National Logistics Cell

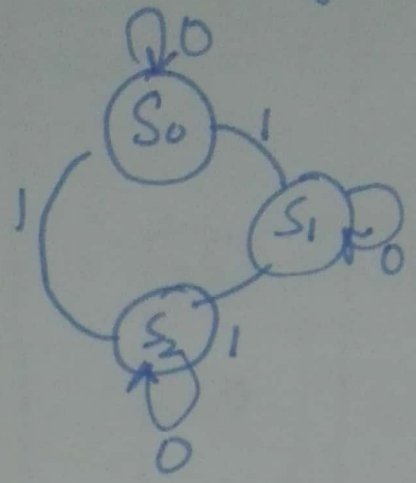
Lecture #20

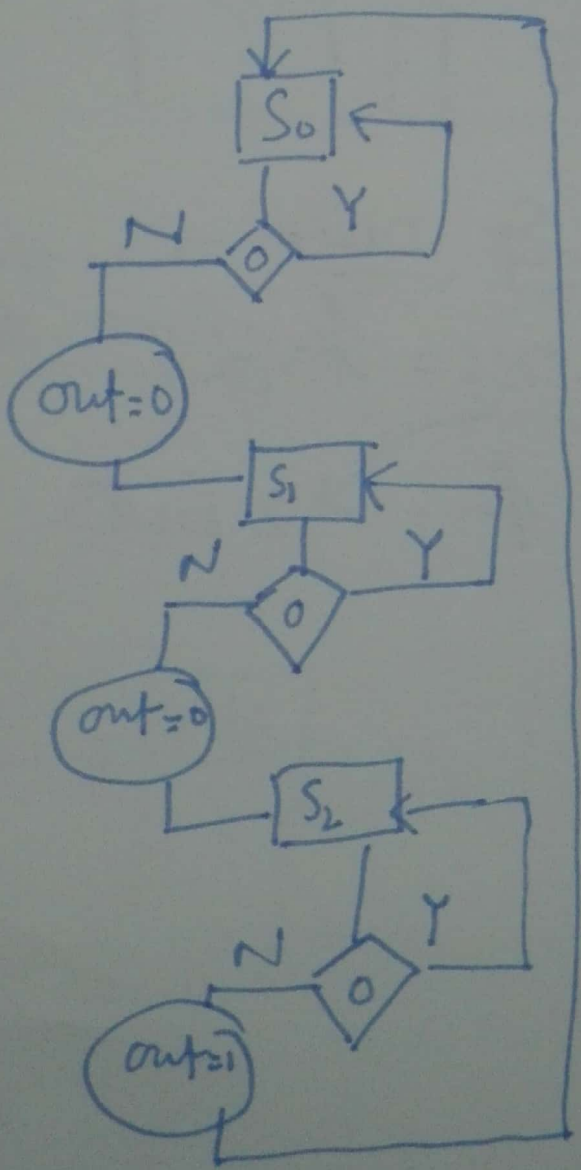
SM (HR)

Digital System Designs:-

FINALS

ASM:





CS	In	NS	Output
00	0	00	0
00	1	01	0
01	0	01	0
01	1	10	0
10	0	10	0
10	1	00	1

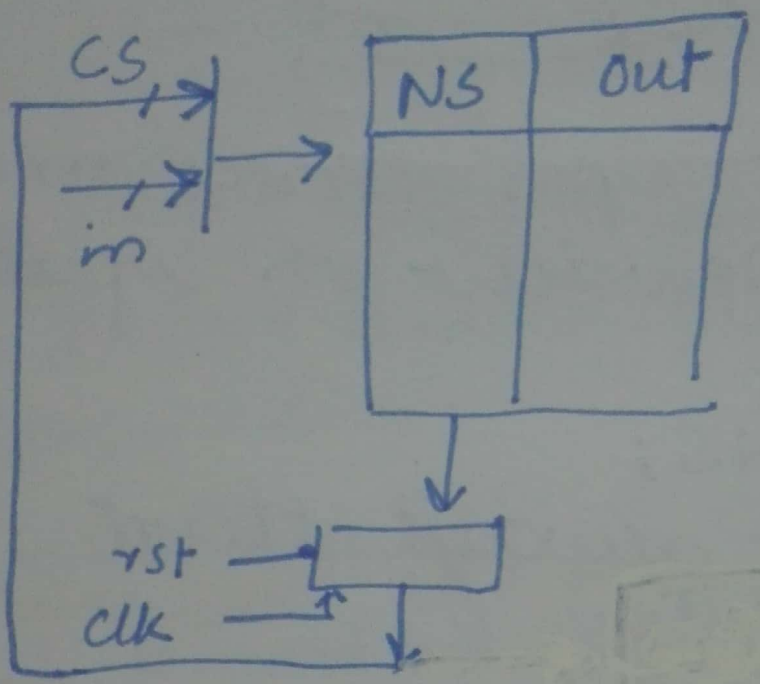
Bits of CS & NS:-

No. of states = 3

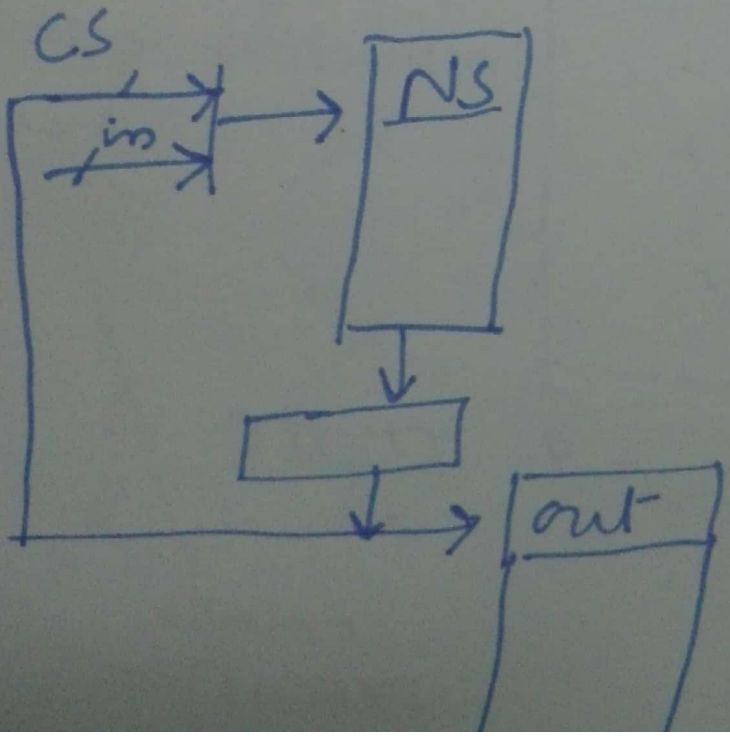
$2^{(2)} = 4$ → 2 bits

DSD

Microprogrammed SM:-

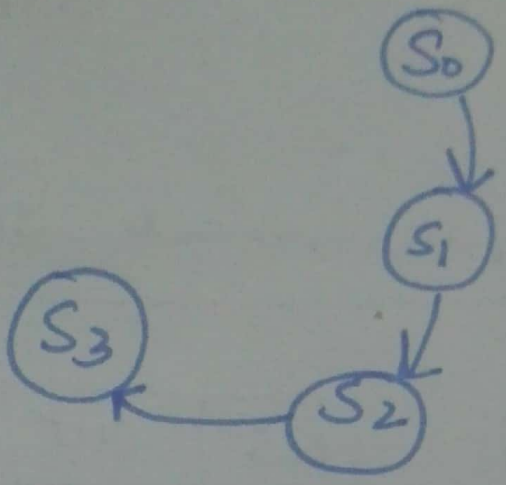


mealy



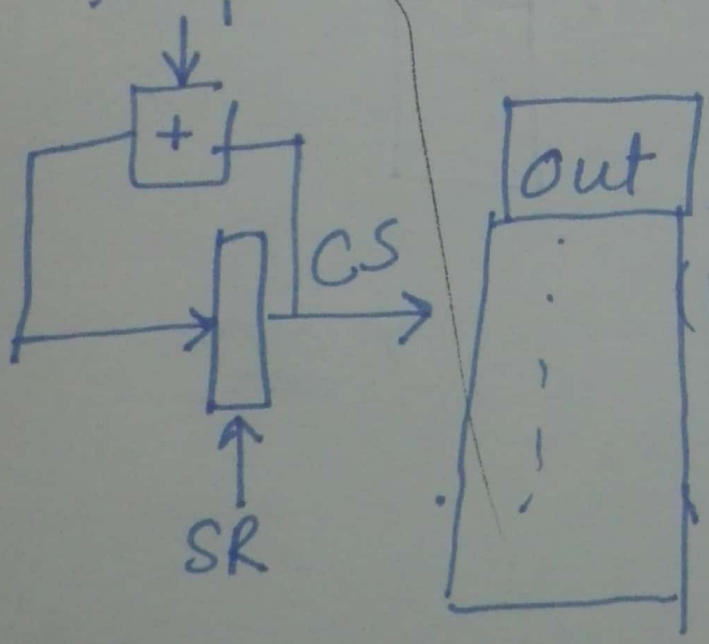
moore

Another Case :-



dependent upon states only not dependent on input.

CS ;
CS + 1 = NS ;
⇒ CS will always start from zero.



counter based state machine.

controls execution

Decode units:-

Interprets instruction.

Custom Processors Design: (ISA)

CPU Instruction set Architecture.

set of instructions:

① ALU instructions (+, −, ×)
② load, store, branch, jump, subroutine.
↓
function calling.

Basic Types of instructions:-

⇒ ① ALU based ② Control Based. (if/else, loop, goto)

① ALU based processor:-

Sequential

- $R_i = R_j + R_k$
  $R_i = R_j - R_k$
  $R_i = R_j \mid R_k$
  $R_i = R_j \,\&\, R_k$
  $R_i = R_j$

R → Register file. (set of

$i, j, k \in \{0, 1, 2, 3\}$

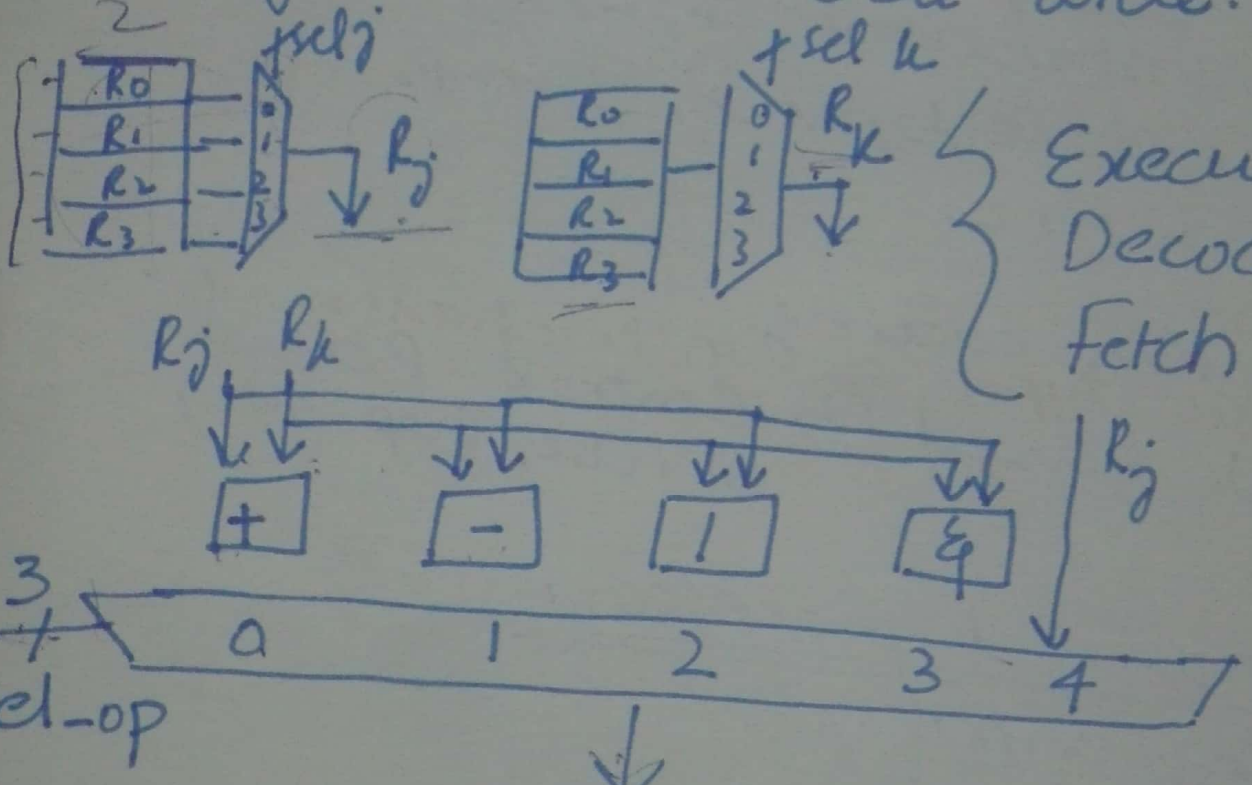
$R_i \in R_0, R_1, R_2, R_3$

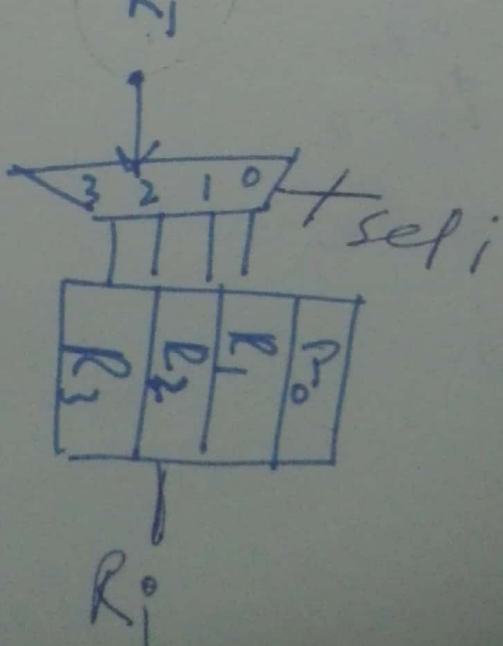
R_j →

R_k →

{ These have four Register each

* all registers are 8 bit wide.

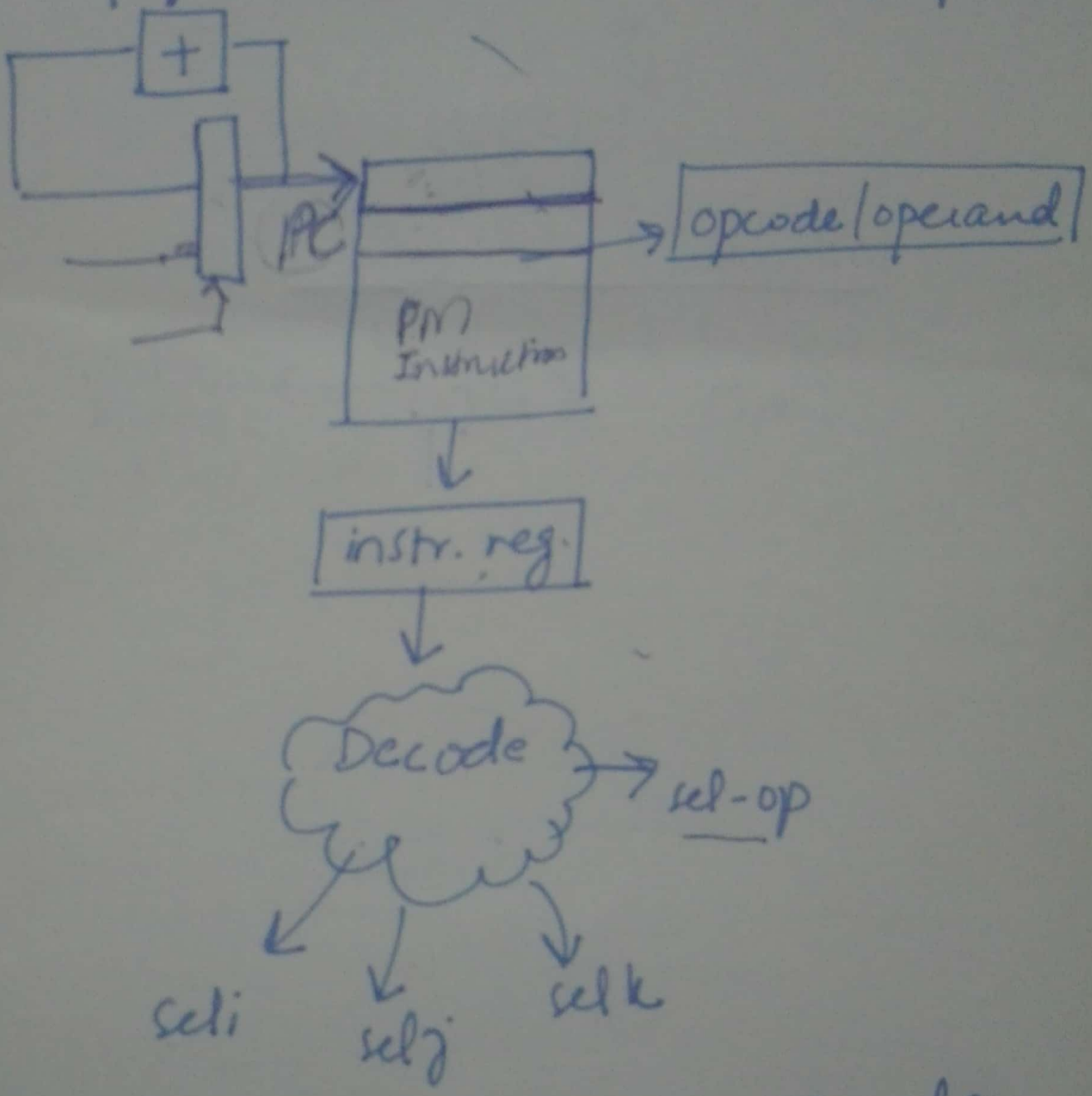




National Logistics Cell

SM (HR)

Control signals:- seli, selj, selk, op-sel



①	$R_i = R_j + R_k$	0
②	$R_i = R_j - R_k$	1
③	$R_i = R_j \mid R_k$	2
④	$R_i = R_j \& R_k$	3
⑤	$R_i = R_j$	4

sel-op = = opcode.

* opcode → no. of instrs. (5) → b
* operands → R_i 2bits
→ R_j 2bits
↓
R_k (2)bits

opcode	R_i	R_j	R_k

3bits 6 bits → address stored here.

Dec, 2017.

Lecture #: 21

Customer processor Design:-
(Sequential manner)
- ALU based instr.
- Goto label

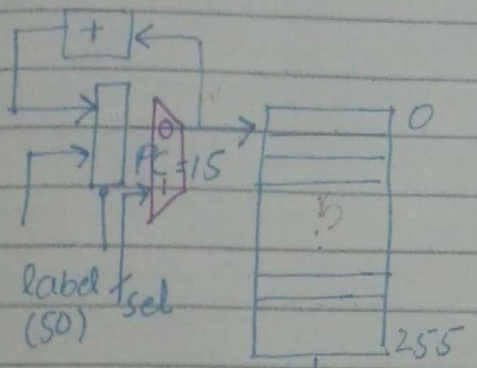
$R_i = R_j$ op R_k
$R_i = R_j$ op R_m or R_k
$R_i = R_j$ op const
R_i = const.
$R_i = R_k$

⇒ ALU based
⇒ Goto Label
(unconditional jump)

⇒ $R_i = R_j$ op const

| opcode | R_i | R_j | Const |

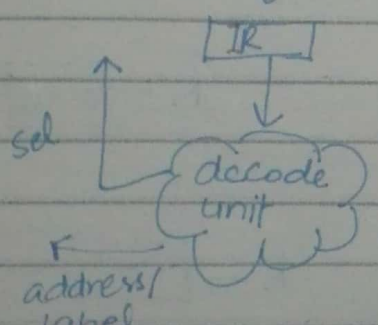
* Control Unit



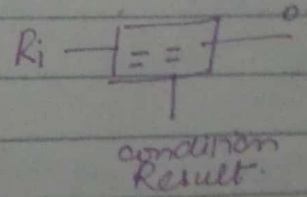

```
If (opcode == Goto)
    sel = 1;
else
    sel = 0;
```

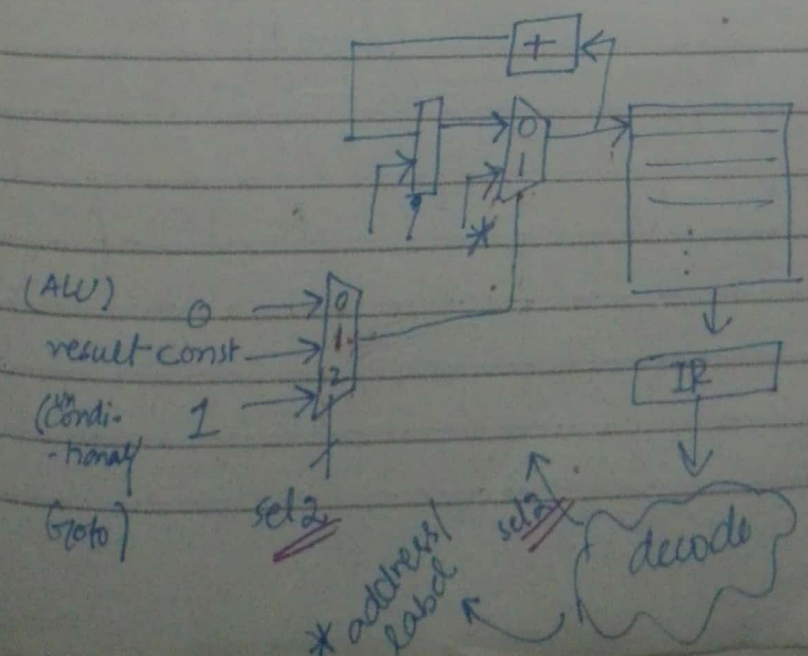
| opcode | address/label |
| 3 bits | 8 bits |

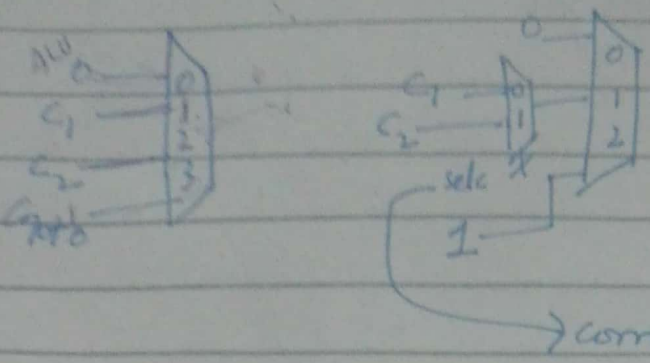
* opcode of Goto will be there.



- Conditional Jump (If /else).







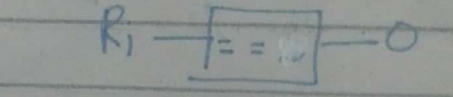
+	0
-	1
|	2
&	3
max	4
Goto	5
if 1	6
if 2	7

```
if (op < 5)
    sel2 = 0
elseif (op = 5)
    sel 2 = 3
else if (op == 6)
    sel 2 = 1
else sel2 = 2
```

---

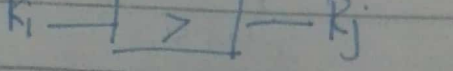
① if ($R_i == 0$) goto label
else normal

opcode	R_i	address/label



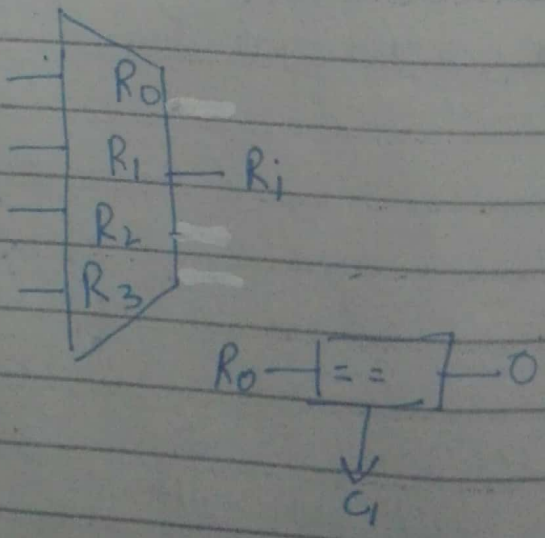
② if ($R_i > R_j$) goto label
else normal

opcode	R_i	R_j	address/label



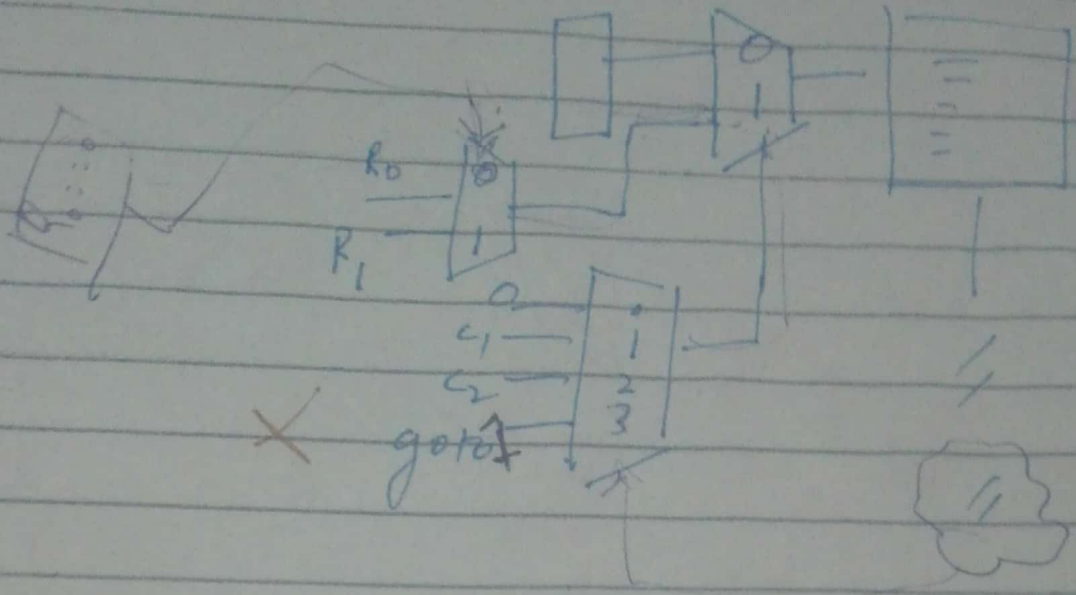
③ if ($R_0 == 0$) goto label
else normal.

opcode	address



④

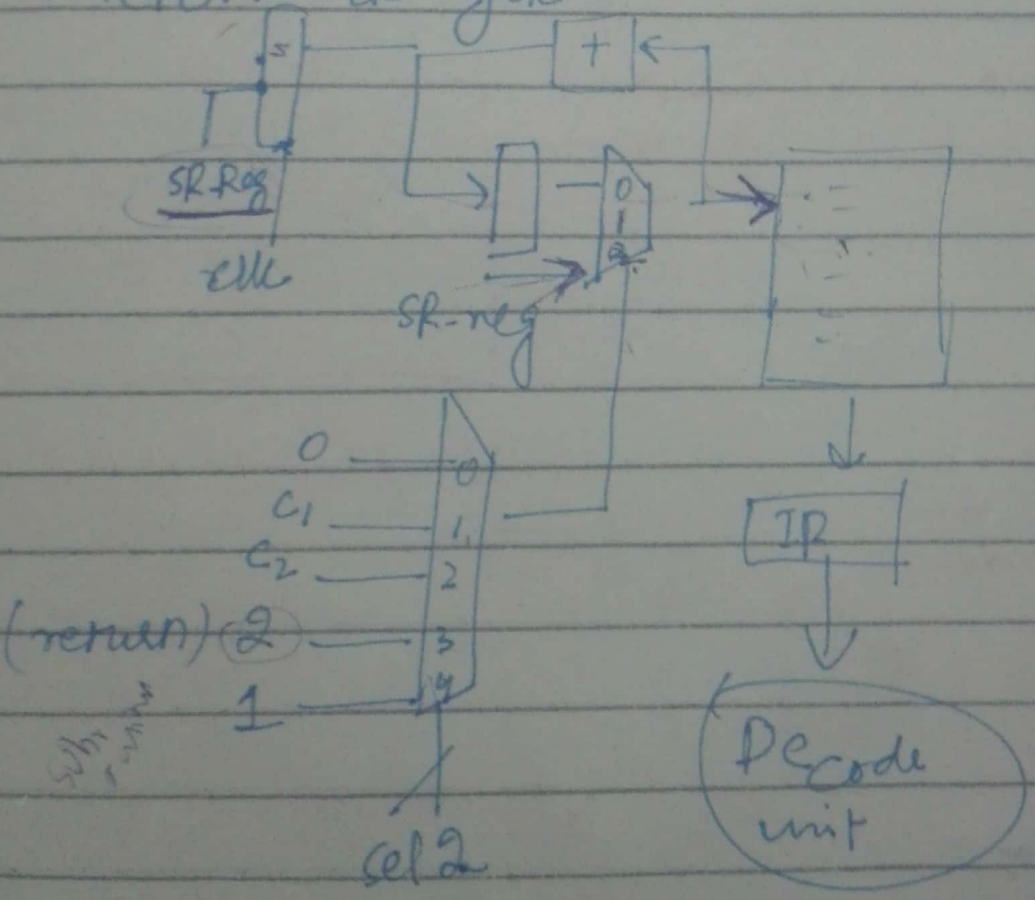
if ($R_0 == 0$) goto R_0 else goto R_1



Subroutine Call: (includes return function).

⇒ same as goto

⇒ return as goto.

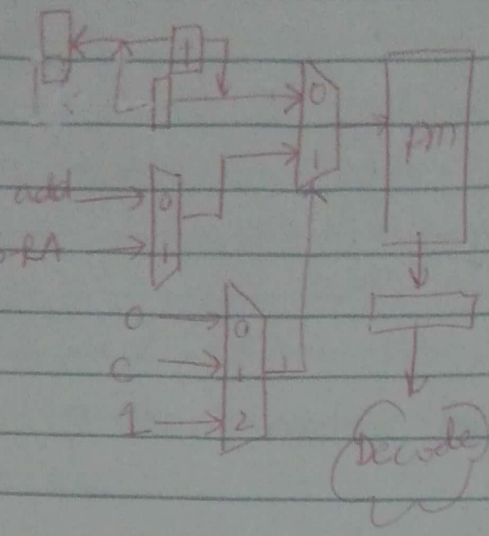


⇒ if nested loop, then double subroutines, use two registers and implement LIFO.

1st January 2018 Lecture #22

Subroutine call @ label return

| opcode | address/label |



sel_a = { 0 ; 1 op = return }

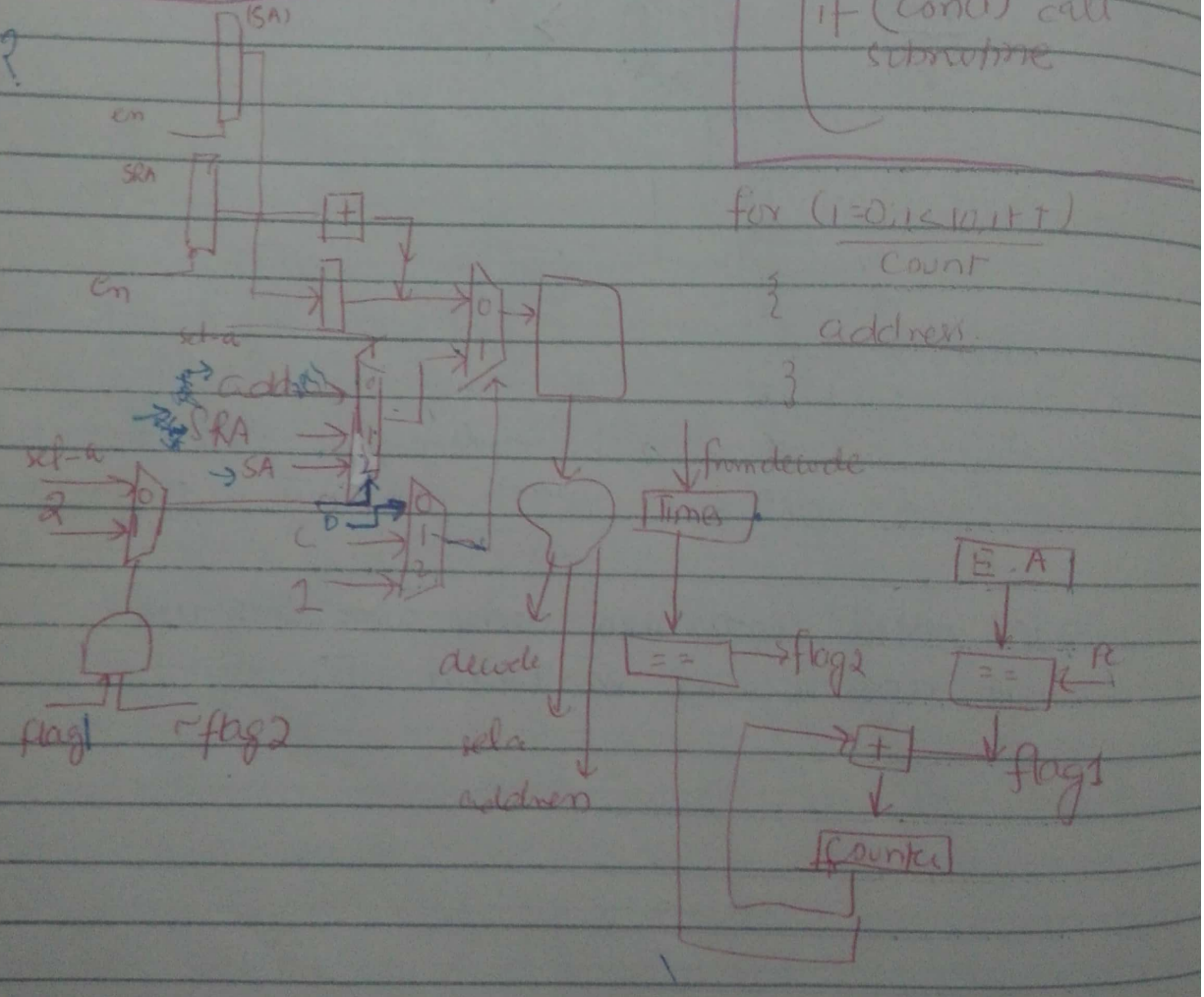
sel = { 0 ALU ; 1 cond. ; 2 content of SR (return) }

en = (op == SR)

How?

SR call @ label
if (cond) call subroutine

FOR time/count .. (count, add address)



for (i=0; i<10; i++)
{ count
address
}

2?

flag → iteration complete
loop incomplete

3rd Jan 2018: Lecture #

C.P.D:-

1- ALU 2- Goto 3- Cond. 4- Subroutine 5- Loop.

[opcode | operands] → includes many other things

choose format that has max. length.

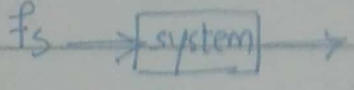
[opcode | count | address] → for loop

— x — x —

★ $f_s \leq f_{sys}$

→ rate of input

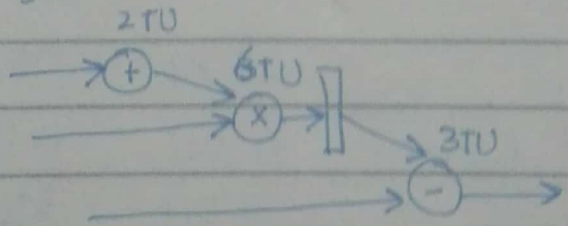
→ rate of system

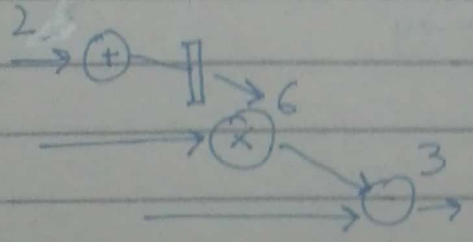


f_{sys}

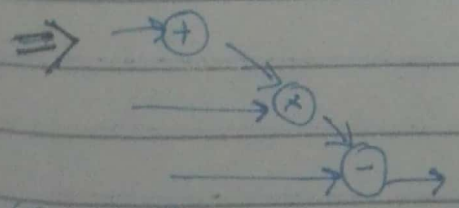
★

input fast ↷ $f_s > f_{system}$ → slow



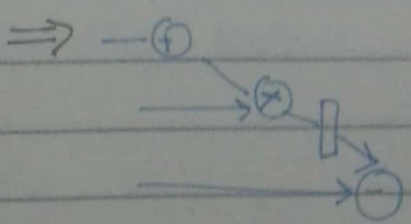


max. time of path / critical path.



CPU = 11 8TU

$f_{sys} = \frac{1}{11}$



CPU = 8TU

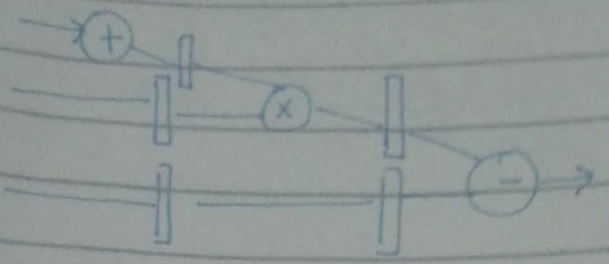
$f_{sys} = \frac{1}{8}$

I/P → reg

reg → reg

reg → out

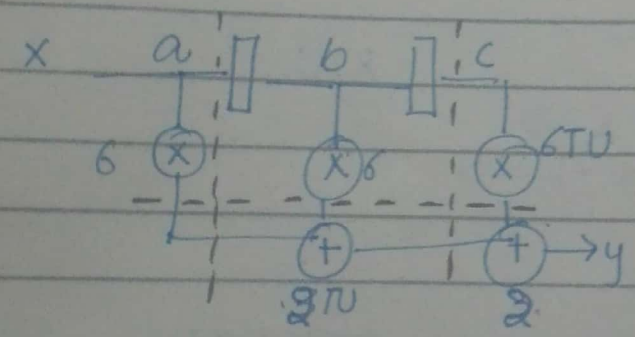
In first cycle only addition occurs.



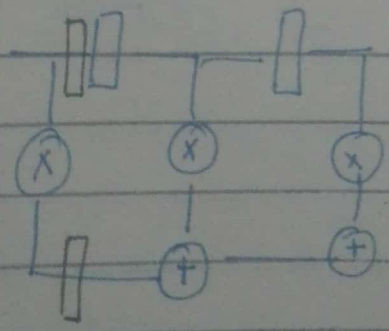
In second register saves value in second reg.
In third we get output.
After 3 cycles, system acts normally.
⇒ delay is generally latency.
⇒ The # of cycles required for I/P to get resp. O/P = 2 clk cycles

⇒ The no. of registers = latency.
or cuts.

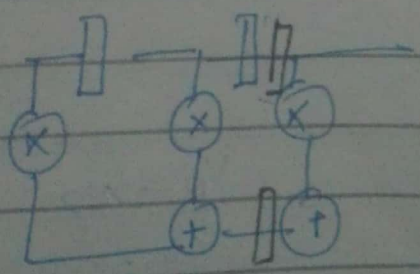
Feedforward cutset:-



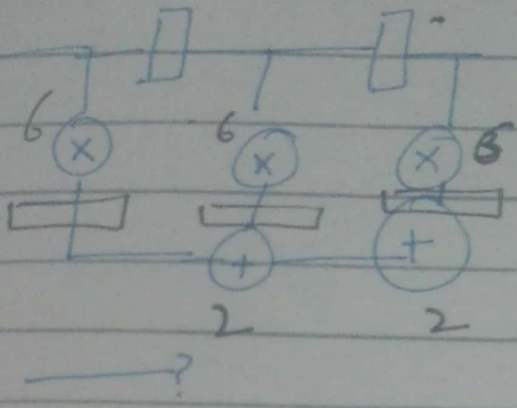
a → y 10TU
b → y 10TU
c → y 8TU



critical path = 10



CP = 8

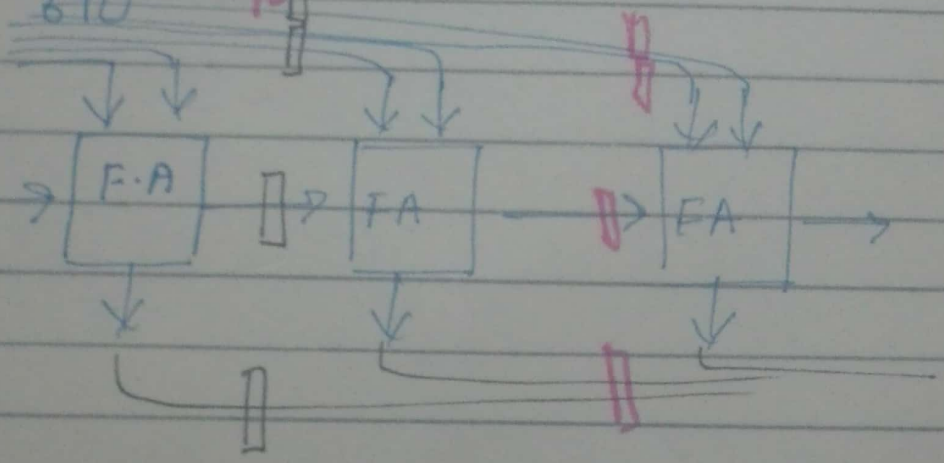


CP = 6 TU

* max resource breakage.
* 2-stage pipeline (1 cut / registers)

latency = 1 clk cycles.
CP = 6 TU
C.R = $\frac{1}{6TU}$ = f_{sys}

Exp#2



* Registers correspond to speed.
* more register more cost.

ALU Based Processor (sequential)

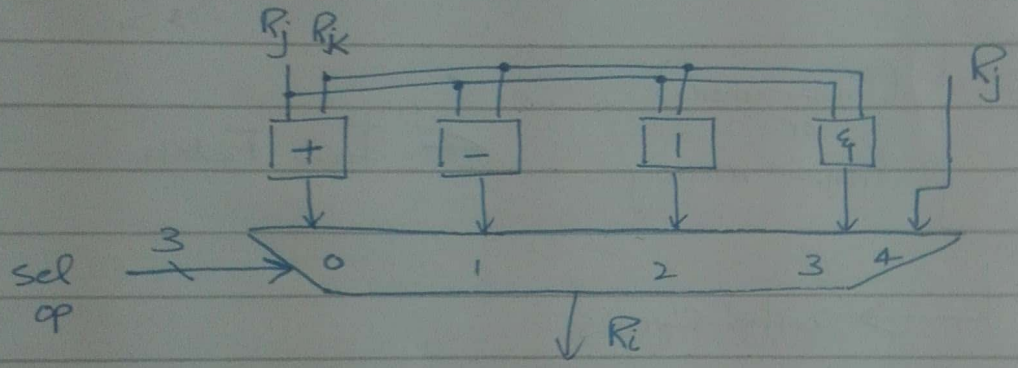
$R_i = R_j + R_K$
$R_i = R_j - R_K$
$R_i = R_j \mid R_K$
$R_i = R_j \,\&\, R_K$
$R_i = R_j$

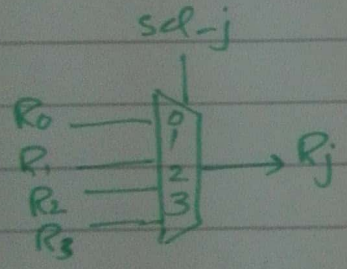
R → register file (set of registers)

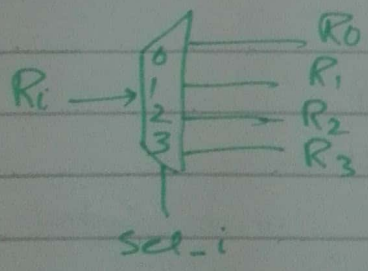
$i, j, K \in \{0, 1, 2, 3\}$

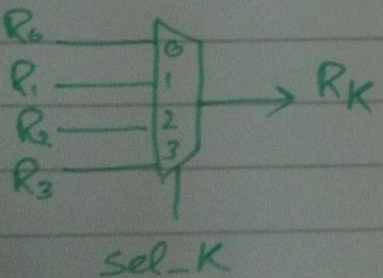
all registers are 8 bit wide

$R_i \rightarrow R_0, R_1, R_2, R_3$
$R_j \rightarrow$
$R_K \rightarrow$



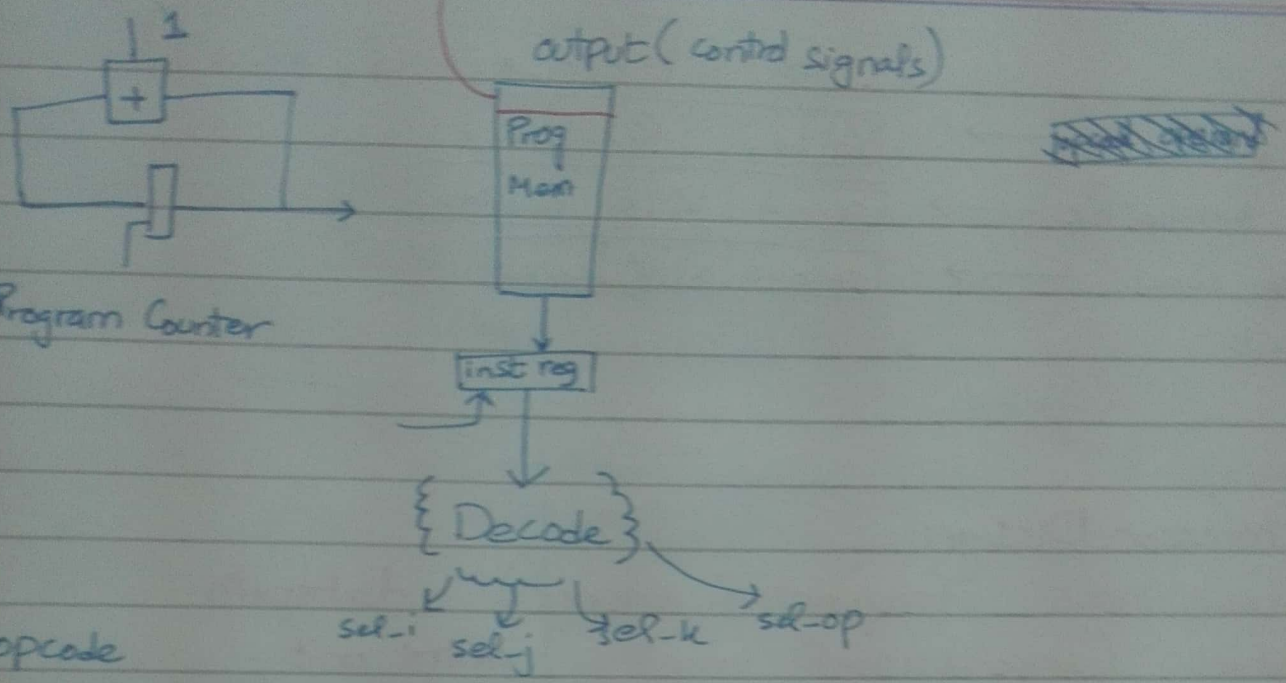




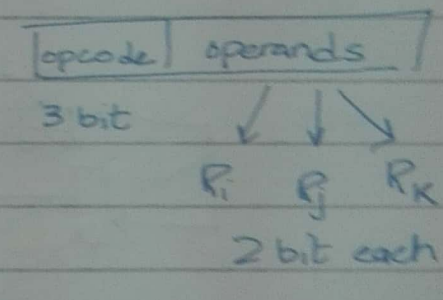


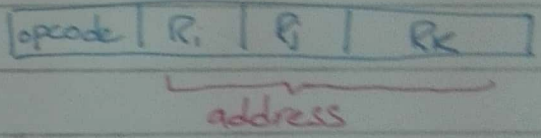
Control Signals: sel_i, sel_j, sel_K, sel_op

{sel_i, sel_j, sel_k, sel_op}



opcode	
0	$R_i = R_j + R_k$
1	$R_i = R_j - R_k$
2	$R_i = R_j \mid R_k$
3	$R_i = R_j \,\&\, R_k$
4	$R_i = R_j$





0–4 same

5 Goto label

(sequential manner)

op ∈ [+, -, *, |, &, ^ ...]

$R_i = R_j$ op R_K ⟶ | opcode | R_i | R_j | R_K |

$R_i = R_j$ op R_m op R_K ⟶ | opcode | R_i | R_j | R_K | R_m |

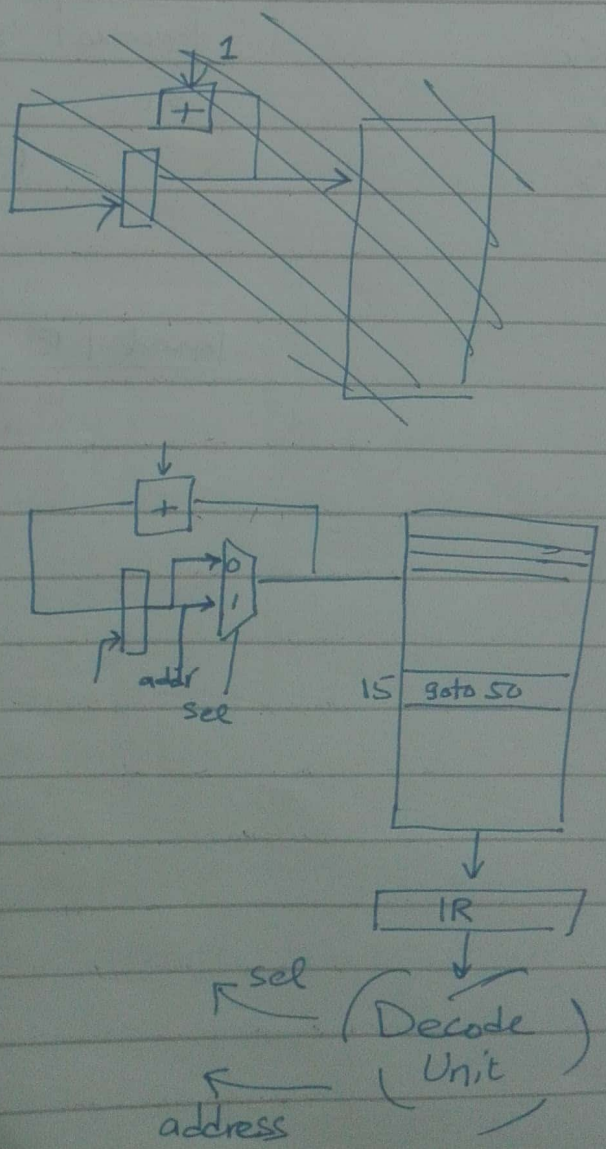
$R_i = R_j$ op const ⟶ | opcode | R_i | R_j | const |

R_i = const

$R_i = R_K$

Custom Processor Design

- ALU Based instr (sequential manner) datapath
- Goto label (unconditional jump) control unit



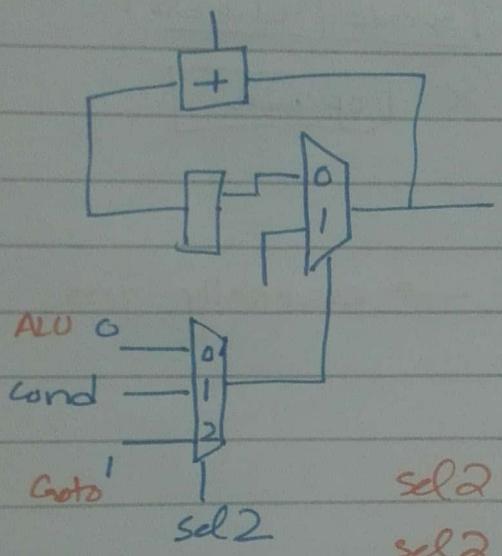
if (opcode == goto)
  sel = 1
else
  sel = 0

Conditional Jump (if else)

sel = 0 ALU

sel = 1 Goto

sel = cond result conditional



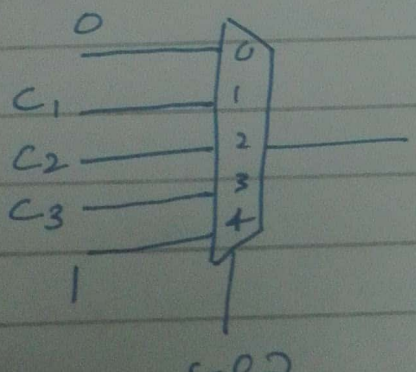

```
if (opcode < 5)
    sel2 = 0
else if (opcode == 5)
    sel2 = 3
else if (opcode == 6)
    sel2 = 1
else
    sel2 = 2.
```

sel2 = 0 ALU

sel2 = ~~Goto~~ 2 Goto

sel2 = 1 if/else

To incorporate multiple if elses



if $R_i == 0$ goto
else normal

opcode	R_i	address

if $R_i > R_j$ goto
else normal

opcode	R_i	R_j	address

~~if R_0 >~~

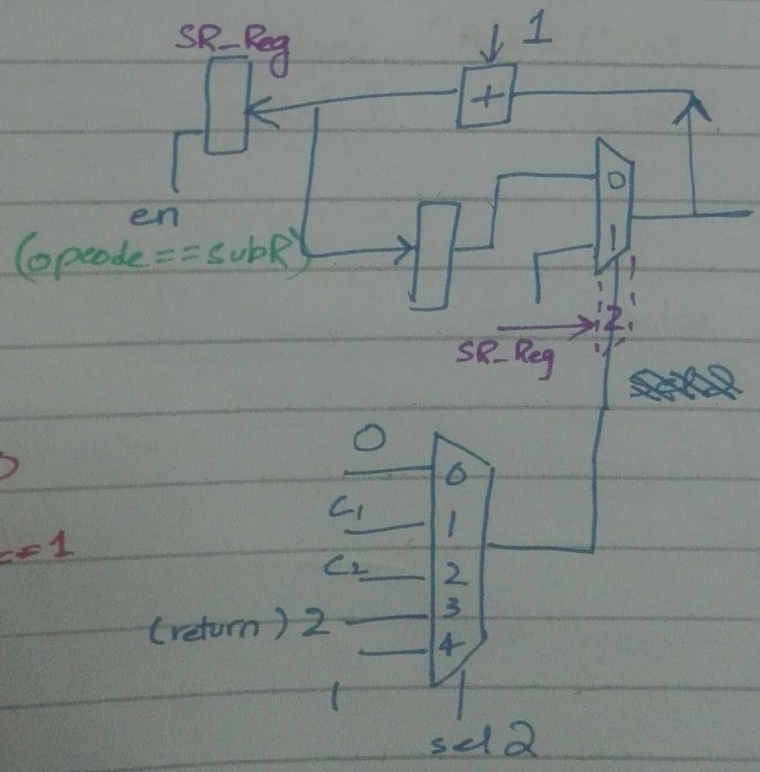
if $R_0 == 0$ R_i ✗

opcode	address

if $R_0 == 0$ goto R_0 address ✗

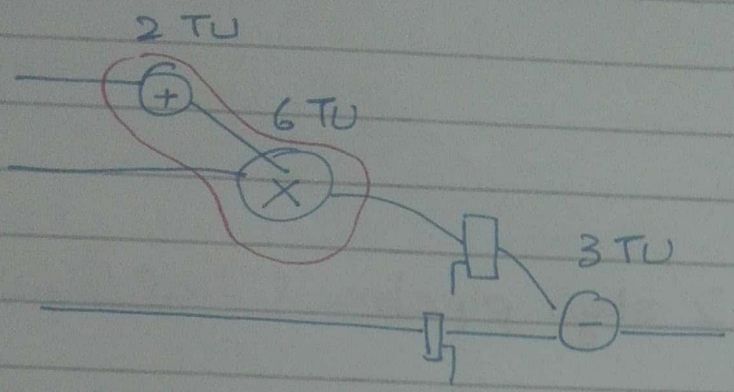
opcode

Subroutine Call. (function) → essentially goto
return → essentially goto



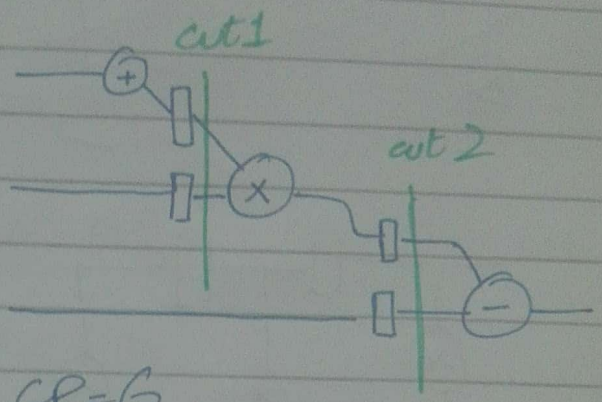
if $C_1 == 0$
normal
else if $C_1 == 1$
jump

Slowest Path → Critical Path
in our design



CP = 8 TU

$f_{sys} = \frac{1}{8\ TU}$



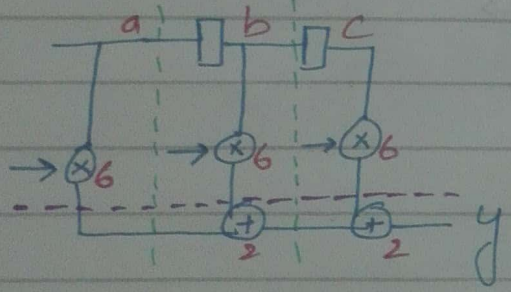
CP = 6

latency : 2 cycles

latency = # of clock cycles req for i/p to get respective o/p

↳ no. of pipeline registers / cuts

feedforwad cutset

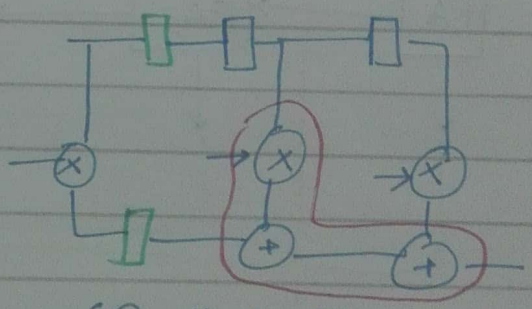


a → y 10 TU

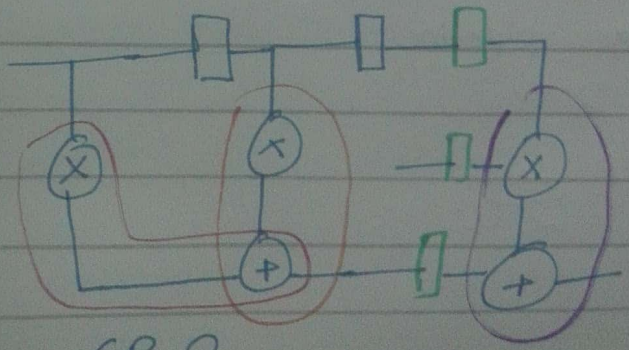
b → y 10 TU

c → y 8 TU

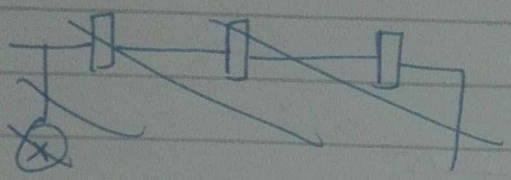
register breaks path

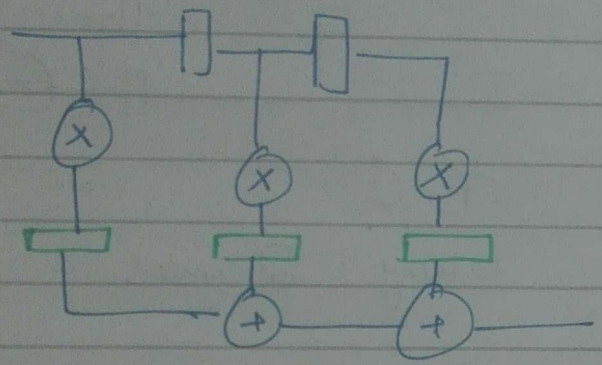


CP still 10 TU due to b → y



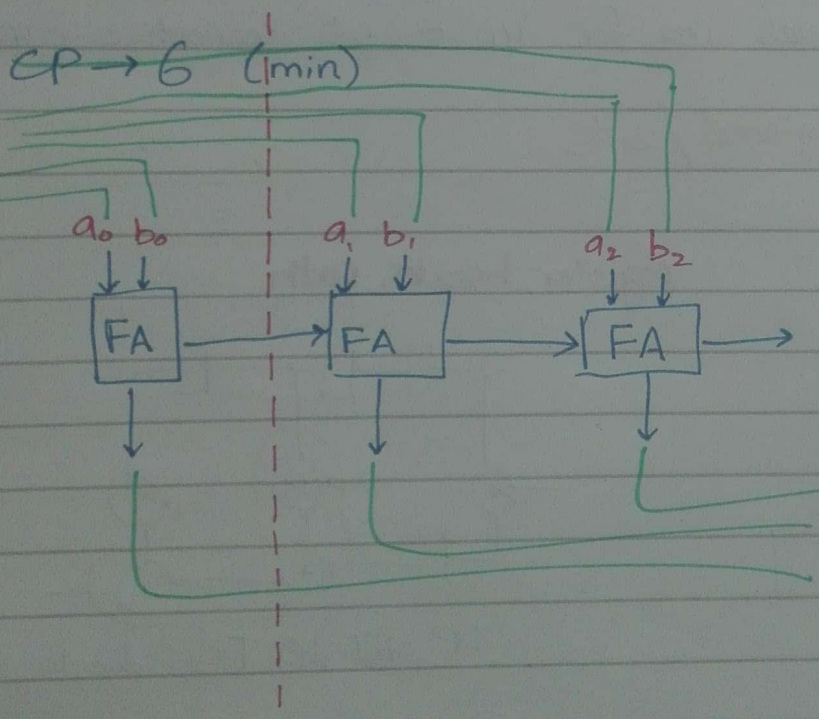
CP 8





2 stage pipeline (1 cut/register)

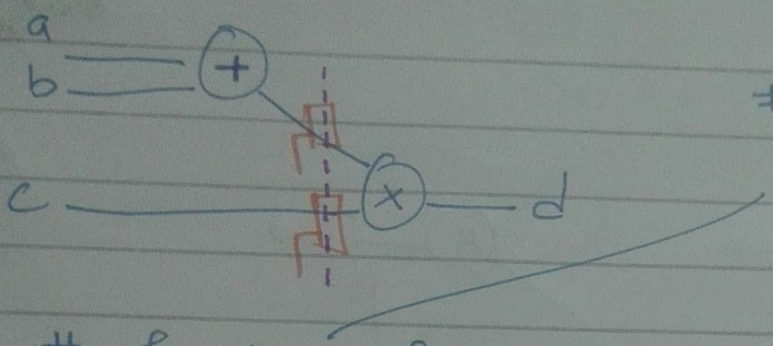
always cut b/w resources



2 stage pipeline

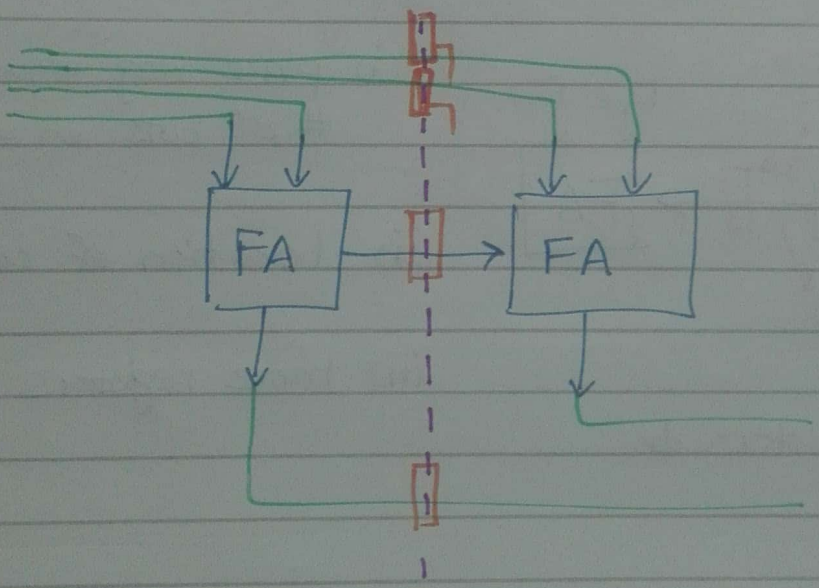
CPD = 2 TU

$f_{sys} = \frac{1}{2TU}$



# of delays from i/p to o/p (for all paths) must be same (equal to no of cuts)

a → d 1
b → d 1
c → d 1

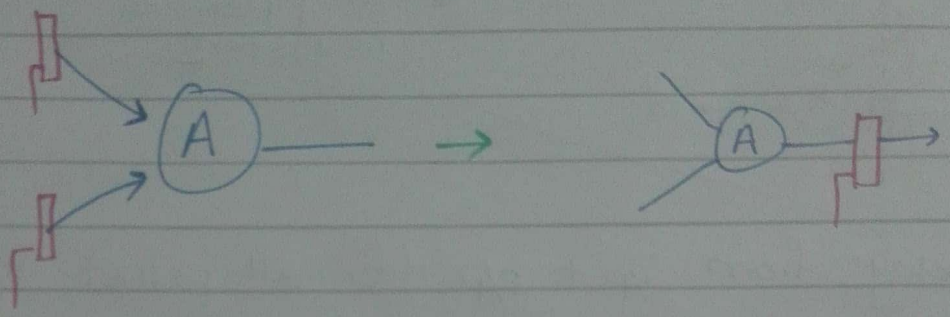


ignore algorithmic registers in count

(delay)

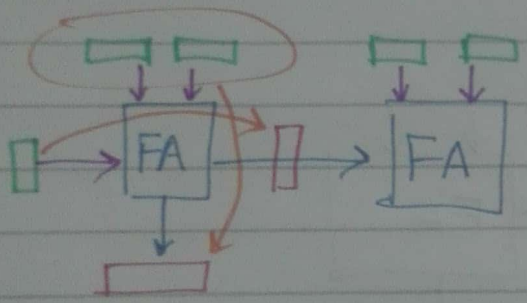
↓

Node transfer theorem



no. of delays = 1 no. of delays = 1

can only move across the node if registers present on all inputs

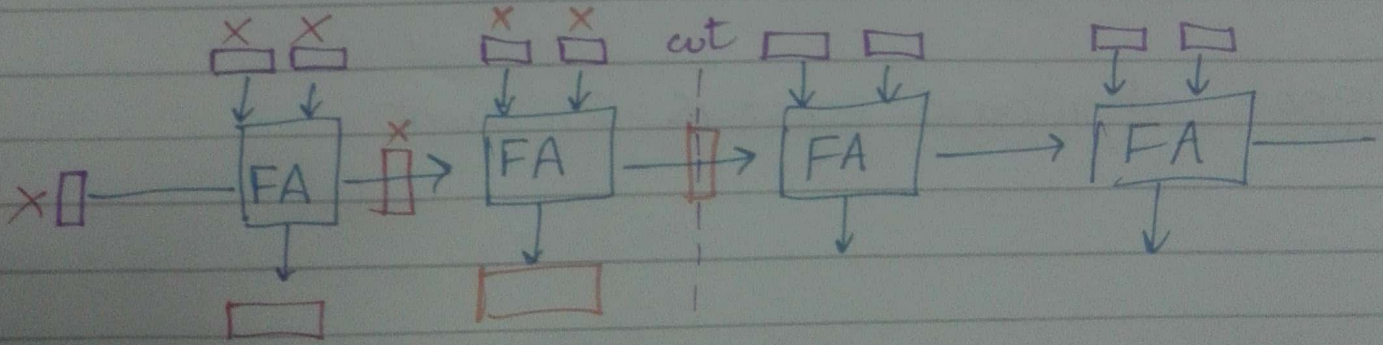


i) place reg equal to # of cuts on all inputs

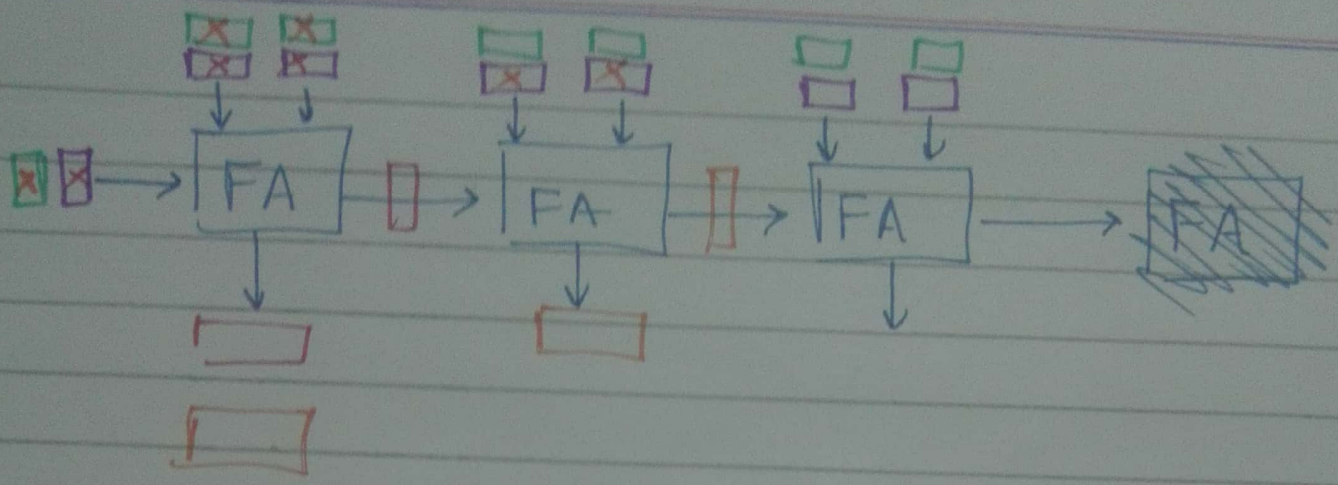
ii) location of cut

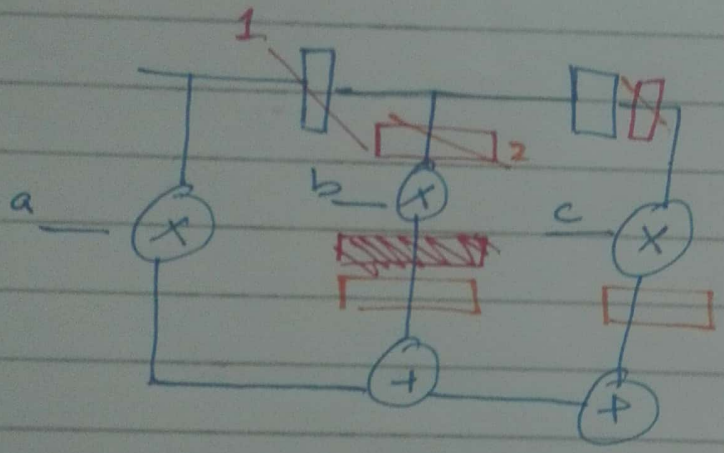
iii) move registers

move all i/p registers to ALL o/p



first move this then this

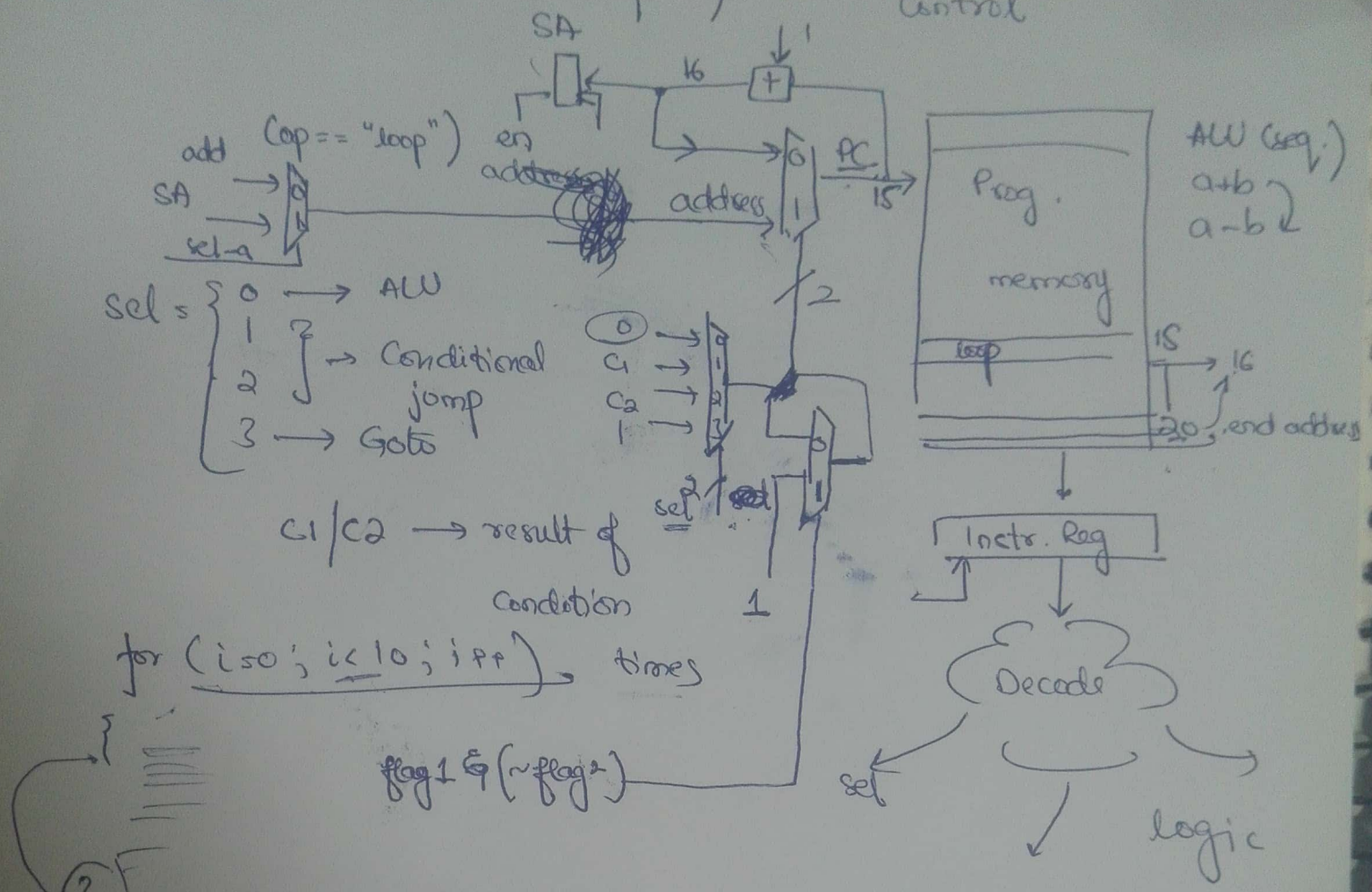

FA
FA
FA
FA


1
2
a
b
c

loop count, end address (for)

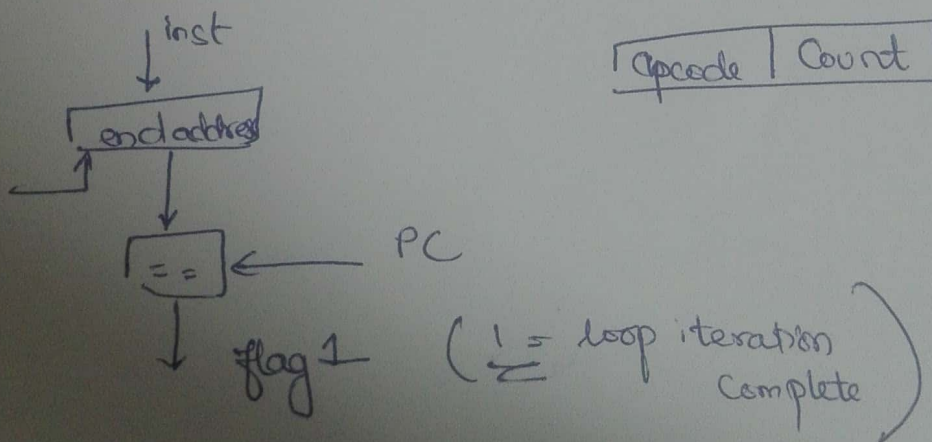
Processor

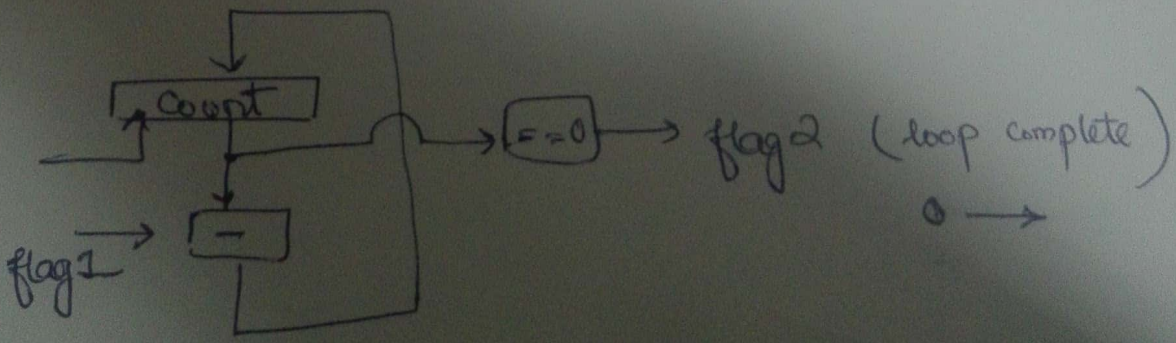
Control



Instr. format

Opcode	Count	End Add!

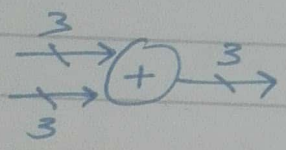


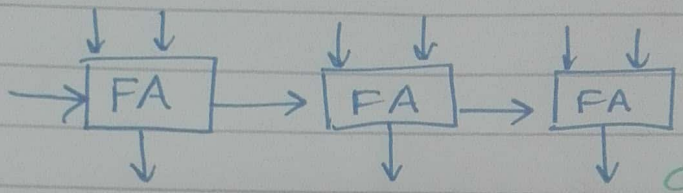


$f_s > f_{sys}$

Optimization of Basic building blocks

reduce critical path delay by optimizing operational units





3 full adder delay

carry ripple adder

Carry Select Adder

1 (0) 1 0	1 (0) 1 1	(0) 1 0
0 1	0 1	0 0
1 0	0 0	1 0 S cin=0
0	1	0 C
1 1	0 1	1 1 S cin=1
0	1	0 C

- Group Size: 2

2 full adder delay

12 full adders.
faster but more resources

1 1 0 0 1 0	S cin=0
0	C
1 1 0 0 1 1	S cin=1
0	C

can also have non uniform group size. (1-2-3)

10th January, 2018

Lecture #26

Conditional Sum Adder:-

Group size = 1

		0	1	0	0	1	1	1	0	
		1	0	1	1	1	0	1	1	
G1·S=1 delay=1	cin=0	1	1	1	1	0	1	0	1	S
		0	0	0	0	1	0	1	0	C
	cin=1	0	0	0	0	0	1	1	0	S
		1	1	1	1	1	1	1	1	C
G1·S=2	cin=0	1	1	1	1	0	1	0	1	S
		0		0		1		1		C
	cin=1	0	0	0	0	1	0	1	0	S
		1		1		1		1		C
G1·S=4	cin=0	1	1	1	1	1	0	0	1	S
		0				1				C
	cin=1	0	0	0	0	1	0	1	0	S
		1				1				C
G1·S=8	cin=0	0	0	0	0	1	0	0	1	S
		1								C
	cin=1	0	0	0	0	1	0	1	0	S
		1								C

→ muxes are so many.
→ 3 mux delay (⋆?)

cin	in1	in2	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
0	1			

Cin=1	in1	in2	S	C
1	0	0	10	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S_0 = in1 \wedge in2$

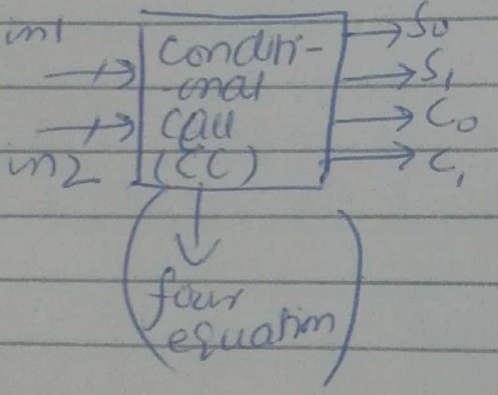
$C_0 = in1 \& in2$

$S_1 = in1 \sim \wedge in2$

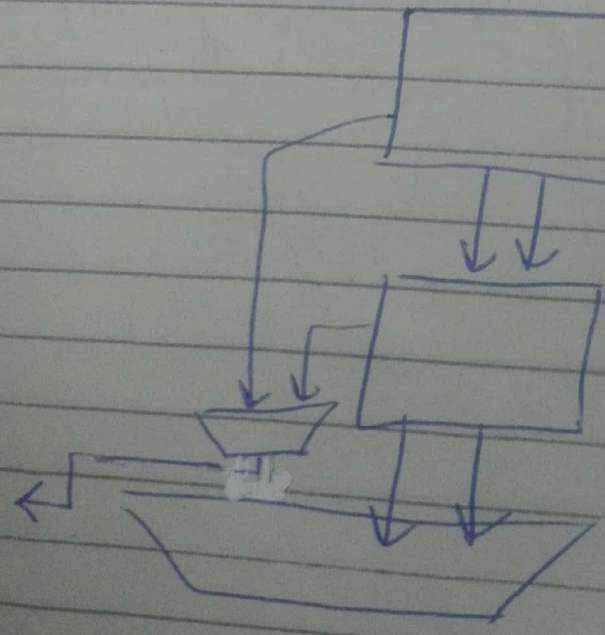
$C_1 = in1 / in2$

where in this one only one gate delay because four equations execute in parallel

Full adder uses two gate delay as two XORs are used.



Carry Select



* Trade off b/w area (Resources) & speed.

Carry-LookHead-Adder :-

→ Relatively less Complex
→ more good result.
→ it can actually calculate the carry.
→ truth table of full adder.

	in1	in2	c_i (current)	sum	c_{i+1}/cout (next)	
delete	0	0	0	0	0	whatever cin cout always 0
	0	0	1	1	0	
propagate (P_i)	0	1	0	1	0	dependent on cin
	0	1	1	0	1	
	1	0	0	1	0	
	1	0	1	0	1	
generate (g_i)	1	1	0	0	1	whenever cin → cout = 1
	1	1	1	1	1	

Signals to propagate / generate

$P_i = in1_i \wedge in2_i$ } either propagate/generate.

$g_i = in1_i \;\&\; in2_i$ }

$cout = C_{i+1} = (P_i \;\&\; C_i) \mid g_i$

for generate { $(0 \;\&\; C_i) \mid 1$
$0 \mid 1 = 1$

for propagate { $(1 \;\&\; 0) \mid 0 \rightarrow$ $1 \mid 0 = 1$

$C_{i+1} = P_i C_i + g_i$

for $i=0$

$C_1 = P_0 C_0 + g_1 \rightarrow 1 \cdot 0 + 0 = 0$ ~~for later~~ ?)

for $i=1$

$C_2 = P_1 C_1 + g_1$

$C_2 = P_1 C_1 + g_1$

$C_2 = P_1 (P_0 C_0 + g_0) + g_1$ → likewise for others.

2 gates @ max.
2 gate delay.

× ×

CLA contd.

Calculate carry (CLA)

4-F.A { C_3 C_2 C_1 C_0 → calculated from process

0 1 0 0

1 1 0 1 → to add them we use four F.A

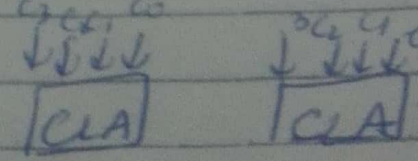
C_4

→ carries from CLA

→ one full adder delay.

C_{in} C_0 C_4 C_3 C_2 C_1 C_0 F.A

CLA CLA CLA CLA → sum



for next block

$C_1 = P_0 g_0 + C_0$ | $C_5 = P_4 C_4 + g_4$

C_2

C_3

C_4

$C_4 = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0 + P_3 P_2 P_1 C_0$

$C_4 = P_0 C_0 + G_0$

$C_8 = G_1 + P_1 C_4$ → intermediate carries are calculated by box.

$C_{12} = G_2 + P_2 C_8$

$C_{16} = G_3 + P_3 C_{12}$ → because sum ones are complex

→ C_4 C_8 C_{12} are @ start so we should do something about d^-

$P_0 C_0 + G_0$ ←

This thing is done @ last large block

$$C_{i+4} = G_i + P_i C_i$$

15th Jan 2018. Lec # 2&7

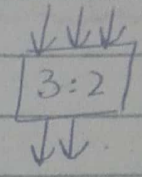
Fast multipliers / Multioperand Addition

- CARRY RIPPLE
- Carry select
- Conditional sum
- Look ahead
- Carry Save Reduction
- Dual carry
- Wallace Tree.

Carry-Save adder:-

a * b → 4bits → overall.	10
a + b + c + d → 4bits → partial product.	5

(3 Inputs → 2 outputs)
Delay = 1 Full adder.




```
  0 1 0 0
  0 0 1 1
  0 0 1 0
  -------
  0 1 0 1   sum
  0 1 0     carry
```

1FA used.

Comp. (3:2) → started off with 3 and got 2
* if we started first with 2 numbers and mein result with the 3rd uses 8 full adders on the whole.

The final answer will be generated using any kind of full adde
* Total five full adder delay.

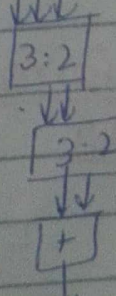
Example #2 a+b+c+d.

normally (12) → Proughmis (6)

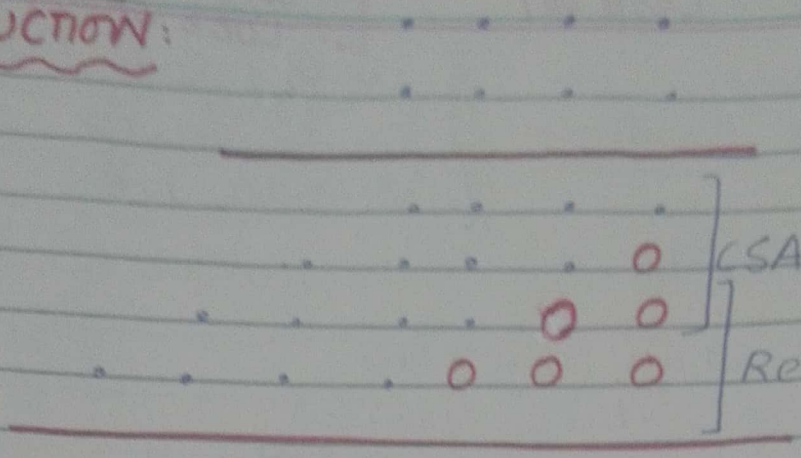
—x— x —

1) Carry Save Reduction
2) Dual "
3) Wallace Tree "

only works in 3 numbers

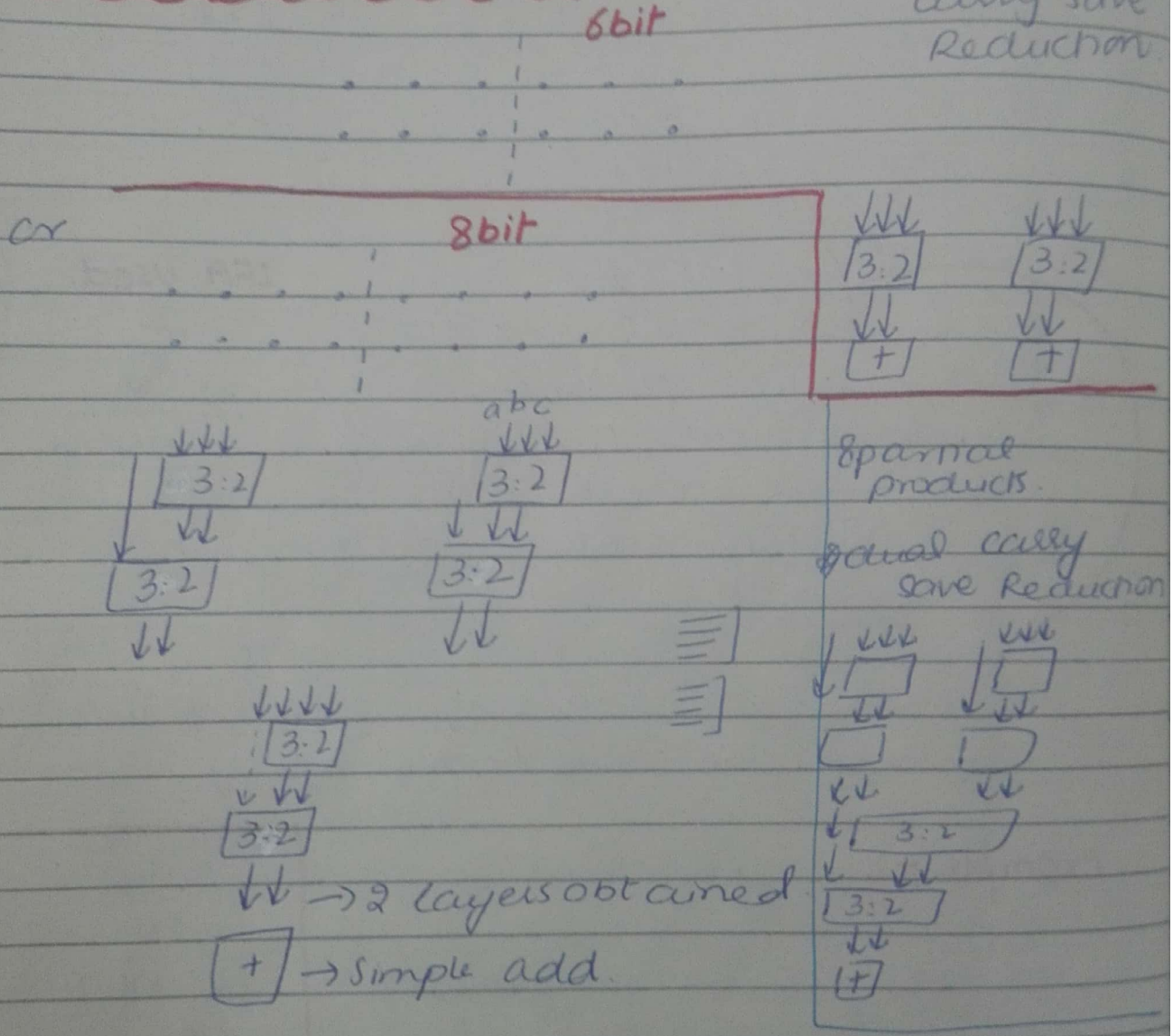


CARRY SAVE REDUCTION:

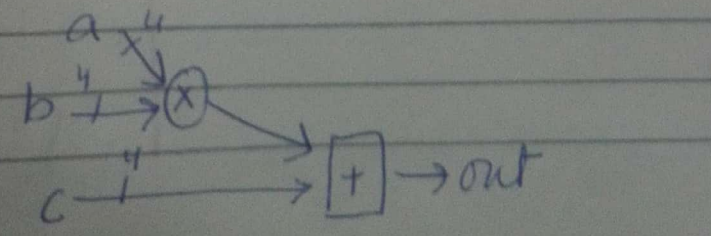


Dual Carry Save Reduction:-

6 bits → divide into 2 Carry Save Reduction

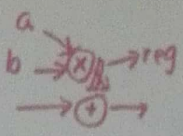


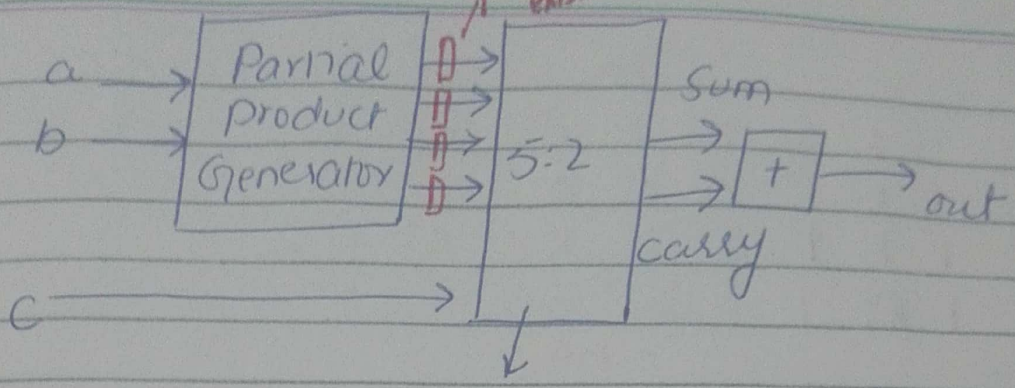
Transform the following using compression: using CRA



a $Q_{1,3}$
b $Q_{1,3}$
c $Q_{2,2}$

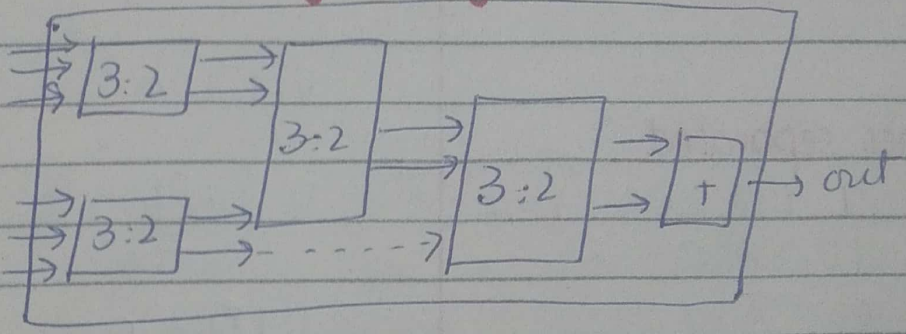
4 layers due to four P.P



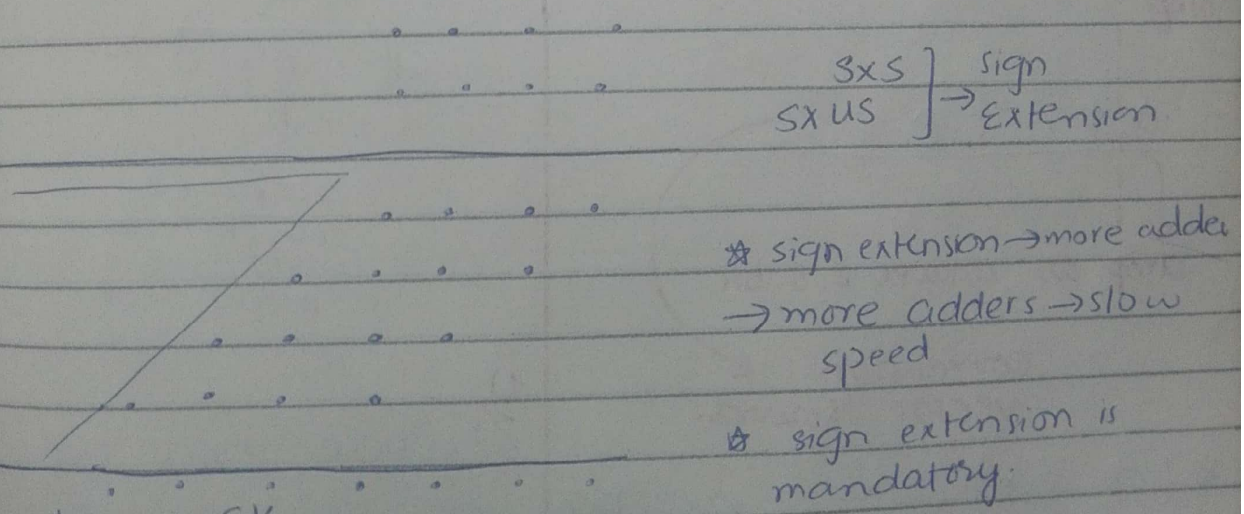


automatically uses any of the previous multipliers and it does not mentions that uses carry save adder.

→ mentioning Carry Save adder :-



Sign Extension Elimination :-



☆ sign extension → more adder → more adders → slow speed

☆ sign extension is mandatory.

⇒ first wrong inter. answer add some correction and it will make answer correct

⇒ through this method no correct intermediate ans but final is correct on hardware.

ignore this

```
            1 0 1    S
    1 1 1 1 1 0      S
  1 1 1 0 0 0
  1 1 1 0 0 1 x
  1 0 1 1 1 x x
```

2s comp
101 → 010
↓
011

```
          1 0 1
    1 1 1 1 1 0
  1 1 1 0 0 0
  1 1 0 0 1 x
  1 0 1 0 x x
          1
```

(B)

ones separated

MSB flip L-1

MSB ← 0 1 1 1 0 0

P.P
1 1 1 0 0 0
1 1 0 0 0 0
1 0 0 0 0 0

1s comp ← 0 0 0 1 0 0

1 0 | 1 0 1 0 0 0

drop as answer is 6 bit

correction vector found.

★ for 3x3 ; SxS ; correction vector will always be this

=> Correction vector

3) sign extend by 1

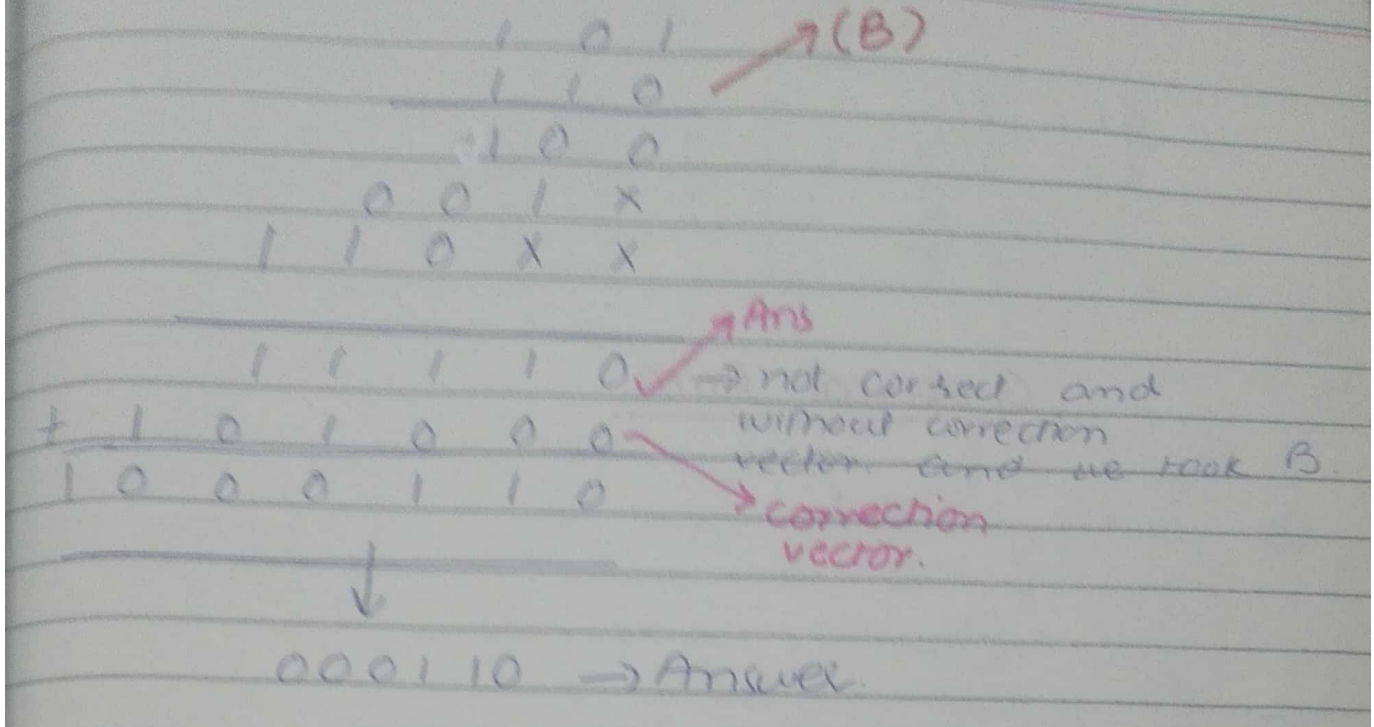
2) Flip MSB and place 1 @ location

Demo

000
↓
111000
↓
111100
↓
111100
↓
111100
1
1|000000
↓
becomes same number that we started

1) where there was 2's complement = 1s comp + 1 for last P.P

=> 1's complement and place 1 @ LSB



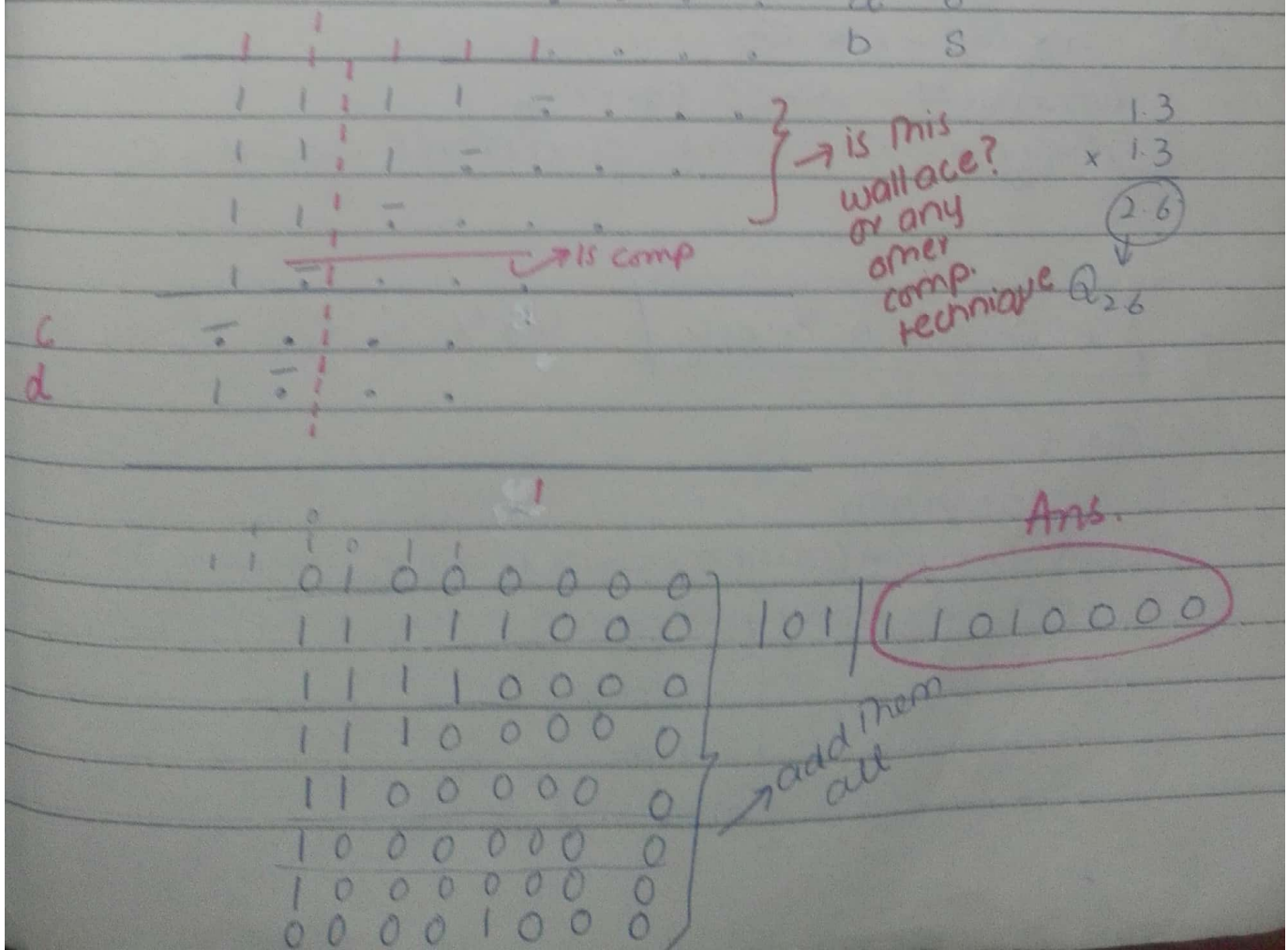
0001 10 → Answer

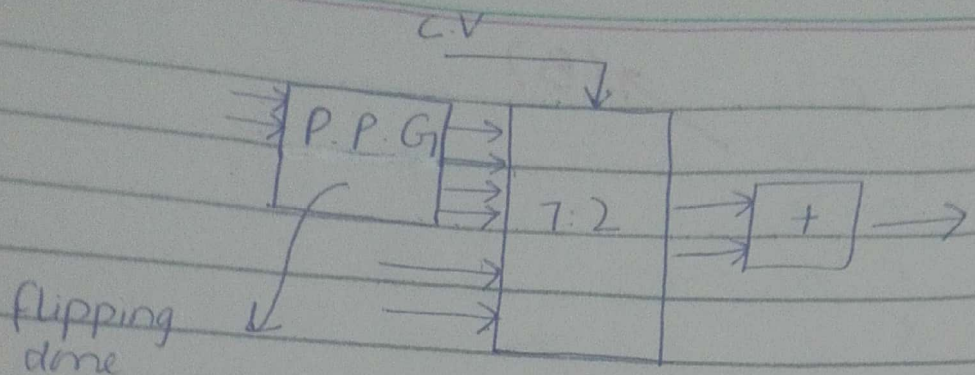
Example # 2.

Q_{n-3} Q_{n-3} $Q_{2\cdot 1}$ Q_{n-3}

$(a * b) + c + d$

Find correction vector for this.





Graphical Form

① $a*b - c*d$

↓

$CV_1 - CV_2$

② if formats are same then CV will be same.

③ $a*b + c*d$

↓

two separate C.V

④ $a*b - c*d$

formats are same

they both will subtract → C.V = 0

1	0	0	1	0	0	0	0

4×4 C.V

40% final.

1 verilog question.

1 Design Question (state machine) (most probably)

for our ease, forgetting processors, numbers are easy to gain

- processor
- design
- numerical based.