4 Pipelining

4.1 FUNCTION OF THE PIPELINE

- Pipeline uses decomposition technique, it means problem statement is divided into independent subproblems, assign them the independent hardware, later connect the hardware in a pipelining sequence, i.e. first hardware unit output is connected as an input to second hardware unit, and so on.
- The definition states that, insert the new input into the pipeline before computation of an old input, so new input is executed along with the old input called as overlapping execution. Here, we can mention input is referring to as an instruction because use of input is not that convincing.
- Overlapping execution sequence is described using the space-time diagram.
- Successful characteristic of pipelining is, in every new cycle, new instruction is inserted into the pipeline.

Definition

Accepts the new input at one end before the previous accepted input appears as an output at the other end.

So, CPI = 1 (clocks per instruction)

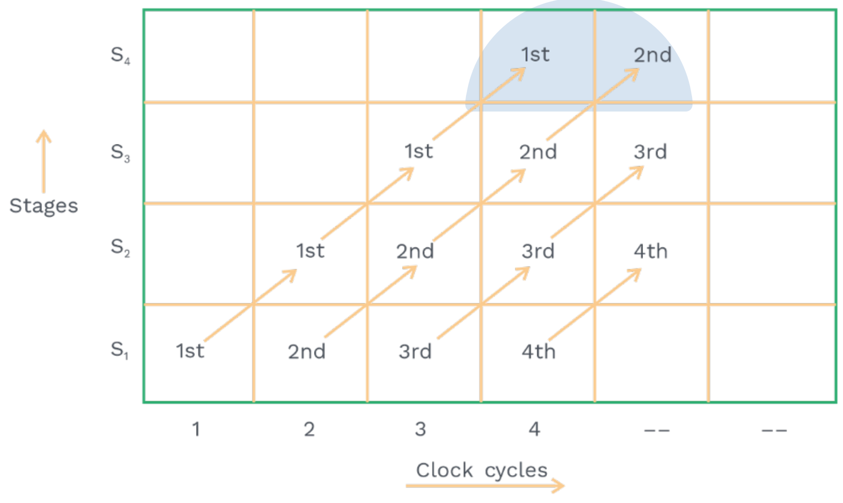


Fig. 4.1

Note:

In the non-pipeline system, new input is inserted after the completion of old input.

Pipeline layout:

- Every pipeline has two ends, known as input and output end. Multiple pipes linking the two terminals are interconnected to satisfy the functionality of the pipeline. Each pipe in the pipeline is called as stage or segment.
- Interface register (buffer) is used between the stages to hold the intermediate output. It is also called as buffer, or latch or pipeline register.
- The respective stages and the interface latches are connected to a common clock. So, clock adjustment is very important in the pipeline design.
  - **a)** Uniform delay pipeline:

    $\boxed{\text{Cycle time}\left(t_p\right) = \text{Stage delay}}$
  - **b)** Non–uniform delay pipeline:

    $\boxed{\text{Cycle time}\left(t_p\right) = \text{Maximum}\left(\text{stage delay}\right)}$
  - **c)** If buffer delay present in the pipeline:

    $\boxed{\text{Cycle time}\left(t_p\right) = \text{Maximum}\left(\text{stage delay} + \text{buffer delay}\right)}$
  - **d)** If skew time/setup time overhead is present in the pipeline design

    $\boxed{t_p = t_p + \text{Overhead}}$

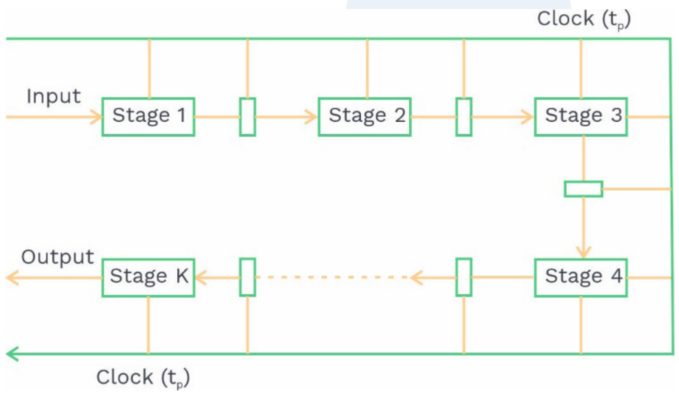


Fig. 4.2 Pipeline Layout

Performance analysis of pipeline:

- Consider a 'K' segment pipeline with the cycle time of 't_p' used to execute 'n' tasks.
- The first task in the pipeline is executed in a non–overlapping order, so it takes 'K' cycles to complete. The remaining 'n–1' tasks emerge from the pipeline at the rate of 1 task per cycle. So 'n–1' tasks take 'n–1' cycles to complete.

- So, total time required to complete ‘n’ tasks in a ‘K’ segment pipeline is:

$$\boxed{\begin{aligned} ET_{pipe} &= (K+n-1)\,\text{cycles} \\ &= (K+n-1)\,t_p \end{aligned}}$$

where ET_{pipe} = Execution time of pipeline.

- Now, consider a non-pipeline processor used to execute ‘n’ tasks in which each task takes ‘t_n’ time to complete, so total time required to complete ‘n’ tasks in the non–pipeline is:

$$\boxed{ET_{nonpipe} = n \cdot t_n}$$

where $ET_{nonpipe}$ = Execution time of non–pipeline

t_n = One task execution time in non–pipeline

Performance gain of a pipeline is:

$$\boxed{\text{Speedup}\,(S) = \frac{\text{Execution time of non-pipeline}}{\text{Execution time of pipeline}}}$$

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

$$\boxed{S = \frac{n \cdot t_n}{(K+n-1)\,t_p}}\ (\text{when n is finite})$$

- With the increase in the count of the tasks, n attains much larger value than ‘K–1’, so ‘n + K –1’ approaches to ‘n’.

Under this condition

$$\boxed{S = \frac{t_n}{t_p}},\ \text{when n approaches to infinity } (\infty)$$

- When the pipeline stages are perfectly balanced (uniform delay) then one task execution time in the non–pipelining system is equal to number of stages in the pipeline system, so t_n = K cycles:

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

- Considering this condition:

$$\text{Speedup}\,(S) = \frac{K \cdot t_p}{t_p}$$

$$\boxed{S = K}$$

- If efficiency ($\eta = 100\%$), then speedup (S) is equal to number of stages in the pipeline (depth of pipeline).
- When the system is operating with 100% efficiency, then maximum speed up is possible. Maximum speed-up is equivalent to the count of the pipeline stages.

$$\boxed{\eta_{pipe}\left(\text{Efficiency}\right) = \frac{\text{Speed–up } (S)}{\text{Number of stages}\,(K)}}$$

$$\boxed{\eta_{pipeline} = \frac{S}{K}}$$

- $$\text{Throughput of pipeline} = \frac{\text{Number of tasks executed}}{\text{Total time taken to execute all the tasks}}$$

$$= \boxed{\frac{n}{(K+n-1)\,t_p}}$$

PRACTICE QUESTIONS

Q1 **Suppose there are 6 stages of a pipeline as given below:**

Stage number	Stage execution time (ns)
S1	10
S2	20
S3	15
S4	35
S5	10
S6	10

If the buffer delay is 30 ns, then calculate the execution time for 1000 instructions using pipelining.

a) 65.325 ns **b) 65.325 µs**
c) 48.975 µs **d) 58.965 ns**

Sol: **b)**

When various stages of pipeline have non–uniform clock time, we take longest clock time or maximum clock time.

Clock time = Maximum of (10 ns, 20 ns, 15 ns, 35 ns, 10 ns, 10 ns) + buffer delay

= 35 ns + 30 ns

= 65 ns

$T_{pipeline}$ = 65 ns

Execution time of pipeline ($ET_{pipeline}$) = (K + n–1) × $T_{pipeline}$

Where K = Number of stages = 6

n = Number of instructions = 1000

$ET_{pipeline}$ = (6 + 1000 – 1) × 65 ns

= 1005 × 65 × 10^{-9} sec

= 65, 325 × 10^{-9} sec

= 65.325 × 10^{-6} sec

= 65.325 μs (micro–second)

Q2 **Assume a pipeline P which operates at 3 GHz clock rate. It has a speedup factor of 10 and efficiency of 40%. Calculate the number of stages in the above pipeline.**

a) 10 **b) 25** **c) 15** **d) 30**

Sol: **b)**

$$\text{Efficiency of pipeline} = \frac{\text{speed up factor}}{\text{Number of stages}}$$

$$\text{Number of stages} = \frac{\text{speed up factor}}{\text{Efficiency}}$$

$$\text{Number of stages} = \frac{10}{0.4}$$

$$= \frac{10}{4} \times 10 = 25$$

Q3 **Consider a 3–stage pipeline having delays of 100 ns, 200 ns and 500 ns, respectively. The third stage is splitted into two stages of 250 ns and 250 ns respectively. Calculate the throughput increase/decrease in percentage. Assume ideal pipelined processor.**

a) 100% increase **b) 60% decrease**
c) 40% increase **d) 50% decrease**

Sol: **c)**

Ist Case:

Delays = 100 ns, 200 ns, 500 ns

Clock cycle time = Maximum (100, 200, 500) ns = 500 ns

Execution time for 1 instruction = CPI × clock cycle time

CPI = 1 (for ideal pipeline)

= 1 × 500 ns

= 500 ns

For execution of 1 instruction, it takes 500 ns.

1 instruction = 500 ns

1 instruction = 500×10^{-9} s

$$1\text{ s} = \frac{1}{500 \times 10^{-9}}\text{instructions}$$

$$1\text{ s} = \frac{10^9}{500}\text{instructions}$$

$$1\text{ s} = \frac{1000}{500} \times 10^6\text{ instructions}$$

Old throughput = 2 MIPS (million instructions per second)

2nd Case:

3rd stage is partitioned into two stages of delay 250 ns and 250 ns.

Delays = 100 ns, 200 ns, 250 ns, 250 ns

Clock cycle time = Maximum (100, 200, 250, 250) ns

= 250 ns

CPI = 1 (for ideal pipeline)

Execution time for 1 instructions

= CPI × clock cycle time

= 1 × 250 ns

= 250 ns

For execution of 1 instruction, it takes 250 ns.

250 ns = 1 instruction

250×10^{-9} s = 1 instruction

$$1\text{ s} = \frac{1}{250 \times 10^{-9}}\text{instructions}$$

$$1\text{ s} = \frac{1000}{250} \times 10^{6}\text{ instructions}$$

$$= 4 \times 10^{6}\text{ instructions}$$

New-throughput = 4 MIPS (million Instructions per second)

Percentage increase in throughput

$$= \frac{\text{New} - \text{Old}}{\text{Old}} \times 100\%$$

$$= \frac{4 - 2}{2} \times 100\%$$

$$= \frac{2}{2} \times 100\%$$

$$= 100\%$$

Q4 **Consider a pipeline having 5 stages with duration 10 ns, 30 ns, 45 ns, 80 ns and 35 ns. If latch (buffer) delay is 20 ns. Calculate the speedup of pipelined over non–pipelined processor.**

a) 2 **b) 2.5** **c) 1.5** **d) 3**

Sol: **a)**

Pipeline execution time

$ET_{pipeline} = (K + n - 1) \times T_{pipeline}$

$T_{pipeline}$ = Maximum delay among all stages + delay due to register (latch delay)

$T_{pipeline}$ = Maximum of (10, 30, 45, 80, 35) ns + 20 ns

= 80 ns + 20 ns = 100 ns

Non–pipeline execution time = (10 + 30 + 45 + 80 + 35) ns

= 200 ns

Note:

In non–pipeline, we don't consider buffer delays.

$T_{non-pipeline}$ = 200 ns

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

Here, we assume n (number of instructions) is very large, i.e. $n \to \infty$

$$\text{Speed-up ratio} = \frac{n \times T_{\text{non-pipeline}}}{(K + n - 1) \times T_{\text{pipeline}}}$$

$$= \lim_{n \to \infty} \frac{n \times 200}{(K + n - 1) \times 100}$$

$$= \lim_{n \to \infty} \frac{n \times 200}{n\left(\frac{K}{n} + 1 - \frac{1}{n}\right) \times 100}$$

$$= \frac{200}{100} = 2$$

Here K = number of stages in pipeline.

Q5 **Assume we have two pipelines P1 and P2, respectively. P1 has 6 stages, having execution time of 12 ns, 14 ns, 19 ns, 20 ns, 22 ns and 25 ns. P2 has 4 stages each having execution time of 10 ns. Calculate the time (in ns) that can be saved while using P2 pipeline over P1 pipeline, if 2000 instructions are executed.**

a) 15250 ns **b) 8765 ns** **c) 30095 ns** **d) 20070 ns**

Sol: **c)**

Consider pipeline P1

K = number of stages = 6

n = number of instructions = 2000

Clock cycle time for P1

T_{P1} = Maximum of (12, 14, 19, 20, 22, 25) ns

= 25 ns

Execution time for P1 = (K + n – 1) × T_{P1}

= (6 + 2000 – 1) × 25 ns

E_{P1} = 50, 125 ns

Now consider pipeline P2

K = Number of stages = 4

n = Number of instructions = 2000

Clock cycle time for P2 = 10 ns

T_{P2} = 10 ns

Execution time (E_{P2}) = (K + n – 1) × T_{P2}

E_{P2} = (4 + 2000 – 1) × 10 ns

= 20030 ns

Time saved = E_{P1} – E_{P2}

$= 50,125 - 20030$
$= 30095$ ns

Q6 **If we have a non–pipeline processor which has cycle time of 40 ns and average CPI of 5.4. Assuming a 5-stage pipeline model, calculate the speedup of pipeline over non-pipeline processor.**

a) 5 **b) 4** **c) 4.5** **d) 6.5**

Sol: a)

We have 5 stages of equal delays in pipelined processors. Each stage will have clock cycle time $= \frac{40}{5}\text{ns} = 8\,\text{ns}$.

In the best case of pipeline, we have CPI = 1.
So in every cycle we will have one instruction as output, i.e. in every 8 ns.

$$\text{Speedup} = \frac{\text{Old cycle time}}{\text{New cycle time}}$$

$$= \frac{40\,\text{ns}}{8\,\text{ns}} = 5$$

Alternate solution:
Maximum speed up = number of stages = 5

Types of pipelines:

1) Linear pipeline:

- This type of pipeline contains forward connections only.
- Linear pipelines are synchronous pipelines.
- Linear pipeline latency is always 1.
- Latency means the clock cycle difference between two successive initiations in the pipeline.

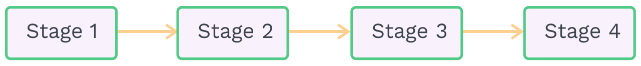


Fig. 4.3

2) Non-linear pipeline:

- This pipeline contains forward and backward connections, so the reservation table is used to process the inputs in the non-linear pipeline.
- Non-linear pipelines are asynchronous pipelines.

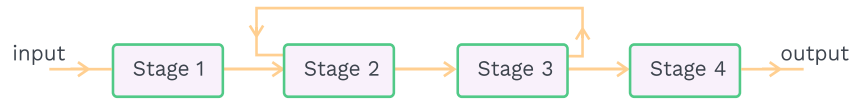


Fig. 4.4 Non-linear Pipeline

Note:

1) If there exist uniform delays among the pipeline stages, same amount of time will be required to execute one job in pipeline as well as non-pipeline layouts.

2) If there persist non-uniform delays among the stages, the time taken to accomplish the first job in the pipeline processor will be always greater than that in a non-pipelined architecture.

PRACTICE QUESTIONS

Q7 **A four-stage pipelining is given below:**

	S1	**S2**	**S3**	**S4**
I_1	**2**	**1**	**1**	**3**
I_2	**1**	**3**	**2**	**1**
I_3	**1**	**1**	**3**	**1**
I_4	**2**	**1**	**1**	**1**

What is the performance gain achieved by the above pipeline over non–pipeline execution?

a) 1.84 **b) 1.72** **c) 1.92** **d) 2.08**

Sol: **c)**

Pipeline execution:

	1	2	3	4	5	6	7	8	9	10	11	12	13
S4					I_1	I_1	I_1		I_2			I_3	I_4
S3				I_1			I_2	I_2	I_3	I_3	I_3	I_4	
S2			I_2	I_2	I_2	I_2	I_3	–	I_4	–	–		
S1	I_2	I_2	I_2	I_3	–	–	I_4	I_4					

$T_{pipe} \Rightarrow 13$ cycles

Non-pipeline execution:

$$T_{nonpipe} = (\overset{I_1}{7} + \overset{I_2}{7} + \overset{I_3}{6} + \overset{I_4}{5})\text{cycles}$$

$T_{nonpipe} \Rightarrow 25$ cycles

$$\text{Performance gain} = \frac{T_{nonpipe}}{T_{pipe}}$$

$$= \frac{25}{13} = 1.92$$

Dependencies in the pipeline:

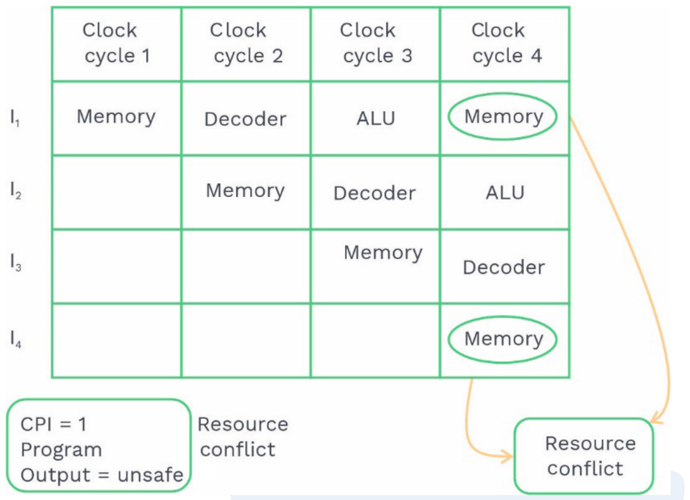
- Dependency is a major problem in the pipeline, it causes extra cycles.
- In a pipeline structure, any clock cycle deprived of new input initiation is called as extra cycle, also named as stall or hazard.
- When stall is present in the pipeline then CPI $\neq$ 1

1. 10 cycles $\rightarrow$ 10 inputs (CPI = 1)
2. 10 cycles $\rightarrow$ 7 inputs (CPI $\neq$ 1)
(with 3 stalls)
3. 10 cycles $\rightarrow$ 9 inputs (CPI $\neq$ 1)
(with 1 stall)

- There are three kinds of dependencies which are possible in the pipeline:
  1) Structural dependency
  2) Data dependency
  3) Control dependency

1) Structural dependency

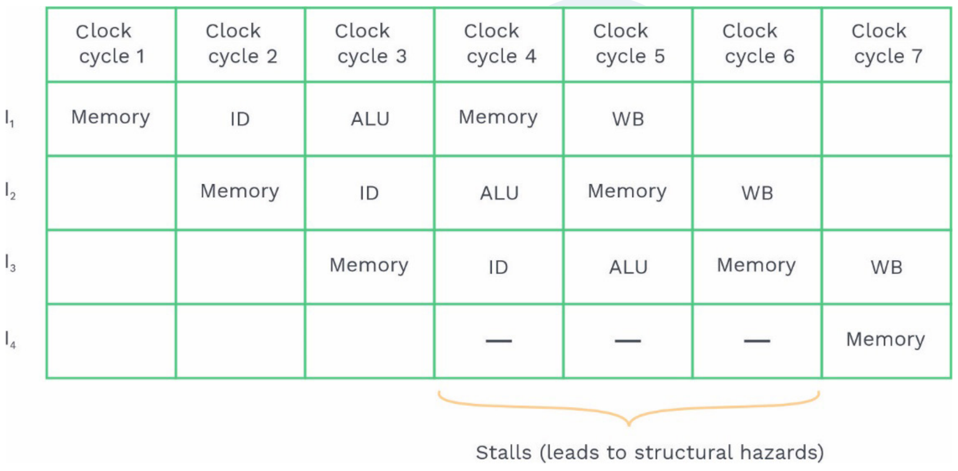
- It occurs in the pipeline due to a resource conflict. Resource may be a register, memory or function unit (ALU).



	Clock cycle 1	Clock cycle 2	Clock cycle 3	Clock cycle 4
I_1	Memory	Decoder	ALU	Memory
I_2		Memory	Decoder	ALU
I_3			Memory	Decoder
I_4				Memory

Fig. 4.5 Resource Conflict

- Conflict is an unsuccessful operation. To handle this condition, keep the instruction I_4 as waiting until the resource becomes available.
- The instruction stand by is referred to as stalls which is elaborated below:



	Clock cycle 1	Clock cycle 2	Clock cycle 3	Clock cycle 4	Clock cycle 5	Clock cycle 6	Clock cycle 7
I_1	Memory	ID	ALU	Memory	WB		
I_2		Memory	ID	ALU	Memory	WB	
I_3			Memory	ID	ALU	Memory	WB
I_4				—	—	—	Memory

Stalls (leads to structural hazards)

Fig. 4.6 Stalls created by Structural Dependency'

$$\begin{aligned} &7 \text{ clock cycles} - 4 \text{ inputs} \\ &\quad = 3 \text{ stalls} \end{aligned}$$

$$\left\{\begin{array}{l} \text{CPI} \neq 1 \\ \text{Program output} = \text{safe} \end{array}\right\}$$

- To minimize the structural hazards, hardware technique is used known as re-naming.
- In Fig. 4.5, instruction I_1 refers the memory in fourth stage of the pipeline to access the data.

- Simultaneously, instruction I_4 refers the memory in the first stage of the pipeline to access the instruction in the same clock cycle CC4.
- When the instruction and data, both are present in the same memory, then the above situation creates conflict.
- 'Renaming' mechanism isolates the memory into two independent sub-memory blocks: code memory (stores instructions) and data memory (stores data).
- Refer the code memory (CM) in first stage and refer the DM (data memory) in fourth stage of the pipeline, so that accessing of these two memories in the same cycle does not create the conflict as described below:

CC = Clock cycles

I_i = Instruction number i

	CC1	CC2	CC3	CC4	CC5	CC6	CC7
I_1	CM	ID	ALU	DM	WB		
I_2		CM	ID	ALU	DM	WB	
I_3			CM	ID	ALU	DM	WB
I_4				CM	ID	ALU	DM
I_5					CM	ID	ALU
I_6						CM	ID
I_7							CM

Fig. 4.7 Resolution of Structural Hazards through Renaming

$$\left\{\begin{array}{l}\text{CM = Code memory}\\ \text{DM = Data memory}\\ \text{ID = Instruction decode}\\ \text{WB = Write back}\end{array}\right\}$$

7 cycles = 7 inputs = 0 stalls

$$\left\{\begin{array}{l}\text{CPI} = 1\\ \text{Program output = safe}\end{array}\right\}$$

2) Data dependency:

- Consider the program segment where instruction J follows instruction I in a program order.

I	:	Instruction
J	:	Instruction
⋮		⋮
⋮		⋮
⋮		⋮

- Data dependency condition will be occurred in the pipeline when the instruction J tries to read the data before instruction I write it.

Read – Before – write

Example:

$$\begin{tikzcd}[column sep=small] I_1 : \text{Add } r_0, r_1, r_2; & r_0 & \leftarrow r_1 + r_2 \\ I_2 : \text{Sub } r_3, r_0, r_2; \; r_3 \leftarrow & r_0 \arrow[u] & - r_2 \end{tikzcd}$$

Note:

A non-pipelined manner of execution does not suffer from any data dependency. Instructions are executed successively, i.e If there exists a register A which is modified by instruction Y and read by instruction X, then instruction X cannot start before instruction Y completes execution.

- If the above code is running on a pipelined processor, then data dependency condition will occur because I_2 is executed along with I_1. So I_2 tries to read the register r_0 data before I_1 writes it. Therefore, I_2 incorrectly accesses the old value from the register r_0 (data loss) as described below:

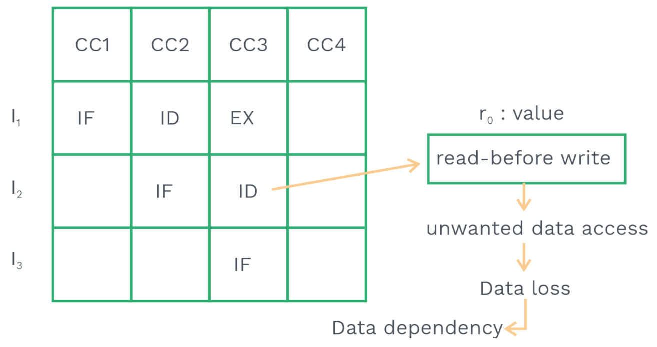


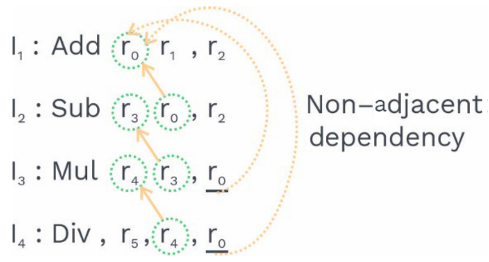
Fig. 4.8 Data Hazards

- To minimize the data hazards, hardware mechanism is used, i.e. operand forwarding, also called as bypassing or short-circuiting.
- This technique states that, use the buffer between the stages to hold the intermediate operation result, so that dependent instruction will be accessing the new value from the buffer before updating the register file.
- In this technique, last stage operation is modified to perform:

$\boxed{\text{WRITE – READ}}$ operation

Means the updated data is immediately available to read, therefore buffer is not required at the end of last stage.

- Consider the following program segment, executed on a RISC pipeline using operand forwarding technique.



- Adjacent data dependency is known as 'true data dependency'.

  That is, $I_2 \rightarrow I_1\ (r_0)$

  $I_3 \rightarrow I_2\ (r_3)$

  $I_4 \rightarrow I_3\ (r_4)$
- Non-adjacent data dependency is known as, data dependency only. Because it is eliminated via true data dependency.

  That is, $I_3 \rightarrow I_1\ (r_0)$

  $I_4 \rightarrow I_1\ (r_0)$

3) Control dependency:

- Control dependency will occur in the pipeline when transfer of control (TOC) instructions are executed in the pipeline.

Code:

1000 : I_1 $\downarrow$ Falling/ falling through path
1001 : I_2
1002 : I_3 (JMP 2000)
1003 : I_4
1004 : I_5
⋮ ⋮
2000 : BI_1 $\downarrow$ Taken path
2001 : BI_2
⋮ ⋮

expected output sequence : $\boxed{I_1 - I_2 - I_3 - BI_1 - BI_2}$

Instruction fetch micro-program (μ) program:

T_1 : PC	$\rightarrow$	MAR
T_2: M[MAR]	$\rightarrow$	MBR
PC + 1	$\rightarrow$	PC
T_3: MBR	$\rightarrow$	IR

PC = 1000	CC1	CC2	CC3	CC4	CC5	CC6
I_1	IF PC : 1001	ID	EX	MA	WB	
I_2		IF PC : 1002	ID	EX	MA	WB
I_3			IF PC : 1003	ID uncond : TOC PC : ~~1004~~ 2000	EX	MA
I_4				IF PC : ~~1004~~	ID	EX
BI_1					IF PC : 2001	ID
BI_2						IF PC : 2002

Actual Output seq. is –

$\boxed{I_1 - I_2 - I_3 - \boxed{I_4} - BI_1 - BI_2}$

Unwanted instruction

uncond : TOC $\longrightarrow$ Unconditional transfer of control

Fig. 4.9 Control Hazards

- In the above execution sequence, unwanted instruction is executed in the pipeline, so it brings the unwanted behaviour in the program output.

This kind of disturbance in the pipeline is called as control dependency.

- To handle the above problem, "Flush" operation is used, also called as freeze operation.

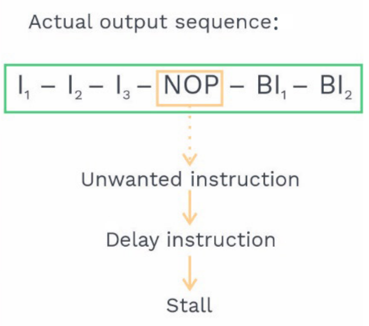
This operation states that, insert the NOP instruction after the JMP instruction to suspend the unwanted instruction fetch.

Code with flush operation:

1000 : I_1
1001 : I_2
1002 : I_3 (JMP 2000)
1003 : I_4 = (NOP)
1004 : I_5
⋮ ⋮
2000 : BI_1
2001 : BI_2
⋮ ⋮

PC = 1000

	CC1	CC2	CC3	CC4	CC5	CC6
I_1	IF PC : 1001	ID	EX	MA	WB	
I_2		IF PC : 1002	ID	EX	MA	WB
I_3			IF PC : 1003	ID uncond : TOC PC : ~~1004~~ 2000	EX	MA
NOP				IF PC : ~~1004~~	ID	EX
BI_1					IF PC : 2001	ID
BI_2						IF PC : 2002



uncond : TOC $\longrightarrow$ Unconditional transfer of Control

Fig. 4.10 Delayed Branch Technique

- Number of stalls created in the pipeline due to a branch instruction is called as branch penalty.

It depends on the availability of the TA (target address) in the pipeline, i.e.

$$\boxed{\text{Branch penalty} = \text{At what stage, Target address is available in the pipeline} - 1}$$

- In the RISC pipeline, branch penalty is always 1 because TA is available in the second stage.

Note:

In the hypothetical pipelines, TA availability is given as:

a) Stage number given, then

$$\boxed{\text{Penalty} = \text{Stage number} - 1}$$

b) Stage name given, then

$$\boxed{\text{Penalty} = \text{Corresponding stage number} - 1}$$

c) Until the instruction is completed (or) all instructions are proceed through all the stages, then

$$\boxed{\text{Penalty} = \text{Last stage number} - 1}$$

Total number of stalls created from the branches/program:

$$= \boxed{\underbrace{\text{Branch frequency}}_{\text{Number of Branch instructions per program}} * \underbrace{\text{Branch penalty}}_{\text{Number of stalls/ branch}}}$$

BRANCH PREDICTION BUFFER

- To minimize the control hazards, hardware mechanism is used, i.e. 'branch prediction buffer', also called as 'branch target buffer' or 'loop buffer'.
- It is a high-speed buffer, present in the first stage of the pipeline used to hold the predicted target addresses.
  If the TA is present in the first stage, then no stall is present. (Prediction is possible when the program contains loops.)

Note:
If the question states that the pipeline is with branch prediction, then we can assume that TA is present in the first stage. Otherwise (without branch prediction) assume that TA is not present in the first stage.

Note:
To minimize the control hazards, software mechanism is also used, i.e. 'delayed branch'.

Delayed branch:
It is a compiler technique, so compiler re–arranges the code to avoid the stall, if possible. Otherwise substitute the NOP instruction after the JMP instruction to preserve the execution path, if not possible to rearrange.

User code:

```
1000 : I1
1001 : I2
1002 : I3 (JMP 2000)
1003 : I4
  :      :
  :      :
2000 : BI1
2001 : BI2
  :      :
```

Expected output sequence : $I_1 - I_2 - I_3 - BI_1 - BI_2$

Actual output sequence : $I_1 - I_2 - I_3 - \underset{\text{Unwanted}}{\underset{\downarrow}{\textcircled{I_4}}} - BI_1 - BI_2$

Delayed branch:

a) Re–arrangement

I_1
I_3 (JMP BI_1)
I_2
I_4
⋮
BI_1
BI_2
⋮

PC = I_1	CC1	CC2	CC3	CC4	CC5
I_1	IF PC : I_3	ID	EX	MA	WB
I_3		IF PC : I_2	ID unconditional : TOC PC : $\cancel{I_4}$ BI_1	EX	MA
I_2			IF PC : $\cancel{I_4}$ BI_1	ID	EX
BI_1				IF PC : BI_2	ID
BI_2					IF PC : BI_3

Actual output sequence is:

$\boxed{I_1 - I_3 - I_2 - BI_1 - BI_2}$

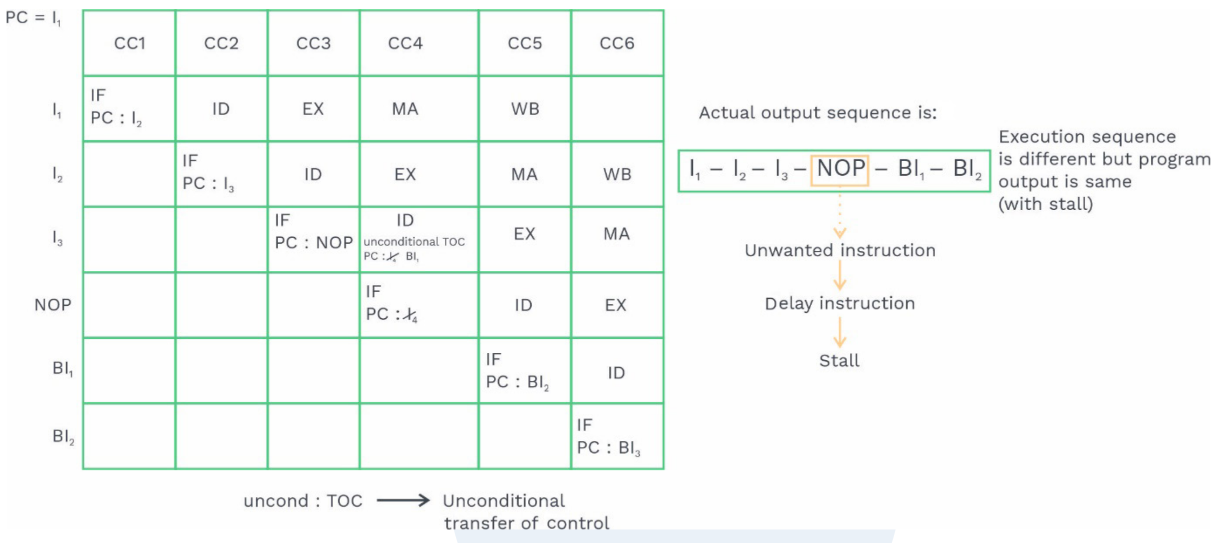
Execution sequence is different but program output is same (without stall)

uncond : TOC $\longrightarrow$ Unconditional transfer of control

Fig. 4.11 Re-arrangement

b) NOP substitution:

I_1
I_2
I_3 (JMP BI_1)
NOP
I_4
⋮
BI_1
BI_2



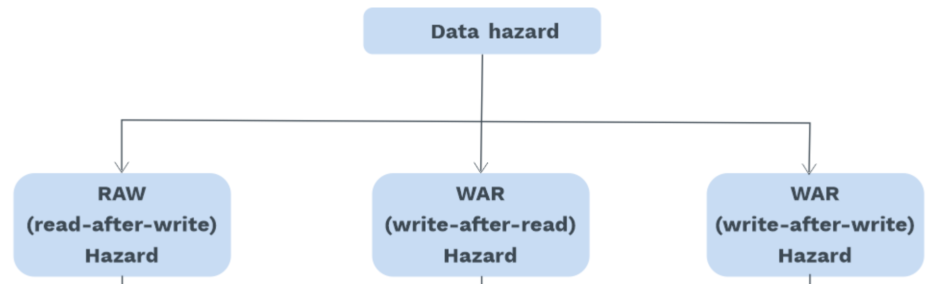
PC = I_1	CC1	CC2	CC3	CC4	CC5	CC6
I_1	IF PC : I_2	ID	EX	MA	WB	
I_2		IF PC : I_3	ID	EX	MA	WB
I_3			IF PC : NOP	ID unconditional TOC PC : $\cancel{I_4}$ BI_1	EX	MA
NOP				IF PC : $\cancel{I_4}$	ID	EX
BI_1					IF PC : BI_2	ID
BI_2						IF PC : BI_3

Actual output sequence is:
$I_1 - I_2 - I_3 - \boxed{\text{NOP}} - BI_1 - BI_2$
Execution sequence is different but program output is same (with stall)
Unwanted instruction
Delay instruction
Stall
uncond : TOC ⟶ Unconditional transfer of control

Fig. 4.12 NOP Substitution

Hazards:

- Hazard is a delay.
- Delay present in the pipeline is due to a dependency conditions.
- Three kinds of a hazards are possible in the pipeline:
  1) Structural hazard
  2) Data hazard
  3) Control hazard
- Data hazard is further classified into three types, based on the order of the read and write operations.



Data hazard
RAW (read-after-write) Hazard
WAR (write-after-read) Hazard
WAR (write-after-write) Hazard

- **RAW** Hazard is created when the instruction J tries to read the data before instruction I writes it. (True data dependency)
- **WAR** Hazard is created when the instruction J tries to write the data before instruction I reads it. (Anti data dependency)
- **WAW** Hazard is created when the instruction J tries to write the data before instruction I writes it. (Output data dependency)

PRACTICE QUESTIONS

Q8 **Consider these two instructions:**
ADD R1, R2, R3
SUB R2, R4, R5
Which of the following is true?
a) It is an anti-dependency.
b) It is an output dependency
c) It is true dependency
d) Instructions are free from all dependencies

Sol: **a)**

ADD R1, R2, R3 is equivalent to R1 $\leftarrow$ R2 + R3
SUB R2, R4, R5 is equivalent to R2 $\leftarrow$ R4 – R5
Clearly, it is write-after-read (WAR) dependency which is also known as anti–dependency.

Q9 **Consider the two instructions given below:**
DIV R1, R2, R3
ADD R1, R4, R3
Which of the following dependencies can be noticed in these instructions?
a) Data dependency
b) Anti dependency
c) Output dependency
d) Instructions are free from any kind of dependency

Sol: **c)**

DIV R1, R2, R3 is equivalent to $R1 \leftarrow \frac{R2}{R3}$
ADD R1, R4, R3 can be converted to R1 $\leftarrow$ R4 + R3
In both the instructions, result is written in R1, hence it is write-after-write (WAW) hazard which is also known as output dependency.

Q10 **Which of the following technique cannot be used to handle the control hazards?**

a) Delayed branch **b) Operand forwarding**
c) Branch prediction **d) Multiple pipelines**

Sol: **b)**

a) Delayed branch is the software solution provided by the compiler for control dependencies.
b) Operand forwarding is the solution for data dependencies.
c) Branch prediction is the hardware solution for control dependencies.
d) Multiple pipelines are the solutions for control hazard. In one pipeline, the execution takes place if the next instruction is sequential instruction after branch, i.e. branch is not taken. In another pipeline, execution of target in struction (not the sequential instruction after branch instruction) takes place.

Q11 **Consider the following code having a sequence of instructions:**

Instruction	**Meaning**
I_1: MUL R1, R2, R3	**R1 $\leftarrow$ R2 × R3**
I_2: ADD R5, R4, R1	**R5 $\leftarrow$ R4 + R1**
I_3: SUB R1, R5, R2	**R1 $\leftarrow$ R5 – R2**
I_4: DIV R3, R4, R5	**R3 $\leftarrow \frac{R4}{R5}$**

Calculate the total number of dependencies including RAW, WAR and WAW. Assume instructions have the same execution sequence as given above.

a) 4 **b) 5** **c) 6** **d) 7**

Sol: **c)**

Consider the following RAW dependencies:

I_1 : MUL R1, R2, R3
R1 $\leftarrow$ R2 × R3

I_2 : ADD R5, R4, R1
R5 $\leftarrow$ R4 + R1

I_3 : SUB R1, R5, R2

$R1 \leftarrow R5 - R2$

I_4 : DIV R3, R4, R5

$R3 \leftarrow \frac{R4}{R5}$

$$\left.\begin{array}{l} I_1 \rightarrow I_2 \\ I_2 \rightarrow I_3 \\ I_2 \rightarrow I_4 \end{array}\right\} \text{RAW}\left(\text{read-after-write}\right)\text{dependencies}$$

$$\left.\begin{array}{l} I_1 \rightarrow I_4 \\ I_2 \rightarrow I_3 \end{array}\right\} \text{WAR}\left(\text{write-after-write}\right)\text{dependencies}$$

$$\left.I_1 \rightarrow I_3\right\} \text{WAW}\left(\text{write-after-write}\right)\text{dependency}$$

RAW = 3 dependencies
WAR = 2 dependencies
WAW = 1 dependency
Total = 3 + 2 + 1 = 6 dependencies

Q12 **Consider the following two instructions:**

MUL R3, R1, R2
DIV R5, R4, R3

Which of the following dependency can be observed in these instructions?

a) True dependency
b) Anti dependency
c) Output dependency
d) Instructions are free from any kind of dependencies

Sol: **a)**

1) MUL R3, R1, R2 is equivalent to $R3 \leftarrow R1 \times R2$

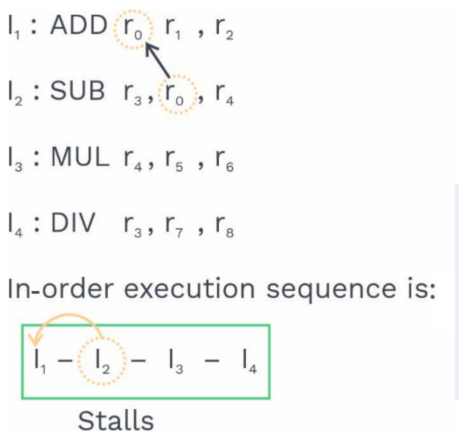
2) DIV R5, R4, R3 is equivalent to $R5 \leftarrow \frac{R4}{R3}$

Here result is written on R3 in multiplication instruction and then contents in R3 is read in division instruction. Clearly it is a case of RAW hazard which is also known as true dependency.

Instruction scheduling (when 'operand forwarding is not supported')

- CPU always executes the program in a sequence called as in-order execution.
- In this execution sequence, if instruction is dependent, then the remaining instructions are also sharing the stall cycles, even if they are independent.

Code:



I_1 : ADD r_0 r_1 , r_2

I_2 : SUB r_3, r_0, r_4

I_3 : MUL r_4, r_5 , r_6

I_4 : DIV r_3, r_7 , r_8

In-order execution sequence is:

$I_1 - I_2 - I_3 - I_4$

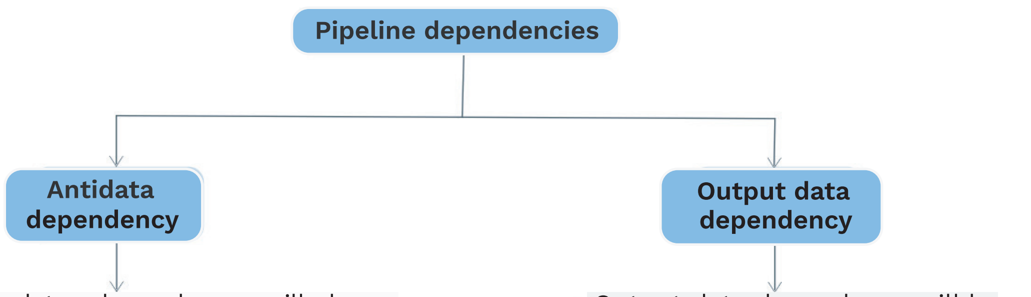
Stalls

- In the above code, I_2 is data dependent on I_1, so I_2 will be waiting until the I_1 execution is completed. This waiting creates stalls in the pipeline. These stalls are also shared by the I_3 and I_4 instructions, even they are independent.
- To handle this problem, 'operand forwarding' technique is used.
- If the CPU doesn't support this technique, then instruction scheduling concept is used to optimize the stalls.
- Instruction scheduling executes the independent instructions first, called as out of execution order (re–order).

  That is,

$$\boxed{I_1 - I_3 - I_4 - I_2}$$

- Out of order execution creates two more dependencies in the pipeline:



- Anti data dependency will be occurred in the pipeline when the instruction tries to write the data before instruction I reads it.
- Output data dependency will be occurred in the pipeline when the instruction J tries to write the data before instruction I writes it.

Note:

To handle the above dependency conditions, hardware mechanism is used. That is 'register renaming'.

Performance evaluation with stalls:

$$S(\text{speed up}) = \frac{\text{Average instruction ET}_{\text{non pipeline}}}{\text{Average instruction ET}_{\text{pipeline}}}$$

$$S(\text{speed up}) = \frac{\text{CPI}_{\text{nonpipe}} * \text{Cycle time}_{\text{nonpipe}}}{\text{CPI}_{\text{pipe}} * \text{Cycle time}_{\text{pipe}}}$$

- Ideal CPI of pipeline is always 1.
- But due to the dependency problem, extra cycles are created in the pipeline. So,

$$\boxed{S(\text{speed up}) = \frac{\text{CPI}_{\text{nonpipe}} * \text{Cycle time}_{\text{nonpipe}}}{(1 + \text{number of stalls per instruction}) * \text{Cycle time}_{\text{pipe}}}}$$

PRACTICE QUESTIONS

Q13 **Consider a 5-stage pipeline having stages as instruction: Fetch (IF), Instruction Decode (ID), Operand Fetch (OF), Execute (Ex) and Write Back (WB). Here we are given 4 instructions. IF, ID, OF and WB stages takes 1 clock cycle each but the EX stage takes 1 cycle for ADD and SUB, 2 cycles for MUL and 3 cycles for DIV operation. If the operand forwarding technique is used from EX stage to OF stage then calculate the total execution time of below program. Assume the clock rate of pipeline processor as 5 GHz.**

Instruction number	Instruction	Meaning
I_1	DIV A, B, C	$A \leftarrow \frac{B}{C}$
I_2	MUL F, E, D	$F \leftarrow E \times D$
I_3	SUB Y, A, F	$Y \leftarrow A - F$
I_4	ADD H, Y, G	$H \leftarrow Y + G$

Sol: 2.2

	1	2	3	4	5	6	7	8	9	10	11
I_1	IF	ID	OF	EX	EX	EX	WB				
I_2		IF	ID	OF	–	–	EX	EX	WB		
I_3			IF	ID	–	–	OF	–	EX	WB	
I_4				IF	ID	–	–	OF	–	EX	WB

Here operand forwarding is used from EX to OF stage.

I_1	DIV A, B, C
I_2	MUL F, E, D
I_3	SUB Y, A, F
I_4	ADD H, Y, G

Instruction I_3 is dependent on both I_1 and I_2, so the operands for I_3 are fetched after the execution of instruction I_1 and I_2.

When execution of I_2 gets completed, we can execute the instruction I_3.
Similarly, I_4 is dependent on I_3. We can only execute I_4 after the execution of I_3.
Total execution time = Total number of clock cycles × clock cycle time

$$\text{Clock cycle time} = \frac{1}{\text{clock rate}}$$
$$= \frac{1}{5 \times 10^9}\text{ s}$$
$$= 0.2 \times 10^{-9}\text{ s}$$
$$= 0.2\text{ ns}$$

Total execution time = 11 × 0.2 ns
= 2.2 ns

Q14 **Given below is a set of instructions:**

I_0: A ← B + C
I_1: B ← A – C
I_2: C ← B × A
I_3: A ← C/B
I_4: B ← A

Where A, B and C are registers.
How many true data dependency, anti-data dependency and output data dependency exist?

a) 3, 8, 3 **b) 3, 6, 2** **c) 4, 6, 2** **d) 4, 8, 2**

Sol: **d)**

True data dependency → RAW hazards

$$\left.\begin{array}{l} I_1 - I_0 \text{ over A} \\ I_2 - I_1 \text{ over B} \\ I_3 - I_2 \text{ over C} \\ I_4 - I_3 \text{ over A} \end{array}\right\} 4$$

Anti data dependency → WAR hazards

$$\left.\begin{array}{l} I_1 - I_0 \text{ over B} \\ I_4 - I_0 \text{ over B} \\ I_2 - I_0 \text{ over C} \\ I_2 - I_1 \text{ over C} \\ I_3 - I_1 \text{ over A} \\ I_4 - I_2 \text{ over B} \\ I_4 - I_3 \text{ over B} \\ I_3 - I_2 \text{ over A} \end{array}\right\} 8$$

Output data dependency → WAW

$$\left.\begin{array}{l} I_0 - I_3 \text{ over A} \\ I_1 - I_4 \text{ over B} \end{array}\right\} 2$$

Q15 **Consider a pipeline architecture having 5 stages. Except the third stage, all instructions spend one cycle in the other stages. The third stage takes 3 clock cycles for instruction LOAD. What is the count of clock cycles required for the execution of the following machine code using optimization?**

I_0: LOAD R_0, 3(R_1); $R_0 \leftarrow [3 + [R_1]]$
I_1: ADD R_2, R_0, R_1; $R_2 \leftarrow R_0 + R_1$
I_2: LOAD R_3, 4(R_4); $R_3 \leftarrow [4 + [R_4]$
I_3: SUB R_5, R_3, R_4 ; $R_5 \leftarrow R_3 - R_4$

Sol: **14**

operand forwarding through interstage buffer register

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I_0	IF	ID	EX	MA	MA	MA	WB							
I_1		IF	ID	ID	ID	ID	EX	MA	WB					
I_2			IF	IF	IF	IF	ID	EX	MA	MA	MA	WB		
I_3							IF	ID	ID	ID	ID	EX	MA	WB

operand forwarding through interstage buffer register

I_1 depends on I_0 for value in R_0. Since I_0 is LOAD instruction that requires memory access, R_0 data will be available after MA stage in the buffer. (Assuming no stage delays.)

[** Similar explanation for I_3 and I_4 **]

∴ Number of clock cycles required = 14

Q16 **A 6-stage pipelined processor is uniformly balanced with a stage propagation delay of 10 ns. A program segment comprising of 40% unconditional branch instructions is executed on this architecture. The target address for branch operation is not available until the last stage. Estimate the average execution time of the processor.**

a) 30 ns **b) 20 ns** **c) 15 ns** **d) 35 ns**

Sol: **a)**

Given that the instruction after branch is not fetched till the branch instruction is completed. So CPI for branch instruction = 6, because we come to known about the target address after the last stage (6th stage) of pipeline.

For normal (non–branch) instructions, CPI = 1.

Given that we have only unconditional branch, so definitely we will have a branch during these unconditional branch instructions.

Let the total instructions = n

Branch instructions = 40% of n

$$\text{Average CPI} = \frac{\begin{pmatrix}\text{Non-Branch Instructions}\\ \text{Percentage} \times \text{CPI of}\\ \text{Non-Branch Instructions}\end{pmatrix} + \begin{pmatrix}\text{Branch Instructions}\\ \text{Percentage} \times \text{CPI of}\\ \text{Branch Instructions}\end{pmatrix}}{\text{Total Instructions}}$$

$$= \frac{(60\%\text{ of n}) \times 1 + (40\%\text{ of n}) \times 6}{n}$$

$$= \frac{0.6n \times 1 + 0.4n \times 6}{n}$$

$$= \frac{0.6n + 2.4n}{n} = 3$$

Average-instruction execution time = Average CPI × $T_{pipeline}$

= 3 × 10 ns (Clock cycle time $T_{pipeline}$ = 10 ns)

= 30 ns

Q17 **Consider a non-pipelined processor running at 4 GHz. it takes 10 cycles to finish an instruction. Designers have converted it into 10-stage pipeline processor but the hardware overhead makes the frequency of new design as 2 GHz. Due to instruction cache miss, 20% of the instructions cause a stall of 15 cycles during the instruction fetch (IF) stage. There are no other hazards in the system. Calculate the speed up of pipelined processor over non–pipelined processor.**

a) 1.75 **b) 1.25** **c) 2.25** **d) 2.75**

Sol: **b)**

Execution time of non–pipelined processor = Number of clock cycles × clock cycle time.

$$\text{Clock cycle time} = \frac{1}{\text{Frequency}}$$

$$\text{Execution time for non–pipelined processor} = 10 \times \frac{1}{4 \times 10^9} \text{ s}$$

$= 2.5 \times 10^{-9}$ s

= 2.5 ns

- Effective CPI for pipelined processor = Ideal CPI + (Percentage of Instruction cache miss) × (Stalls due to instructions cache miss)
  = 1 + (20%) × 15
  = 1 + 0.2 × 15
  = 1 + 3 = 4

Execution time of pipelined processor = Effective CPI × Clock cycle time

$$\text{clock cycle time} = \frac{1}{\text{Frequency}}$$

$$= \frac{1}{2 \times 10^9} \text{ s}$$

Execution time of pipeline processor

$$= 4 \times \frac{1}{2 \times 10^9} \text{ s}$$

$$= 2 \times 10^{-9} \text{ s}$$

= 2 ns

$$\text{speedup} = \frac{\text{Non – pipelined execution time}}{\text{Pipelined execution time}}$$

$$= \frac{2.5\text{ns}}{2\text{ns}} = \boxed{1.25}$$

Q18 **Consider a pipeline of 5 stages. In this pipeline we have 50% of the unconditional branch instructions whose target address is known after third stage only, rest of the instructions are normal instructions. There is no penalty for normal instructions. The clock cycle time for this pipelined processor is 100 ns. Calculate the throughput in MIPS (million instructions per second).**

a) 4 MIPS **b) 8 MIPS** **c) 5 MIPS** **d) 7 MIPS**

Sol: **c)**

Let total instructions = x

Unconditional branched instructions = 50% of x

= 0.5 x

CPI for unconditional branched instructions = 3 (target address is known after third stage only).

CPI for normal instructions = 1

$$\text{Average CPI} = \frac{0.5x \times 3 + 0.5x \times 1}{x} = \frac{1.5x + 0.5x}{x} = \frac{2x}{x} = 2$$

Note:

$$\text{Average CPI} = \frac{\begin{pmatrix}\text{Percentage of}\\ \text{unconditional}\\ \text{branch instructions}\\ \times\text{CPI of unconditional}\\ \text{branch instructions}\end{pmatrix} + \begin{pmatrix}\text{Percentage of}\\ \text{normal instructions}\\ \times\text{CPI of normal}\\ \text{instructions}\end{pmatrix}}{\text{Total instructions}}$$

- Execution time for 1 instruction = Average CPI × Clock cycle time

  = 2 × 100 ns = 200 ns

  1 instruction takes = 200 ns

  1 instruction = 200 × 10^{-9} s

  $1\text{ s} = \frac{1}{200 \times 10^{-9}}$ instructions

  $1\text{ s} = \frac{10^9}{200}$ instructions

  $1\text{ s} = \frac{10^3}{200} \times 10^6$ instructions

  = 5 × 10^6 instructions

  = 5 MIPS

Q19 **Consider a 5-stage pipeline without branch prediction buffer. It is operated with a 2ns clock and allows all instructions except JMP instruction. The target address is not available until fourth stage. A program segment is to be executed on this pipeline. The program segment contains 40% JMP instructions. Among them 30% are conditional branch instructions, since 40% of the conditional JMP instructions fails to satisfy the condition, no stalls are present for them. What is the performance gain achieved by the system?**

a) 2.43 **b) 2.56** **c) 3.43** **d) 3.08**

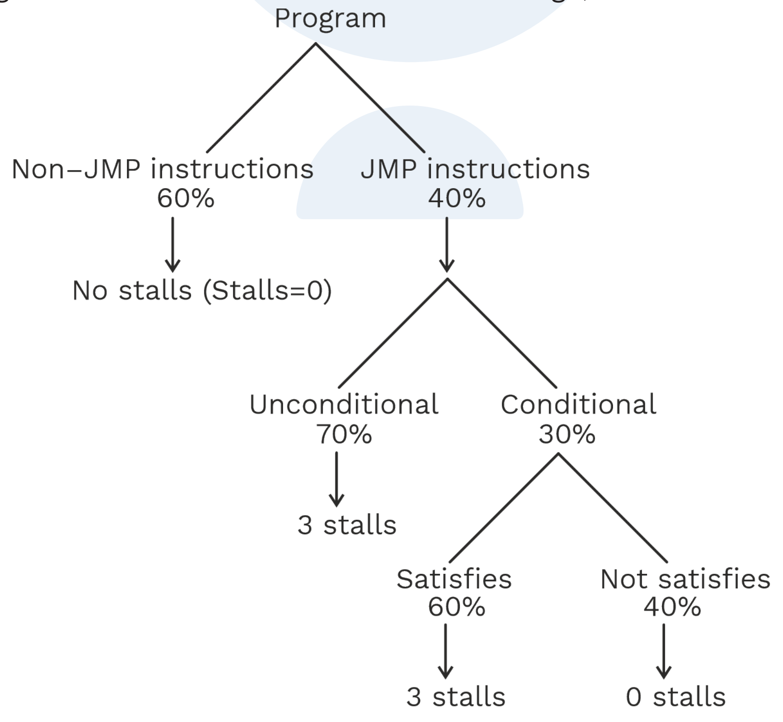
Sol: **a)**

Pipeline is without branch prediction buffer

$\Downarrow$

Stalls are present

Since, target address is available at the fourth stage, number of stalls = 4 – 1 = 3



Number of stalls/instruction = (0.6 × 0) + (0.4 × 0.7 × 3)

+(0.4 × 0.3 × 0.6 × 3)

+(0.4 × 0.3 × 0.4 × 0)

= 1.056

$$\text{Performance gain} = \frac{K}{1 + \text{Number of stalls / instructions}} = \frac{5}{1 + 1.056}$$

$= 2.43$

Q20 **Consider a 8–stage pipeline operated with 3ns clock pulse is used to execute a program segment. There are 20 instructions in that program numbered from I_1 to I_{20}. I_5 is an unconditional JMP instruction that transfers the control to I_{17}. Assuming that the target address is available at the last stage of the pipeline. What is the execution time of the program (in nanoseconds)?**

a) 72 **b) 69** **c) 75** **d) 66**

Sol: **b)**

Since target address is available at the last stage of the pipeline, number of stalls = 8 – 1 = 7.

Successful execution:

$I_1 - I_2 - I_3 - I_4 - I_5$ – NOP – NOP – NOP – NOP – NOP – NOP – NOP – $I_{17} - I_{18} - I_{19} - I_{20}$

n = 16

$\therefore$ Execution $\text{time}_{\text{pipe}} = (K + n - 1) \times tp$

$= (8 + 16 - 1) \times 3\text{ns}$

$= 69$ ns

Chapter Summary

- **Pipeline definition:**
The definition states that, insert the new input into the pipeline before computation of an old input, so new input is executing along with the old input called as overlapping execution.
- **Design of pipeline:**
  a) Uniform delay pipeline:
  $$\boxed{\text{Cycle time}\left(t_p\right) = \text{Stage delay}}$$
  b) Non-uniform delay pipeline:
  $$\boxed{\text{Cycle time}\left(t_p\right) = \text{Maximum}\left(\text{Stage delay}\right)}$$
  c) If buffer delay present in the pipeline:
  $$\boxed{\text{Cycle time}\left(t_p\right) = \text{Maximum}\left(\text{Stage delay} + \text{Buffer delay}\right)}$$
- **Performance analysis of pipeline:**
  $$\boxed{S = \frac{n.t_n}{(K+n-1)\,t_p}}\left(\text{When n is finite}\right)$$
  $$\boxed{S = \frac{t_n}{t_p}}\text{, when n approaches to infinity } (\infty)$$
- **Types of pipelines:**
  1) **Linear pipeline:** This type of pipeline contains forward connections only.
  2) **Non-linear pipeline:** This pipeline contains forward and backward connection.
- **Dependencies in the pipeline:**
  1) **Structural dependency:** It occurs in the pipeline due to a resource conflict
  2) **Data dependency:** Data dependency condition will occur in the pipeline when the instruction J tries to read the data before instruction I write it.
  3) **Control dependency:** Control dependency will occur in the pipeline when transfer of control (TOC) instructions are executed in the pipeline.
- **Delayed branch:** It is a compiler technique, so compiler rearranges the code to avoid the stall.
- **Hazards:** Hazard is a delay which is present in the pipeline due to a dependency condition.

Chapter Summary

Three kinds of a hazards are possible in the pipeline:

1) Structural hazard

2) Data hazard

3) Control hazard

- Data hazard is further classified into three types:

  i) RAW (read-after-write) hazard

  ii) WAR (write-after-write) hazard

  iii) WAW (write-after-write) hazard
- **Operand forwarding:** It is a hardware mechanism used to minimize the data hazards.
- **Instruction scheduling:** It is used when operand forwarding is not supported.
- **Performance evaluation with stalls:**

$$\boxed{S(\text{speed up}) = \frac{CPI_{\text{nonpipe}} * \text{cycle time}_{\text{nonpipe}}}{(1+\text{number of stalls per instruction}) * \text{cycle time}_{\text{pipe}}}}$$