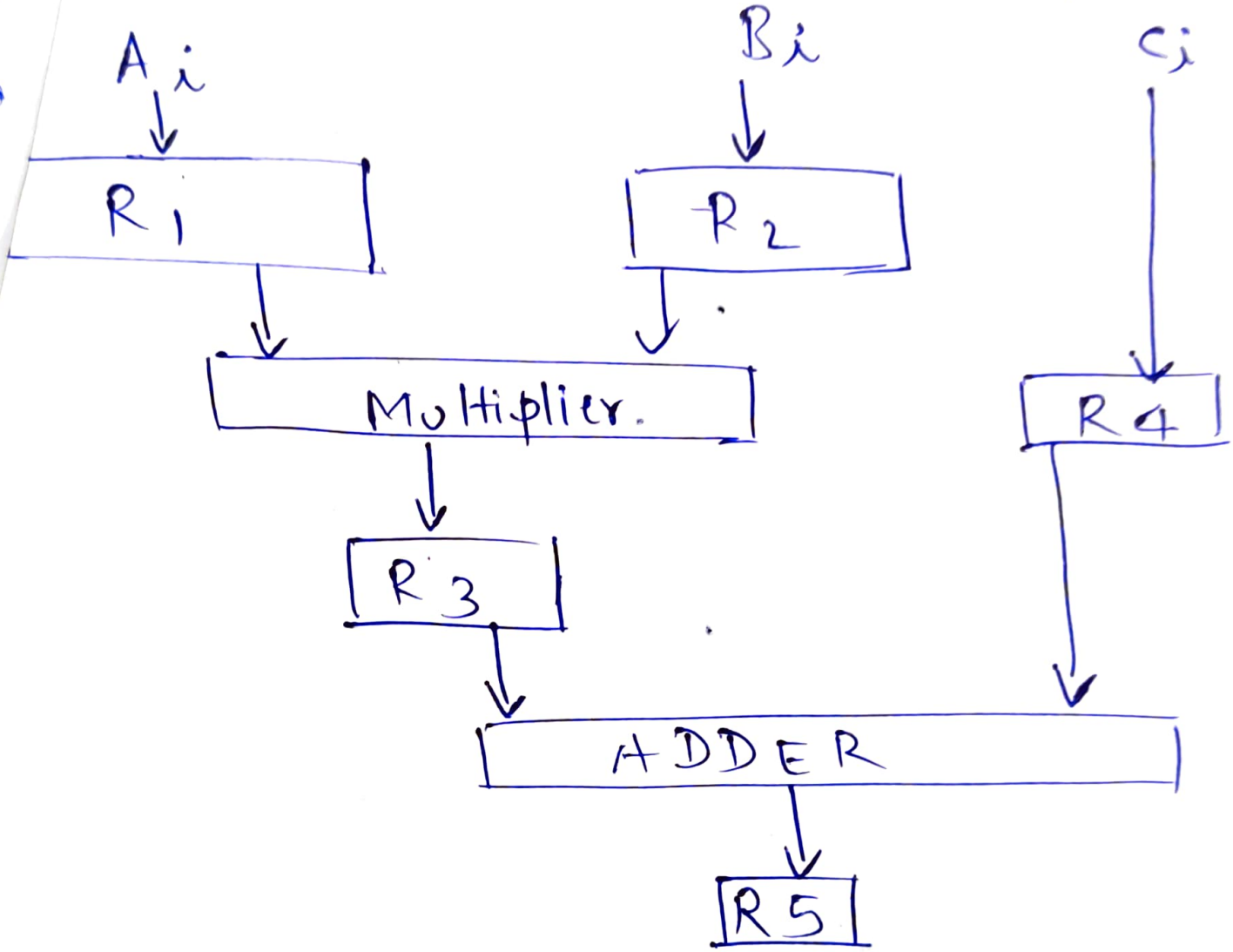
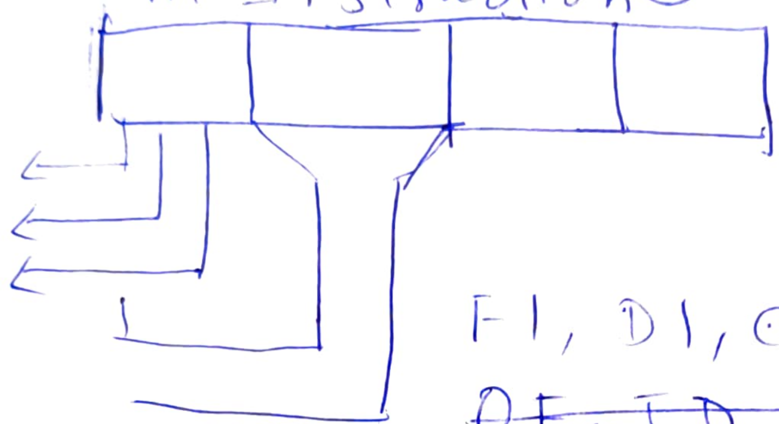
$A_i * B_i + C_i$ for $i = 1, 2, \ldots 5$



Clock Pulse	Seg 1		Seg 2		Seg 3
	R_1	R_2	R_3	R_4	R_5
1	A_1	B_1			
2	A_2	B_2	$A_1 * B_1$	C_1	
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 \times B_4$	C_4	$A_3 * B_3 + C_3$
6	—		$A_5 \times B_5$	C_5	$A_4 * B_4 + C_4$
7			—		$A_5 * B_5 + C_5$
8					

m-Instruction ②



SISD (von neumann)
SIMD (parallel comp)
MISD (hypothetical)
MIMD (super scalar or super comp)

FI, DI, CO, EI, WO

~~OF, ID, EX, MEM, RW~~

---

⑦ No. of cycles → no. of input

no. of input = 5 so → 7 cycles

for non-pipelined

3 cycle for 1 input

So → 3×5 = 15 cycles.

	Sequential	Pipeline
n = 5	3×5 = 15	7
n = 6	3×6 = 18	8
n = 7	3×7 = 21	9

Q
1(a) 3 temp register ~~Control Sig~~ ~~HCU~~ (2)
Vertical ex. (3)
2) Raw war Raw example (2)
pipeline ~~example~~ numerical
3) ~~...~~ discuss various hazards
Instruction pipeline numerical with Branch and its resolution (3)

Pipeline Cycle Time

(*) min. time in which all segments can perform their respective sub-operations.

→ in previous examples each clock pulse needs min. amt. of time to complete one segment. (or perform their sub oprs)

denote => t_p

General Consideration About Pipeline

Consider k-segment pipeline with clock cycle time = t_p to perform n tasks

→ Time req. to perform 1st task = $k t_p$

→ Time req. " " remaining $(n-1)$ tasks

$= (n-1) t_p$

Time req. for all 'n' tasks

$= (k + n - 1)\, t_p$ $\quad [t_p k + (n-1) t_p]$

→ verifying it from previous ex.

$k = 3,\ n = 5,$

total cycles $= (3 + (5-1)) =$ 7 cycles

total time $= 7 \times n_p$

* As soon as one output comes out of pipeline next output starts coming out at each time cycle

→ Consider a non-pipeline system that takes t_n time to perform a task (sequential, non-pipelined)

Time req. for 'n' task $= n * t_n$

eg:- water flowing out of pipe, firstly it will take some time for water to come out but then the water will keep flowing till you stop the tap or 'n' tasks are complete.

[It is irrespective of pipe length]

Speed up Ratio

if speed up ratio is 'n' it means pipelined architecture is 'n' times faster than non-pipelined architecture

$$\text{Speed up ratio} = \frac{\text{non-pipeline time}}{\text{pipeline time}}$$

$$S = \frac{n * t_n}{(k+n-1)\,t_p} \quad - (1)$$

as the number of tasks increases, 'n' value is extremely large

$n >> k$ [ignore $k-1$ from (1)]

$$S = \frac{n\,t_n}{n\,t_p} = \frac{t_n}{t_p} \quad - (2)$$

or S_{max} [ideal case]

[ignore $k-1$ cycle to fill pipe]

→ with initial $k-1$ cycle we are not getting avg output after that one cycle one output

→ Special case: If to perform one task pipeline & non-pipeline system take equal time

$$t_n = k * t_p$$

in (2)

$$S_{ideal} = \frac{k\,t_p}{t_p} = k$$

» Speed up can never be more than 'k'.

Q) A non-pipeline system takes 50 ns to process a task. The same task can be processed in a 6-Segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

$t_n = 50$ ns $k = 6$, $t_p = 10$ ns

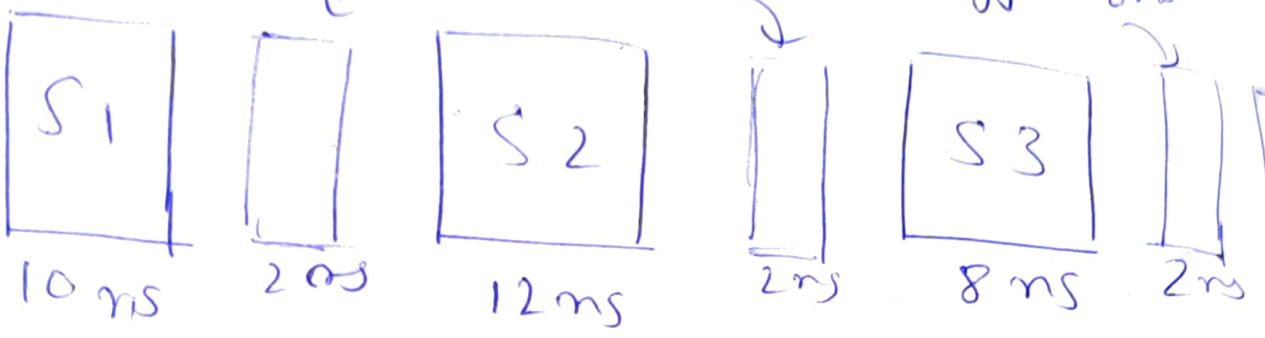
$n = 100$

$$S = \frac{n * t_n}{(k+n-1)\,t_p} = \frac{100 \times 50}{(6+99)\,10}$$

$$= \frac{500}{105} = 4.76$$

$$S_{max} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$

Synchronous Pipeline



$$t_p = \max(\text{seg delay})$$
$$= \max(10, 12, 8, 9)$$
$$= 12 \text{ ns}$$

t_p = max of seg. delays + reg delay
$= \max(10, 12, 8, 9) + 2 = 14$ ns

→ For non pipelined no intermediate register needed

t_n = Sum of all seg delay
$= 10 + 12 + 8 + 9 = 39$ ns

Cycle time in Synchronous Pipeline

t_p = max of seg. delays + reg delay

t_n = Sum of seg. delay.

① If in question value of 'n' is given
$\frac{n t_n}{(k+n-1)t_p}$ If 'n' not given $S_{max} = \frac{t_n}{t_p}$

atency & Throughput

time after which machine takes next input

Latency in non-pipelined = t_n

Latency in pipelined = t_p

Throughput

No. of inputs processed per unit time

$$n/(k+n-1)t_p$$

in $(k+n-1)t_p$ time, no. of inputs processed = n

$$\frac{\text{no. of inputs}}{\text{time}} = \frac{n}{(k+n-1)t_p}$$

ideal case $(n >> k)$

$$\frac{n}{n\,t_p} \Rightarrow \text{throughput} = \frac{1}{t_p}$$

$[n >> k]$

Q) Consider a 5 stage pipeline with cycle time of 15 ns. Calculate processing time of pipelined for 500 tasks)

A) $n = 500$, $k = 5$, $t_p = 15$

pipeline time

$= (k + n - 1)t_p = (5 + 500 - 1)15$

$= 7560$ ns

Instruction Pipeline

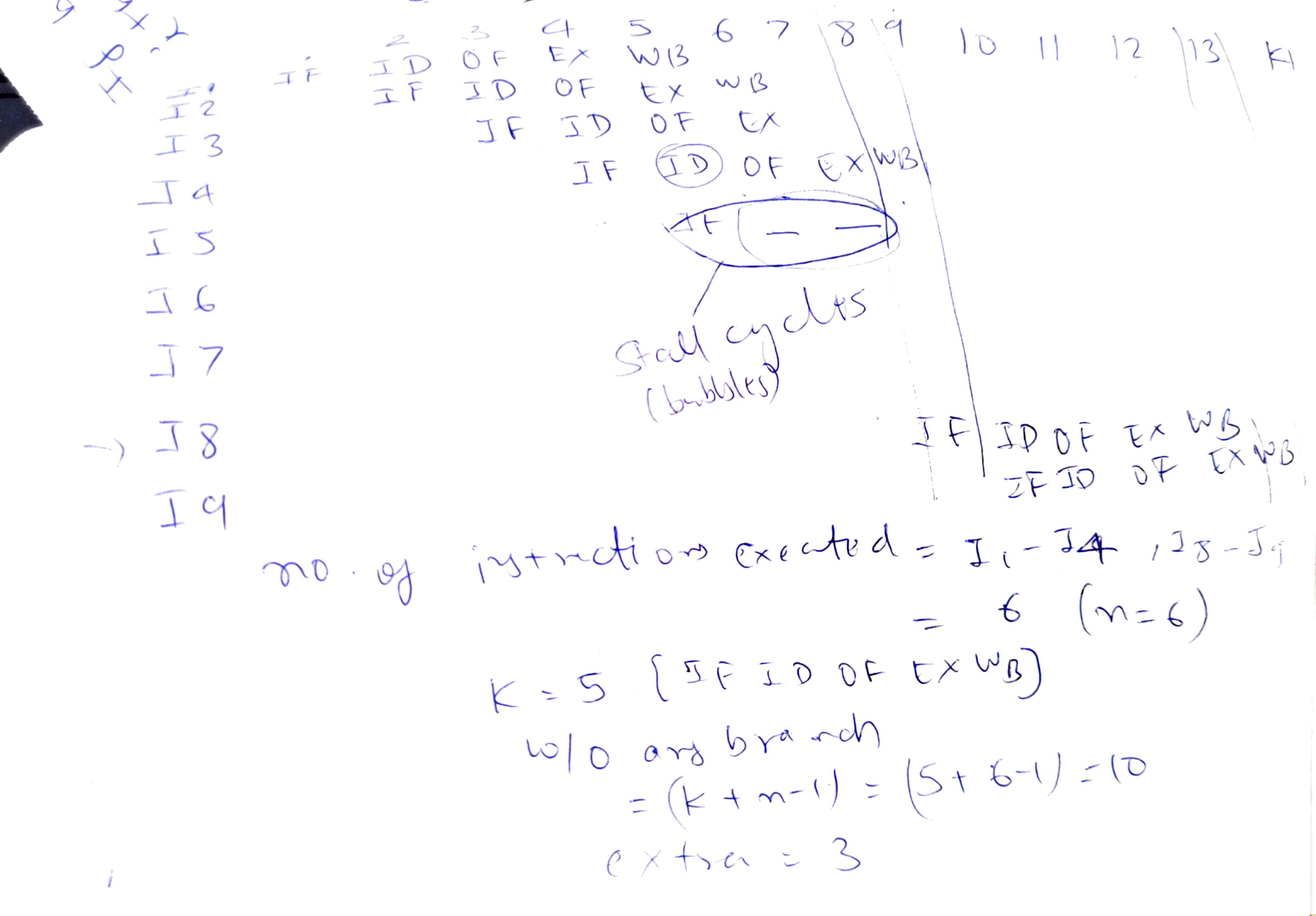
IF : Instruction Fetch

ID : Instruction Decode

OF : Operand Fetch

EX : Execution

WB : Write Back



no. of instruction executed = $I_1 - I_4$, $I_8 - I_9$

= 6 ($n = 6$)

$K = 5$ (IF ID OF EX WB)

w/o any branch

$= (k + n - 1) = (5 + 6 - 1) = 10$

extra = 3

If branch is evaluated at i^{th} stage of pipeline.

no. of stalls because of branch = $i-1$

Q) in a k-segment pipeline, if branch outcome is available after i^{th} stage.

By standard ⇒ branch decision is made in 'Execution' Phase

Actually only 6 instructions executed

So $n = 6$, $k = 5$,

→ After completion of branch instruction Next instruction is fetched

If one cycles time = 10 ns

so total execution time

= no. of cycles * 10

= 13 × 10 = 130 ns

actual

10 × 10 = 100 ns

Q) FI : 5

DI : 7 LD : 1 ns

FO : 10

EI : 8

WO : 6

t_p = max (5, 7, 10, 8, 6) + 1

= 10 + 1 = 11

n = 12, $I_4 \rightarrow I_9$ [Stall = 4 - 1 = 3]

(k + n - 1) = (5 + 8 - 1) = 12

No. of cyles

= 15 × 11 = 165

Pipeline Hazards

Situations that prevent the next instruction from being executing during its designated clock cycle.

↳ hazards leads to have stall cycles.

1) Structural hazards / Resource Conflict
2) Data hazard / Data dependency
3) Control Hazard / Branch Difficulty

Structural Hazard

(X) 2 diff. ~~segments~~ inputs try to use same resource at same time

ex.

	T1	T2	T3	T4	T5	T6	T7
MUL	IF	ID	DF	EX	EX	EX	WB
~~ADD~~	~~IF~~	~~ID~~	~~DF~~				
ADD		IF	ID	DF	-	-	EX W

Example

	0	1	2	3	4	5	6	7	8
I1		IF	ID	(OF)	EX	WB			
I2			IF	ID	OF	EX	WB		
I3				(IF)	ID	OF	EX	WB	
I4					IF	ID	OF	EX	WB

If at T3, I1 and I3 both are fetching instruction & operand from memory. So there will be a resource hazard causing stalls and hence some stall cycles.

Resolution

↳ Inclusion of two cache memories one for operands (data) and one for instructions.

Data Hazard / Data dependency

(we are considering General Pipeline hardware)

result of an instruction is used as inputtt in next.

i: R1 ← R2 * R3

i+1: R4 ← R1 + R5

	T1	T2	T3	T4	T5	T6
i	IF	ID	OF	EX	WB	
i+1	IF	ID	—	—	OF	

Stall cycles

(*) we cannot use R1 value in the next instruction (i+1) as it is yet not updated in R1 (i.e. WB stage) So we have to wait for 2 cycles to get updated R1 value

→ So R1 data dependency was there

(*) A general pipeline hardwire can't detect data hazard

Solution

(*) Software Solution (provided by compiler)

Compiler generates binary seq. for CPU so compiler detects hardware

and judge whether hardware can handle data dependency or not otherwise it will generate instructions in such a manner that there will not be any data dependency.

compiler soln → delayed load

* Hardware Solution (Smart h/w)
  - hardware interlock
  - operand forwarding.

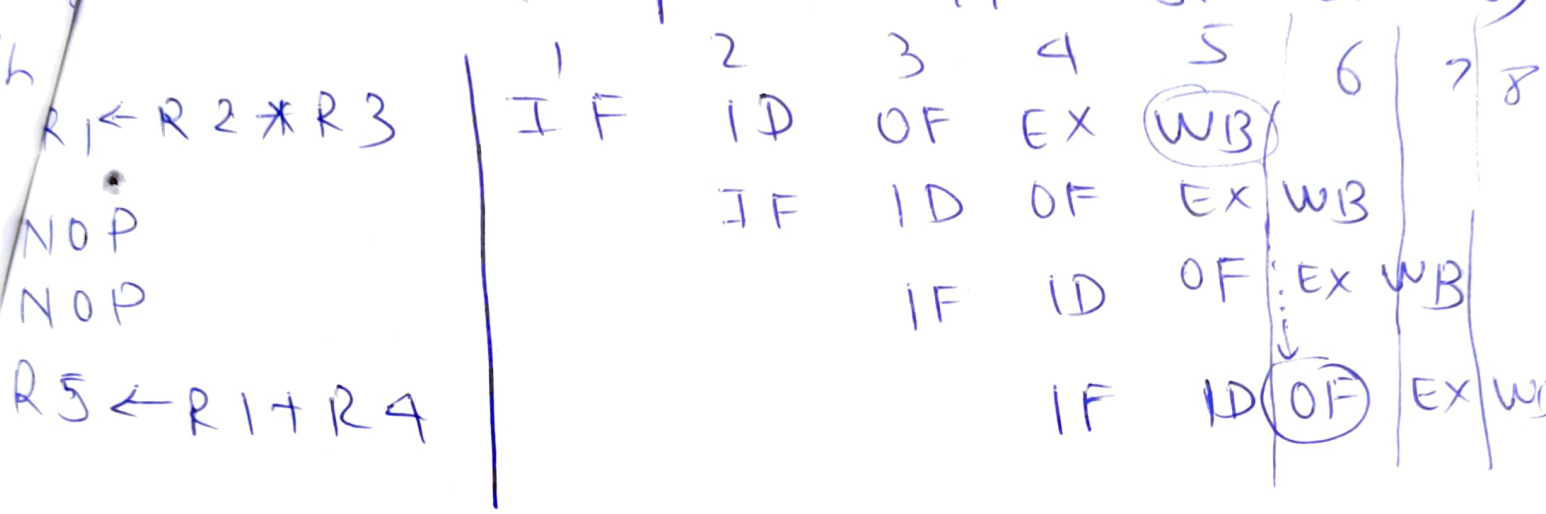
Delayed Load

I1	R1 ← R2 * R3
I2	R5 ← R1 + R4

→ there is data dependency in R1 so there shld be 2 cyle delay in OF of I2

→ Compiler will add two independent instruction in between I1 & I2

compiler will insert no-operation instructions in between I1 & I2 (dependent instructions)



→ So compiler solved the data dependency.

→ IN NOP no register values will change nothing will change.

↳ give example of no item on assembly line but still all operations will perform

Smart hardware

Hardware interlock:

	1	2	3	4
I1 R1 ← R2 * R3	IF	ID	OF	EX WB
I2 R4 ← R1 * R5	IF	ID	—	— OF EX W

OF hardware will be locked for I2 in 4th cycle as no control signal will be received.

Operand Forwarding (bypassing is a (expensive

$R1 \leftarrow R2 + R3$ IF ID OF EX WB

$R5 \leftarrow R1 * R4$ IF ID OF EX WB

Example: You and friend giving viva, ~~Of Exam~~ he is inside and you outside. Both your projects are in one pendrive. You tell the viva cordinator that you also have the pendrive (Bluff) when your friend comes out you take pendrive and proceed for viva.)

→ In operand forwarding the result of EX stage for I1 is forwarded to WB as well as EX stage of of I2 instr. inside ALU itself. So we saved 2 cycles

This will only happen for data dependency.

Decision unit will decide that operand forwarding will happen or ~~not~~.

ALU to ALU data dependency

↳ no stall cycle because of data dependency if operand forward used.

ex LOAD R1 ← M[add]
R4 ← R1 * R5

Operand forward will not help coz data coming from outside processor.

STORE R4 ← R1 * R5
MEM[ADD] ← R4

(*) in viva example your friend not inside viva room

Data Hazards

Write after write (WAW) Hazard

I_1 : $R_3 \leftarrow R_1 * R_2$
I_2 : $R_3 \leftarrow R_4 + R_5$

→ RAW (Read after Write (True dependency)

ADD R1, R2, R3 — R1 ← R2 + R3
ADD R4, R1, R5 — R4 ← R1 + R5
→ Reading of R1 after writing of R1

→ WAR (Write after Read) (Anti dependency)

Add R1 R2 R3
Add R2 R6 R5
Read R2 earlier then writing R2

→ (WAW) (Write after write) (output dependency)
Add R1 R2 R3
Add R1 R6 R5
R1 writes after R1 in both instruction

→ RAR (Read after Read) (No dependency)
Add R1 R2 R3
Add R7 R6 R3
R3 read after R3 read

Control Hazard or Branch difficulty

→ After the branch instruction CPU not able to decide which instruction to execute as this depends on whether condition is met for branching or not

→ So this leads to generation of Stall cycles. and control hazard

→ General hardware can't detect this control hazard (branch difficulty)

Handling

- Software
  - Compiler
    - delayed branch
- Hardware
  - Prefetch Target Instruction
  - Branch Prediction
  - Loop Buffer
  - Branch Target Buffer

Delayed Branch

Instruction (branch)	IF	ID	OF	EX	WB
NOP		IF	ID	OF	EX WB
NOP			IF	ID	OF
NOP				IF	ID OF

→ Number of NOP in inserted by compiler depending upon After how many ~~cycles~~ time cycles next instruction needs to be executed

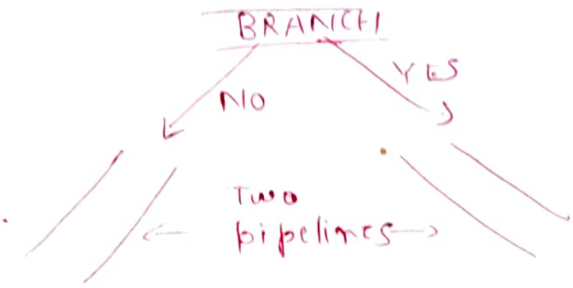
Based on condition the PC value will be updated and so the next instruction will be executed

→ Because of NOP no stall cycles generated

Hardware Solution

→ Prefetch Target Instruction

↳ Hardware is able to handle branch



→ Two separate pipelines will be started

↳ Decode phase

→ Understands which instruction (whether branch or not)

→ Calculate address (target)

→ So in decode phase processor knows where to jump but doesn't know whether it should jump or not

→ 2 pipelines will go together

↳ As per condition one pipeline is discarded and one is accepted

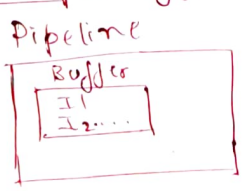
Branch Prediction

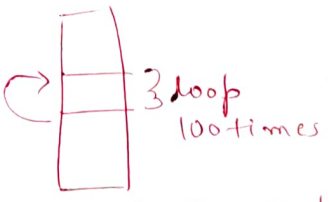
→ Hardware will predict whether instruction will jump or not

→ Static
- Branch not taken (always)
- Branch taken (always)

→ Dynamic (intelligent)
- decide on current execution of program
- current hardware (80-90% correct prediction)

Loop Buffer





→ If we are running a loop we have to jump 100 times, so to accomodate this there is a special buffer in pipeline which feeds these instructions and executes them for 100 times without fetch

Branch Target Buffer

Branch	Target
I_4	I_{12}
I_5	I_{16}

we keep track of where the jump happened in previous instance. By looking at the previous data we may be able to guess branch.