Binary Counting

Decimal	Binary	Decimal	Binary
0	0	8	1000
1	1	9	1001
2	10	10	1010
3	11	11	1011
4	100	12	1100
5	101	13	1101
6	110	14	1110
7	111	15	1111

Binary Addition:

- $1+1 = (2)_{10} = (10)_2$ => write 0, carry over 1 to the next significant bit.

- machines use binary:

  < 2.5 V = off 0

  > 2.5 V = on 1

Converting Between Bases

1) $(7392)_{10} \rightarrow (?)_2$

7392/2	3696	0	$a_0 = 0$
3696/2	1848	0	$a_1 = 0$
			$a_2 = 0$
			$a_3 = 0$
			$a_4 = 0$
			$a_5 = 0$
			$a_6 = 1$
			$a_7 = 1$
			$a_8 = 1$
			$a_9 = 0$
			$a_{10} = 0$
3/2	1	1	$a_{11} = 1$

$(7392)_{10} \rightarrow (?)_{16}$

7392/16	462	0	$a_0 = 0$
462/16	28	14	$a_1 = E$
28/16	1	12	$a_2 = C$
1/16	0	1	$a_3 = 1$

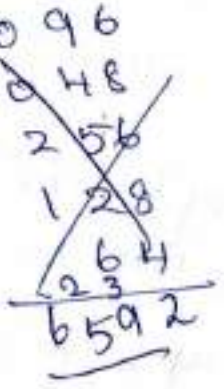
$(1CE0)_{16}$

$= (1110011100000)_2$

MSB ... LSB

$(1100111000000)_2$

4096 + 2048 + 256 + 128 + 64



e.g.2) $(41)_{10} \rightarrow (?)_2$.

	i	r	
41/2 =	20	1	$a_0 = 1$
20/2 =	10	0	$a_1 = 0$
10/2 =	5	0	$a_2 = 0$
5/2 =	2	1	$a_3 = 1$
2/2 =	1	0	$a_4 = 0$
1/2 =	0	1	$a_5 = 1$

$(101001)_2$.

$32+8+1$ ✓.

16 / 48 / 128 — 5 16 / 9 / 14H

e.g.3) ~~135~~ $(110111011)_2$

$1+2+8+16+32+128+256$

$10 \qquad 160$

256
187
443

$8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0\ -1\ -2\ -3\ -4\ -5\ -6 \ldots -9$
$2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ 2^{-5}\ 2^{-6} \ldots 2^{-9}$
$1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1 . 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1$

- decimal fractions to binary:

$(0.38)_{10} \rightarrow (?)_2$

	i	f	
0.38 × 2	0	0.76	$a_{-1} = 0$
0.76 × 2	1	0.52	$a_{-2} = 1$ etc.

e.g.4) $(0.124)_{10} \rightarrow$ binary

	i	f	
0.124 × 2 =	0	0.248	$a_{-1} = 0$.
0.246 × 2			

* binary to octal conversion → collect 3 bits at a time, write their octal equivalent.
* binary to hexadecimal → collect 4 bits at a time, write their hexadecimal equivalent.
* octal to binary → write each octal bit as its equivalent 3 binary bits.

e.g.) $(127)_8 \rightarrow (?)_2$. $(1010111)_2$.

~~2 | 11~~

e.g) Binary Subtraction

e.g.) 101 ⇝ Minuend
100111 ⇝ subtrahend.
000110

e.g.)

```
  1011
   101
 x
    11
```

HWQ) 1011
× 101

e.g.)

```
  111
- 1001
------
```

e.g.) $(10100111)_2 = (?)_{10}$

$1+2+4+32+128 = (167)_{10}$.

e.g.) $(145)_{10} = (?)_2$

145|2	72	1	$a_0 = 1$
72|2	36	0	$a_1 = 0$
36|2	18	0	
18|2	9	0	
9|2	4	1	
4|2	2	0	
2|2	1	0	
1|2	0	1	

$(10010001)_2$

Octal Counting System

- 1 octal bit = 3 binary bits.

- Hexadecimal System
  ↳ 1 hexadecimal bit = 4 binary bits.

e.g.) $(673)_8 \longrightarrow (??)_2$.

111
110

$(110111011)_2$ //.

e.g.) $(0011000000110)_2 \longrightarrow (?)_8$.

$(1406)_8$

8+4+2+0.

A = 10 B = 11 C = 12
D = 13 E = 14 F = 16.

e.g.) (CAFE).

1010

$(1100101011111110)_2$.

e.g.)

- machines communicate in binary. hard for humans to use binary all the time (cumbersome).

Signed Numbers

- a bit is left out to designate a sign to a number.
  - 0 = +ve sign
  - 1 = -ve sign.

  } e.g. check signed bit before addeng.

- ~~Three~~ methods to do this:

signed bit:

Complement
- radix r's → $r^n - N$
- diminished radix (r-1)'s → $r^n - N - 1$

Formulas

*n = no. of dig. in the number given (N)

Base	Radix	Diminished Radix
r	r	r-1
10	10's	9's
2	2's	1's

eg) $N = (546700)_{10}$

find 10's complement.

r=10, n=6

10's C: $(10^6 - 546700)$ → (453,300)

9's C: $(10^6 - 546701)$ → (453,299)

eg:) $(012398)_{10}$.

r=10, n=6

*count the preceding 0 as a digit.

10's C: $10^6 - 12398 = 987602$

9's C: $10^6 - 12399 = 987601$

e.g.) $N = (1011000)_2$

r=2, n=7

1's C: 0100111

2's C: $2^7 - 1011000 = 1000000 + 1011000$

$= 0101000$

convert to binary and subtract

preceding 0's are significant!!!

• 2^x:

$2^x - 1$:

1 0 0 0 0 0 0 0

3 2 1 0

↑ x

1 1 1 1 1 1 1

↑ (x-1).

eg) N = 0101101

n = 7 r = 2

2's C: 2^7 - 0101101.

1 1 1 1 1 1 1 2

1 0 0 0 0 0 0 0

- 0 1 0 1 1 0 1

$(1010011)_2$

1's C: $(1010010)_2$

* For binary numbers: 1's complement is obtained by replacing 0 with 1 and 1 with 0's

* Diminished radix $^{(r-1)'s}$ is obtained by subtracting each bit from (r-1).

eg.) N = 246,700.

r = 10 (n = 6.

10's C: 10^6 - 246,700 = 753,300.

9's C: 753,299 eg.)

e.g.) N = 1011000

1's C: 0010011

2's C: 0010100

- If we want to subtract a number, we usually find the (r-1)'s complement of

(complement of 0 is 0)

e.g.) 72532
- 03250
= 69282

it's 10's C →

  725320
+ 167450
= 732070   no carry ∴ take 10's C of ans
  ~~267930~~

$r's\ C = r^n - N$

can change sub to add.

(ALWAYS append a 0)

e.g.) 110110
- 0101011
= 0111110

2's complement →

  1001010
+ 1010101
= 1011111

→ H.S.B
↳ 0 ⇒ +ve

~~Signed bit~~

① for 2's complement ignore this digit. (carry from addition)

② if no carry, take complement of answer. (r's) and put (−) sign

for designing computers it is simpler to do all the operations through addition.

↳ ∴ subtraction = adding of negative.

- we need to find M − N

↳ convert to two's complement

$\Rightarrow N' = 2^n - N$

↳ add the two's complement.

$M + (2^n - N) = (M - N) + 2^n$

$= 2^n - (N - M)$.

(1) $12 $32 ① append a 0 always.
cannot leave a bit unassigned.

$$\begin{array}{r} 03250 \\ -\ 69282 \\ \hline 96749 \end{array}$$

② take its 10's C: 96750. and do addition instead

③

$$\begin{array}{r} 72532 \\ +\ 96750 \\ \hline 1\ 69282 \end{array}$$

1 carry over from addition ∴ ignore.

⇒ 69,28218 the ans //.

Q) X = 1010100
Y = 1000011

i) X − Y.

0101011
+ 1
0101100

find 1's C by 0↔1, 1↔0.
then +1 to find 2's C for binary.

2's complement of Y = 1000011 (corrected)

1010100
⇒ 0111101
+ 1
1 0010001

carry ∴ ignore ∴ X − Y = 0010001

ii) Y − X: 1000011
+ 0101100
1101111

no carry. take 2's C and put (−)

(−) 0010001 // = Y − X

• Using 1's complement to do subtraction:

Q) X = 1010100
Y = 0111100

i) X − Y using 1's C:

Y: 1's C = 1000011

1010100
+ 1000011
1 0010111
+ 1
0011000

"as in 1's C"

carry over ⇒ in 1's C method, add the carry to the final ans.

= 0011000 //.

ii) Y − X using 1's C:

0111100
+ 0101011
1100111

no carry ∴ take 1's C of this and put (−) sign.

⇒ (−) 0011000

Q. add -6 add -13 in binary.

6 in binary: 110 → (-6) in binary 1110

13 in binary 1101 (-13) in binary 10011

Binary Coded Decimal (BCD)

- 170

0001 0111 0000 → each dig. is represented by 4 bits.

1 7 0

- 7.2 5 3 2 ⇒ 20 bits to represent in BCD

- $\overset{4}{4} \rightarrow 0100$

  $+\ 5 \rightarrow +0101$

  $9 \qquad 1001$ ✓

- $\overset{4}{8} \rightarrow 0100$

  $+\ 8 \rightarrow +1000$

  $12 \qquad 1100 \rightarrow$ 12 but not BCD.

  $\overset{1}{1}\overset{0}{0} \rightarrow$ convert BCD add (110) (6).

  $10010 \quad \therefore \quad 0001 \ 0010$

  $1 \quad 2$ in BCD.

$$\begin{array}{r} 8 \\ +\ 9 \\ \hline 17 \end{array} \rightarrow \begin{array}{r} 1000 \\ +1001 \\ \hline 10001 \\ \quad\ 110 \\ \hline 10111 \\ \hline \end{array}$$

→ 17 in binary + 110 to BCD.

0001 0111

17 in BCD//

Q) A = 1110, B = 0110 → numbers are in signed magnitude representation. ✗

i) B − A.

$$\begin{array}{r} 0001 \\ \quad\ 11 \\ \hline 0010 \end{array}$$

$$\begin{array}{r} 0111 \\ 1111 \\ \hline 1000 \end{array}$$

−(1000)

$$\begin{array}{r} 0110 \\ +\ 0010\ (2\text{'s C}) \\ \hline 1000 \end{array}$$

→ no carry, ∴ take 2's C of this and put a (−).

ii) 0110 (−1000)

$$\begin{array}{r} +\ 0001\ \ 1\text{'s C} \\ \hline 0111 \end{array}$$

→ no carry ∴ take 1's C and add a (−) sign.

Q) BCD of 9+9

9 → 1001 add 6 110

+ 9 → 1001

18 10010

1110

11000

∴ 18 = 0001 1000 //

Q) A = 1110 → Signed magnitude.
B = 0110

∴ A = −0110
B = 0110.

↳ take 2's C. of A.
↓
becomes −A = 1010.

B − A = 0110
+ 1010
1 0000

* complement of the complement is the number itself.

$B - A = B + A^C$

$A = 1110$, $A^C = 0001 + 1 = 0010$.

IMPORTANT

0110
+ 1010
1 0000

A = −0110.

0110 − (−0110).

let C = −0110.

$B - C = B + C^C$

$C^C = -1001 - 1 = -1010$

∴ 0110 − 1010

↓

of the form X − Y.

0110
+ 0110
1100

0101 − 0110

Gray Code

- only 1 bit changes value from going from one number to the next.
- reduces errors.

Binary to Gray code

B: 1 0 0 1

G: 1 1 0 1

$B = B_3\ B_2\ B_1\ B_0$

$G = G_3\ G_2\ G_1\ G_0$

keep MSB as it is.

$G_3 = B_3$.

$G_2 = B_3 \oplus B_2$

$G_1 = B_2 \oplus B_1$

$G_0 = B_1 \oplus B_0$.

Gray code to Binary

G: 1 1 0 1

B: 1 0 0 1

$G = G_3\ G_2\ G_1\ G_0$

$B = B_3\ B_2\ B_1\ B_0$

keep MSB as it is.

$B_3 = G_3$

$B_2 = B_3 \oplus G_2$.

$B_1 = B_2 \oplus G_1$

$B_0 = B_1 \oplus G_0$.

• 2 bit gray codes:

0	0
0	1
1	1
1	0

1) → possible bits are 0/1 write them like this
2) Keep a mirror and reflect
3) above mirror 2nd bit 0's
below mirror 2nd bit 1's

•

0:	0	0	0
1:	0	0	1
2:	0	1	1
3:	0	1	0
4:	1	1	0
5:	1	1	1
6:	1	0	1
7:	1	0	0
8:	1	1	0	0
9:	1	1	0	1
10:	1	1	1	1
11:	1	1	1	0
12:	1	0	1	0
13:	1	0	1	1
14:	1	0	0	1
15:	1	0	0	0

for 3 bit:

like so.

first 16 decimals (0–15) in gray code.

Other Decimal Codes

- for 4 bit numbers

→ $\underset{2^3}{\_}\ \underset{2^2}{\_}\ \underset{2^1}{\_}\ \underset{2^0}{\_}$

8 4 2 1 → weights ∴ 8421 code

→ alternatively:

2 4 2 1 code.) 8 4 −2 −1 code

* kind of encrypting the signal by creating all these different codes.

Excess 3 code

→ whatever number we get, add 3 to it.

* in some of these codes, a ~~unique~~ number may not have a unique way of representing it,

ASCII code

- American Standard Code for Information Interface.

- $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

- How do we know if the signal was received as intended or not?

↳ Error Detecting Code:

- count.
- Transmit 'A' in ASCII: ~~1000~~ 0 1000001 ← parity.

- Input:

Is class open	Is student present	Lecture will happen
0	0	0
0	1	0
1	0	0
1	1	1

→ just multiply it works

- And representation of two events $x \cdot y \equiv \wedge$
- NOT of x is $\bar{x}$ (negation $\neg$).
- OR representation of two events $(x+y) \equiv \vee$.

↳ just add, it work

- we can

Q) x: 0 1 1 0 0

y: 0 0 1 1 0

$x \cdot y$: 0 0 1 0 0

$x+y$: 0 1 1 1 0

$\bar{x}$: 1 0 0 1 1

Chapter 2: Boolean Algebra and Logic Gates

Postulates of Boolean Algebra

① Closure: A set S is a closed set wrt a binary operator if for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S.

e.g. Set of $\mathbb{N}$ is closed wrt '+' operation.
if any a+b then $\exists\, c \in \mathbb{N}$ s.t. $c = a+b$
ie c lies in $\mathbb{N}$ only.

② Associative law.
$$x * (y * z) = (x * y) * z$$

* arithmetic:
$A * (B+C)$
* is distributive over + not other way.

③ Commutative law:
$$x * y = y * x.$$

* in boolean algebra, any two operators can distribute over each other

④ Identity element:
A set S has an identity element wrt a binary operation * on S if $\exists\, e \in S$ s.t.
$\forall x \in S$, $x * e = e * x = x$.

⑤ Inverse:
A set S having an identity element, has an inverse whenever. $\forall x \in S, \exists\, y \in S$ s.t
$$x * y = e.$$

A+(B·C) ☹ if confused, convert to A∨(B∧C) Propositional

⑥ Distributive law:
* is distributive over · wherever;
$$x * (y \cdot z) = (x * y) \cdot (x * z).$$

• What is a postulate?

↳ Statements you make which you experimeed and find out.

e.g. atm pressure acts over us.

(+, ·) only; −, ÷ can be phrased as (+, ·)

Postulates of Boolean Algebra

1) Structure is closed.

2) a) 0 is identity element wrt ⊕, +

b) 1 is identity element wrt ·.

3) commutative.

4) both operators (+, ·) are distributive over each other.

5) $\forall x \in B, \exists x' \in B$ s.t. $x + x' = 1$ (inverse)

6) $\exists (x, y) \in B$ s.t. $x \neq y$.

↳ (there are only 2 elements in the boolean set $\{1, 0\}$).

HW) x y z $y+z$ $x \cdot (y+z)$ xy xz $(xy + xz)$

Duality Principle

- if we have a function and we want its dual function:

$x \rightarrow x'$
$'\cdot' \rightarrow '+'$
$'+' \rightarrow '\cdot'$

A dual expression and the og expression return the same truth value/logically equivalent.

e.g.) $x + 0 = x$

$(x+0=x)' \Rightarrow$ dual expression.

$x' \cdot 1 = x'$

e.g.) $x + x' = 1$

$(x+x'=1)' \equiv x' \cdot x = 0.$

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

$(x+y'+z) \cdot (x+y+z') = T.$

- used in proving, simplifying etc. before circuiting.
- how do we design circuits for boolean expressions.

Theorems of Boolean Algebra

- Theorem 1) $x+x=x$

Proof:

$x = x \cdot 1$

LHS: $x \cdot 1 + x \cdot 1 = (x+x) \cdot 1$

use $1 = x + x'$

$(x+x) \cdot (x+x')$

$x + x \cdot x' = x + 0 = RHS.$

Theorem 2) $x+1=1$

~~$x' \cdot 0 = 0$~~

~~$(x' \cdot 0 = 0) \Rightarrow x + 1 = 1$ //.~~

$(x+1) = (x+1) \cdot 1$

$(x+1) \cdot (x+x')$

$x + x' \cdot 1$

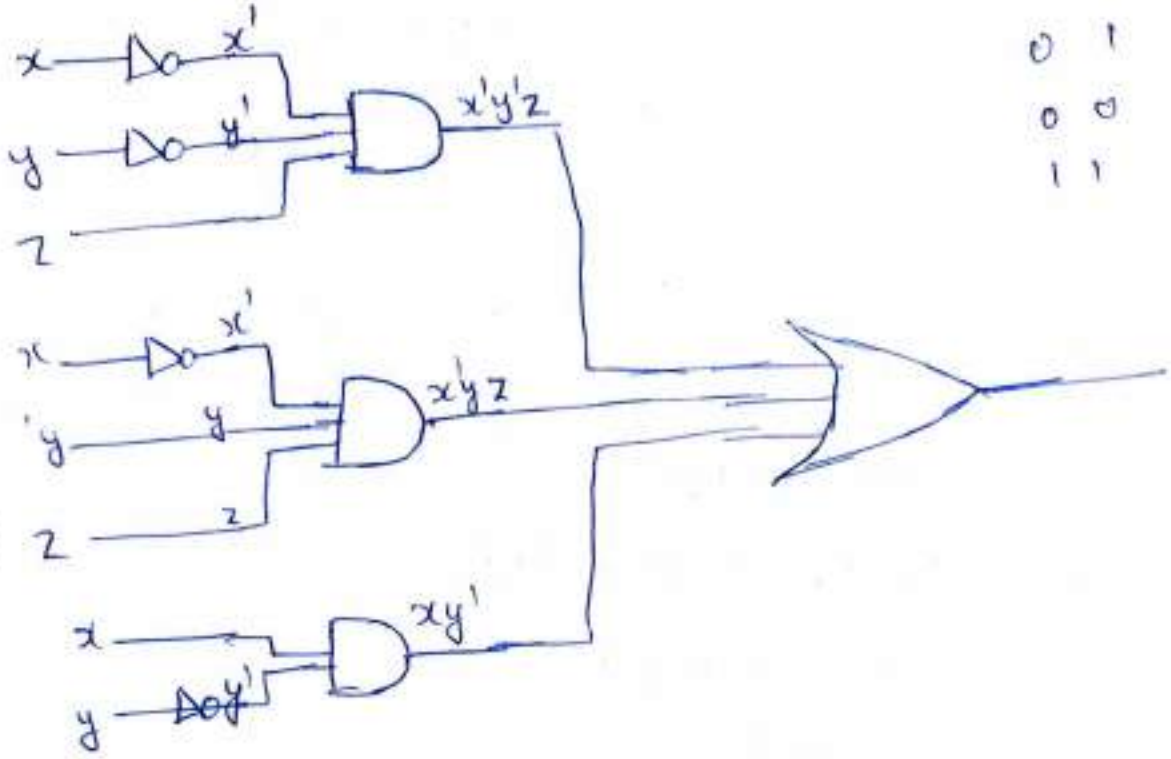
$x + x' = 1$

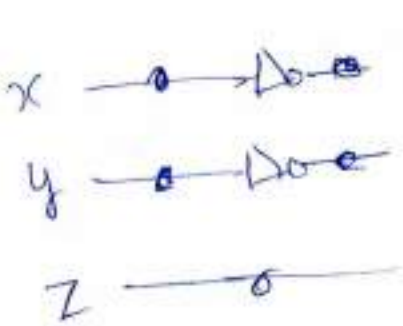
Functions

- $F = x'y'z + x'yz + xy'$
  Use logic gates, x y z inputs.

XOR

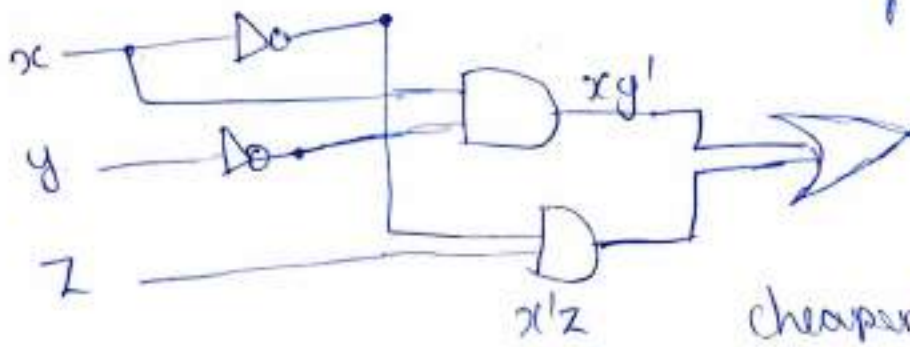
x	y	x ⊕ y
1	0	1
0	1	1
0	0	0
1	1	0.



-  → take nodes and use, do not repeat x, y, z inputs thrice.

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

  $F = x'z + xy'$ (simplified ∵ $y + y' = 1$)



→ circuits of reduced expressions are the best.

They are cheaper to make and consume lesser power.

Ex. 2.1) ii) $x + x'y = F$

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

$F' = x' \cdot x + x'y'$

$F' = x'y'$

$F = x+y.$

$x + x'y$

$= x(1) + x'y$

$x(y+y') + x'y$

$xy + xy' + x'y.$

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

iii) $xy + x'z + yz = F \rightarrow F' = (x'+y') \cdot (x+z') \cdot (z'+y')$ ✗

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

$F = xy + x'z + yz \cdot (x+x')$

$F = xy + x'z + xyz + x'yz$

$F = xy(z+1) + x'z(y+1)$

$F = xy + x'z$

→ can either algebraically simplify or use truth tables

x	y	z	F		
0	0	0	0	$x'y'z'$	M_0
0	0	1	1	$x'y'z$	M_1
0	1	0	0	$x'yz'$	M_2
0	1	1	1	$x'yz$	M_3
1	0	0	1	$xy'z'$	M_4
1	0	1	1	$xy'z$	M_5
1	1	0	0	xyz'	M_6
1	1	1	0	xyz	M_7

follows a sequence while writing (add 1 each time)

$F = x'y'z + x'yz + xy'$

↓

take those min values which output 1 and OR them together

each term has literals:

$x'y'z = 3$ lit.

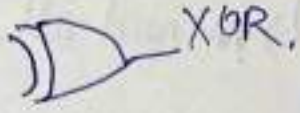
$xy' = 2$ lit.

→ called min terms (product) small m.

→ Maxterm M (dual sum of minterms).

- can we implement logic gates for binary addition?

Half-Adder:



x	y	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sum is the XOR gate.
Carry is the AND gate.

- x y z Sum carry.

- $F = x'y'z + xy'z' + xyz$. → standard product form → 9 literals.
  b) F is defined as a function of literals.

$F = m_1 + m_4 + m_7$ ($\because x'y'z = m_1$)

$\therefore F = \Sigma(1,4,7)$

- $F_3 = x'y'z' + x'yz' + x'yz + xyz'$

$= m_0 + m_2 + m_3 + m_6$

$\therefore F_3 = \Sigma(0,2,3,6)$ //.

- $F = x + y'z$ → how do we express F in min-terms.

Step 1 ; truth table!

Step 2 ; sum those min values which return 1 for F.

- How do we do this without truth tables?

  ↳ we need all terms to have all inputs in some form.

$F = x + y'z$

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

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

$F = xyz + xyz' + \underline{xy'z} + xy'z' + \underline{xy'z} + x'y'z$.

↳ look for repitition and simplify.

$F = \cancel{xy(z+z')}$

$F = xyz + xyz' + x(y'z + xy'z' + x'y'z$

Expressing Functions In Maxterms

- minterm → combo of products of literals
- maxterm → combo of sums of literals
- M_i = dual of m_i

$F = x'yz + xy'z + xyz' + xyz$

$F_2 = F' = (x+y'+z') \cdot ($

F' → pick all minterms that are 0

F → product of Maxterms giving F as 0

→ $F' = m_0 + M_1 + m_2 + M_4$ (taking 0's)

$(F')' = F = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi(0, 1, 2, 4)$.

- a function expressed in maxterms / minterms is called a canonical form.

~~(Pitt) F = 0~~

Q) $F = xy + x'z$ express in Maxterms.

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

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

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

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

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

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

- $F = M_0 M_2 M_5 M_4 = m_1 m_3 m_6 m_7$.
- How do we design a circuit for this?
- Min/Max terms only have 2 levels of gates (reduces delay time) → less delay but large no. of inputs

- Two input variables can be manipulated to form 16 types of functions.

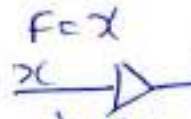
Table 2.7: F_1 = and, $F_2 = x \cdot y'$ (inhibition), $F_3 = x$, $F_4 = x'y$, $F_5 = y$

F_6 = XOR, F_7 = OR, F_8 = NOR, F_9 = XNOR, $F_{10} = y'$, $F_{11} = y' + x$ (implication function $x \leftarrow y$)

$F_{12} =$, $F_{13} = x' + y$ $x \rightarrow y$, F_{14} = NAND, $F_{15} = 1$, $x \uparrow y$

* $XOR(x,y) = x'y + xy'$ $(x \oplus y)$
* $NOR(x,y) = x \downarrow y$.
* XNOR = equivalence function $= xy + x'y'$

$F = x$



buffer gate

* strengthens signal, prevents decay

* NOR is NOT associative, * for 3 input gates.

$x \downarrow (y \downarrow z) \neq (x \downarrow y) \downarrow z$

* NAND Gate (3 inputs) is not associative!

~~x y z~~ $x \uparrow (y \uparrow z) = (x \uparrow y) \uparrow z$? Check for HW.

- x+y +ve logic 

x'+y' -ve (inverted logic) 

- integration (small scale) → putting 2-3 gates together on a bread board.
- VLSI → chips, millions of transistors, gates in a small area
- Logic families –

TTL → transistor to transistor logic.

MOS → Metal Oxide Semiconductor, logic.

CMOS → complementary MOS.

Q) $(1010110.01011)_2 \times (256)_{10}$.

28.

$1010110.01011 \{000 = ans$

$+ \quad 100000000.00000$

Q) A ~~bag~~ vault has 3 locks, each as 1 key. Eeach key is owned by diff. people. can only insert 2 at a time, and need 2 for it to open.

Chapter 3 : Gate-Level Minimization

- Gate-level minimization is about finding the optimal gate-level implementation for boolean functions.
- complexity of logic gates & complexity of algebraic expr.
- algebraic simplification can be done, however, it is not a set / defined procedure as this involves awkward manipulative steps.

Karnaugh Map Methods (3.2)

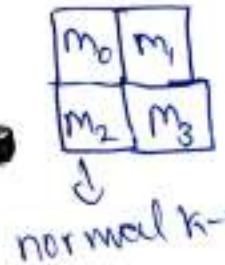
* arriving at the minimal expression depends on the ~~K-Map~~ grouping of 1's.

- pictorial form of truth table
- ~~Three~~ Two Variable K-Maps

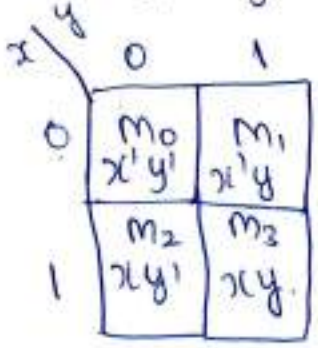
x	y
0	0
0	1
1	0
1	1

→ truth table

- ∵ a two variable function has four min-terms, the K-map has four squares, one for each ~~be~~ minterm, arranged in a binary sequence.



normal K-map

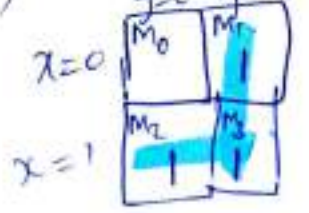


→ redrawn to give a clearer idea.

- ~~e.g.~~ We can mark the squares who's minterms belong to a given function ~~or~~ (+shade their area) with a 1.

e.g. $F = xy$ →

$F = x + y$

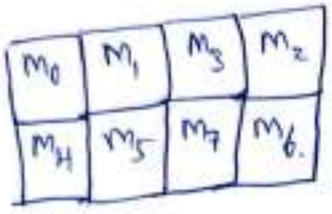


* We MUST mark those minterms at which the function is asserted with a 1.

Three Variable K-Map

- eight minterms for 3 variables
  ∴ K-Map has 8 squares.

* Minterms are not arranged in a binary sequence, but in a sequence similar to grey code.



→

x \ yz	00	01	11	10
0	m_0 $x'y'z'$	m_1 $x'y'z$	m_3 $x'yz$	m_2 $x'yz'$
1	m_4 $xy'z'$	m_5 $xy'z$	m_7 xyz	m_6 xyz'

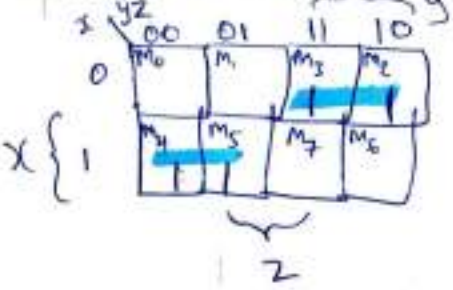
y = 1 (11, 10); x = 1 (1); z = 1 (01, 11)

- there are 4 squares in which each var. = 1, and 4 where each var. = 0.

- any two adjacent squares (↔diagonally) differ only by one variable – primed in one and unprimed in the other.
  ∴ any two adj. minterms ORed results in a removal of the dissimilar variable.

- e.g. Simplify the boolean $F = \Sigma(2,3,4,5)$.

$F(x,y,z) = m_2 + m_3 + m_4 + m_5$



$F = (x'yz + x'yz') \rightarrow m_3 + m_2$.

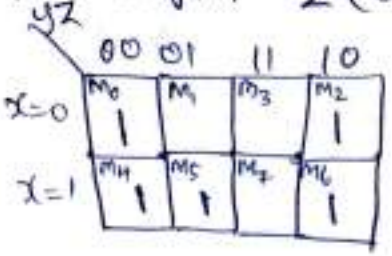
$+ (xy'z' + xy'z) \rightarrow m_4 + m_5$

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

$F = x'y + xy'$ //.

* two squares in a K-Map may be adjacent even if they do not touch each other, but their minterms differ by only one-variable.

* $F(x,y,z) = \Sigma(0,2,4,5,6)$



$F = (m_0 + m_4) + (m_2 + m_6) + (m_5)$

$F = \{x'y'z' + xy'z'\} + \{x'yz' + xyz'\} + m_5$

$F = y'z' + yz' + xy'z = z' + xy'z$

$\because xy'z' = xy'z' + xy'z'$

*

We can OR m_5 with an adjacent square that has already been used ones ($x = x + x$).

$\therefore F = z' + xy'z + xy'z' = z + xy'$ //

* To simplify F no. of squares to be combined is always 2^n.

- One square in the K-Map represents 3 literals terms.

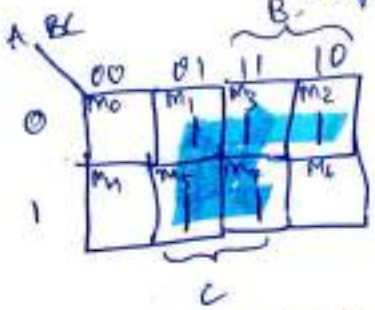
Two adjacent squares = represent 2 literal terms.

Four adjacent squares = represent 1 literal terms.

Eight adjacent squares = represent a function that always gives 1.

*e.g) $F = A'\bar{C} + A'B + AB'C + \underline{BC}$.

↳ 2 literal terms are represented by two adjacent squares in the K-Map.



$m_1 + m_3 = A'C$.

$m_3 + m_2 = A'B$.

$m_3 + m_7 = BC$.

$\therefore F(A,B,C) = \Sigma(1,2,3,5,7)$.

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

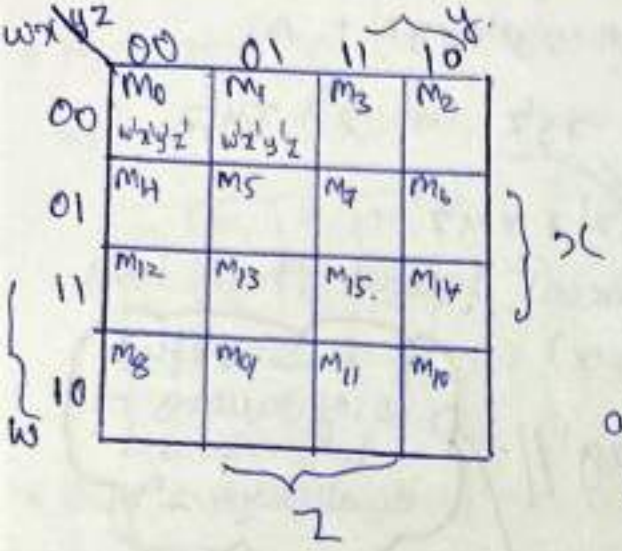
$= A'C + AC + A'BC' + A'BC$

$F = C + A'B$ //.

→ ORing m_2 with an already used adjacent m_3 square.

Four Variable K-Map

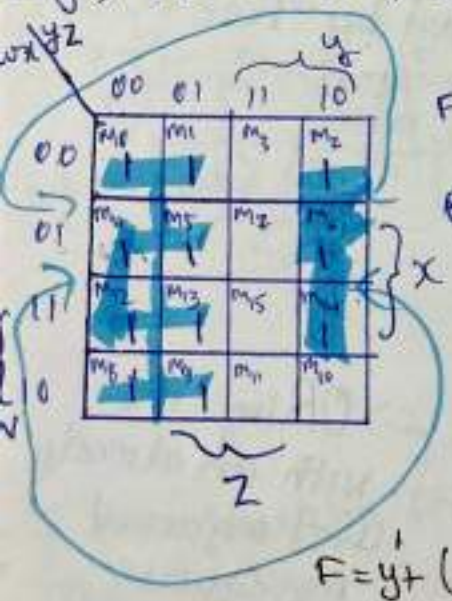
- map for boolean functions with four variables (w, x, y, z)
- rows, columns numbered in gray code sequence.



- if we index 2nd row 2nd column we get 0101 which when interpreted in binary is 5 so m_5 is placed there.
- here the top row squares are adjacent to the bottom row squares. Similar for left column and right column.

- One square represents a term with four literals
  Two squares represent the terms with three literals
  Four squares " with two literals
  Eight squares " with one literal
  Sixteen squares ← not a function that always gives 1.

E.g.) $F(w,x,y,z) = \Sigma(0,1,2,4,5,6,8,9,12,13,14)$



$F = w'x'y' + w'xy' + wxy' + wx'y' + w'x'yz' + w'xyz' + wxyz' + wxyz'$

$F = w'y' + wy' + w'yz' + wxyz'$

$F = y' + w'yz' + wxyz' + w'xyz'$

$F = y' + w'yz' + xyz'$

$F = y' + [(m_2 + m_0) + (m_3 + m_4)] + [(m_6 + m_{14}) + (m_{12} + m_{14})]$

$F = y' + (w'yz' + w'y'z') + (xyz' + xy'z')$

$F = y' + w'z' + xz'$

* squares on the right needed to be combined with 2 or 4 adj squares.... therefore cleverly re-use to more simplify F.

Prime Implicants

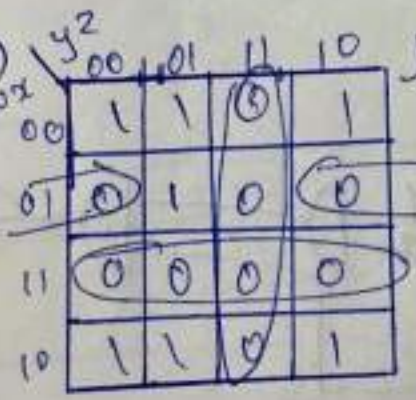
- Simplified version – trying not to repeat grouping (redundant) ~~ones~~ multiple times.
- cover all the 1's at least once.
- this is to minimize the no. of terms in the expression.

This form is called prime implicant.

-  → ideal grouping, no redundant / repeating square.
- Essential 1 → a square which has only 1 possibility for grouping.

Simplification of Product of Sums

- take complement of F (which is in product of sums).
  - ↳ becomes sum of products.
    - ↳ combine '0's squares to simplify F'
      - ↳ after simplification, take complement of F' again to get F.

e.g.) 

→ assuming ~~product of sums~~ sum of product map of a product of sums function.

∴ combine adj 0's.

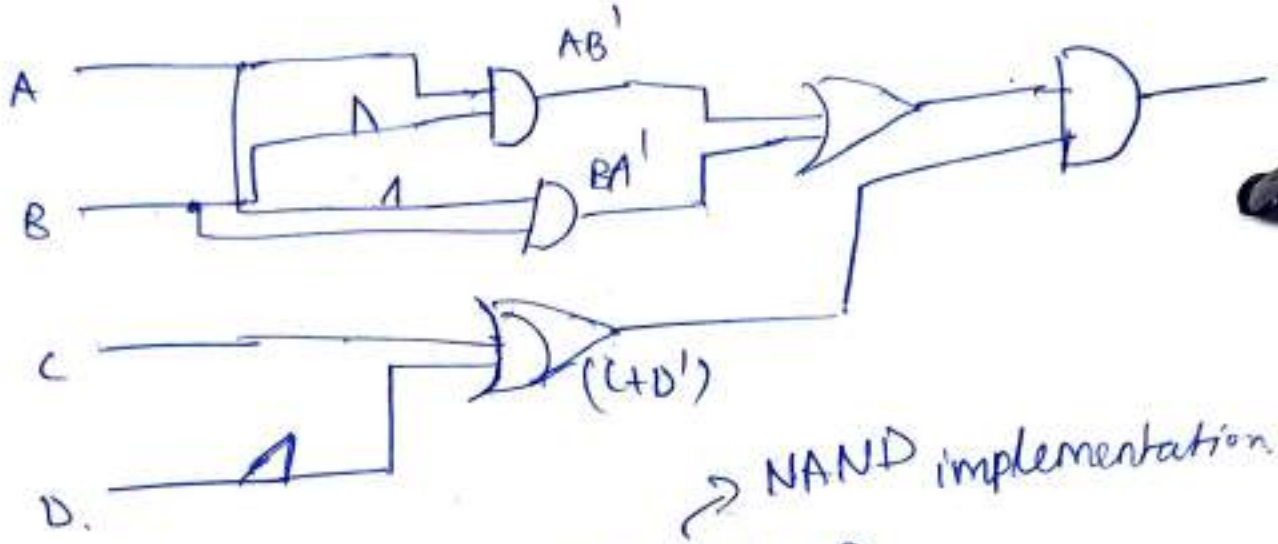
Q) Draw the NAND ckt for

$F = (AB' + BA')(C + D')$ .. !!

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

NAND		
A	B	
0	0	1
0	1	1
1	0	1
1	1	0

normal ckt;



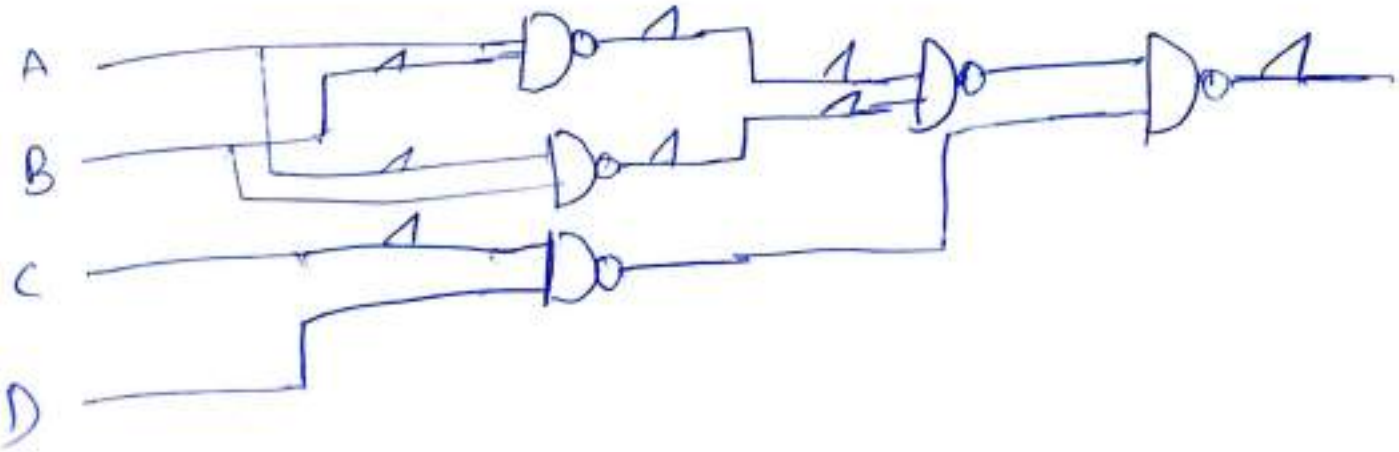
→ NAND implementation.

how to convert all to NAND gates?

AND → just  → invert the signal.

OR → $x+y = (x'y')'$.

$(C'\cdot D')'$



NOR Implementation

... select variables.

- AND = ? in NOR terms. xy. $(x+y)' = NOR$.
- OR = NOR inverted. → $(x'+y')' = xy$

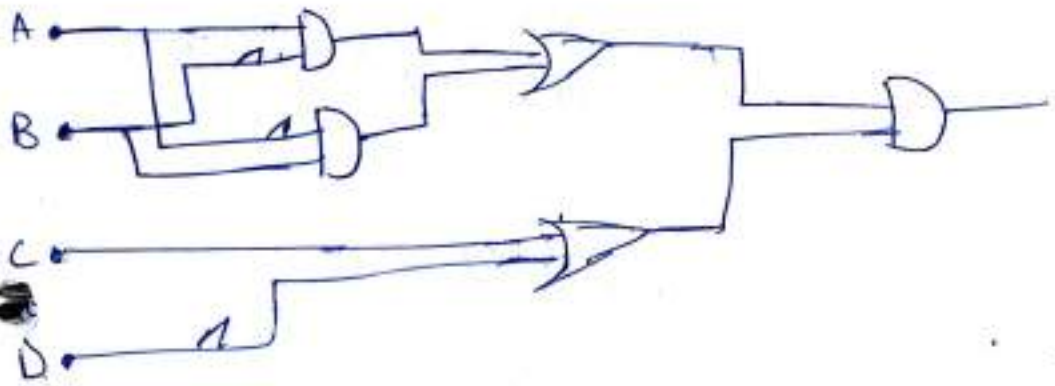
- AND = x, y → NOR gate → xy.

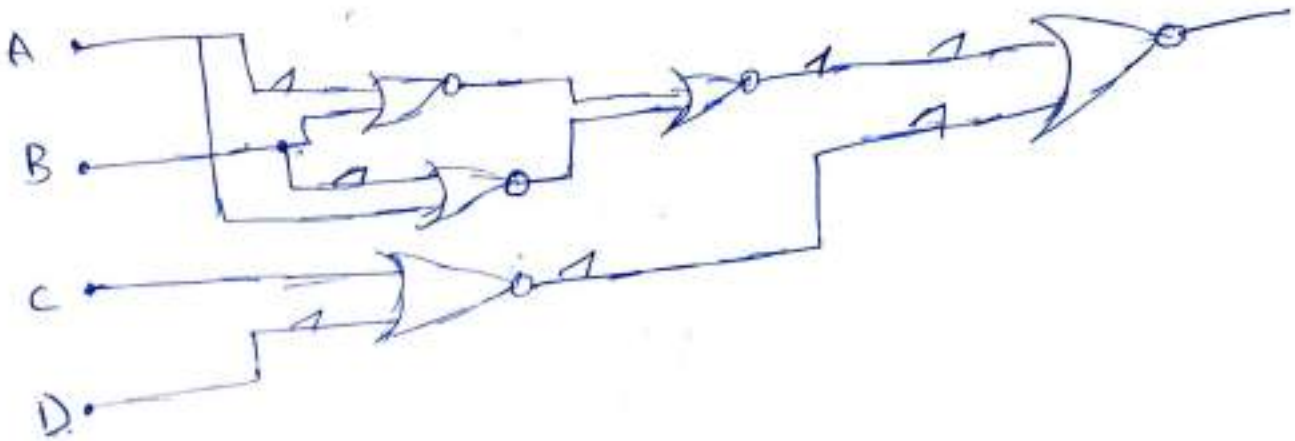
1) Take inversion of 2 inputs and NOR them.

- OR =

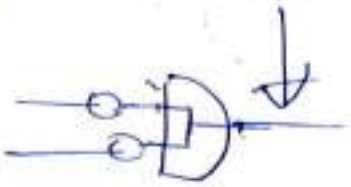
- $F = (AB' + BA')(C + D')$.

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

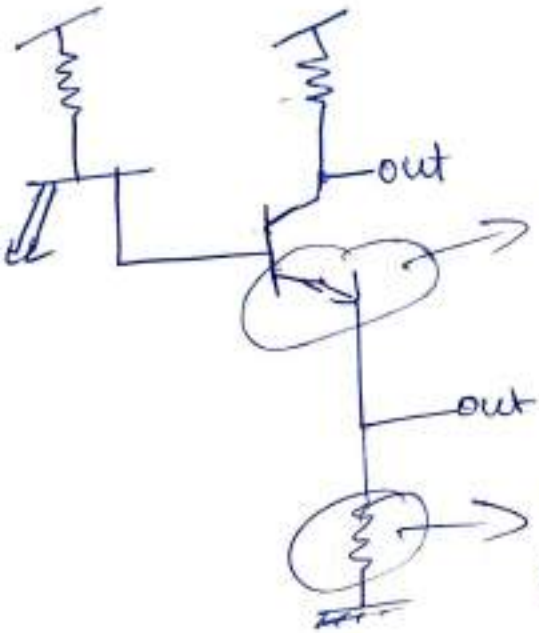




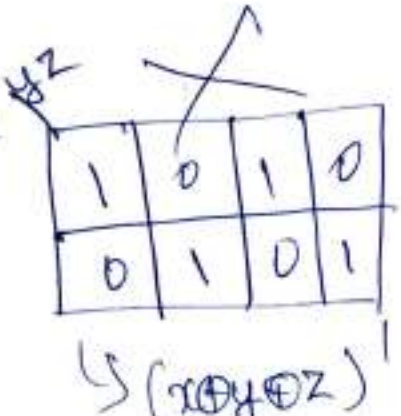
- Some implementations don't need transistors.
- draw wires inside gate – wired logic (not actually a gate, but same output),



- TTL:



-



↳ $(x \oplus y \oplus z)'$

0	1	0	1
1	0	1	0

↳ checkboard config. for $x \oplus y \oplus z$

$(x \oplus y) \oplus z$

x	y	z	$x \oplus y \oplus z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
1	1	1	1

- 3 input XOR is used in parity generation and parity checker.

3 input XOR is associative.

- $w \oplus x \oplus y \oplus z$

truth table, K-Map and function for above.

Chapter 4 : Combinational Logic

- combinational and sequential (ways in which we can combine circuits).

- combinational circuits:
  
  with given inputs, the output is determined at that same moment in time.

  - No memory element
  - F is a real time function of inputs.

- Sequential circuits have a storage / memory elements.

- What if there is only 1 output line but many inputs?
  ↳ have 1 transmitter which sends outputs one-by-one.
  ↳ Multiplexing the data

which logic gates can we use here for this task?

select input 'S' 0 / 1

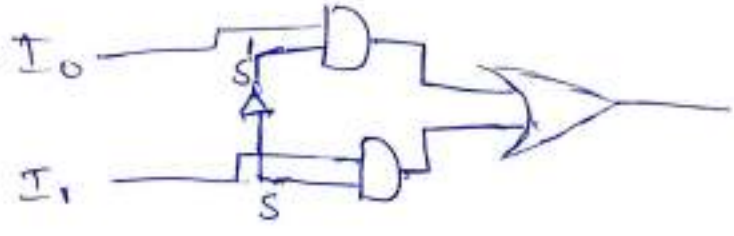
↳ whenever S is 0, we select I_0
1, " I_1,

I_1	I_2	S	out	
0	0	0	0	I_2
0	1	0	1	
1	0	1	0	
1	1	1	1	

I_1	I_2	S	out	S'
0	x	0	0	1
1	x	0	1	1
x	0	1	0	0
x	1	1	1	0

MUX
(multiplex)

≡



- 4 input lines, 1 output line;

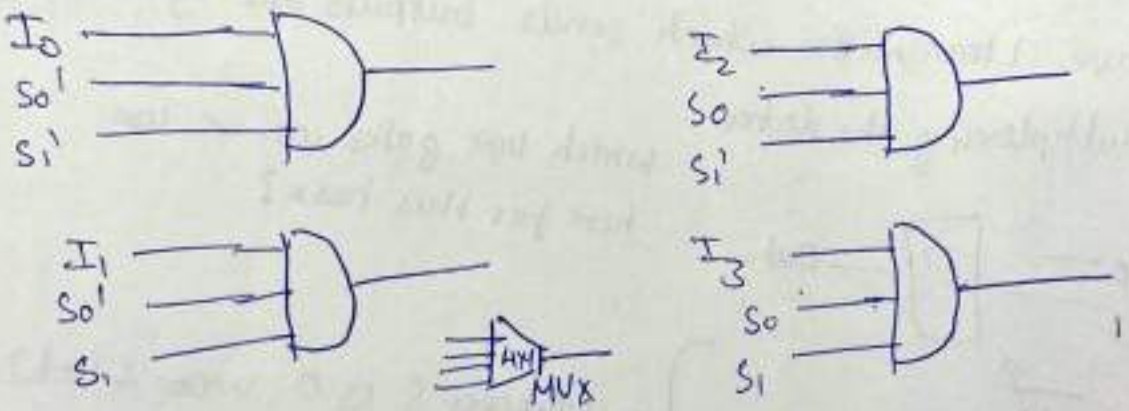
I_0	I_1	I_2	I_3	S_0	S_1	Out
				0	0	

- for 4 input lines, we need 2 select signals

S_0	S_1	
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3.

→ 4 selections for ~~eee~~ representing each of the 4 variables.

- take the select signal and AND it with the input signal.



only one of the four $Y = 1$, rest $3 = 0$ for one combination of (S_0, S_1).

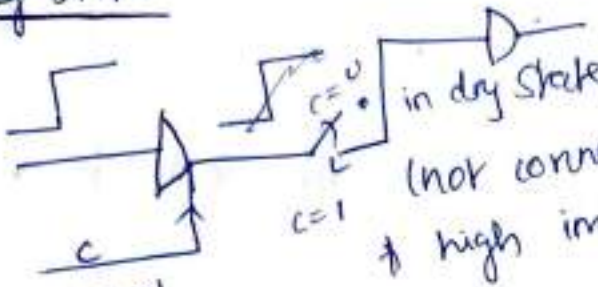
∴ final output by OR ing the AND outputs.

- $2^n = N$; N = total no. of input lines
  n = no. of select lines //

- enabling a gate is the same as selecting.
  ↳ disables the entire circuit - gives all 0's.

- we can use the inputs themselves as select variables.
- for n inputs, first n-1 variables are used as selects and the one remaining is compared with F.

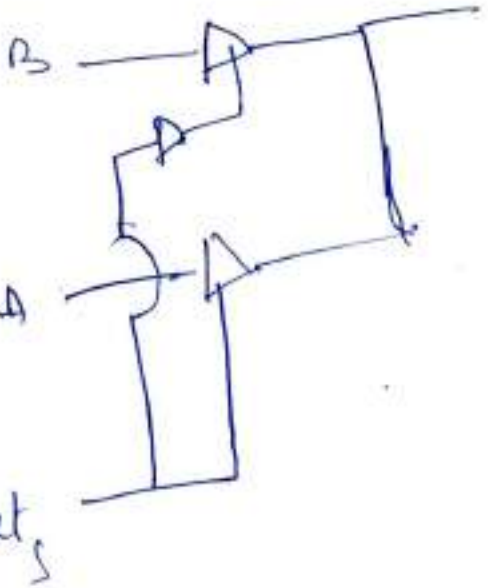
- Tri State



in dry state it is an open line (not connected anywhere), # high impedance state.

If C = 1, output follows input
If C=0, it is in high impedence state.

- decoder is the opposite of mux.



Analysis of Circuits

- write expression from gates
  - ↳ draw truth table
  - ↳ use K-Maps to simplify fully. //

Design for Circuits

- e.g. we want to do excess 3 coding.
- fill input columns, directly fill corresponding outputs.
  - ↳ K-Map
  - ↳ draw gates.

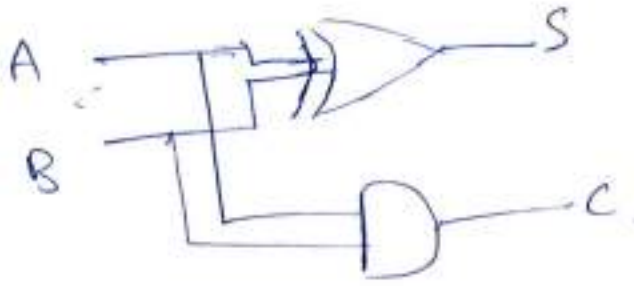
Half Adder

- $x \oplus y$ = sum bit
  $x \cdot y$ = carry bit.

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +0 & +1 & 0 & 1 \\ \hline 0 & 1 & 1 & 10 \end{array}$$

∴ sum bit = xor

carry bit = and.



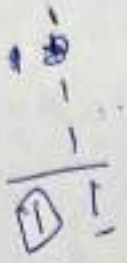
- we use two half adders to make a full adder.

Full Adder

1) $x \oplus y$

2) $(x \oplus y) \oplus z$

- using two half-adders; $C_1 + C_2$ for final carry



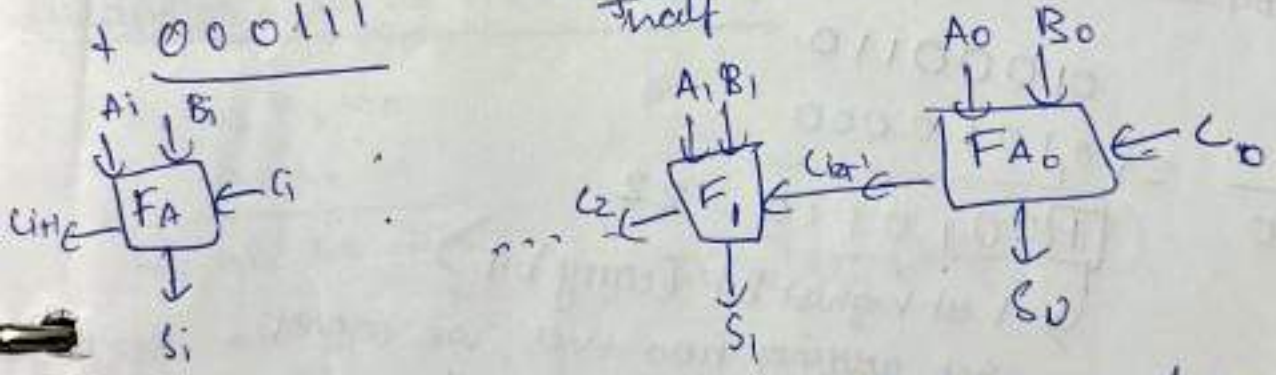
Binary Adder

- How would we add a ~~#~~ binary numbers?

$$\begin{array}{r} 101011 \\ +\ 000111 \end{array}$$

^{0} → take first $C_{in} = 0$.

use ~~full~~ half adder for each bit,



- the implementation of consecutive half adders has a lot of delay, so we use a carry lookahead generator:

- $P_i = A_i \oplus B_i$; $C_{i+1} = G_i +$
  $G_i = A_i \cdot B_i$; $S = P_i \oplus C_i$

$P_0 = A_0 \oplus B_0$

$G_0 = A_0 B_0$ } half adder.

$S_0 = P_0 \oplus C_1$

$C_1 = G_0 + P_0 C_0$

$C_1 = A_0 B_0 + (A_0 \oplus B_0) C_0$.

- 2 gate delay
- tradeoff is OR gates need more and more input lines.

~~Full Sub~~

Binary Subtractor:

- use 2's complements,

- ~~Four bit adder~~

- 70
  +80
  150

  01000110
  +0,1010000
  1 0010110 = 22.

  ↳ 1 at highest bit (carry bit) when adding two +ves, we cannot get a -ve number.

* Something about carries, overflows and warning bits //.

-70
-80
-150

  10111010
  +1 0,10000
  1 0 1101010

  ↳ highest bit is 0.

Decimal Adder

- BCD encoded adder.
  - ↳ (each decimal digit is represented by 4 bits)
    (whenever during adding, if we get ans > 9, add 0110)

- Till 9, same adder logic
  if > 9, how to convert back?

- $B_3\ B_2\ B_1\ B_0$ } 2 BCD numbers.
  $A_3\ A_2\ A_1\ A_0$
  1 } some K.
  4

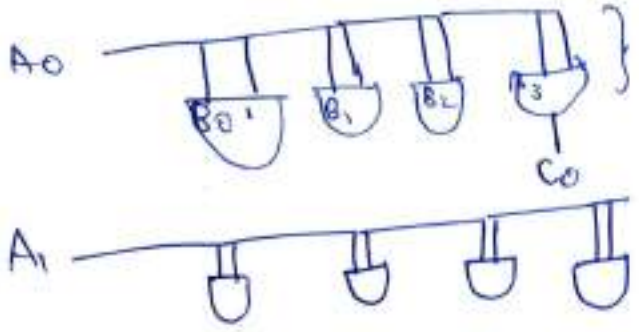
Multiplying a 3 bit by 4 bit number :

$B_3\ B_2\ B_1\ B_0$
$\times\quad A_2\ A_1\ A_0$

$A_0B_3\ \ A_0B_2\ \ A_0B_1\ \ A_0B_0$
$A_1B_3\ \ A_1B_2\ \ A_1B_1\ \ A_1B_0\ \ \times$
$A_2B_3\ \ A_2B_2\ \ A_2B_1\ \ A_2B_0\ \ \times\ \ \times$

$(A_0B_1 + A_1B_0)\ A_0B_0$

} 4-bit adder (half?/full?)



} and-gate for multiplication levels to get the rows in the adder.

HW

draw the circuit for 3 BCD # and carry.

Magnitude Comparator.

- how do we compare two bits??

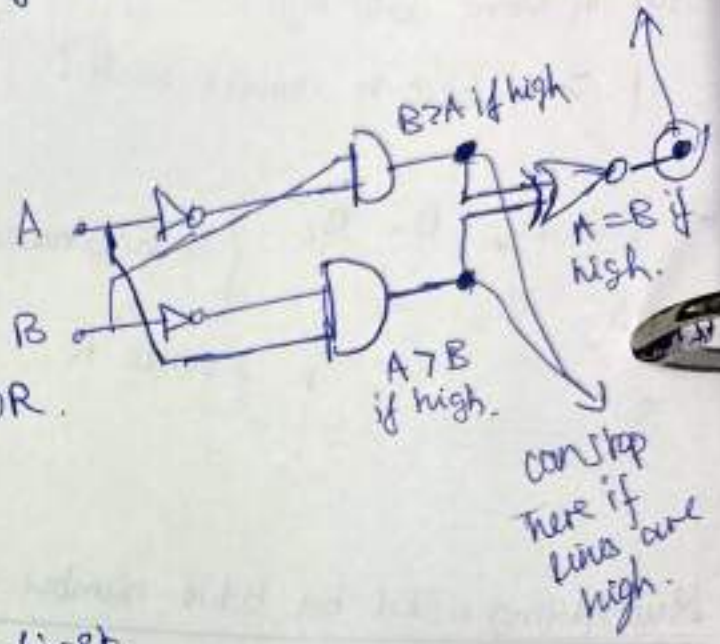
A=B	A	B	A>B	A<B
1	0	0	0	0
0	0	1	0	1
0	1	0	1	0
1	1	1	0	0

$y \wedge [(x \wedge y) - \&(x,y)]$

$A > B \Rightarrow AB'$

$A < B \Rightarrow A'B$

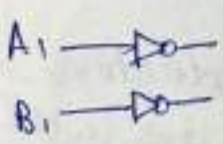
$A = B \Rightarrow (A \oplus B)' = \text{XNOR}$



- what about A_1A_0 B_1B_0.?
  - ↳ compare the highest bit first.
  - ↳ if highest bit is the same, go to the lower bit.

A_1 A_0

B_1 B_0



Decoders

$\frac{|x+y|}{1+|x+y|} \leq \frac{|x|}{1+|x|} + \frac{|y|}{1+|y|}$.

- 2/3 inputs, get multiple outputs out of it.

  3 to 8 decoders are most well known.

- each output line can be 0/1

  ∴ no. of outputs = $2^{\#\text{ input lines}}$ //.

- 3 input lines, 8 outputs.

- 4×2 array of sensors

  want to have 8 outputs but only 3 input pins.

  ↳ decoder!

  3 input lines, 8 outputs //.

- 2 to 4 decoder with an enable pin.

* converts binary to octal.

Encoders

. 8 to 3 encoder

↳ octal input (0-7).

- there are types of encoders

  if D_3 is high, rest else ignored.

  if D_3 low, checks if D_2 is high, rest ignored.

  else if D_2 low, check D_1 - ignore rest

  else if D_1 low, check D_0.

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

validity bit.

- priority encoder :

D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

0000 → x = x
1000 → x = 0
x100 → x = 0
xx10 → x = 1
xxx1 → x = 1

D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

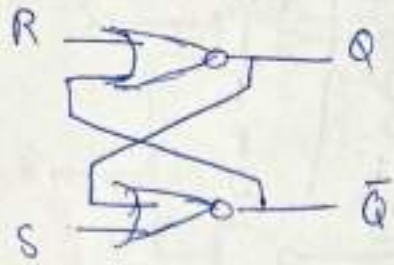
: y

$x = D_2 + D_3$

$y = D_3 + D_1'D_2$

- carry look-ahead was used to reduce delay
-

NOR- RS Latch



* in digital electronics lab
i) set state we mean we set the output $Q=1$
ii) Reset state we mean to set the output $Q=0$.

T.T for 2-input NOR

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Case 1: $S=1, R=0 \therefore Q=1, \bar{Q}=0$
Set State.
now change $S=0$ only
memory state holds previous state
$S=0, R=0 : Q=1, \bar{Q}=0$
(Q was 1 before, so $(S+Q)' = (0+1)' = 0$.
start analysis from $\bar{Q}$). → (no diff.)

Case 2: $S=0, R=1 : Q=0, \bar{Q}=1$
Reset state.
now change $R=0$ only.
$S=0, R=0 : Q=0, \bar{Q}=1$ memory state.
(Q was 0 before so $(S+Q)' = (0)' = 1$)

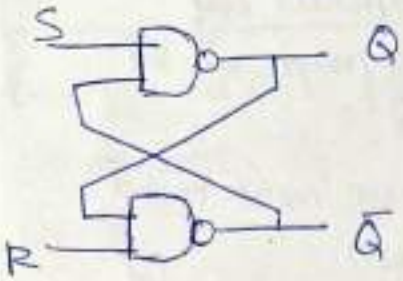
S	R	Q	$\bar{Q}$	
0	0	hold previous		
1	0	1	0	set
0	0	1	0	hold
0	1	0	1	reset
0	0	0	1	hold

Case 3: $S=1, R=1 : Q=0, \bar{Q}=0$ } contradiction already.
change S and R to 0
$S=0, R=0 : Q=0, \bar{Q}=1$
i) Q was 0 before, S is 0 now $\therefore (R+Q)' = 1$ for $\bar{Q}$
$(R+\bar{Q})'$ then is 0
ii) ~~$\bar{Q}$ was~~ $S=0, R=0 : Q=1, \bar{Q}=0$.
$\bar{Q}$ was 0 before, R is 0 now $\therefore Q=1$
$Q=1 \rightarrow \bar{Q}=0$
* circuit has entered an unpredictable state. Behaviour depends on which of S/R returns to 0 first?
$\therefore S=1, R=1$ is forbidden.

NAND-SR latch

TT for 2 I/P NAND

A	B	$(AB)'$
0	0	1
0	1	1
1	0	1
1	1	0



Case1: $S=1, R=0$: $Q=0, \bar{Q}=1$
Reset state
now change $R=1$ only now (nand logic is opposite to nor)

$S=1, R=1$: $Q=0, \bar{Q}=1$ memory, state, held the previous value

Q was 0 before $\therefore (0\cdot 1)' = 1 = \bar{Q}$
Start analysis with Q or $\bar{Q}$ does not matter.

Case2: $S=0, R=1$: $Q=1$ $\bar{Q}=0$.
set state
now change $S=1$ only now

$S=1, R=1$: $Q=1, \bar{Q}=0$ memory state

Q was 1 before $(1\cdot 1)' = 0$.

S	A	Q	$\bar{Q}$	
1	1	ho	ld	
0	1	1	0	set
1	1	1	0	hold
1	0	0	1	reset
1	1	0	1	hold

Case 3: $S=0, R=0$: $Q=1, \bar{Q}=1$ } contradiction already.

Let $S=1$ and $R=1$ now

$S=1, R=1$: $Q=1, \bar{Q}=0$ } circuit behaves unpredictably.

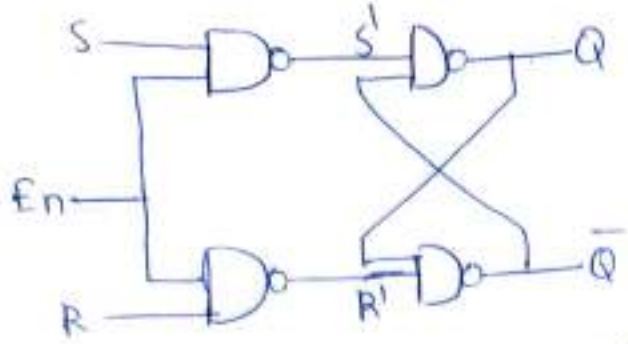
i) Q was 1 before $(1\cdot 1)' = \bar{Q} = 0$
$S=1, R=1$: $Q=0$ $\bar{Q}=1$
ii) $\bar{Q}$ was 1 before $(1\cdot 1)' = Q = 0$.

$\therefore S=1, R=1$ is the memory state for NAND-SR.

$S=0, R=1$ is the set state ie $Q=1$

$S=1, R=0$ is the reset state i.e. $Q=0$.

NAND - SR Latch With an Enabler



T.T for 2 I/P NAND

A	B	(AB)'
0	0	1
0	1	1
1	0	1
1	1	0

- If $E_n = 0$; Regardless of (S, R), $S' = R' = 1$. This means the latch is in hold states only.
  S, R do not affect the output Q. It is always at the previous state.

* $S=0, R=0$ is hold
  $S=1, R=0$ is Set
  $S=0, R=1$ is reset

- $E_n = 1$:

Case 1: $S=0, R=1$: $Q=0$ $\bar{Q}=1$
$S'=1, R'=0$. Reset State.

now change $S=1$ only

~~$S=1, R=1$: $Q=$ $\bar{Q}=$~~

$S'=0, R'=0$ X forbidden case

now change $R=0$ only

$S=0, R=0$: $Q=0$ $\bar{Q}=1$ hold. memory State.
$S'=1, R'=1$

Q was 0 before.

Case 2: $S=1, R=0$: $Q=1$, $\bar{Q}=0$
$S'=0, R'=1$ set state.

now change $S=0$ only

$S=0, R=0$: $Q=1, \bar{Q}=0$ memory State.
$S'=1, R'=1$

Q was 1 before

Case 3: $S=1, R=1$: $Q=1, \bar{Q}=1$ } contradiction already.
$S'=0, R'=0$

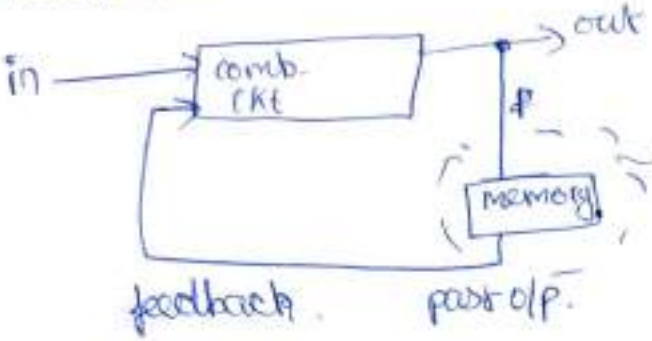
* NAND-SR with enabler has same functionality of SR as NOR-RS latch.

Sequential Circuits

Introduction

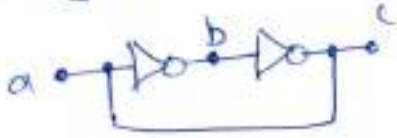
- The present output depends on the present input as well as past outputs. / present input + present state of storage elements

internal states



. everything is same as combckt except memory and feedback.

- initially people used cascaded NOT gates to store bits.



. say a=1, then b=0, then c=1

the value of A has been stored.

- To fully specify a sequential circuit, we need to know how the inputs, outputs and internal states change over time

i) i/p fed in change over time

ii) o/p change over time, depending on i/p and internal states.

iii) internal states change over time as i/ps change

L> At any time the current state can be defined by

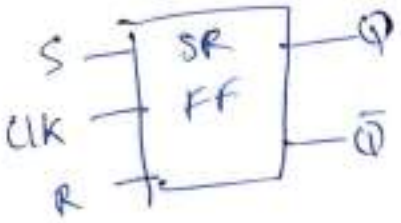
1. current i/p
2. Circuit's internal state → (depends on previous i/p to the ckt).

- ~~2 Two types of sequential ckts~~

~~Triggering Methods~~

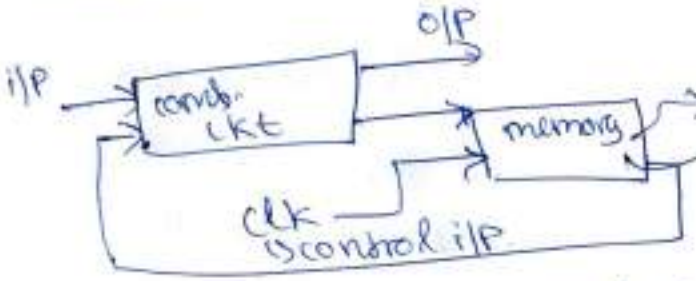
What is a clock?

- clocking times the ckt
  - ↳ decides timing of inputs



FF will only be functional when clk is high. (can design either way)

Triggering Methods in Flip Flops



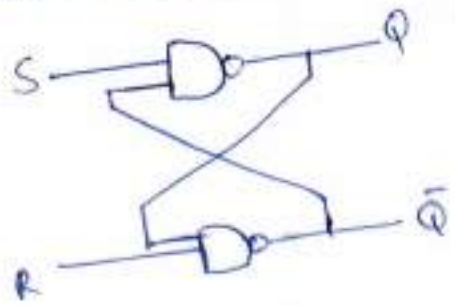
- which part do we need to trigger?
  - ↳ need to trigger memory element
  - ↳ state of memory is changed as a result of a change in i/p to it (+ depends on prev. state)
    - ↳ Stored state will be changed depending on the clock pulse aka "Triggering of FF"

① Level triggering - whenever clk is at high / low, there will be changes in the memory element

② edge triggering - (+ve E.T., -ve E.T.)
+ve: ↑ only allow a change in state stored in memory element.

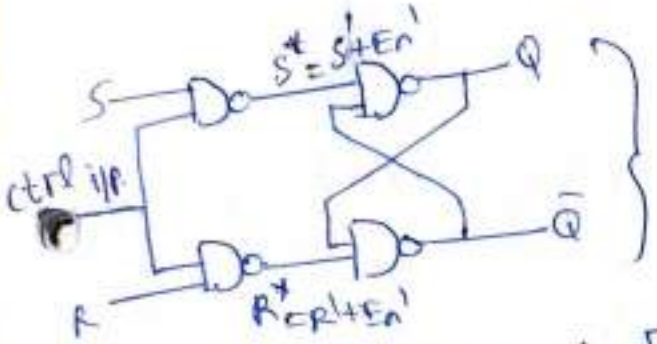
-ve : ↓ only allow a change in state stored in memory.

Difference b/w Latch and Flip Flop?



SR latch (NAND form)

- no control i/p
  - ↳ SR can change accidentally affecting o/p : introduce a control.



can be used as a latch and a FF.

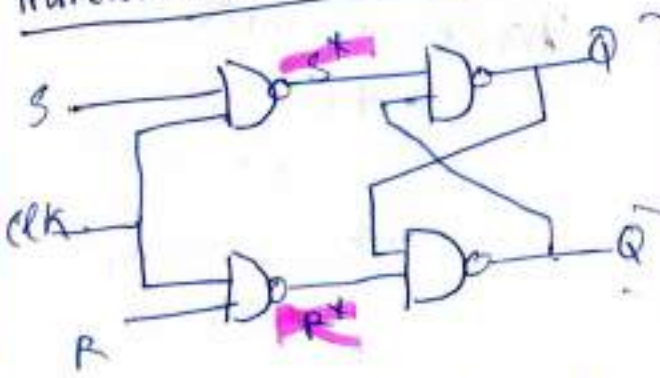
En = 1	En = 0
$S^* = S'$	$S^* = 1, R^* = 1$ → memory state

↳ if ctrl i/p is En (⎍⎍⎍) level triggering. ckt acts as a latch.

↳ if ctrl i/p is clk (⎍⎍⎍) and it is edge triggered, ckt acts as a flip flop.

* latch → level sensitive
  flip-flop → edge sensitive (+ve / -ve)

Introduction to SR Flip Flop

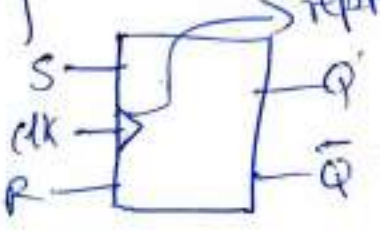


$S^* = (S \cdot clk)' = S' + clk'$

$R^* = R' + clk'$

S^*	R^*	Q	$\bar{Q}$	
0	0	forbid		
0	1	1	0	set
1	0	0	1	reset
1	1	hold prev.		

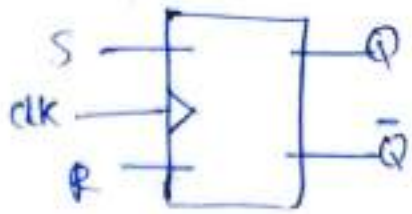
SR FF. edge triggered.



- T.T. for SR FF

clk	S	R	Q	$\bar{Q}$
0	X	X	memory.	
1	0	0	memory.	
1	0	1	0	1 reset.
1	1	0	1	0 set
1	1	1	forbidden state.	

- if clk=0, $S^* = R^* = 1$ hold state
- whenever clk is low, inputs are ignored, hold state persists.



Characteristic & Excitation Table for SR FF (rand implement only)

clk	S	R	Q_{n+1} (next state)
0	x	x	Q_n (prev. state)
1	0	0	Q_n (prev. state (hold))
1	0	1	0 reset
1	1	0	1 set.
1	1	1	invalid.

T.T. only.

- Characteristic table: find value for Q_{n+1}

  ↳ Q_{n+1} (nxt state) is dependent on inputs S,R and also the previous state:

* * clk=1

Q_n	S	R	Q_{n+1} (nxt state)	
0	0	0	0	hold
0	0	1	0	reset
0	1	0	1	set
0	1	1	invalid.	
1	0	0	1	hold.
1	0	1	0	reset
1	1	0	1	set
1	1	1	invalid.	

Characteristic table for SR FF (edge triggered).

- can write boolean expr. for Q_{n+1} using K-maps

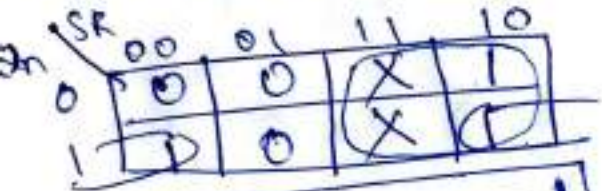
Excitation Table;

↳ current State Q_n and next State Q_{n+1} are known, need to find the current inputs S, R which give the next State.

I/P Q_n	I/P Q_{n+1}	O/P S	O/P R	
0	0	0	X	reset or hold
0	1	1	0	set.
1	0	0	1	reset.
1	1	X	0	hold or set.

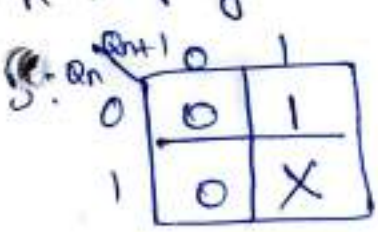
- K-map of char. Table of SR FF;

X invalid are marked with don't care's.

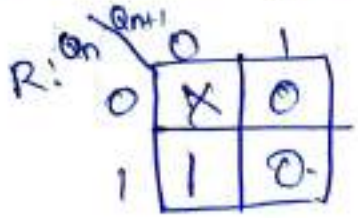


$Q_{n+1} = S + Q_n R'$

- K-map of excitation table of SR FF!



$S = Q_{n+1}$

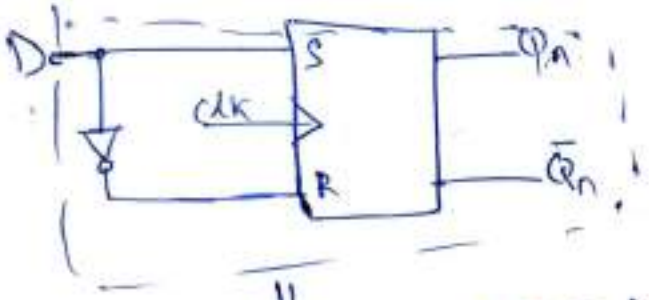


$R = Q_{n+1}'$.

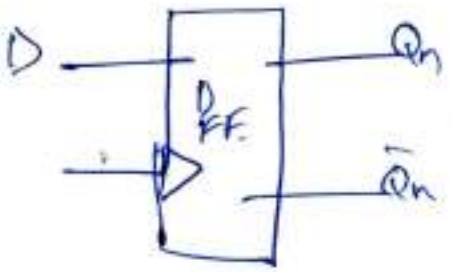
Introduction to D FF

- if $S=0, R=1$, we have 0 stored in the SR FF
  if $S=1, R=0$, we have 1 stored in the SR FF } only useful combos.
  - ↳ notice S,R are complements of each other!
    - ↳ ∴ give inputs, store data, remove clk, data will be there.

- ∴ SR to D ff; use inverters; (D stands for data i/p).



⇓ → TT for D FF;



Clk	D	Q_{n+1}
0	x	hold Q_n
1	0	0 reset.
1	1	1 set.

Characteristic & Excitation table for DFF

- Characteristic: given ~~current state~~ + current i/p, find + curr state Q_{n+1} as a function of i/p and Q_n.
- excitation table: given Q_n, Q_{n+1}, find inputs needed at n, to execute Q_{n+1} next:

* DFF will d/fo latch onto data (i/p) at next edge.

Char. table: CLK = 1

Q_n	D	Q_{n+1}	
0	0	0	reset
0	1	1	set
1	0	0	reset
1	1	1	set

$Q_{n+1} = D$ → transparent!!!

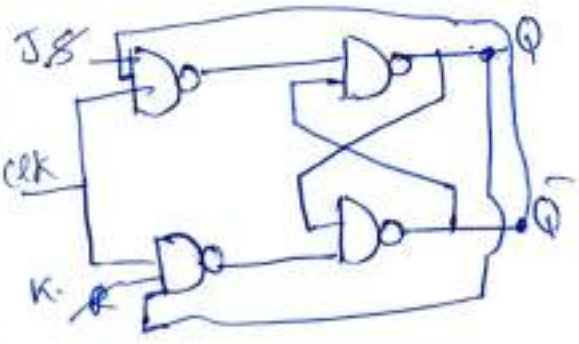
- Excitation table:

Q_n	Q_{n+1}	D	O/P
0	0	0	reset (no hold?)
0	1	1	
1	0	0	
1	1	1	

→ holds until triggering causes it to change.

Introduction to JK FF

- why an extra FF? D, SR not sufficient?
  - ↳ D FF only has 2 i/p
  - ↳ SR FF has 2 i/p when clk is 1, but one state (S=R=1) is forbidden
    - ↓
    - try to make this state useful → obtained FF is JK FF



JK FF:
- if clk = 0, (Q, $\bar{Q}$) → hold. (NAND latch)
- → if clk = 1:
  - J=1, K=0, Q=1, $\bar{Q}$=0
  - J=0, K=1, Q=0, $\bar{Q}$=1
  - J=0, K=0, Q=Q, $\bar{Q}$=$\bar{Q}$ (hold)
  - J=1, K=1, $Q = \bar{Q}_{prev}$, $\bar{Q} = Q_{prev}$

• If clk = 1, J = 1, K = 1, assume $Q_{prev} = 1$, $\bar{Q}_{prev} = 0$

$Q_{n+1} = 0, 1, 0, 1, \cdots$

$\bar{Q}_{n+1} = 1, 0, 1, 0, \cdots$

clk	J	K	Q_{n+1}	
0	x	x	Q_n	T.T.
↓	0	0	Q_n	
↓	0	1	0 reset	
↓	1	0	1 set	
↓	1	1	$\bar{Q}_n$ Toggle	

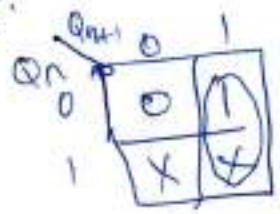
Excitation Table for JK FF

Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Characteristic Table for JK FF

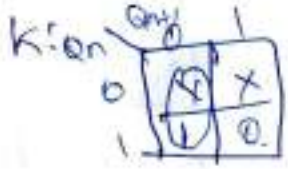
Q_n	J	K	Q_{n+1}	
0	0	0	0	
0	0	1	0	
0	1	0	1	
0	1	1	1	($\bar{Q}_n$)
1	0	0	1	
1	0	1	0	
1	1	0	1	
1	1	1	0	($\bar{Q}_n$)

J:

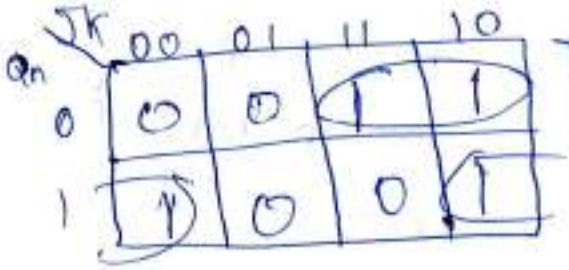


$J = Q_{n+1}$

K:



$K = \bar{Q}_{n+1}$



$Q_{n+1} = \bar{Q}_n J + Q_n \bar{K}$

✗ remember characteristic and excitation table.

Race Around Condition in JKFF

- when clk=1, J=1, K=1

  assume $Q_n = 0$, $\bar{Q}_n = 1$

	clk 1	clk 2	clk 3	clk 4	...
Q_{n+1} =	1	0	1	0	...
$\bar{Q}_{n+1}$ =	0	1	0	1	...

} output keeps changing every clk trigger.

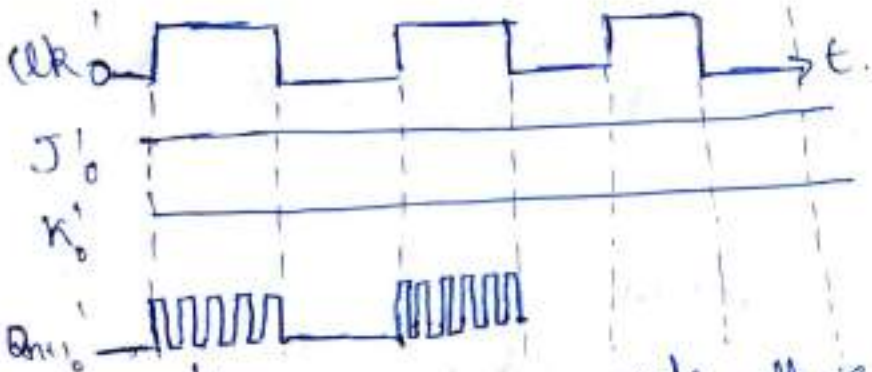
outputs are racing around.

Is toggle is controlled - happens once

Race around is uncontrolled.

while clk is high?

Is it edge triggered ff?



Racing → happens while clk is high.

Ways to overcome Racing:

i) $T/2 <$ propogation delay of FF (not practical, not used)

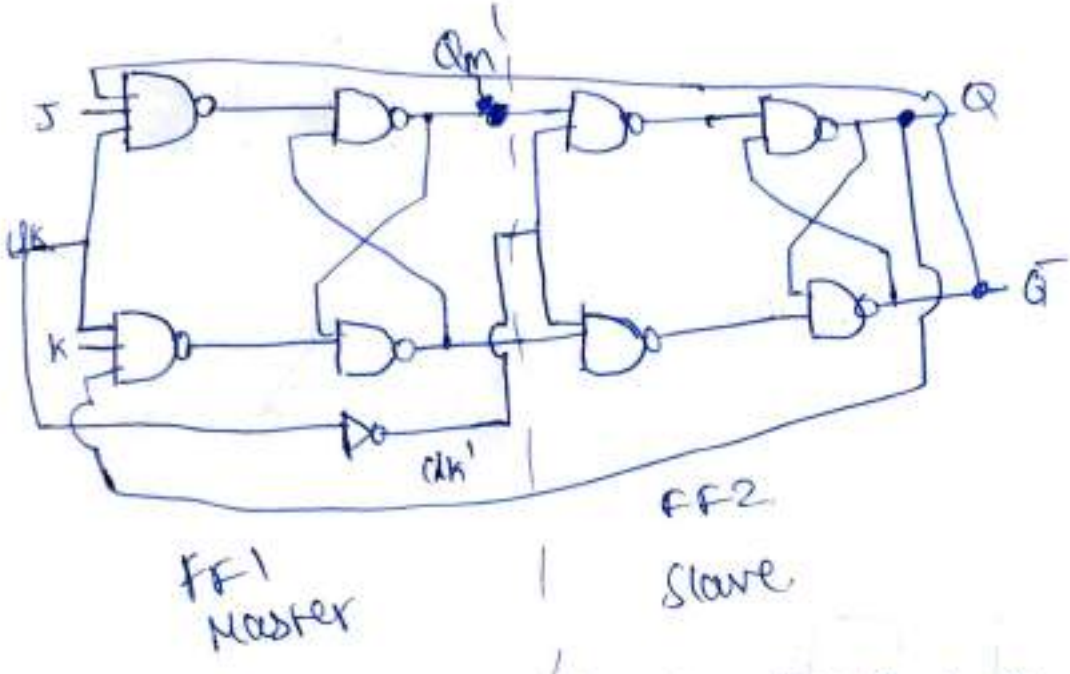
ii) use edge triggering (used)

iii) use JK Master Slave (most imp.)

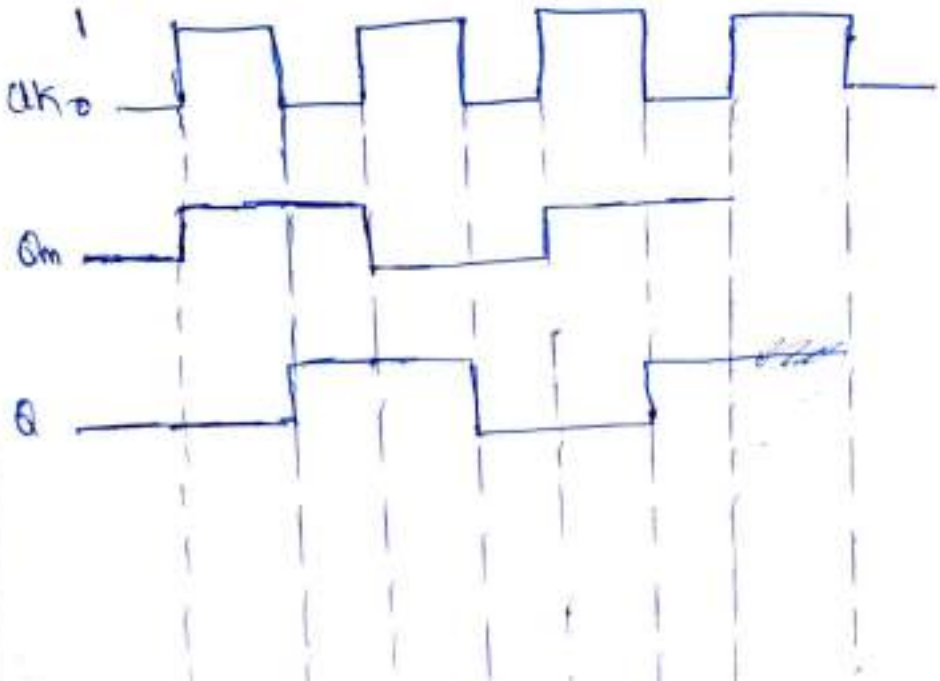
Master Slave operation is same as -ve edge triggering FF.

Master Slave JK FF

- racing due to feedback loop.
  - ↳ add one more stage.



- clk=1, master is operational, slave is not ∴ slave is at memory state.
- if clk=0, slave only is functional, output changes, bc
  - ↳ feedback from output goes into FF1, which is disabled ∴ no effect of feedback and hence no racing
  - o/p changes only once in a clk cycle (toggling)

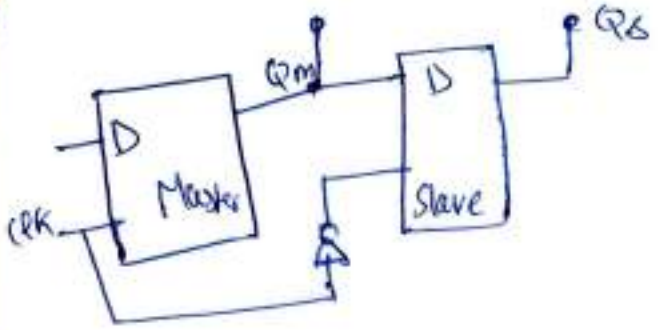


clk	J	K	Q_{n+1}
↓	1	1	$\bar{Q}_n$

↳ toggling.

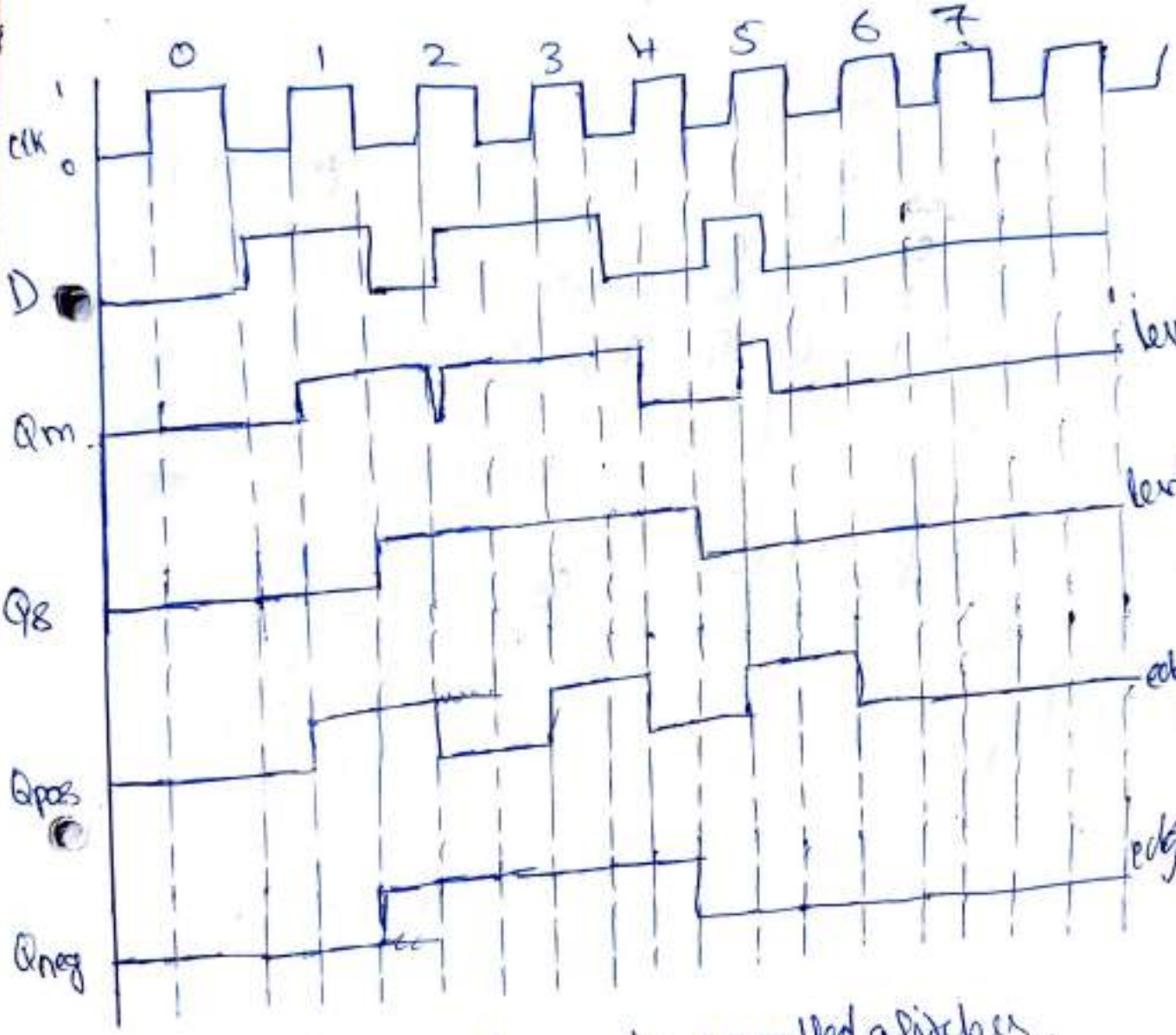
used in counters.

Behaviour of Master Slave D FF



TT for D FF:

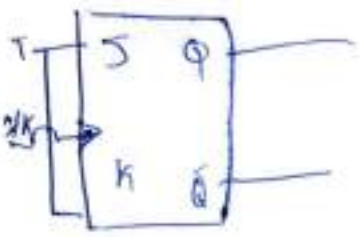
Clk	D	Q_{n+1}
0	X	Q_n
1	0	0
1	1	1



- Sudden changes in output signals are called glitches.
  - ↳ they occur in Qm, due to changing inputs when Clk signal is high.
  - ↳ They are undesireable.
  - ↳ Qs slave output has no glitches! ∴ MS FF remove glitches
  - ↳ * MS FF works as a negative edge triggered FF.
    $Q_{neg} = Q_s$

Introduction to T FF

- only store data – use D
- overcome forbidden in SR – use JK
- MS FF toggling action – controlled change of outputs
- what if we only want toggling action?



(JK MS FF) with one I/P T = TFF.

TT for TFF

Clk	T	Q_{n+1}
0	X	Q_n
1	0	Q_n (memory)
1	1	Q_n' (toggle state)

Characteristic Table for TFF

Q_n	T	Q_{n+1}
0	0	0 (memory)
0	1	1 (toggle)
1	0	1 (hold)
1	1	0 (toggle)

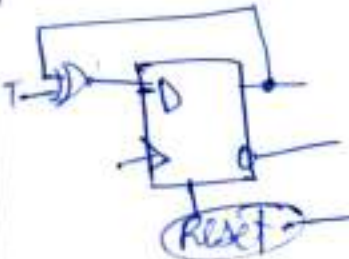
$Q_{n+1} = Q_n \oplus T$

Excitation Table for TFF

Q_n	Q_{n+1}	T	
0	0	0	hold
0	1	1	toggle
1	0	1	toggle
1	1	0	hold.

$T = Q_n \oplus Q_{n+1}$

$T = D \oplus Q$ → T using D. → but no reset?



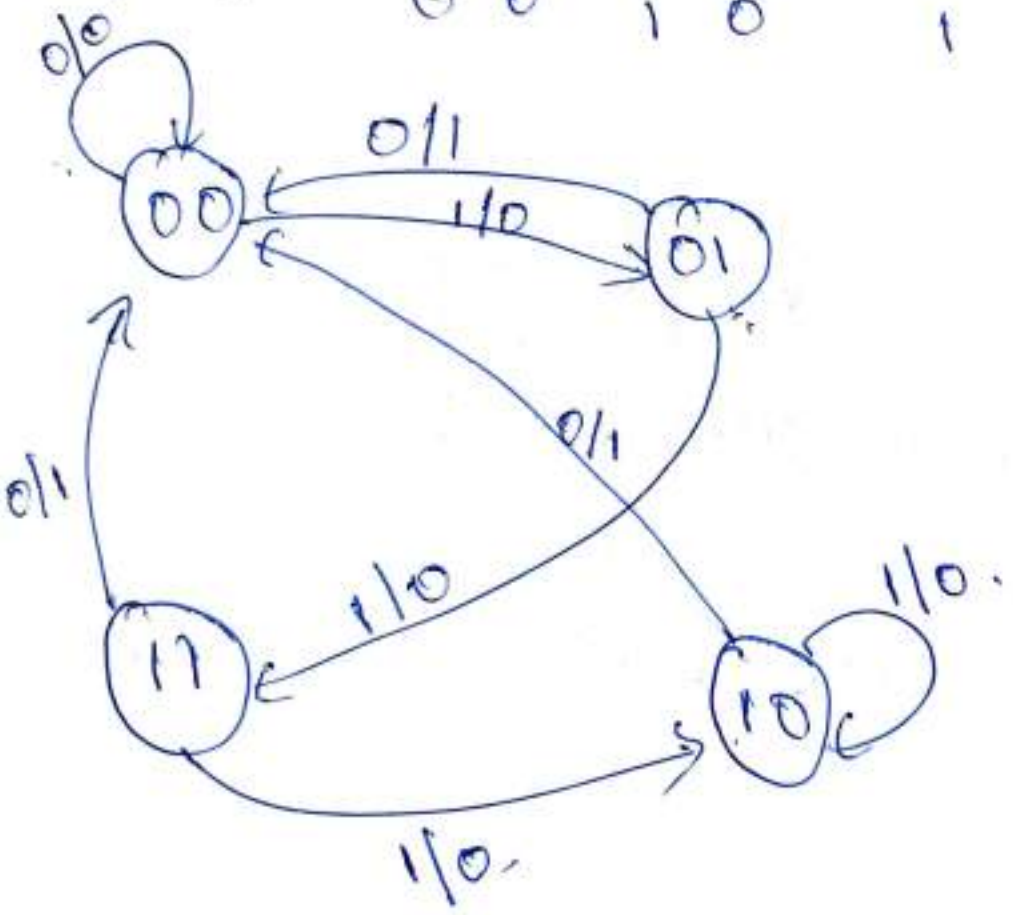
→ asynchronous i/p.

Analysis of Sequential Ckts

- need i/ps, o/ps, state of flip-flops
- Write state eq^ns for a ckt diagram:
  - ↳ A(t+1) = ... terms of A(t), B(t) i/ps to reg.
  - ↳ State eq^n specifies the cond. for a FF state transition.
  - ↳ can omit t on RHS for convenience.
- State Table
  - ↳ present states of memory elements + i/ps } list all possible binary combinations.
  - ↳ determine next state from state eq^n's or logic diagrams
- * seq. ckt w. m FF, n i/ps needs 2^{m+n} rows in the table.
- State Diagram:
  - ↳ all possible states in ○.
  - ↳ arrows showing state transitions from one state to another
  - ↳ write the i/ps needed to cause that state transition above the arrow.

Eg:)

Present St.		Next State				Output	
		$x=0$		$x=1$		$x=0$	$x=1$
A	B	A	B	A	B	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



Design Procedures

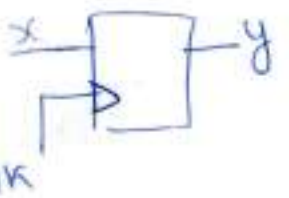
1) Derive a state diagram from the logic statement.
2) assign binary values to each state in the diagram.
3) Obtain binary-coded state table for present state + next state
4) Derive relation b/w i/p and o/p from state table.
5) Draw ckt.

Sequence of 3

- Detect a seq. of 3 consecutive "1's" in a string of bits coming through an i/p line. → or more

*need a seq. ckt only (need memory).

x → 0 1 1 0 1 1 1

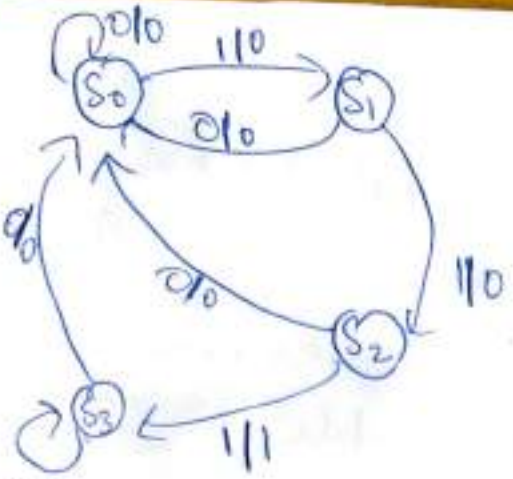


y = 1 if 3 consecutive 1's are in x
y = 0 otherwise.

- → State Diagram:

Assume initial state is S_0 (no 1's have been detected).

- if we detect a 1, we want to change state if a 1 is detected
- if we detect a 0, stay in S_0 and wait for the first 1 to arrive.
- if we get a 0 at S_1, we go back to S_0 initial state
- from S_1, if we get a 1, go to S_2.



- only way to reach S_2 is 11 as i/p sequence.
- we get to state S_3 iff 3 consec. 1's have appeared.

- ∵ 1 i/p for this ckt, each state must have two outbound arrows, for the state diagram

n i/p ⇒ each state must have 2^n outbound arrows.

- Mealy Machines and Moore Machines;

A state machines whose o/p's depend only on current state only and not i/p, is called a moore machine.
Mealy o/p depends on current state and i/p.

→ $S_0 = 00$, $S_1 = 01$, $S_2 = 10$, $S_3 = 11$. assigning binary values to states. (arbitrary assignment) to make state table

4 States to memorise ⇒ need 2 FFs.

Present State A	Present State B	Input x	Next State A	Next State B	Output y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

y only depends on present state

A_{t+1}, B_{t+1}

$A(t+1) = \Sigma(3,5,7)$ → depends on $A_t B_t x$

$B(t+1) = \Sigma(1,5,7)$ → depends on $A_t B_t x$

$y(A_t, B_t) = \Sigma(6,7)$ → depends on A_t, B_t

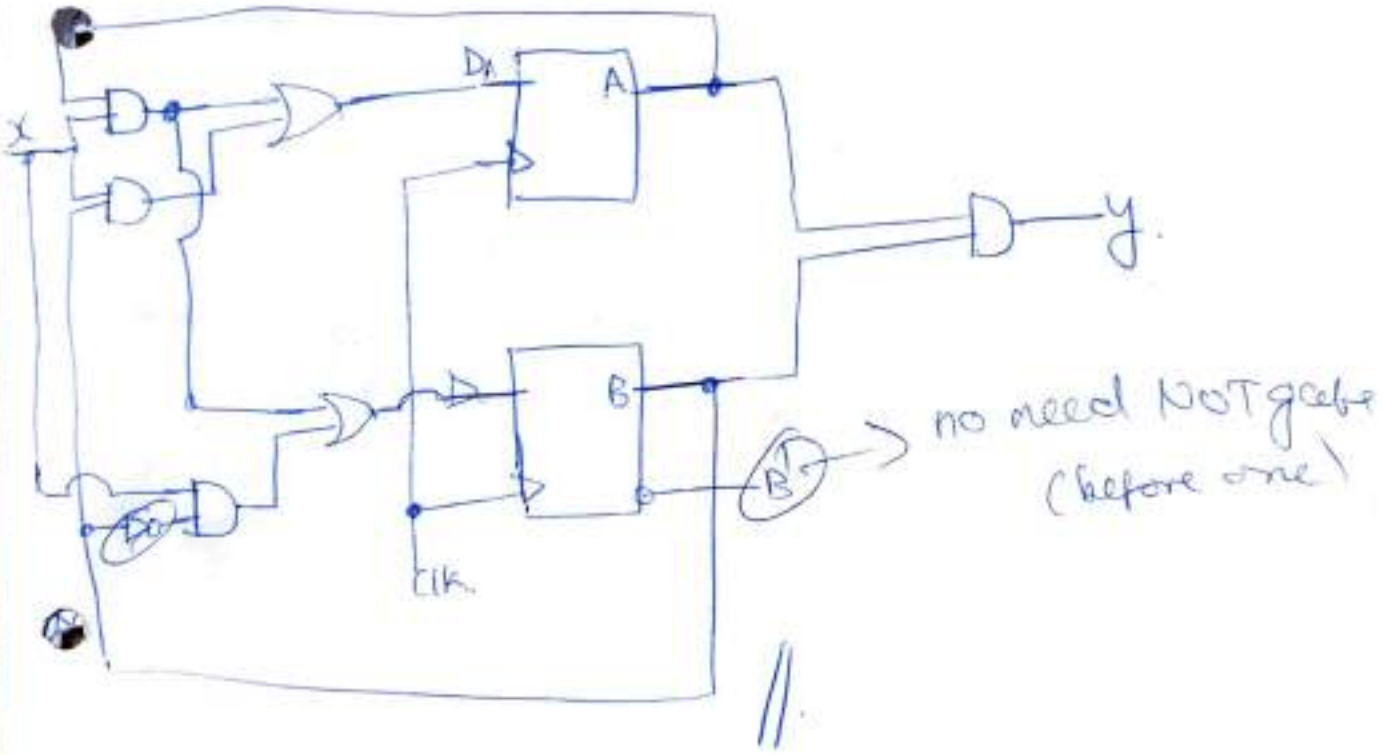
} k-map and find eqn.

→ $D_A = A(t+1) = Ax + Bx$

$D_B = Ax + B\bar{x}$

$y = AB$.

This ckt gives o/p of 1 when x is consecutively 3 times.



//

The Vending Machine

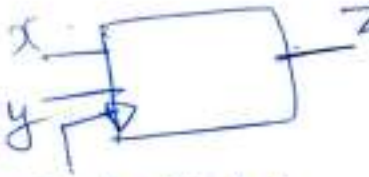
- Design a ckt for a V.M. that dispenses candy for Rs. 3. I/p consists of a coin slot that can accept Rs 1 and Rs 2 coins.

The deposit of these coins is detected by a ckt that gives out two outputs x and y — when Rs 1 is inserted, y goes to one, and when Rs 2 is inserted, x goes to one, for one clock cycle. x and y are at 0 by default.

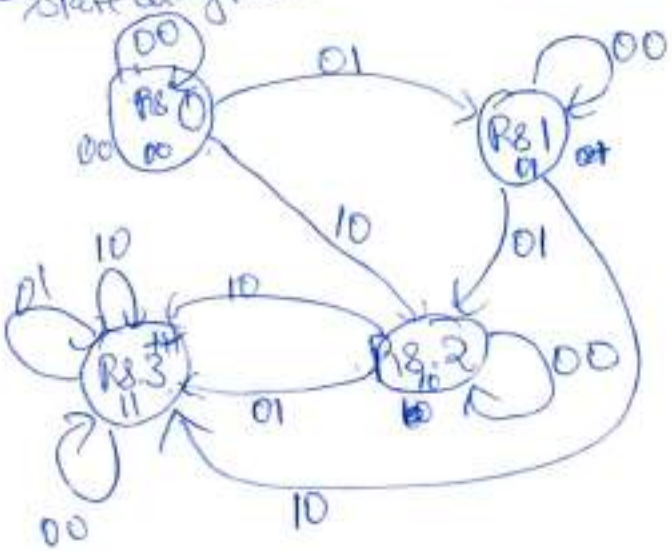
Only one coin is entered at once.

- Design a ckt that takes x and y as i/ps and o/p a 1 if the sum ≥ 3, so that candy is dispensed.

- If Rs 1: x=0, y=1 } else x=0, y=0 (no coin).
  Rs 2: x=1, y=0 }
  ↳ 3 possible i/p combinations. ∴ 3 outbound arrows for each state.



→ State diagram:



Present State A	Present State B	I/P x	I/P y	Next State A_{n+1}	Next State B_{n+1}	O/P Z
0	0	0	0			
0	0	0	1			
0	0	1	0			
0	0	1	1	x	x	
0	1	0	0			
0	1	0	1			
0	1	1	0			
0	1	1	1	x	x	
1	0	0	0			
1	0	0	1			
1	0	1	0			
1	0	1	1	x	x	
1	1	0	0			
1	1	0	1			
1	1	1	0			
1	1	1	1	x	x	

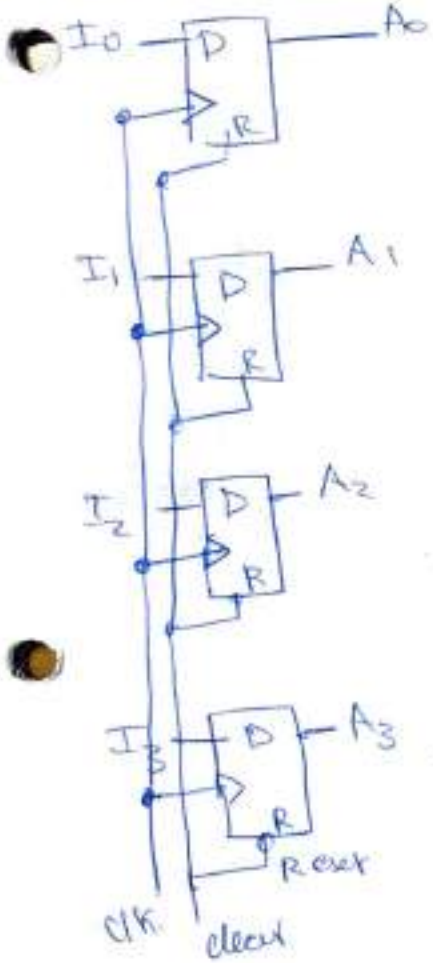
- Z is only dependent on present state.
  i.e when is (Rs 3) Z = 1;

Chapter 6: Registers and Counters

- A register is a group of FFs, each one of which shares a common clk and is capable of storing one bit information.
- An n-bit register contains a group of n-FFs capable of storing n bits of binary info.
- A counter is a register that goes through a predetermined sequence of binary states.

Registers

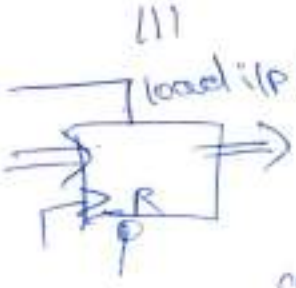
(this is a 4-bit reg.)



- clears are all connected together and clks are all connected together
- if we clear, $A_0 = A_1 = A_2 = A_3 = 0$. it means to reset all individual FFs
- Here, all i/ps are simultaneously latched at the o/p because of a common clk.
- clear = 1 for normal operation.
- Problem w this ckt: everytime there is a ↑ or ↓ info from i/ps is transferred to the register. Latching when we don't want it to latch

  ↳ create a load i/p line
  when load = 1, all info to be loaded
  when load = 0, don't allow info to be loaded

  ↳ sort of like an En signal for the registers' latching action.

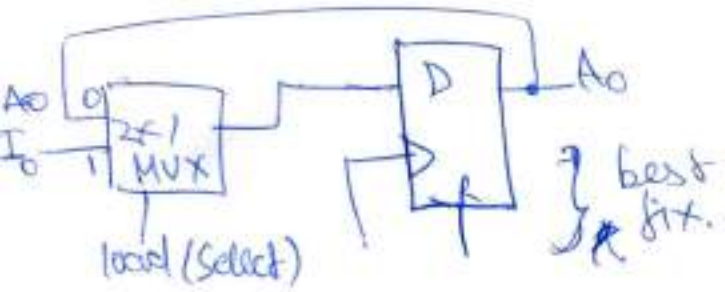


can use JK MS FF, no need load!

Register With Load Input

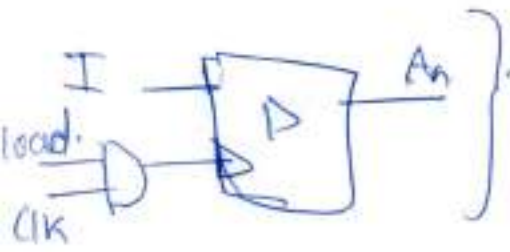
L	I_0	A_n	$D(A_{n+1})$	
0	x	0	0	} load=0 ⇒ just hold →
0	x	1	1	
1	0	x	0	
1	1	x	1	

L	D	
0	A_0	} like a MUX!
1	I_0	

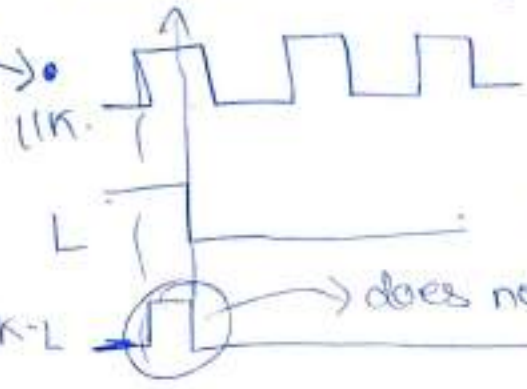


- ~~if use~~ for load=0, we select A_n state and hold in the reg.;
  When load=1, we accept i/p and store that in the reg.

- Another way to fix this issue:



- always have a gate delay, due to the extra "and" gate.



can solve by synchronising load with CLK

does not correspond to clk edges.

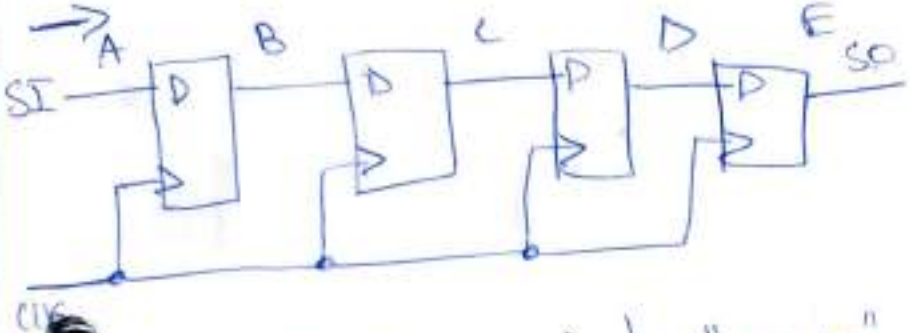
- even wires have propagation delay.
  ↳ generator one side, receiver one side L.

Shift Registers

for serial i/p / communication (serial)

- single wire communication
- serial bit stream devices
- processor needs to process info as a junk

} used in these kinds of situations



- 4 bit ~~serial~~ shift register.

every clk edge:
data from D → E
C → D
B → C
A → B.

'→' ⇔ "goes to"

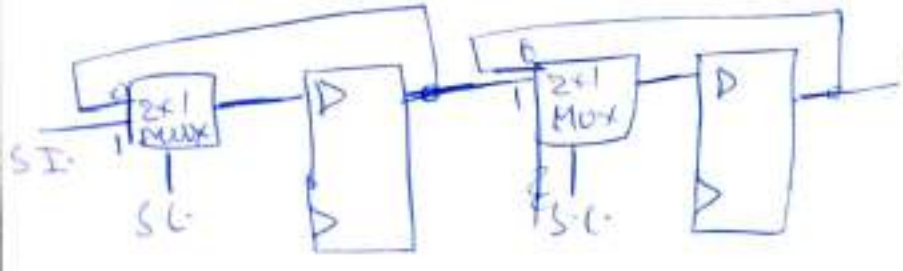
- A register capable of shifting the binary information held in each cell to its neighbouring cell, in a selected dir., is a shift reg.

- Consists of a chain of flip-flops in cascade, with o/p of one FF connected as i/p to the next FF.

- Sometimes it is necessary to control the shift so that it only occurs with certain pulses but not with others

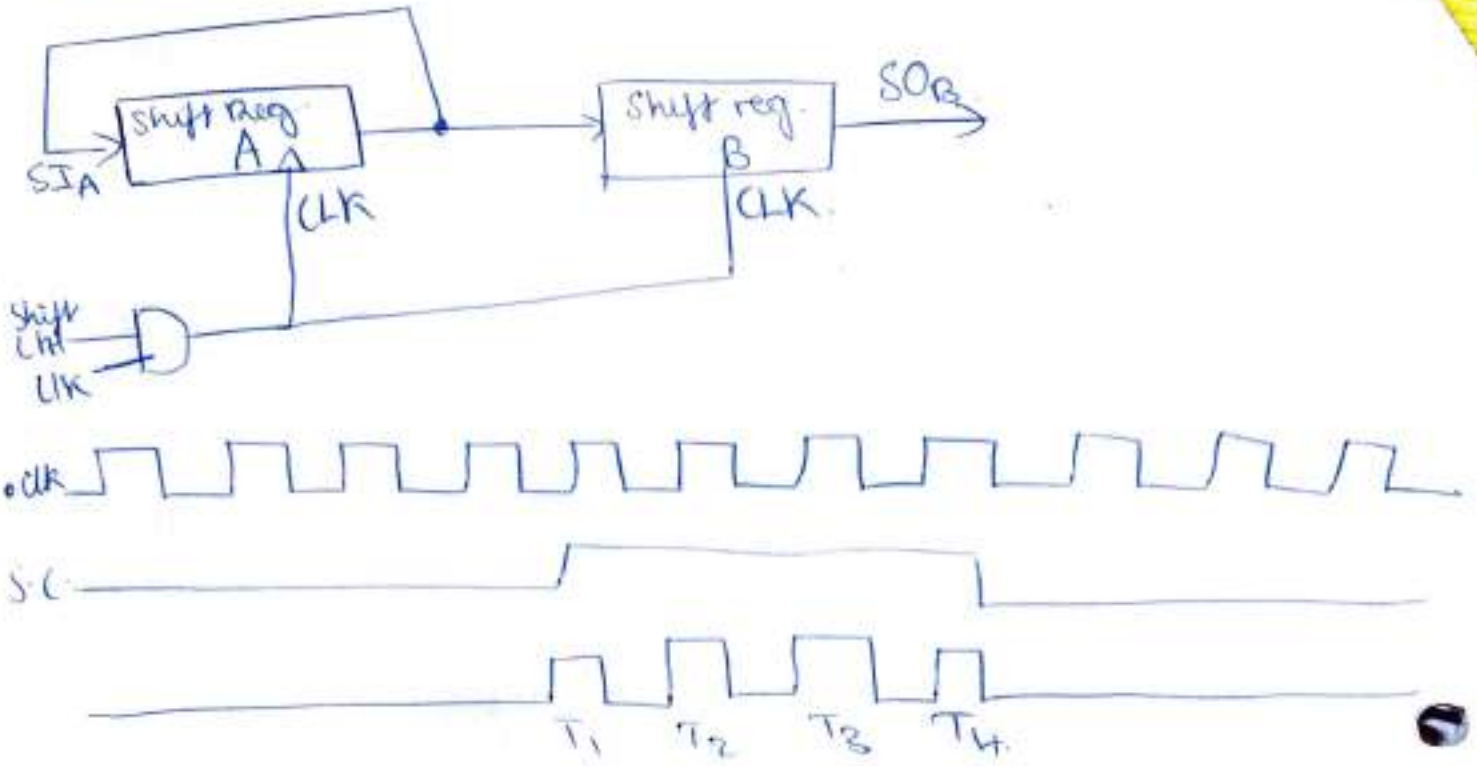
add a shift ctrl i/p line } again
S.C. = 0, don't shift, } MUX
S.C. = 1, shift.

- Recirculate the o/p of each cell back through a two-channel mux whose o/p is connected to the i/p of the cell.



- When S.C. = 1, shifting occurs.

Serial Transfer

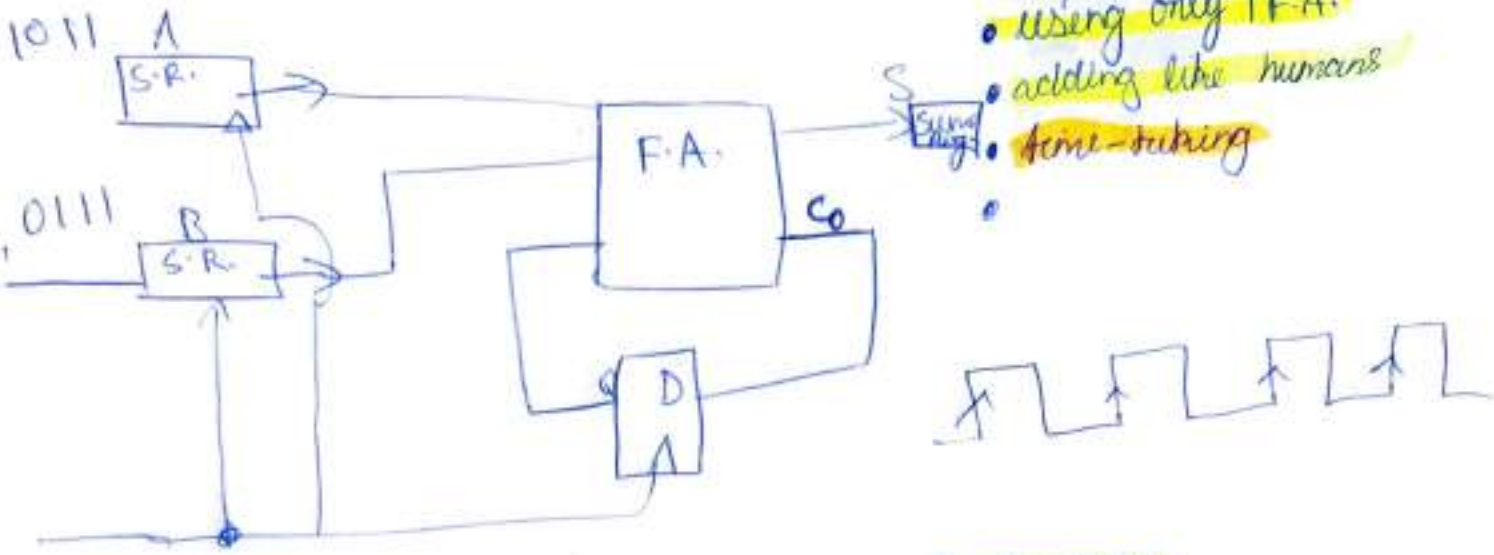


- Serial mode operation → transferring info from A to B one bit at a time.
- Say A is storing 1011 at 0th clock and B is 0000.
  - After first clock. A = 1101, B = 1000
  - After 2nd clk A = 1110, B = 1100
  - After 3rd clk A = 0111, B = 0110.
  - After 4th clk A = 1011, B = 1011.
  - info in A has been stored in A and transferred to B.
- if B had any original info, it is replaced with info from A

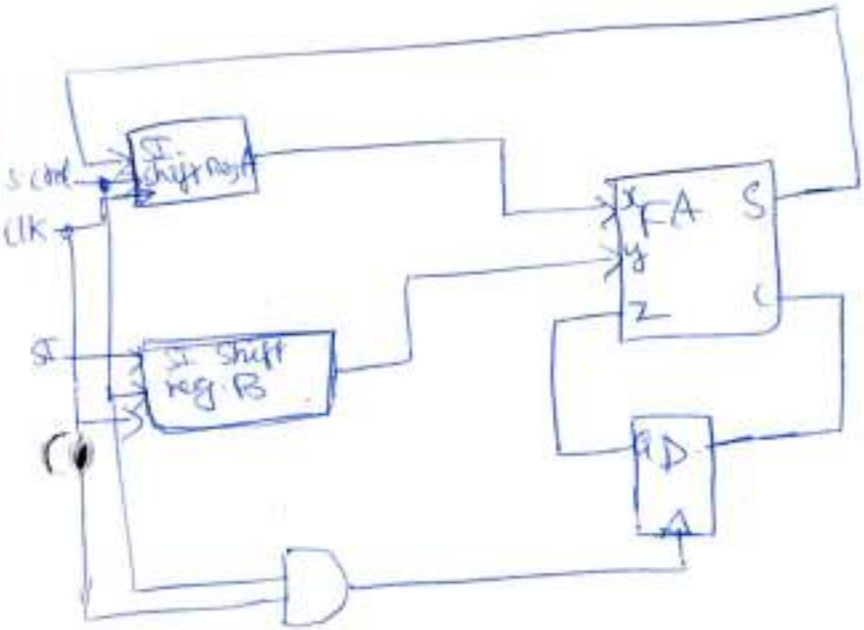
rial Addition

Two binary no.s that are serially added

- smaller chip, scalable.
- using only 1 F.A.
- adding like humans
- time-taking



disadv: comb. full adder gives output at once, serial adder needs n-clk cycles to give o/p.

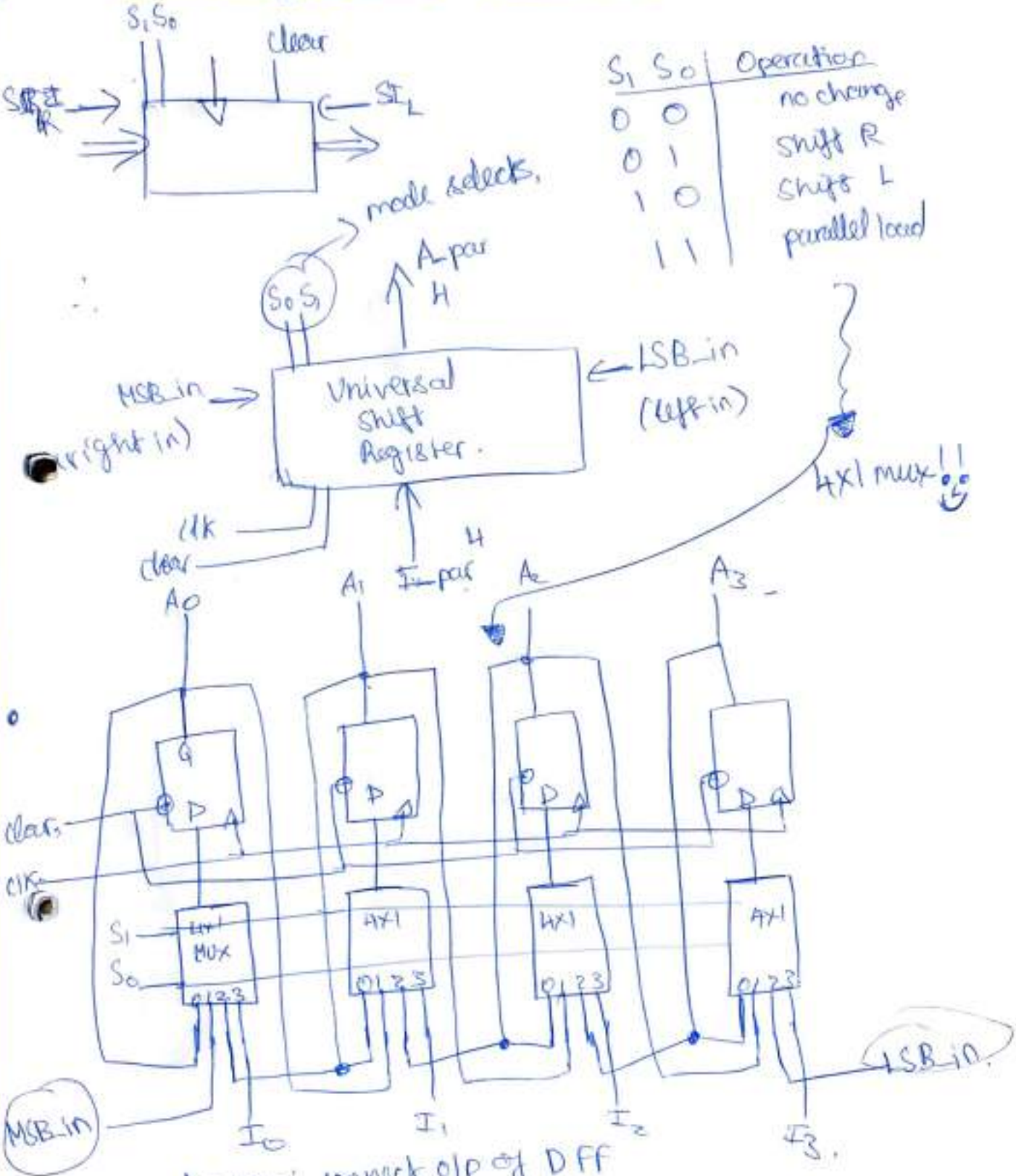


Universal Register

- if o/ps of FFs in a shift register can be accessed, then information entered serially by shifting can be taken out in parallel from the o/ps of FFs.
- if a parallel load capability is added to a shift register, then data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.
- Some reg. can shift-right and shift-left.
- Most general register has;
  1) clear control to clear reg. to 0
  2) clk to synchronise
  3) shift-right ctrl to enable shift-right and the serial i/p and o/p lines associated with shift-right.
  4) shift-left ctrl to enable shift-left and the serial i/p and o/p lines associated with shift-left.
  5) A parallel-load control to enable parallel transfer and the n input lines associated with the parallel transfer.
  6) n parallel o/p lines
  7) control state that leaves info in reg. unchanged in response to a clk.

left shift, right shift, parallel load, do nothing

4 modes of operation ∴ 2 var. to control mode



S_1	S_0	Operation
0	0	no change
0	1	shift R
1	0	shift L
1	1	parallel load

0 → no change ∴ connect o/p of DFF

1 → Right shift → left FF o/p as i/p to right

2 → L shift ⇒ right FF's o/p as i/p to left

3 → 11th load ⇒ info in I_i is transferred to reg. next edge.

Counters

- also kind of registers
  - ↳ store info, particularly a binary number indefinitely
  - ↳ don't give it a clk / power is on it will hold on to that information forever.
  - ↳ kind of registered.

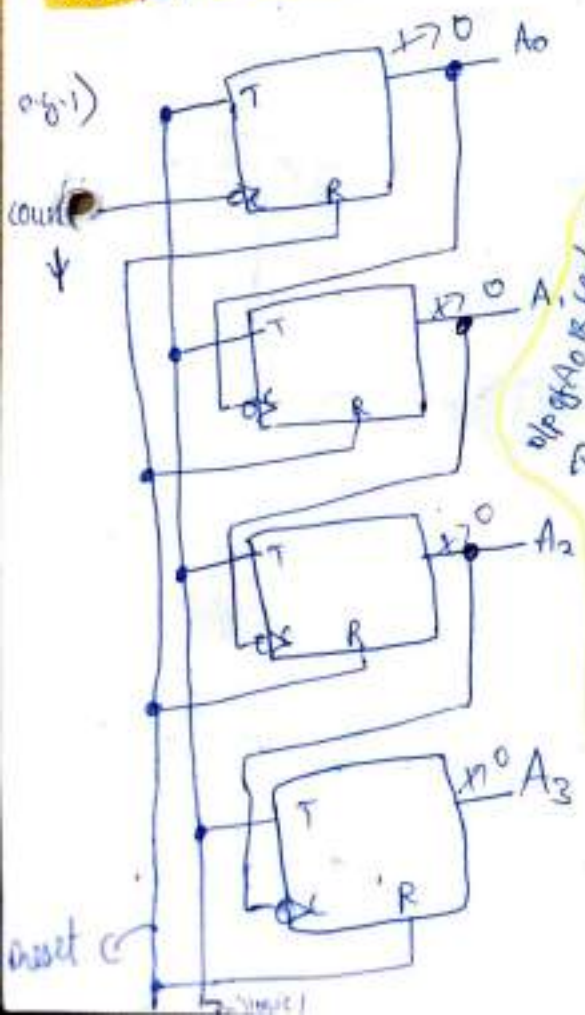
input not really needed: give it a reset and start the clk, it will loop on its own.

- given successive clk cycles it will go from one state to the next automatically.
- it's counting binary states
- two types of counters: ripple, synchronous

synchronous counter: common clk and ckt elements are used to control what kind of counting we are doing.

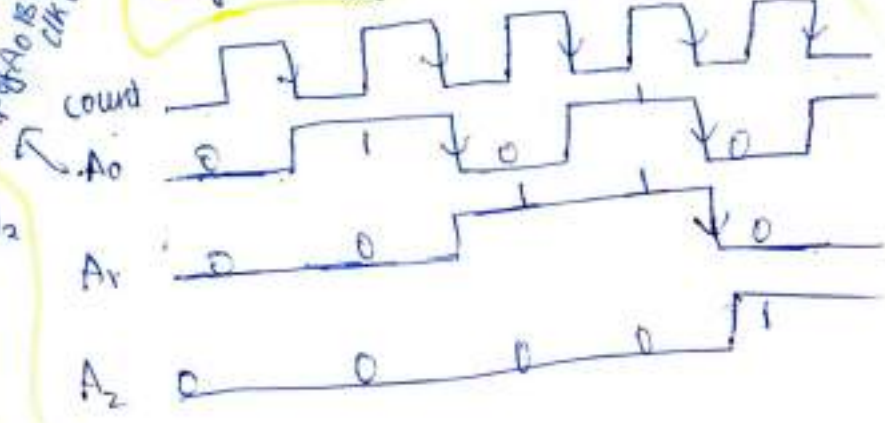
Ripple Counters

- the transition of a flip-flop is taken as a clk for the next / other FFs.



- All the T's i/ps of all the FFs are set to the logic 1
  ⇒ will always toggle
- All the T FFs are negative edge triggered FF

o/p of A0 is fed as clk for A1



you get the first take 16 counts pulse

- If we stop giving clk at any time, the number it holds onto the same as the no. of previous clk -ve edges. (Cgraph).

- Why can't this be acheived with +ve edge triggering?

A_2	A_1	A_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0

- A_0 flips only when A_0 goes from 1 to 0
- A_2 flips only when A_1 goes from 1 to 0.
- A_i flips only when A_{i-1} goes from 1 to 0. (toggles)

∴ 1 to 0 is the -ve edge!

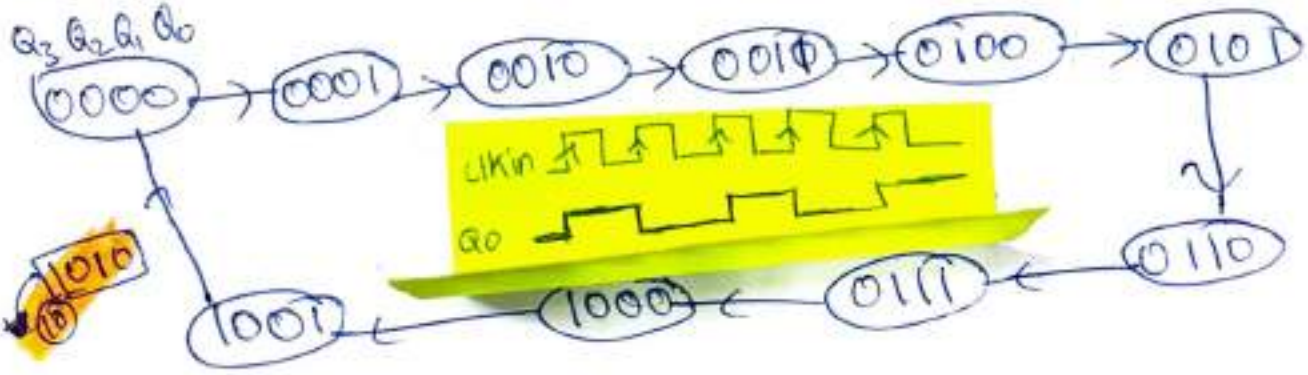
Eg- Transition from 0011 to 0100
i) A_0 is toggled from 1 to 0 @ every clk edge
ii) A_0 produces -ve edge for A_1 ∴ A_1 toggles from 1 to 0
iii) A_1 produces -ve edge for A_2 ∴ A_2 toggles from 0 to 1
iv) A_2 produced a +ve edge ∴ FF A_3 does not respond.

- 0 to 15 and then back to 0:

reset: 1111 → 0000

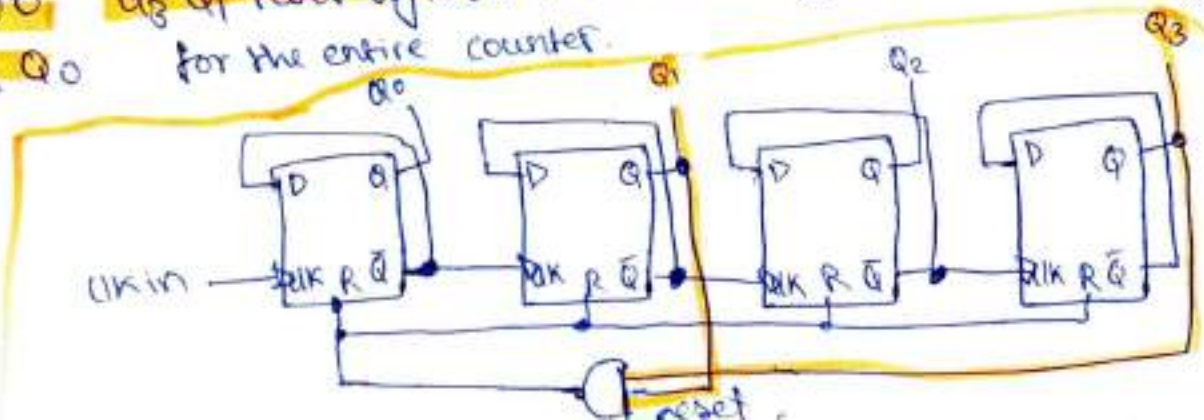
Decade Counter

- 0 to 9, after 9 back to 0
- need 4 FFs (4 bits to rep. 9)
- similar to a binary counter (0→15→0) but (0→9→0) here.



(1) 0 (1) 0 $Q_3 \cdot Q_1$ reset signal. Put an AND gate into a reset signal for the entire counter.

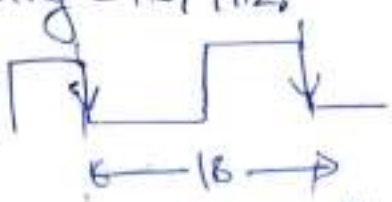
$Q_3 Q_2 Q_1 Q_0$



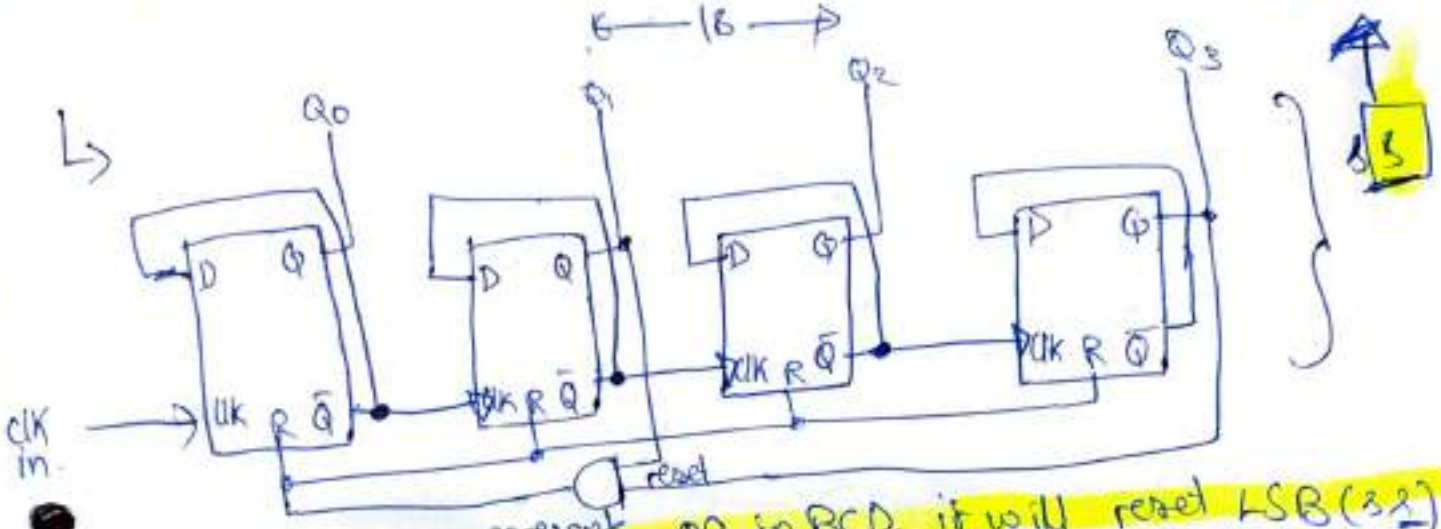
Clocks (irl) are just counters + 7 segment display

frequency of clock counting = 1Hz / 1Hz.

hh:mm:ss



decode counter



Once $Q_3Q_2Q_1Q_0$ represents 09 in BCD, it will reset LSB (ss)

to 00, and we would like the next 4 FFs (representing the next S.B.) to show 1 so that after 09 s we get 10s. How do we do this?

i) Use the reset signal: when R=1, it means 09 → 0 for LSB

Use this to start next 4 FFs from 1?

or Use reset as clk i/p for next 4 FFs.

↳ NO! reset: → almost a glitch. don't want to run things with it?

ii) Answer: When Q_3 toggles from 1 to 0, we get a -ve edge which we can use as a trigger clk_in for the next significant position.

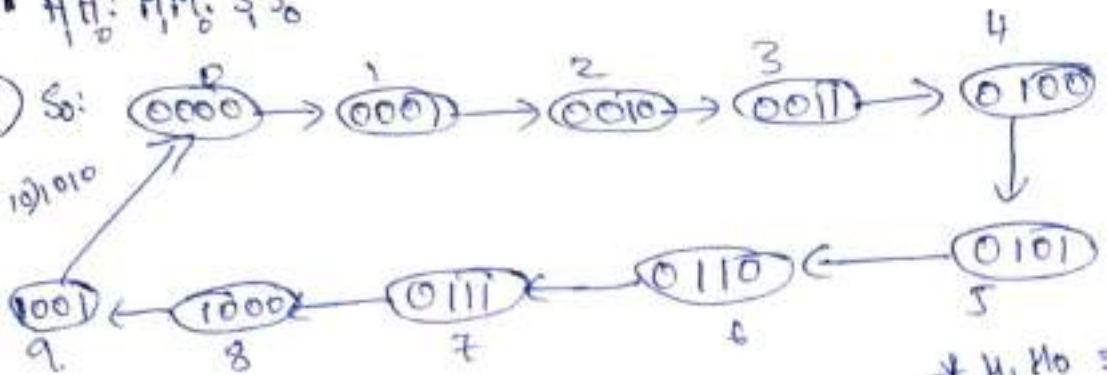
→ 19... 29... 39... 49... 59 (60) → need to change ckt.

∴ 4FFs for the 10s place of s has to reset at 6 instead of 9 like the units place of s.

∴ 0110 = 6 ∴ $Q_2 \cdot Q_1$ = reset signal for 10's place ckt.

- $H_1H_0 : M_1M_0 : S_1S_0$

i) S_0:

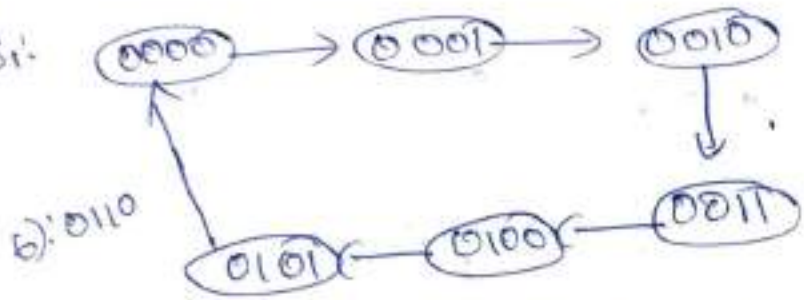


$\therefore Q_3 \cdot Q_1$ = Reset signal.

* $H_1H_0 \equiv S_1S_0$ ckts

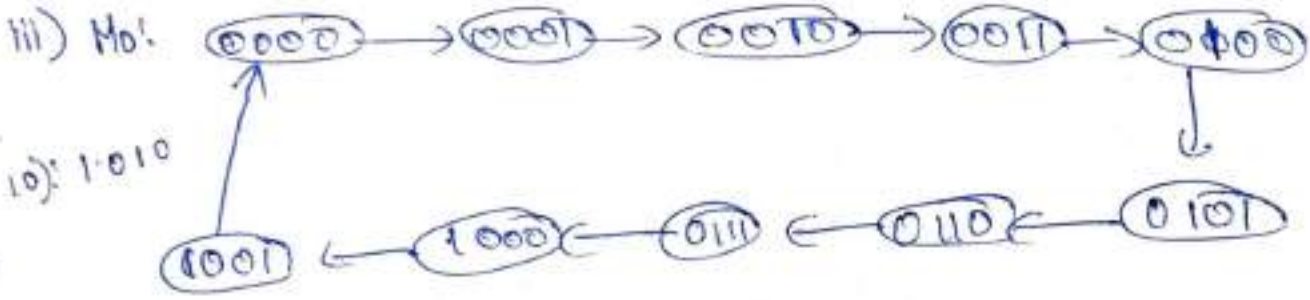
00 : 59 : 59
↓
01 : 00 : 00.
23 : 59 : 59. → return 00:00:00
↓
getting 24:00:00

ii) S_1:



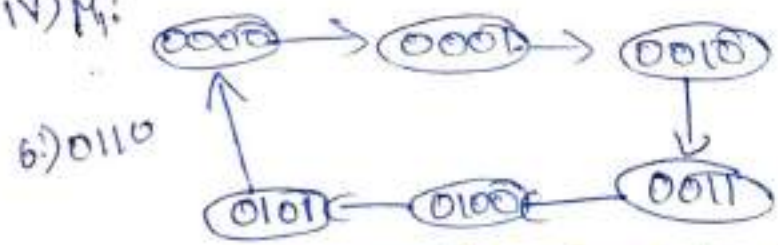
$\therefore Q_2 \cdot Q_1$ = reset ~~signal~~) Q_3 of S_0 ckt is clk in for S_1 ckt.

iii) M_0:



$Q_3 \cdot Q_1$ = reset signal, Q_2 of S_1 ckt is clk in for M_0 ckt.
↳ produces -ve edge when 59 → 00 transition happens

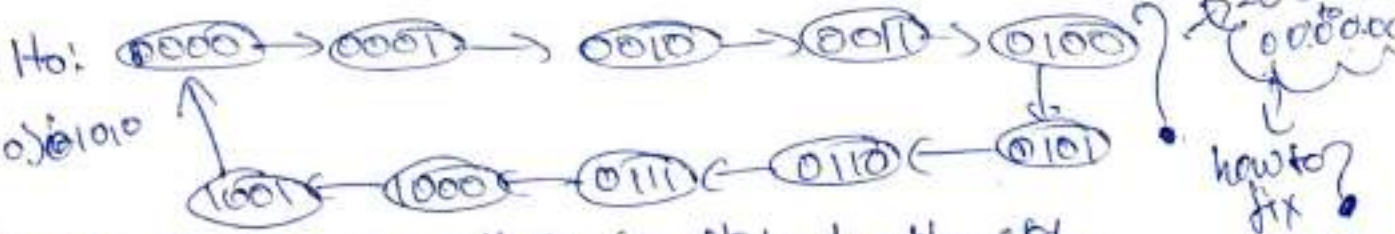
~~iv)~~ iv) M_1:



- count from 0 to 999.
- 3 decade counters

$Q_2 \cdot Q_1$ = reset signal, Q_3 of M_0 is clk in for M_1 ckt.

v) H_0:



reset: $Q_3 \cdot Q_1$, Q_2 of M_1 ckt is clk in for H_0 ckt.

Disadvantages of ripple counters:

↳ don't have a common clk.

↳ ∵ FFs are being triggered by each other, there will be a ripple delay.

↳ do not update their states immediately after the clk (it ripples) (a little bit of FF delay is there) → need to wait for ripple to go through all the FFs.

↳ Key problem

Synchronous Counters - Binary

- One clk is applied to all FFs
  - all FFs immediately change state (after 1 FF delay)
- to toggle or not to toggle depends on present situation of the counter.
- designing:

1) Make a table

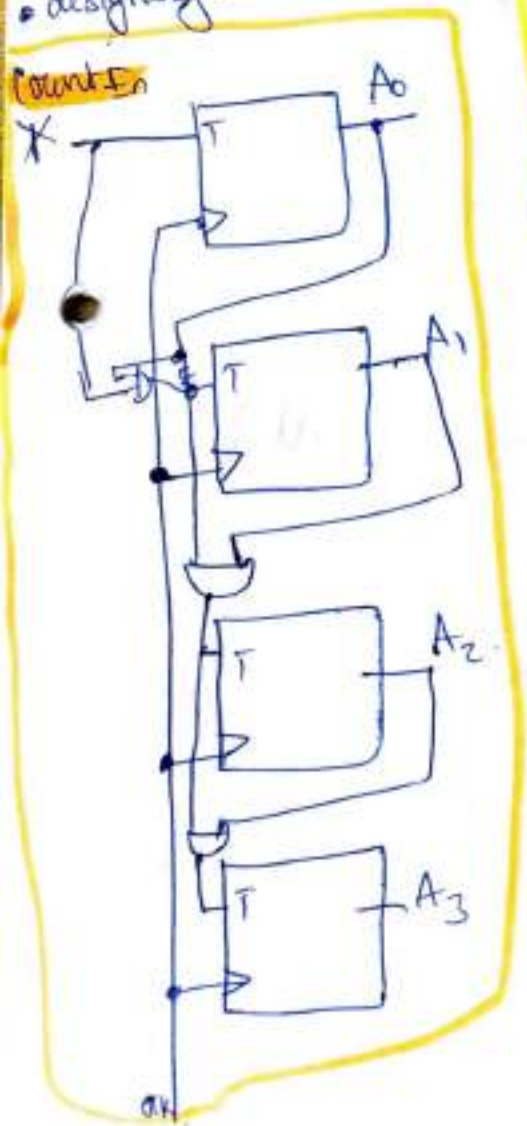
Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1

i) Q_0 toggles each clk pulse ∴ $T_0 = 1$ always

ii) Q_1 toggles when Q_0 is 1

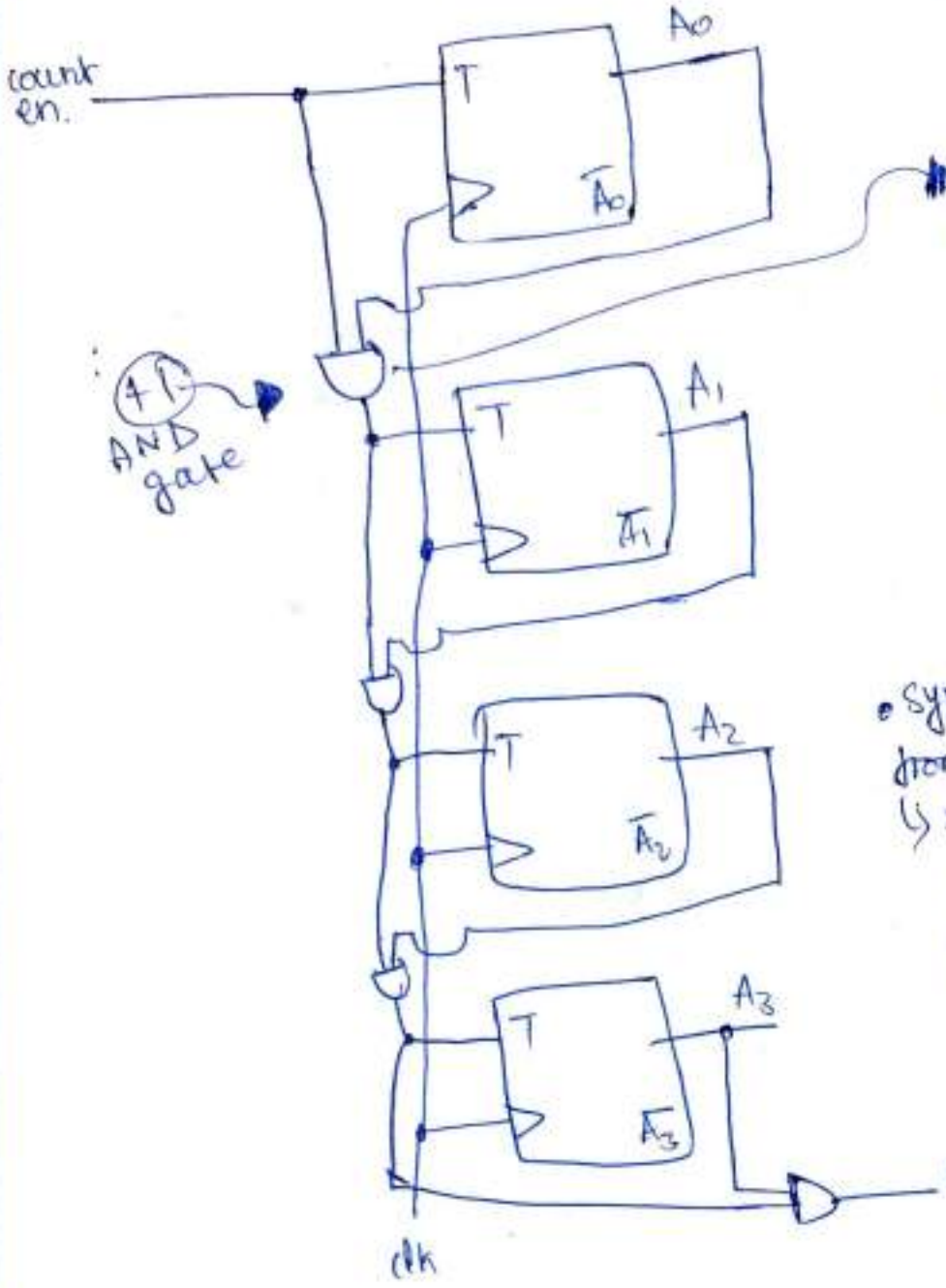
iii) Q_2 toggles when Q_1 and Q_0 are 1.

iv) Q_3 toggles when Q_2, Q_1, Q_0 are all 1.



* A FF in any position is complemented / toggled only when all its previous flip flops are equal to 1.

- Problem with synchronous counters: with a large ckt, this will immediately start counting, as soon as clk is given.
  - ↳ instead of 1, call it "Count En" (to stop counting all FFs)
  - EZ (count it) → why?



- can use +ve edge triggering here.
- Why this AND?
- if $A_0 = 1$, TFF_0 can stop but TFF_1 onwards will keep counting.
  ∴ to stop the count, we need 0 at all T i/p's.
- Synchronising is viewed from the clk edge always. ↳ right after clk edge, am I getting the right o/p or not?

Synchronous Down Counter

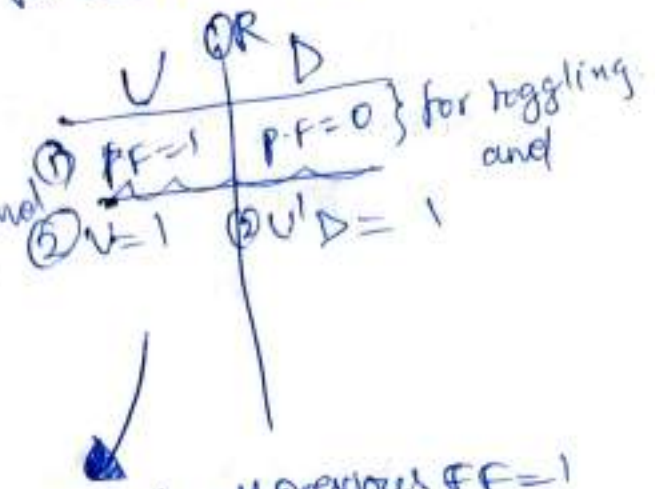
1111
1110
1101
1100
1011
1010
1001

- A bit in any position is flipped if all the LSBs are 0
- inputs for the above xored gates come from complimented o/p's of prev. FF.

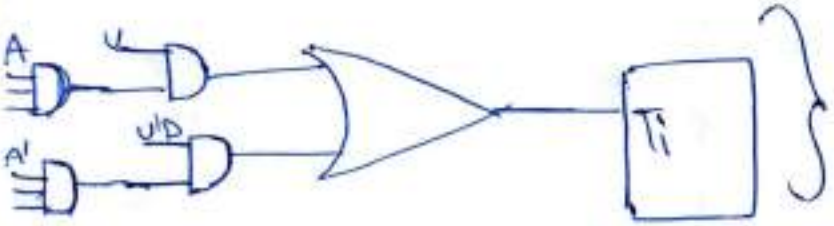
Synchronous Up+Down Counter.

MUX i A_0, A_0' MUX select for each FF ezz.

U	D	
0	0	no change
0	1	down
1	0	up.
1	1	up.



toggle if U=1 and all previous FF=1
OR
if U'D=1 and all previous FF = 0.



cascade this logic for each T_i and we get an up-down counter

Synchronous BCD Counter

- When we do resets 09 → 00.
  we do it for 10 condition i.e. (1010) $Q_3 \cdot Q_1$ = reset.

1010 state is momentarily achieved / recognised
  ↳ this may cause faults.

∴ need a better way of going from 9 to 0 in one shot.

Present States				Next State				O/P	FF I/Ps			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	y	TQ_3	TQ_2	TQ_1	TQ_0
0	0	0	0	0	0	0	1	0				
0	0	0	1	0	0	1	0	0				
0	0	1	0	0	0	1	1	0				
0	0	1	1	0	1	0	0	0				
0	1	0	0	0	1	0	1	0				
0	1	0	1	0	1	1	0	0				
0	1	1	0	0	1	1	1	0				
0	1	1	1	1	0	0	0	0				
1	0	0	0	1	0	0	1	0				
1	0	0	1	0	0	0	0	1				

- if we use D FFs, draw K maps for Q_3^+ Q_2^+ Q_1^+ Q_0^+ as a function of present states

- if we use T FFs, draw the excitation table for each next bit. make functions for TQ_i in terms of $Q_3 Q_2 Q_1 Q_0$ present.

- can use O/P y as the enable / count / clk of the next higher sgn decade

$TQ_0 = 1$

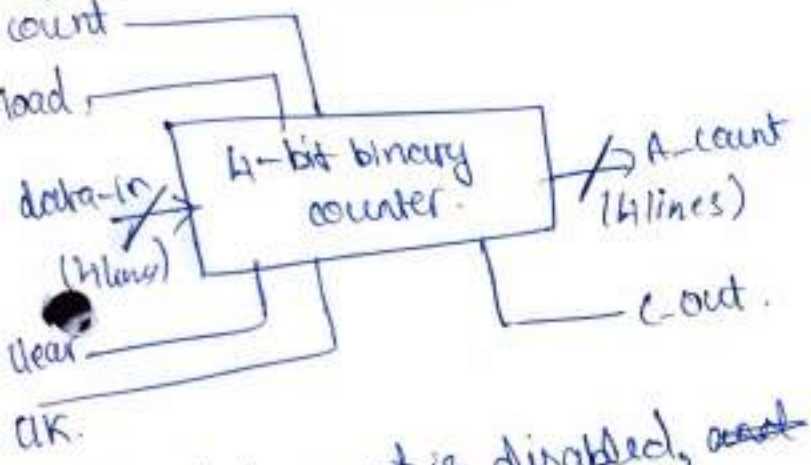
$TQ_1 = Q_8' Q_1$

$TQ_2 = Q_2 Q_1$

$TQ_3 = Q_8 Q_1 + Q_4 Q_2 Q_1$

$y = Q_8 Q_1$

Synchronous Counter With "Load"

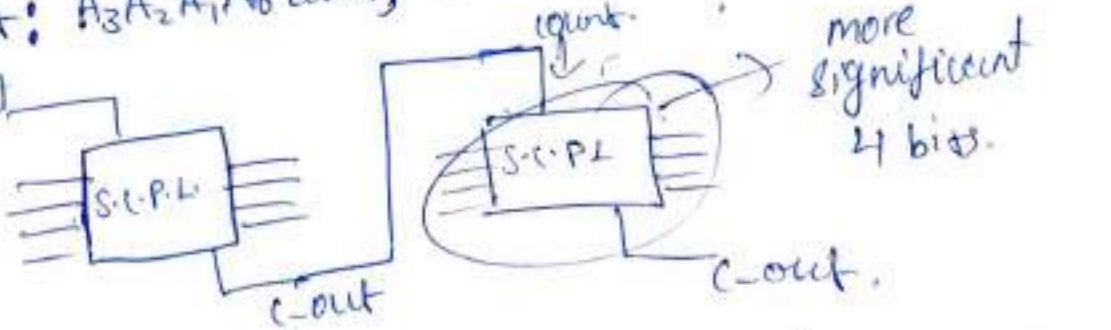


- need a "load" capability to transfer an initial binary no. before the count begins.

- if load=1, count is disabled, ~~and~~ FFs are fed with data from i/p lines.
- if load=0, count=1, count starts from load onwards and progresses
- if load=count=0, FF's state does not change.

Quiz_b clk Load Count Function, table in mano 6.6.

- c_out: $A_3 A_2 A_1 A_0 \cdot count$;



↳ helps in connecting two counters

∴ 8 bit counter.

Chapter 7: Memory and Programmable Logic

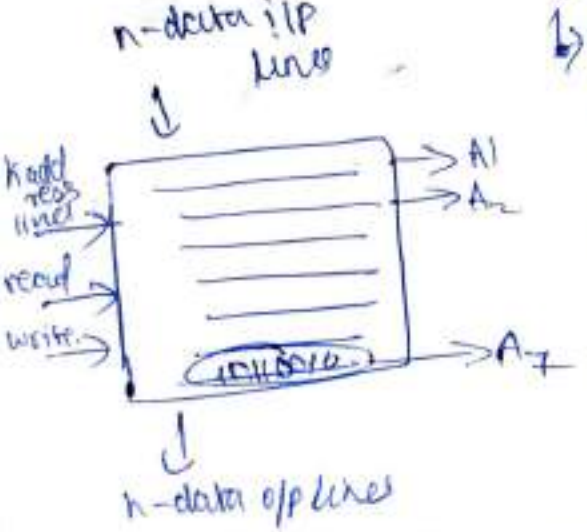
RAM

- "random-access"; the time it takes to read or write from/to any desired random location is always the same.
  - ↳ given an address and some control signals, we can read/write info to any part of RAM. in the same amount of time.
    - ↳ Magnetic tape | HDD | CDROM etc., retrieval time depends on where the info is stored physically. (disk defragmentation)
- memory unit is a collection of storage cells together with associated ckts needed to transfer information into and out of a device.
- A memory unit stores binary info in chunks/groups of bits of arbitrary length called words.
  - ↳ cannot break down into smaller groups

Can only read and write "multiples of word length" from/to the memory units.

- most computers have 8 bit long words.
- capacity of a memory unit is the total no. of BYTES that a unit can store
- each word is a collection of n-bits.
  - ↳ belongs to a specific address which can be provided from the address select lines.

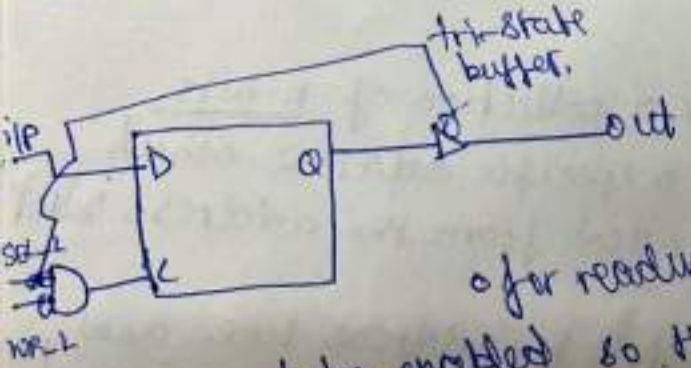
e.g.) pass A5 to address line and
i) enable read, i/p A5 is read and we get o/p from bottom
ii) enable write, A5 is written to by n-data i/p lines at the top.



- ~~need~~ cannot change only 1 bit, need to change whole word.
- Communication b/w memory and its environment is achieved through data i/p lines and o/p lines, address selection lines, and control lines that specify direction of transfer.
- n data i/p lines provide info to be stored in memory.
  n data o/p lines supply info coming out of the memory.
- K address lines $\Rightarrow 2^K$ words can be stored.
  e.g. A_1 A_0 be two address selection lines we have 4 addresses identified by $(A_1, A_0) = (0,0)(0,1)(1,0)(1,1)$.
- e.g.) capacity of a memory unit with 1000 words of 16 bits each?
  $\frac{16 \text{ bits}}{8 \text{ bits}}$ = 2 bytes per word, 1000 words $\Rightarrow$ 2KB //.

Internal Structure of RAM

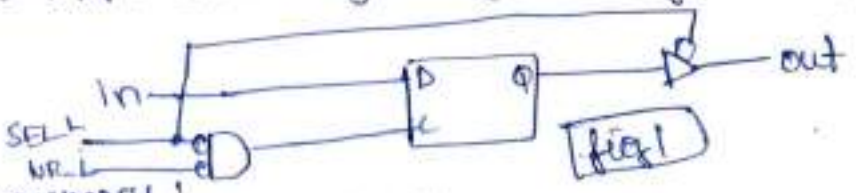
- S-RAM cells (level 0/1) : close to the processor / inside
- D-RAM cells (level 2) : outside CPU.
- We will use D-Latches as the primary RAM element and proceed.



- level sensitive D-latch.
- Say we want D i/p to be latched or enable writing, C should be high
- for reading, we want the tri-state buffer to be enabled so that whatever is at Q goes to o/p, and for writing we want C to be enabled so that D i/p gets latched.

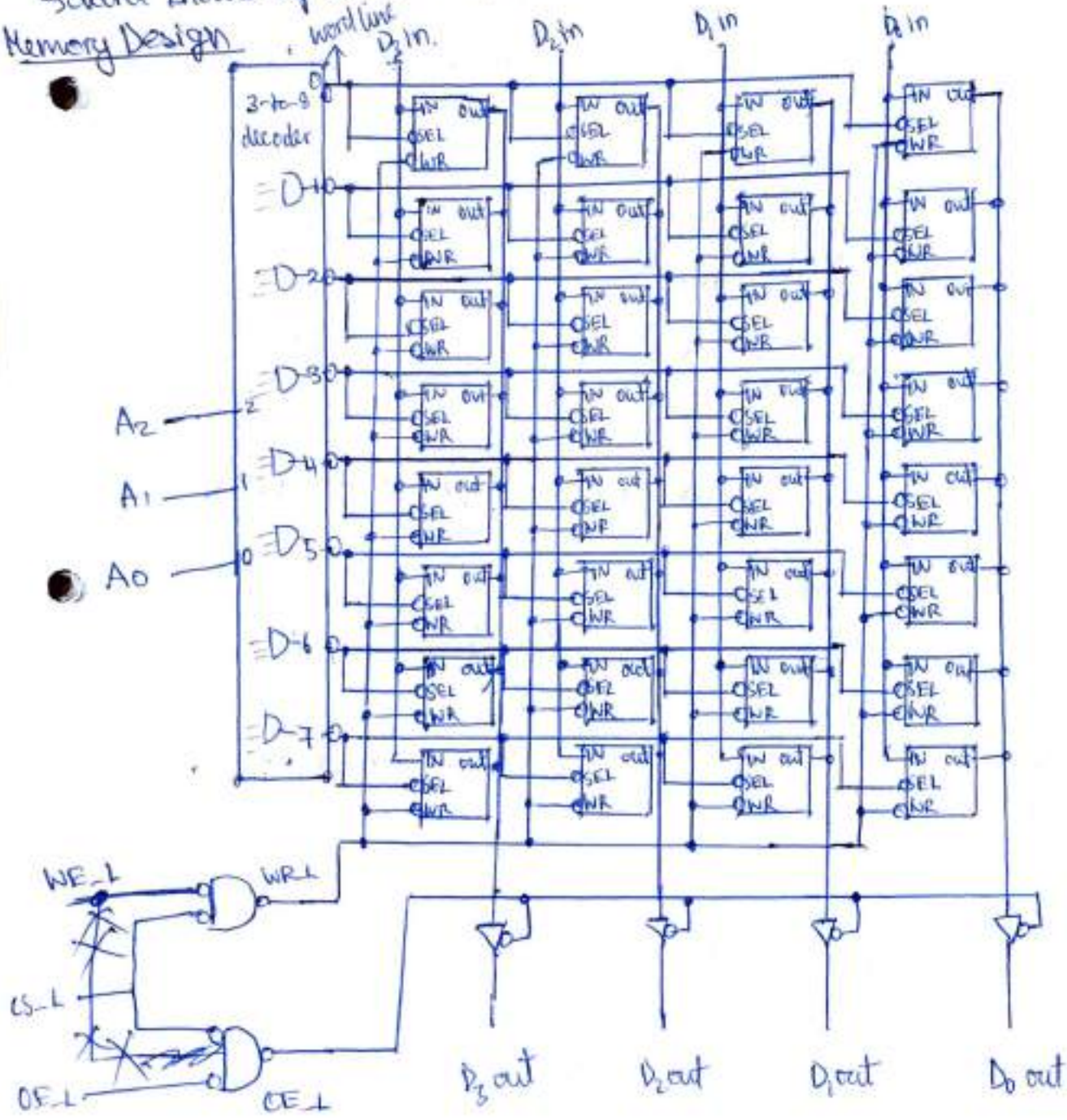
ave control signals to control behaviour of RAM:

- to transfer a new word into memory:
  1) Apply the binary address of the word to the address lines
  2) Apply chip select and write enable signals (latching happens)
  3) Apply data i/ps.



- to read a word from memory:
  1) apply the selects for that address (tri-state buffer is enabled)
  2) data shows up at the o/p.

Memory Design



• each word has 4 bits, there are 3 address lines which give 2^3 addresses.

• size of memory is 4 bytes.

• We have
① Address lines ③ o/p lines ⑤ Decoder.
② Control signals ④ i/p lines

* Decoder: (3 to 8) (combinational)
↳ given 3 i/ps ($A_2 A_1 A_0$) and 8 o/p lines, activitate only one o/p line and de-activate everything else. (each o/p is the minterm).
↳ put an address eg. 011 ∴ line 3 will be activated (high) rest will be low.
↳ line 3 now activates the latches' select lines for all latches of the word with address mapped to '3'.
∴ given an address, we are activating the select lines for one specific word.

• after selecting an address we can read/write to that address.

* To write:
↳ we have info coming in from the bit-lines (vert, hoz = word lines) for writing.
↳ these i/ps manifest at all latches, but only 1 word has active selects.

↳ enable write-enable and chip select
↳ the WR_L is going to all latches.
↳ when WR_L is enabled, all latches are ready to write, but actual writing only happens to one word (only 1 word select is enabled). ↳ (Sel_L is also anding with WR_L). (fig)

~~chip select~~ output enable is sort of like the read enable (OE)

- chip select: enables entire chip
  - ↳ goes to write enable and OE to give us actual write and read operations.

↳ entire fig. 2 is called a "page".
We can have multiple pages in memory, with the same addressing used to store other information
↳ can diff. blw pages by having diff. chip selects for each page.

* Read:
  - ↳ Apply the address.
    - ↳ selects already, sets 3-state buffer to high,
    - ∴ Q - passes to o/p automatically.
  - ↳ use the OE to enable the manifested o/ps and give the o/p.

- this ckt is sequential;
  - ↳ Given the same address and read enable, we may get diff. o/ps depending on what was written previously. ∴ not combinational.

* There is no clk, how do we make this ckt synchronous?

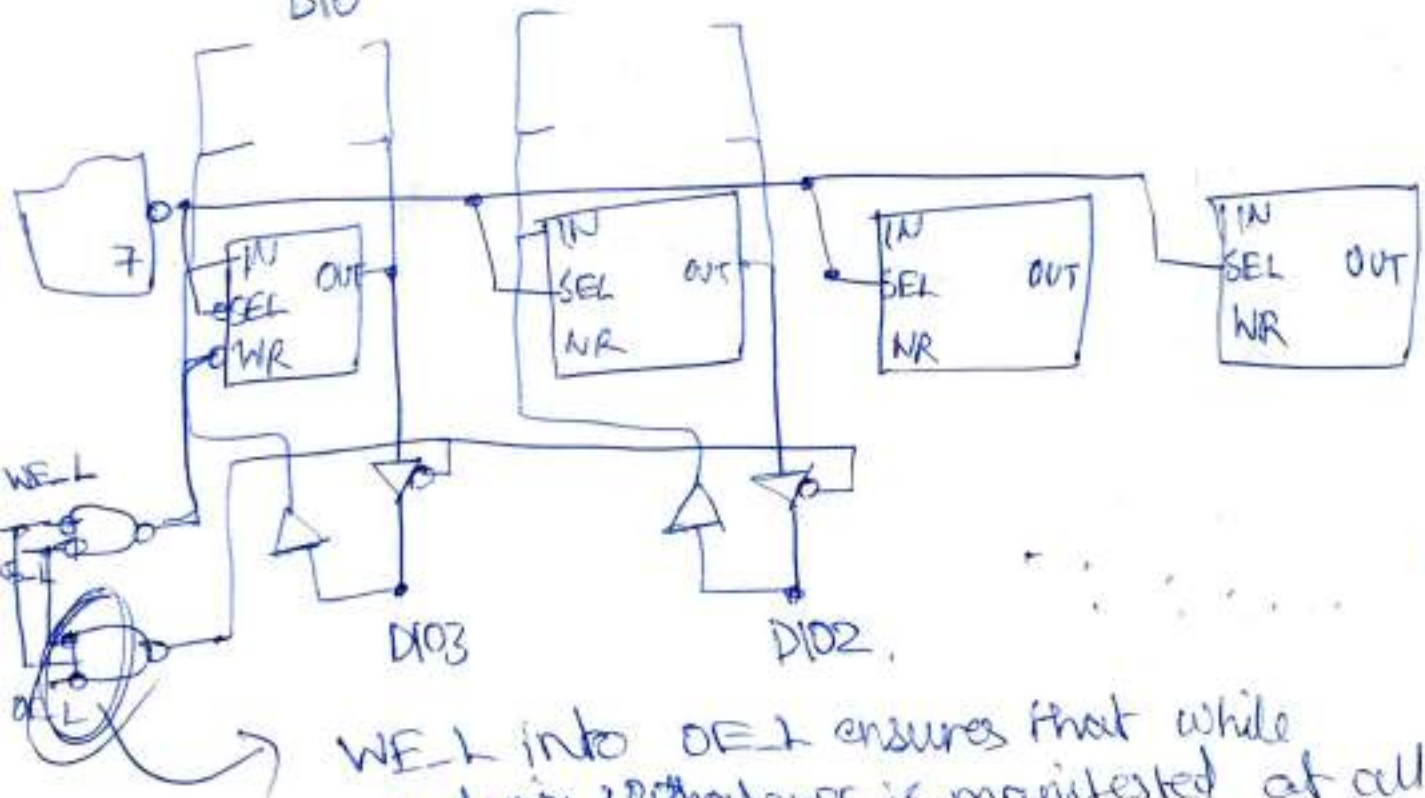
- Decoder selects a particular word line based on the address. The write enable, chip select and other i/ps are common to all the latches.

The bit-line ~~determines what the bit at a particular~~ gives i/p and o/p.

- ∃ an i/p bit line and an o/p bit line. Can we combine them into one bitline?

↳

D10 pin : takes i/p and gives o/p.
if o/p is enabled, we read o/p
else we force it to take i/p.

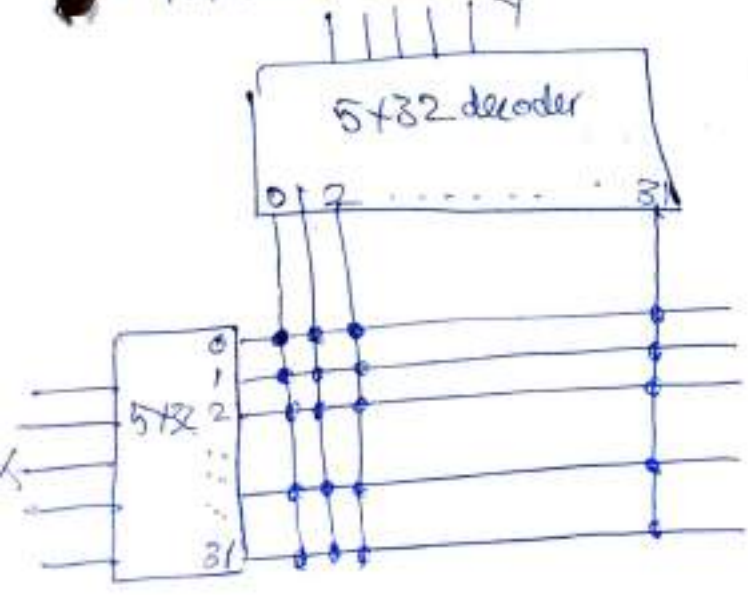


WE_L into OE_L ensures that while reading, whatever is manifested at all the i/p is not latched onto.

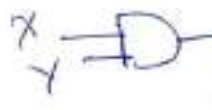
~~Synchronous RAMs~~

Coincident Decoding

- a decoder takes K i/ps and gives 2^K o/ps obtained by having 2^K AND gates with K-inputs per gate.
  - ↳ converts from the address provided by the user to the word line that is controlling the entire words selection.
- the total no. of gates and the no. of i/ps per gate can be drastically reduced by employing two decoders
  - in a 2D-scheme.
- Arrange memory cells in an array that is as close as possible to a square.
  - ↳ two K/2 i/p decoders are used instead of one K i/p decoder.
  - ↳ one decoder performs row selection, other col. selection.
- Say we have a 10-bit select line. ~~Use two~~ Instead of one 10×1024 decoder, use two 5×32 decoders.



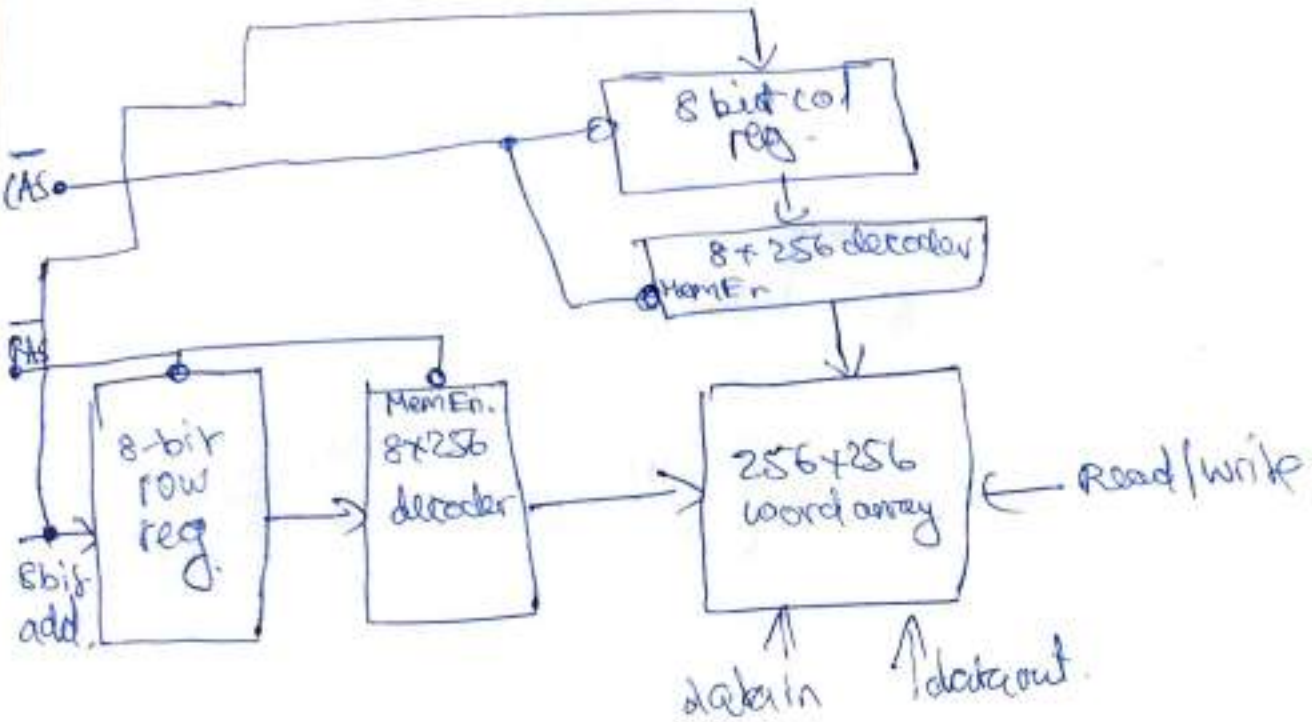
- connect an AND gate at all



(imagine word is coming out of the paper).

- X line 4 MSBs
  Y line 4 LSBs

Address Multiplexing



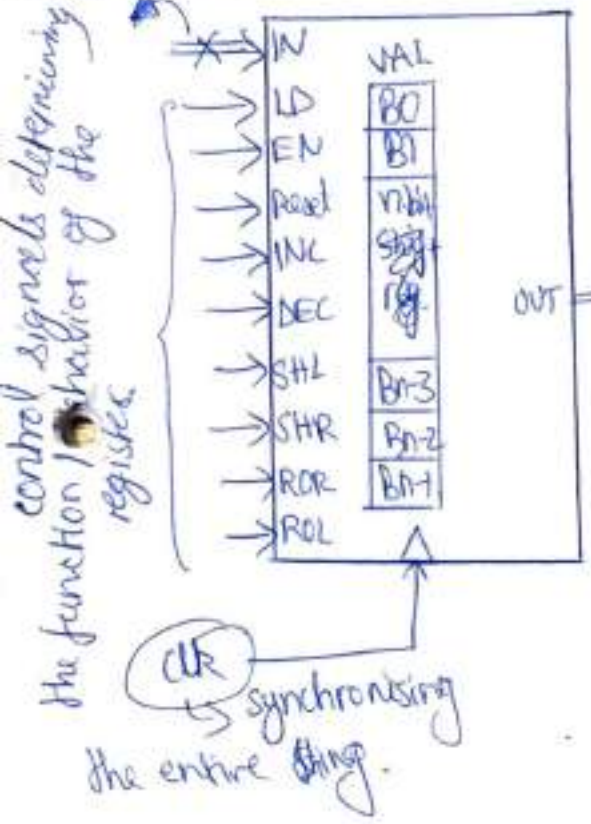
- say we have a 16 bit address. We could have 16 bit address select lines but;
  - ↳ use only 8 bit address line coming into memory.
  - ↳ divide address into two parts of 8 bits (X, Y)
- put the X part onto the i/p address lines, activate RS; latching will take place onto the reg.
- then, disable RAS, ~~activate~~. put in Y into on address lines, enable CAS; during nxt clk, Y will get latched.
- decode it using 8×256 decoders (coincident decoding).

∴ Instead of using 16 pins, we use only 8

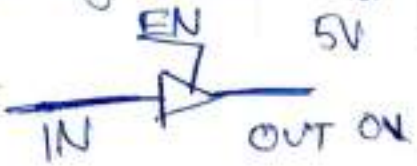
Chapter 8: Processor Design

Lecture 1

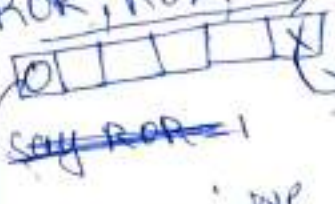
The Universal ~~Shift~~ Register



- How do we design a universal register?
- With all of the control signals, the universal reg. can not only store bits of info, but can also perform basic arithmetic.
- LD=1 and clk edge arrives; info from i/p lines gets stored in the register.
- if EN=1, info inside the reg. manifests at the o/p lines
  if EN=0, o/p will be floating
  assume that o/p is controlled by tri-state buffer



- Reset=1 and clk comes, value inside register is reset → SYNCHRONOUS reset (∵ need clk edge)
- SHL is shift left
- if INC, increment or decrement the value stored in the register.
- ROR; ROL:



~~say ROR~~ : we get a new 0 here.

if we SHR=1, this bit is discarded (∄ serial out line)

∴ in ROR, least bit is NOT lost, it is fed back to the first position.

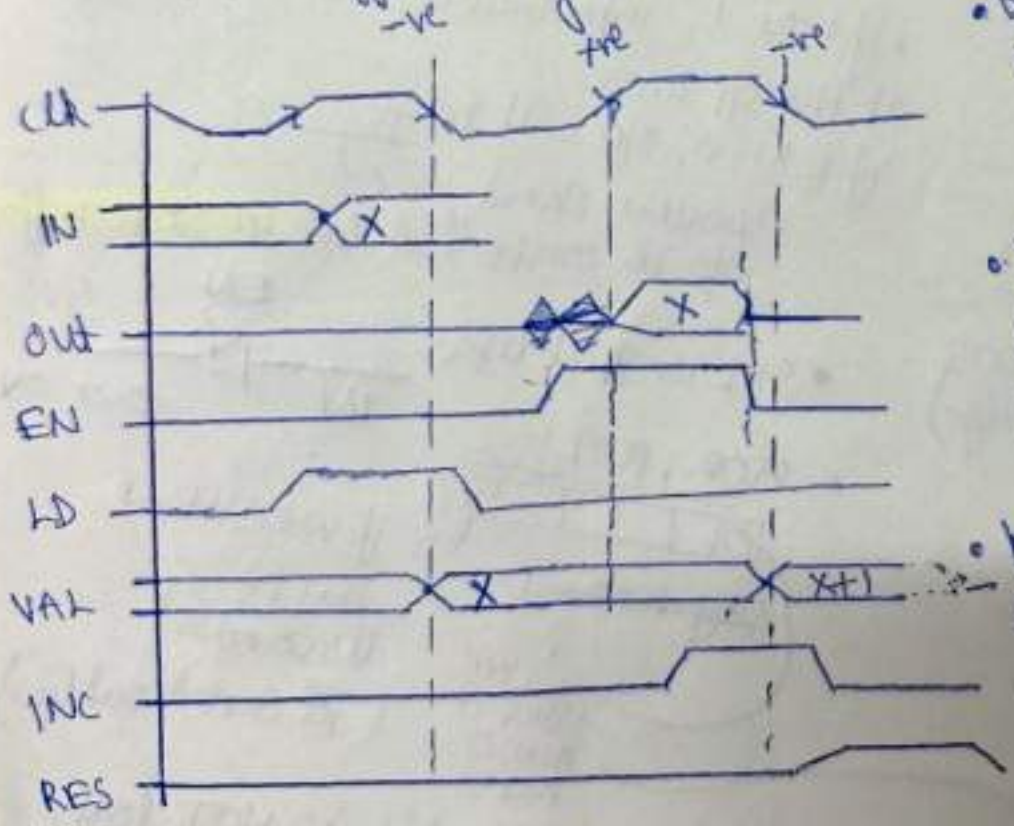
∴ SHL is used for multiplying by 2 in binary, that's why 0 defaults there.

- asynchronous resets are useful when we are powering on the register for the first time.
  ↳ the clk ~~ar~~ is still taking shape.

Timing Diagram

- depends on how we design the UR.
- * design U.R. s.t. all actions taking place (LD, INC, R.OP etc) happen at the -ve edge of the clock while EN (op en) puts data on the o/p bus at the positive edge of clk.

- this makes the ckt complex but saves time.
  ↳ let's us use both +ve edge and -ve edge more effectively



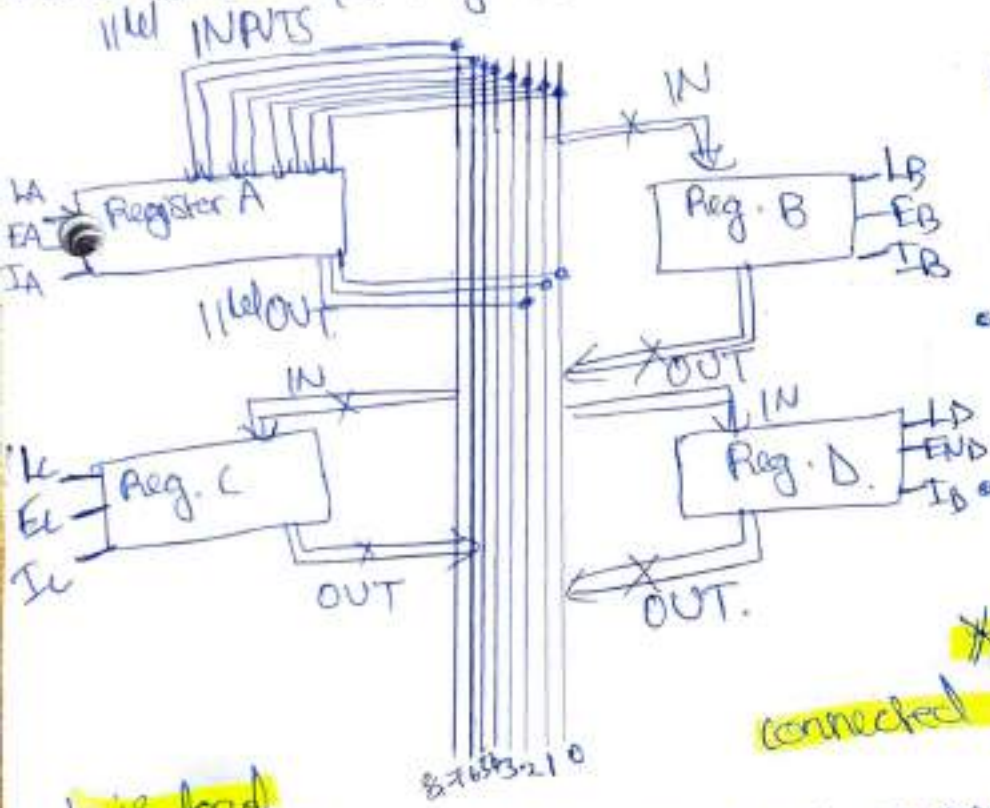
- Set LD to 1 bf -ve edge
  ↳ at -ve edge, data at IP line is stored as val in register.
- Set EN to 1 bf +ve edge
  ↳ at +ve edge, data from val is manifested as o/p.
- Set INC to 1 bf -ve edge
  ↳ at -ve edge, val is changed to X+1.
- Set RES to 1 bf next -ve edge
  ↳ at -ve edge, val is reset to 0

because EN and other control signals / functions are detatched, we can have EN and something else done, in one clk cycle.

- O/p is the value with a tri-state buffer (n o/p lines ⇒ n-tristate buffers but all with same EN?) Yes.

Creation of A Common Bus

- one common / set of common wires called a bus.



- all are 8/16/32 i/p o/p registers.
- all 8-8 wires to all ins/outs.
- 8-bit bus is shown.

* all registers are connected to a clk. &

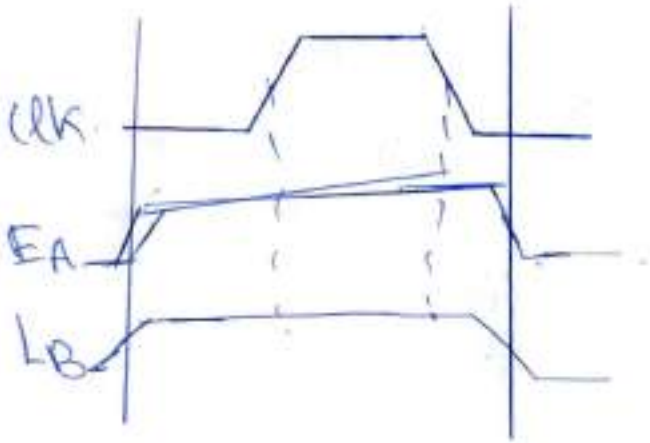
L is load
E is enable
I is increment
others may be that.

* Why can we connect things to a common bus? Without bus-contention / short ckt.

↳ Tri-state buffers, let us connect all of these to a common bus They have the ability to just float and not drive any o/p.

*↳ We need to ensure that only one reg. is enabled at a time - else ∃ bus contention and ckt explodes.

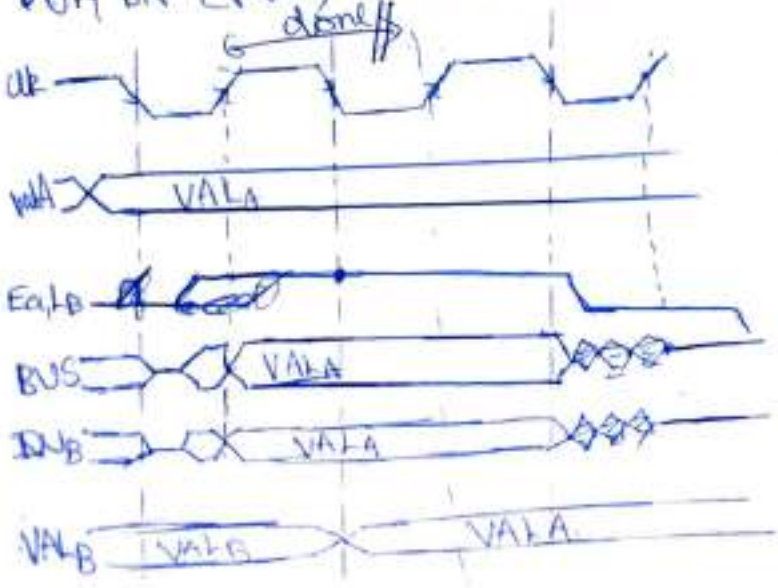
- How do we transfer information from A to B?
  - ↳ $E_B = E_C = E_D = 0$.
    $E_A = 1 \Rightarrow$ O/P from A is manifested on the entire bus and hence i/p of B C D also.
  - ↳ Set $L_B = 1$ to accept I/P into B register.
    done.
  - ↳ This can be done in a single clock cycle.



- Set E_A and L_B to high in the same clk cycle
- at +ve edge, E_A functions
  ⇒ A gets enabled
  ⇒ bus has value of reg. A
- at the -ve edge, L_B is high
  ∴ input present at B (value of A) gets accepted into B.

*HAVE

- ∴ we can transfer information in 1 clock cycle (from reg to reg)
- 6H bit CPU ⇒ 6H bit main external bus is what it means.
- activate E_A, L_B simultaneously.
- eg.
- we have moved info from A to B like assignment instruction
  B = A; // code
  MOV B A } machine/assembly programming
- all we did was $L_B = E_A = 1$, rest all was hard-wired



What if we activate L_B, E_D, L_C, Λ_A together?

↳ 

↳ D is manifested onto the bus and hence i/ps of all ABC, at ~~after~~ +ve edge

↳ at ~~after~~ -ve edge, inputs at ABC are internalised into the register.

∴ in one clk we did { $A=D_j$, $B=D_j$, $C=D_j$ } we did a simultaneous assignment

- What if we activate F_B, L_A, I_B ~~together~~? ? doubt.

↳ at +ve edge, B manifests onto Bus, and hence at inputs of ABCD too.

↳ at -ve edge, B at A's i/p moves into reg. A and is stored.

also $I_B=1 \Rightarrow$ ∴ B is incremented by 1, to B+1

↳ if E_B is still 1 at -ve edge als~~o~~ ~~stp~~

Bonly gets fed into A.

B+1 to load into bus takes time ∴ only B is fed,

by the time B+1 comes, clk is gone.

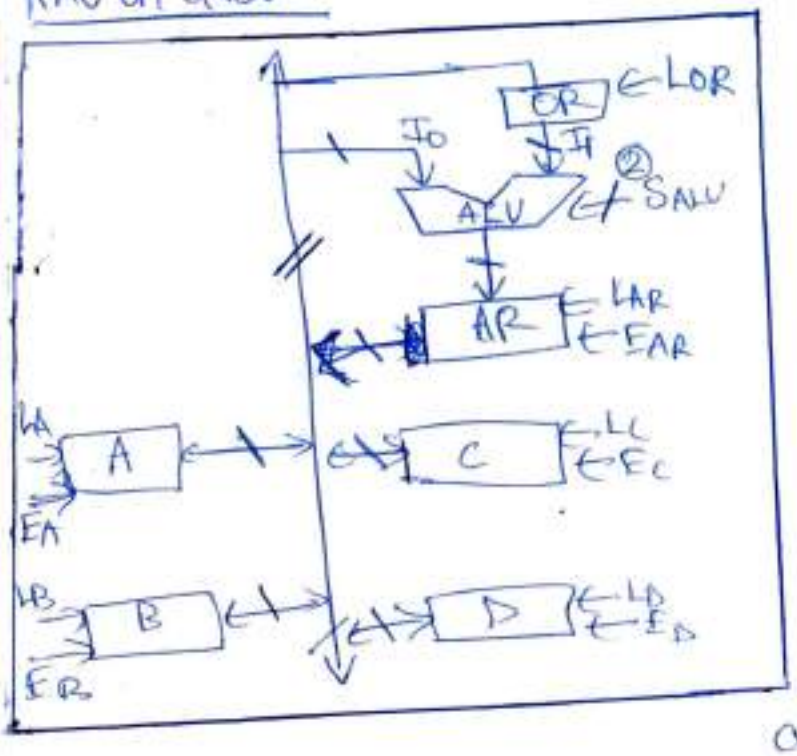
(↑ -ve edge)

* if ~~EN=1 at rising edge of a cycle~~ ~~it remains high for the rest of the~~

ALU Design

- ALU is a combinational circuit, that takes two inputs, and performs operations based on selects and gives one o/p
  - ↳ implemented using a combination of MUXes within the ALU.
- can do many things by connecting a few gates at the right places

ALU on a BUS

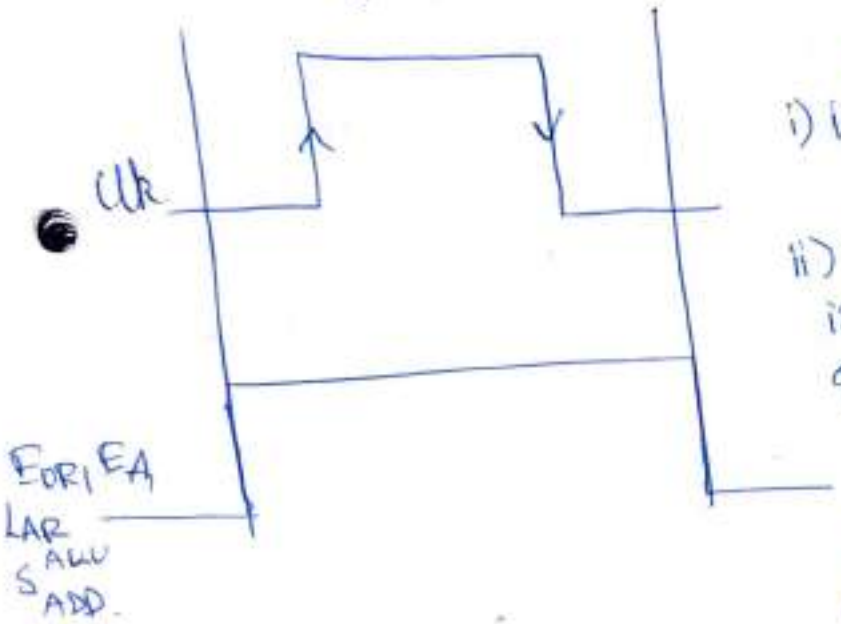


- OR is the operand reg. inputs of O.R. are connected to bus, o/ps are connected to ALU i/p
- o/p of ALU goes to Accumulator Register whose o/p goes to bus
  - ↳ sort of created a loop with ALU, AR, BUS.

- Why do we need AR and all?

Lecture 2

- the OR and AR registers are used just to make sure the ALU is able to interact with the bus properly.
- every register has all the control signals of our generic multipurpose register. (INC, DEC, SHL, SHR, ROR, ROL, IN, RESET)
- What happens if we activate $E_{OR}, E_A, L_{AR}, S_{ADD}^{ALU}$.



- at the +ve edge,
  i) info stored in A manifests onto bus and hence I0.
  ii) info stored in OR (operand) is manifested into I_1 input of ALU.

  ⇓

  ALU has generated add o/p and its at i/p of AR already before -ve edge.

- at the -ve edge, the o/p of A+OR, is internalised into AR reg.

∴ overall, we did AR = A+OR; in one clk cycle only!
↳ just because of the way wiring is made and +ve, -ve edging.

∴ we can do complex tasks in just one clk cycle.

to do meaningful things, we need to do things over multiple clocks.

• Consider this multiclock instruction:

→ clk 1 : E_A, L_{OR} ⇒ Whatever is in A is loaded into OR.

→ clk 2 : $E_B, E_{OR}, L_{AR}, S^{ALU}_{SUB}$ ⇒ B is at I_0, OR is at I_1, B−OR or OR−B is o/p from ALU and LAR feeds it into AR.
(remember OR is A from clk 1 instruction)
B−A is in AR.

⇒ clk 3 : E_{AR}, L_C ⇒ B−A from AR is loaded into C.

* notice that we have done C = B − A in 3 clks.

∴ we can take info from any 2 reg, operate them and store the o/p in any other reg. in 3 clk cycles!

• Say if we want to perform D = D AND C.

→ clk 1 : E_C, L_{OR}

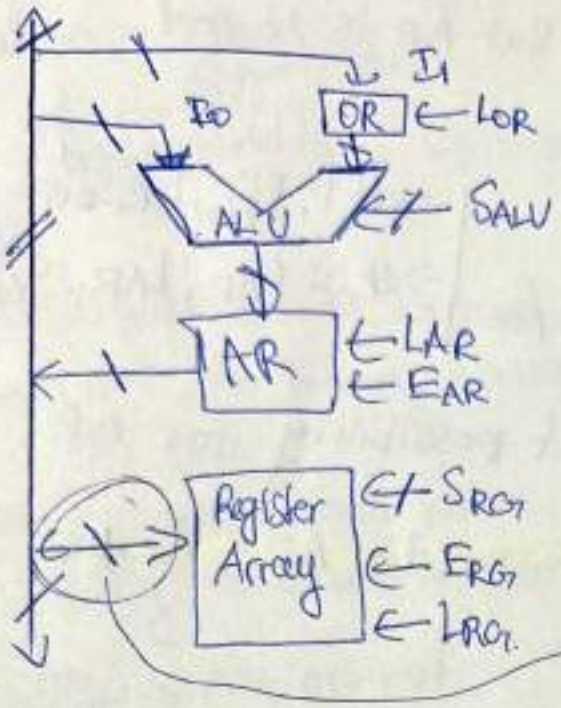
→ clk 2 : $E_D, E_{OR}, L_{AR}, S^{ALU}_{AND}$

→ clk 3 : E_{AR}, L_D.

• This is already a simple processor.
We ~~used~~ can do so many things by just controlling control signals for each clock.

• each clock cycle is called a micro instruction

Enhanced Single Bus Architecture



- same but register array.
- instead of having 2/3/4 registers connected separately combine them, use common enable and load controls and select lines to select one particular reg

connected to all Reg. but select magic

- info about the latest operation performed by the ALU and its o/p is stored in AR. This defines a lot of things in processor flow.
  - ↳ for, while terminations depend on this.

- Say we have 12 reg. {R0, R1, ..., R11}. We have 4 selects.

Adv. of Reg-Arraying	Disadv. of Reg. Arraying
• less E,L lines (common) drastically less (6).	• need E,L for each reg. Many ctrl signals (≥24) • need 2 clk cycles instead of 1 to cpy data from 1 reg to another • if all reg. were connected to the bus individually, in 1 clk cycle, we can cpy whatever is on the bus to all regs.

* Enhance further by adding more functions and more select lines to the ALU.

NONE, ADD, SUB, AND
OR, XOR, PASS left, ?

↳ whatever is at I0 is passed to O/p unchanged.

- ADD R_i : (AR = AR + RI) ⇒ this is an assembly level instruction (we have not specified which reg.'s info we add to R_i or to which reg. we store o/p. But AR is so cool.

→ ~~clk1: S^{01}_{RG}, E_{RG}~~

→ clk1: E_{AR}, L_{OR}

→ clk2: E_{RG}, S^{01}_{RG}, E_{OR}, S^{AND}_{ALU}, L_{AR}.

→ clk1: E_{RG}, L_{OR}, S^{01}_{RG}

→ clk2: E_{AR}, L_{AR}, S^{ADD}_{ALU}

- no E_{OR} ∵ there is no bus conflict possibility for OR.
- given any operation and R_i we can do AR = AR ✱ R_i in two clk cycles.

- SUB R_7: (AR = AR − R7)

→ clk1: E_{RG}, L_{OR}, S^{7}_{RG}

→ clk2: E_{AR}, S^{SUB}_{ALU}, L_{AR}

- XOR RII : (AR = AR XOR RII)

→ clk1: E_{RG}, L_{OR}, S^{II}_{RG}

→ clk2: E_{AR}, S^{XOR}_{ALU}, L_{AR}

E_{AR} — +ve edge; L_{AR} — −ve edge.

- can perform a lot of operations in just two clock cycles.
- we also need a way to load values from reg to AR and save results from AR to a different register.

- ~~load from R_i to AR~~:

clk1: E_{RG}, S^{i}_{RG}, $S^{passleft}_{ALU}$, L_{AR}.

* call this LOAD R_i (defined wrt AR always)

- ~~store from AR to R_i~~

clk1: E_{AR}, L_{RG}, S^{i}_{RG}

* call this STOR R_i

Instructions

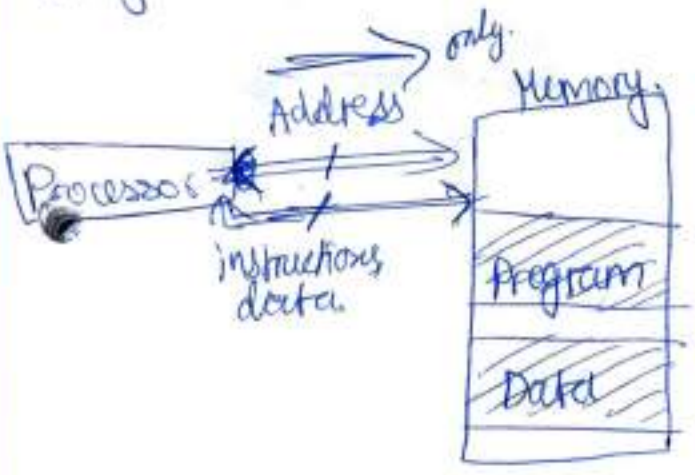
- At the lowest level, processors can only handle binary information.

C++ —GNU GCC→ assembly (load / store in RAM) } in binary only.

- all instructions to be carried out by a processor needs to be represented as binary strings.
- All basic instructions have to have a unique, unambiguous binary string associated with them.

binary words. → ADD R0 inst. → STORE R11 inst. } need to write binary strings to encode these instructions

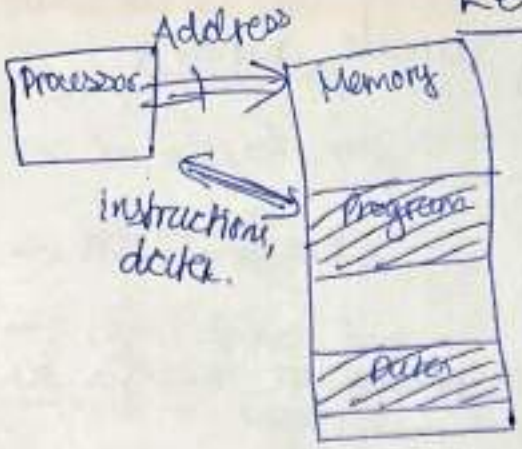
- the hardware must then decode the strings and carry out the corresponding instruction



- in the same memory, we have program and data - Von Neumann Arch.
- if prgm/data is stored in diff memory (Stored program?) Arch.

- asks for instruction, executes, then asks for next instruction
- each instruction is a binary string that encodes operations to be performed. (∞ fetch, execute loop)
- processor fetches instructions one-by-one, executes,
- meaningful operations occur as these instructions can read stored data in memory ....., and store data back into memory.

Lecture 3



- say we load up a pptx, in RAM. (file).
- Microsoft pptx 360 is a program, but the file is data.
- the program contains instructions on how to open the file, handle it etc.

- Program has a set of instructions in ~~assembly~~ binary. when in RAM.
- Given a binary string, processor needs to decode it and execute a particular action.
  2) binary codes should be unique, unambiguous, for the proc. to figure out what it should do.

- Machine language is basically binary. very tough to interpret by us
- Assembly is one level higher, use ~~pneumonics~~ mnemonic so that we can grasp them easily.

Designing An Instruction Set - ALU

- assume length of words is 8 bits to encode instructions.

8 bit len instructions in memory } expect processor to fetch-execute.

let all entities we will handle (instr. + data) be 8-bits.

- instruction set will have to correlate with what all the ALU is doing.
- we can come up with an arbitrary assembly to machine code mapping.

Assembly Instruction	Machine Code	Action
add <R>	10-1F	[AR] ← [AR]+[R].
sub <R>	20-2F	[AR] ← [AR]-[R].
xor <R>	30-3F	[AR] ← [AR] ⊕ [R]
and <R>	40-4F	[AR] ← [AR] ∧ [R]
or <R>	50-5F	[AR] ← [AR] ∨ [R]
comp <R>	60-6F	[AR] - [R]

- 10 → add R_0
  11 → add R_1
  1F → add $R_F = R_{15}$
  | 20 = sub R_0
  2F = sub R_F
  | ∴ first dig reps. op. second dig. reps. reg.

- add $R_5 \equiv \underline{15}_{16} \Rightarrow$ "00010101" means add R_5.
  this is in hexadecimal.
  An 8 bit binary no. needs only 2 bits in hexadecimal
  ∴ all these 2 bit codes represent 8 bit binary strings.

- sub $R_{10} \equiv 2A \equiv$ "00101010".

"0x" ≡ "Hexa"
"0b" ≡ "bina"

- 0x27 stands for SubR7 ∴ Processor is able to unambiguously interpret this machine code ↔ instr. mapping.

- cmp: ∃ a flag reg., with flag bits, like overflow bit, carry bit, compare bit,
  The flag bits are used in checking conditionals, branching etc. loops etc.

- till now add Ri 's instruction has Ri as a variable
- we should have some instructions in which the operand is specified in the instruction as a constant-

  eg. add IF → AR ← [AR] + [IF]

  add 0xXX ⇒ add XX to AR and store result in AR.

- if we wanted to make an instruction set for this, what problem would we encounter?

  ↳ add (IF) → 8 bit long number,

  instruction code is of length 8-bits only, how can we pass the instruction and the constant?

  ↓

  break it down into 2:

addi = add immediate

↳ can have an 8-bit string to encode this instruction.

↳ right after addi in memory is the constant we want to add to AR.

- these are the instructions used to embed constants into the program / memory?
- we can do movi (move immediate) and store the const in some reg. etc.

Assembly Instruction	Machine Code	Action
adi XX	01	[AR] ← [AR] + XX
sbi XX	02	[AR] ← [AR] − XX
xri XX	03	[AR] ← [AR] ⊕ XX
ani XX	04	[AR] ← [AR] · XX
ori XX	05	[AR] ← [AR] + XX
cmi XX	06	

→ once processor encounters "0000 0001" in memory, it will realise that the next word contains the operand ∴ go back to memory, fetch the next word, add it to AR and store result in AR.

Instruction Set - Data Movement

- Want to move data from AR to Reg. array, Reg. array to AR, memory to AR, AR to memory, memory to reg. array.
  - ↳ want an instruction set defined for this too.
- movs instruction moves information from a register to accumulator.
  movd moves info from ~~accu~~ AR to reg.
- movi moves the immediate word in memory (constant) into a register.
- load and stor instructions: taking value from reg ↔ memory.
- memory resides outside the processor.
- to access memory, we need to give an address to indicate which of its words is to be accessed.
  - ↳ here 8-bit address is the best for us (and each word is also 8 bits long)
  - ⇒ we can have 256 bytes of memory

- the contents of the corresponding memory word will be given to the processor on a read.
- The processor has to supply the content to be written to memory for a write
- Assembly Instruction | Machine Code | Action

movs <R>	70-7F	[AR] ← [<R>]
movd <R>	80-8F	[<R>] ← [AR]
movi <R> XX	90-9F	[<R>] ← XX.
stor <R>	A0-AF	[[AR]] ← [<R>].
load <R>	B0-BF	[<R>] ← [[AR]]

- while doing addi, we never defined a new variable, ∴ no range for M.C
but for movi, we define a variable R.

[AR] ⇒ value of AR

[[AR]] ⇒ value of the value of AR

⇓

whatever is in AR can be used as address. (∵ 8 bit).

- stor <R> ! [[AR]] ← [<R>].

⇓

whatever is the value of "[AR]", use this as an address to memory and store the contents of [<R>] at this location in memory.

- load <R> [<R>] ← [[AR]]

⇓

whatever information is in the memory address [AR] take it and store it in <R>.

- stori <R> / loadi <R> ???

The Fetch-Execute Cycle

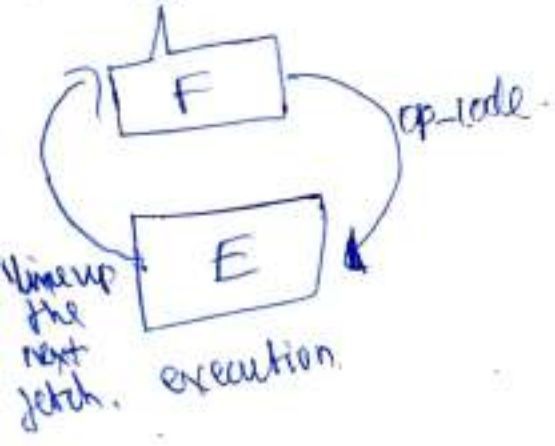
- how does an instruction get fetched and executed?
- The processor works autonomously as a continuous fetch-and-execute engine, with no other input than an external clk.

→ each word is an individual instruction.
→ fetch this, then execute.
→ then fetch this, then execute ...
forever loop.

∴ the processor should be able to continuously fetch and execute. We will just give it power and a clk.

- the machine code is in the memory outside the processor so the instructions have to be brought to the processor one-by-one and executed.
- the instruction at the $(i+1)^{th}$ address has to be fetched and executed after instruction i, ∵ instructions of a program are stored consecutively in memory.
- the processor has to do all this by itself

* Every execution lays groundwork for the next fetch

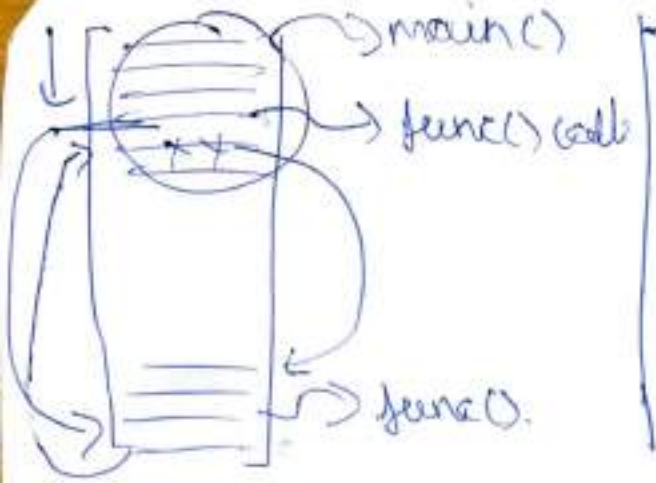


- Processors have a special kind of register inside the that manages the process of instruction fetching, by keeping track of the address of the next instruction to be fetched at all times.
  - ↳ This reg. is called the program counter PC.
    - ↳ :: Stores the next address (8 bit) PC is an 8 bit reg.

information about 05 is stored in PC then we increment PC and it stores the next address.

- Processing of an instruction begins with fetching the opcode from the memory, whose address is in the P.C.
  - ↳ fetching contents from the address that is stored in the P.C
  - ↳ contents of PC are incremented while the execution is going on so that we are ready with the address for the next fetch cycle
  - ↳ opcode is brought to the processor and appropriate action is performed in the execution phase.
  - ↳ once this is completed, the next instruction is obtained from the memory.
- The PC does a lot more than just storing incremental numbers:

PC also keeps track of the program's control flow.



→ the processor jumps from executing the main program to executing the function once called.

↳ can do this by arbitrarily changing PC to contain the functions address in memory.

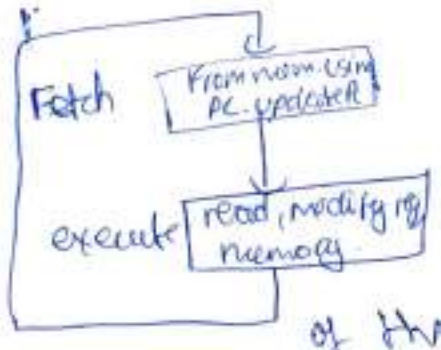
This jump instruction may work by having the "func()'s address as the next word, so we can put this word in the PC.

Then we continue fetching.

↳ we will start executions related to the function, increment PC ... etc.

↳ When function is finished, it will have one more jump to go back to main.

- function names are associated to addresses.
  ↳ ∴ everytime we call a fn the same address gets called each time, same address goes into the PC, the same function gets executed.
- How will this whole process start?
  - once an instruction is done with, the next is taken up by incrementing the program counter.
  - how do we start this at the beginning?
  - need to know where the starting point of that program is.

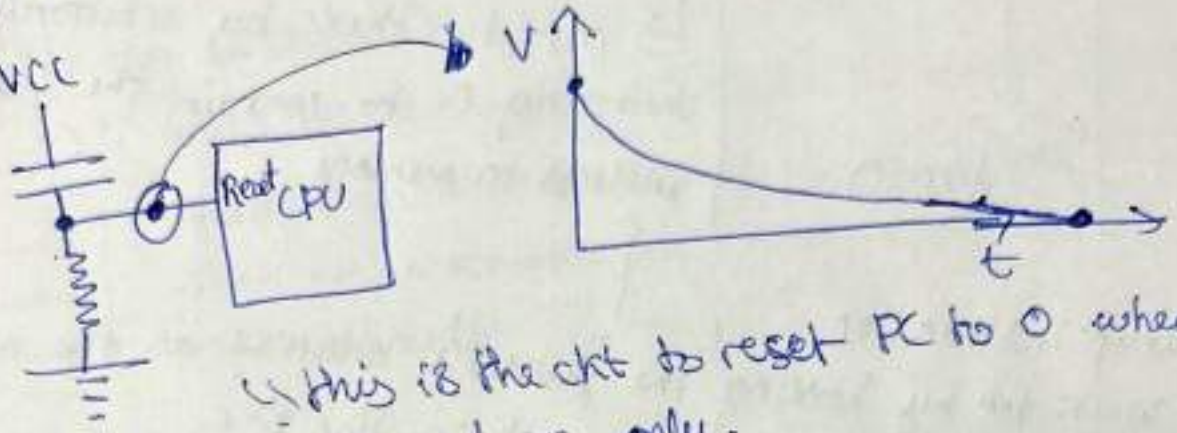


* how does the very first program start?

- how does BIOS get control?

↳ asynchronous clear i/p for P.C.

Whenever PC is reset to 0000 0000, we start the fetch cycle.



↳ this is the ckt to reset PC to 0 when the computer is switched on. only.

- the very first program that gets control is the one saved at memory address 0. (Firmware) or (BIOS) → OS

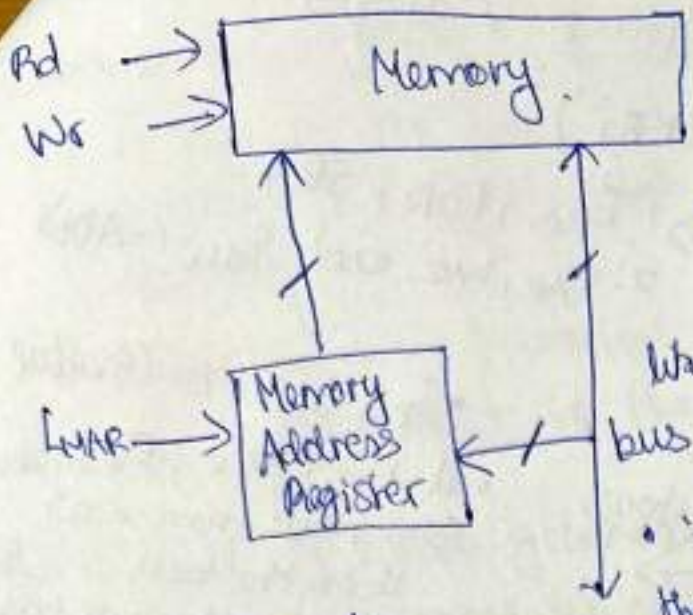
BIOS → OS

↳ any corruption in BIOS/FIRMWARE is detrimental to booting.

Memory Access

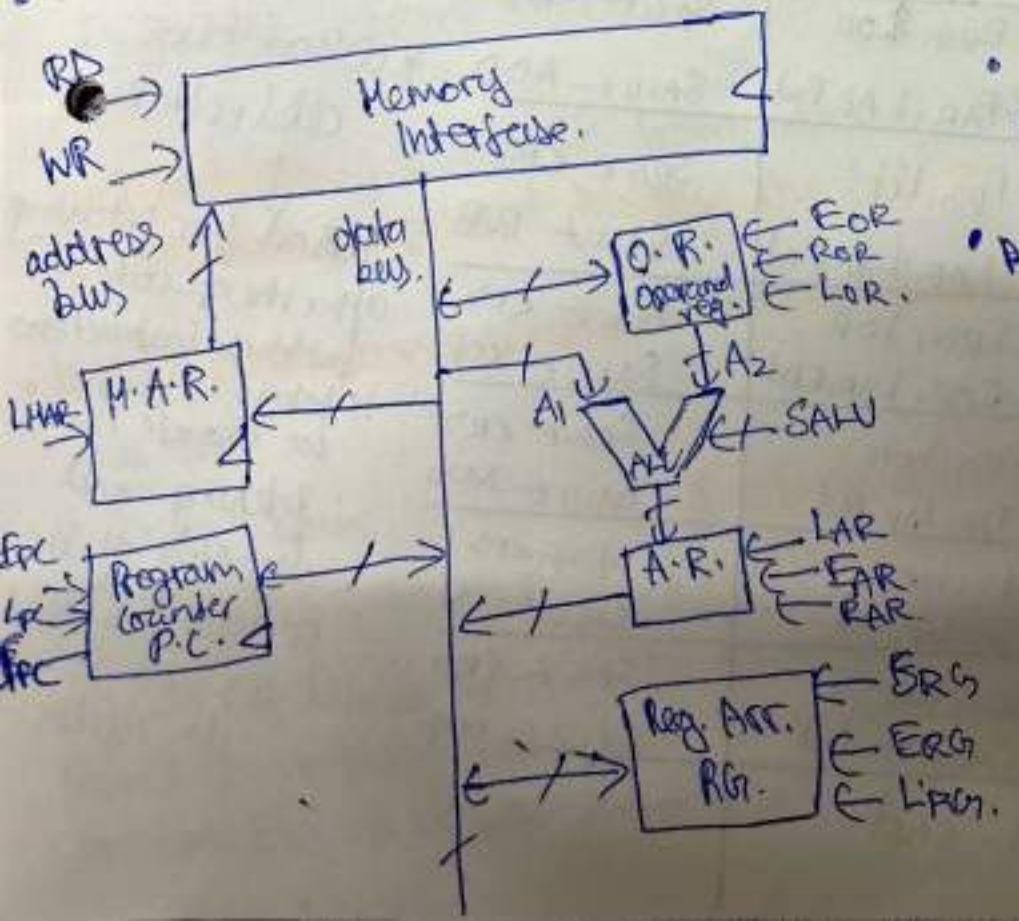
- how does the processor access memory?
- use a memory interface that supplies addresses to the memory along with signals to indicate read/write.
- need both i/p and o/p data bus for read/write to memory and separate connection for addresses.

1) provide address to memory
2) control signals to read/write from this address
3) have a data bus so that it can carry out data from memory while reading and send in memory while writing.



- Whatever i/o we get from the memory is directly placed on the data bus.
- MAR stores the address we are currently working with.
  2) Load addresses from data bus like we load everything else.
- bus (data bus / arterial bus)
- ∵ MAR is outputting to Memory, there is no real need for En.

- Rd, Wr are controls.
- If we activate Rd, the word at the location address in memory is read and manifests on data bus
- If we activate Wr, whatever is on the data bus will get written to the address provided by MAR.
- Memory is similar to the register array, but it is larger and outside the ~~memory~~ processor.
- The address should be supplied b4 rd/wr operation.



- added 3 more components.
- PC → next instruction's address
  MAR → current address to instr.
- ~~PC stores the~~

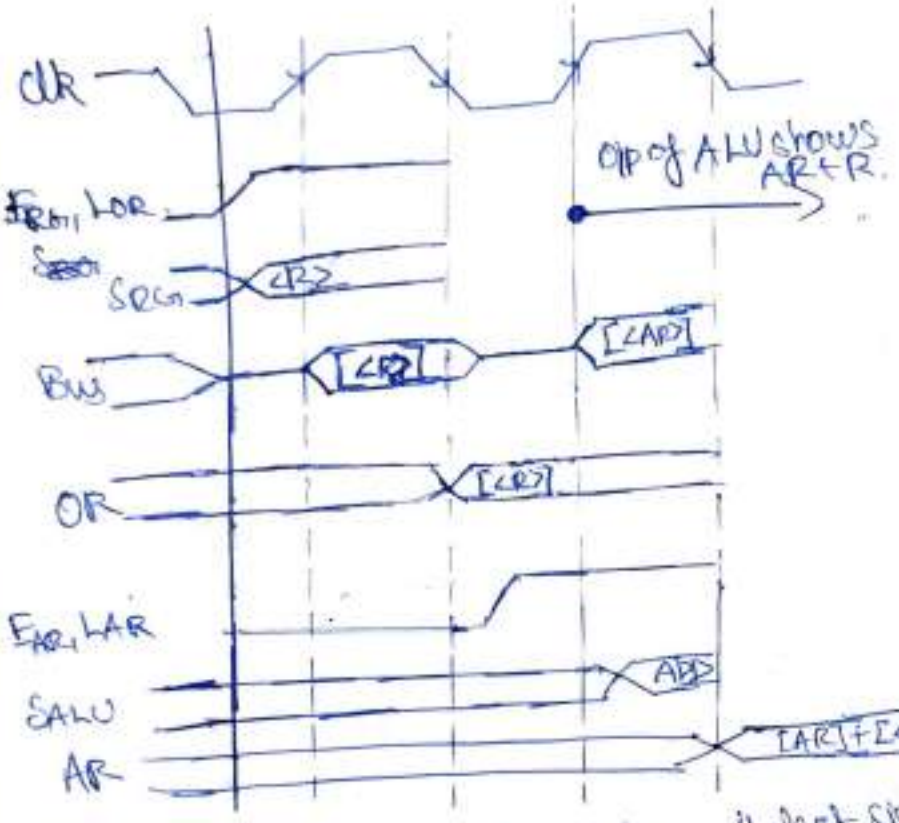
Lecture 5

Implementing Instructions - ALU (from prev. diagram).

- consider ADD <R> ⇒ (AR = AR + Rj)

i) enable RGi, load OR
ii) enable AR, SALU ← ADD, LAR

⇒ 1: ERGi, LOR, SRGi
2: EAR, LAR, end, SALU ← ADD



- SRGi some 0-16 value
- at the edge, ERGi, SRGi, some value from <R> is on the bus right after rve edge.
- ALU is always giving garbage values

Instruction	control signals	select signals.
add <R>	clk3: ERGi, LOR clk4: EAR, LAR, End	SRG ← <R> SALU ← ADD
SUB <R>	clk3: ERGi, LOR clk4: EAR, LAR, End	SRG ← <R> SALU ← SUB
XOR <R>	clk3: ERGi, LOR clk4: EAR, LAR, End	SRG ← <R> SALU ← XOR
AND <R>	clk3: ERGi, LOR clk4: EAR, LAR, End	SRGi ← <R> SALU ← AND
OR <R>	clk3: ERGi, LOR clk4: EAR, LAR, end	SRG ← <R> SALU ← OR
cmp <R>	clk3: ERGi, LOR clk4: EAR, LAR, end	SRGi ← <R> SALU ← CMP
nop	clk3: end	-

* nop: no op.

* What are clk1, clk2?
↓
used for fetching. after the op-code for these instructions is fetched, then we execute it.
∴ fetching and decoding of the op-code is done in first 2 clk cycles

addition requires 4 clk cycles. (2 fetch, 2 exec).

- what is 'End'? Why is it everywhere?
  - ↳ it signals the end of execution of current instruction.
  - ↳ it tells the processor that it's time to fetch the next op-code.
  - ↳ end is an actual control signal that goes to 1 for clk H. but where is the end control signal going to?
- the number of microcycles needed for diff. machine instruction depends on the processor architecture - some may be hardwired to do things quicker.

Implementing Instructions - Data Movement

- movd: moving from AR to a reg.
  EBAR, LRG SRG ← <R> } needs only 1 clk cycle.
- movs: moving from <R> to AR.
  ERG, LAR, SRG ← <R>, SALU ← pass 0 } needs only 1 clk cycle.
- load and store; read/write from memory into a reg; from the address that is pointed to by the AR.
- load; reads a value from memory to a reg, (address supplied by AR)
  ∴ EAR, LMAR.
  RD, LRG, SRG ← <R> } *RD is executed at +ve edge
  *WR is executed at -ve edge.
- in 2 clk cycles, we took info about the address from AR, stored it in MAR, read from this address in memory and loaded the o/p value into RG.

• Stor <R>: take value in <R>, store it in memory at the address pointed to by AR.

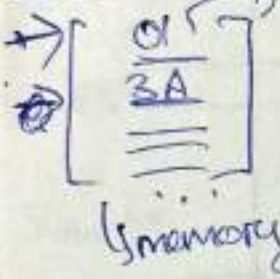
EAR, LMR
ERG, WR SRG ← <R>

*(WR happens at -ve edge).
first clk cycle is same for load, stor.

• Memory interfaces are slow; takes a few clks to read/write.
↳ address multiplexing } takes many clks.
↳ coincident decoding }

• In our architecture, data movement involving memory interface is 2 clks; involving AR, RGs only is 1 clk.

Addi



(memory

op-code for adi; when we fetch this, we realise we need to go back to memory and read and add to AR and store it in AR.
How do we do this?

(PC is pointing to the address in memory where 3A is stored) after fetch-execute.

* When we fetch, we fetch whatever is there in memory at the address pointed to by the P.C.

i)ii) EPC, LMAR and in the next clk cycle read from memory.
While fetching we also IPC (to make sure we are ready for next instruction)

∴ just after fetching 01, PC is already pointing to the address of 3A in memory. (even before execution of 01 this happens)

1) EPC, LMR (if 3A is in OR, then just add) } takes 3 clk cycles to add; (after xxxxx word is obtained).
2) AD, LOR
3) EAR, LAR, SALU ← ADD
* make sure PC is incremented during execution too.

ori <R>

EPC, LMR, IPC (happens during fetch)
PC is pointing to i word.
∴ EPC, LMR, IPC ⇝ to go to next op-code.
RD, LRG, SRG ← <R>.

- for all i's, get the data from memory and store it in DR.
- to do repeated operations using ALU, it's better to store the data first in RG. (saves time, memory slow).

Instruction	Control Signals	Select Signals
movs <R>	CK3: ERG, LHR END	SRG ← <R>, SALU ← pass 0.
movd <R>	CK3: EAR, LRG, End	SRG ← <R>.
load <R>	CK3: EAR, LMR CK4: RD, LRG, End	- SRG ← <R>
stor <R>	CK3: EAR, LMR CK4: ERG, WR, End	- SRG ← <R>.
movi <R> XX	CK3: EPC, LMR, IPC CK4: RD, LRG, End	- SRG ← <R>
adi XX	CK3: EPC, LMR, IPC CK4: RD, LOR CK5: EAR, LAR, End.	- - SALU ← add
sbi XX	CK3: EPC, LMR, IPC CK4: RD, LOR CK5: EAR, LAR, end	- - SALU ← sub
ani XX	CK3: EPC, LMR, IPC CK4: RD, LOR CK5: EAR, LAR, end	- - SALU ← AND
ori XX	CK3: EPC, LMR, IPC CK4: RD, LOR CK5: EAR, LAR, end	- - SALU ← OR
cmi XX	CK3: EPC, LMR, IPC CK4: RD, LOR CK5: EAR, end	- - SALU ← CMP.

Lecture 6

- so far, we had no branching instructions; all programs had to be a strict linear sequence of instructions.
- we need the capability to branch/break sequential flow to implement loops.
- we also need capabilities to branch based on some condition based on the values of registers.
- There are two forms of branching: one based on an immediate address and the other based on a register value (flag reg)
- branching involves the shifting of program execution from one point to another
  - this is a change in control flow of the program and may be used to perform different actions based on the results obtained so far.

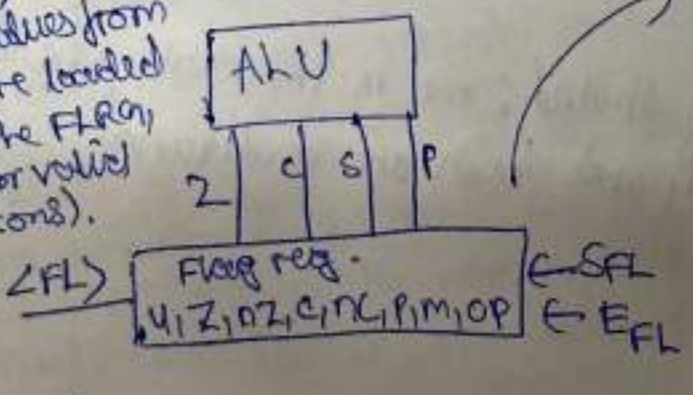
- Jump Instructions: a type of branching where control flow is transferred absolutely, without any memory of the branching point.

The Flag Register

- branching may be conditional/based on a current state of the processor.
- ∃ a flag register in the processor to indicate particular cond.
- The condition of one of the flag reg. bits can be used for branching.
  - e.g. if jump is conditioned on a certain flag bit being set, the branching happens if the bit=1 and the program proceeds with the instruction at the branch address
    If flag bit=0, execution proceeds normally with next instr. as if the conditional branch instr. is a nop instr.

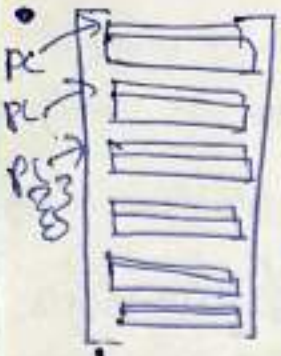
- Flag reg. may include flag bits which answer q's like
  - did the last arithmetic op result in overflow?
  - did it result in a carry from MSB?
  - was the result of a prev. op zero?
  - etc
- Say we want to loop n times some set of computations:
  - initialise count (stored in flag reg.) to n.
  - after one set of computations is over, reduce count by 1
  - the program can branch to the start of the computations if the count is still not zero.
- Our processor will have 4 flag bits: Zero, Carry, Sign, Parity.
  - zero flag is set if the previous ALU op resulted in 0.
  - carry flag is set if the prev. ALU op resulted in a carry out or borrow in from MSB.
  - S bit copies the sign bit of the last arithmetic operation, (1 if negative).
  - parity flag counts the number of bits = 1 in the result of the last operation. if this no. is odd then parity is 1.
- Instructions which do not use ALU (data movement, ~~instr.~~ branching) do not change flag values.

(flag values from ALU are loaded into the FLAG, only for valid operations).



The flag register has 8 bits:
Z, C, S, OP are H,
U is 1,
nZ, nC, M are 3. (complements)

Two Jump Instructions



① jumpd <FL> XX:

clk1: E_{PC}, L_{MR}, I_{PC}
clk2: RD, ~~end~~ L_{IR}, end ⇒ find out it's a jumpd <FL>XX

- This instruction makes the program jump to address XX if <FL> is true,

clk3: E_{PC}, L_{MR}, I_{PC}, End if <FL>', E_{FL} ; S_{FL} <FL>.
clk4: RD, L_{PC}, End.

→ if <FL> ≠ 1, then this execution ends and we fetch next.
if <FL> = 1, it goes to clk H, and puts XX in PC.
The next fetch is from mem. location XX

② jumpr <FL>:

clk1: E_{PC}, L_{MR}, I_{PC}
clk2: RD, L_{IR}, ~~end~~
clk3: E_{PC}, L_{MR}, I_{PC}, E_{FL}, end if <FL>
clk4: RD, L_{PC}, end.

→ fetch turns out to be jumpr <FL>.

- This instruction makes the program jump to address [AR] if <FL> is true.

Instruction Fetch

clk1: E_{PC}, L_{MR}, I_{PC}
clk2: RD, L_{IR}.

- The word at memory location [PC] is read and loaded into the instruction register.

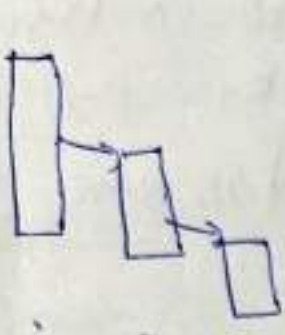
- The instructions register holds the 8-bit op-code which needs to be decoded to select an action.
- Every instruction needs two clock cycles 100% just to fetch and decode the action.

* The two types of branching studied; one is immediate address and other is register value based, and both are conditional (use <FL>)

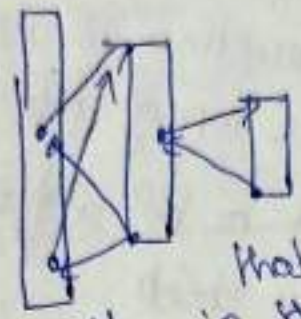
and Return Instructions

jumping is one kind of branching - it provides no opportunity to return back to the same address

- calls are another kind of branching with the provision to return to the same place.



jump.



call.

- call kind of branching is used to invoke functions in code.
- a function is a block of code that can be invoked from any place in the program.

- The called function performs its operations and when done, the program proceeds from where the function was invoked.
  - ∴ When branching to functions, we need to remember the point of branching so that it can return there at the end of func..
- call instruction invokes the function.
  return instruction brings control back to where the call happened.
  - these instructions can be conditional too.

* calls can be nested - each call has to just return to its right place

- To support nested calls, a sequence of return addresses ~~values~~ needs to be remembered.
  - The last remembered return address enables the return from the most recent call operation.

∴ Return addresses will be used in reverse order from which they are saved.
- ∴ we stack up the return addresses on a single pile with the latest one placed on top.

- Removals are performed one-at a time and from the top of the pile.

* Such a structure is called a Stack.

∴ a call instruction remembers the point to return to by placing the address of the next instruction in the stack.

↳ The address of the next instruction is available in PC, thus PC has to be added to the top of the stack.

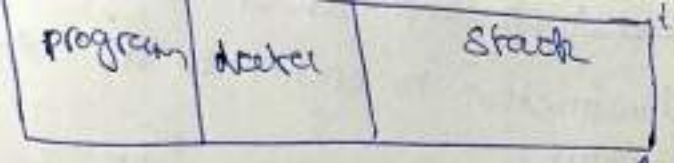
↳ The top of the stack needs to be adjusted as more additions may happen to the stack.

- The address to branch to, is given by the call instr.
- The value on top of the stack is used as the address from which the next instruction is fetched when returning.
  ↳ ∴ this value is loaded into PC
  ↳ The top of the stack needs to be adjusted to reflect this.
- Adding a new element to the stack is called a "push"
  Removing an element from the stack is called a "pop"

Stack Manipulation Instructions

- The "system stack" is stored in memory (RAM) so that it can have large capacity.
- Inside the processor, ∃ a special register called the Stack Pointer.
  ↳ this holds the address of the next empty location in the system stack.

* SP's value is decremented by 1 when a value is pused onto it and incremented by 1 when it is popped out of it. ∴ SP is initialised to highest address of the stack.



SP → initialised to highest address that belongs to the stack.

- turns out: (for us), the S.P. is always locating the topmost value in the stack. (implement push/pop accordingly).

Implementing Branching and Stack Instructions

① push <R> : store the value in <R> into the ~~top~~ of the stack pile.

clk3: DSP (∵ SP is pointing to the top word, due to fixed empty space for new val).

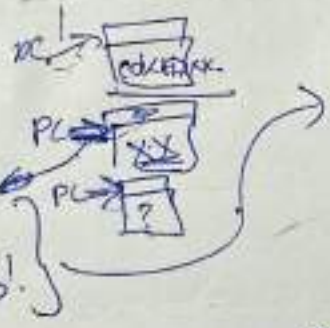
clk4: E_{SP}, LMR ; (load the add of empty space)

clk5: E_{REG}, WR, end ; REG ∈ <R> ⇒ put this val.

∵ decrement takes place at the falling edge, we need to use a full micro~~instruction~~cycle for this.

↳ can fix by designing SP reg. in a way that it decrements during +ve edge and increments for -ve edge. (life for now).

② cd <FL> XX



clk3: E_{PC}, LMR, IPC, E_{FL}, end if <FL>.

Loads PC into MR, incs PC, checks for interrupt. If OK branch else continue next fetch. ∴ PC at next loc already.

clk4: R_D, L_{OR}, D_{SP} ⇒ XX is in OR. SP is pointing to empty position at the top of the pile (has its mem. loc. rather).

clk5: E_{SP}, LMR.

⇓

Store this vacancy (in stack) address in MR

clk6: E_{PC}, WR ⇒ at this vacancy in stack, it writes the location of ?.

clk7: E_{OR}, I_{PC}, end ⇒ stores "XX" val, which is prolly some address to a func. block, into PC. this is a call!!

③ pop <R>: load the SP's address into <R> and inc SP:

ck3: ESP, LMR, ISP

ckH: RD, LRG, end SRG ← <R>.

④ ret <FL>: put SP's address-val into PC so that we return for next fetch.

ck3: ~~ESP, LMR, ISP~~ EFL, end if <FL>' ; SFL ← <FL>.

ckH: ESP, LMR, ISP

ck5: RD, LPC, end.

- ~~instructions for branching and stack manipulation~~

⑤ CT<FL>. : call instruction to branch to ~~the~~ an address, loading it as [AR]'s value.

ck3: EFL, end if <FL>' ; SFL ← <FL>.

after fetch. (return here only)

ck4: Dsp → SP showing top of pile

ck5: Esp, LMR → store this addr. in MAR.

ck6: Epc, WR → write address of PC ~~this~~ stack.

ck7: EAR, Lpc, end. → replace pc with [AR] : . next fetch can start the function

B//.