Pipelining -

1. Pipelining is a H/w mechanism.
2. Pipelining uses the decomposition technique.
3. The function of the pipeline is one pipe output is connected as input to the another pipe.

(4) Definition of the pipeline is "Accepting the new i/p's at one end before previously accepted i/p appears as an o/p at the other end."

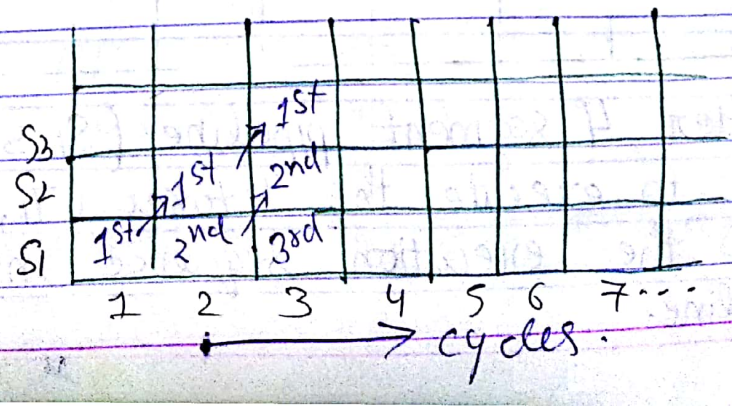
(5) Definition states that new i/p's are inserted into the pipeline before completion of the old inputs. So that new i/p's are executed along with the old i/p's.

Therefore, the new i/p's are utilizes the CLK cycles of the old inputs.

(6) Pipelining allows the over-lapping execution.

(7) In the non-pipelining process, new i/p's are inserted only after completion of the old inputs. Therefore, it allows the non-overlapping execution.

(8) Overlapping execution seq. in the pipeline is represented using space-time diagram.
i.e.



⑨ Successful characteristic of the pipeline is every new cycle, new i/p must be inserted into the pipeline. Therefore,

$$CPI = 1$$

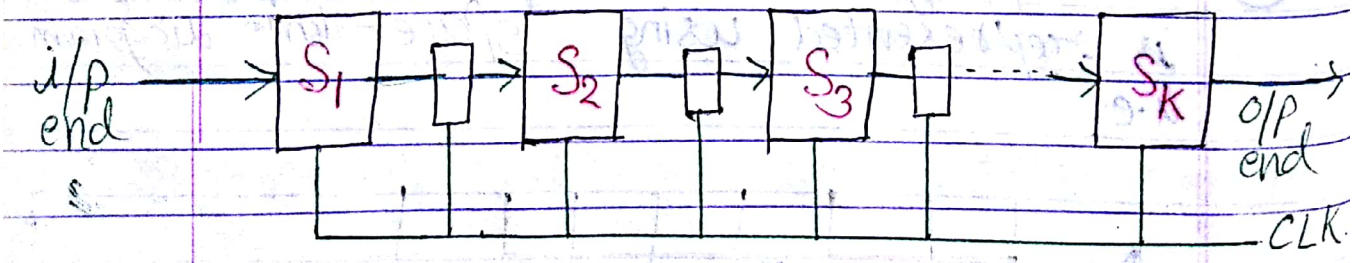
Designing of Pipeline:-

① Pipe has two ends i.e. input end & output end.

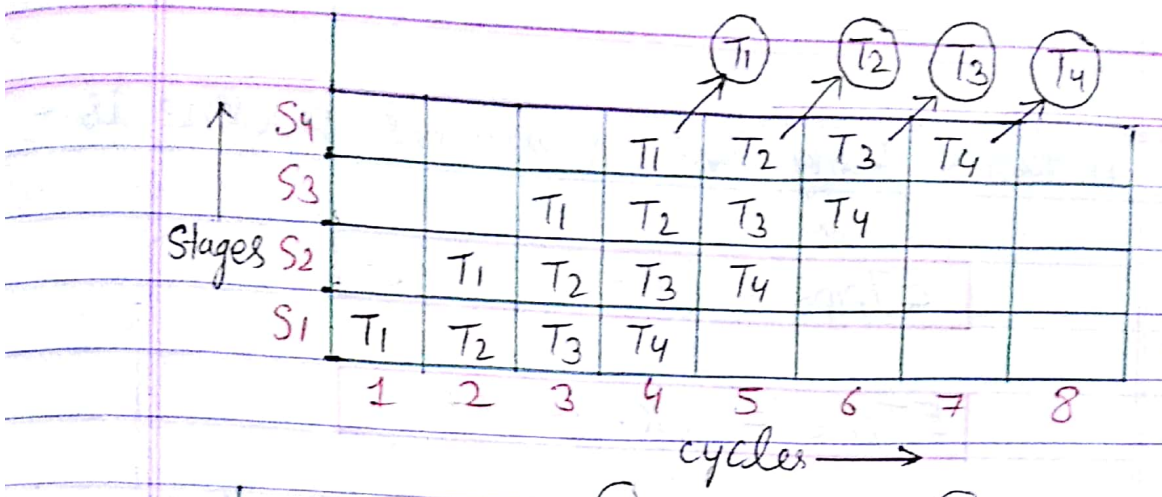
② Between the i/p & o/p ends, multiple pipes are interconnected to satisfy the objective of the pipelining. Each pipe is called as the stage or segment.

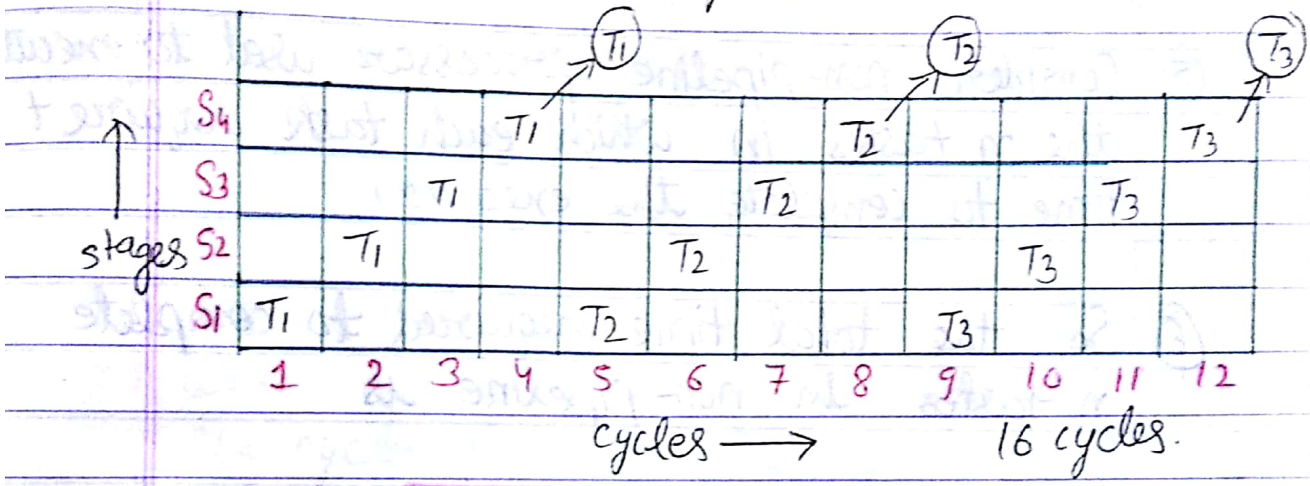
③ B/w the stages, interface registers are used to hold the intermediate result i.e. also called as Latch/Buffer / pipeline register.

④ All the stages along with the interface registers are controlled by the common clock.



⑤ Consider 4 segment pipeline (S_1, S_2, S_3, S_4) used to execute the 4 tasks (T_1, T_2, T_3, T_4). Show the execution sequence in the pipeline.





Performance Evaluation of Pipeline Processor –

① Consider K-segment pipeline with clock cycle time tp used to execute n tasks.

② The very first task in the pipeline is executed as non-overlapping order. So, it takes K-cycles to complete the operation.

③ The remaining (n-1) tasks are emerge from the pipe at 1 task/cycle. Therefore, (n-1) cycles are required to complete (n-1) tasks.

④ So, the total time required to complete the

n tasks using the k-segment pipeline is -

$$ET_{pipe} = (k+n-1) \text{ cycles}$$

$$ET_{pipe} = (k+n-1) * tp$$

⑤ Consider non-pipeline processor used to execute the n tasks in which each task require tn time to complete the execution.

⑥ So, the total time required to complete n tasks in non-pipeline is

$$ET_{non\text{-}pipe} = n * tn$$

⑦ The performance gain of the pipeline processor over the non-pipeline is

$$\text{Speed Up (S)} = \frac{\text{Performance}_{pipe}}{\text{Performance}_{non\text{-}pipe}}$$

$$= \frac{1/ET_{pipe}}{1/ET_{non\text{-}pipe}}$$

$$= \frac{ET_{non\text{-}pipe}}{ET_{pipe}}$$

$$S = \frac{n * tn}{(k+n-1) * tp}$$

⑧ When the n-value is increases the $(K+n-1)$ is approach to the value of n.

⑨ Under this condition,

$$S = \frac{n * t_n}{n * t_p}$$

$$\boxed{S = \frac{t_n}{t_p}}$$

10.) When all ins^n are taking same no. of CLK cycles then one task execution time is also equal to the no. of stages in the pipeline. i.e. $t_n = K$ cycles.

$$\boxed{t_n = K * t_p}$$

$$S = \frac{K * t_p}{t_p}$$

$$\boxed{S = K \text{ (pipeline depth)}}$$

⑪ When the system is operating with 100% efficiency then max. SpeedUp = pipeline depth

100% η — S_{max}

? η — S

$$\boxed{\eta_{pipe} = \frac{S}{S_{max}} = \frac{S}{K}}$$

(12) Throughput of the pipeline is no. of tasks completed per total time required to comple the task.

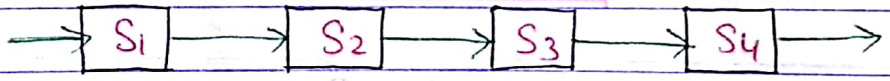
$$TP_{pipe} = \frac{\#\text{ tasks completed}}{\text{total time req. to complete task}}$$

$$TP_{pipe} = \frac{n}{(k+n-1)*tp}$$

Types of Pipeline-

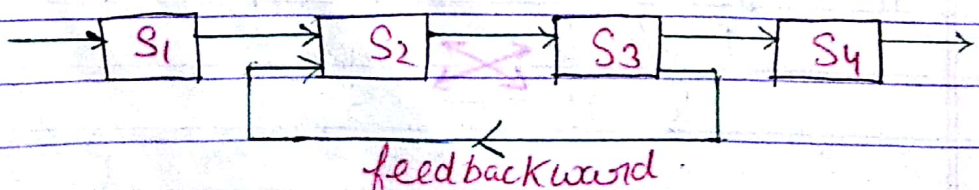
① Linear Pipeline-

This pipe is used to process only one functionality. In this pipeline, only the feedforward connections are present



② Non-Linear Pipeline -

This pipeline is used to process the multiple functions. In this pipeline feedforward & feedbackward connections are maintained.



③ Synchronous Pipeline –

In this pipeline, the initiations are controlled based on the CLK. Linear pipelines are the synchronous pipelines.

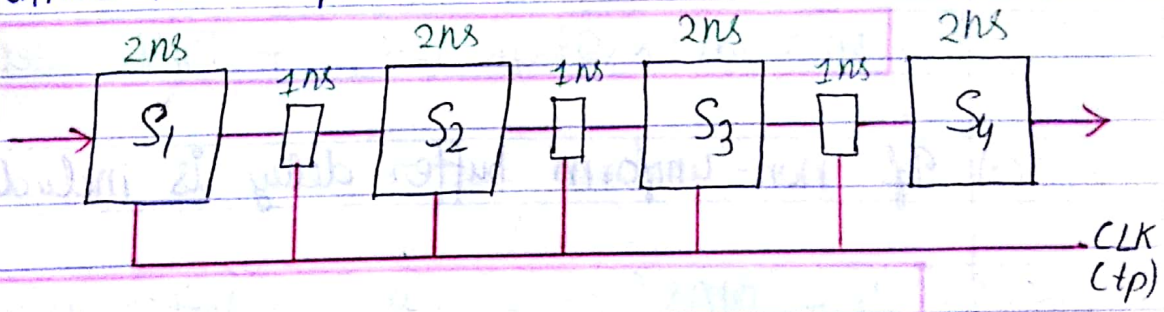
④ Asynchronous Pipeline –

In this pipeline, operations are initiated based on the handshaking signals. It means that before initiating the operation, there is a need of taking the feedback.

Hypothetical Pipelines –

① Uniform Delay Pipeline –

In this pipeline, all the stages are taking the same amount of time to complete the operation.



a) t_p = stage delay

b) If buffer delay is included

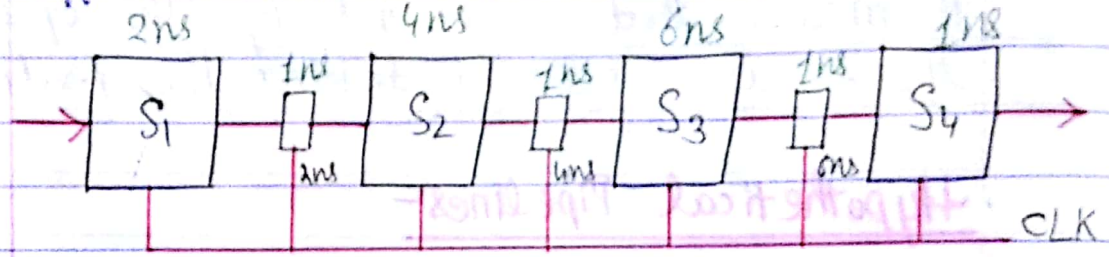
t_p = stage delay + Buffer delay

c) If any skew time / setup time overheads are included

$$t_p = t_p + \text{overhead}$$

(3) Non-Uniform Delay Pipeline –

In this pipelin
different stage delays are maintained.



a) $t_p = MAX(\text{Stage delay})$

b) If buffer delay is included.

$$t_p = MAX(\text{Stage delay}) + \text{Buffer delay}$$

c) If non-uniform buffer delay is included

$$t_p = MAX(\text{stage delay} + \text{Buffer delay})$$

Ques-1 A ins^n pipeline has the speedup factor. So whi
operating with 80% efficiency. What could be the no
of stages in the pipeline.

$$\eta = \frac{S}{K}$$

$$0.8 = \frac{10}{K} \Rightarrow K = \frac{10}{0.8} = 12.5 \simeq 13$$

2) Consider 2 pipelines A & B where pipeline A having 8 stages of uniform delay of 2ns, pipeline B having 5 stages with respect to stage delays of 2, 4, 6, 3 & 1. How much time is saved when 100 tasks are pipelined using A instead of B.

$ET_A = (K+n-1) * tp$

$= (8+100-1) * 2$

$= 214$

$ET_B = MAX(2, 4, 6, 3, 1) = 6$

$= (5+100-1) * 6$

$= 624$

Save time $= ET_B - ET_A$

$= 624 - 214$

$= 410$ sec Ans.

3) Consider 4 segment pipelines with the respective stage delays 20, 30, 40, 10 ns. What is the approx speed up when the very large no. of instructio are pipelined.

$$S = \frac{tn}{tp}$$

$tp = MAX(20, 30, 40, 10) = 40$

$tn =$ One task execution time

$tn = S_1 + S_2 + S_3 + S_4.$

$= 20+30+40+10$

$= 100$

$$S = \frac{100}{40} = 2.5$$

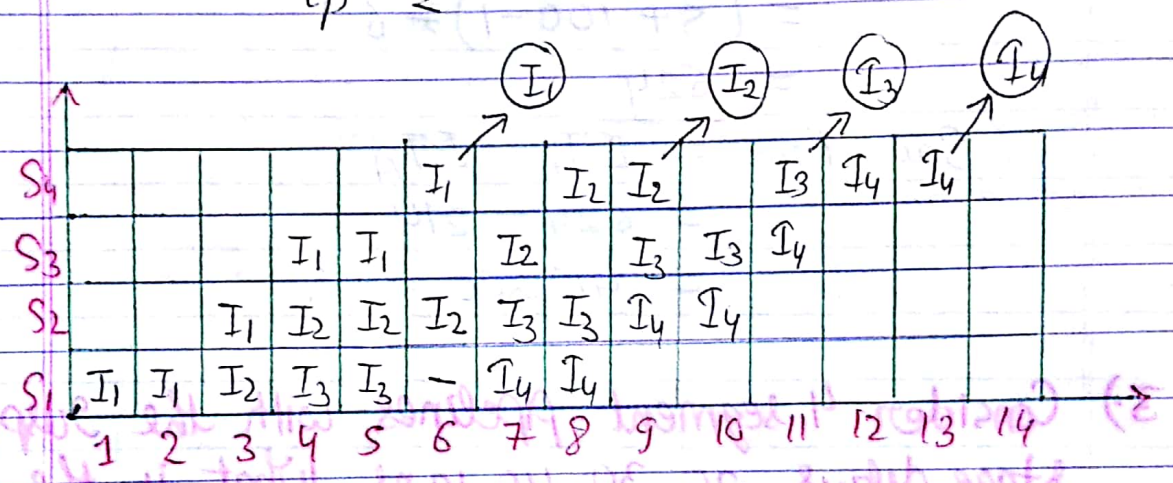
$$\eta = \frac{S}{k} = \frac{2.5}{4} = 62.5\% \text{ Ans.}$$

4) Consider 4 stage instn pipeline where different instn are spending different amount of time at different stages i.e. shown below. What is the speed up.

	S_1	S_2	S_3	S_4	
I_1	2	1	2	1	= 6
I_2	1	3	1	2	= 7
I_3	2	2	2	1	= 7
I_4	2	2	1	2	= 7

asynchronous.

$t_p = 2$



$ET_{pipe} = 13$ cycle.

$ET_{non\text{-}pipe} = I_1 + I_2 + I_3 + I_4$.

$= 27$

$$S = \frac{ET_{non\text{-}pipe}}{ET_{pipe}}$$

$$= \frac{27}{13}$$

$S = 2.07$.

$$\eta = \frac{S}{K}$$

$$= \frac{2.07}{4} = 0.517 = \underline{51.7\%}$$ Ans.

Lecture-9

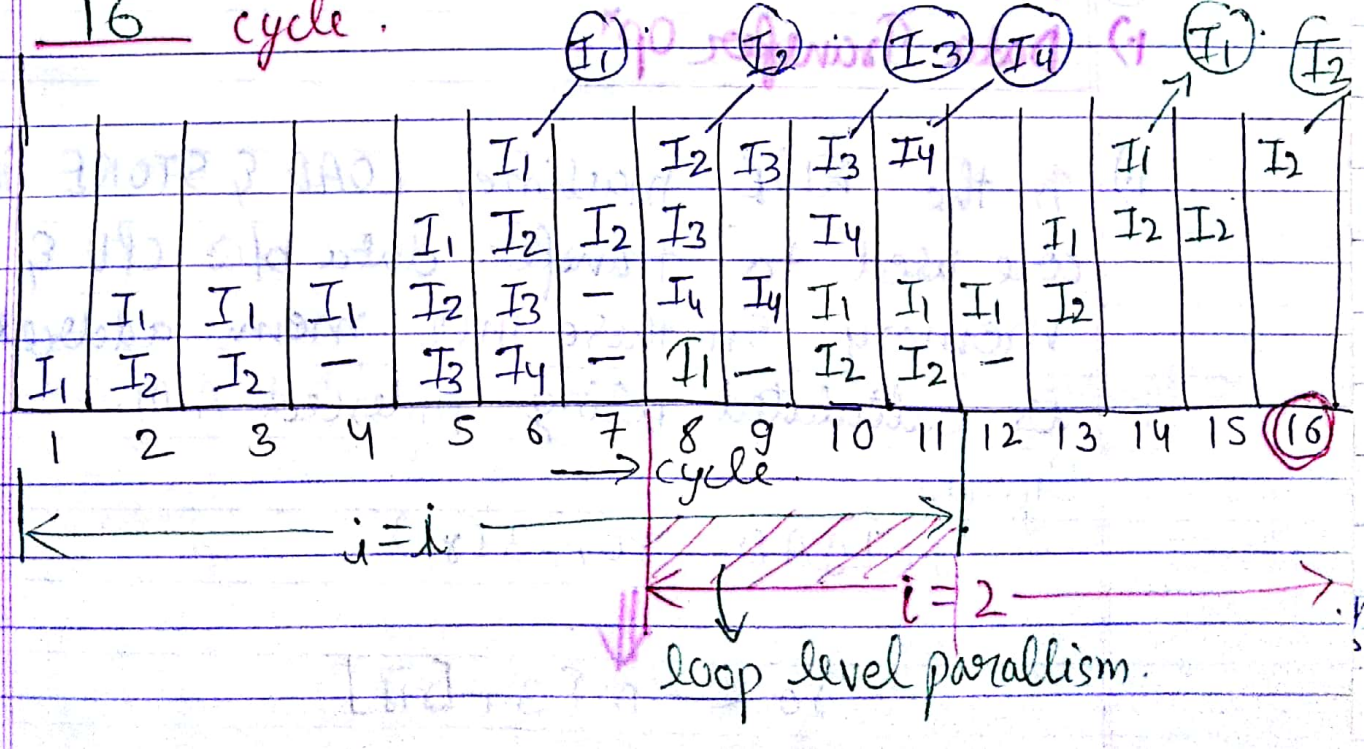
Ques-1 Consider 4 stage pipeline where diff. ins^n are pending different amount of time at diff. stages shown below:-

	S_1	S_2	S_3	S_4
I_1	1	3	1	1
I_2	2	1	2	1
I_3	1	1	1	2
I_4	1	2	1	1

The following loop is executed in the pipeline

```
for (i=1; i<=1000; i++)
{
  I1;
  I2;
  I3;
  I4;
}
```

The o/p of "I_2" for ($i=2$) will be available after 16 cycle.



RISC Processor –

① To analyse the implementation issue of pipelining, let us consider RISC pipeline as a reference model.

② RISC processor supports 5 stage ins" pipelin used to execute the ins".

③ RISC processor ins" set contain 3 category of the op":-

④ D

i) Data Transfer Op" –

i) In the RISC machine, LOAD & STORE ins" are used to transfer data b/w CPU & memory. In these ins", mem. address is calculated using indexed A.M.

Syntax –

1.) LOAD r0, 3(r1)

⇓

r0 ← m[3+[r1]]

2) Store 3(r1), r0

⇓

M[3+[r1]] ← r0

2) Data Manipulation op^n-

1) In the RISC CPU, ALU op^n are performed only on the register data.

Syntax-

ADD r0, r1, r2

⇓

r0 ← r1 + r2

3) Transfer of ctrl op^n-

1) Unconditional TOC

Syntax- JMP 2000

⇓

PC ← ~~seq. Addr~~ 2000

2) Conditional TOC-

1) In this op^n, condition is evaluated based on current ins^n only.

Syntax- DJNZ r0, 2000

⇓

Dec r0

r0 compare NZ

True (PC ← ~~Seq. Addr~~ 2000)

False. (PC ← seq. Addr)

To execute the above insⁿ, RISC processor supports 5 stage insⁿ pipeline -

1.) Stage I (Insⁿ Fetch):

(1) In this stage, CPU reads the insⁿ from memory simultaneously, PC is incremented to next seq. Insⁿ Adder.

Stage II (Insⁿ Decode)

(1) In this stage, two different opⁿs are performed i.e. decode the insⁿ & access the register file.

(2) This stage is also contain the comparator logic to evaluate the branch condⁿ.

Stage III (Execute)

In this stage only the ALU opⁿ are performed

Stage IV (Memory Access)

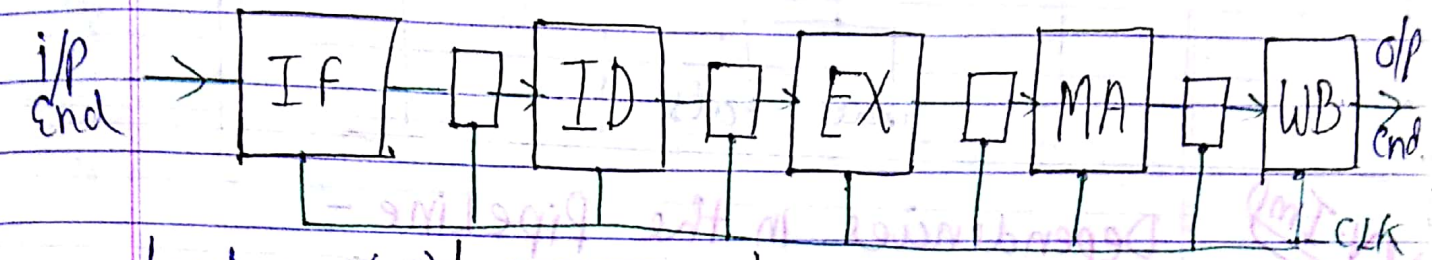
In this stage, only the memory read & memory write opⁿs are performed.

Stage V (Write Back).

This stage opⁿ is divided into two parts

In the first part, register write op^n is performed.

In the 2nd part, register read op^n is performed.



Load r_0, 3(r_1) PC ← PC+SS	MemRD S: mem(3,r_1) D: reg (r_0) r_1 = value.	3+[r_1] → Mem → Binary → r_0 ← M[3+[r_1]]
Store 3(r_1), r_0 PC ← PC+SS	MemWR S: reg (r_0) D: mem(3,r_1) r_0 = value r_1 = value	3+[r_1] → mem → r_0 M[3+[r_1]] ← r_0
Add r_0, r_1, r_2 PC ← PC+SS	+ S: reg (r_1, r_2) D: reg (r_0) r_1 = value r_2 = value	r_1+r_2 → mem → r_0 ← r_1+r_2
JMP 2000 PC ← PC+SS	uncond. TOC TA = 2000 PC ← ~~reg Addr.~~ 2000	— — —

JNZ R0, 2000 PC ← PC + S.S.	Cond. Toc TA : 2000 R0 = Value Dec R0 R0 compare NZ True False	—	—	—

V.V. Imp

Dependencies in the Pipeline -

(1) 3 Kinds of dependencies are possible in the pipeline named as
1.) Structural Dependency.
2.) Data Dependency.
3.) Control Dependency.

(2) Dependency creates the extra cycles in the pipeline.

(3) Cycle without new input insertion is called as Stall Cycle.

(4) When the stalls present in the pipeline, then CPI ≠ 1. Therefore, 100% Efficiency never possible.

1.) Structural Dependency -

(1) This dependency is created in the pipeline due to the Resource Conflict. Resource

may be a memory or register or functional unit.

Diagram I

	CC_1	CC_2	CC_3	CC_4	CC_5
I_1	Mem	ID	ALU	Mem.	
I_2		Mem	ID	ALU	
I_3			Mem	ID	
I_4				Mem	

② In this above cycle diagram, Ins^n I_1 & I_4 both are trying to access the same resource (memory) in the same cycle (CC_4). This situation is called as Resource Conflict.

Resource Conflict in the pipeline is named as Structural Dependency.

③ Conflict is a unsuccessful op^n, so to make it successful keep I_4 into waiting until the resource becomes available. This waiting creates stalls in pipeline.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1	Mem	ID	ALU	Mem	WB		
I_2		Mem	ID	ALU	Mem	WB	
I_3			Mem	ID	ALU	Mem	WB.
I_4							

stalls.

④ To minimize the no. of stalls in pipelin due to the structural dependency, one of H/w mech. is used called as Renamin

⑤ In the Cycle diagram-1, I_1 is accessing the memory in 4th stage to read/write the data whereas I_4 is accessing the memory in the 1st stage of pipelining to read the insn in same cycle of CC4.

⑥ When the insn & data both present in same memory. then the above situation creates the memory conflict.

⑦ Renaming mech. states that Divide the mem. into independent sub-modules to store insn & data separately called as code memory (CM) & data memory (DM) respectively.

⑧ Refer the code memory in the stage 1 & data memory in the stage 4 of pipeline. Then, accessing the code memory & data memory in same cycle doesn't creates the conflict.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
I_1	CM	ID	ALU	DM	WB	
I_2		CM	ID	ALU	DM	WB
I_3			CM	ID	ALU	DM
I_4				CM	ID	ALU
I_5					CM	ID

Lecture-10

1) Data Dependency -

① Consider the program segment in which the insⁿ j follows insⁿ i.

② Data dependency is existed when the insⁿ j tries to read the data before insⁿ i writes it.

Note:-

① Insⁿ j, 1 of the input may be the output of the insⁿ i.

Eg:- I_1: Add (R0), R1, R2
I_2: Mul R3, (R0), R4.

② Acc. to the pipeline concept, I_2 is inserted into the pipe before completion of the I_1. Therefore, I_2 reads the register R0 before i.e. updated by I_1. So, old value is undergoes processing instead of the new value.

③ To avoid this problem, keep I_2 into the waiting state until completion of the I_1. This waiting creates the stalls in the pipeline.

④ To identify the data dependency condition some kind of the database will be maintained at the decoding stage.

⑤ The database contain the following field name

S.No.	Functional Unit	Destn	Independent Source 1	Ind. Source 2	Dep. S1	Dep. S2
I_1	+	R_0	R_1	R_2	—	—
I_2	*	R_3	—	R_4	R_0 (I)	—

I_1: Execution stage will be allocated
I_2: Execution stage will not be allocated.

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
I_1	IF	ID + S: R_1, R_2 (I) (I) D: R_0 R_1 = value R_2 = value	EX	MA			
I_2		IF	ID * S: R_0, R_4 (D), (I) D: R_3 R_4 = value			EX	
I_3			IF			ID	
I_4				Stall		IF	

⑥ To minimize the no. of stalls in the pipeline due to the data dependency, 1 hardware technique used called as operand forwarding. It is also called as By Passing / Short circuiting.

⑦ Operand forwarding means accessing the new data from the interface register before updating the register file.

⑧ Consider the program segment :-

I_1 : Add $R0, R1, R2$
I_2 : SUB $R3, R0, R4$
I_3 : Mul $R5, R0, R6$
I_4 : Div $R7, R0, R8$

True data dependency → possible b/w Adjacent ins^n

	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
I_1	IF	ID	EX	MA	WB	
I_2		IF	ID	EX	MA	
I_3			IF	ID	EX	
I_4				IF	ID	

⑨ In the above sequence diagram, the dependent ins are accessing the data from the interface registors whereas I_4 is accessing the data from register file i.e. clock cycle (CC_5) is

divided into 2 halfs.

(10) In the 1st half of CC_5, insⁿ (I_1) updates the reg. no & insⁿ (I_4) dicodes the operation.

(11) In the 2nd half of CC_5, I_4 is accessing the register file, at that time, register no already contain the new value.

3) Control Dependency –

1.) When the branch operations are executed, the program control will be transferred from one location to another location.

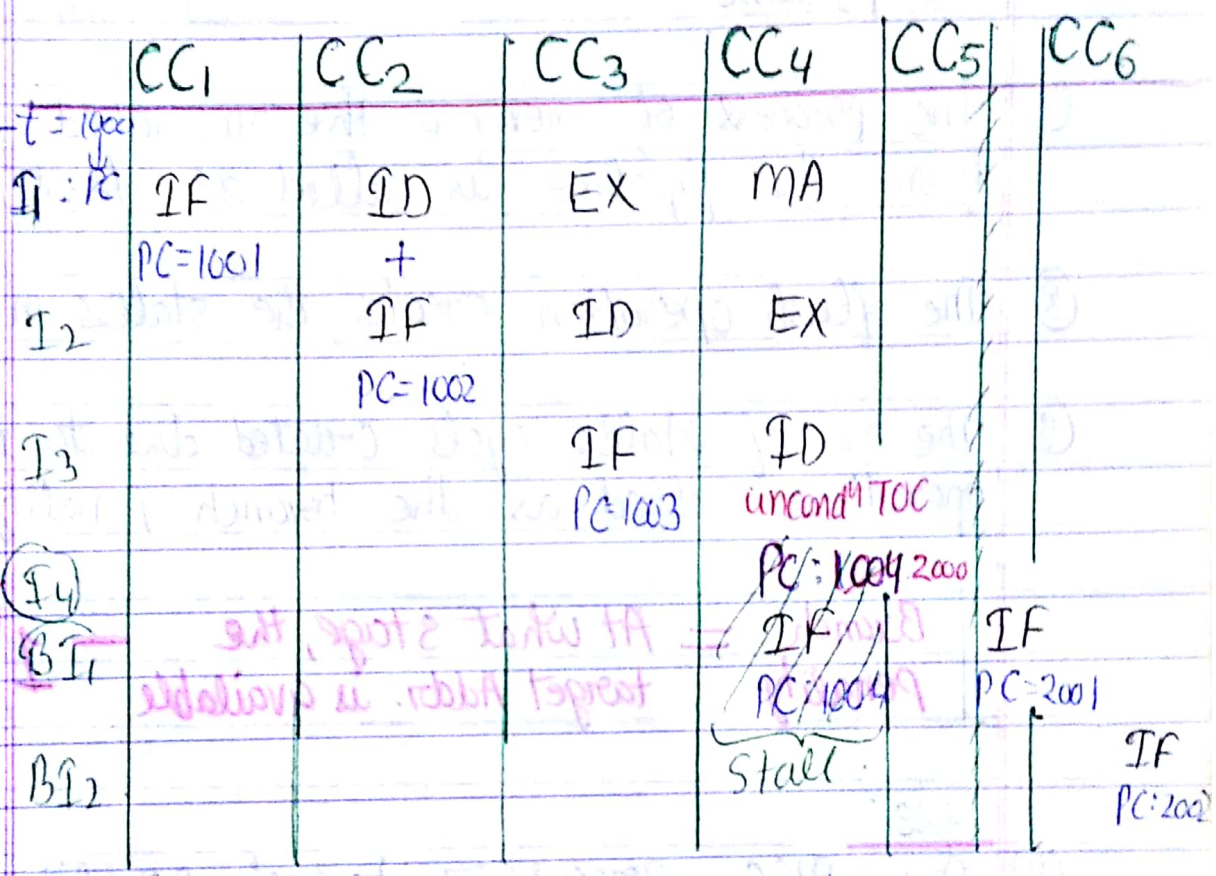
2) When the unconditional & conditional TOC of are executed in the pipeline some extra cycles are created.

Unconditional TOC –

(1) Consider the program segment

```
1000 : I1
1001 : I2                    ↓ falling path
1002 : I3  (JMP 2000)
1003 : I4
  ⋮    ⋮
2000 : BI1                   ↓ taken Path.
2001 : BI2
  ⋮    ⋮
```

② The Program execution is -
$I_1 - I_2 - I_3 - BI_1 - BI_2$.



③ Instⁿ fetch stage is overlapping with the decoding stage. That means before decode the current instⁿ, the next sequential instⁿ is already inserted into the pipeline.

④ When the decoding stage decodes the instⁿ as data transfer / data manipulation then the next sequential instⁿ is always the wanted instⁿ.

⑤ When the decoding stage decodes the instⁿ as unconditional TOC, then the next sequential instⁿ becomes the unwanted instⁿ.

⑥ To maintain the correct functionality, there is a need of remove the unwanted insⁿ from the pipeline.

⑦ The process of remove the unwanted insⁿ from the pipeline is called as flush / freeze

⑧ The flush operation creates the stalls in the pipe

⑨ The no. of stalls cycle created due to the branch operations called as the branch penality.

Branch penality	= At what stage, the target Addr. is available	— 1

Note:-

① The RISC processor branch penality is always 1 becoz. the target address is available at the 2nd stage.

② The # stall cycles created during the execution of the program due to the branch operations is

No. of stall cycles/Branch = Branch freq. * Branch penal

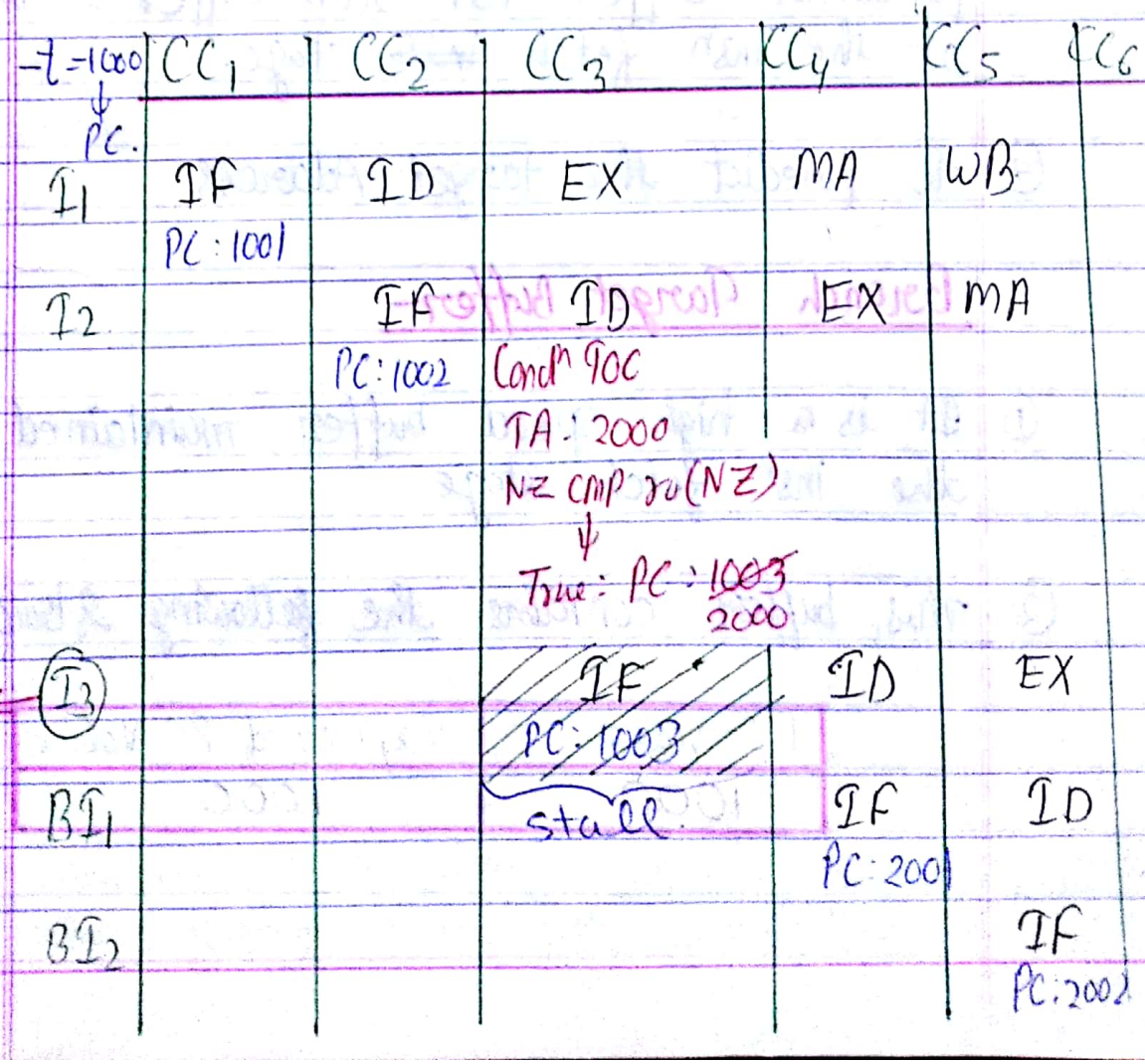
Conditional TOC -

① Consider the program segment:-

1000 : I_1
1001 : I_2 (JNZ R0, 2000) ↓ Falling Path
1002 : I_3
1003 : I_4
⋮ ⋮
2000 : BI_1 ↓ Taken Path.
2001 : BI_2
⋮ ⋮

② Execution Sequence with True Condition :-
$I_1 - I_2 - BI_1 - BI_2$

③ Execution Sequence with False Condition :-
$I_1 - I_2 - I_3 - I_4$



④ When the decoding stage decodes the insⁿ as conditional TOC with True Condition then the next sequential insⁿ becomes the unwanted insⁿ. Therefore, flush out the unwanted insⁿ from the pipe. It creates the stall.

⑤ When the decoding stage decodes the current insⁿ as conditional TOC with false condition the next sequential insⁿ becomes the wanted insⁿ so no flush → no stall.

⑥ To minimize the # stalls in the pipeline due to ctrl dependency, branch target buffer / branch prediction buffer or loop buffer is used at the insⁿ fetch ~~insn~~ stage.

⑦ To predict the target Address.

Branch Target Buffer-

① It is a high speed buffer maintained at the insⁿ fetch stage.

② This buffer contains the following structure.

PC-Value	Expected PC-Value
1003	1000

Eg:- 1000 : I_1
1001 : I_2
1002 : I_3 (DJNZ r_0, 1000) $r_0 = 4$.
1003 : I_4.

Execution Sequence:-

PC : 1000
1001
1002 : $r_0 = 3$ (NZ) True.
stall ~~1003~~ 1000
1001
1002 : $r_0 = 2$ (NZ) True.
~~1003~~ 1000
1001
1002 : $r_0 = 1$ (NZ) True.
~~1003~~ 1000
1001
1002 : $r_0 = 0$ (NZ) F
1003 ~~1000~~ 1003

Insert the entry into target buffer & fetch the ins^n from expected-PC

delete the entry from the target ~~address~~ buffer. & fetch the ins^n from the corresponding PC-value.

② This concept is suitable for the loop execution, when there is a stall wrt True cond^n,

then the unwanted insⁿ address is maintained in the PC-value field & target address will be maintained in the expected PC value field.

④ In the next fetch onwards, the coorent PC value is compared with the PC entry field in the target buffer.

⑤ When it is matching, the CPU fetch the ins based on the coorresponding expected PC-valu otherwise CPU fetch the insⁿ based on the coorent value.

⑥ When there is a stall in the pipeline wort False condition, then delete the target addr from the expected PC value field & fetch the next insⁿ based on the coorresponding PC-value field.

Delayed Branch -

(1) Delayed Branch is a compiler technique.

(2) In this technique, compiler rearranges the instⁿ if possible to rearrange (or) substitutes the NOP instⁿ after the branch instⁿ if not possible to rearrange to preserve the execution path in the pipeline.

User Program - Compiler Re-arrangement Program.

I_1 : Add r_0, r_1, r_2	I_1
I_2 : Add r_3, r_4, r_5	I_3
I3: JMP BI_1	I_2
I_4 : MUL r_6, r_7, r_8	I_4
⋮	⋮
BI_1 : Add r_9, r_{10}, r_{11}	BI_1
BI_2 : Sub r_{12}, r_{13}, r_{14}	BI_2
⋮	⋮

Re-arrangement

PC:	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
I_1	IF PC: I_3	ID	EX	MA	WB	
I_3		IF PC: I_2	ID uncod. TOC PC: ~~I_4~~ BI_1	EX	MA	
I_2			IF PC: ~~I_4~~ BI_1	ID	EX	
BI_1				IF PC: BI_2	ID	
BI_2					IF	

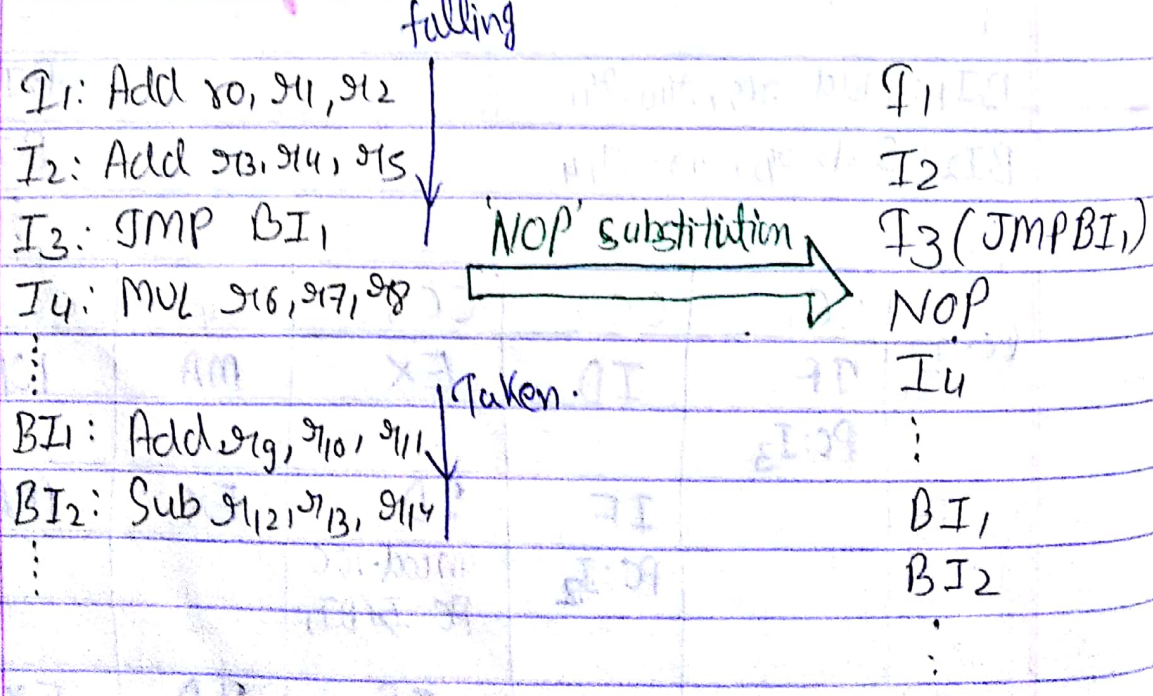
③ In the above program segment, executing the I_2 before I_3 & after I_3, the functionality is remain same. But when the I_2 is executing before I_3 then stalls are created in the pipeline. When it is executing after I_3 then no stalls present in the pipeline.

NOP Substitution:-

① Compiler substitute the NOP insⁿ after branch if not possible to rearrange. to preserve executable path in pipeline.

② NOP substitution is a logical ~~expansi~~ suspension of the unwanted insⁿ.

User Program — Compiler



PC: I_1	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6
I_1	IF	ID	EX	MA	WB	
	PC: I_2					
I_2		IF	ID	EX	MA	WB
		PC: I_3				
I_3			IF	ID	EX	MA
			PC: NOP	Uncond. JOC		
				PC: I_4 BI_1		
NOP				IF	ID	EX
				PC: I_4		
BI_1					IF	ID
					PC: BI_2	
BI_2						IF
						PC: BI_3

Ins^n Scheduling -

① Processor executes the program always in a sequential order called as InOrder Execution.

② In the inorder execution, sequence, if any ins^n is data dependent then the remaining ins^n are also showing the stall cycles which are created by the dependent ins^n.

Eg:-

I_1: Add r_9, r_{11}, r_{12}
I_2: Sub r_{13}, r_{10}, r_{14}
I_3: MUL r_{14}, r_{15}, r_{16}
I_4: DIV r_{13}, r_{17}, r_{18}.

Inorder Exec^n Seq:-

I_1 — I_2 — I_3 — I_4.
wait
stalls.

③ In above prog seg, I_2 is data dependent on I_1 so it is undergoes waiting until completion of I_1. Due to this I_3 & I_4 both are also sharing the stall cycles of I_2 even if they are independent.

④ To handle the above problem, there is a need of schedule the insⁿ. means that execute the independent insⁿ first. Scheduling causes the out-of-order (Reorder) execution.

⑤ Out of Order execⁿ sequence is
$I_1 - I_3 - I_4 - I_2$.

⑥ Out-of-Order execⁿ creates two more dependencies in the pipeline:
1.) Anti-dependency.
2.) Output-dependency.

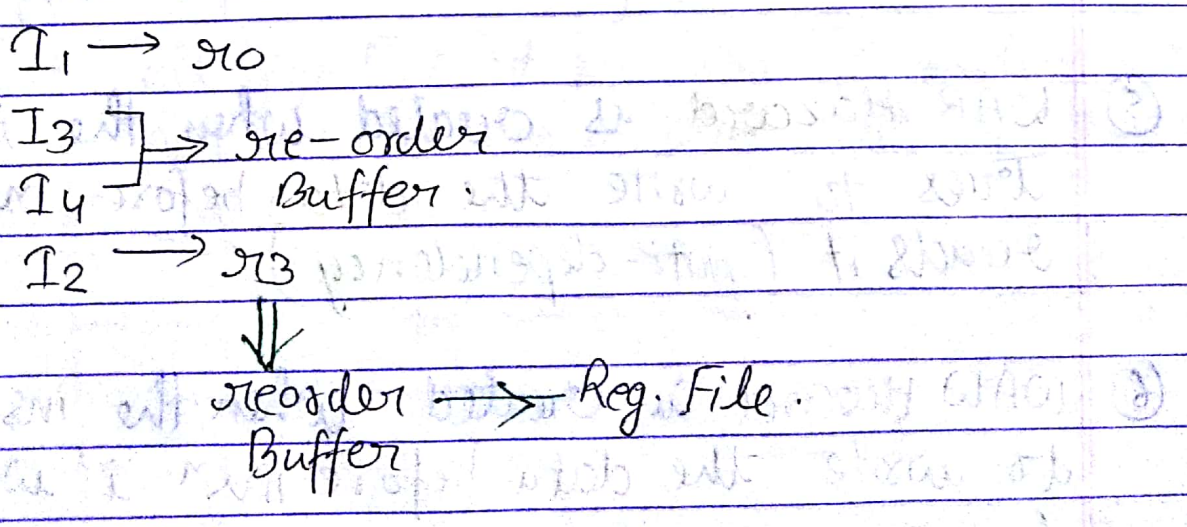
⑦ Anti-dependency is occured when the insⁿ 'j' tries to write the data before insⁿ 'i' reads it.
Ex- I_3 is executed before I_2, So, I_3 modifies the reg. R_4 before I_2 reads it. Therefore, at the time of executing I_2 wrong value will be injected (data-loss).

(8) Output dependency is occured when the insⁿ 'j' tries to write the data before insⁿ 'i' writes it.
Eg:- I_4 is executing before I_2. So, I_4 modifies the reg. r_3 before I_2 writes it. Therefore the destination (r_3) is finally containing the old value (data loss).

(9) To handle the above dependencies, one of the H/w mechanism is used called as register-renaming.

(10) Reg-renaming states that use some temporary storage (reorder buffer) to store the out of order insⁿ results. After completion of the dependent insⁿ execⁿ, the register file is updated with the reorder buffer contents. Therefore, data is safe.

Out-of-order execⁿ -



Hazards :-

① Hazard is a delay. Delay is created in the pipeline due to the dependencies.

② 3 Kinds of Dependencies creates the 3 Kinds of Hazards in pipeline :-

1) Structural Hazard.
2) Data Hazard.
3) Control Hazard.

③ Data Hazards are classified into 3 Kinds based on the order of read & write opⁿ.

(1) RAW (Read After Write) Hazard.
(2) WAR (Write " Read) "
(3) WAW (" " Write) "

④ RAW Hazard is created when the insⁿ 'j' tries to read the data before insⁿ 'i' writes it (TRUE Data Dependency).

⑤ WAR Hazard is created when the insⁿ 'j' tries to write the data before insⁿ 'i' reads it (Anti-dependency).

⑥ WAW Hazard is created when the insⁿ j tries to write the data before insⁿ 'i' writes it (output dependency).

	Ins^n 'j'		Ins^n 'i'	
①	In-Reg	—	OUT-Reg	(TRUE DATA)
②	Out-Reg	—	In-Reg	(ANTI)
③	Out-Reg	—	Out-Reg	(OUTPUT)

Performance Evaluation of Pipeline with Stall-

$$S = \frac{\text{Avg Ins}^n \text{ ET}_{\text{non-pipe}}}{\text{Avg Ins}^n \text{ ET}_{\text{pipe}}}$$

$$S = \frac{\text{CPI}_{\text{non-pipe}} * \text{cycle time}_{\text{non-pipe}}}{\text{CPI}_{\text{pipe}} * \text{cycle time}_{\text{pipe}}}$$

① The ideal CPI of pipeline is always almost 1, but due to dependency stalls are created in the pipeline. Therefore,

$$S = \frac{\text{CPI}_{\text{non-pipe}} * \text{cycle time}_{\text{non-pipe}}}{(1 + \#\text{stalls/Ins}^n) * \text{cycle time}_{\text{pipe}}}$$

② When all pipeline stages are perfectly balanced, then pipeline & non-pipeline both of the cycle times are Equal.

$$S = \frac{\text{CPI}_{\text{non-pipe}}}{(1 + \#\text{stalls/Ins}^n)}$$

③ When all the ins^n are taking same no. of cycles, then one ins^n exec^n time is also equal to

no. of stages in the pipeline. Therefore,

$$S = \frac{\text{pipeline depth}}{(1 + \#\text{stalls/Ins}^{n})}$$

(4) When the processor is operating with 100% efficiency then no stalls present. Therefore

$$S = \text{pipeline depth}$$