Logic:-

* Convert 625_{10} to $(\quad)_2$.

$625 = 512 + 113.$

$= 2^9 + 64 + 49$

$2^9 + 2^6 + 2^5 + 2^4 + 2^0$

1 0 0 0 1 1 0 0 0 1

→ Depecting signed magnitudes in binary:-

1. using MSB for sign
   - Sign, mag bits should be treated differently
   - Overflow is difficult to detect.
   - 0 has two notations.
   - Add^n & Subtract^n need different circuits.

- Single bit adder:-

a, b ⟶ S, C.

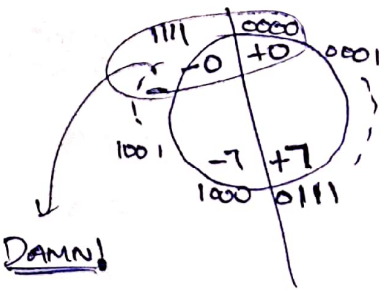
Full adder:-

a, b, cin ⟶ S, C.

Single bit Subtracter:- ✓

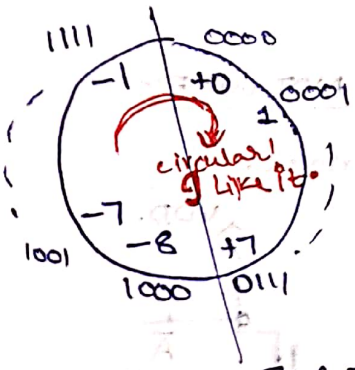
full subtracter:- ✓

2. 1's complement:-



3. 2's complement:-

to get 2's complement:
- > add 1 to 1's complement.
- > 2^n − (the number).



$5 - 7 = 5 + (-7)$

$= 5 + (2^n - 7)$

add the ~~decimal~~ binary rep;

even for subtraction.

Kool!.

Logical Function:-

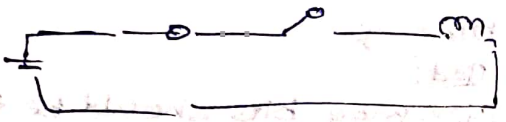
n inputs $\longrightarrow 2^n$ domain elements

2 co-domain elements

$\longrightarrow 2^{2^n}$ possible n-input functions overall.

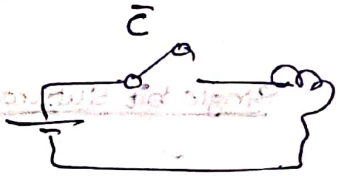
Implementation:-

1) using Switches:-



AND.

OR



NOT: we use switches, such that

logic 1 is open

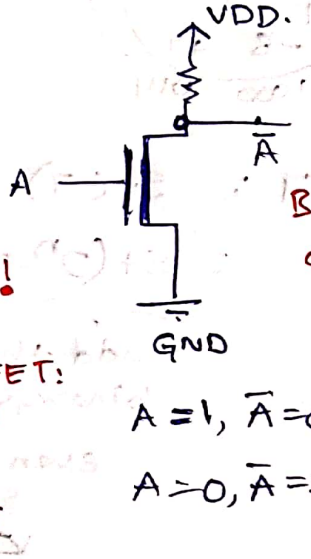
0 is closed.

Switch technology has improved a lot:-

- relays
- vaccum tubes
- Bipolar transistors
- FETs (1970-)
- ICs. (1980-)

* **MOSFET NOT GATE:-**

As bool variable.



But! when A=1 current leakage causes power dissipation

Solution:- complementary MOS design proposed by

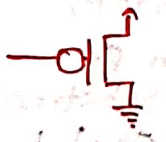
Paper:-

" Nanowatt logic using FET".

Feb 20, 1963

N MOSFET!

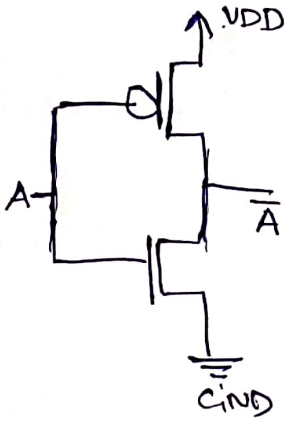
in PMOSFET:



A = 1, $\bar{A}$ = 0

A = 0, $\bar{A}$ = 1

Modern NOT Gate.



NANOWATT Logic.

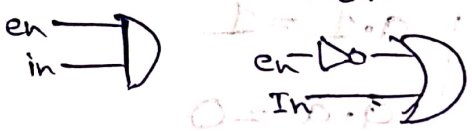
1) enabling function:-

when en=0; constant output.

0 1

use AND

use OR & NOT.



2) decoders:-

output are minterms.

3) encoders:-

$A_3 = D_8 + D_9$

$A_2 = D_4 + D_5 + D_6 + D_7$

outputs

3) Selectors / mux / multiplexers.

n selectors, 2^n inputs, 1 output.

or less.

Built using

decoders + enablers + OR gate

at end.

Optimisation:-

Area : # switches

Performance: (Delay) # switches in series

Power : # Switches.

Testability : interconnect network.
security
intelligence

→ The duality principle:-

$$\text{Dual}\left(f(x_1, x_2 \cdots x_n)\right) = \overline{f(\bar{x}_1, \bar{x}_2 \cdots \bar{x}_n)}$$

* every equation ( ) = ( )
also its dual holds.

$a+0=a$ | $a \cdot 1 = 1$

$a+\bar{a}=1$ | $a \cdot \bar{a} = 0$

$A+BC = (A+B)(A+C)$

seems not visible;

but ... its simple distribution.

Theorems:-

1. Idempotency / invariant $a+a=a$, $a \cdot a = a$.
2. Null elements exist: $a+1=1$, $a \cdot 0 = 0$.
3. Involution $\bar{\bar{a}} = a$.
4. Absorption:-

$a+ab = a$

$a \cdot (a+b) = a$!

5. Adsorption:-

$a+\bar{a}b = a+b$

$a \cdot (\bar{a}+b) = ab$.

6. uniting:-

$a+\bar{a}=1$ $\quad ab+\bar{a}b = b$

$a \cdot \bar{a} = 0$ $\quad (a+b) \cdot (a+\bar{b}) = a$

7. demorgans:-

$\overline{a+b} = \bar{a}\cdot\bar{b}$

$\overline{ab} = \bar{a}+\bar{b}$

↓

Generalisation:-

$\wedge \longrightarrow \vee$

$\vee \longrightarrow \wedge$

$a \longrightarrow \bar{a}$

$0 \longleftrightarrow 1.$

$F = \bar{x}\cdot y\cdot\bar{z} + x\bar{y}\bar{z}$

$\bar{F} = (x+\bar{y}+z)(\bar{x}+y+z)$

8) consensus:-

$ab+\bar{a}c+bc = ab+\bar{a}c$ like resolution.

$(a+b)\cdot(\bar{a}+c)\cdot(b+c) = (a+b)\cdot(\bar{a}+c)$

OH Lord! what is this!

universal operators:-

NAND, NOR.

SOP, POS → ¬(F)

↓

write in NAND, NOR. ✓.

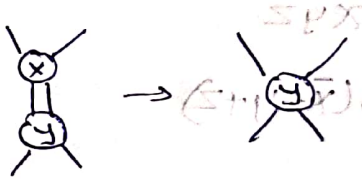
Canonical form:-

- useful to put in a form, that allows for comparision for equality.

  - Truthtable
  - Sum of Minterms
  - Product of Max terms
  - BDD
  - Reed muller expansion.

ROBDD reduction rules:-

1) merge equivalent leaves
2) merge isomorphic nodes
3) Eliminate redundant tests



* Shanon's expansion.

$$f(x_0, x_1, \cdots x_n) = x_0 \cdot f_{x_0} + \bar{x_0} f_{\bar{x_0}}$$
$$= x_0 f_{x_0} \oplus \bar{x_0} f_{\bar{x_0}}$$

* $f_1 \, OP \, f_2$

(if roots are same) $= \bar{x}\,(f_1|_{x=0} \, OP \, f_2|_{x=0}) + x\,(f_1|_{x=1} \, OP \, f_2|_{x=1})$

if $root_1 \neq root_2$ $= \bar{x_1}\,(f_1|_{x_1=0} \, OP \, f_2) + x_1\,(f_1|_{x_1=1} \, OP \, f_2)$

* +ve davio decomp

$$= f_{\bar{x}} \oplus x(f_x \oplus f_{\bar{x}})$$

- we davio;

$$= f_x \oplus \bar{x}(f_x \oplus f_{\bar{x}})$$

dont distribute $\oplus$ over anything ! wrong?

- And invert graphs.
- Checking Circuits
- DPLL.

special functions:-

1) Dual of a function:-

> exchange $\wedge$ & $\vee$.

> exchange 0 & 1.

Note that $dual(F(x_1, x_2 \cdots x_n)) = \overline{F(\bar{x}_1, \bar{x}_2 \cdots \bar{x}_n)}$.

2) Self-Dual function:-

$Dual(F) = F$.

i.e. $F(x_1, x_2 \cdots x_n) = \overline{F(\bar{x}_1, \bar{x}_2 \cdots)}$

3) Totally symmetric function:-

under all renamings of inputs, function is same.

4) Partially Symmetric:-

under all renamings of subset of inputs.

5) Symmetry Theorem:-

A function $f(x_1, x_2, \cdots x_n)$ is totally symmetric iff it can be specified by a list of integers

$A = \{a_1, a_2, \cdots a_m\}$; $a_i \in [0, n]$ such that

$f = 1$ iff exactly a_i of the n vars are 1.

(& others = 0).

Eg:

$f = ab + bc + ca$ $\quad A = \{2, 3\}$

$f = a \oplus b \oplus c \oplus d$ $\quad A = \{1, 3\}$

6) Unate function:-

if p occurs, $\neg p$ doesn't appear & vice versa.

Positive unate:- all occur as p.

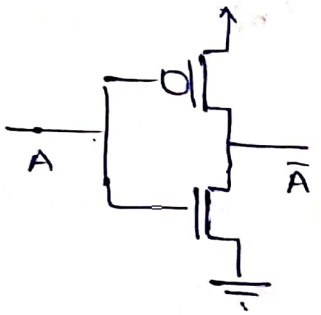
Neg. unate:- all occur as $\neg p$.

7) Reversible:-

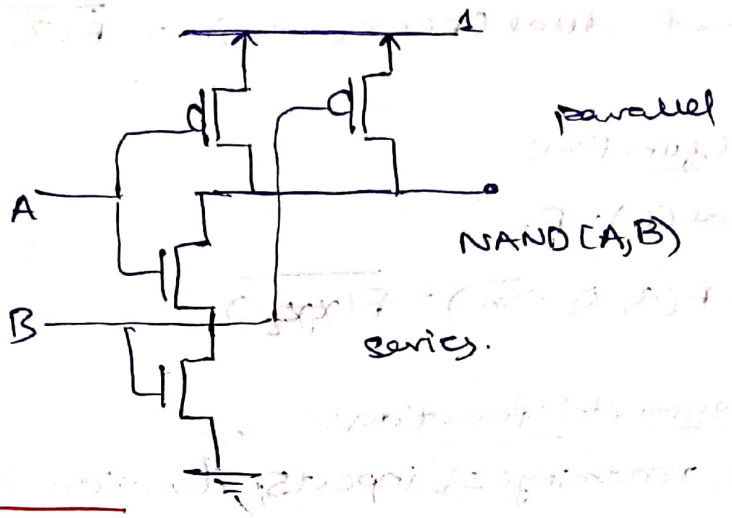
one-one function! $f(x) = \bar{x}$ ✓✓ $f(x,y) = x \wedge y$ ✗.

each output, maps to unique input.

NOT:

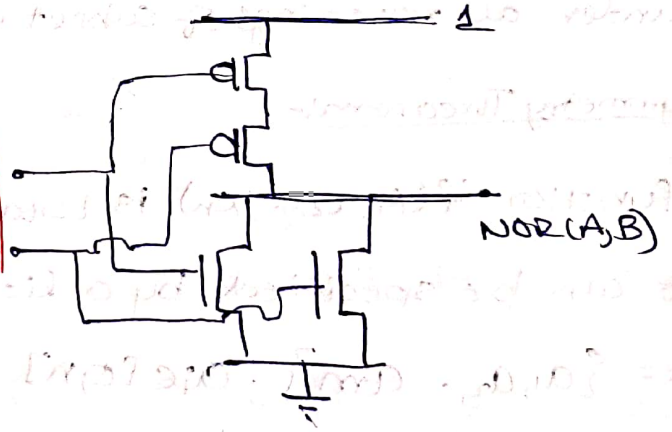


NAND:-



Logic	1 or 2 inp		N inp	
	cost	delay	cost	delay
NOT	2	1	—	—
NAND	4	2	2N	N
NOR	4	2	2N	N
AND	6	3	2N+2	N+1
OR	6	3	2N+2	N+1

NOT:-



→ Implicant or Cube:-

A set of literals ANDed together.

min-term → 0-implicant; 0-cube.

* m-implicant → (n-m) variables.

K-MAPS:-

prime Implicants:-

An implicant; cannot be groun further.

EPI - essential PI:-

* if among the min-terms of PI; atleast one min-term exists, such that its covered by this & only this PI.

RPI - Redundant PI:-

* if all its minterms; are covered by EPIs.

→SPI - selective PI:-

Neither of above two.

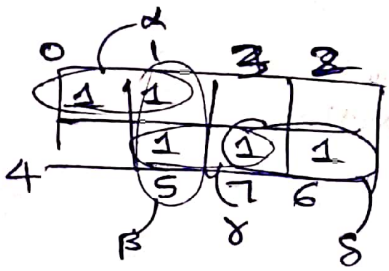
- Not essential
- NOT redundant

occur in pairs

- finding MSOP:-

1. Find all PI.
2. Include all EPI.
3. Find the set of uncovered minterms (UCM).
4. Choose the largest PI from remaining SPI; for any one min-term.
5. Go to step 3.

→ Patrick's method



the minterms are

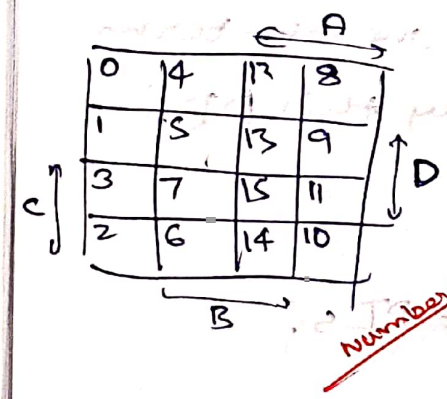
0 – α

1 – α+β

5 – β+γ

6 – δ

7 – γ+δ



ABCD.

Numbering!

$\alpha \cdot (\alpha+\beta) \cdot (\beta+\gamma) \cdot \delta (\gamma+\delta) = 1$

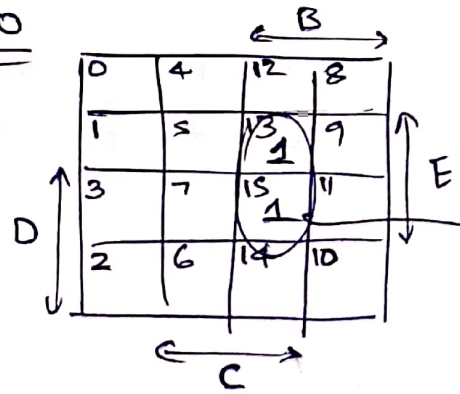
min. solution =?

$\alpha = \delta = 1$

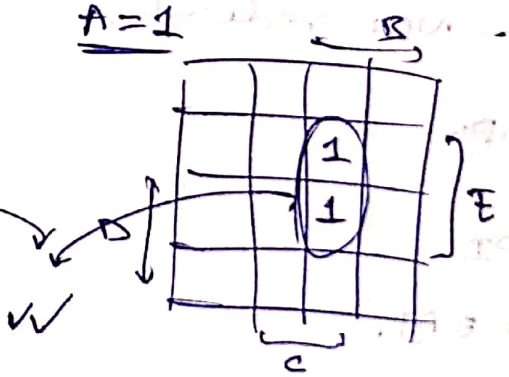
one from $\beta, \gamma = 1$.

5-vars:- ABCDE

A=0



A=1

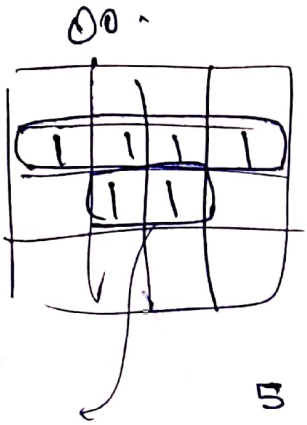


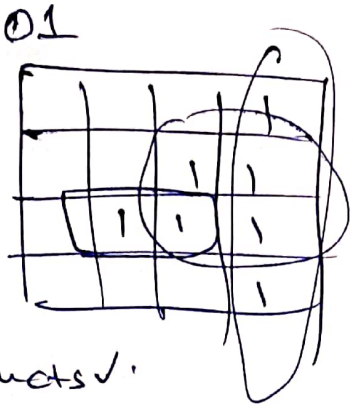
→ with don't care minterms. ✓

→ min POS:-

↳ Same grouping! write MAX terms !

Multiple output minimisation:-





5 products ✓.

NOT optimal;

but if we take 

then

6 products ✓

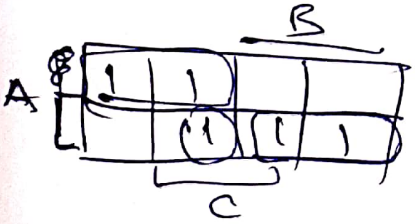
⭐ Reed muller representation from K-map:-

XOR → disjoint sum.

$a+b$; $a \wedge b = 0$ then $a+b \equiv a \oplus b$.

Now; choose Implicants (Need not PI):-

So they don't overlap



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

all all are disjoint mutually!

$= \bar{A}\bar{B} \oplus A\bar{B}C \oplus AB$

$= (1 \oplus a)(1 \oplus b) \oplus AC(1 \oplus b) \oplus Ab$

$= 1 \oplus a \oplus b \oplus ab \oplus ca \oplus abc \oplus ab$

$\equiv 0$

$= 1 \; a \; b \; ca \; abc.$

→ Quine-McCluskey:- to optimize. (K maps - at max 6 variables!)

* we need a scalable method.

: Handle 100s of variables.

* This also uses; uniting theorem. (as K-maps)

$$a\bar{b} + ab = a$$

Step-1° Tabulate all the min terms; in increasing no. of true variables.

2° Conduct linear search to identify all prime implicants.

3° Tabulate PI's vs minterms to identify EPI's

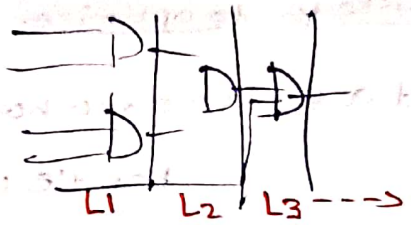
4. Select min. no. of PI to cover the remaining min-terms.

dirty, crude method.

QM Minimizer on the web:-

https://quinemccluskey.com/

→ Multi-level optimization:-



levels.

- For a given function, multilevel circuits can have reduced gate cost, compared to two level circuits.

SOP, POS

* we'll perform "transformations" to go from two-level to multilevel.

1) Factoring:-
   - Algebric : no axioms, specific to bools.
   - Boolean : axioms specific to bools

2) Decomposition:-

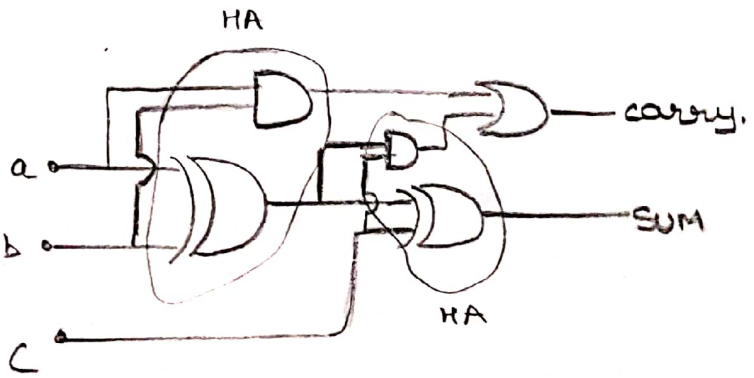
* But, cost effective solution, might not be optimal for performance.

Often, we face this tradeoff.

parallel computing will decrease delays, but increase costs & gates.

Adders:-

- Half-adder
- Full-adder

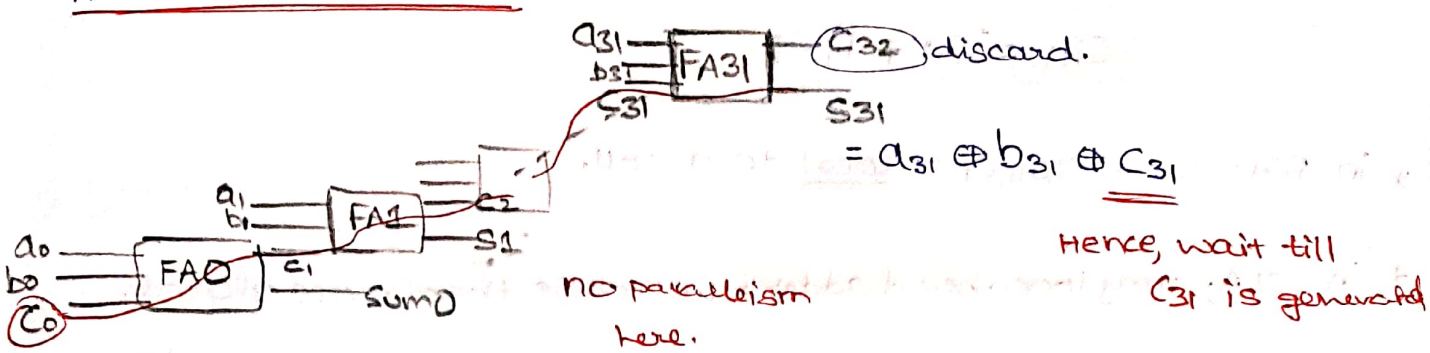


Full-adder

1) 32 bit Ripple carry adder:-

if $C_0 = 0 \rightarrow$ addition.

if $C_0 = 1 \rightarrow$ Subtraction...



* Delay $\propto [32 \times (\text{Delay FA})]$

we should think of better ways for handling carry.

* for n bit ripple adder;

$$\text{delay} = 2 + 4(n-1) \quad O(n)$$

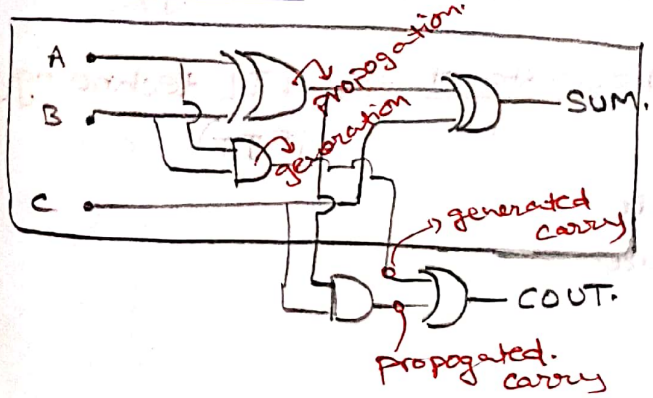
* - Good hierarchy
  - Tileable structures
  - Small hardware.
  - Slow! :(

→ Faster Addition:-

* for big adders, carry chain is very long.

Seperate the sum and carry part.

Partial full adder:-



$$COUT = \underbrace{(A \cdot B)}_{G} + \left(C \cdot \underbrace{(A \oplus B)}_{P}\right)$$

$$C_{out} = G + P \cdot C_{in}$$

"carry lookahead"

∴ at any Adder i;

$$P_i = a_i \oplus b_i$$

$$G_i = a_i \cdot b_i$$

$$\boxed{C_{i+1} = G_i + P_i \cdot C_i}$$

* in RCA; each $P_i G_i C_i$ is local to a cell.

* in CLA; carry lookahead adder; we make them more global.

$$C_1 = G_0 + P_0 C_0$$

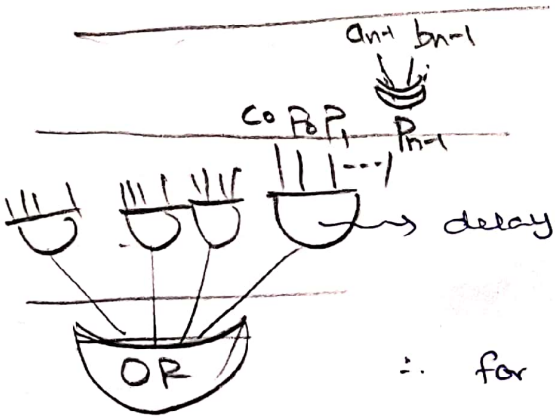
$$C_2 = G_1 + P_1 C_1$$

$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

flattened the chain! (increases speed)

3 logic-level ↓ but not 2-inp basic gates.

$$C_4 = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 + C_0 P_3 P_2 P_1 P_0.$$



∴ for CLA; delay = 8+2n

∴ Better than RCA; but not "that" good.

NOTE: Above discussion;

delay(n-inp gate) α n for CMOS technology.

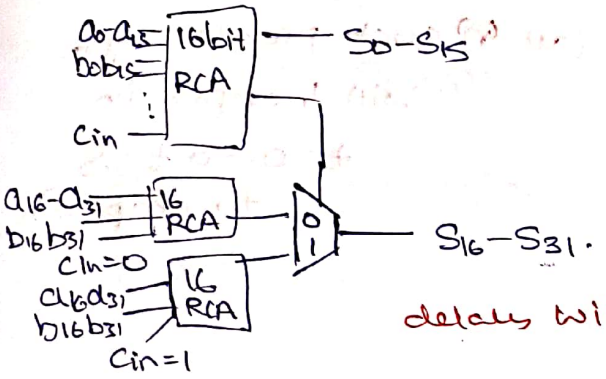
delay(n-inp gate) ≡ constant for TTL technology. (BJT)

∴ in TTL; time α no. of levels in circuit.

* CLA → very expensive in cost.
  - little efficient in performance - CMOS.
  - flattening the logic; needs more gates, but reduces delay.

"parallel" computation.

3. carry select adder:-



delay will be that of 16-RCA + a mux.

4. Carry lookahead development:-

complete lookahead → O(n) CLA

no lookahead → O(n) RCA

partial looking ahead → O(log n) delay

** Kogge-Stone...
Brundt-kung...

no. of levels reduced to $\log_2^N$.

gates increased by $\log_2^N$ times. parallel computation requires more gates n.a. Since, one input (ai) will go to many gates.

→ prefix computation

key architectures for carry computation in a parallel manner.

1960: J-Sklansky
1973: Kogge-Stone adder
1980: Lander-fisher adder
1982: Brent-Kung
!

Prefix addition:

Carry generation $G(i:j)$

&

Propagation $P(i:j)$

over a bunch of cells.

---

1. **Pre-processing:-**

p_i, g_i as $p_i = a_i \oplus b_i$

$g_i = a_i \cdot b_i$

(all of such processings, will happen in $t=0$ to $t=\delta$ timelapse).

2. **Prefix Computation:-**

$$G_{[i:K]} = G_{[i:j]} + P_{[i:j]} \cdot G_{[j-1:K]}$$

$$P_{[i:K]} = P_{[i:j]} \cdot P_{[j-1:K]}$$

* This step is where, we'll show our genius
  - where we'll plan our "ARCHITECTURE"
  - where there is scope to get better.

3. **Post-processing:-**

we'll have $C_0 \rightsquigarrow$ input carry.

& $C_{i+1} = G_{[i:0]} + P_{[i:0]} \cdot C_0$

$S_i = p_i \oplus C_i$

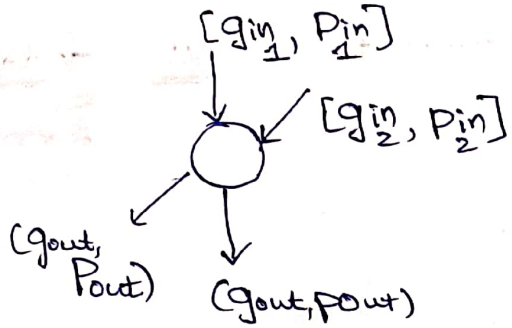
Hence, we need every C_i (in an indirect manner)

we need every $G[i:0]$ & $P[i:0]$.

→ Prefix Graphs: easier to "show" our architecture.

Basic components:-

processing component:-

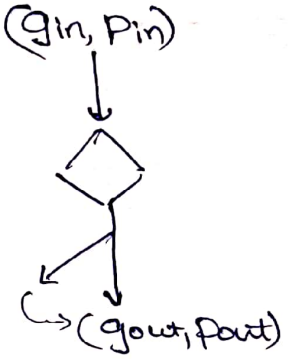


$g_{out} = g_{in_1} + P_{in1} g_{in2}$

$p_{out} = P_{in_1} \cdot P_{in_2}$

See slides Lec-18. They are elaborate

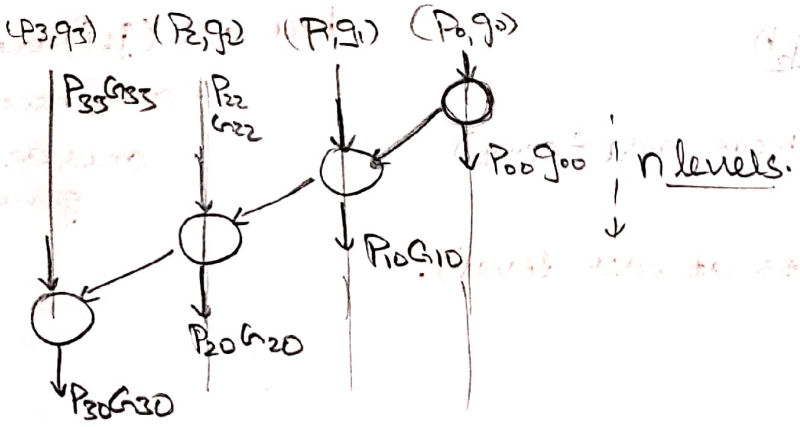
Buffer component:-



$g_{out} = g_{in}$

$P_{out} = P_{in}$

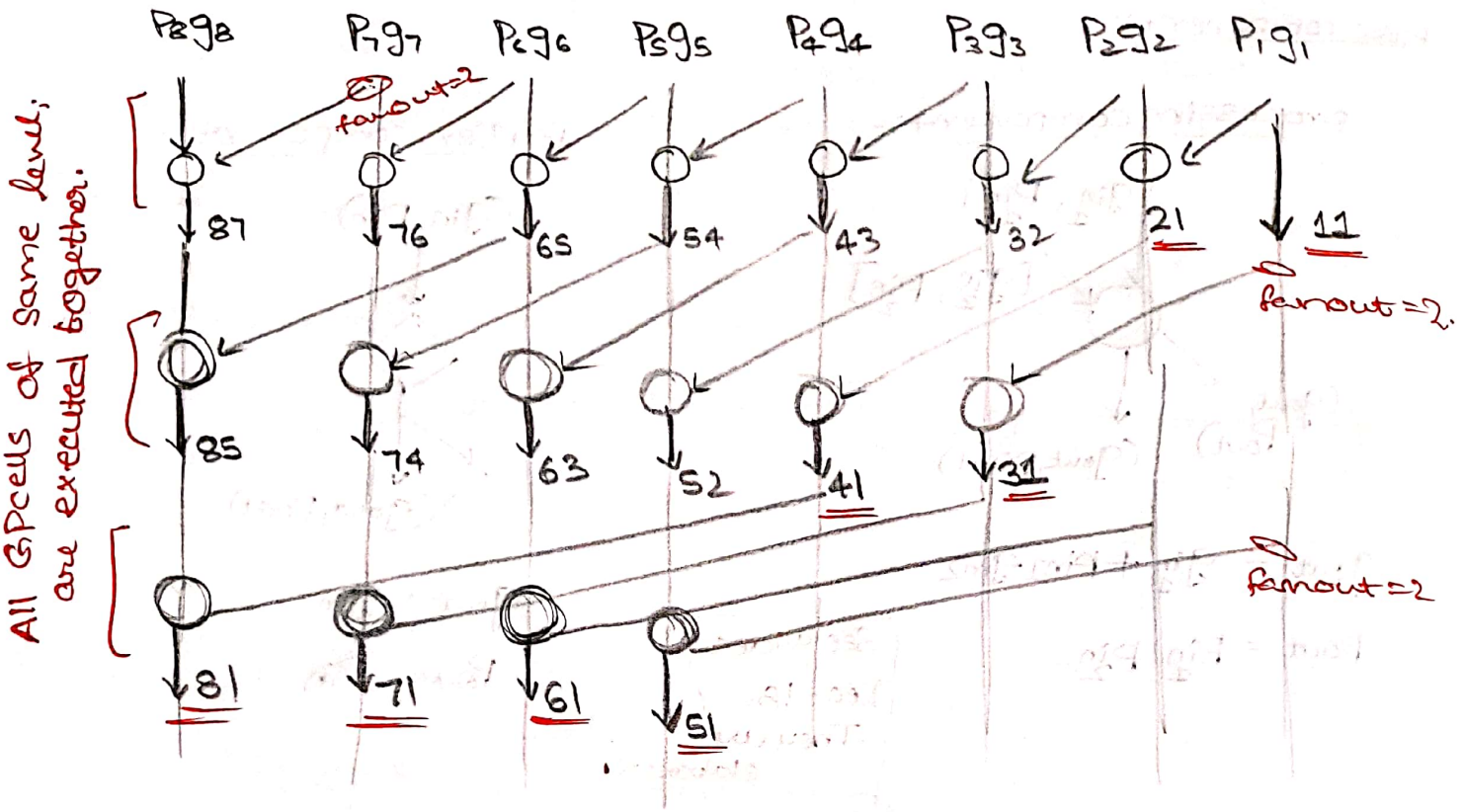
* RCA:



<See slide 16> → has elaborate explanation.

- GP generator (from a_i, b_i) Pre processing
- GP cell (from $g_{in_1}, P_{in_1}, g_{in_2}, P_{in_2}$) prefix computation
- Sum generator (from $C_0, G_{i:0}, P_{i:0}$) post processing.

Kogge-Stone Adder:-



* The Kogge-Stone adder has

1) Low depth $\equiv \log_2 n$

2) High node count (more area & cost)

3) Max fan-outs = 2.
(splits at each level).

all (i:1) are generated !
but all (j:i) are not!
• only necessary ones are generated.

• Brunt-Kung:-

1) depth = $2 \cdot \log_2 n - 1$ <lookup the "architecture" in slides>.

2) node count: moderate.

3) max fanouts = 2.

→ Multiplier:-

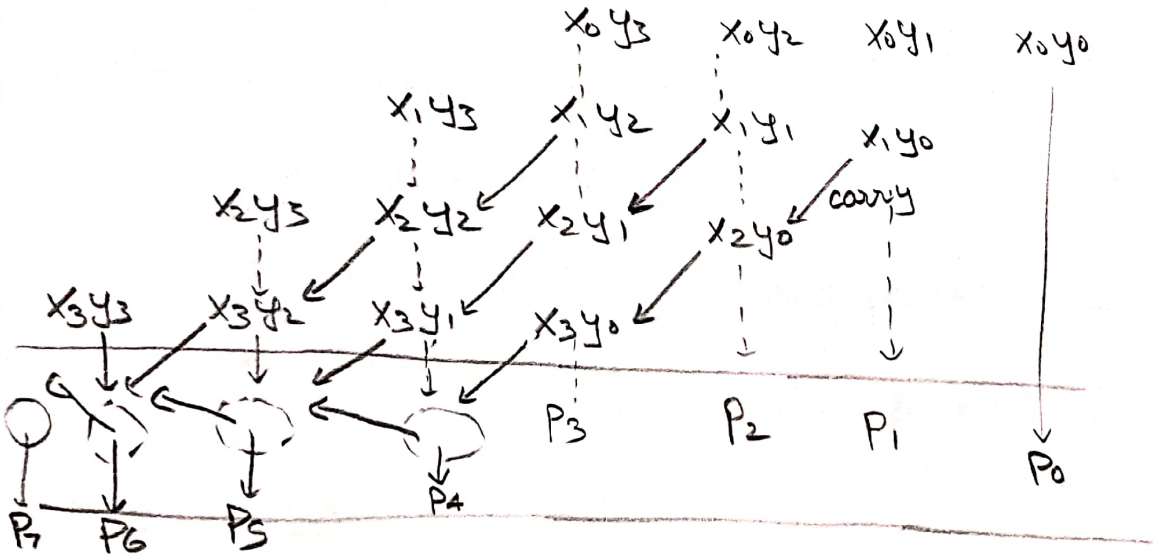
```
    1000      8₁₀
  × 1001      9₁₀
  ------
    1000
   0000(0)
  0000(0)(0)   ← don't exist....
 1000(0)(0)(0)
 ------------
 1001 1000     (72₁₀) ✓
```

→ each row is a partial product.

* Array multiplier: Carry forward:-

y_3 y_2 y_1 y_0 — multiplicand

x_3 x_2 x_1 x_0 — multiplier



* carry from k^{th} partial product; is added to $(k+1)^{th}$ partial product (carry-save addition)

* if we have 3 bits to be added;

we use FA. Haha...

& we notice that, above;

we'll never have more than 3 bits addition together.

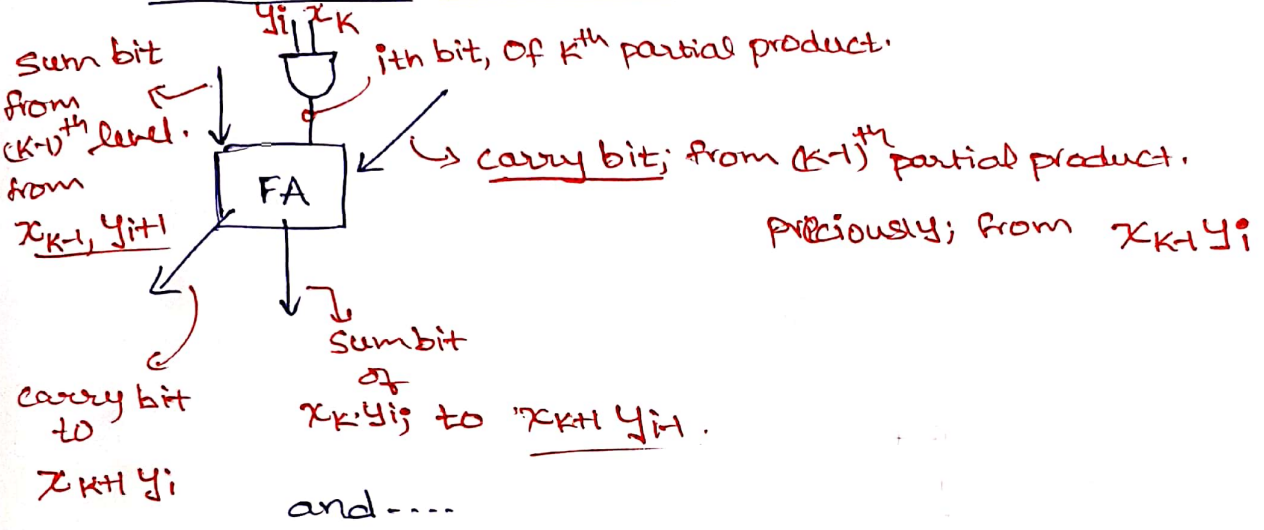
ANYWHERE!

Basic building blocks:-

- Two input (and) g- for partial product terms.

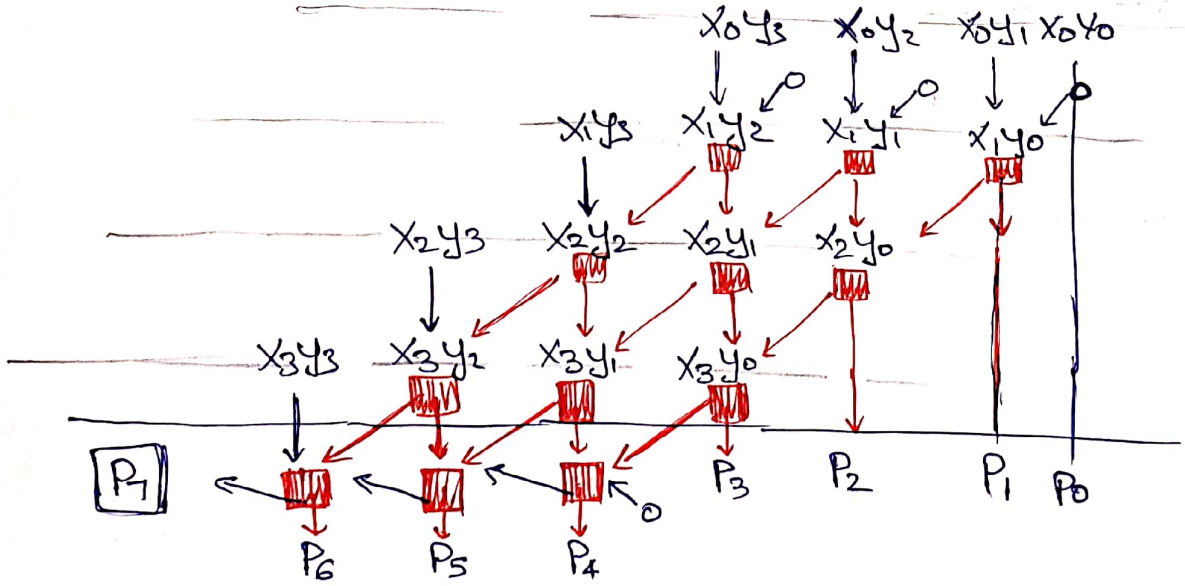
$P_{0i} = x_0 y_i$

- Full adder... or rather 2-AND input to a FA:-



and....

ARRAY MULTIPLIER:- <plz see slide 9 of lec 19A>. lol.



FA:-

In slide; the leftest most adl carries are '0'. He didn't mention they are '0'. But I skipped them here!

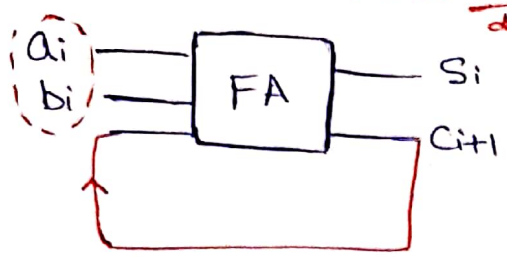
done!

Sequential Circuits:-

→ Serial adder:-

An FSM & its STG → Same thing.

send the next one; after 10 units of time. delay(FA)



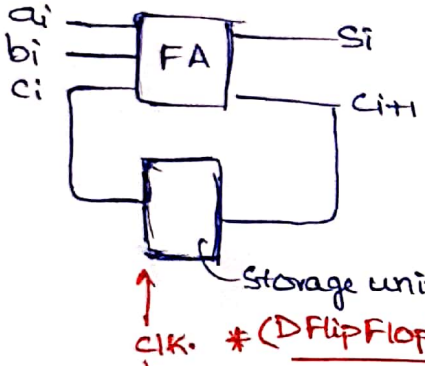
feed the Ci+1; after 10 units time.

delay will be same as RCA;

but cost = 1 FA.

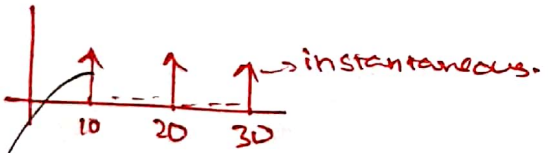
* Party poppers
* Screams crazy!

*



* (D FlipFlop)
* (some example with two inverters...)

will be like



clock period = $\frac{1}{\text{clock frequency}}$

1GHz, 10GHz...

could be rising edge of a signal

*

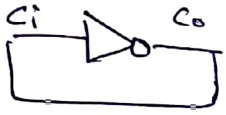
Here, existance of clock, is absolute! Storage.

If we simply feed the output to input; then short circuit will occur (-_-).

there won't be a "Stable" state.

well; there could be; but very limited! useless, to develope circuits

Eg:



no stable values na!

One stable state is

Ci, Co = 0

another is

Ci, Co = 1

but never

$Ci \oplus Co = 1$.

we might need this for our circuit. Hence use clks.

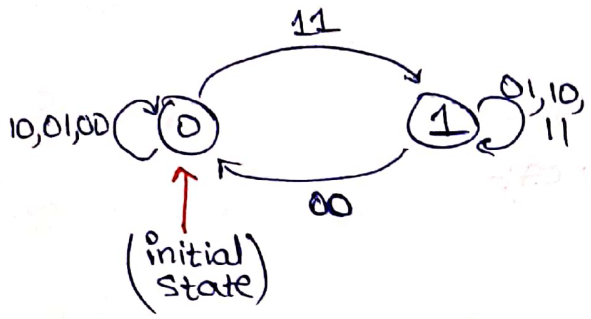
* Output dependent on, not only current inputs, but also prev. inputs

temporal behaviour.

* State:- past behaviour memorized.

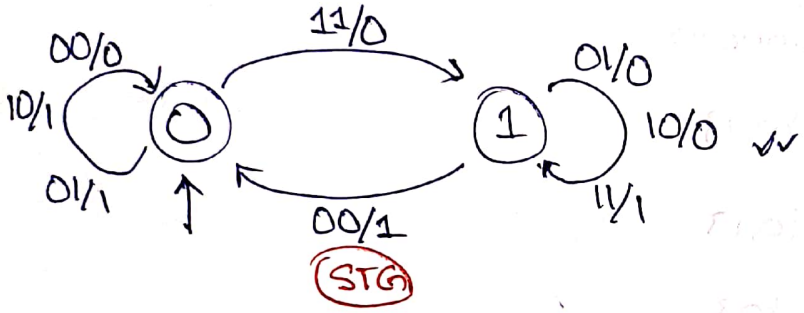
  Here; carry is memorized.

→ State transition diagram:-



nodes ≡ state

edges ≡ input/output transition!



→ The logic circuits:-

we'll have two logic requirments:-

Output logic : from inputs & pseudo inputs (current state)

↑

Both are combinatorial

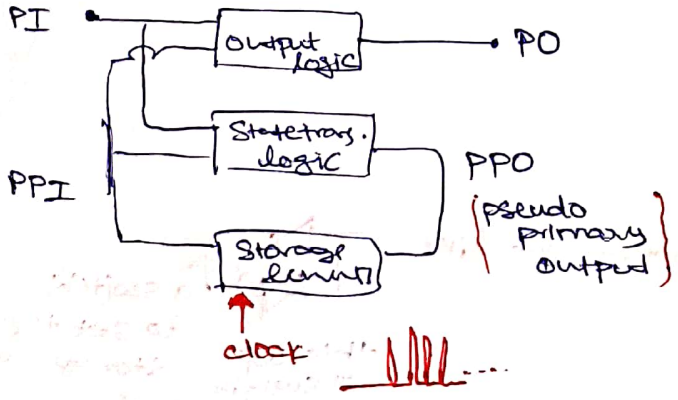
↓

State transition logic: from inputs & current state

Storage elements : to store the current state.

'n' unique states → $\log_2 n$ elements needed.

*



State	Inputs 00	01	10	11
0	0/0	0/1	0/1	1/0
1	0/1	1/0	1/0	1/1

→ State transition table.

New state / output.

Finite State Machines:-

$M = (I, O, S, S_0, \delta, \lambda)$

I = input symbols = {00, 01, 10, 11}

O = output symbols = {0, 1}

S = set of states = {0, 1}

S_0 = initial state = {0}

$\delta : S \times I \longrightarrow S$ State transition function.

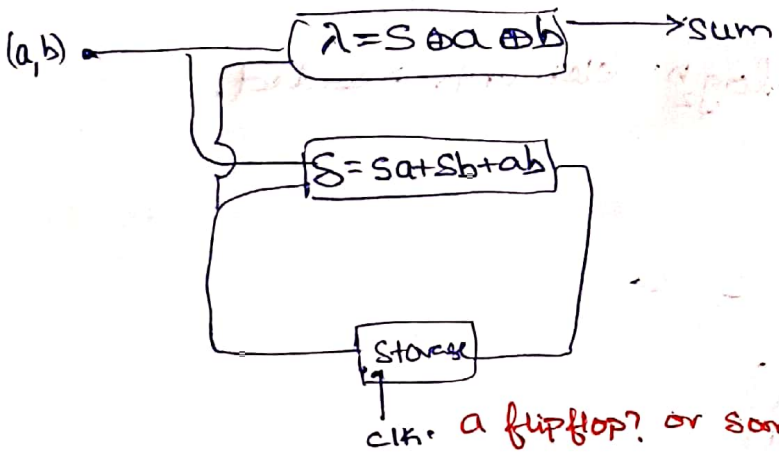
$\lambda : S \times I \longrightarrow O$ output function.

if more state storages; $\delta_0, \delta_1, \ldots$

if more outputs; $\lambda_0, \lambda_1, \ldots$

use Kmaps or something to build δ, λ. (your lab assignments!)

Serial adder:-



a flipflop? or something like

a switch; to set the storage value.

this loop will sustain the signal.

* hence,

> define states | we will note down the state using $\log_2 S$ flipflops.

> construct STG. or STT.

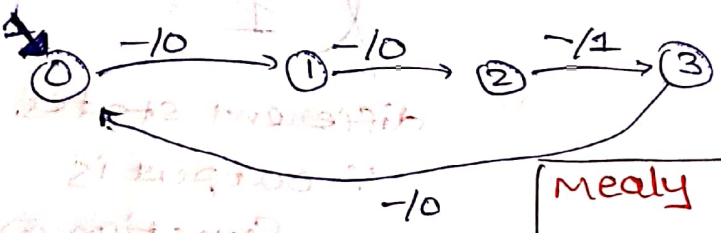
> # storage elements = $\log_2 |S|$

** > define encoding on states, inputs, outputs...

> compute δ_i, λ_j for all i,j....

→ mealy-moore machine:-

counter:-

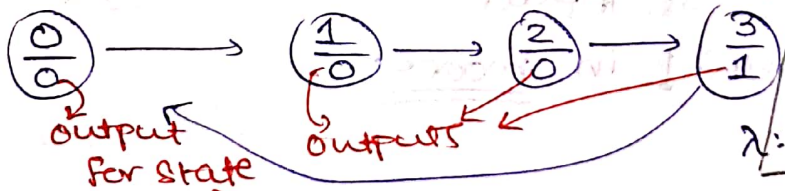


Mealy

$\lambda : S \times I \rightarrow O$

Next, upcoming:-

- mealy-moore machine
- state minimisation
- storage
- verification



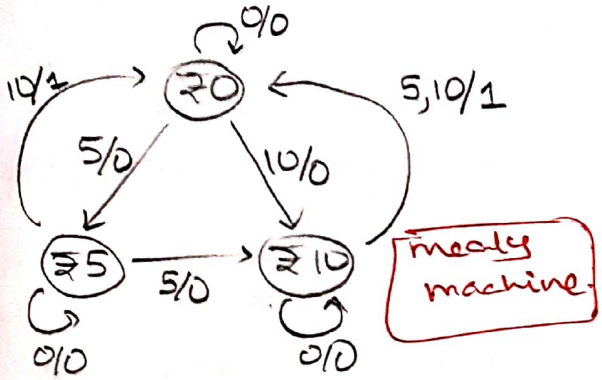
Moore

$\lambda : S \rightarrow O$

note that state transition is still function of inputs.

• vending machine:-

(15₹ items)



Mealy machine

3 input values → 2 bits

5 states → 2 bits.

→ can I synthesize this as a moore machine?

Yes, ofcourse!

Whaaa!....

δ is SxI → S for moore too!

moore machine

9 states

"dispense"!

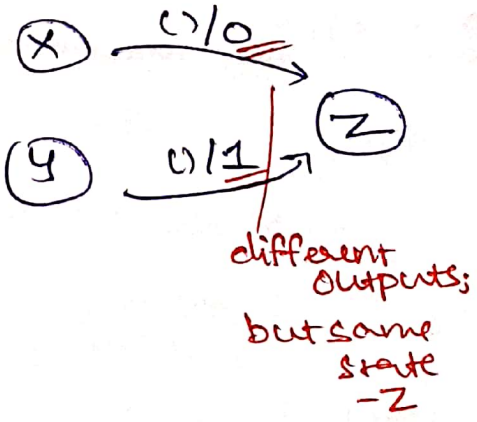
output is function of state.

• Here moore has more states; bcoz, mealy have have one state, multi-possible outputs.

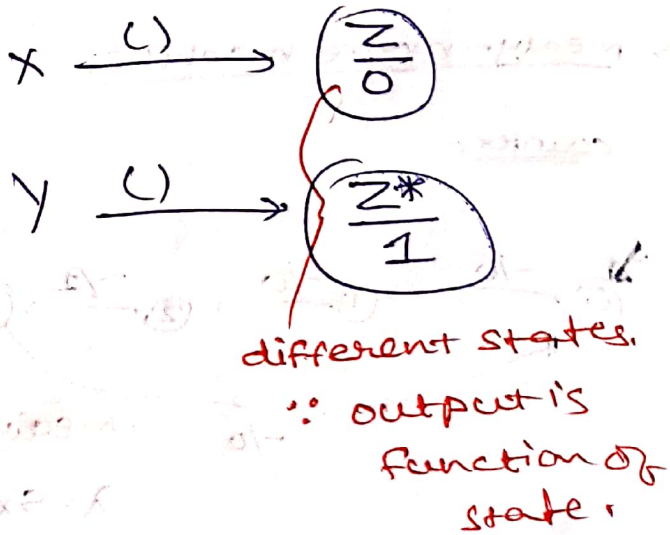
but moore has
1 state — 1 output.

like;

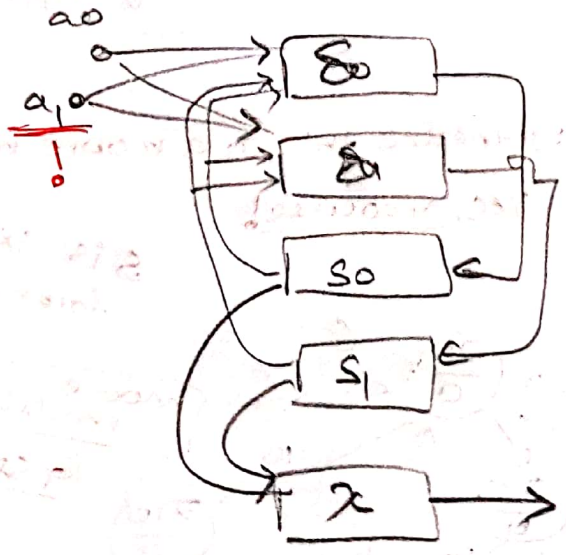
mealy :-



moore :-



* note that δ can have
I_1, I_0 as inputs
λ shouldn't! in moore.



→ State minimization:-

X-Successor:-

$S_i \xrightarrow{seq\,X} S_j$ then S_j is X-successor of X_i.

Strongly connected:-

For every pair (S_i, S_j) there should a seq X to go.

* S_i, S_j are distinguishable; only if there exists atleast one finite seq, of machine M when applied to M_i causes different output sequences.

depending whether S_i, S_j was initial state.

seq → distinguishing sequence of (S_1, S_2)

Of len K → (S_i, S_j) are K-distinguishable

if (S_i, S_j) are not K-distinguishable] than K-equivalent.

no such K length seq exists.

* Two states S_i, S_j are equivalent, if for every possible input seq, output is same!.

output seq. not just last output!.

∴ States that are K-equivalent for K=n-1 are equivalent

n → no. of states

property:

if S_i, S_j are equivalent;

their X-successors are also equivalent.

* Hence, States of a FSM can be divided into equivalence classes.

1. Partition the states into subsets.
2. of equivalences.

Two states are ~~0~~ 1-equivalent if their output is same.
are 2-equivalent if their 1-successors are 1-equivalent

<img L23, p11>.

lol.

PS	NS, Z	
	x=0	x=1
A	E,0	C,0
B	C,0	A,0
C	B,0	G,0
D	G,0	A,0
E	F,1	B,0
F	E,0	D,0
G	D,0	C,0

$P_0 = (ABCDEFG)$

$P_1 = (ABCDFG)(E)$ "equivalence classes"

same outputs.

$P_2 = (AF)(BCDG)(E)$

check if their 1-successors are 2-1 equivalent

$P_3 = (AF)(BD)(GC)(E)$

$P_4 \neq$

$P_5 \neq$

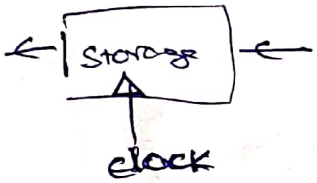
* members of a k-equivalence class,
if members are having successors of same k-equiv class,
then,
these members are
(k+1) equivalence

x——minimisation——x

→ Storage:-

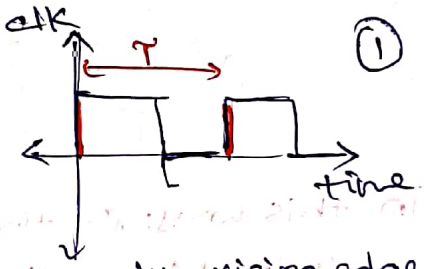
if we have S-States ⟶ S flipflops } One HOT storage. Certly!

↳ optimal $\lceil \log_2 S \rceil$



T = longest path between PI/PPI & PO/PPO

we can have 2 kinds of clock



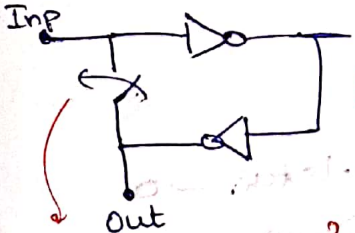
① with rising edge detection in storage element

We use this

② apply differentiator to ①.

normal storage bit.

Designing storage:-

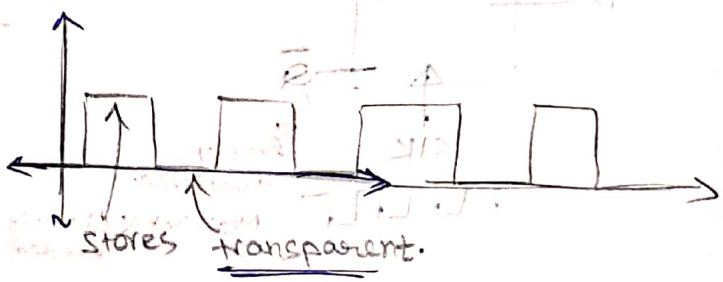


LATCH → stores when clk is HIGH (not just rising edge).

closed when clk=1 } stores value

open when clk=0 } transparent!

out = = in.



this not an "instant"!

this is an interval.

the storage "might" change more than once.

Wrong results.

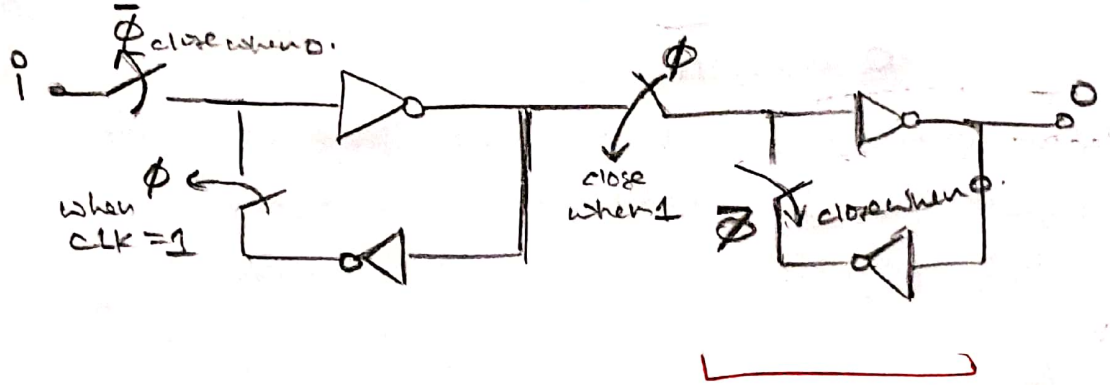
if delay in circuit > $\frac{T}{2}$ then; changed only once! → then, no problem

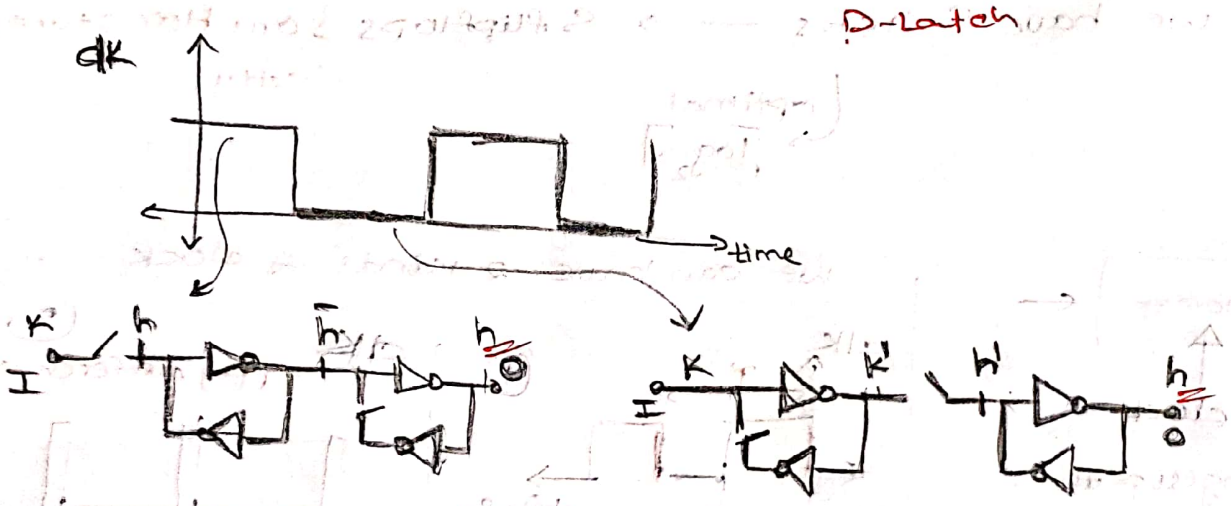
in any paths there are techniques to ensure such delays.

SA will come here if time permits

lets try to be sensitive to rising edge.

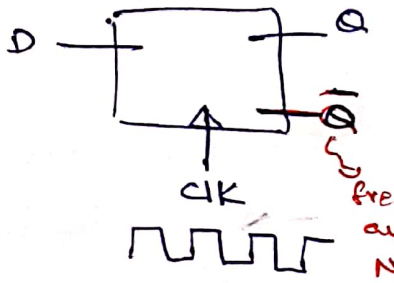
By cascading two latches.



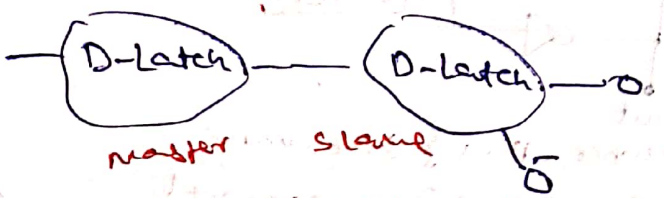


in this way; for the complete cycle; output (or rather storage) is preserved; & changes only at rising edge of clk. (& not at falling edge)

* D-flipflop diagram:-



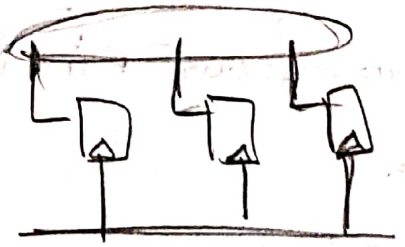
we saw above.



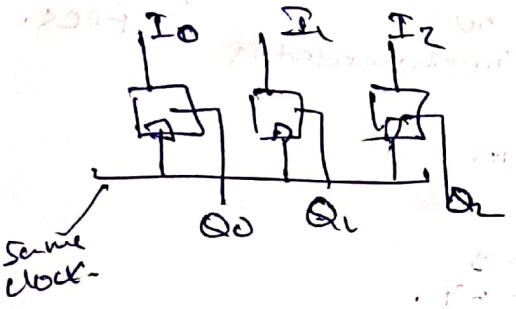
master-slave design.

"this is THE only flip-flop being used in any practical circuit".

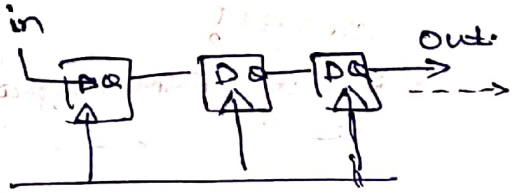
* I wanna store multiple bits; not a single bit...
use more D-FFs for making a register.



1) parallel-in parallel out



2) serial in-serial out:-

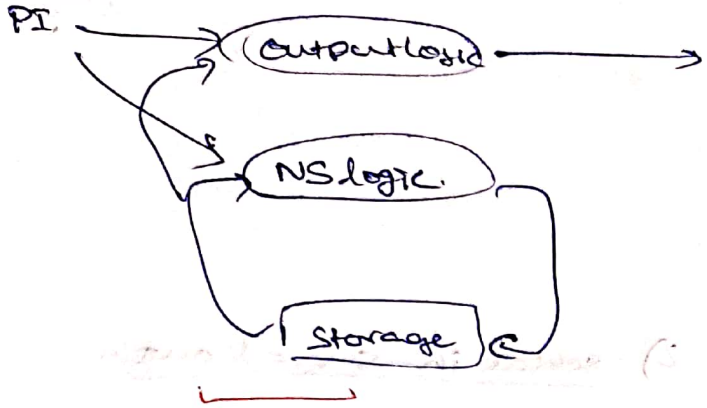


3) "pseudo random numbers".

<L24, Pg 10>

Verification:-

FSM:



circuit!
& it has a FSM diagram

How to say; our FSM is correct.

i.e.

How to say

$M_1 \equiv M_2$

M_1 → our implementation

M_2 → Specs.

Combinational circuits:- ROBDD;
SAT checking

Sequential:-

1) represent each circuit as FSM.
sequential

Approach1:- Reduction to combinational circuit

* unroll over 'n' timeframes

Now, its combinational logic.

But: unrolling will become large.

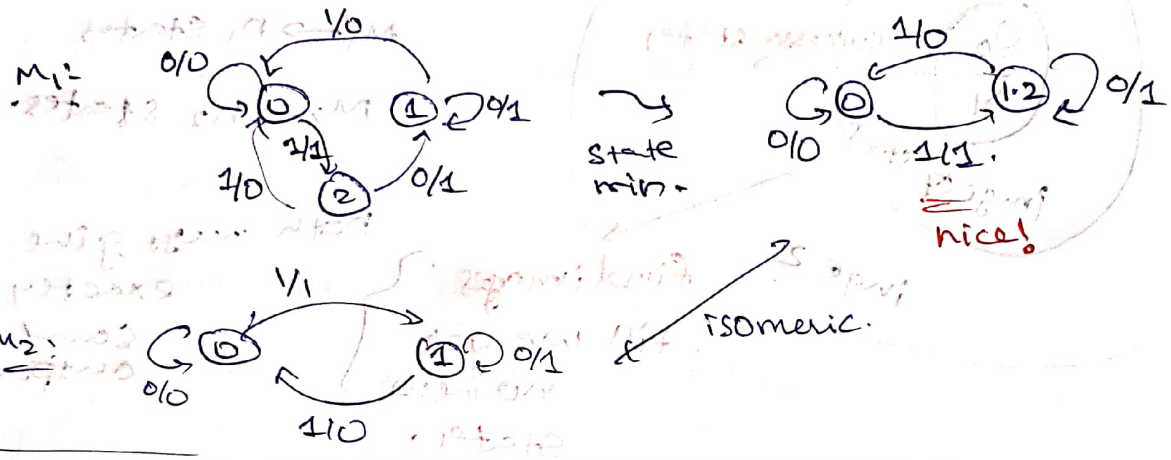
* unroll both M1, M2 as such & put XOR b/w corresponding output vals.
& OR them.
OR must be 0;
for equivalence.

Approach 2:- Based on isomorphism of STG.

* two FSM M_1, M_2 are equivalent, if their STG are isomorphic.

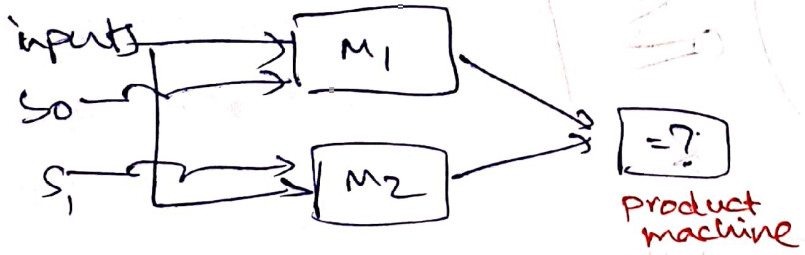
→ First perform state minimisation on each

→ check if STG (M_1) & STG (M_2) are isomorphic.

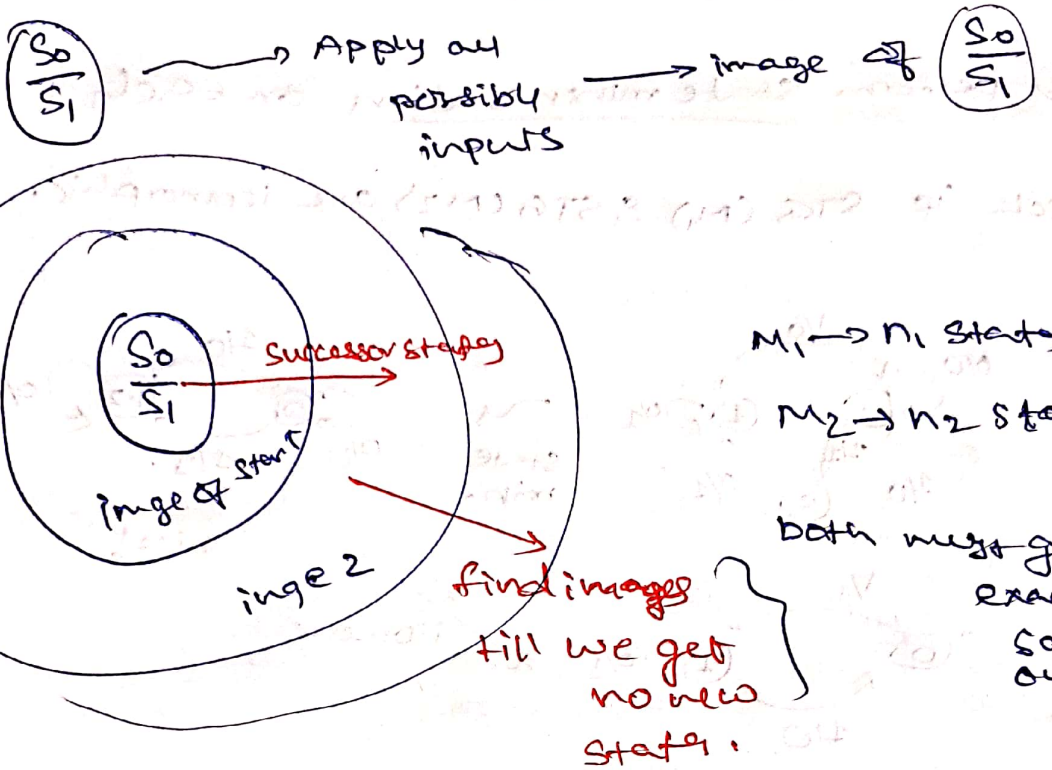


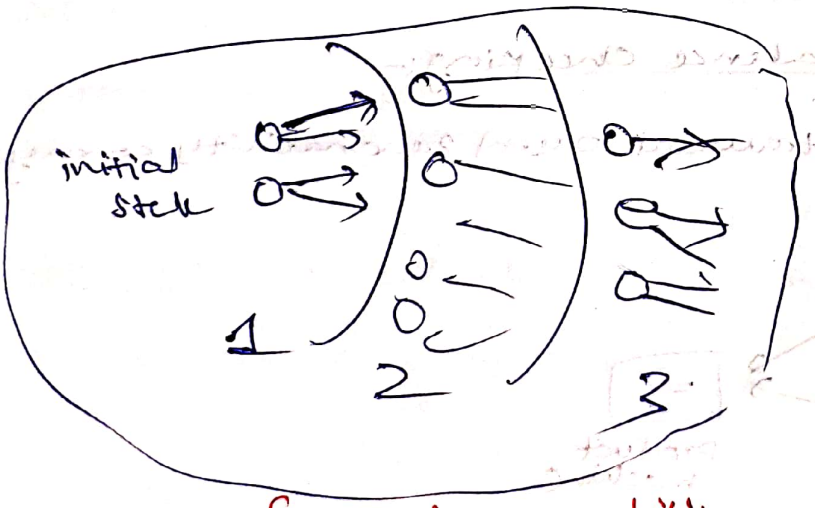
Reachability based equivalence checking:-

Approach-3:- Symbolic traversal based reachability analysis.



1. build product machine of M_1, M_2.
2. traverse state-space of product machine; starting from reset state S_0, S_1.
3. Test equivalence of outputs in each state.
4. Can use any state-space traversal technique.

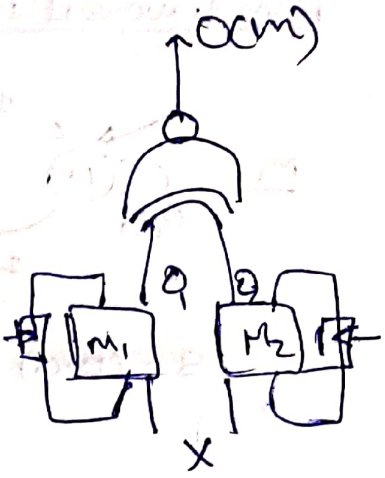




forward reachability.

* Given two FSMs $M_1(X, S_1, \delta_1, \lambda_1, O_1)$

$M_2(X, S_2, \delta_2, \lambda_2, O_2)$



create product FSM $\Rightarrow M = M_1 \times M_2$.

> traverse the States of M & check its output for each transition. ($n_1 \times n_2$ states)

> if output(M) = 1 → then same $O_1 = O_2$

> if all outputs are 1, then $M_1 \equiv M_2$

> otherwise; error state is reached.

> trace back, to show $M_1 \neq M_2$

$M(X, S, S^0, \delta, \lambda, O)$

$O = \{0, 1\}$;

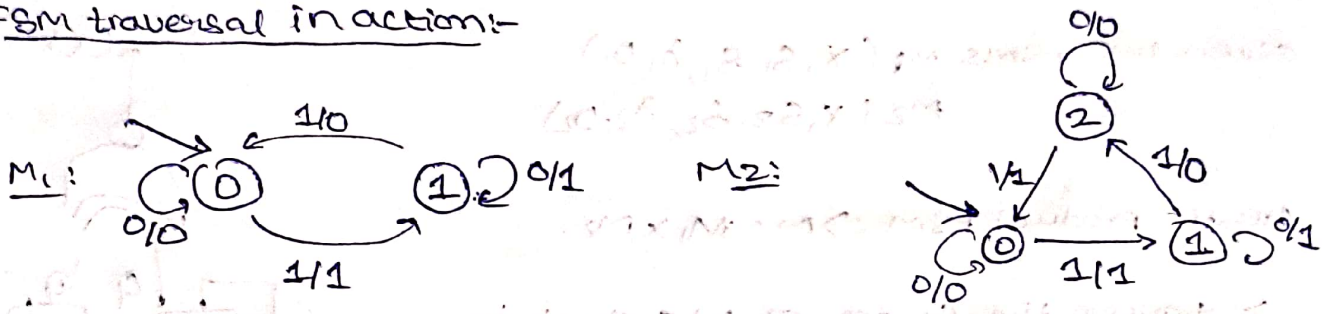
$S = S_1 \times S_2$.

$\delta(S, X) : (S) \times X \longrightarrow S$

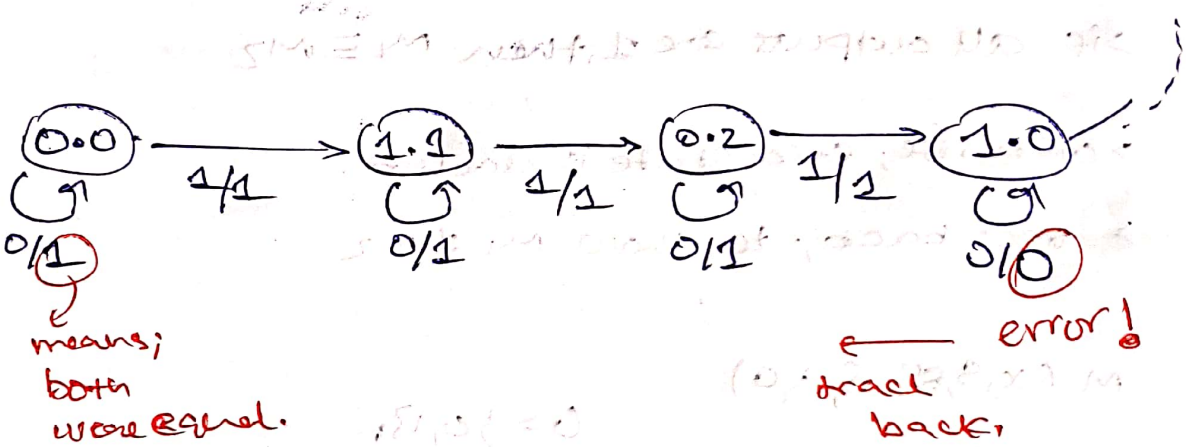
output : $(S) \times X \longrightarrow \{0, 1\}$

$$\lambda(S, X) = \lambda_1(S_1, X) \overline{\oplus} \lambda_2(S_2, X)$$

FSM traversal in action:-



$S = (0,0)$



$\therefore x = \{1\,1\,1\,0\}$ is distinguishing sequence.

don't go, till you get new states!

*Graphically okay. But how in machine?

Traverse the product machine $M(X,S,\delta,\lambda,O)$

> initial state S_0

> iteratively compute symbolic image $Img(S_0,R)$ (set of next states):-

$$Img(S_0,R) = \exists_X \exists_S S_0(S) \cdot R(X,S,t)$$

(input, state, next state.)

$$R = \Pi_i R_i = \Pi_i (t_i \equiv \delta_i(S,X))$$

until error state reached.

* M doesn't have all $n_1 \times n_2$ states. we only have a set of reachable states.

(represent by boolean funct^n)

showing set of reachable states;

say states are

St_1	St_2	St_3	St_4
00	01	10	11

∴ 2 boolean variables

S_0, S_1

q if only

{St_1} is reachable; represent by expression $\bar{S_0}\bar{S_1}$

S_0	S_1	reachable	
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1
		$\bar{S_0}$	$\bar{S_0} + S_0\bar{S_1}$

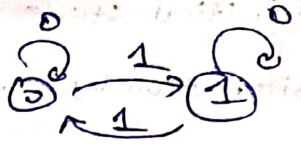
as we traverse states; this new ROBDD will be very compact

use ROBDD to save set of reachable states.

whatever! Okay!

showing the transition ~~function~~ relation.

x, s, t
- x → input
- s → initial state (symbolic)
- t → final state (symbolic).



relation!

R → transition diagram....

x	s	t	
0	0	0	1 true
0	0	1	0 false
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

∴ R(x,s,t) is a bool function. or ROBDD.

s, t are single bit here; but can be bit vector!

all possible transitions.

* NOW C(s) → set of states reachable by now.

R(x,s,t) • C(s)

(and)

Boolean expressions.

→ transitions possible from C(s).

Say; we have δ_1 δ_2 of two machines.

each machine has 2 states.

s_1, t_1 — t_1 : next state

s_2, t_2

$$t_1 = \delta_1(s_1, x) \qquad t_2 = \delta_2(s_2, x)$$

machine 1 machine 2

∴ our boolean expression $R(x, s, t)$ is written as

for product machine

$$R(x,s,t) = (t_1 \Leftrightarrow \delta_1(s_1,x)) \cdot (t_2 \Leftrightarrow \delta_2(s_2,x))$$

transition RELATION.

transition FUNKTION

boolean expression

AND operation

boolean expression...

in general; Symbolic state representation may need more than one bit.

like

$s_{a_2} s_{a_1} s_{a_0}$; $s_{b_1} s_{b_0}$.

$t_{a_2} t_{a_1} t_{a_0}$; $t_{b_1} t_{b_0}$

well have:

$\delta_{a_2} \delta_{a_1} \delta_{a_0}$; $\delta_{b_1} \delta_{b_0}$.

$$\therefore R(x,s,t) = \prod_{\text{every subscript } (i)} (t_i \Leftrightarrow \delta_i(s,x))$$

over all machines.

∴ $R(x,s,t) = 1$ denotes the set of encoded tripples, each representing a transition in the FST.

∴ $f(x,s,t)$ is $R(x,s,t) \cdot C(s)$

transition function relation

current state

we need to eliminate x's to get $f(t)$

set of image steps.

* we'll use a new boolean operation.

- existential abstraction

$$\exists_{x_i} \cdot f = f_{x_i} + f_{\overline{x_i}}$$

okay..... got it!

∴ now abstract out x's from $f(x,s,t)$

∴ we got $f(t)$

image steps

boolean function.

hmm-

∴ Procedure:-

→ compute TF; $R(s,x,t)$

→ compute conjunction of R and C

→ existentially abstract all s variables $(s_0, s_1, s_2, \ldots)$ & all x variables $(x_0, x_1, \ldots)$

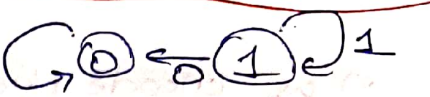
we get $Img(t)$.

→ $Img(t)$ is the smallest function independent of s,x which contains all tuples of $f(x,s,t)$

Eg: 2 states possible; 2 inputs possible.

example currently; we are discussing for single machine

∴ S, t x.



say $\delta(S,x) = S \cdot x$

now; $R(S,x,t)$ is $(t \Leftrightarrow S \cdot x)$

$t \cdot S \cdot x + \bar{t} \cdot \overline{(S \cdot x)}$

$[R(S,x,t) = tsx + \bar{t}\bar{s} + \bar{t}\bar{x}$

encompasses all triples; valid....

say initial is state 0 $\Rightarrow \bar{s}$

$\therefore f(x,s,t) = R(x,s,t) \cdot \bar{c}(s)$

$= (tsx + \bar{t}\bar{s} + \bar{t}\bar{x}) \cdot \bar{s}$

$f(x,s,t) = \bar{t}\bar{s} + \bar{t}\bar{x}\bar{s} = \bar{t}\bar{s}$

↓ existential abstraction over x, s

$= f_{xs} + f_{\bar{x}s} + f_{x\bar{s}} + f_{\bar{x}\bar{s}}$

$= f_s + f_{\bar{s}}$

$= \bar{t} \cdot 1 + \bar{t} \cdot 0$

$f(t) = \bar{t}$

$\Rightarrow$ image state is 0

if; say $c(s) = s$

then $f(x,s,t) = tsx + \bar{t}s\bar{x}$

$f_{xs} + f_{x\bar{s}} + f_{s\bar{x}} + f_{\bar{x}\bar{s}}$

$= t \cdot 0 + 0 + \bar{t}$

true! $f(t) = 1$ means; $t = 0$ or 1 image state is 0, 1.

Eg:

1) we store the states using a boolean expression (ROBDD)

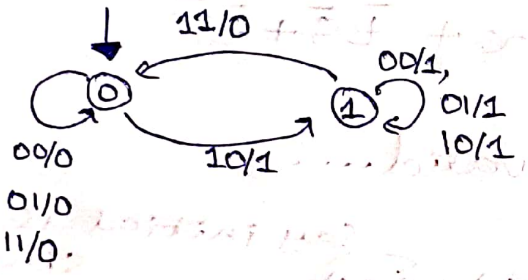
2) we encode the transition function, as a boolean expression (complete!) (ROBDD)

3) we find the images as a boolean function.

4) we do existential abstraction to find the set of image states

5) we disjunct this new states with old CCS to get all states.

M_1:



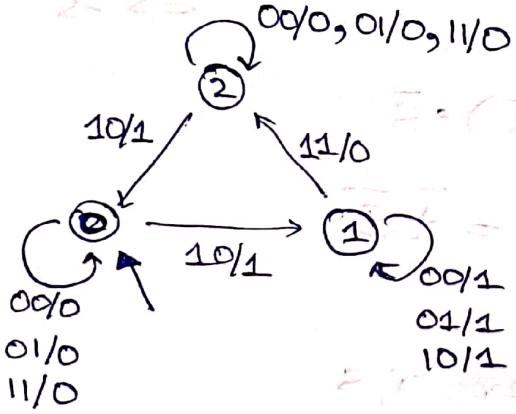
we'll show states as

S_1 for M_1 t_1

S_2S_3 for M_2. t_2t_3

inputs as x_1x_2 for both machines.

M_2:



M_1

$t_1 = \delta_1^1 = x_1x_2' + s_1x_1'$ (K-maps)

$\lambda^1 = \delta_1^1$

M_2

$t_2 = \delta_2^2 = s_3x_1x_2 + s_2(x_1' + x_2)$

$t_3 = \delta_3^2 = s_2'x_1x_2' + s_3x_1'$

$\lambda^2 = s_3x_1' + x_1x_2'$

} by K-maps.

The product ~~function~~ machine will have

State as $s_3s_2s_1$

input as x_2x_1

→ need to calculate just once.

Transition relation $R(x,s,t)$ is $(t_1 \Leftrightarrow \delta_1^1) \cdot (t_2 \Leftrightarrow \delta_2^2) \cdot (t_3 \Leftrightarrow \delta_3^2)$

& initial state is 0·0 (or) 000.

The $f(x,s,t) = R(x,s,t) \cdot C(s)$

$f(x,s,t) = (t_1 \Leftrightarrow \delta_1^1) \cdot (t_2 \Leftrightarrow \delta_2^2) \cdot (t_3 \Leftrightarrow \delta_3^2) \cdot s_1's_2's_3'$

Solve ahead; from this stage only.

Don't simplify here.

we'll do existential abstraction wrt S.

* only $s_1=0=s_2=s_3$ is needed; since all other conjunctions are zero. !!

$\therefore f(x,t) = (t_1 \Leftrightarrow x_1 x_2')(t_2 \Leftrightarrow 0)(t_3 \Leftrightarrow x_1 x_2')$

$\exists_x f(x,t) = \sum_{x=00,01,10,11} f(t)$

$= (t_1 \Leftrightarrow 1)(t_2 \Leftrightarrow 0)(t_3 \Leftrightarrow 1) + (t_1 \Leftrightarrow 0)(t_2 \Leftrightarrow 0).(t_3 \Leftrightarrow 0)$

$f(t) = t_1 \bar{t_2} t_3 + \bar{t_1} \bar{t_2} \bar{t_3}$

$\therefore$ image states are 101, 000

1.01 0.00

= 1.1 = 0.0

we'll again $C(S) = C(S) + f(t)$.

& continue same!

until error state is reached!

"forward reachability"

* we need to keep checking whether $\lambda_1(S_1,X) \oplus \lambda_2(S_2S_3,X)$ is 0 anywhere.

everywhere we wrote boolean;

but its ROBDD.

"extremly compact" - virendra singh sir.

Same can be applied for software verification.