I_1 S_0$

in behavioural way:

```
assign f = ((!S0) && I0) || (D1 && S0)
```

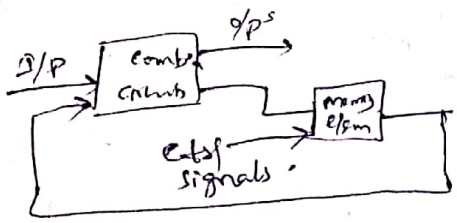
using conditional operator:

```
assign f = S0 ? D1 : D0;
```

if $S_0 = 1$ D_1

else D_0

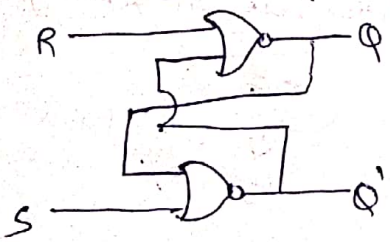
Sequential Circuits:



Memory elements

- Latch
  - S-R
  - D
  - J-K
  - T
- Flip-flops

NOR based SR latch:



S	R	Q	Q'
1	0	1	0
0	0	1	0
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
0	1	0	1
0	0	0	1
1	1	forbidden state	

memory

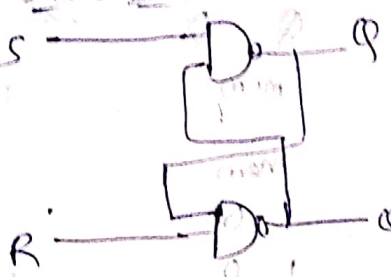
NOR

0	0	1
0	1	0
1	0	0
1	1	0

S	R	Q	Q'
0	0	1	0
1	1	1	0
0	1	1	0
1	1	1	0
1	0	0	1
1	1	0	1
0	0	forbidden	

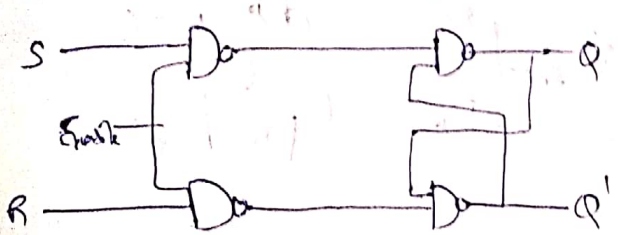
memory state

NAND based SR latch:

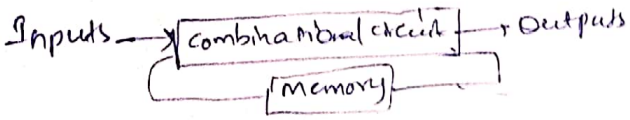


NAND

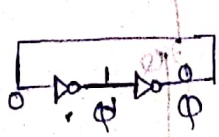
0	0	1
0	1	1
1	0	1
1	1	0

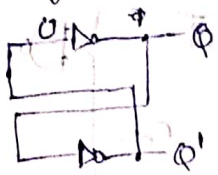


Sequential = input + state. | Combinational = input + output



→ The basic idea of a storage:





$Q_{next} = (R + Q'_{current})'$

$Q'_{next} = (S + Q_{current})'$

Verilog for fulladder:

```
module full_adder(a, b, c, sum, carry);
  input a, b, c;
  output sum, carry;
  assign sum = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c;
  assign carry = a & b | a & c | b & c;
~~end module~~
end.
end module.
```

Sum = diff = $A\bar{B}c + \bar{a}bc + ab\bar{c} + \bar{a}\bar{b}\bar{c}$

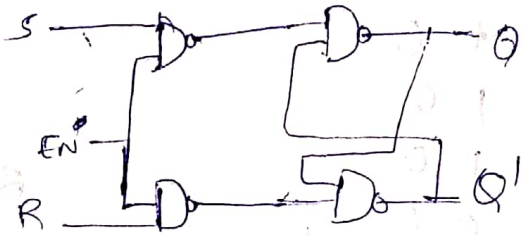
carry = $\bar{a}b + bc + Ca$

Bim = $\bar{a}b + bc + \bar{a}c$

assign sum = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c;

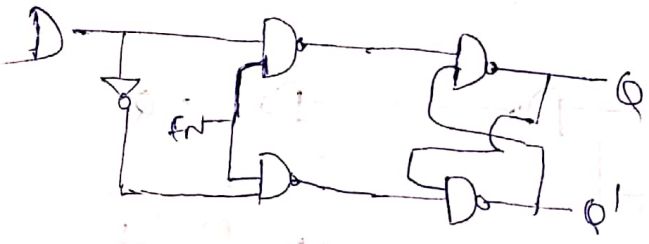
SR latch:

	S	R	Q	Q'
EN=0	X	X	mem	
EN=1	0	0	mem	
	1	0	1	0
	0	0	1	0
	0	1	0	1
	1	1	Undefined	



→ D-latch:

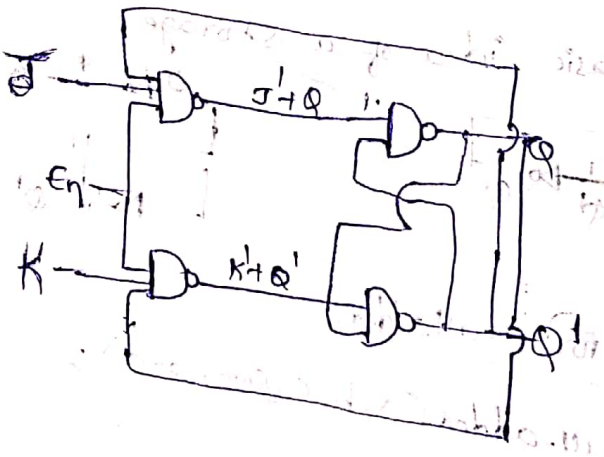
	D	Q	Q'
EN=0	X	mem	
EN=1	1	1	0
	0	0	1



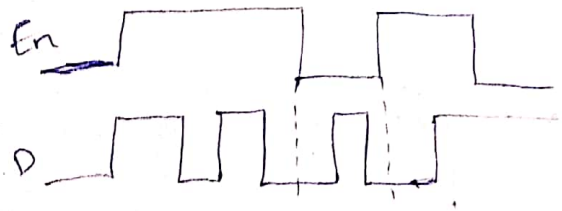
→ J-K Latch:

Q_{prev}	J	K	Q_{next}	Q'
X	EN=0	X	X	mem

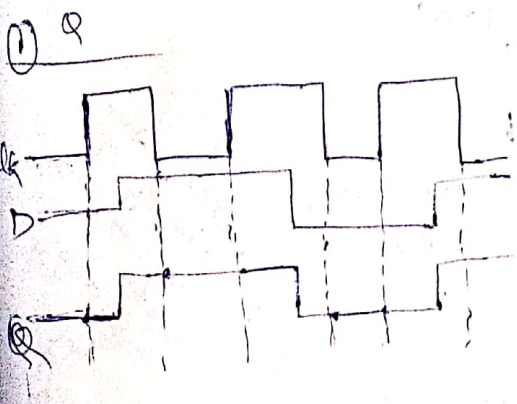
Q_{prev}	EN	J	K	Q_{next}	Q'
0	1	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	1
1	1	0	1	1	0
0	1	1	1	0	1
1	1	1	1	0	1
0	1	1	1	1	0
1	1	1	1	0	1
0	1	1	1	1	0

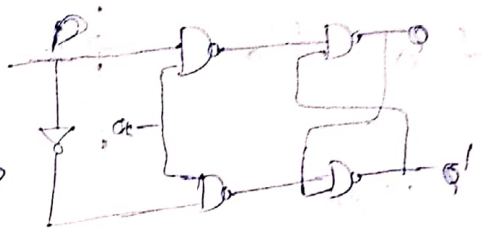


When J=1, K=1 → race around condition



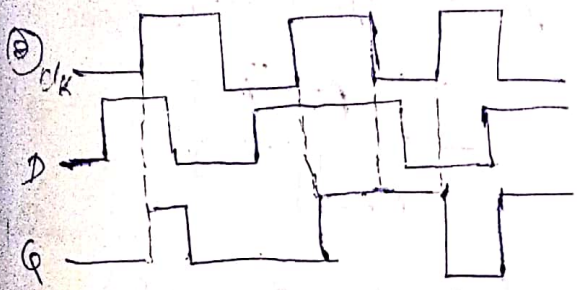
D-flip flop

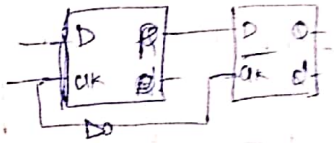




Clk=0	memory state		
Clk=1	D	Q	Q'
↓	0	0	1
	1	1	0

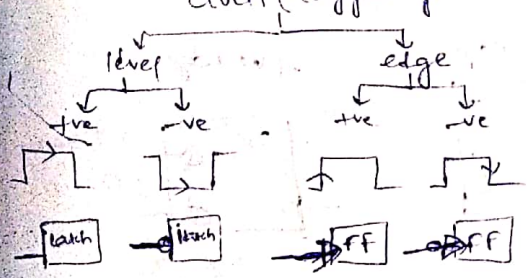
as long as
Clock is 1 output follows inputs.

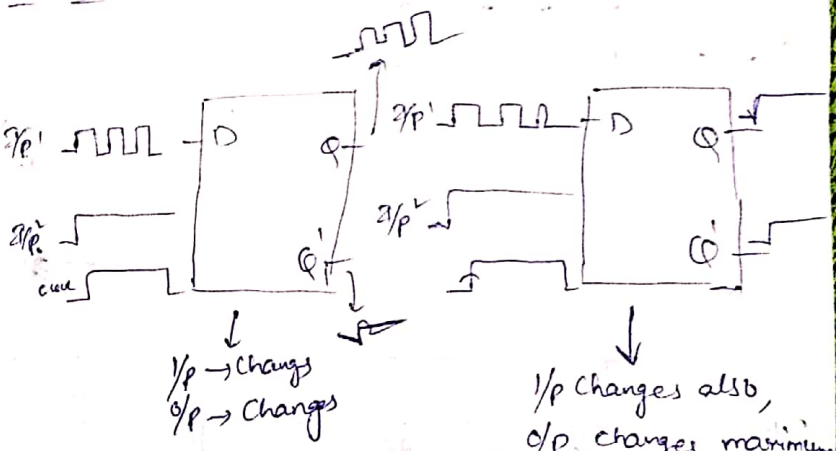




Basics of Sequential circuits:

Clock (triggering)





I/p → Changes
O/p → Changes

I/p changes also,
O/p changes maximum once in a Clock Cycle.

Race around condition:

* difference between toggling & race around condition

(1→0→1)

↓

will happen in an uncontrollable manner.

Latch

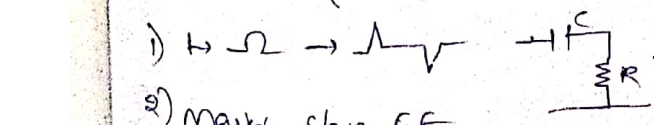
↓

SR latch (drawback S=1, R=1 avoid)

JK latch (J=1, K=1, race around condition)

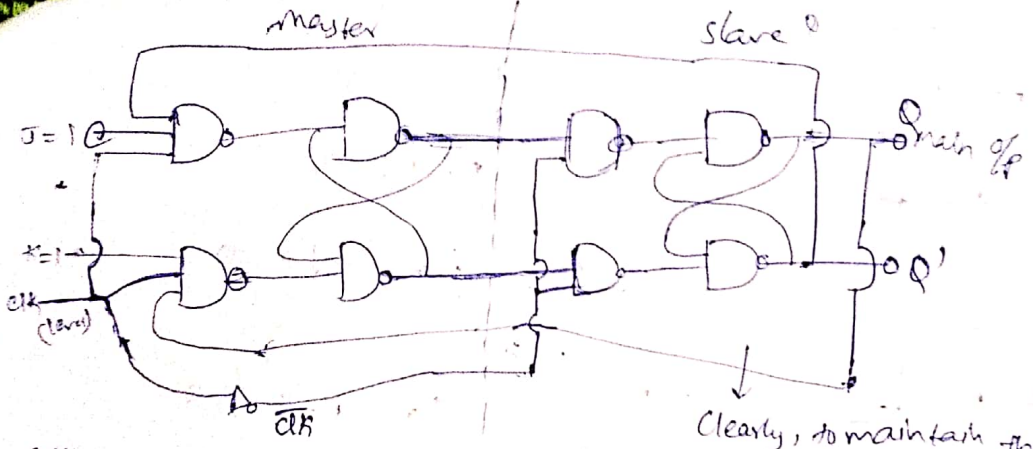
↓

JK FF (J=1, K=1, toggling)

1) 

→ Differentiator.

2) Master slave FF

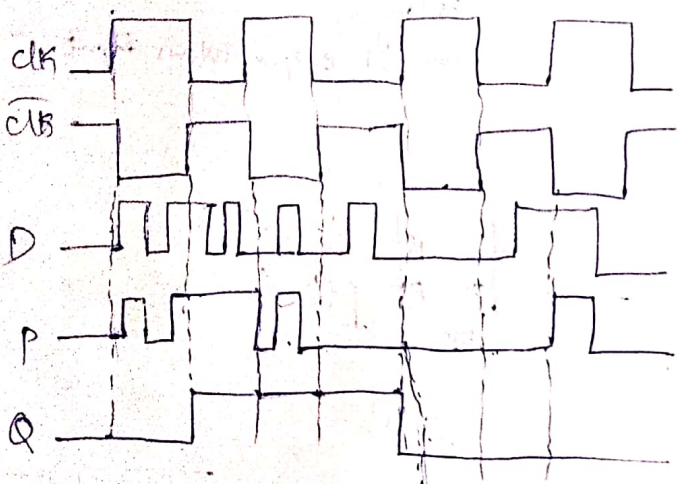


Clearly, to maintain the stable state.

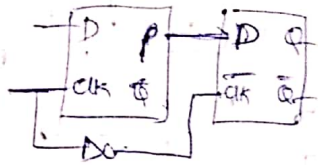
clk = 1 : master activates (Race around condition)

clk = 0 : slave activate (Toggle)

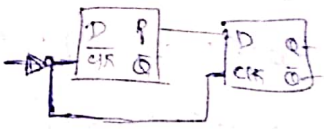
master slave operation of JK flip flop is same as -ve edge triggering.

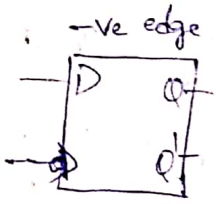


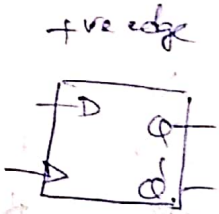
Negative edge D-flip flop:

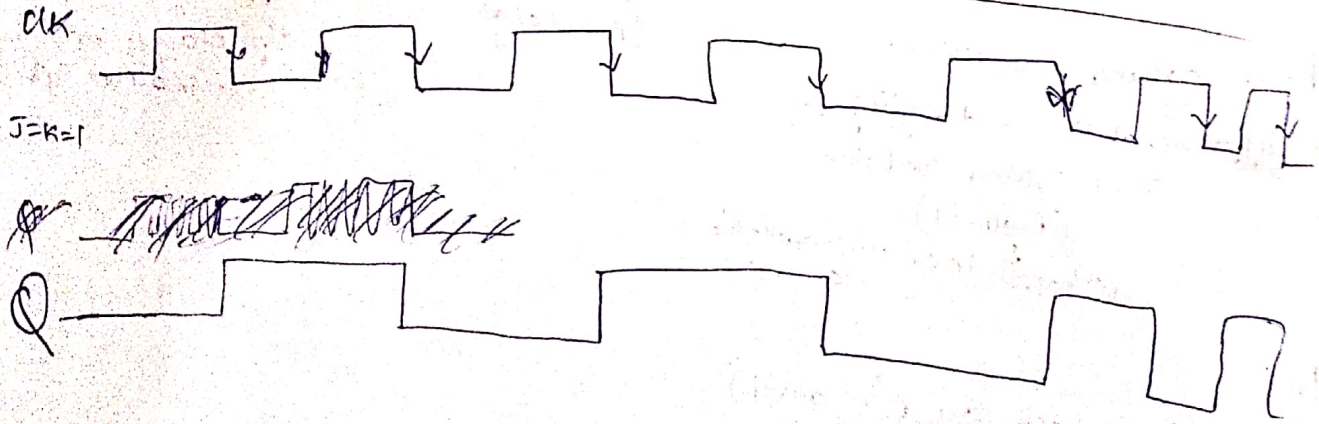


+ve edge D-flip flop.





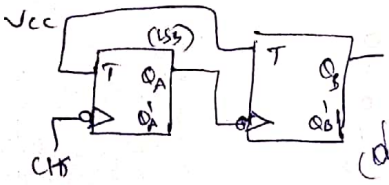




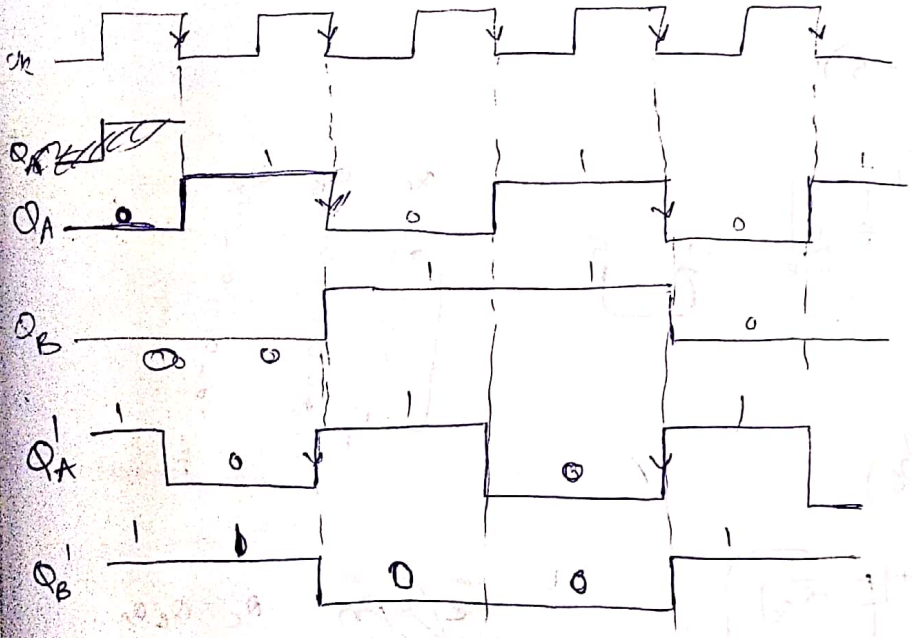
Counters : → Asynchronous (Ripple)
→ Synchronous → Ring
→ Johnson

Asynchronous (ripple)

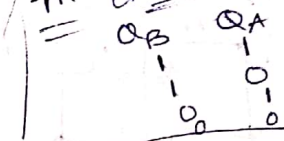
$T = J+K$ → toggle



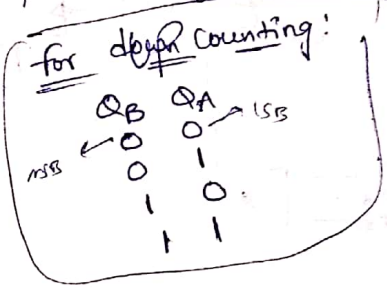
(up counting)



for up counting



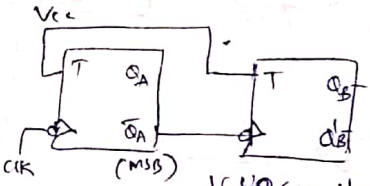
for down counting:



for ($J=K=1$)

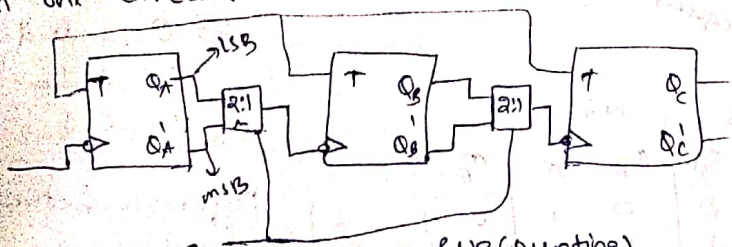
If you have n-flip-flops it will count up to $0 - (2^n - 1)$

for doing down counting we will take Output as $\overline{Q}_A$, $\overline{Q}_B$ —



(down counting)

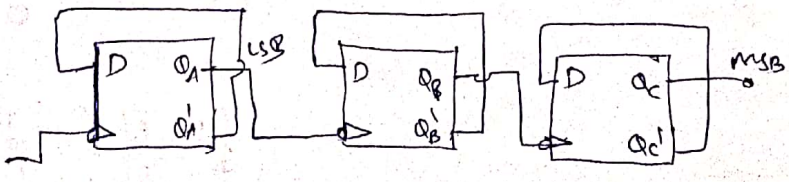
→ for doing both upcounting and downcounting in one circuit.

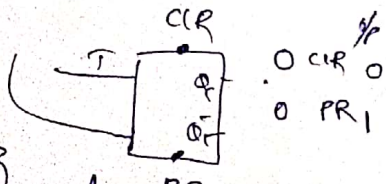


$S = 0/1$, $S = 0$ (upcounting)
$S = 1$ (down counting)

mod-n
↓
n → no. of states
mod-5
0-1-2-3-4

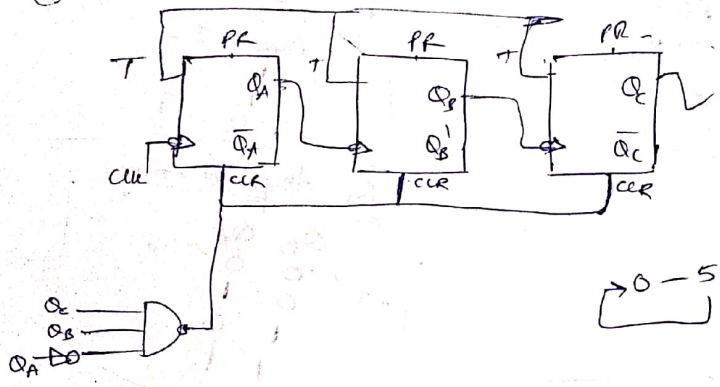
→ Using D-flip-flop for doing upcounting and downcounting.





We will use CLR and PR pins to change count signal

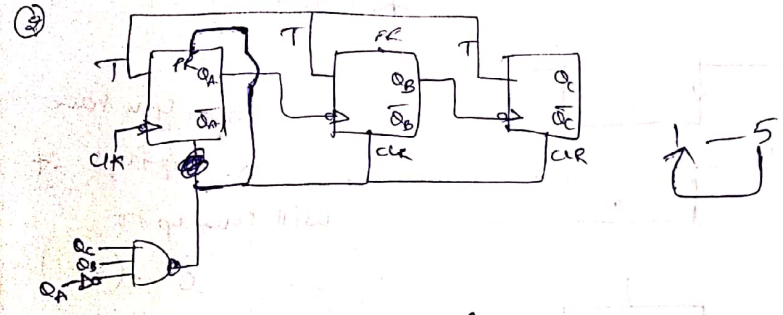
① Upcounting from (0-5)



Q_C	Q_B	Q_A
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

MSB ← → LSB

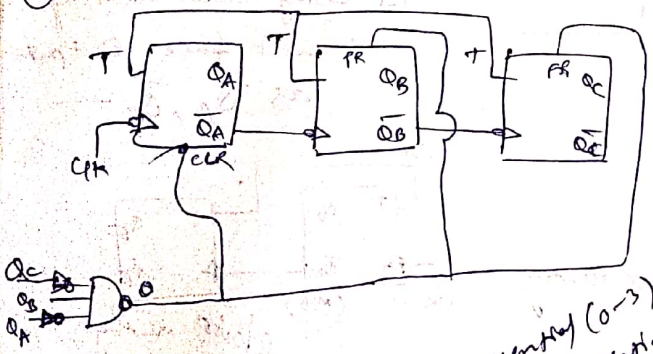
② Upcounting from (1-5)

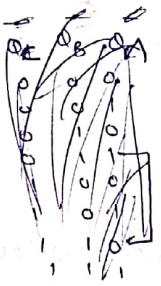


Q_C	Q_B	Q_A
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

1-5

③ Down counting from (6-3/6)

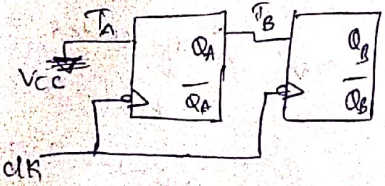




Q_C	Q_B	Q_A
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

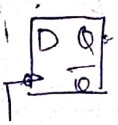
6-3

Synchronous Counters → sequential (0-3) (0-7)
→ non-sequential (0-1-2-4-6)

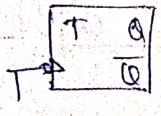


PS		NS			
Q_B	Q_A	Q_B	Q_A	T_B	T_A
0	0	0	1	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1

Difference between PS and NS of D-flipflop and T-flip-flop:

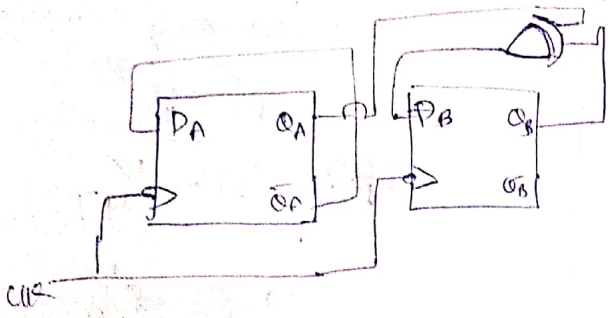


PS	Input	NS
0	0	0
1	0	0

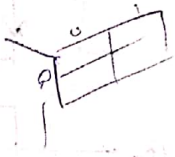


PS	Input	NS
0	0	0
1	1	0
0	1	1
1	0	1

* Design a syn up counter using D-flipflop (0-3).



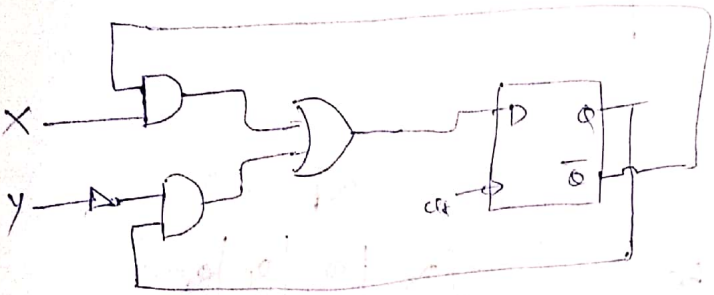
PS		NS		D_B	D_A
Q_B	Q_A	Q_B	Q_A		
0	0	0	1	0	1
0	1	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0



$D_B = Q_B \oplus Q_A$

$D_A = \bar{Q}_A$

*



State equation

$D = xQ' + y'Q$

state table

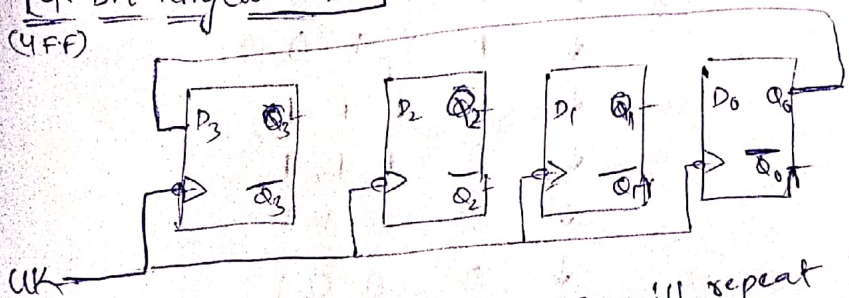
Inputs		PS		NS
x	y	Q(t)	↓	Q(t+1)
0	0	0		0
0	0	1		1
0	1	0		0
0	1	1		0
1	0	0		1
1	0	1		1
1	1	0		0
1	1	1		0

Implementing J-K flipflop from D-flipflop.

*

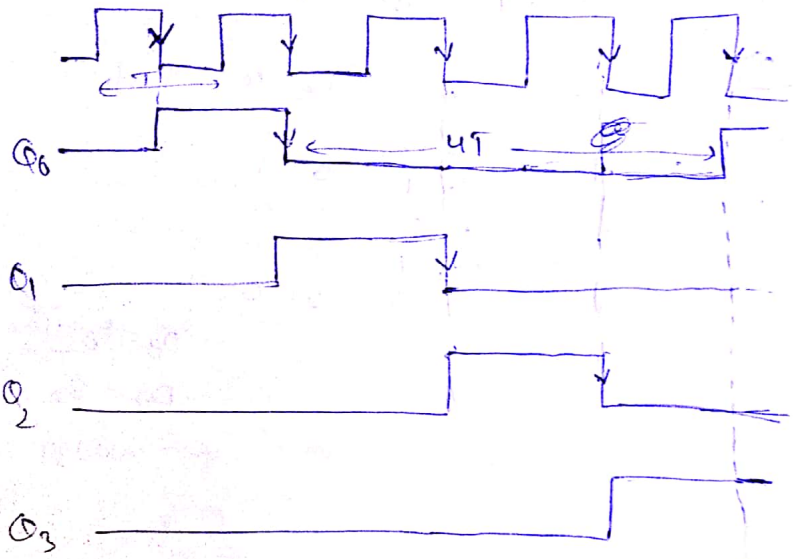
4-bit Ring Counter:

(4 FF)



No:	CLK	Q_3	Q_2	Q_1	Q_0
0	↓	0	0	0	1
1	↓	1	0	0	0
2	↓	0	1	0	0
3	↓	0	0	1	0
4	↓	0	0	0	1

It will repeat in a sequence manner if it is 4-bit then 4 different patterns will come.



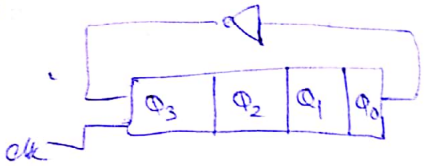
Timing diagram of 4-bit Ring Counter:

$f = 1/T$

Ring $(f) = 1/4T$

$\Rightarrow f = f'/4$.

4-bit Johnson Counter :

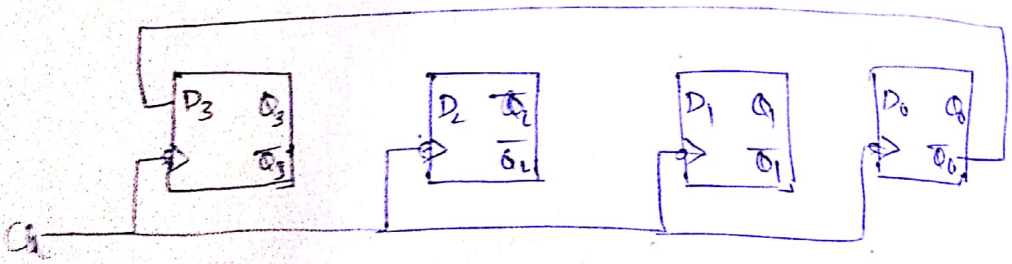


Advantage: no. of states will be double compared to ring counter.

Ring (N FF) = N states

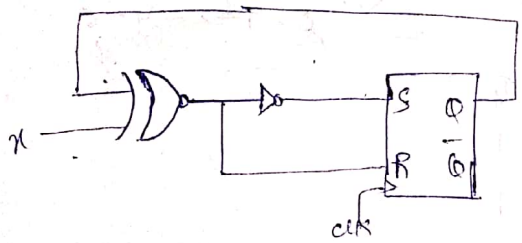
Johnson (N FF) = 2N states

Construction: similar to ring counter, expect $\overline{Q_0}$ connected to D_3.



Clk	Q_3	Q_2	Q_1	Q_0
↓ (neg edge)	1	0	0	0
↓	1	1	0	0
↓	1	1	1	0
↓	1	1	1	1
↓	0	1	1	1
↓	0	0	1	1
↓	0	0	0	1
↓	0	0	0	0

* Write state equation and state table:



State equation:

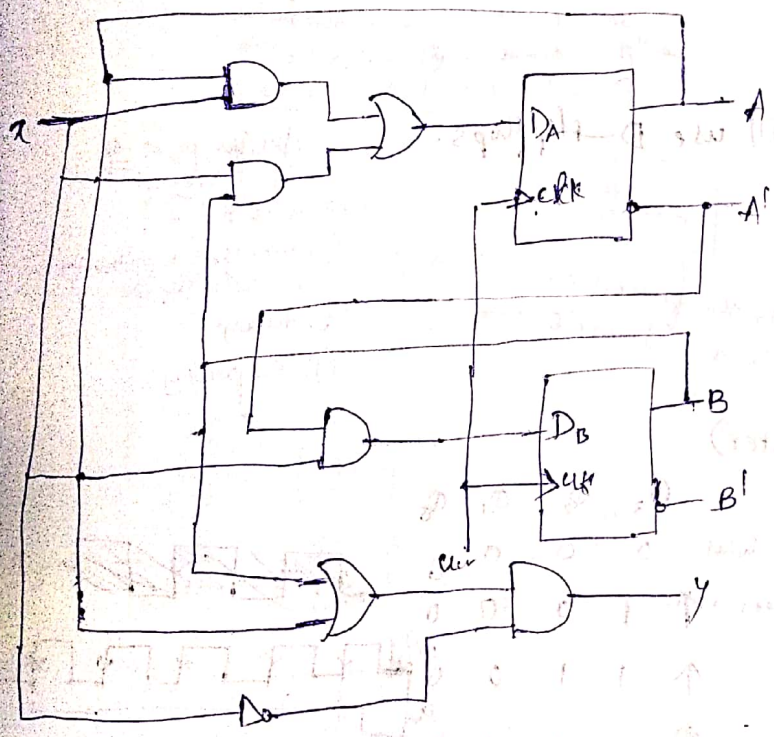
$S = (x \oplus Q)$

$R = (x \odot Q)$

State table:

S	R	PS Q(t)	Input x	NS Q(t+1)
0	1	0	0	0
1	0	0	1	1
1	0	1	0	1
0	1	1	1	0

* Write State equation and state table:



state equation:

$D_A = Ax + Bx$

$D_B = A'x$

$y = x'(A + B)$

PS A(t)	PS B(t)	Input x	NS A(t+1)	NS B(t+1)
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

Non-Sequential Synchronous Counter:

Ex: 0 → 1 → 2 → 4 → 5 → 6 → 0

PS			NS					
Q_C	Q_B	Q_A	Q_C	Q_B	Q_A	T_C	T_B	T_A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	1	0	0	1	1	0
0	1	1	x	x	x			
1	0	0	1	0	0	1	1	0
1	0	1	x	x	x			
1	1	0	0	0	0	1	1	0
1	1	1	x	x	x			

SI = N bit = N clock pulses
SO = N bit = (N-1) clock pulse
(Visualize) PO = N bit = 1 clock pulse.
PI = N bit = 1 clock pulse

Registers: → we will use D-flipflops.

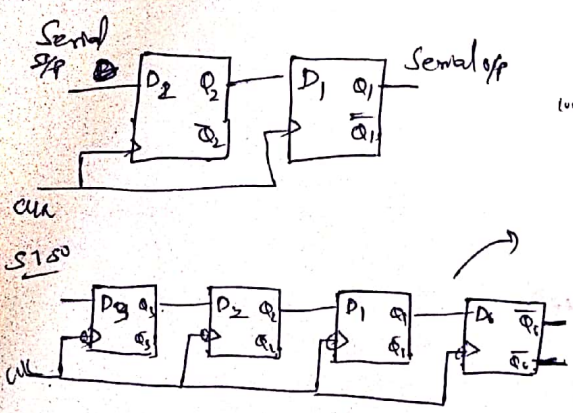
1 → clock pulse, (N-1) → clock pulses

four types:
- SI SO
- SI PO → 1 clock pulse
- PI PO
- PI SO

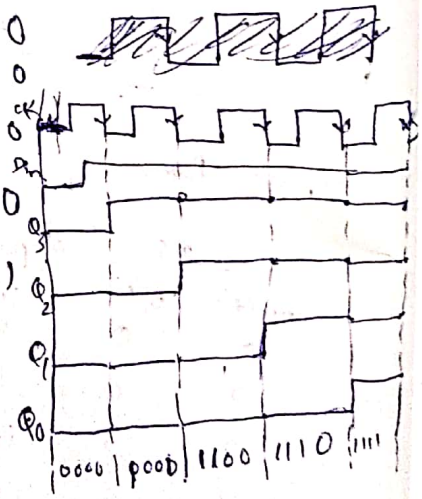
PIPO is the fastest one
SISO is the slowest one

design point of view:
- (a) which FF ?
- (b) whether synchronous (or) Asynch ?
- (c) delay
- (d) capacity

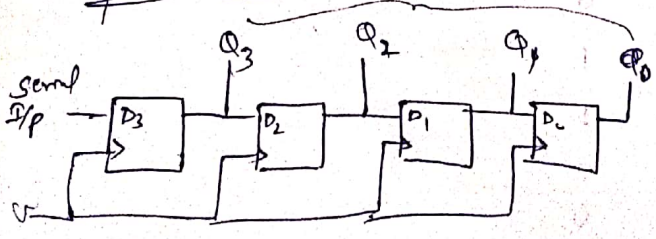
Ex: ① SISO register: (shift register)



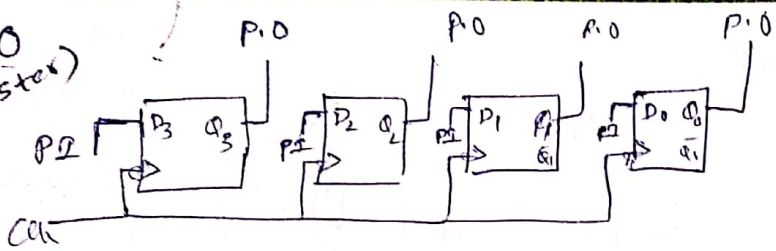
	Q_3	Q_2	Q_1	Q_0
initial	0	0	0	0
load 1111 ↑	1	0	0	0
↑	1	1	0	0
↑	1	1	1	0
↑	1	1	1	1



② SIPO Register:

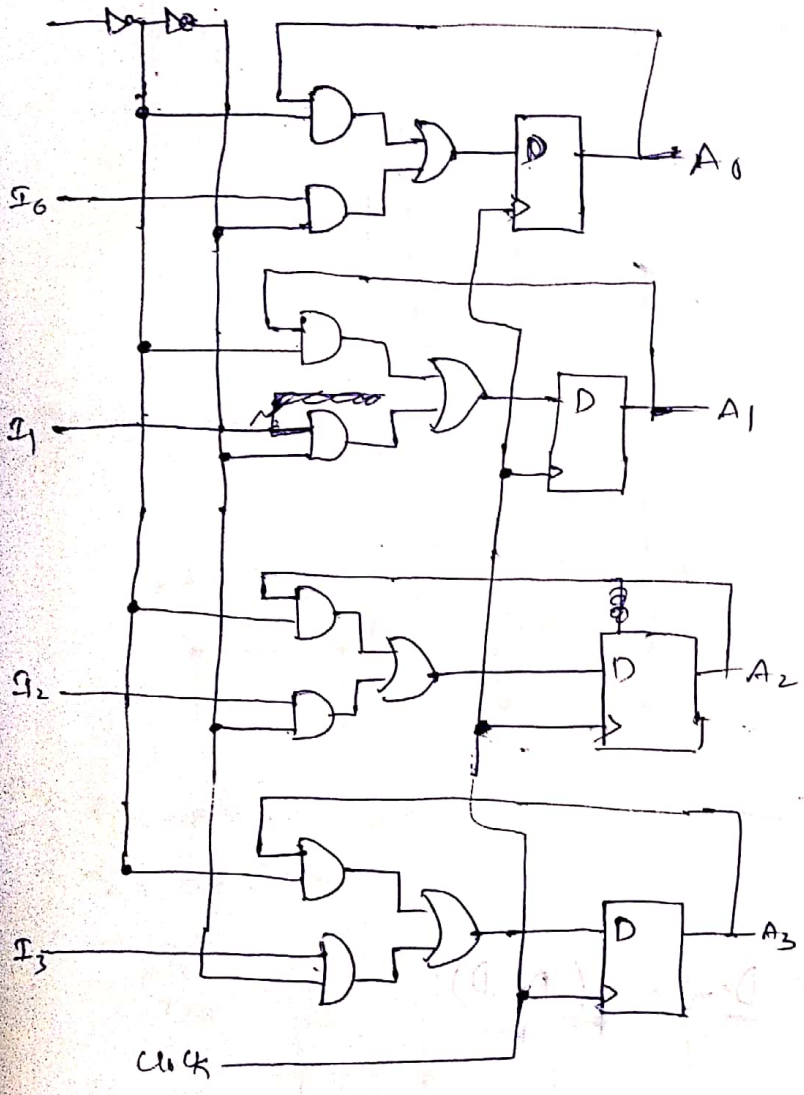


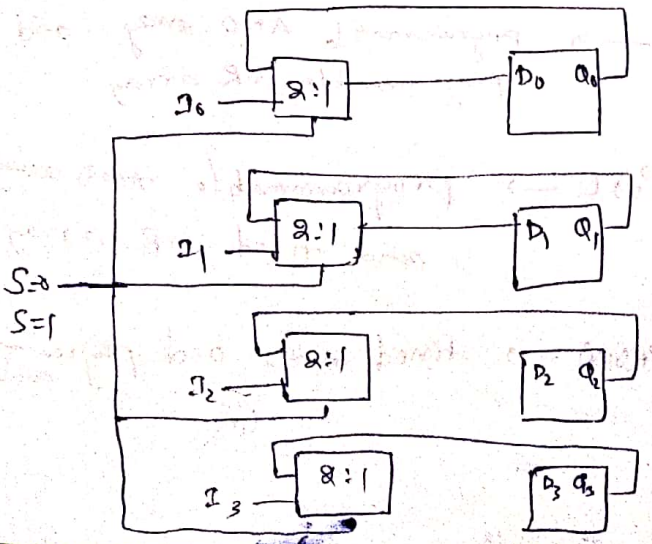
③ PIPO (Storage register)



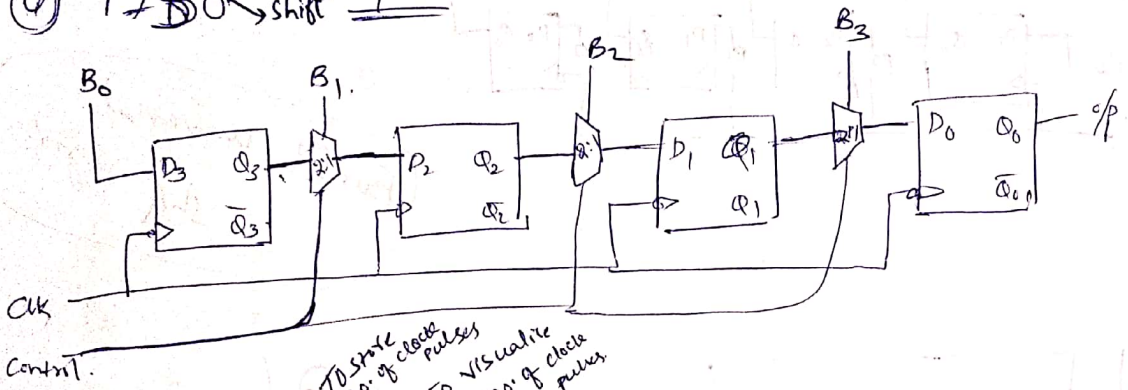
Four bit universal shift register

Ex: load





(4) PISO → load, shift — 4 bit



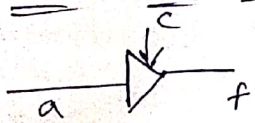
To store no. of clock pulses → To visualize no. of clock pulses

4 bit:

SISO	4	3	= 7
SIPO	4	0	= 4
PISO	4	3	= 4
PIPO	1	0	= 1

Note:

Tri-state Buffer:



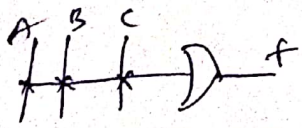
a	c	f
0	1	z
1	1	z
0	0	0
1	0	1

Programmable Logic Device (PLD):

PLA
PAL

PLA (programmable Logic Array) → programmable AND array and programmable OR array

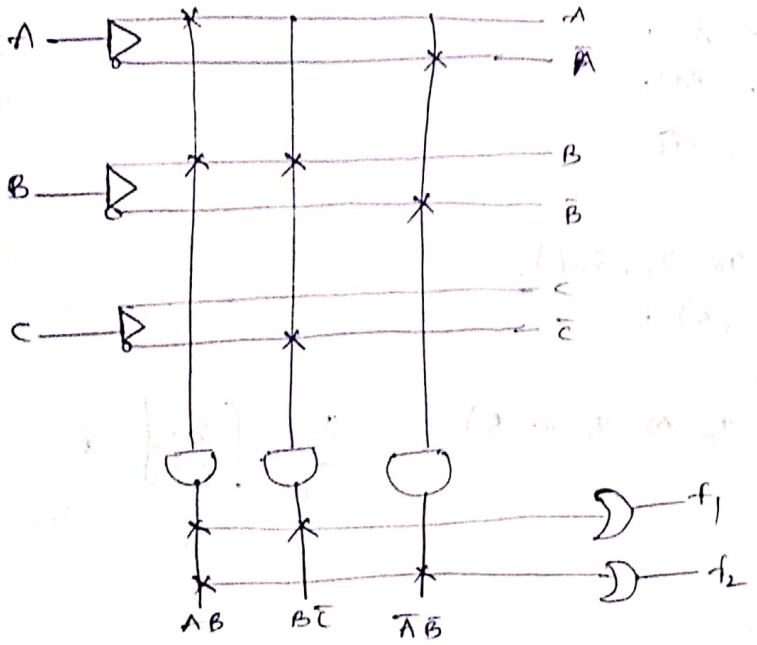
$f = A\bar{B} + C$



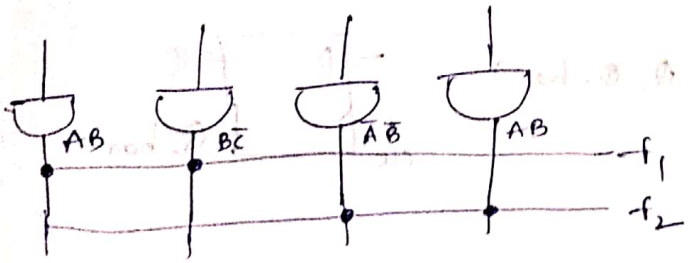
PAL → programmable AND array and fixed OR array

PROM → fixed AND and programmable OR

Ex: $f_1 = AB + B\bar{C}$
$f_2 = AB + \bar{A}\bar{B}$
} Design with PLA

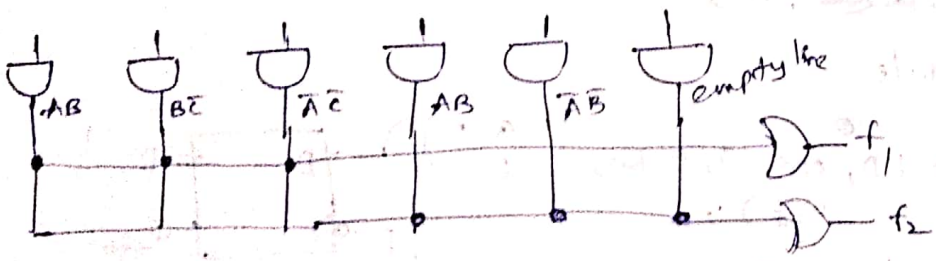


Design with PAL:



Ex: $f_1 = AB + B\bar{C} + \bar{A}\bar{C}$
$f_2 = AB + \bar{A}\bar{B}$

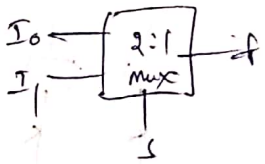
Note: In PAL equal no. of vertical lines should be there for f_1 and f_2.



Verilog for Sequential circuits

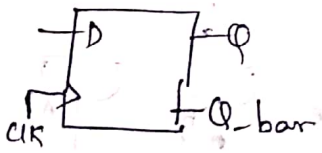
Blocking	Non Blocking
B = A ;	B <= A ;
C = B+1 ;	C <= B+1 ;
C = A+1 ;	C = B+1

Ex: ①
```
module ckt (I0, I1, S, f);
  input (I0, I1, S);
  output f;
  always @ (I0 or I1 or S)
    if (S)
      f = I1;
    else
      f = I0;
  end module
```

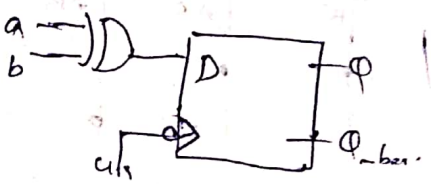


② D-flipflop:

```
module ckt (D, clk, Q, Q_bar)
  input  D, clk;
  output Q, Q_bar;
  reg    Q, Q_bar;
  always @ (posedge clk)
      Q <= D;
      Q_bar <= !D;
  end module
```



③
```
module ckt (D, clk, Q, Q_bar,
                a, b);
  input  a, b, clk;
  output Q, Q_bar;
  reg    Q, Q_bar;
  always @ (posedge clk)
      Q <= D;
      Q_bar <= !D;
```



→ assign D = a & b;

(or) Q <= a & b;

(4)

```
module ckt ( a, b, c, clk, Q1, Q2 );
input ( a, b, c, clk );
output ( Q1, Q2 );
reg Q1, Q2 ;
always @ (negedge clk)
        Q1 <= a ^ b ;
always @ (posedge clk)
        Q2 <= b & c ;
end module
```