Pipelining

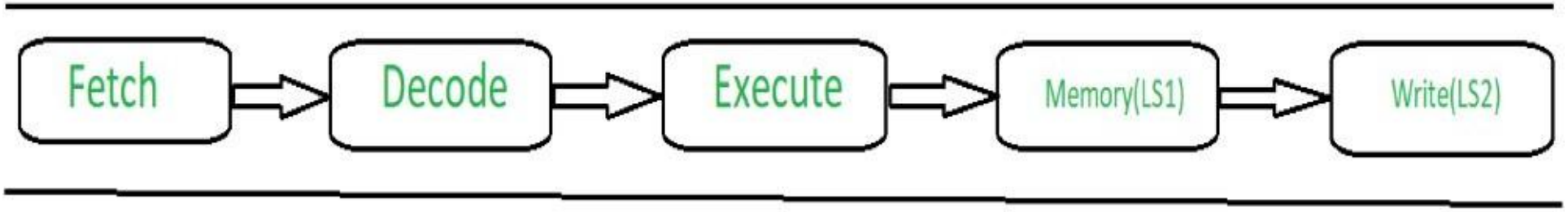
Speedup – An Example

- Let's assume that
  - tp = 20ns
  - the pipeline has k = 4 segments
  - executes n = 100 tasks in sequence
- The pipeline will
  - (k + n - 1) tp = (4 + 100 -1) × 20 ns
  - = 2060 ns to complete the task
- Non pipeline
  - tn = 4 × 20 = 80 ns, it will require
  - ktp = 100 × 80 = 8000 ns
- Therefore the speedup is 8000/2060 = 3.88
- which will approach 4 as n grows

Note:

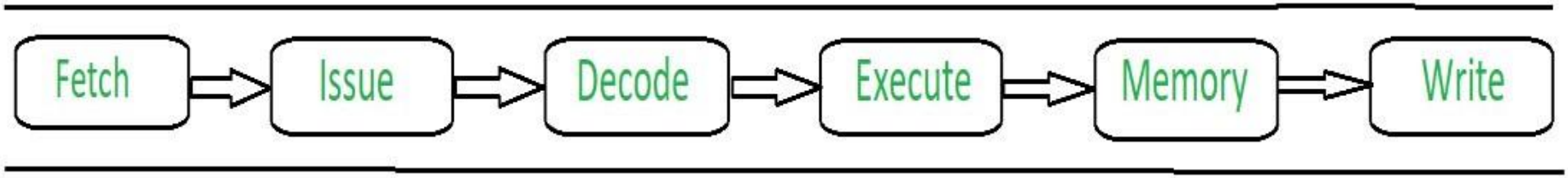
- The stages of pipelining may increase or decrease on the basis of the instruction sets processed per cycle (In maximum situations, stages tend to increase to increase efficiency).

- 5 stages.
- It takes 5 cycles to complete the process.



Six stage pipeline.

- It takes 6 cycles to complete the process.



Advantages of Pipelining

- The cycle time of the processor is reduced.
- It increases the throughput of the system
- It makes the system reliable.

Disadvantages of Pipelining

- The design of pipelined processor is complex and costly to manufacture.
- The instruction latency is more.

Types of Pipeline

- Arithmetic Pipeline
- Instruction Pipeline

Arithmetic Pipeline

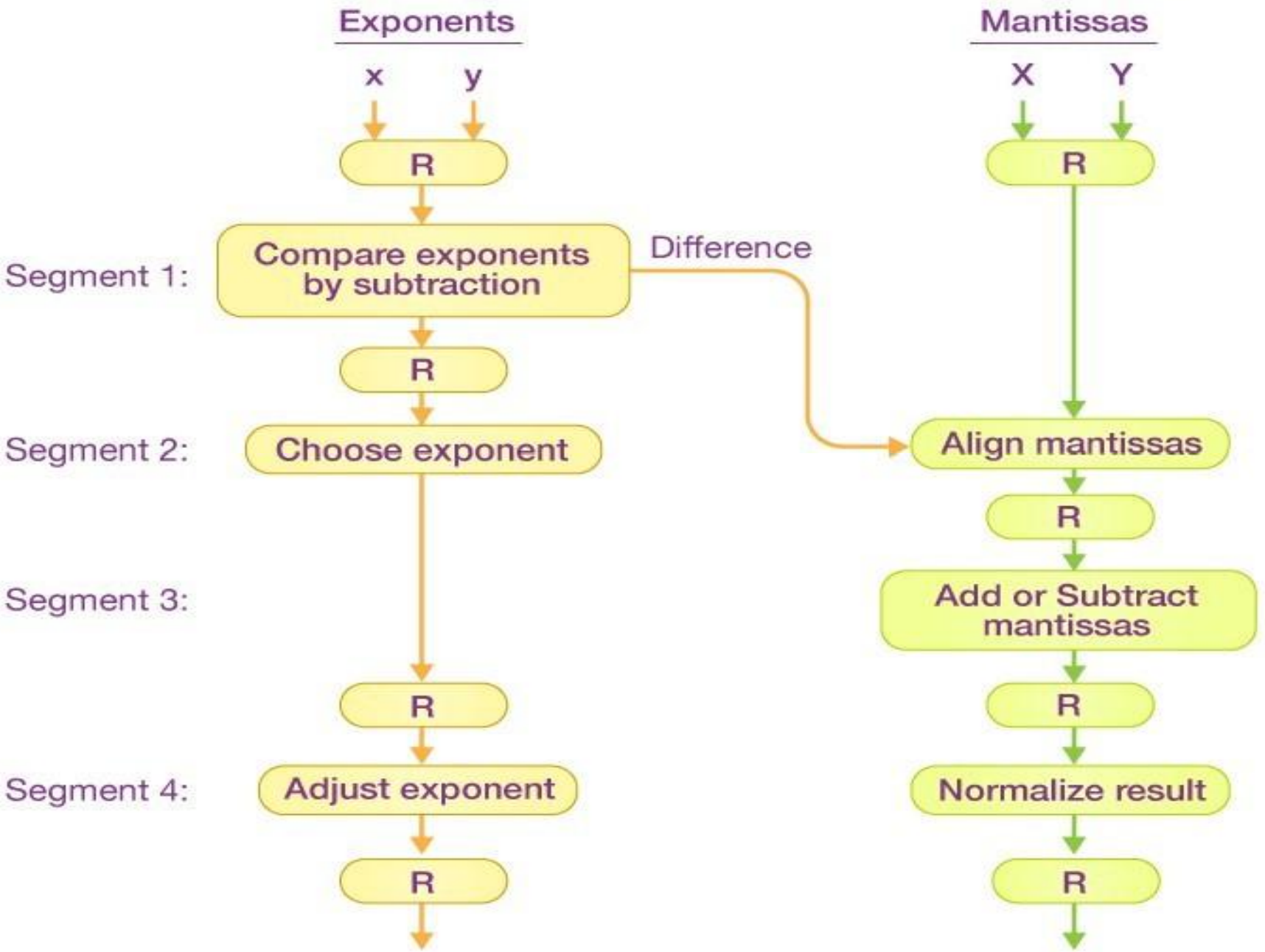
- Arithmetic pipelines are usually found in most of the high speed computers.
- An arithmetic pipeline separates a given arithmetic problem into sub problems that can be executed in different pipeline segments.
- They are used for floating point operations, multiplication of fixed point numbers etc.

Example

- The input to the Floating Point Adder pipeline is:
- $X = A*2^a$
- $Y = B*2^b$
- Here **A** and **B** are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

- Compare the exponents.
- Align the mantissas.
- Add or subtract mantissas
- Produce the result.
- Registers are used for storing the intermediate results between the above operations.


Exponents
Mantissas
x
y
X
Y
R
R
Segment 1:
Compare exponents by subtraction
Difference
R
Segment 2:
Choose exponent
Align mantissas
R
Segment 3:
Add or Subtract mantissas
R
R
Segment 4:
Adjust exponent
Normalize result
R
R

Instruction Pipeline

- In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle.
- This type of technique is used to increase the throughput of the computer system.
- An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline.
- Thus we can execute multiple instructions simultaneously.

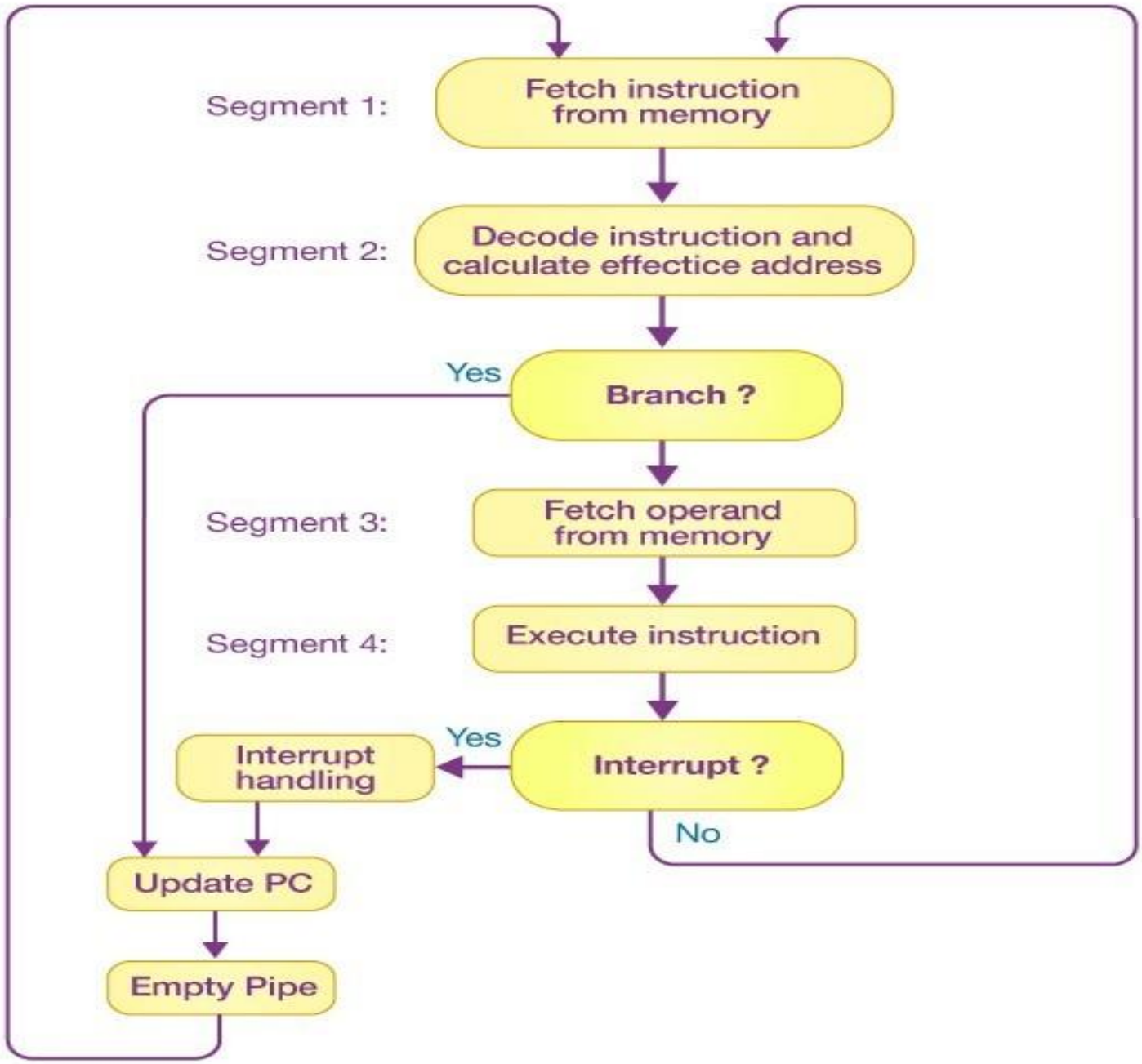
- The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.
- An instruction pipeline receives sequential instructions from memory while prior instructions are implemented in other portions.
- Pipeline processing can be seen in both the data and instruction streams.

- In general, each and every instruction must be processed by the computer in the following order:

1. Fetching the instruction from memory.(FI)
2. Decoding the obtained instruction.(DA)
3. Calculating the effective address.(DA)
4. Fetching the operands from the given memory.(FO)
5. Execution of the instruction.(EX)
6. Storing the result in a proper place.(EX)

- Each step is carried out in its own segment, and various segments may take different amounts of time to process the incoming data.

- Here operations are performed using 4 segment.


Segment 1:
Fetch instruction from memory
Segment 2:
Decode instruction and calculate effectice address
Yes
Branch ?
Segment 3:
Fetch operand from memory
Segment 4:
Execute instruction
Yes
Interrupt handling
Interrupt ?
No
Update PC
Empty Pipe

Performance Measures of Instruction Pipelining

Latency

- The **latency of a pipeline** is defined as the time required for an instruction to propagate through the pipeline.
- Its unit is time unit like microseconds, nanoseconds.
- Latency is higher if there are lots of exceptions/hazards in the pipeline.

Latency = Number of pipeline Stages (m) x Cycle Time (T)

Throughput

- **Throughput of a pipeline** is the rate at which instructions can start and finish i.e. the number of instructions finished per second.
- Throughput is inversely proportional to Latency; More the latency less is the throughput.

$$\text{Throughput of the Pipeline} = \frac{\text{Number of Instructions Executed}}{\text{Total Execution Time}}$$

Speed Up

- Pipelined design is faster than non-pipelined design.
- The speedup is a quantitative measure of the performance achievement, generally compare with non pipelined scenario.

$$\text{Speed Up achieved due to Pipeline} = \frac{\text{Non Pipelined Execution Time}}{\text{Pipelined Execution Time}}$$

Pipeline Conflicts

- There are some factors that cause the pipeline to deviate its normal performance.
- Some of these factors are given below:

1. **Timing Variations**
2. **Data Hazards**
3. **Branching**
4. **Interrupts**
5. **Data Dependency**

- Timing Variations

All stages cannot take same amount of time.

This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

- Data Hazards

When several instructions are in partial execution, and if they reference same data then the problem arises.

We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

- Branching

In order to fetch and execute the next instruction, we must know what that instruction is.

If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

- Interrupts

An interrupt is a request for the processor to interrupt currently executing code

Interrupts effect the execution of instruction.

- Data Dependency

It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Pipelining Hazards

- The CPU's speed is limited by memory.
- There's one more case to consider, i.e. a few instructions are at some stage of execution in a pipelined design.
- There is a chance that these sets of instructions will become dependent on one another, reducing the pipeline's pace.
- Dependencies arise for a variety of reasons, which we will examine shortly.
- The dependencies in the pipeline are referred to as hazards since they put the execution at risk.

There are three kinds of hazards:

- Structural Hazards
- Data Hazards
- Control Hazards

Structural Hazards

- Structural hazards arise due to hardware resource conflict amongst the instructions in the pipeline.
- A resource here could be the Memory, a Register in ALU.
- This resource conflict is said to occur when more than one instruction in the pipe is requiring access to the same resource in the same clock cycle.
- This is a situation that the hardware cannot handle all possible combinations in an overlapped pipelined execution.

Instructions / Cycle	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

- The above table contains the four instructions I_1, I_2, I_3, and I_4, and five cycles 1, 2, 3, 4, 5.
- In cycle 4, there is a resource conflict because I_1 and I_4 are trying to access the same resource.
- Here, the resource is memory.
- The solution to this problem is that we have to keep the instruction on wait as long as the required resource becomes available.
- Because of this wait, the stall will be introduced in pipelines .

Instructions / Cycle	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				-	-	-	IF(Mem)	

Solutions for Structural dependency

- With the help of a hardware mechanism, we can minimize the structural dependency stalls in a pipeline.

- The mechanism is known as **renaming.**

- The renaming mechanism states that it splits the memory into two independent sub-modules to store instruction and data separately.
- The module used to store instruction is called code memory (CM), and the module used to store data is called data memory(DM).

Data Hazards

- Data hazards in pipelining emerge when the execution of one instruction is dependent on the results of another instruction that is still being processed in the pipeline.
- Data Hazards occur when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed.

There are four types of data dependencies:

- Read after Write (RAW),
- Write after Read (WAR),
- Write after Write (WAW), and
- Read after Read (RAR).

Read after Write (RAW) :

- It is also known as True dependency or Flow dependency.
- It occurs when the value produced by an instruction is required by a subsequent instruction.

- For example,
- i: ADD R1, R2, R3
- j: SUB R4, R1, R3

- Add modifies R1 and then Sub should read it.
- If this order is changed, there is a RAW hazard

Stalls are required to handle these hazards.

Write after Read (WAR) :

- It is also known as anti dependency.
- These hazards occur when the output register of an instruction is used right after read by a previous instruction.

- i: SUB R4, R1, R3
- j: ADD R1, R2, R3
- Instruction i has to read register r1 first, and then j has to modify it.
- Otherwise, there is a WAR hazard.
- There is a problem because of R1.
- If some other register had been used, there will not be a problem
- Solution is register renaming, that is, use some other register. The hardware can do the renaming or the compiler can do the renaming

Write after Write (WAW) :

- It is also known as output dependency.
- These hazards occur when the output register of an instruction is used for write after written by previous instruction.

- i: SUB R1, R4, R3
- j: ADD R1, R2, R3
- Instruction i has to modify register r1 first, and then j has to modify it.
- Otherwise, there is a WAW hazard.
- There is a problem because of R1.
- If some other register had been used, there will not be a problem
- Solution is register renaming, that is, use some other register. The hardware can do the renaming or the compiler can do the renaming

Read after Read (RAR) :

- It occurs when the instruction both read from the same register.
- For example,

ADD --, R1, --;

SUB --, R1, --;

- Since reading a register value does not change the register value, these Read after Read (RAR) hazards don’t cause a problem for the processor.

- **Handling Data Hazards :**

  These are various methods we use to handle hazards:

- Forwarding,
- Code reordering , and
- Stall insertion.

- **Forwarding :**
  This technique involves rerouting outputs from one pipeline stage to feed another stage without needing to go through the whole pipeline process, hence eliminating dependencies.
- **Code reordering :**
  It involves compiling code in such a way that instruction that are affected by data hazards are spaced out, ensuring that no read/write conflicts would occur.

- **Stall Insertion :**

  It inserts one or more stall (no-operation instructions) into the pipeline, which delays the execution of the current instruction until the required operand is written to the register file, but this method decreases pipeline efficiency and throughput.

Control Hazards (Branch Hazards)

- Branch hazards are caused by branch instructions and are known as control hazards in computer architecture.
- The flow of program/instruction execution is controlled by branch instructions.

- The conditional statements are used in higher-level languages for iterative loops and condition testing.
- These are converted into one of the BRANCH instruction variations.
- As a result, when the decision to execute one instruction is reliant on the result of another instruction, such as a conditional branch, which examines the condition's consequent value, a conditional hazard develops.

- When we transfer the control instructions, the control dependency will occur at that time.
- These instructions can be JMP, CALL, BRANCH, and many more.
- On many instruction architectures, when the processor wants to add the new instruction into the pipeline, the processor does not know the target address of these new instructions.
- Because of this drawback, unwanted instructions are inserted into the pipeline

- For this, we will assume a program and take the following sequence of instructions like this:
- 100: I1
- 101: I2
- 102: I3
- .
- .
- 250: BI1
- Expected Output is described as follows:
- $I_1 \rightarrow I_2 \rightarrow BI_1$

Instructions / Cycle	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID(PC:250)	EX	MEM	WB
I_3			IF	ID	EX	MEM
BI_1				IF	ID	EX

- The output sequence is described as follows:
- $I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow BI_1$

- So the above example shows that the expected output and output sequence are not equal to each other.
- It shows that the pipeline is not correctly implemented.
- We can correct that problem with the help of stopping the instruction fetch as long as we get the target address of branch instruction.

- For this, we will implement the delay slot as long as we get the target address, which is described in the following table:

Instructions / Cycle	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID(PC:250)	EX	MEM	WB
Delay	-	-	-	-	-	-
BI_1				IF	ID	EX

- The output sequence is described as follows:
- $I_1 \rightarrow I_2 \rightarrow$ Delay (Stall) $\rightarrow BI_1$

- In the above example, we can see that there is no operation performed by the delay slot.
- That's why this output sequence and the expected output are not equal to each other.
- But because of this slot, a stall will be introduced in the pipeline.

- The elimination or reduction of control hazards can significantly enhance processor performance by ensuring the pipeline's continuous operation.

Ways to resolve Control Hazards

1. **Branch prediction.**

- In the control dependency, we can eliminate the stall in the pipelines with the help of a method known as **Branch prediction.**
- The prediction about which branch will be taken is done at the 1st stage of branch prediction.
- The branch prediction contains the 0 **branch penalty.**
- **Branch Penalty:** Branch penalty can be described as the number of stalls that are introduced at the time of branch operation in the pipelined.

2. **Branch Delay Slots**

- Instructions that immediately follow a branch are executed in the pipeline stages behind the branch, regardless of whether the branch condition is satisfied or not.

3. **Pipeline stall cycles.**

- Freeze the pipeline until the branch outcome and target are known, then proceed with fetch.