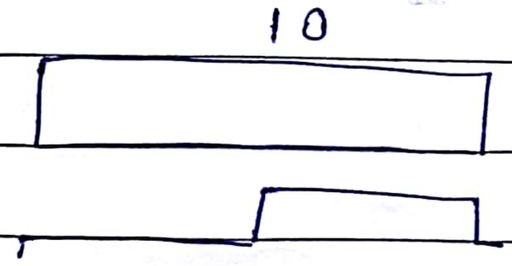
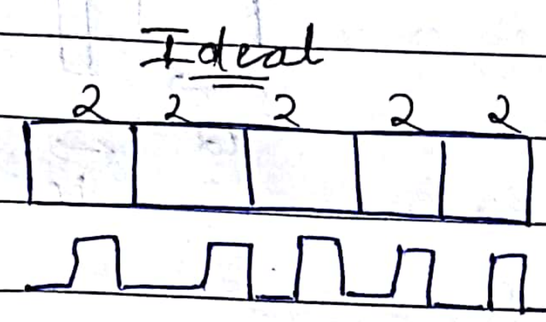
CEC 16
High Performance Computing

★ **Pipelined Processor :**

IPC → Instr per cycle
CPI → Cycles per instr



IPC = 1
CPI = 1



CPI = 5
IPC = $\frac{1}{5}$

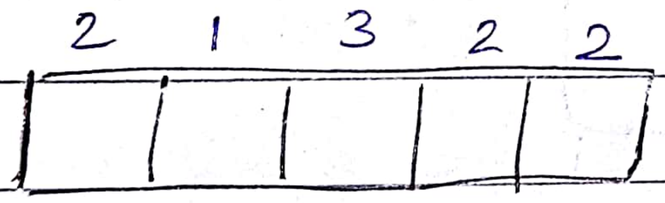
10^4 cycles for 10^4 instr
cycle time = 10
Total time
$= 10 \times 10^4$

$5 + 9999$
$= 10004$ cycles
10,000 instr

IPC ≃ 1 CPI ≃ 1
Cycle time = 2
Total time = 10004×2
$\simeq 2 \times 10^4$

∴ Speedup ≃ 5

~~Speedup ≃ $\frac{2 \times 10004}{10 \times}$~~



Non-Ideal

Uneven delay

S_1 S_2 S_3 S_4 S_5

0 — 2 — 3 — 6 — 8 — 10

output every 3s

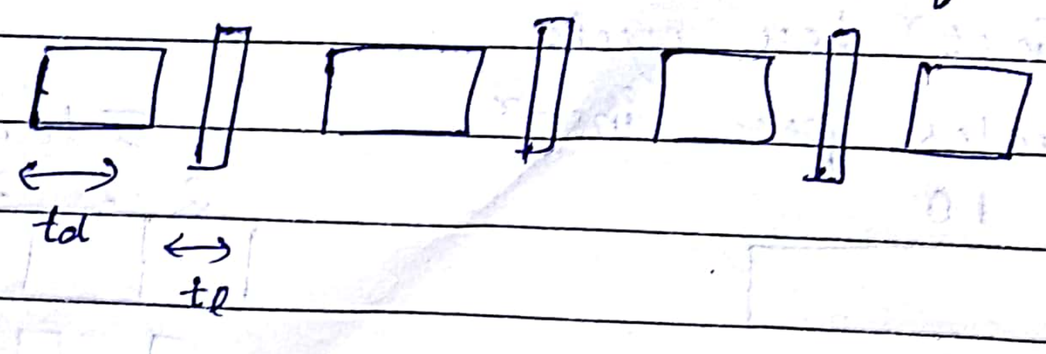
2 — 4 — ~~3~~ 6 — 9 — 11 — 13

4 — 6 — ~~7~~ 9 — 12 — 14 — 16

Note:

① We seperate different stages by latches. Otherwise the changes will affect the subsequent stages.

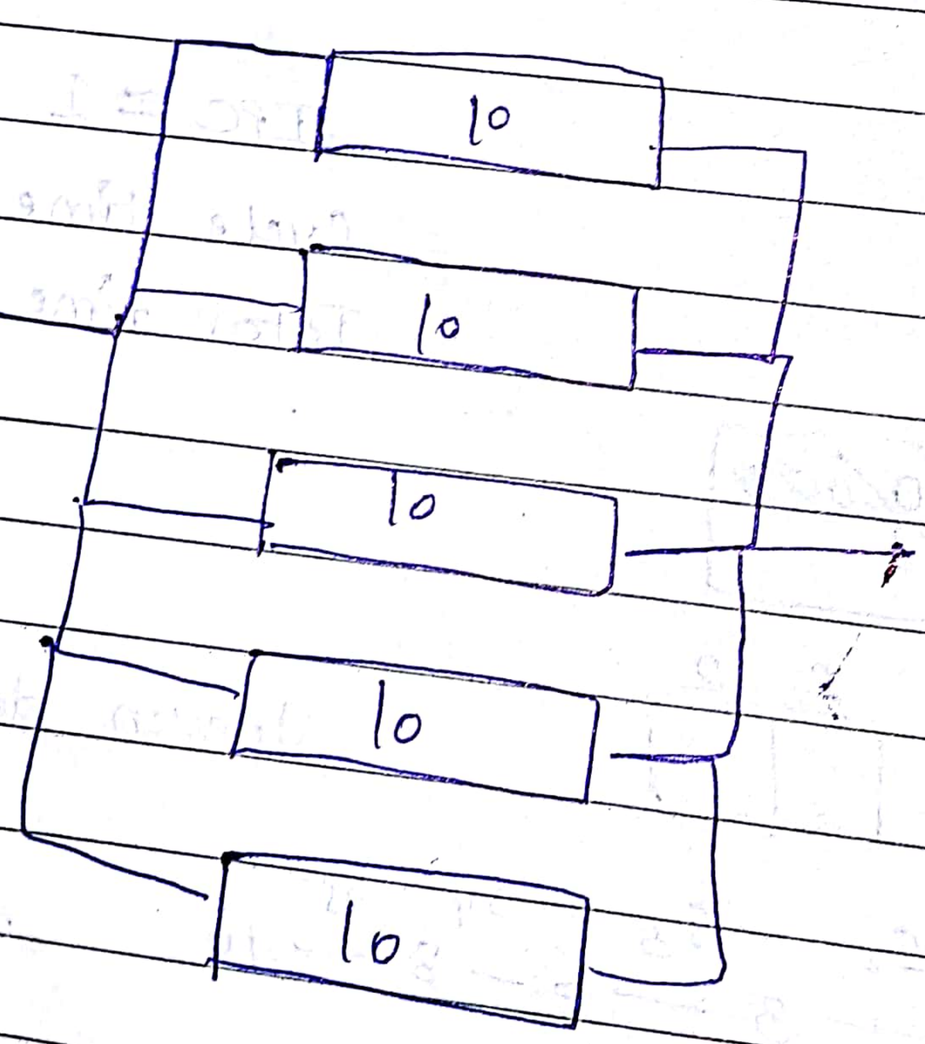
② Increase the number of stages till the delay due to latches is not greater than delay of each stage.



td → delay of each stage
tl → delay of latch.

③ As the pipeline depth increases (higher freq clock), more heat will ~~increase~~ dissipate.

☆



* **Dependency:**
  - Data Dependency
    - True (Flow)
    - False (Name)
      - Anti-dependency
      - output dependency
  - Control Dependency
  - Structural Dependency

I1 : add r_1, r_2, r_3
I2 : sub r_4, r_1, r_5

Example of True (Flow) data dependency
RAW - Read after write

I1 : add r_1, r_2, r_3
I2 : sub ~~add~~ r_1, r_4, r_5

Example of output dependency
WAW - Write after write

I1 : add r_1, r_2, r_3
I2 : sub ~~add~~ r_3, r_4, r_5

Example of Anti-dependency
WAR - Write after read

Note: All processors of pentium series are called superscaler processors.

~~Note~~ Name dependencies can be handled using register renaming. It can be implemented either in software (compiler) or hardware. There are set of registers called rename registers ~~can~~ hidden from Assembly language access. i.e., only compiler can access them.

★ → add r_1, r_2, r_3
sub r_4, r_5, r_1

RISC ISA

IF	ID	EX	MEM	WB

→ If we had r_6 instead of r_1 in 2nd instr, there will be no ~~dependency~~ dependency

	1	2	3	4	5	6
	IF_1	ID_1	EX_1	MEM_1	WB_1	
		IF_2	ID_2	EX_2	MEM_2	WB_2

1st instn after 5 cycles
2nd after another 1 cycle

Average CPI = 1

→ But in our case, we cannot fetch the values in 2nd instr before WB stage of 1st instruction.

	1	2	3	4	5	6	7	8	9
	IF_1	ID_1	EX_1	MEM_1	WB_1				
		IF_1	×	×	×	ID_2	EX_2	M_2	WB_2

→ Due to dependency, next result is delayed.

1st instr → after 5 cycles
2nd instr → after 4 cycles ∴ Penalty of 3 cycles due to dependency

Ques 10000 instr. 2000 pair of instr ~~have~~ are dependent on other.

4000 with dependencies
6000 → ~6000 cycles

1	2	3	4	5	6	7	8	9	10	
IF	ID	EX	MEM	WB						
	IF	X	X	X	ID	EX	MEM	WB		
		X	X	X	IF	ID	EX	M	WB	
						IF	X	X	X	ID

	W	W/o			
1st Instr →	5	5			
2nd " →	9	6	1 dep on 2	12	EX
3rd " →	10	7	3 dep on 4	13	M
4th " →	14	8		14	WB

★ No. of cycles for 1 instr :-

→ 1 + No. of Avg Penalty Cycles

→ $1 + \frac{3}{2}$ (∵ 3 penalty in every 2 instr)

★ Note:

Wait cycles are known as stall cycles which results in hazards.

★ Control Dependencies:

★ ~~Instruc~~

★ Instr Reordering :

add r_1, r_2, r_3 → 3 wait cycles
sub r_4, r_5, r_1

insert independent instrs btw these two instr
So, by the time add and independent instr execute, value of r_1 will be available for sub instr with no wait cycle.

★ Name dependencies : (No data flow but name dependency)

Anti WAR — Output WAW

Output dep

add r_1, r_2, r_3
sub r_1, r_4, r_5

↳ renamed to r_{11},
and all the instances to r_1 to r_{11}
then we can reorder sub to place it before add instr.

Anti dep

add r_1, r_2, r_3
sub r_3, r_4, r_5

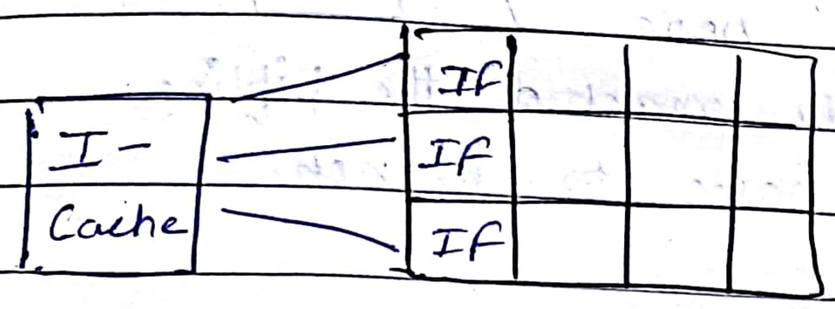
↳ all instances of r_3 after this are renamed, to say r_{33} in this and

Above method is known as Register Renaming

Register Renaming

Hardware → Superscaler processor → Multiple pipelines active at the same time

Software (Compiler) → VLIW
Very large instr word process



Fetching 3 Instr at a time

8086, 8088, 80186, 286, 386, 486 → Scalar (Single pipeline)

Pentium 1, 2, 3, 4, 5 → Multiple pipeline

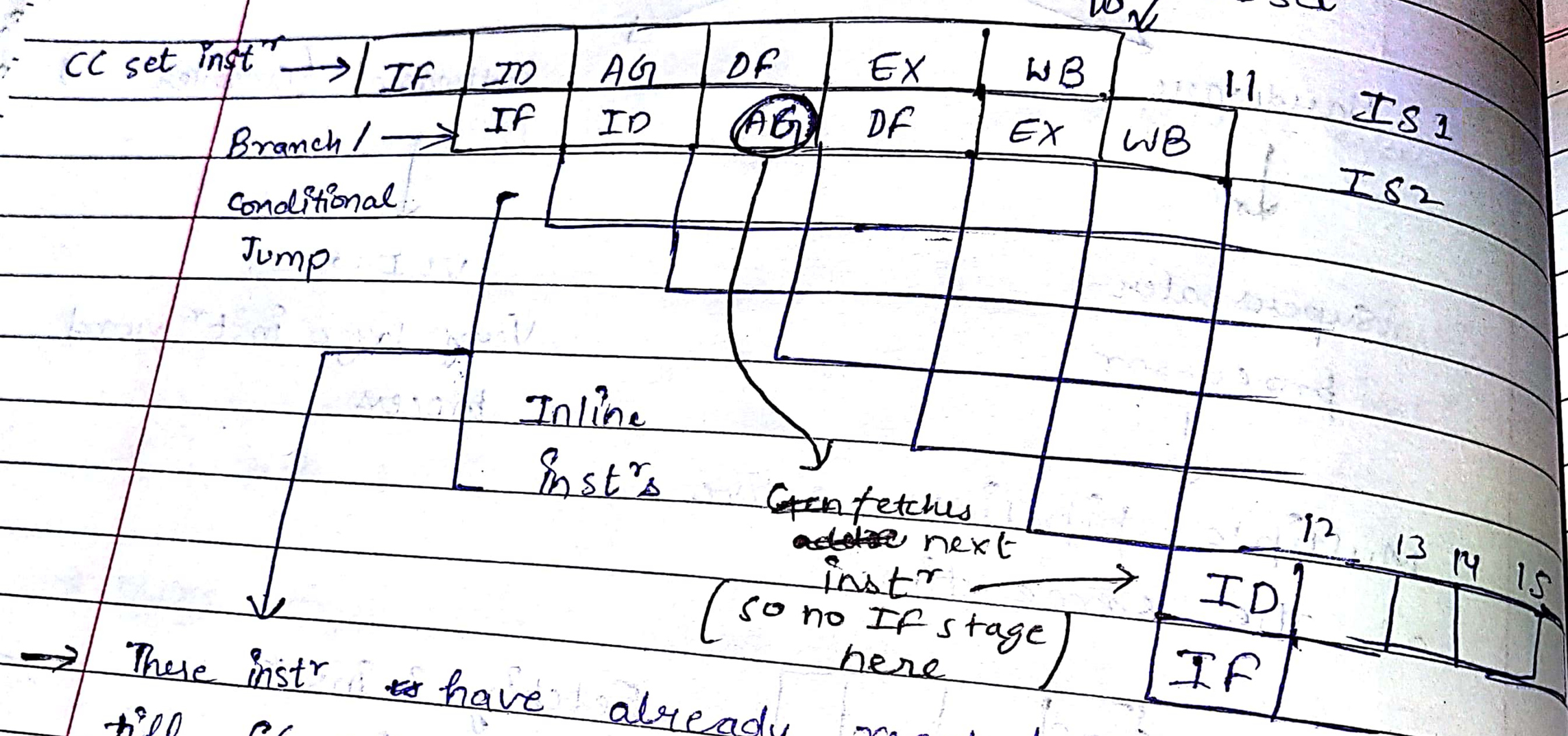
i3, i5, i7, i9 → multipipline with each processor of superscalar type.

☆ Control Dependencies:

Program Control instr
↓
Jump

- Conditional (flag (condition codes)): JZ, JNZ
- Unconditional: JMP, CALL, RET, RST, Interrupts

CISC Instr Template



→ These instr have already reached the pipline till CC set. Now, we have to branch.

So, we flush out all the pipelined instrs.

	Normal	Branch
IS1	10	10
IS2	11	11
(if no branch) →	12	15 (if branch → target)

Penalty cycles = 15 − 12

⇒ 3 cycles

☆ ~~a = 0;~~
~~b = 10;~~
~~do~~
~~{~~
~~a = a+1;~~
~~b =~~

→ Note:

Branch history table → We usually store branch target address. Like a cache.

example:

Branch instr address	History bits	Target Addr

It becomes B.T.A.C. (Branch target Address Cache).

★ Branch Penalty:

Ways to avoid

- Branch ~~Predict~~ Predict^n
- Branch eliminat^n
- Branch speedup
  - uses instr reordering
    - Early cc set
    - Delayed Branch

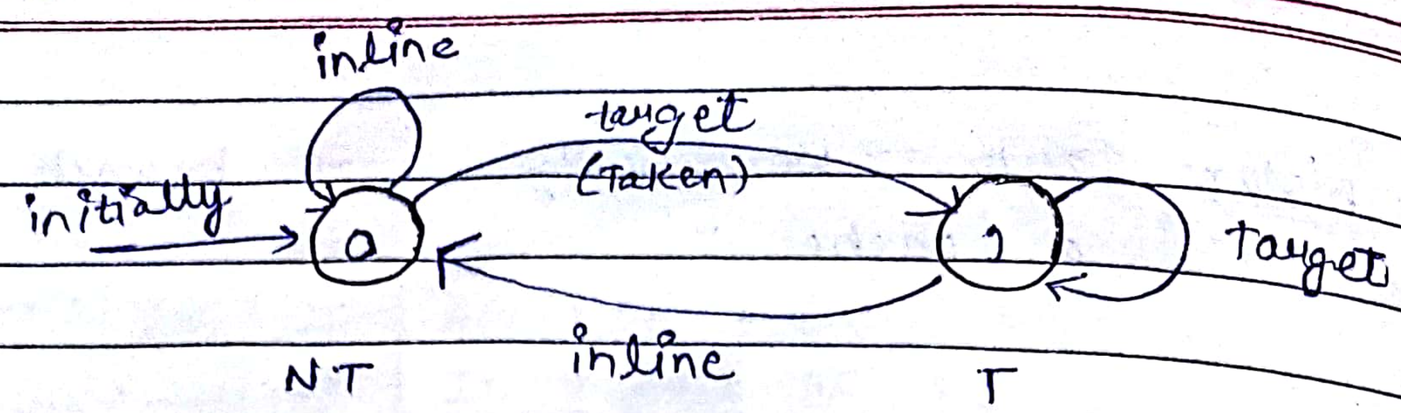
★ Branch Predict^n:

Makes use of branch history bits.

1 bit branch Predictor → Two states. (history bit 0 or 1)

(0) Predict Not Taken

(1) Predict Taken (Goes to target)



Example:

```
AGAIN:  MOV C,10
        ____
        ____
        ____
        ____
        DCR C
        JNZ AGAIN
```

Order:

Not in table

Set 0

Change to 1

remain to 1 till C=0

Change to 0

So, in subsequent decrement cycles, we are reusing the target address. Thus, speedup.

Pred	Actual
NT	T
T	T
T	T
⋮	
T	NT

2 miss predictions

Example (2):

```
AGAIN2: ____
AGAIN:  ____
        ____
        ____
        ____
10 [ DCR C
     JNZ AGAIN
        ____
10 [ DCR B
     JNZ AGAIN2
```

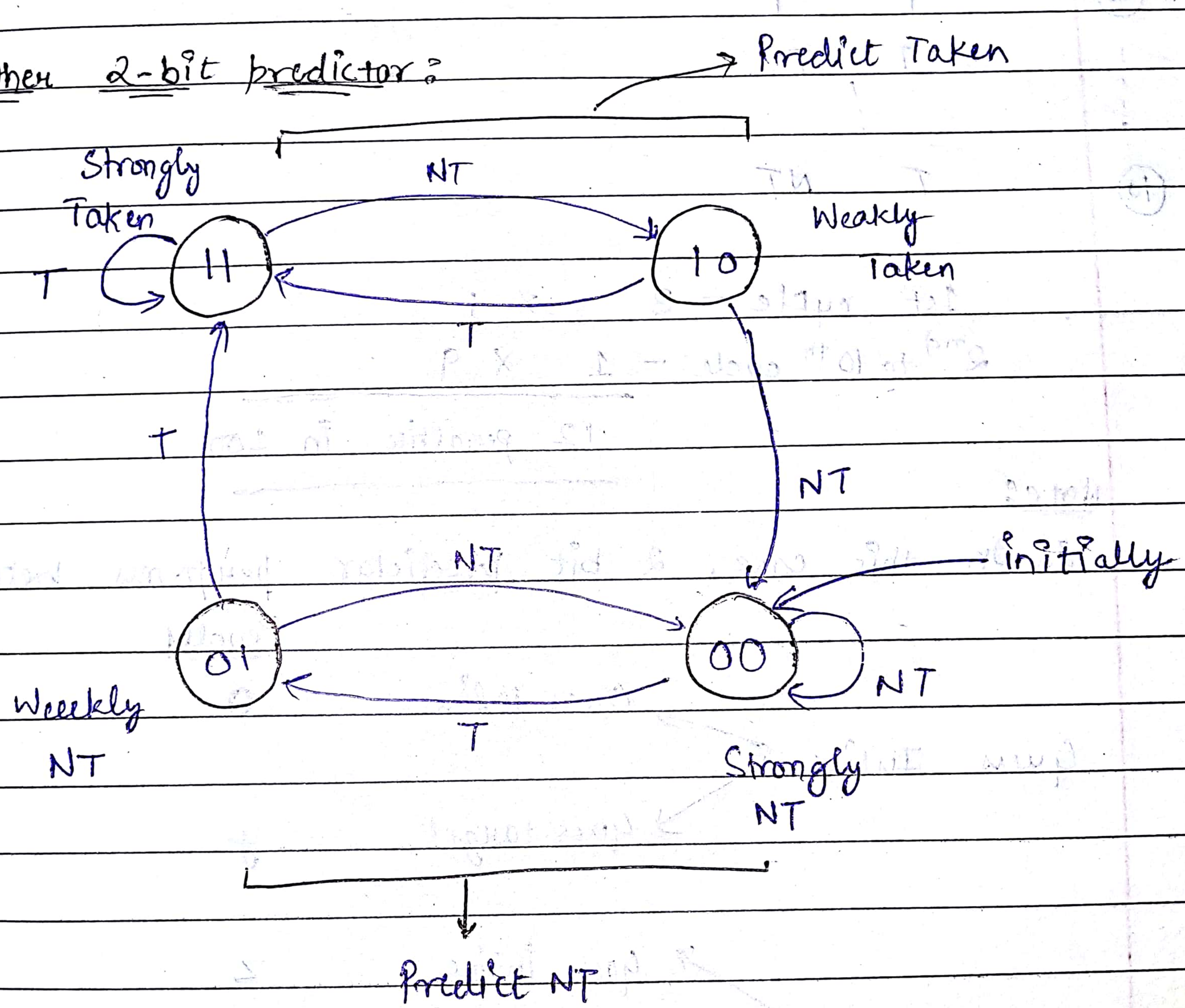
→ 2 out of 10 predictions mismatch

For inner loop,

Pred	Actual
NT	T
T	T
⋮	
T	NT
NT	T
⋮	
T	NT

20 miss predictions in 100 predictions.

☆ Another 2-bit predictor:



Pred	Actual	
(1) NT	T	(00 to 01)
NT	T	(01 to 11)
T	T	
T	T	

	T	T	
	⋮		
	T	NT	(11 to 10)
2	T	T	(01 to 11)
	T	T	
	T	T	
	T	NT	(11 to 10)
3	T	T	
⋮	T	T	
	⋮		
10	T	NT	

1st cycle – 3 × 1
2nd to 10th cycle – 1 × 9

12 penalties in 100

Note:

∴ In this case, 2 bit predictor performs better.

		Penalty
Guess Inline	Goes inline	0
	Goes target	y
Guess Target	Goes inline	z
	Goes target	0

Note:

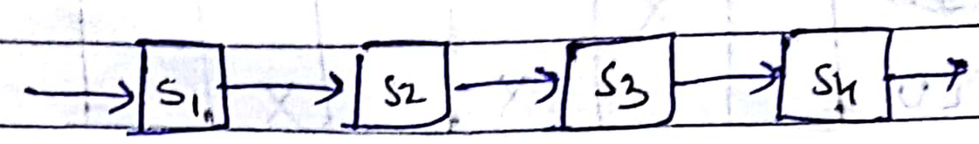
The inst's in predicted path are not stopped from excution except they cannot update state until the direction is not known.

Ch-6 in Hwang

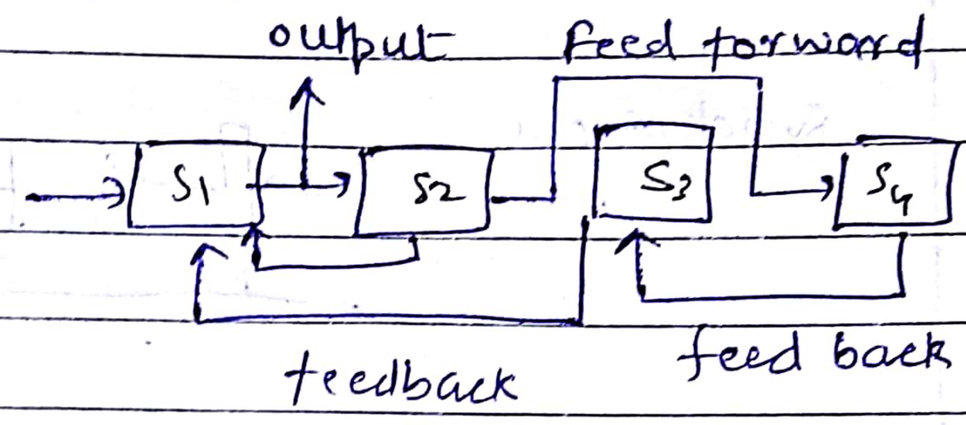
DELTA Page No.
Date / /

★ Pipeline Models :

① Linear Pipeline



② Non-Linear Pipeline
↓
Feed forward conn^n
or
Feedback conn^n



→ Represented using a Reservation table.

★ Reservation Table:

	clock cycles	1	2	3	4
Linear →	S1	X			
	S2		X		
	S3			X	
	S4				X

X → on the diagonal

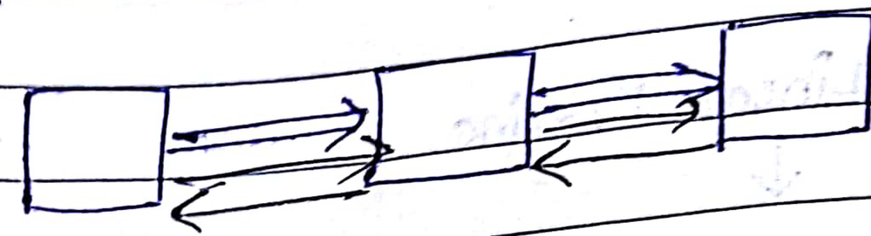
		1	2	3	4	5	6	7
Non-Linear →	S1	X		X				X
	S2		X		X			
	S3						X	
	S4					X		

1 → 2 → 1 → 2
↓
1 ← 3 ← 4

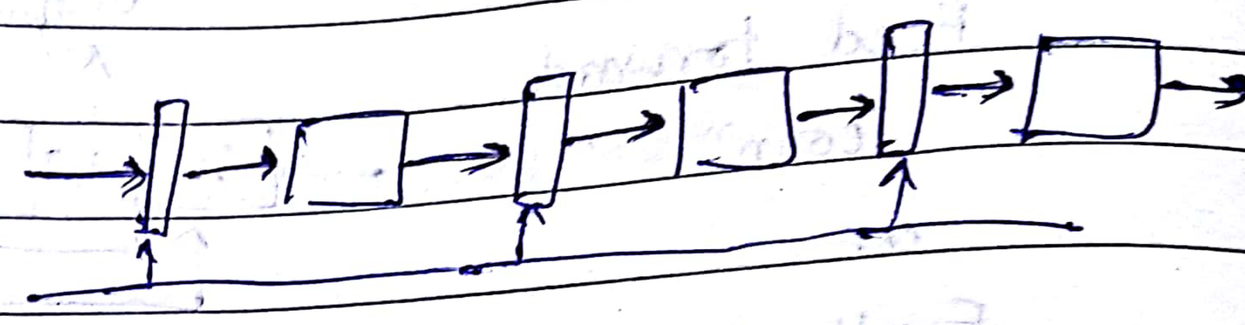
CISC → IF, ID, AG, DF, EX, WB
Von-Neumann Arch → Same add^n space for data and instructions.

	1					
Mem ~~Fetch~~ IF	X			X		X
Decode ID		X				
ALU			X		X	

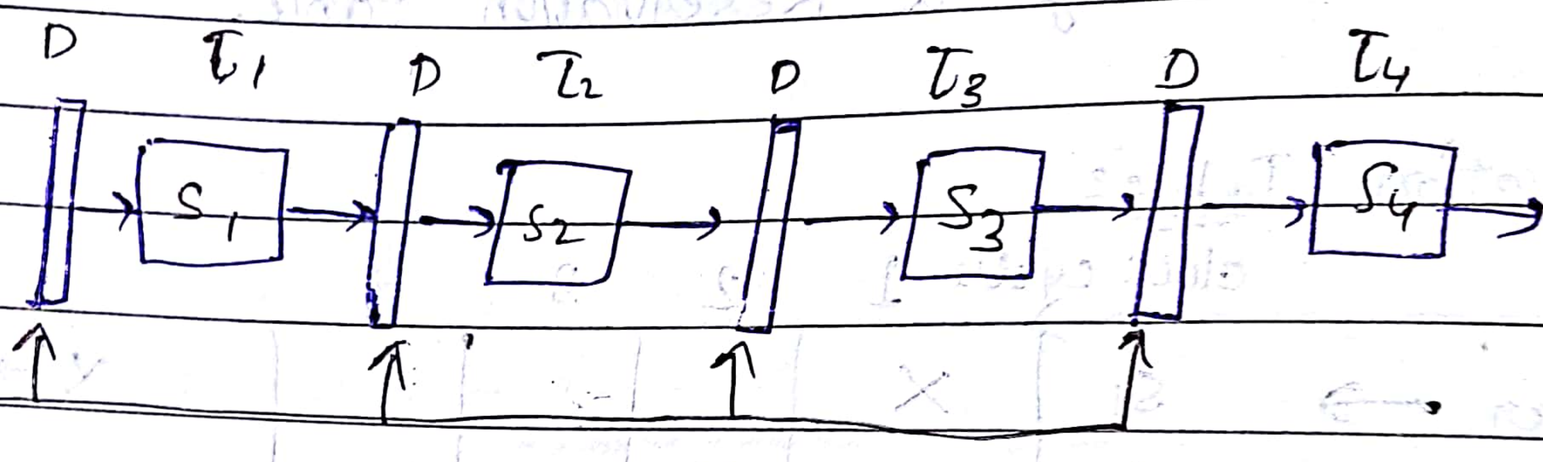
→ Asynchronous



Synchronous



☆ Clocking and Timing :-



$$\text{clock cycle} = \left(\max_{i=1}^{N} \tau_i\right) + d = \tau$$

$$\text{clock freq} = \frac{1}{\tau}$$

$$\text{Speedup} = \frac{\text{Time Taken in a Seq System}}{\text{Time Taken in a Pipelined system}}$$

Ques N tasks, K stage, τ stage delay, $d \rightarrow$ negligible wrt τ

Date / /

$$S = \frac{NK\tau}{K\tau + (N-1)\tau}$$

$K\tau + 0$

$K\tau + \tau$

$K\tau + 2\tau$

$K\tau + 3\tau$

$$S_K = \frac{NK}{K + (N-1)}$$

$$\lim_{N \to \infty} \frac{NK}{N\left(\frac{K}{N} + \frac{N}{N} - \frac{1}{N}\right)}$$

$$\lim_{N \to \infty} S_K = \frac{K}{2}$$

→ Ideally, the speedup due to pipelining = Number of stages = 4

$$\text{Throughput} = \frac{N}{K\tau + (N-1)\tau}$$

$$\lim_{N \to \infty} \frac{N}{N\left(\tau + \frac{1}{\tau} - \frac{1}{\tau}\right)}$$

$$= \frac{1}{\tau}$$

$\tau \to$ clock cycle time

∴ 1 output every clock frequency.

N tasks

K tasks

Efficiency = $\frac{N}{N+K-1}$

	1	2	3	4	5	6	7	
S_1	T_1	T_2	T_3	T_4				
S_2		T_1	T_2	T_3	T_4			
S_3			T_1	T_2	T_3	T_4		
S_4				T_1	T_2	T_3	T_4	

N inst^r K stages

$$\text{Effeciency} = \frac{N}{N+k-1}$$

$$\lim_{N \to \infty} \frac{N}{N\left(1+\frac{k}{N}-\frac{1}{N}\right)}$$

$$= 1$$

★ Reservation Table and Latency Analysis:
(Collision free Scheduling)

	1	2	3	4	5	6
S1	X			X		
S2		X				
S3				X	X	
S4			X			X

Latency → At what interval, a newer instance of a task is initiated.

Forbidden → latency that may lead to collision

Permissible → Latency with which if a task is initiated does not suffer from collision

Note: These can be represented by a binary vector (collision vector).

Permissible → 0 in collision vector
Forbidden → 1 in collision vector.

→ Find, the every two combination in row of a collision table and find the distance between them.

S1 → 3 ~~Forbidden~~
S2 → ~~6~~ — ~~Per~~
S3 → 1
S4 → 4

	1	2	3	4	5	6	7
3 ← S1	X T_1	T_2		X T_1	T_2		
5 ← S2		X T_1					X T_1 T_2
1 ← S3				X T_1	X T_1	T_2	
2 ← S4				X T_1	T_2	X T_1	

When two tasks have been initiated, 2, 3, 1 (2→4, 2→5, 4→6)

Collision vector → 1 0 1 0 1
5 4 3 2 1

MSB will always be 1.

1, 3, 5 → Forbidden latency
2, 4 → Permissible latency

Initial State → 10101

~~2 4 2 4~~

00101(01

(OR) 10101
10101 00001(0101

(same as initial) 10101
10101

2 simple cycle
2 4

① 10101 → right shift by 1 bit
01010

② Bitwise OR with original collision vector. to get a new state collision vector.

~~Sum of simple cycles > greedy cycle~~

→ From each of the node in the cycle, if their minimal latency is used, then it is a Greedy cycle.

In our example,
Greedy cycle = 2

→ Average latency of a cycle $= \frac{2}{1} = 2$ → of 4, $\frac{4}{1} = 4$

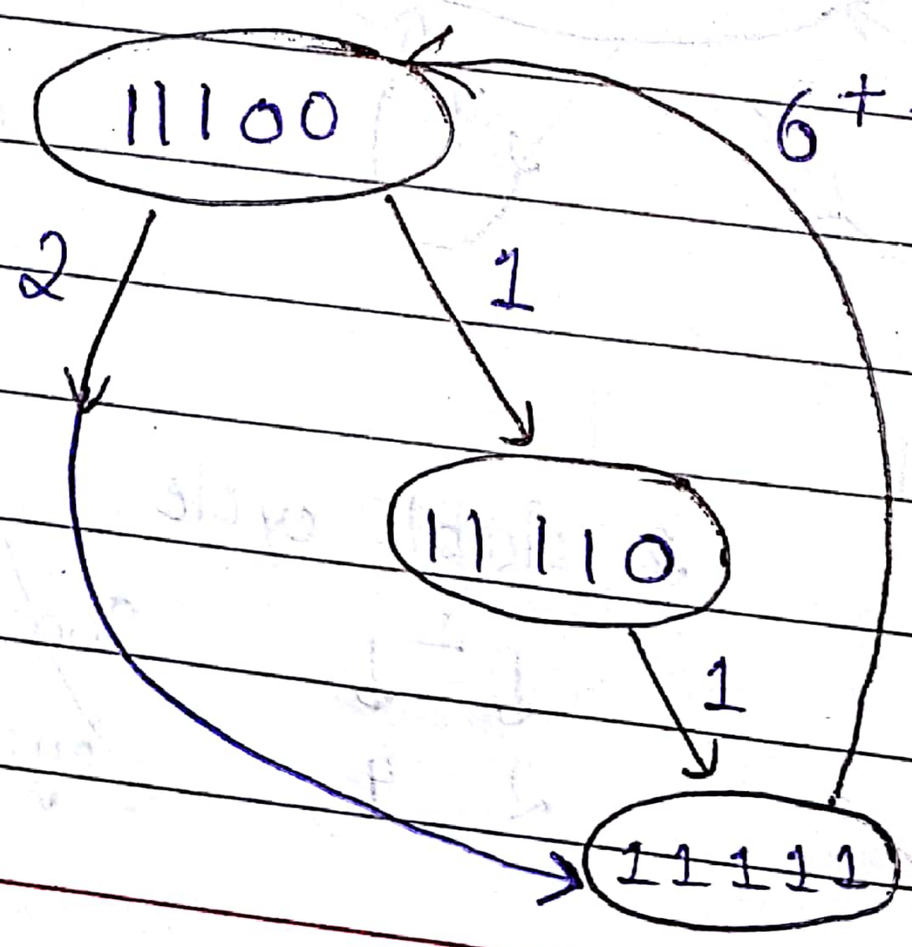
→ Minimal average latency = 2
(MAL)
↓
Pick greedy cycle

Q

S1	X					X	→ 5
S2		X			X		→ 3
S3			X				→ _
S4				X			→ _
S5		X				X	→ 4

Collision vector = $\underset{5}{\underline{1}}\ \underset{4}{\underline{1}}\ \underset{3}{\underline{1}}\ \underset{2}{\underline{0}}\ \underset{1}{\underline{0}}$

00111(00
11100
11111



→ 6, 7, 8, ...

01110(0
11100
11110

1. Simple cycle = 1, 1, 6
2, 6

2. Average latency = $\frac{1+1+6}{3}$ = $\frac{2+6}{2}$
= 2.67 = 4

3. Greedy cycle = 1, 1, 6

4. MAL = 2.67

→ Optimal MAL
↳ in any row in reservation table, optimal MAL is only possible if number of crosses in every row is greater than MAL.

↳ In our example, optimal MAL needed = 2
Actual MAL = 2.67. ∴ Not Optimal

↳ insertion of non-compute delays (Technique to resolve)

Q

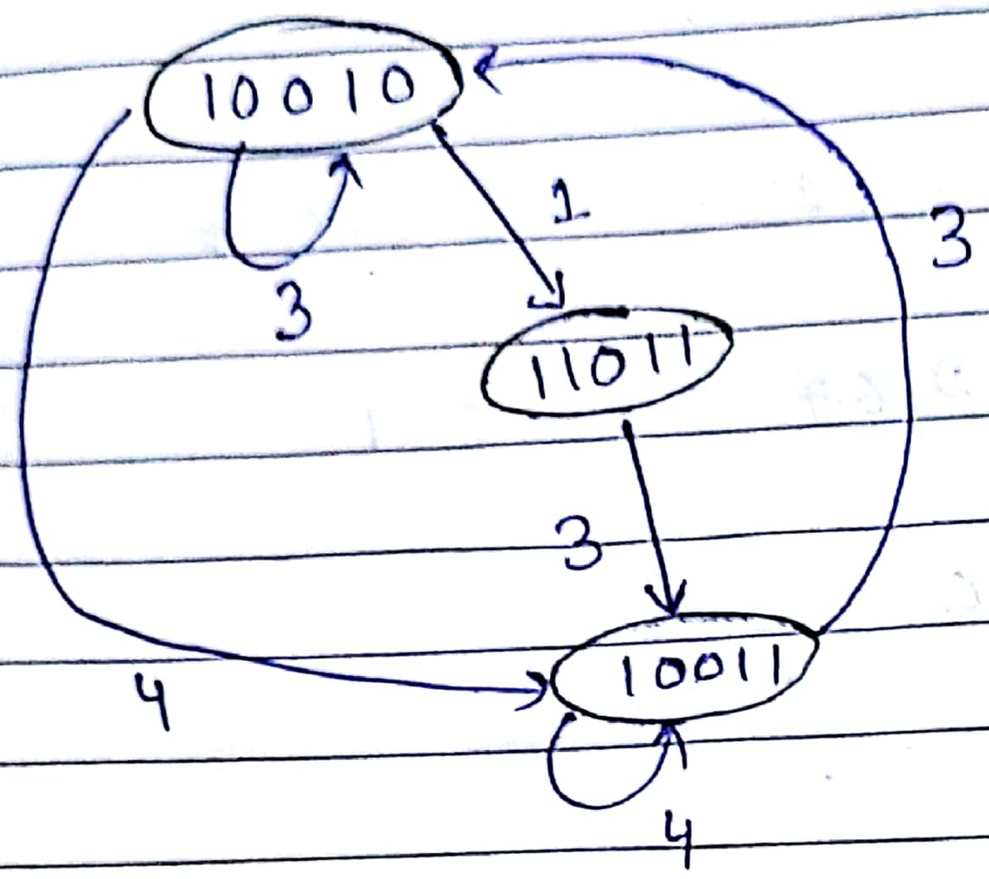
S1	X					X	→ 5
S2		X		X			→ 2
S3			X		X		→ 2
S4				X			→ -
S5			X		X		→ 2

Collision vector → 10010

Date / /

level 1

010010
10010



00010 (010
10010

00001 (0010
10010
10011

Simple cycles → 3
1, 3, 3
4, 3
4

Greedy cycles → 1, 3, 3

MAL → $\frac{1+3+3}{3} = \frac{7}{3} = 2.33$

optimal MAL needed = 2
2.33 > 2 ∴ not optimal

★ Programming Assn:

Q FIND MAL. List all simple cycles, greedy cycles, collision vector
Read reservation table from a text file as Input

X, , , , X, , ,
, X, , X, , , X
, , X, , , ,
, , , , , X,

Read name of input file from command line

★ Bounds on MAL:

① MAL is upper bounded by the number of 1's in initial collision vector plus 1.

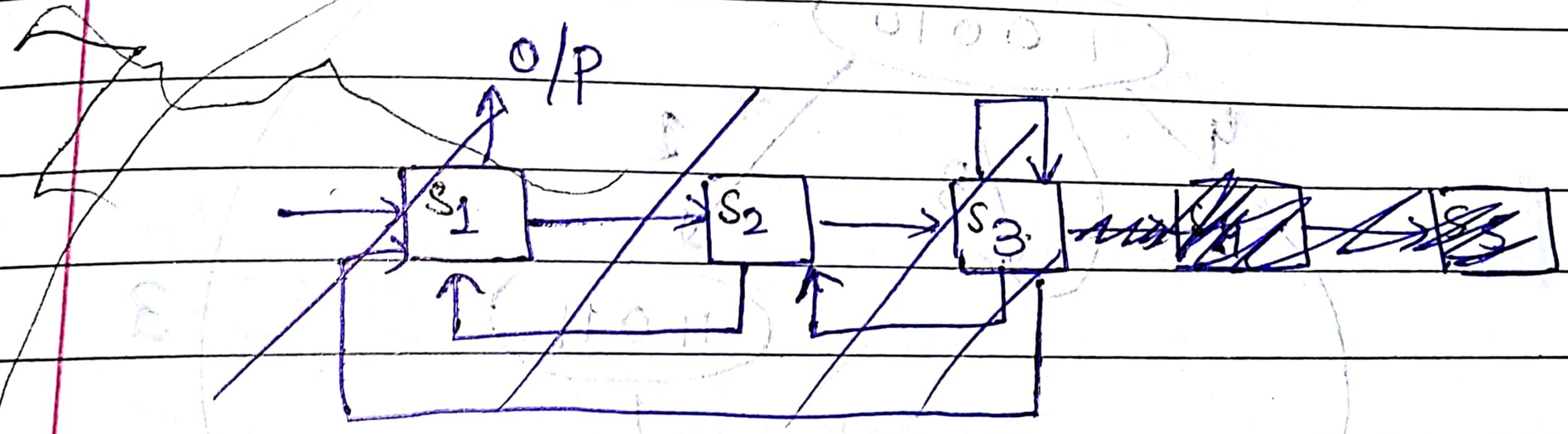
② MAL will be less than or equal to the average latency of a greedy cycle.

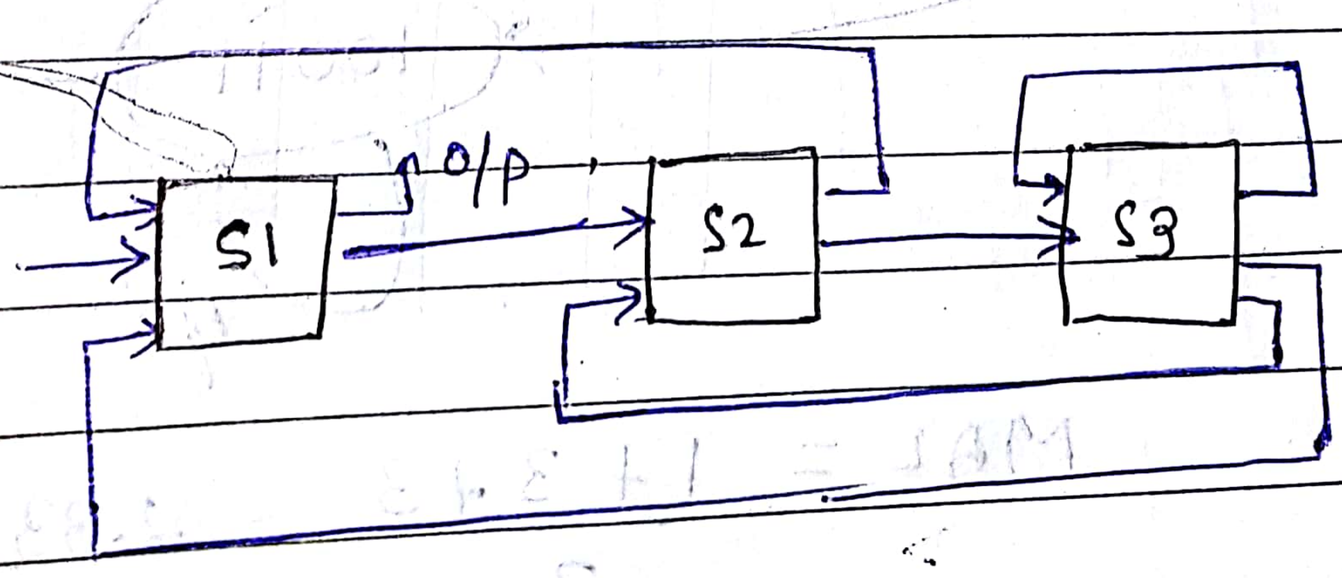
③ the average latency of a greedy cycle is lower bounded by the max no. of crosses in a row in reservation table. This is the lower bound for MAL also.

★ Insertion of non-compute delay:

(so as to get optimal MAL)

	1	2	3	4	5	
S1	X				X	4
S2		X		X		2
S3			X	X		1

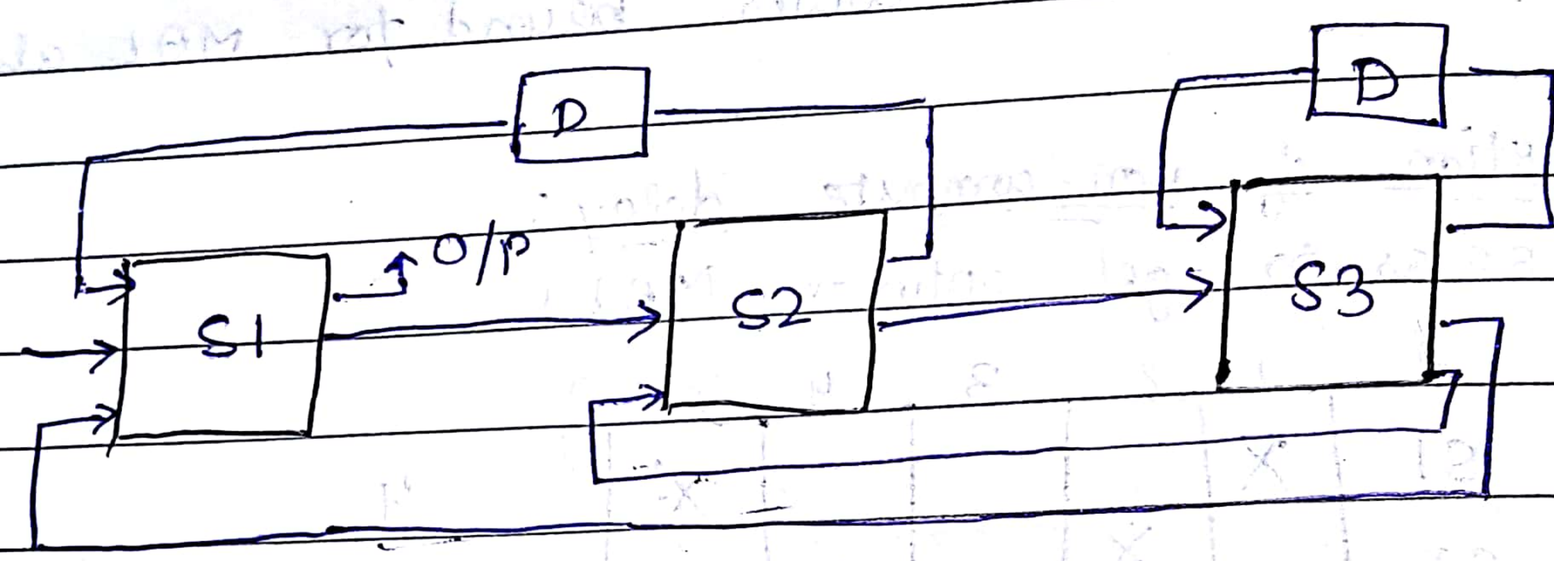


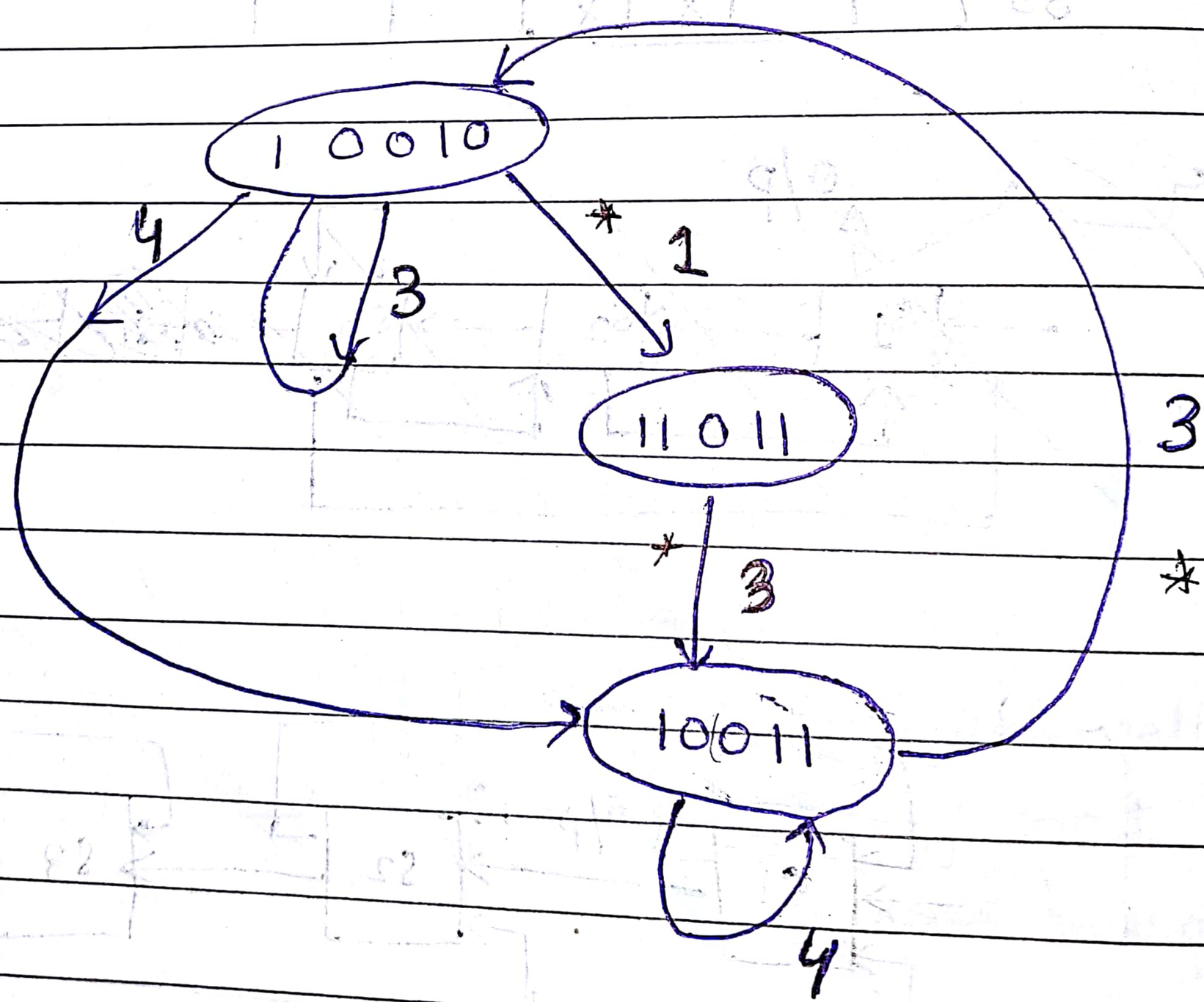


~~After~~ 1011 5 3 MAL = 3 (Non-Optimal)

→ After delay insert^n

	1	2	3	4	5	6	
S1	X					X	5
S2		X		X			2
S3			X		X		2
D1				X			–
D2							–





$$MAL = \frac{1+3+3}{3} = 2.33$$

Simple cycles → 3

1, 3, 3

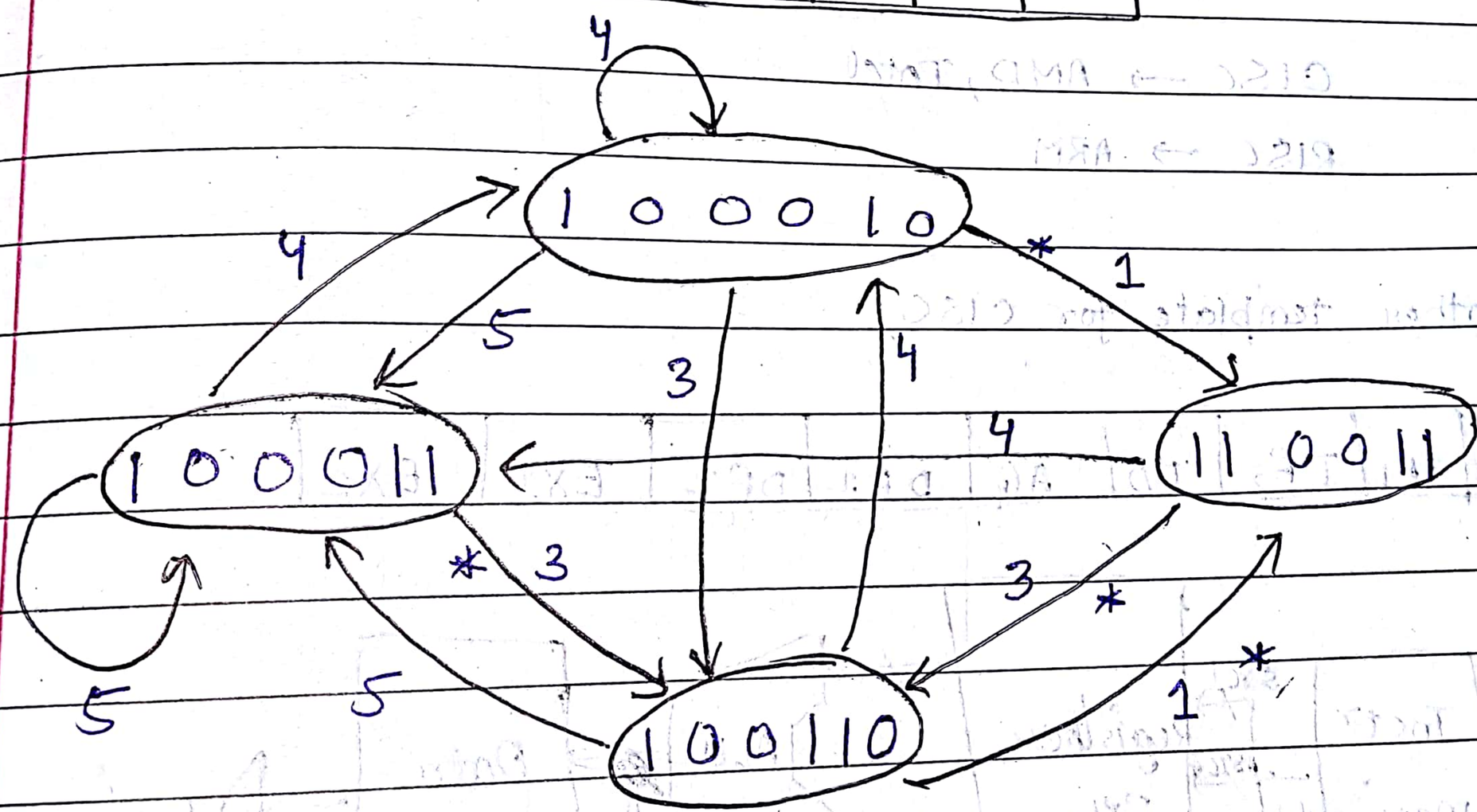
4

4, 3

Greedy cycle → 1, 3, 3

→ Another delay, insertion

	1	2	3	4	5	6	7	
S1	X							
S2		X		X			X	6
S3			X		X			2
D1				X				2
D2						X		



1, 3, 1, 3, 1, 3, 1, 3, 1

Greedy → 3, 1

★ Instruction Pipeline:

Instr Execn → Fetch, Decode, Execute o/pn

Instr Template

CISC	IF	ID	AG	DF	EX
RISC	IF	ID	EX	MEM	WB

→ write result may be seperate or may be treated as part of EX stage.

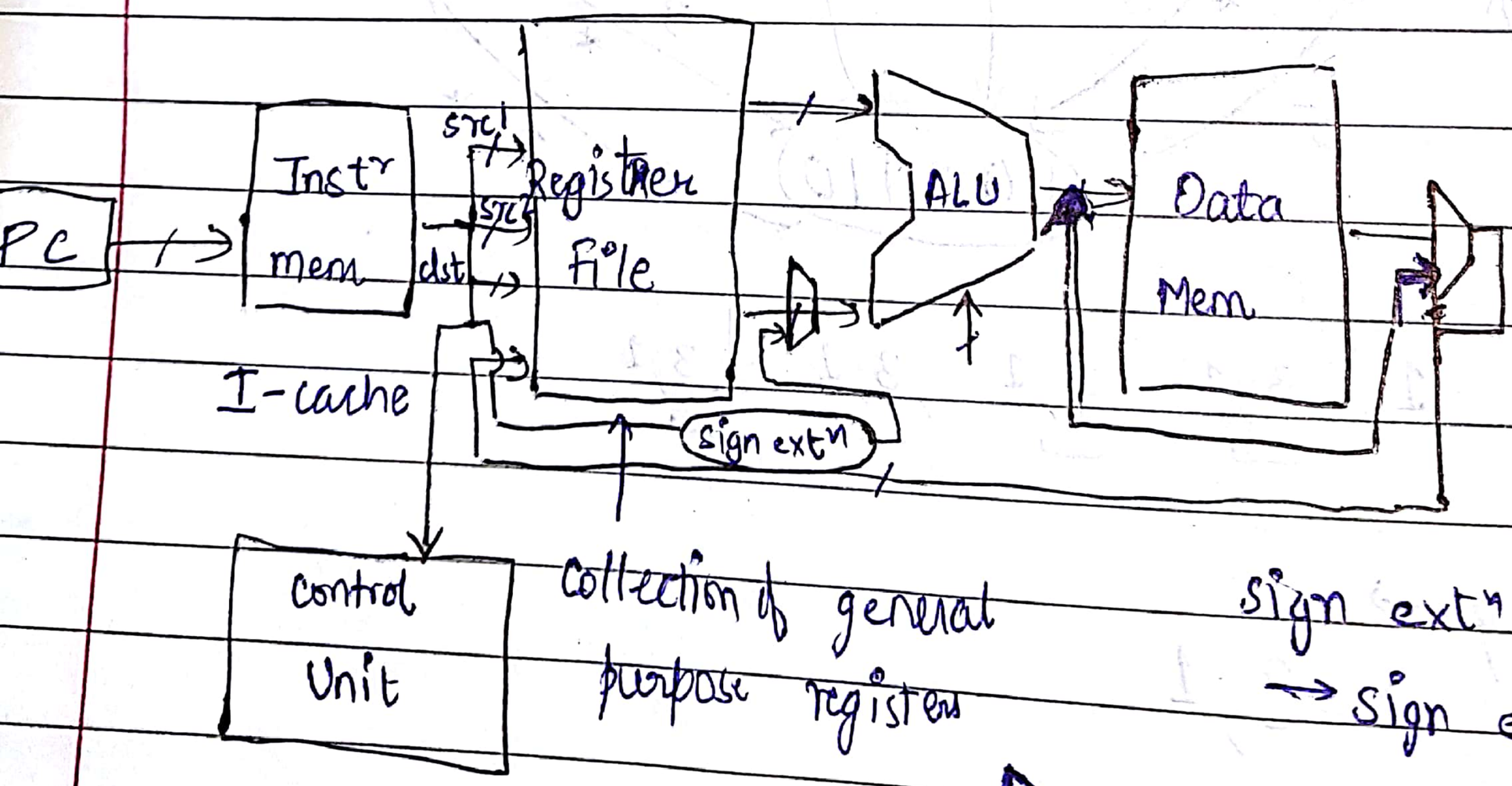
	INTEL			
8085	8080	80286	P3	i5
↑	8085	80386	P4	i7
belongs to	8086	80486	Dual Core	i9
CISC x86	8088	PI	Core 2 DUO	
ISA	80186	P2	i3	

CISC → AMD, Intel

RISC → ARM

Another template for CISC,

IF_1	IF_2	ID	AG	DF1	DF2	EX1	EX2



collection of general purpose registers

sign extn → sign extension

→ multiplexer

Henessey Patterson
1. Basic CA
2. Adv CA

1) → Register only instr
add r_1, r_2, r_3 IF ID EXE ~~MEM~~ WB
dst src1 src2

2) → RISC → Load H, 10(r_2) IF ID EXE MEM WB
Address generate

sign extⁿ → it converts back (10) to the number of bits ~~compare~~ of the other operand.

+5 in 16 bits → 0.....0101
-3 in 16 bits → 1.....1101

~~3) → store reg~~

3) → store r_1,