DIGITAL SYSTEMS & MICROCONTROLLERS

- Tutorial Test/Quiz (20%)
- Mid-Sem (20%)
- End-Sem (30%)
- Lab (30%) (20% End Sem + 10% Report Submission)

__/__/____

*] Digital Number System:

1] Decimal - (0 → 9) Base 10 b/w 2^3 & 2^4 : 4 used as 3 insufficient

2] Binary - (1,0) Base 2

3] Octal - (0 → 7) Base 8 $\Rightarrow 2^3$ possibilities : 3 bits sufficient

4] Hexadecimal - (0 → 9, A → F) Base 16 $\Rightarrow 2^4$: 4 bits

(A = 10, F = 15)

•] Representation:

Ex:1] $(1475)_{10}$ ($10^3 10^2 10^1 10^0$)

$= 1\times10^3 + 4\times10^2 + 7\times10^1 + 5\times10^0$

Ex:2] $(453)_5$ × nos inside ( ) have to be less than base

Ex:3] $(1011)_2$

$= 1\times2^3 + 0\times2^2 + 1\times2 + 1\times2^0 = 8+0+2+1 = 11$

Ex:4] $(11.0101)_2$

$= 1\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}$

$= 2 + 1 + 0 + 0.25 + 0 + 1/16$

*] Base Conversions:

Ex: $(345)_{10} = (\quad)_2 = (\quad)_8$

		remainder
2	345	
2	172	1
2	86	0
2	43	0
2	21	1
2	10	1
2	5	0
2	2	1
2	1	0
	0	1

write upwards

until quot = 0

$= (1011001)_2$ (10: $2^6 2^5 2^4 2^3 2^2 2^1 2^0$)

→ Ily, for 8, 16, etc.

__/__/____

Ex: $(5.642)_{10} = (\ \ )_2 \Rightarrow 101.xyz$

separately treat — Now for decimal as $^{n-1}$ × instead of ÷

required base

$0.642 \times 2 = 1.254 \Rightarrow 1$

$0.254 \times 2 = 0.568 \Rightarrow 0$ $\Rightarrow (5.642)_{10} = (101.101)_2$

$0.568 \times 2 = 1.136 \Rightarrow 1$

→ If integer formed terminate process

1] Octal to Binary:

Ex: $(345)_8$

011 100 101 as octal 3 bits $\Rightarrow (345)_8 = (011100101)_2$

→ & vice versa # if decimal not grouping into 3 add zeroes

2] Hexa to Binary:

Ex: $(345)_{16}$

0011 0100 0101 as hexa 4 bits $\Rightarrow (345)_{16} = (1101000101)_2$

→ & vice versa # if not grouping into 4 add zeroes

*] BINARY ARITHEMATIC:

1] Addition:

```
  1 1 0
  1110
+ 1110
+ 1110
+ 1110
-------
111000
```

10 → 4 = 100 to be carried over as such

∴ 111000

2] Subtraction:

*Do not mistake it as 10

When borrowing from multiple zeroes make 2 & borrow 1 from it making 1 from 2 & taking 1

```
  0 10 10 (read as 2 ∴ binary)
  1110
- 1011
------
  0011
```

∴ 1110 − 1011 = 11

3] Division:

```
      1001 01 → Quotient
101 | 1101101 |
     -101 ↓↓
     -----
      00111
      - 101
      -----
       01001
       - 101
       -----
         100
          ↳ remainder
```

∴ 1101101 = 1001 × 101 + 100

4] Multiplication:

```
    1101
  × 101
  ------
    1101
+ 0000
+ 110100
---------
1000001
```

∴ 1101 × 101 = 1000001

→ Octal & Hexa Arithematic exactly same logics.

__/__/____

*] COMPLEMENTS:

1] Raddix (r):

- for digit N with n bits with base r:

$$[r = r^n - N]$$

2] Diminished Raddix (r-1):

- for digit N with n bits with base r:

$$[(r-1) = (r^n - 1) - N]$$

◦] Use:

→ Raddix

Ex: $(4567)_{10}$ ⇒ 10's : $10^4 - 4567$ → not easy to calc.

⇒ 9's : $(10^4 - 1) - 4567$

$= 9999 - 4567 = \underline{5432}$ → now just +1 to get 10's complement

↳ Diminished Raddix

⊛ ⊛ $(04567)_{10}$ ⇒ 10's : $10^5 - 4567$

Ex: $(10101101)_2$

⇒ 2's : $2^8 - N = \underline{1010011}$

⇒ 1's : $(2^8 - 1) - N$

$= 11111111$

$- 10101101$

$\underline{01010010}$

NOTE: $2^n - 1 = \underbrace{11\ldots11}_{n}$

◦] Subtraction using complements:

→ for M-N do:

1] M + r's complement of N ⇒ $M + (r^n - N)$

2] M > N → discard the carry (last wala only)

3] M < N : $r^n - (N-M)$ ⇒ r's complement of (N-M) (Ex: on next pg)

Ex: $\overset{M}{72532} - \overset{N}{3250}$

⊛ Equalize no. of digits first ⇒ 03250

⇒1] 9's : $(10^5 - 1) - 3250$

$= 99999 - 3250 = 96749 \Rightarrow 10^s = 96750$

⇒ 72532

$+ 96750$

$\cancel{*}69282$ ⇒ $\underline{72532 - 3250 = 69282}$

__/__/____

Ex: 3250 - 72532 → 10's : 27468 → 03250
+ 27468
30718

* Now answer is -ve of 10's of → -69282
⇒ -69282

Ex: $(1010100)_2 - (1000011)_2$
→ 1010100
= 0111101 ← 2's
*0010001 ⇒ $(10001)_2$

Ex: $(1000011)_2 - (1010100)_2$
→ 1000011 ← 2's
+ 0101100
1101111 → $-(0010001)_2$ ← 2's

*] SIGNED & UNSIGNED SYSTEM:

	Unsigned	Signed	
Sign Bits when system declared: 0 10101	21	+21	→ 0 = +ve
1 10101	53	-21	1 = -ve

*] BCD: BINARY CODED DECIMALS:

→ Arithemaric using signed systems

Ex: +6 } always use 8 bits → 00000110
+13 } + 00001101
19 00010011

Ex: -6 (2's of +6) 11111010
+13 * 00001101
7 00000111
↳ discard carry

-6 -13 → 11111010
* 11110011
-19 = 11101101 → Cross checking: 2's of 19 = 11101101

→ When you arrive at answer if it starts with 1 its the 2's complement and hence specify
→ If number itself is 8 bits take 4 more

P.T.O.

→ In BCD each digit to be represented binary

Ex: $16 = (10000)_2 = (0001\ 0110)_{BCD}$

*] Ex-3: Excess 3 Codex:

→ +3 to all digits i.e. 0 = 0011 (as in 3 in binary)
lly, each digit represented

o] BCD Arithematic:

1] Addition:

Ex: 764 + 856 = 1620 ⇒ 0111 0110 0100 + 1000 0101 0110 = 1111 1011 1010

→ these digits can't be represented using BCD thus add 6 to them to shift them to next system that can't be represented only

0110 0110 0110 + 0001 0110 0010 0000 = 1 6 2 0

Ex: 56 + 44 = 100 ⇒ 0101 0110 + 0100 0100 = 1001 1010

* No discarding carries

yaha kabhi 18, 17, 16 bana raha ho to take 5 bits

1001 1010 + 0110 = 1010 0000 + 0110 = 0001 0000 0000

as 1010 again can't be represented

∴ 0001 0000 0000

2] Subtraction:

Ex: 1001 0011 - 0101 0110

→ Use radix complement method, but * find 10's compliment of each 4-bit digit,

- 9's complement of a BCD number can be obtained by subtracting it from 9.
- 10's = 9's + 1

o] 10's complement method: A - B

1] Find 10's of B.

2] Add A & 10's complement of B.

3] Discard carry.

4] If carry not produced, no. is -ve take 10's of result

5] Validate BCD no. by +6 if necessary.

→ 10's of 0101 0110 = 1001 - 0101 + 0100 + 0001 = 0101 0100

Don't do this first find 9's: 1001 0011 - 0100 0100 = 0100 0011 + 0000 0001 = 0100 0100

→ 1001 0011 + 0100 0100 = 1101 0111 → validate ke baad -ve check

+ 0110

✗ 0011 0111

∴ 1001 0011 - 0101 0110 = 0011 0111

__/__/____

*] LOGIC GATES:

→ AND, OR, NOT, NAND, NOR, XOR, XNOR, etc.

→ self-explanatory → universal Gates: can be used to form every other gate.

◦] Closure:

- A closure set for an operator is the set of which when elements operated upon gives result from the same set.

◦] Associative law: $(x \square y) \square z = x \square (y \square z)$

◦] Commutative law: $x \square y = y \square x$

◦] Identity element:

- Depends on operator: $x \square e = x$

$\Rightarrow$ for * $e = 1$ & + $e = 0$

◦] Inverse: $x \square y =$ identity element

◦] Distributive law: $x \square (y * z) = (x \square y) * (x \square z)$

*] BOOLEAN ALGEBRA:

- Given in 1854 by George Boole → Boolean
- Two-valued boolean algebra given by Claude E Shannon 1938

◦] EV Huntington Postulates of 1904:

① + & * are closure in boolean algebra.

② Identity elements: $+ \longrightarrow 0$, $* \longrightarrow 1$

③ $x+y = y+x$, $xy = yx$

④ $x(y+z) = xy + xz$

⑤ $x + \bar{x} = 1$, $x\bar{x} = 0$

⑥ $\exists$ 2 elements $\in B$ s.t. $x \neq y$

◦] Associative law:

- $(x \square y) \square z = x \square (y \square z)$

★ ◦] Distributive law:

$$\left[x + (y * z) = (x+y) * (x+z) \right]$$

→ Boolean algebra does not have additive/multiplicative inverses due to the absence of subtraction & division.

__/__/____

0] **Duality Principle:**

→ $x + 0 = x \longleftrightarrow x \cdot 1 = x$, i.e. operator change can be carried out through changing identity.

0] **Postulates:**

1] $x+0=x$ & $x \cdot 1 = x$

2] $x + \bar{x} = 1$ & $x\bar{x} = 0$ (He writes x' so prefer x' over $\bar{x}$)

3] $x+y=y+x$ & $x \cdot y = y \cdot x$

4] $x(y+z) = xy + xz$ & $x + (yz) = (x+y)(x+z)$ ∵ $(x+y)(x+z)$
$= xx + xz + xy + yz$
$x(1+z) + xy + yz$
$x + xy + yz \Leftarrow x + yz$

0] **Theorems:**

1] $x+x=x$ & $x \cdot x = x$

2] $x+1=1$ & $x \cdot 0 = 0$

3] $(\bar{x}) = x$

4] $x + (y+z) = (x+y)+z$ & $x(yz) = (xy)z$

De-Morgan 5] $\overline{(x+y)} = \bar{x} \cdot \bar{y}$ & $\overline{(xy)} = \bar{x} + \bar{y}$

Absorption 6] $x + xy = x$ & $x(x+y) = x$

0] **Operator Precedence:**

1] Parantheses

2] NOT

3] AND

4] OR

Q] Simplify $\bar{x}\bar{y}z + xyz + \bar{x}yz + x\bar{y}z$.

→ $\bar{x}\bar{y}z + yz(x+\bar{x})^{*1} + x\bar{y}z$

$= \bar{x}\bar{y}z + yz + x\bar{y}z \Rightarrow \bar{y}z(\bar{x}+x)^{*1} + yz$

$= yz + \bar{y}z \Rightarrow z(y+\bar{y})^{*1} = z$

∴ $\bar{x}\bar{y}z + xyz + \bar{x}yz + x\bar{y}z = z$

---

*] **Half Adder & Full Adder:**

- SUM → XOR
- CARRY → AND

Half Adder

Half Adder = performs add^n of 2 binary bits

Full Adder = ——— 3 ———

↳ SUM = $A \oplus B \oplus C$

CARRY = $(AB) + (C \cdot (A \oplus B))$

__/__/____

*] Complement of a Function:

- $F = A+B+C$

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

Ex: $F = xyz + x'yz \Rightarrow F' = (x'+y'+z') \cdot (x+y'+z')$

*] Canonical Form:

x	y	z	canonical form using min terms	canonical forms using Max terms
0	0	0	$x'y'z' \rightarrow m_0$	$x+y+z \rightarrow M_0$
0	0	1	$x'y'z \rightarrow m_1$	$x+y+z' \rightarrow M_1$
0	1	0	$x'yz' \rightarrow m_2$	$x+y'+z \rightarrow M_2$
0	1	1	$x'yz \rightarrow m_3$	$x+y'+z' \rightarrow M_3$
1	0	0	$xy'z' \rightarrow m_4$	$x'+y+z \rightarrow M_4$
1	0	1	$xy'z \rightarrow m_5$	$x'+y+z' \rightarrow M_5$
1	1	0	$xyz' \rightarrow m_6$	$x'+y'+z \rightarrow M_6$
1	1	1	$xyz \rightarrow m_7$	$x'+y'+z' \rightarrow M_7$

Ex: Truth table of func is: 0 1 0 0 1 0 0 1

$\Rightarrow F_1(\text{minterms}) = x'y'z + xy'z' + xyz$

Truth value of F=1

$\therefore F_m = \sum(m_1, m_4, m_7) = m_1 + m_4 + m_7$

Ily, $F_M = \Pi(M_0, M_2, M_3, M_5, M_6)$

Truth value of F=0

↳ To derive write $(F'_m)'$

•] Conversion to canonical form:

Ex: $F = xy + x'z$ as product of max terms

$\Rightarrow F = (xy + x') \cdot (xy + z) \quad \because (x+yz = (x+y) \cdot (x+z))$

$= (x+x') \cdot (x'+y) \cdot (x+z) \cdot (y+z)$

$= (x'+y+zz') \cdot (x+z+yy') \cdot (z+y+xx')$

$= (x'+y+z) \cdot (x'+y+z') \cdot (x+y+z) \cdot (x+y'+z)$

$M_4 \quad M_5 \quad M_0 \quad M_2$

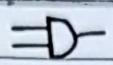
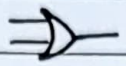
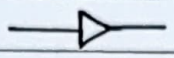
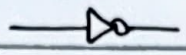
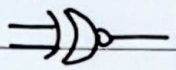
Ex: $F = xy + x'z$ as sum of min terms

$= xy(z+z') + x'z(y+y')$

$= xyz + xyz' + x'yz + x'y'z$

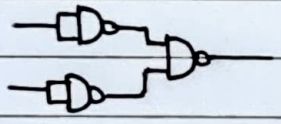
$m_7 \quad m_6 \quad m_3 \quad m_1$

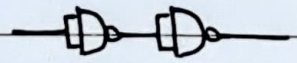
*] Logic Gates contd.:

#	Gate	Representation	Output
	AND (·)		0 0 0 0 1 0 1 0 0 1 1 1
	OR (+)		0 0 0 0 1 1 1 0 1 1 1 1
	Buffer/Data		0 0 1 1
	NOT		0 1 1 0
	NAND (↑)		0 0 1 0 1 1 1 0 1 1 1 0
	NOR (↓)		0 0 1 0 1 0 1 0 0 1 1 0
	XOR ⊕ Non-equivalence		0 0 0 0 1 1 1 0 1 1 1 0
	⊙ XNOR equivalence		0 0 1 0 1 0 1 0 0 1 1 1

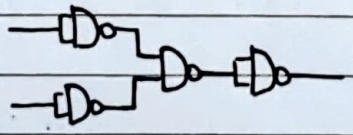
o] All gates using NAND:

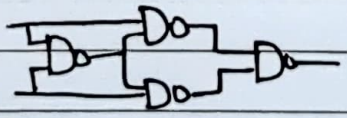
1] AND - 

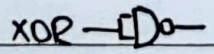
2] OR - 

3] Buffer - 

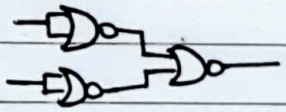
4] NOT - 

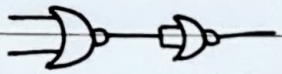
5] NOR - 

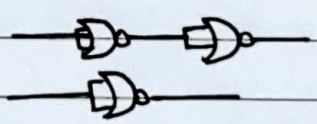
6] XOR - 

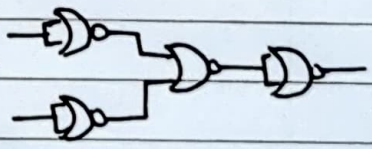
7] XNOR - XOR 

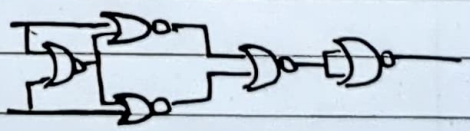
o] All gates using NOR:







5) NAND - 



XOR

__/__/____

weighted codes

#	BCD (or 8421)	Excess - 3	84-2-1 (=a×8+b×4+c(-2)+d(-1))	2421 (=a×2+b×4+2×c+d×1)
	0000	0011	0000	0000
	0001	0100	0111	0001
	0010	0101	0110	0010
	0011	0110	0101	0011
	0100	0111	0100	0100
	0101	1000	1011	1011
	0110	1001	1010	1100
	0111	1010	1001	1101
	1000	1011	1000	1110
	1001	1100	1111	1111
unused	1010	0000	0001	0110
	1011	0001	0010	0111
	1100	0010	0011	1000
	1101	1101	1100	1001
	1110	1110	1101	1010
	1111	1111	1110	0101

*] Gray Code:

- Only one bit can change blw 2 consec no.s

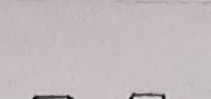
⇒ 0 = 0000 5 = 0111 10 = 1111 15 = 1000
1 = 0001 6 = 0101 11 = 1110
2 = 0011 7 = 0100 12 = 1010
3 = 0010 8 = 1100 13 = 1011
4 = 0110 9 = 1101 14 = 1001

*] ASCII charachter code: (Key board)

does not envelop all languages ↓ unicode

- Letters & symbols associated to no.s
- @ = 64 then 65 to 90 = A to Z
- [= 91, \ = 92,] = 93, ^ = 94, _ = 95, ` = 96
- 97 to 122 = a to z
- { = 123, | = 124, } = 125, ~ = 126 ⇒ 7 bits

Backspace = 8, Tab = 9, ESC = 27, Enter = 13, Delete = 127

timing diagram =  → ye wala diagram

*] Even & Odd Parity:

Ex: 1 1010100 → ASCII
↳ parity bit ⇒ no. of 1's = even ⇒ even parity
= odd ⇒ odd parity

•] Parity Error Detectors:

- If parity of input changes ⇒ error
- Fails to detect error if change in even bits occurs as parity stays same. Ex: 1011 → 1101 (error but stays odd so not caught)

*] Negative Logic:

- kuch alag nahi hai except | Boolean True → 0 | Boolean False → 1

*] Karnaugh-Maps (K-Maps):

→ Gray code followed

•] 2-variable K-Map:

x \ y	0	1
0	m_0	m_1
1	m_2	m_3

•] 3-variable K-Map:

x \ yz	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

→ Gray Code

Ex:

x \ yz	00	01	11	10
0	1			1
1	1		1	1

① Draw circles enclosing the 1's such that maximum number of 1's are included in it & $\#_{1s} = 2^n, n \in \mathbb{N}$

② Now across that circle some of the variables would be constant, those would be the necessary minterms

③ Do not circle diagonally.

④ Can go across edges.

•] 4-variable K-Map:

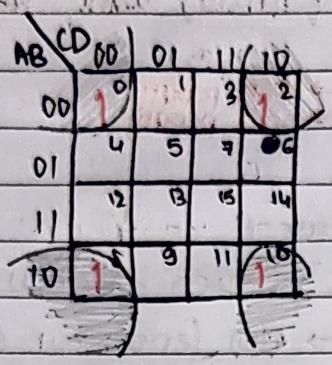
wx \ yz	00	01	11	10
00	m_0	m_1	m_3	1 (m_2)
01	1 (m_4)	1 (m_5)	m_7	1 (m_6)
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	1 (m_{11})	m_{10}

⇒ $w'yz' + w'xy' + wx'yz$ + $w'xz'$ (from m_4, m_6 circle)

termed as prime implicants, essential but unnecessary as minimum circle terms doesn't change func^n

__/__/__

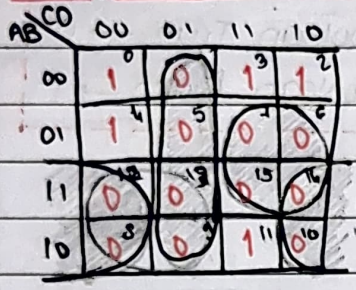
(#)



→ This circle of four is valid

↳ B'D'

•] K-Map to Max-terms (POS):



$F = \sum (0,2,3,4,11)$

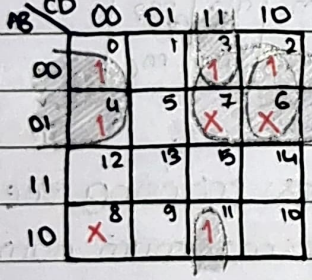
$= \pi (1,5,6,7,8,9,10,12,13,14,15)$

$\Rightarrow F' = AD' + BC + C'D$

$\Rightarrow (F')' = (A'+D)(B'+C')(C+D')$

$=$ POS

•] "Don't care" conditions in K-Maps:



$\Rightarrow F = \sum (0,2,3,4,11) + d(6,7,8)$

don't care → can make use with both 0 & 1 otherwise can ignore as well

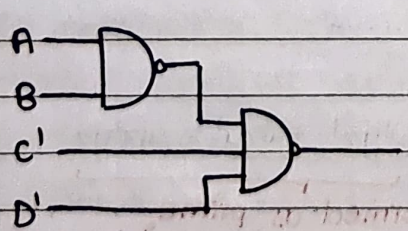
$= A'C + A'D' + B'CD$

*] Gate Level Implementations:

•] NAND Implementation:

Ex: AB + C + D

→ Reduce it to NAND using De-Morgan's → $((AB)'(C)'(D)')'$

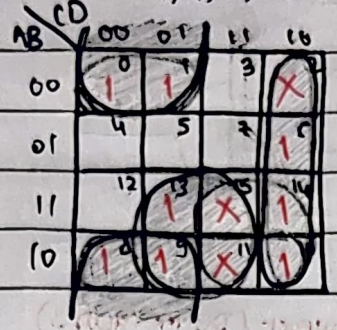


= NAND Implentation of AB+C+D

__/__/____

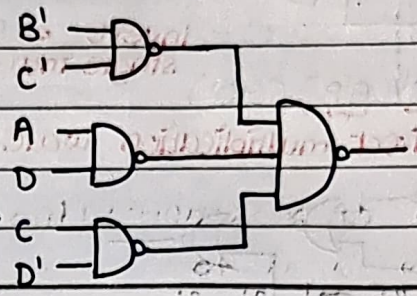
0] Implement : $F = \Pi(3,4,5,7,12) + d(2,11,15)$ using NAND

→ $F = \Sigma_i(0,1,X,6,8,9,10,X,13,14,X) + d(2,11,15)$

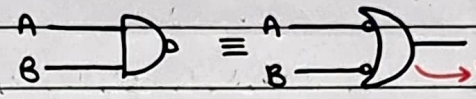


$= CD' + AD + B'C'$

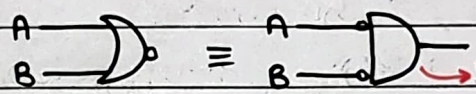
$\Rightarrow ((CD')' \cdot (AD)' \cdot (B'C')')'$



NOTE: ① **NAND:**



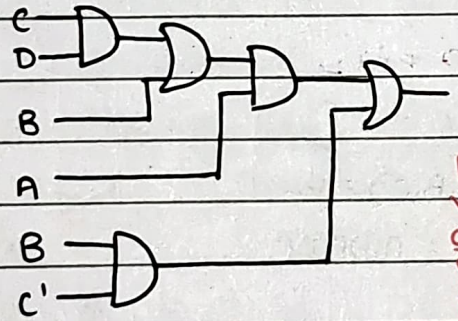
② **NOR:**



° **Multilevel to NAND directly:**

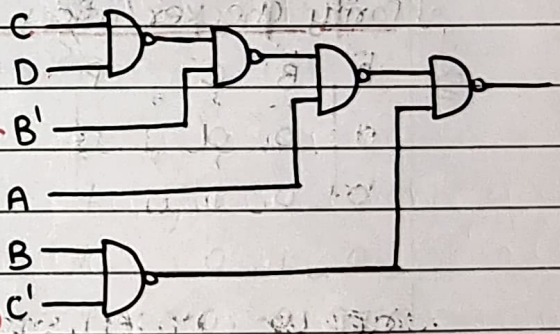
Ex: $F = A(CD+B) + BC'$

AND-OR



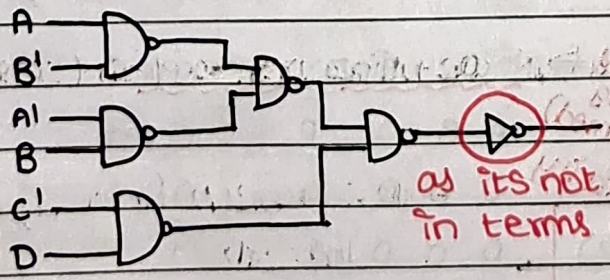
≡

NAND



Invert the variable directly going to OR in NAND

(*) **Ex:** $F = (AB' + A'B)(C+D')$ → isolated into P



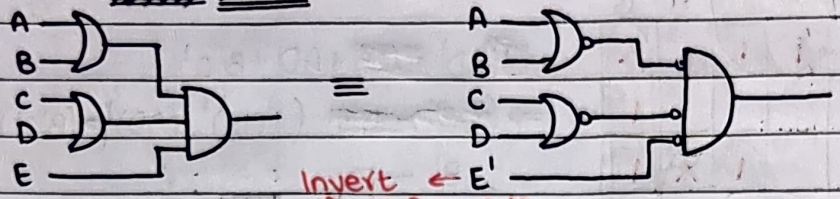
as its not in terms of SOP & is ending in terms of '&' ∴ NOT the NAND

__/__/____

◦] NOR Implementation:

Ex: $F = (A+B)(C+D)E$

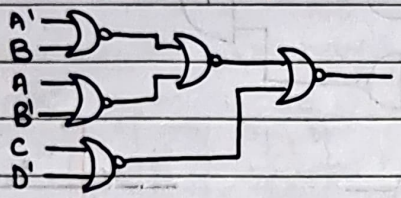
OR-AND NOR



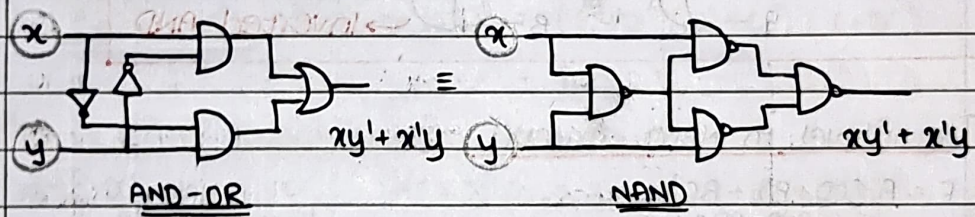
Invert ← E' single into AND

Ex: $F = (AB' + A'B)(C+D')$

direct multiplication wale bhi invert (De-Morgan's)



*] X-OR:



AND-OR NAND

◦] Parity Checker using XOR:

Ex:

A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	1
1	0	0	1
0	1	1	0
1	0	1	0
1	1	0	0
1	1	1	1

$\Rightarrow F(A,B,C) = P$

$\therefore P = A \oplus B \oplus C$

$\therefore$ Parity $= A_1 \oplus A_2 \oplus A_3 \oplus \ldots\ldots$

parity i.e. og (assumed) → even parity

$\therefore$ Checker $= (A \oplus B \oplus C \oplus P)$

$= (A \oplus B \oplus C \oplus P)'$

↳ odd parity (assumed) i.e. actual parity

→ Now, using checker:

A	B	C	P	Checker
0	0	0	0	0
0	0	0	1	1
1	1	1	0	1
1	1	1	1	0

→ just XORs

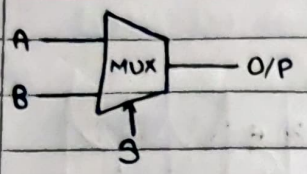
→ Checker 0 aaya → no parity error

-] Combinational circuits:
- Circuits without feedback, storage elements

__/__/____

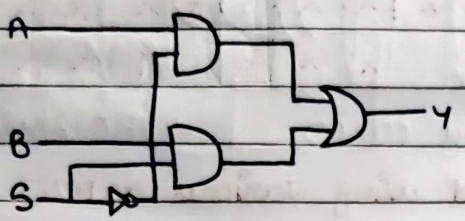
→ De-Mux = exact ulta

*] Multiplexers:



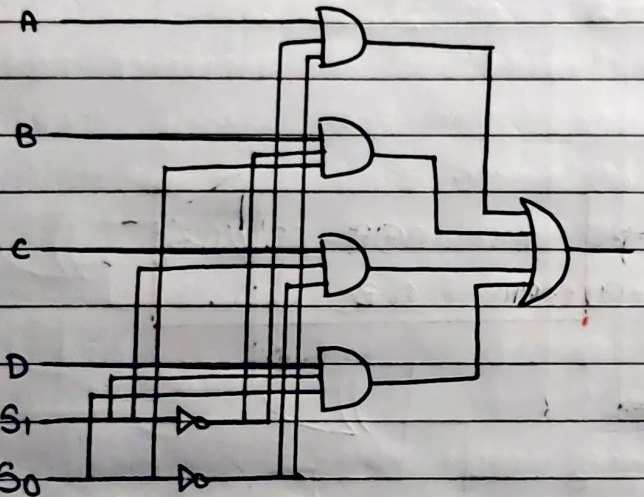
↳ Types: n × m → mux
inputs ↵ ↳ outputs

Ex: 2×1 MUX:



S	Y
0	A
1	B

Ex: 4×1 MUX



S_1S_0	A	B	C	D
0 0	1	0	0	0
0 1	0	1	0	0
1 0	0	0	1	0
1 1	0	0	0	1

$\Rightarrow S_1'S_0' + S_1'S_0 + S_0'S_1 + S_0S_1$

$\Sigma(0,1,2,3)$

* minterms

-] Implementing logic Gates using MUXs:

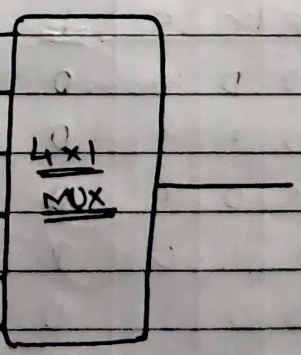
Ex: $F(x,y,z) = \Sigma(1,2,6,7)$

S_1 x	S_0 y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

→ order important, if change order fails

0 → 0 ; 1 → z ; z' → z' ; 1 → 1

$\equiv x(S_1)\ y(S_0)$



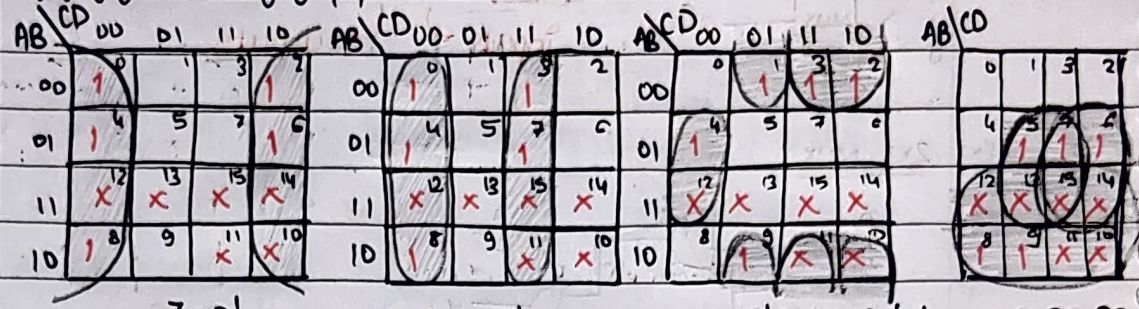
→ Output in terms of 1 variable ∴ I/P = variable, variable', 0, 1
& other two variables act as selection lines.

__/__/____

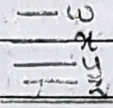
BCD to Ex-3 → don't care conditions: (10,11,12,13,14,15) as not used

A B C D → w x y z

0 0 0 0 ... 0 0 1 1



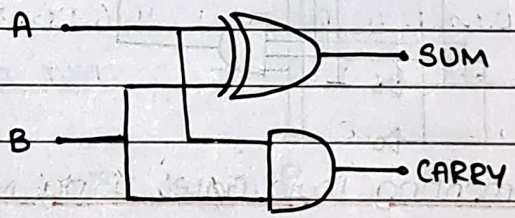
→ Now can be implemented using various gates



*] Adders:

◦] Half Adder:

A	B	S	C_1	
0	0	0	0	⇒ SUM = A⊕B = AB' + A'B
0	1	1	0	⇒ CARRY = A·B
1	0	1	0	
1	1	0	1	⇒



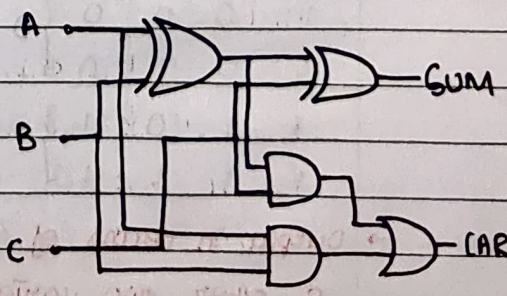
◦] Full Adder:

$C_{out} = C \cdot (A \oplus B)$

A	B	C	S	C_2	$C_{out} = C_1 + C_2$
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	1	1
1	1	0	0	0	1
1	1	1	1	0	1

⇒ SUM = A⊕B⊕C

$C_{out} = C \cdot (A \oplus B) + A \cdot B$

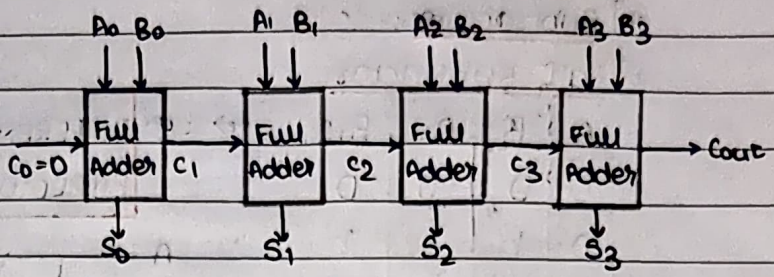


__/__/____

→ lly. n bits

• 4-bit Binary Adder:

Ex: Cout C3 C2 C1 C0
1 1 1 0 0 ← Cin
A → A3 A2 A1 A0 → by default
0 1 1 0
B → B3 B2 B1 B0
1 0 1 1
0 0 0 1
S3 S2 S1 S0



Also called Ripple adder

↳ carry propogation problem: → for an n-bit adder, there are 2n gate levels for carry to propogate

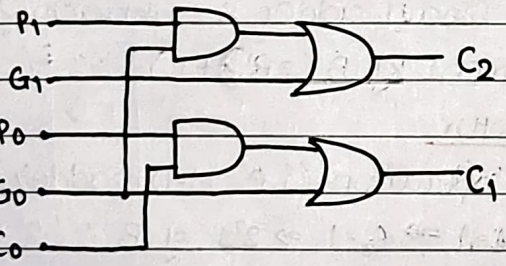
- The gates succeeding are delayed due to the time taken to forward the carry.

• Carry lookahead Adder: (To solve), but ↑ hardware complexity

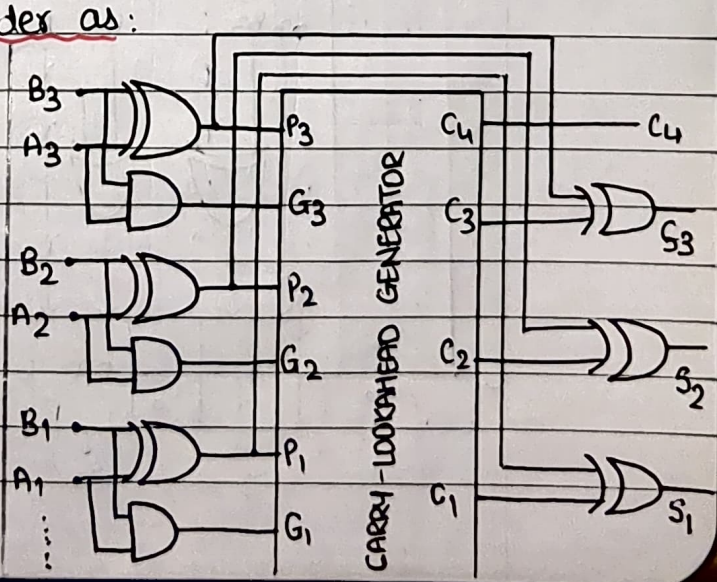
- $C_3\ C_2\ C_1\ C_0$ $P_i = A_i \oplus B_i$, $G_i = A_i \cdot B_i$ → carry generate
  $A_3\ A_2\ A_1\ A_0$ $S_i = P_i \oplus C_i$, $C_{i+1} = P_i C_i + G_i$
  $B_3\ B_2\ B_1\ B_0$ carry propogate

⇒ $C_1 = (A_0 \oplus B_0) C_0 + A_0 B_0$, $C_2 = (A_1 \oplus B_1)((A_0 \oplus B_0) C_0 + A_0 B_0) + A_1 B_1$, ...

- As 2^n for each output carry expressed as SOP & only dependant on P & G, ⇒ one level AND followed by OR (or 2-level NAND)
- ★ This circuit can add in less time as C_2 does not have to wait for C_1 to propogate, it propogates @ the same time.



- We can make 4-bit adder as:
- Each sum requires 2 XORs.
- output of 1st gives P_i, AND gives G_i
- carry propogated through generator, applied as inputs to second XOR
- All carries generated only after delay of 2-gate levels
- $S_1 S_2 S_3$ equal propogation delay times



__/__/__

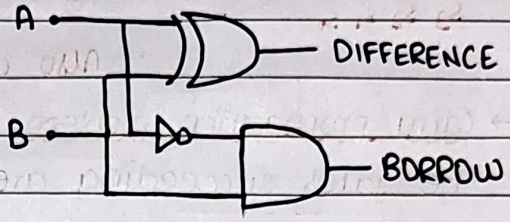
*] Subtractors:

o] Half-Subtractor:

A	B	D_1	B_1
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

→ DIFFERENCE = $A \oplus B$

BORROW = $A' \cdot B$

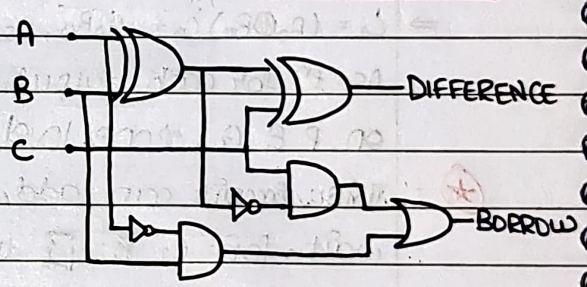


o] Full-Subtractor:

A	B	C	Diff	B_1	B_2	$B_{out} = B_1 + B_2$
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	1	0	1	1

⇒ DIFFERENCE = $A \oplus B \oplus C$

BORROW = $(A \oplus B)' \cdot C + A' \cdot B$



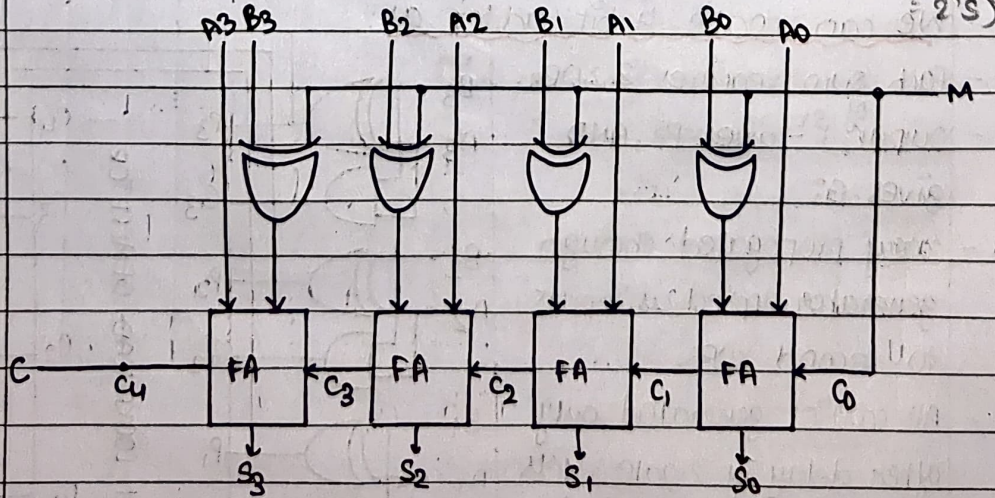
Q] Can we combine the binary adder & subtractor?

→ A-B = A + 2's complement of B ($\equiv B' + 1$)

↳ B' = 1's

⇒ Binary Adder-Subtractor:

- Mode input M controls operation (i.e. M=0 ⇒ adder, M=1 ⇒ subtractor)
- M=0 ⇒ C_0=0 ⇒ B, M=1 ⇒ C_0=1 ⇒ 2's of B ∵ (B' = 1's + C_0 = 2's)



__/__/____

*] The BCD Adder:

BCD → carry

- Sum ranges from 0 to 19 (9+9+1)
- If sum upto 1001, BCD ✓, if > 1001 +6 (produces output carry)

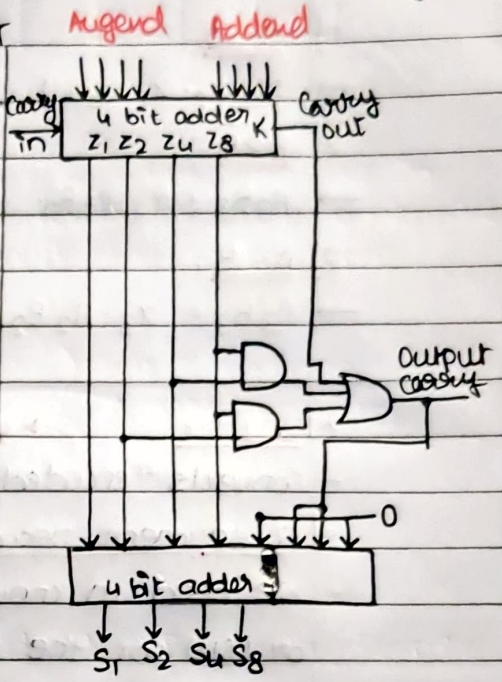
→ A+B → A+B $\overset{>9?}{\underset{\text{YES}}{\overset{\text{NO}\rightarrow +6}{}}}$ → Output

Binary sum					BCD sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
First 9										
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19
same										9

- Obvious that correct needed when K = 1.
- 1010 to 1111 also need correct, 1 in Z_8, To distinguish them from 1000 & 1001 we further specify either Z_4/Z_2 must have 1.

→ $Cor = K + Z_8Z_4 + Z_8Z_2$

↳ When = 1 ⇒ +6



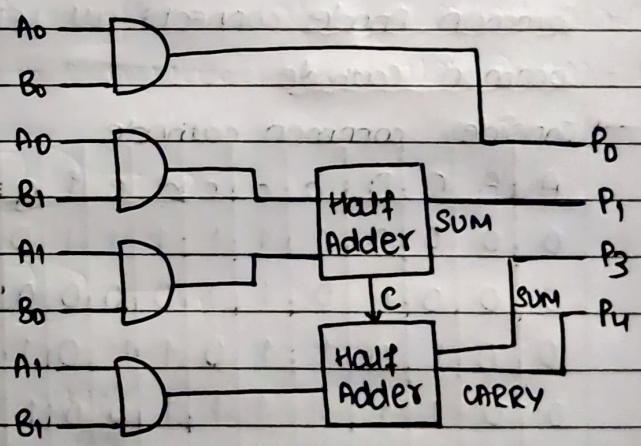
*] Multipliers:

Ex: B_1 B_0 × A_1 A_0

A_0B_1 A_0B_0

A_1B_1 A_1B_0

if carry add using HA as 2 bits



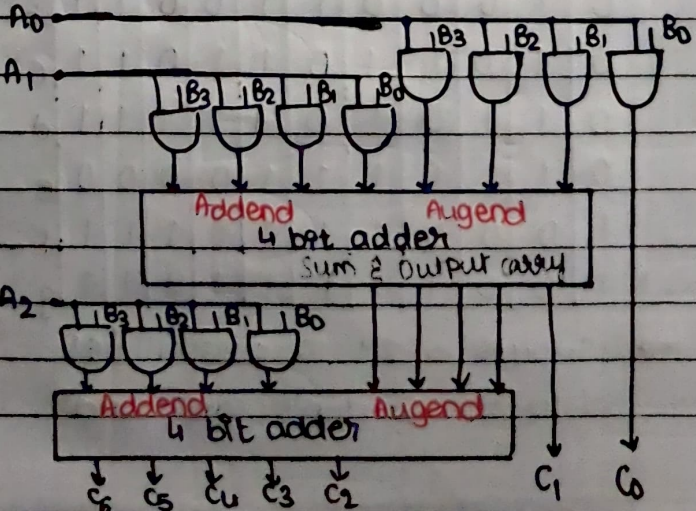
Ex: B_3 B_2 B_1 B_0 × A_2 A_1 A_0

A_0B_3 A_0B_2 A_0B_1 A_0B_0

A_1B_3 A_1B_2 A_1B_1 A_1B_0 ×

A_2B_3 A_2B_2 A_2B_1 A_2B_0 × ×

Full Adder Req



__/__/____

*] Binary Comparator:

- For $A = A_3\ A_1\ A_2\ A_0$ & $B = B_3\ B_2\ B_1\ B_0$

  Let $x_i = A_iB_i + A_i'B_i'$

1] $A \equiv B$:

→ $x_3 \cdot x_2 \cdot x_1 \cdot x_0 = 1$

2] $A > B$:

→ $A_3B_3' + x_3 \cdot A_2B_2' + x_3 \cdot x_2 \cdot A_1B_1' + x_3 \cdot x_2 \cdot x_1 \cdot A_0B_0' = 1$

3] $A < B$:

→ $A_3'B_3 + x_3 \cdot A_2'B_2 + x_3 \cdot x_2 \cdot A_1'B_1 + x_3 \cdot x_2 \cdot x_1 \cdot A_0'B_0 = 1$

*] Decoder:

→ going from one representation to another = encoding/decoding

- converts "encoded" inputs into outputs (basically spits out minterms)

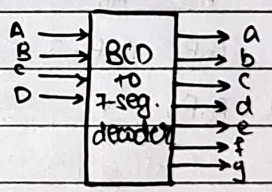
0] 7-segment decoder: (from BCD)

- We need to know whether a particular input should be HIGH or LOW for the LED to glow.
- 2 types: Common Anode & Common Cathode
- Common Anode connects all LEDs to +VCC ⇒ input LOW = LED glow
- Common Cathode ——— GND ⇒ input HIGH = ———

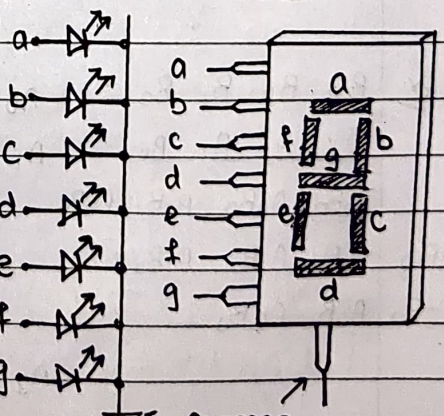
Photo google karlo

→ Consider common cathode:

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1



Common Cathode:



→ 4 inputs ⇒ $2^4 = 16$ possibilities

∴ 10-15 = 6 don't care condn.

__/__/____

⇒ K-Map for output a:

AB \ CD	00	01	11	10
00	1 (0)	0 (1)	1 (3)	1 (2)
01	0 (4)	1 (5)	1 (7)	1 (6)
11	X (12)	X (13)	X (15)	X (14)
10	1 (8)	1 (9)	X (11)	X (10)

⇒ $F_a = A + C + BD + B'D'$

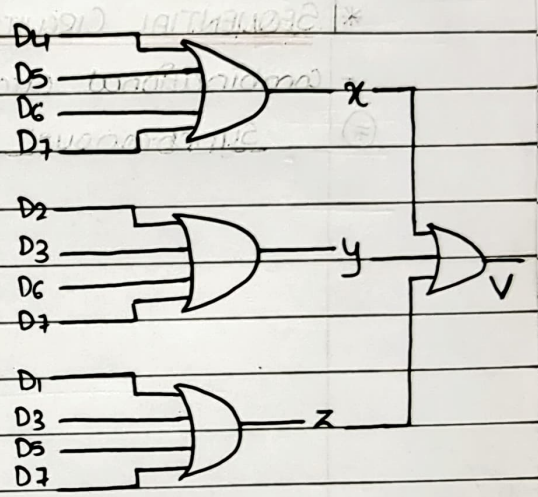
$= (B'+C+D)(A+B+C+D')$

⇒ Same can be done for each output segment & thus 7-segment decoder can be implemented.

*] Encoder:

- 2^n inputs ⟶ n outputs

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



→ This simple encoder is a little ambiguous as it gives 000, for all outputs 0 & various outputs for cases with situations with more than one input being 1. Thus, a priority encoder is required.

*] Priority Encoder:

- Priority is given to highest subscript

D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

⇒ $V = D_0 + D_1 + D_2 + D_3$

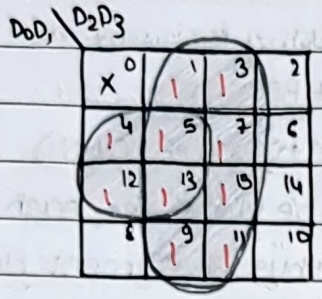
∴ K-Mapping: (x)

D_0D_1 \ D_2D_3	00	01	11	10
00	X (0)	1 (1)	1 (3)	1 (2)
01	(4)	1 (5)	1 (7)	1 (6)
11	(12)	1 (13)	1 (15)	1 (14)
10	(8)	1 (9)	1 (11)	1 (10)

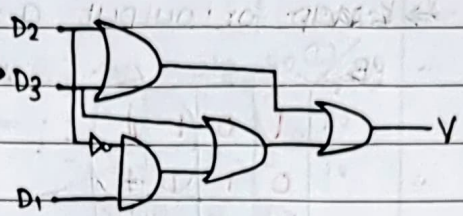
$x = D_2 + D_3$ ⟸

__/__/____

- lly, for y:



$\Rightarrow y = D_3 + D_1D_2'$



Priority Encoder

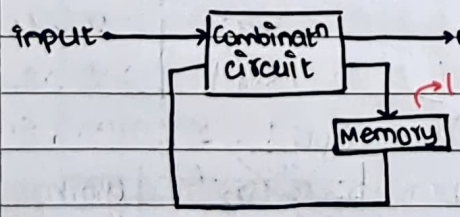
MID SEMESTERS UNTIL HERE

*] **SEQUENTIAL CIRCUITS:**

- Combinational circuit + Memory (Storage Elements)

# Synchronous Circuit	Asynchronous Circuit
- All state variable changes are synchronized with a universal clock signal.	- Are not synchronized to change simultaneously, change irrespective of each other.
- Under strict control of a master clock and are thus less prone to failure or to a race condition & hence are more reliable	- Internal state changes as soon as any of the inputs change and hence are more prone to a race condition.

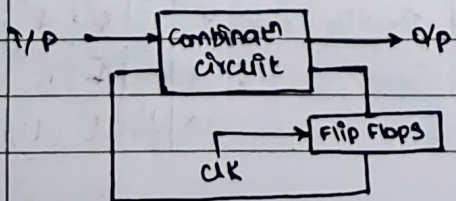
#



↳ 1 Memory element stores 1-bit

≡ Sequential circuit

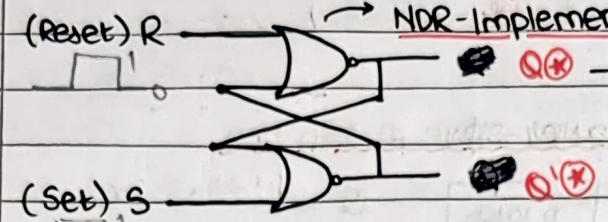
- Clock used to make synchronous circuits stable.
- The above block diagram is of a typical asynchronous circuit.
- Following is the block diagram for synchronous circuits:



- clock represented as clk.

__/__/____

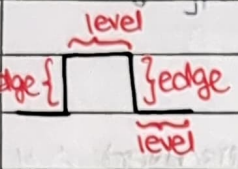
*] Latch:

- 

→ NOR-Implementation of SR latch:
→ Function Table:

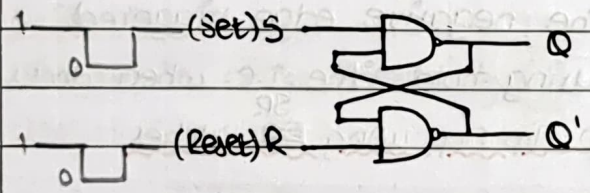
S	R	Q	Q'
1	0	1	0
0	0	1	0 (after S=1, R=0)
0	1	0	1
0	0	0	1 (after S=0, R=1)
1	1	0	0 (forbidden)

- Ily, NAND as well (func. table diff)

- 

- SR-latches are level triggered,
i.e. change output only on level changes, i.e. 0 to 1 or 1 to 0
- Acts as one bit storage for set & reset = 0,0.

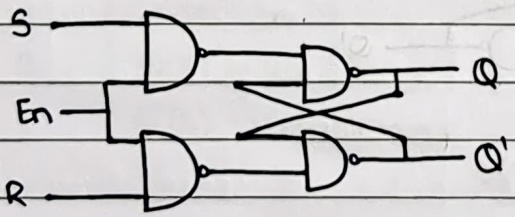
*] NAND-Implementation of SR Latch:

- 

S	R	Q	Q'
1	0	0	1
1	1	0	1 (after S=1, R=0)
0	1	1	0
1	1	1	0 (after S=1, R=0)
0	0	1	1 (forbidden)

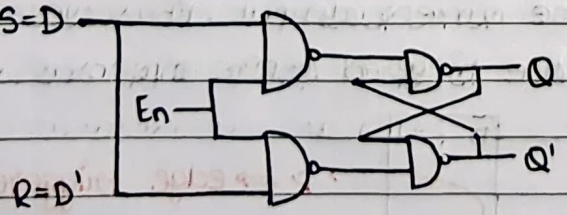
- Acts as one bit storage for set & reset = 1,1.

*] Enabled SR Latch:

- 

En	S	R	Next state Q	
0	x	x	No change	Acts as Memory element
1	0	0	No change	
1	0	1	Q=0, reset state	
1	1	0	Q=1, set state	
1	1	1	Indeterminate	

*] D-Latch: (Transparent Latch)

- 

En	D	Next state Q	
0	x	No change	Acts as Memory element
1	0	Q=0	
1	1	Q=1	

- eliminates indeterminate possibility.

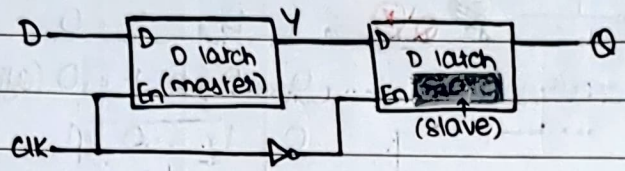
__/__/____

*] Flip-Flops:

- Edge triggered: +ve edge, -ve edge

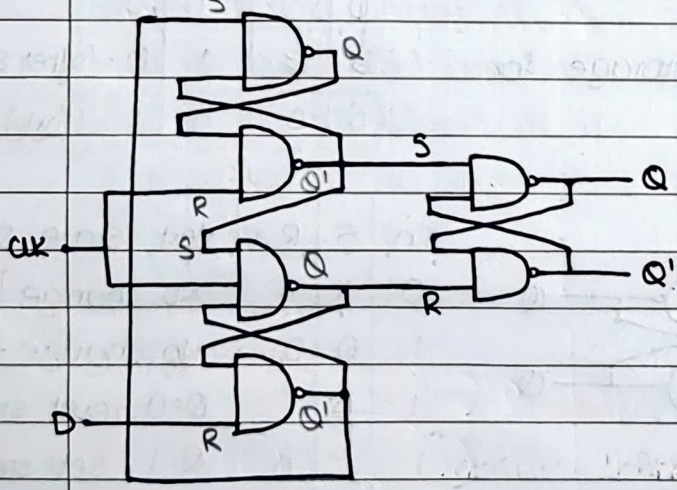
•] D-Flip Flop:

→ -ve edge triggered master-slave D-Flip Flop.



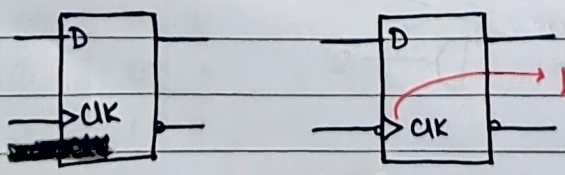
- During +ve edge, the master-D-latch is disabled & Y goes to Q through slave-D-latch.
- During -ve edge the slave D-latch is disabled & Y changes through master-D-latch. This Y propogates during the next +ve edge.
- Thus, it is said to be negative edge triggered.
- The input is held during hold time, i.e. when clocks are on level

→ +ve edge triggered D-Flip Flop using SR latches:



- During -ve edges SR latches go into the 1,1 state, i.e. they don't change the current output. Thus, +ve edge triggered.

→ Graphic Symbol for edge-triggered D-Flip Flop

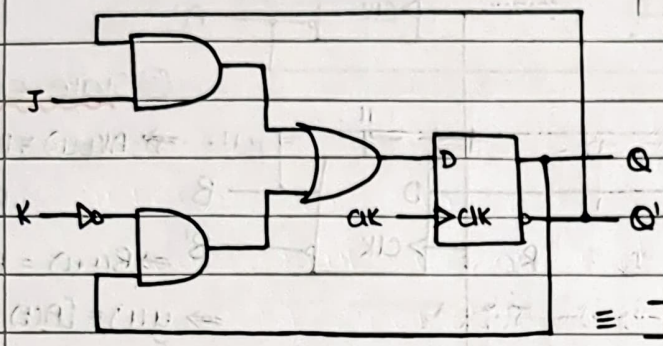


a] +ve edge b] -ve edge

__/__/____

◦] JK-Flip Flop:

- There are four operations we are looking to do in a flip flop. Set to 1, reset to 0, retain or complement.
- D fails to do all 4, (only retain, set, reset)
- Synchronized by a clk, JK uses two inputs to do all 4.
- It follows the charachteristic function: $D = JQ' + KQ$

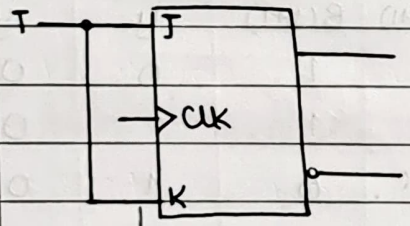


J	K	Q(t+1)
0	0	Q(t) (No Change)
0	1	0 (Reset)
1	0	1 (Set)
1	1	Q'(t) (complement)

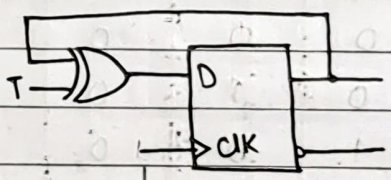
≡

◦] T-Flip Flop:

- T(toggle) is a complementing flip-flop, - It can also be constructed
- Useful for designing binary counters, using a D-flip flop



- $D = T'Q + TQ'$



(*)

T	Q(t+1)
0	Q(t) ⇒ No Change
1	Q'(t) ⇒ Complement

T	Q(t+1)
0	Q(t) ⇒ No change
1	Q'(t) ⇒ Complement

*] Analysis of Sequential Circuits:

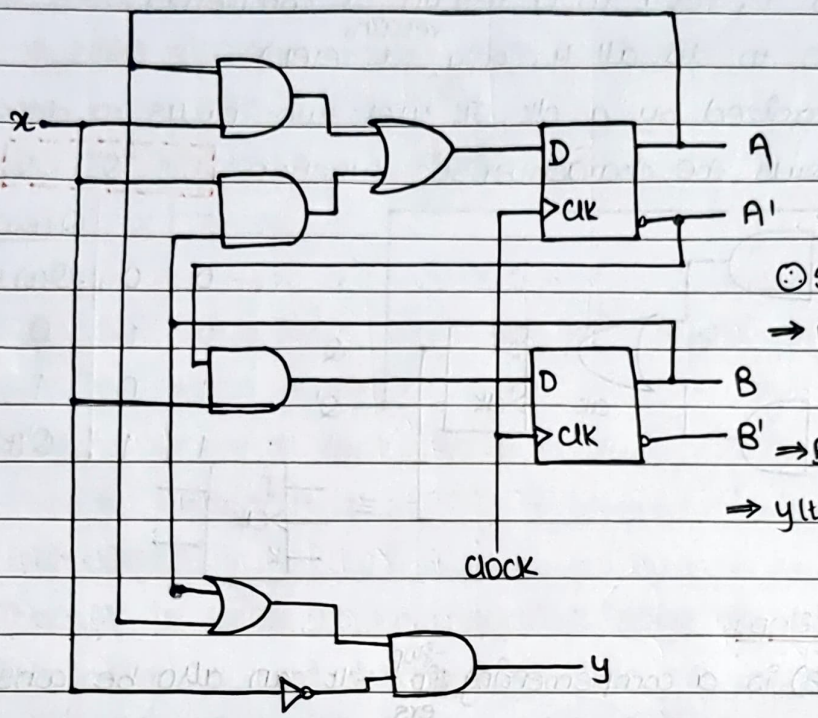
- It consists of obtaining a table or a diagram for the time sequence of inputs, outputs & internal states.
- Write an expression for (t+1) in terms of (t).

P.T.O.

__/__/____

°] Analysis of a sequential circuit with D-flip-flops:

- Consider the following circuit:



☺ State Eqns:

→ $A(t+1) = A(t)x(t) + B(t)x(t)$

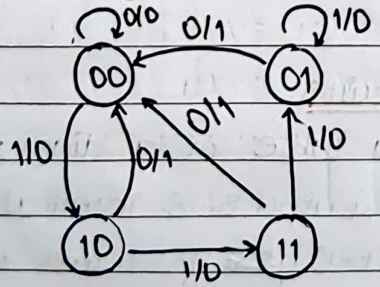
⇒ $B(t+1) = A'(t)x(t)$

⇒ $y(t) = [A(t) + B(t)]x'(t)$

→ Making State Table:

		x = 0		x = 1		x = 0	x = 1
A(t)	B(t)	A(t+1)	B(t+1)	A(t+1)	B(t+1)	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

→ State Diagram:



→ Steps:

1] Derive state diagram
2] Assign binary values to states
3] Obtain binary coded-state table
4] Derive state/output eqns.
5] Draw logic diagram

*] State Reduction:

state	*a*	*a*	*b*	*c*	*d*	*e*	*d*	*d*	*e*	*d*	*e*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Table 5.6
State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

Table 5.7
Reducing the State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*e*	*f*	0	1

Table 5.8
Reduced State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*d*	0	1
e	*a*	*d*	0	1

In fact, this sequence is exactly the same as that obtained for Fig. 5.25 if we replace *g* by *e* and *f* by *d*.

Checking each pair of states for equivalency can be done systematically by means of a procedure that employs an implication table, which consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table.

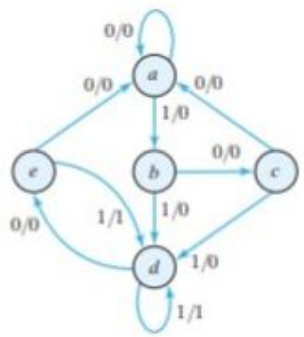
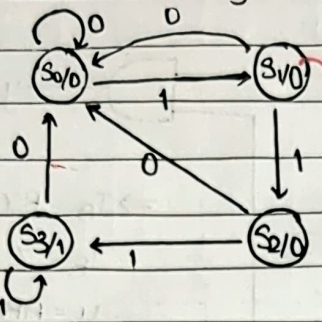


FIGURE 5.26
Reduced state diagram

__/__/____

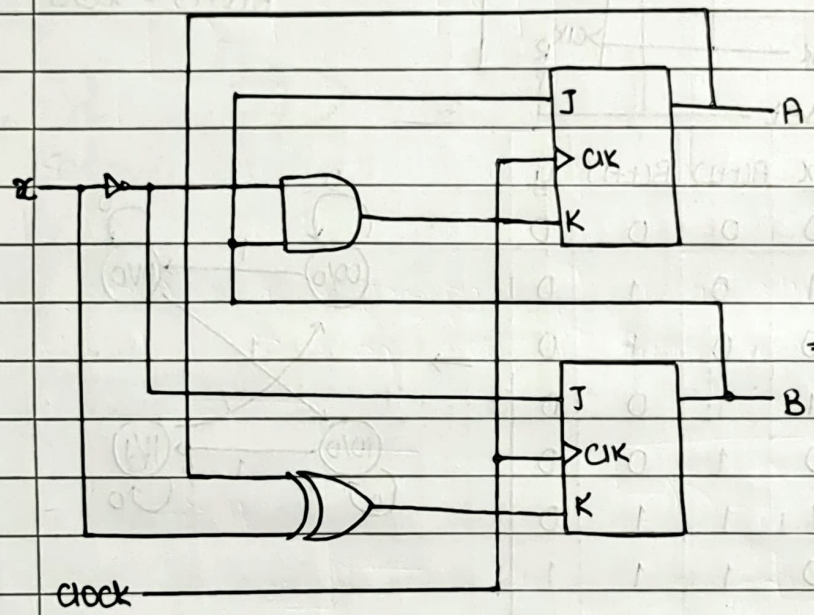
Ex: Draw state diagram for sequence of three detector:



→ If outputs fixed can be represented as such

→ Make table & minimise & implement using K-Maps

1] Analysis with JK Flip-Flops:



$\Rightarrow J_A = B,\ K_A = Bx'$

$J_B = x',\ K_B = A \oplus x$

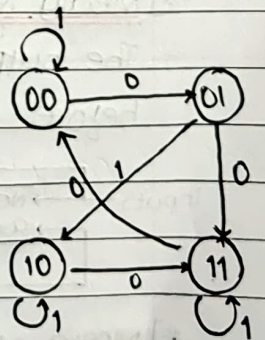
$\Rightarrow A(t+1) = JA' + K'A$

$B(t+1) = JB' + K'B$

⇒

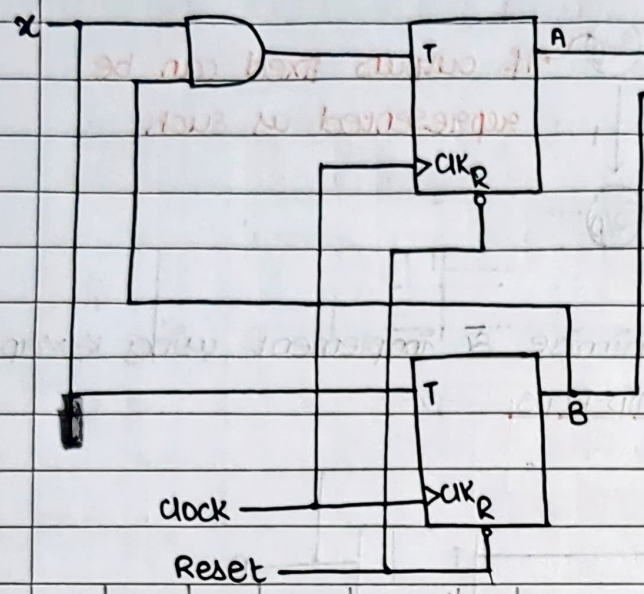
A(t)	B(t)	x	A(t+1)	B(t+1)	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

⇒



__/__/____

Analysis with T Flip-Flops:



$\Rightarrow T_A = Bx$

$T_B = x$

$y = AB$

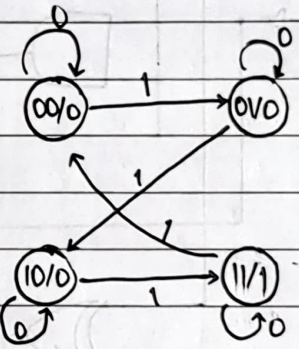
$\Rightarrow A(t+1) = A \oplus T_A$

$B(t+1) = x \oplus B$

$\Rightarrow$

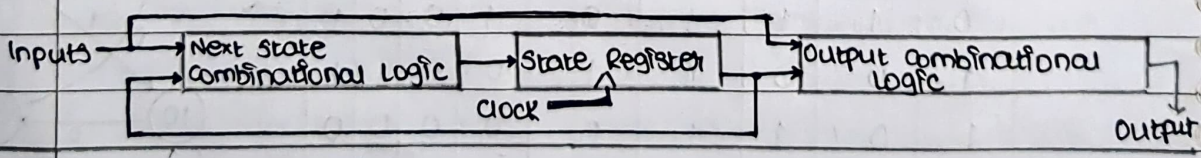
A(t)	B(t)	x	A(t+1)	B(t+1)	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

$\rightarrow$



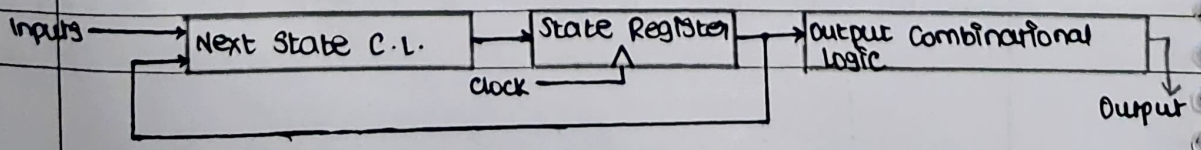
* Mealy Machine:

- The output is the value that is present immediately before the active edge of the clock.



* Moore Machine:

- Outputs synced with clock, as directly dependant on f-f outputs



__/__/__

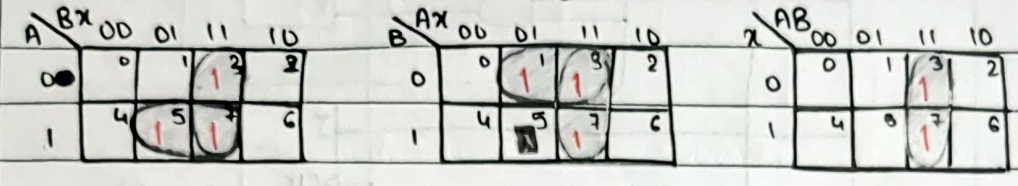
*] Synthesis of [redacted] sequential circuits:

◦] Synthesis using D flip-flops:

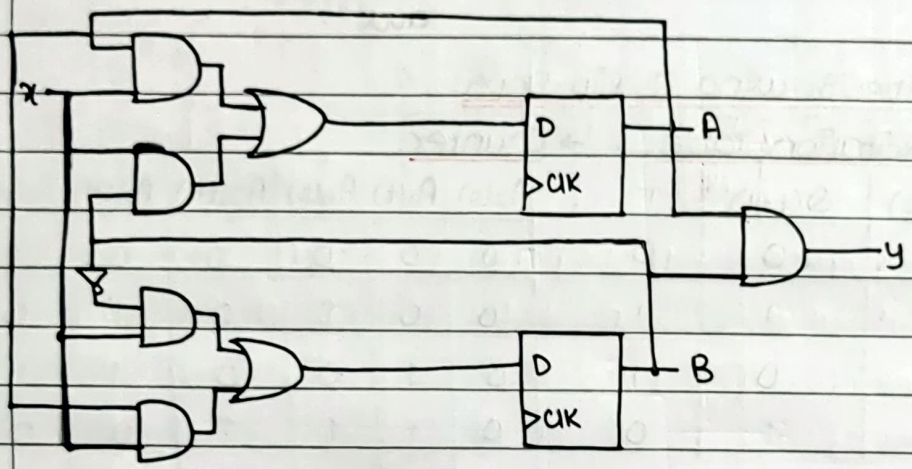
- Lets use 3 1's as example.

⇒ 2 flip-flops required as 00 01 10 00 = 4 states

- Now, reduce next state using K-Maps:



$D_A = Ax + Bx$ $D_B = Ax + B'x$ $y = AB$



◦] Synthesis using JK flip-flops:

- First we have to make an excitation table.

→

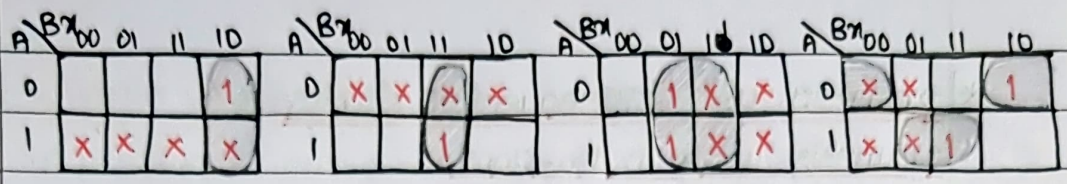
Q(t)	[redacted] Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

} Reverse Engineer from Characteristic Table of JK flip-flop

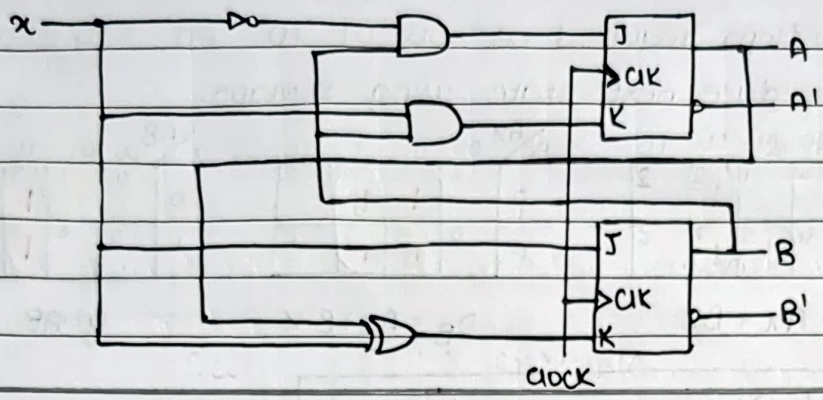
⇒

A(t)	B(t)	x	A(t+1)	B(t+1)	J_A	K_A	J_B	K_B
0	0	0	0	0	0	x	0	x
0	0	1	0	1	0	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	1	0	x	x	0
1	0	0	1	0	x	0	0	x
1	0	1	1	1	x	0	1	x
1	1	0	1	1	x	0	x	0
1	1	1	0	0	x	1	x	1

__/__/____



$J_A = Bx'$ $K_A = Bx$ $J_B = x$ $K_B = (A \oplus x)'$



* Synthesis using T flip-flops:

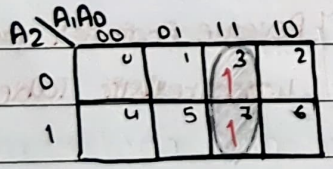
↳ Excitation table:

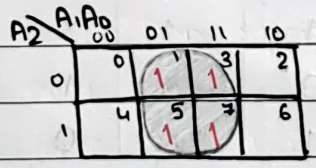
Q(t)	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

→ $TA_0 = 1$

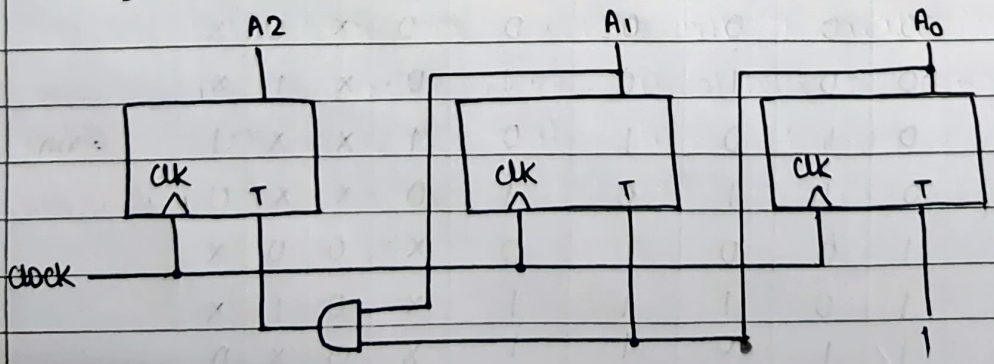
↳ Counter:

$A_2(t)$	$A_1(t)$	$A_0(t)$	$A_2(t+1)$	$A_1(t+1)$	$A_0(t+1)$	TA_2	TA_1	TA_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1





$TA_2 = A_1A_0$ $TA_1 = A_0$



__/__/__

*] Registers:

- A combination of several flip flops to store larger bits of memory i.e. 4 bits, 8 bits, 16 bits, etc.

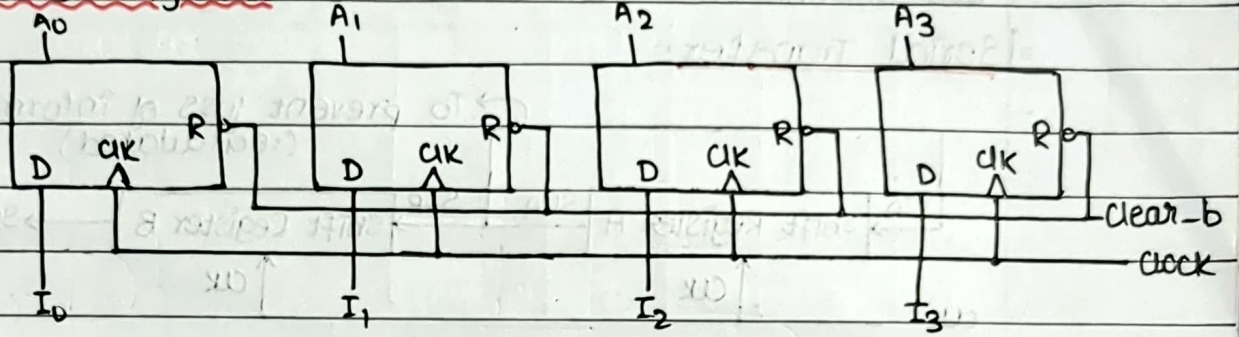
°] Parallel Registers:

- clock cycle causes all bits to change

°] Series Registers:

- clock cycle shifts bits to next flip flop

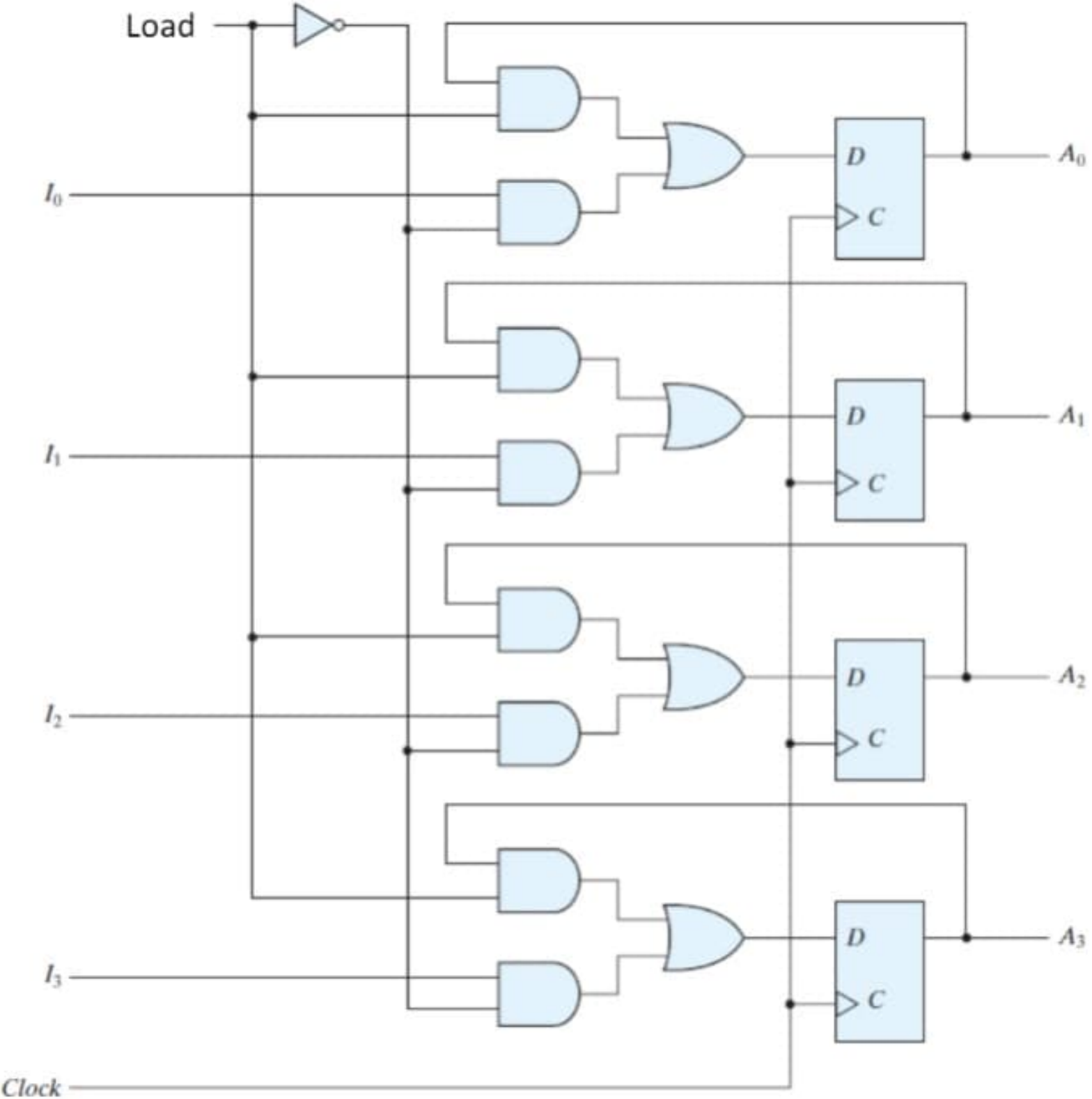
Ⓘ Parallel Register:



- The common clock input triggers all flip-flops on the +ve edge, the binary data transferred to register.
- When clear_b (connected to reset) goes to 0, flip-flops reset.

Ⓘ Registers with load input:

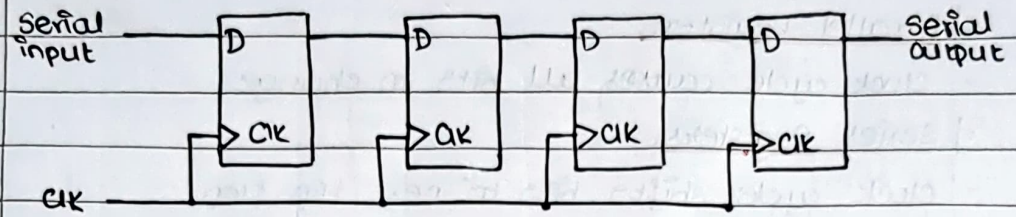
- We might not be interested in changing data every cycle.
- A seperate control signal must be used, transfer of new information is called loading/updating.
- We want clock pulse to reach all at same time, ~~clock~~ inserting gates in the clock path ill advised as that means logic performed with clock pulses
- Two do this a two channel mux is used. When load input is 0, data/or external points transferred to register @ next +ve edge. When load 1: output flip-flops connected to their respective inputs.
- The feedback from output to input necessary because a D-flip-flop does not have a "no change" condition.


Load
I_0
I_1
I_2
I_3
Clock
D
C
A_0
A_1
A_2
A_3

__/__/____

(III) Shift Register:

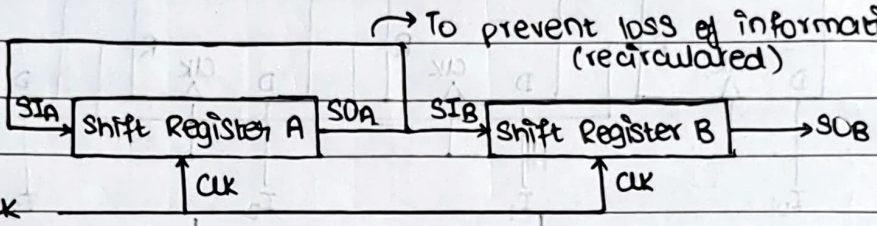
- All flip-flops recieve common pulses, which activate the shift of data from one stage to next.



- Two-channel MUX can be used to regulate here as well

◦] Serial Transfer:

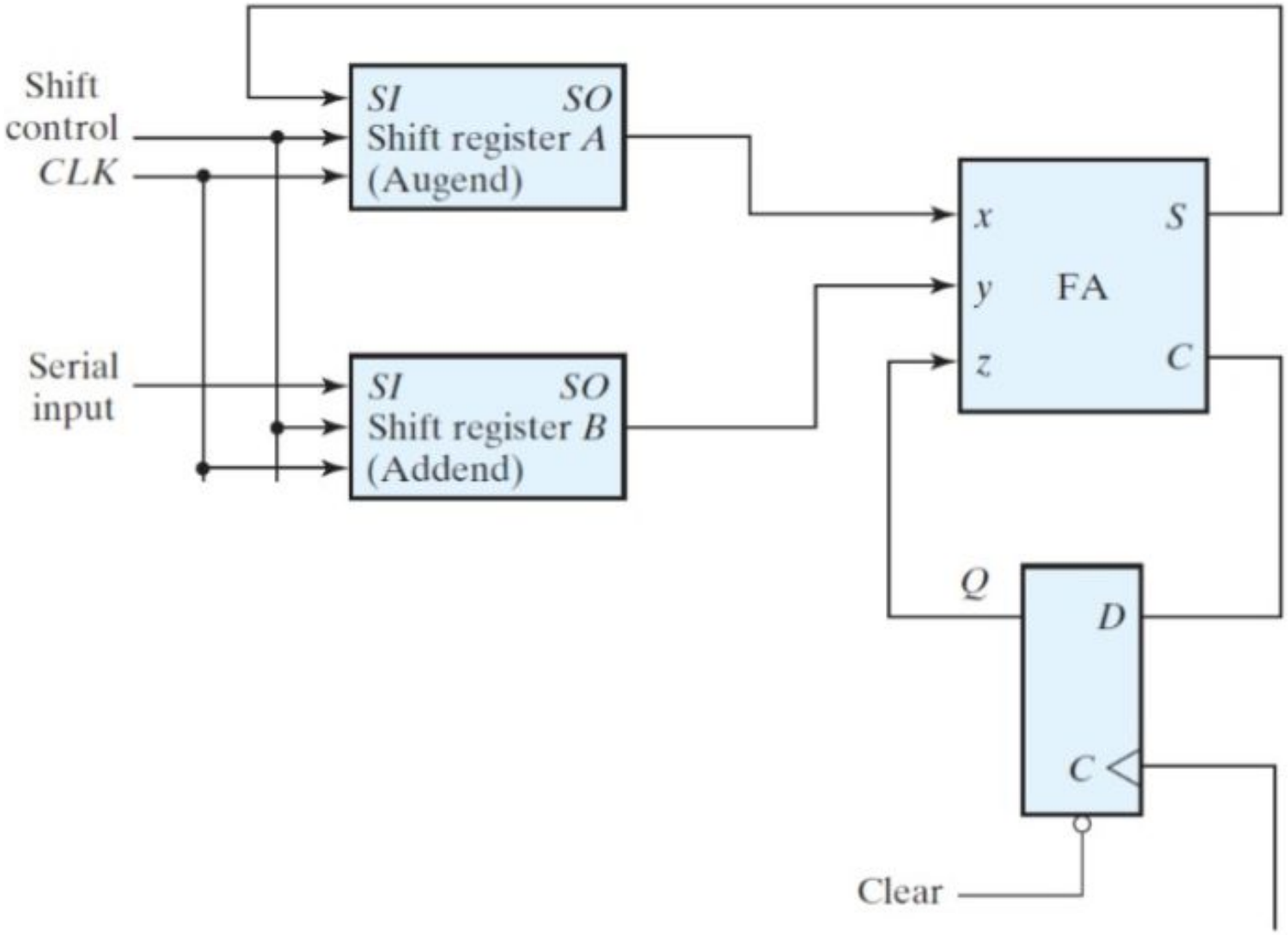
-



(#) Timing pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	1	1	0	1	1	0	0	1
After T_2	1	1	1	0	1	1	0	0
After T_3	0	1	1	1	0	1	1	0
After T_4	1	0	1	1	0	0	1	1

◦] Serial Addition:

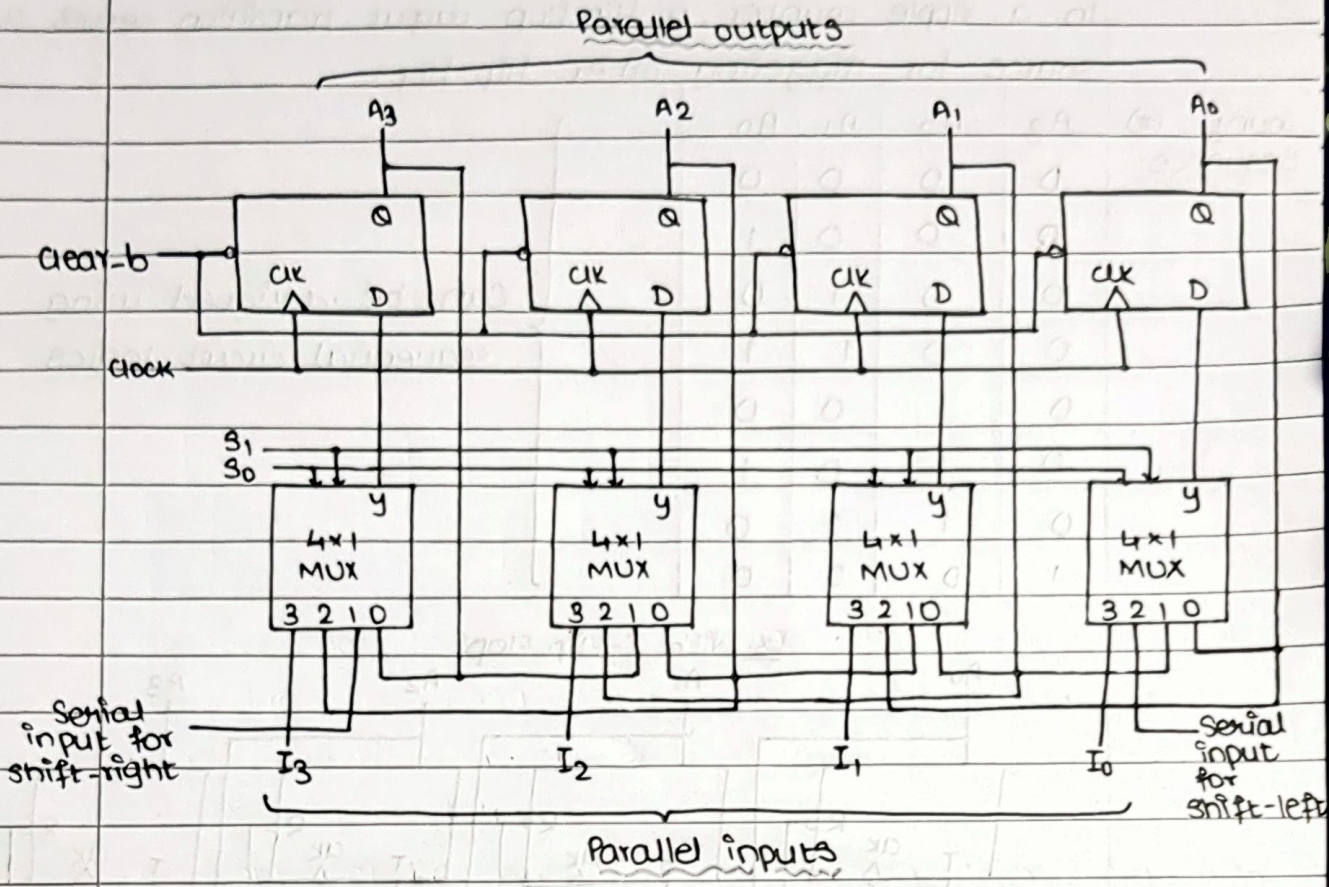
- Beginning with the least significant pair of bits, the circuit adds one pair at a time through a single FA
- Carry transferred to D flip-flop, output of which used as carry input for next pair.
- By shifting sum to A while bits of A shift out, it is possible to use one register for storing augend & sum
- The parallel adder uses registers with a parallel load. The no. of FAs in parallel adder = no. of bits in binary number whereas serial = 1 FA & 1 flip-flop
- Excluding the registers, parallel adder ⟶ combinat^n circuit serial adder ⟶ sequential circuit.


Shift
control
CLK
SI
SO
Shift register A
(Augend)
Serial
input
SI
SO
Shift register B
(Addend)
x
y
z
FA
S
C
Q
D
C
Clear

__/__/____

6] Universal Shift Register:

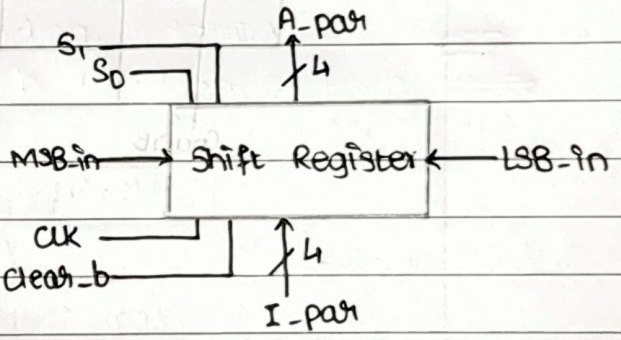
- Can shift right as well as shift left.



↳ Function Table:

S_1	S_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

↳ Block Diagram:



→ When $S_1S_0 = 00$, the present value of the register is applied to the D inputs of the flip-flops. This condn forms a path from the output of each flip-flop, so that the output recirculates to the input.

→ When $S_1S_0 = 11$, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock edge.