PREMIER UNIVERSITY CHATTOGRAM

DEPARTMENT OF **COMPUTER SCIENCE AND ENGINEERING**

Complex Engineering Problem

COURSE NAME	Computer Organization Architecture	
COURSE CODE	CSE 3737	
ASSIGNMENT TOPIC	**Designing a Pipelined Processor for a Simple Instruction Set Architecture**	
DATE OF ASSIGNMENT	23 February 2025	
SUBMITTED TO		
Ms. Tanni Dhoom ***Assistant Professor*** ***DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING***		
REMARKS	**SUBMITTED BY**	
	NAME	Rimjhim Dey
	ID	0222220005101039
	SEMESTER	5th
	BATCH	42nd
	SESSION	Fall 2024
	SECTION	A

1. Introduction

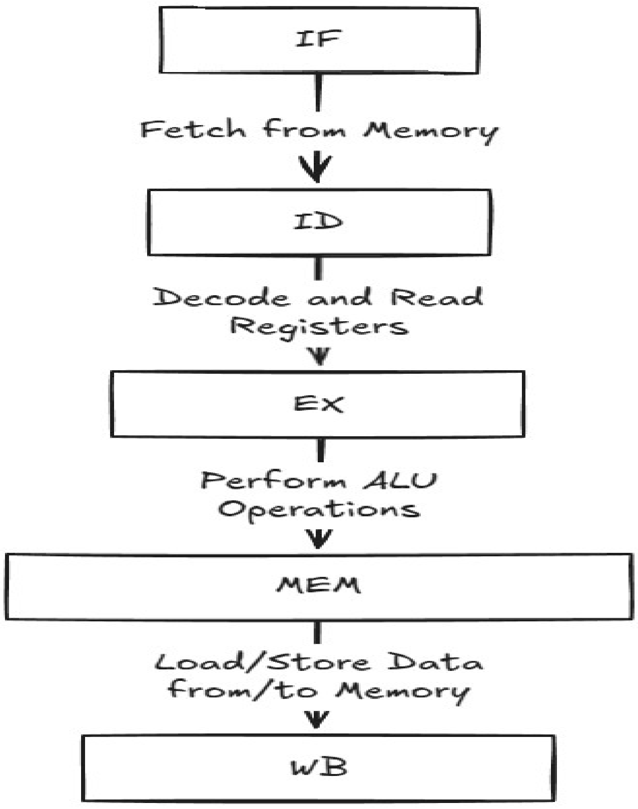
Pipelining is a key technique used in modern processors to improve instruction throughput by overlapping the execution of multiple instructions. This assignment focuses on designing a pipelined processor for a simple Instruction Set Architecture (ISA) that supports basic arithmetic and logical operations. The goal is to maximize instruction throughput while minimizing stalls caused by data hazards.

2. Problem Scenario

The problem involves designing a pipelined processor that handles basic instructions such as **ADD**, **SUB**, **LOAD**, **STORE**, and **MUL**. The processor must efficiently manage pipeline stages and resolve data hazards to ensure correct and high-performance execution.

Initial Diagram: Pipeline Stages-

Below is the initial diagram of the 5-stage pipeline:



3. Objectives

1.Design a pipelined processor architecture that effectively utilizes instruction-level parallelism.
2.Implement hazard detection and resolution mechanisms to handle data hazards.
3.Minimize stalls and improve overall instruction throughput.

4. Investigation

To design the pipelined processor, the following concepts were investigated:

4.1 Pipelining Concepts

•Pipeline Stages: Breaking instruction execution into stages (Fetch, Decode, Execute, Memory, Write-back) to overlap operations.
•Instruction-Level Parallelism: Executing multiple instructions simultaneously in different stages.

4.2 Data Hazards

1.Structural Hazards: Occur when two instructions require the same hardware resource simultaneously.

2.Data Hazards: Occur when an instruction depends on the result of a previous instruction that is still in the pipeline.

- •**RAW (Read After Write)**: The most common type, where an instruction reads a value before it is written.
- •**WAR (Write After Read):** Occurs when an instruction writes to a register before a previous instruction reads it.
- •**WAW (Write After Write):** Occurs when two instructions write to the same register out of order.

3.Control Hazards: Occur due to branch instructions, where the next instruction is unknown until the branch is resolved.

4.3 Hazard Resolution Techniques

•Forwarding (Bypassing): Directly passing the result of an instruction to the next instruction without waiting for it to be written back.
•Stalling: Inserting bubbles (NOPs) into the pipeline to delay instructions until dependencies are resolved.
•Branch Prediction: Predicting the outcome of branch instructions to avoid control hazards.

5. Design

The pipelined processor design includes the following components:

5.1 Pipeline Stages

The processor is divided into five stages:

1.**Fetch (IF):** Fetches the instruction from memory.

2.**Decode (ID):** Decodes the instruction and reads registers.

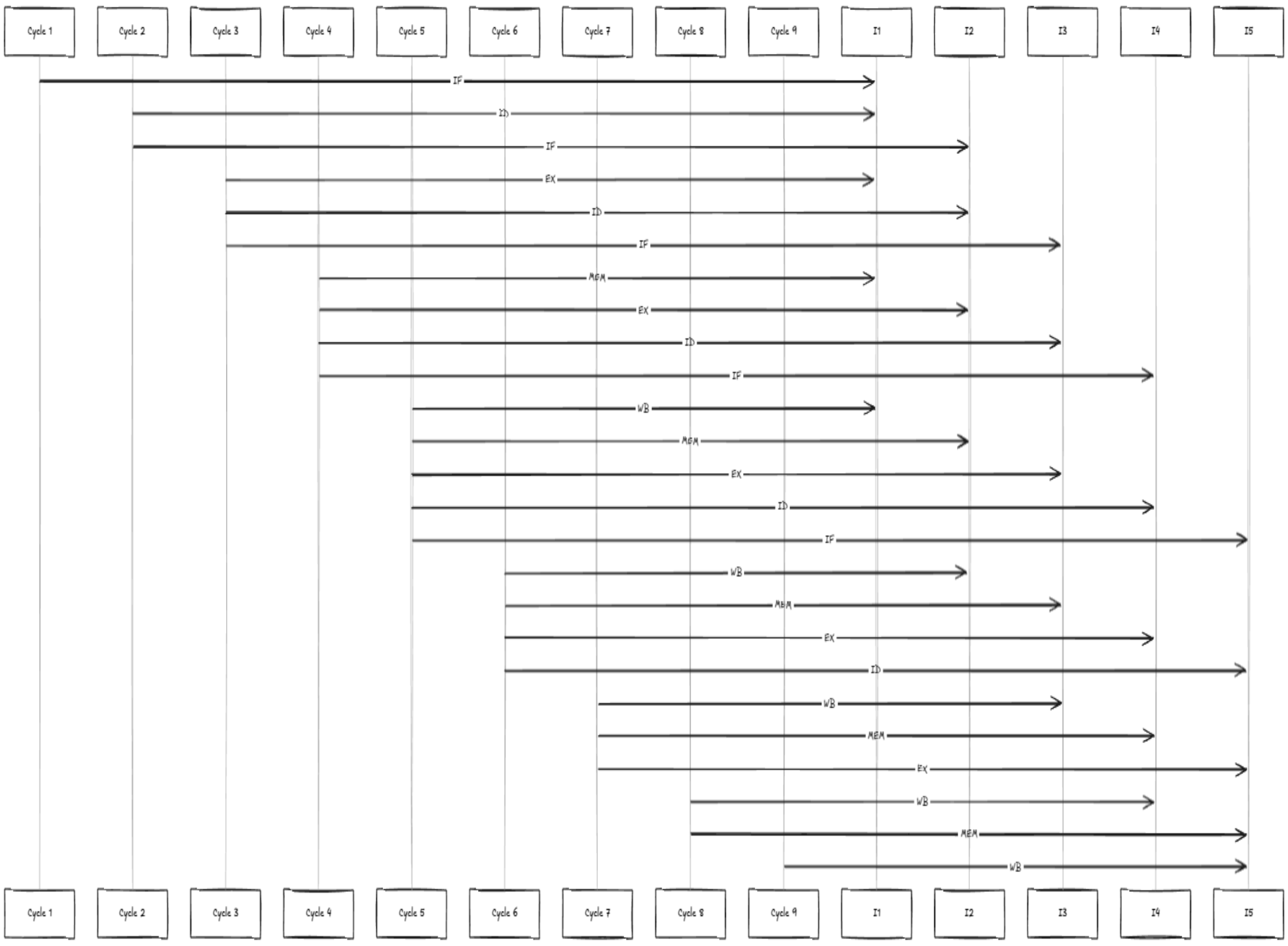
3.**Execute (EX)**: Performs arithmetic or logical operations.

4.**Memory (MEM):** Accesses memory for load/store operations.

5.**Write-back (WB):** Writes results back to the register file.

Pipeline Timing Diagram:

The following diagram shows how instructions move through the pipeline over multiple cycles. Each row represents a clock cycle, and each column represents an instruction (I1, I2, I3, I4, I5).



Detailed Explanation of the Diagram

1. **Cycle 1:**
    - I1 is in the Fetch (IF) stage.

- No other instructions have started yet.

2. **Cycle 2**:
    - I1 moves to the Decode (ID) stage.
    - I2 starts and is in the Fetch (IF) stage.
3. **Cycle 3**:
    - I1 moves to the Execute (EX) stage.
    - I2 moves to the Decode (ID) stage.
    - I3 starts and is in the Fetch (IF) stage.
4. **Cycle 4:**
    - I1 moves to the Memory (MEM) stage.
    - I2 moves to the Execute (EX) stage.
    - I3 moves to the Decode (ID) stage.
    - I4 starts and is in the Fetch (IF) stage.
5. **Cycle 5:**
    - I1 moves to the Write-back (WB) stage.
    - I2 moves to the Memory (MEM) stage.
    - I3 moves to the Execute (EX) stage.
    - I4 moves to the Decode (ID) stage.
    - I5 starts and is in the Fetch (IF) stage.
6. **Cycle 6:**
    - I1 completes and exits the pipeline.
    - I2 moves to the Write-back (WB) stage.
    - I3 moves to the Memory (MEM) stage.
    - I4 moves to the Execute (EX) stage.
    - I5 moves to the Decode (ID) stage.
7. **Cycle 7:**
    - I2 completes and exits the pipeline.
    - I3 moves to the Write-back (WB) stage.
    - I4 moves to the Memory (MEM) stage.
    - I5 moves to the Execute (EX) stage.
8. **Cycle 8**:
    - I3 completes and exits the pipeline.
    - I4 moves to the Write-back (WB) stage.
    - I5 moves to the Memory (MEM) stage.
9. Cycle 9:
    - I4 completes and exits the pipeline.
    - I5 moves to the Write-back (WB) stage.

5.2 Pipeline Hazards

1. Data Hazards

Occurs when instructions depend on previous results that have not yet completed execution.

Example:

```
ADD R1, R2, R3  ; Produces R1
SUB R4, R1, R5  ; Uses R1 before WB stage
```

Solution: Implement forwarding (bypassing) and stall cycles using the hazard detection unit.

2. Control Hazards

Arise from branch instructions affecting program flow.

Example:

```
BEQ R1, R2, LABEL  ; Changes PC if R1 == R2
```

Solution: Use branch prediction and stall cycles.

5.3 Data Hazards Handling

To handle data hazards, the following techniques are implemented:

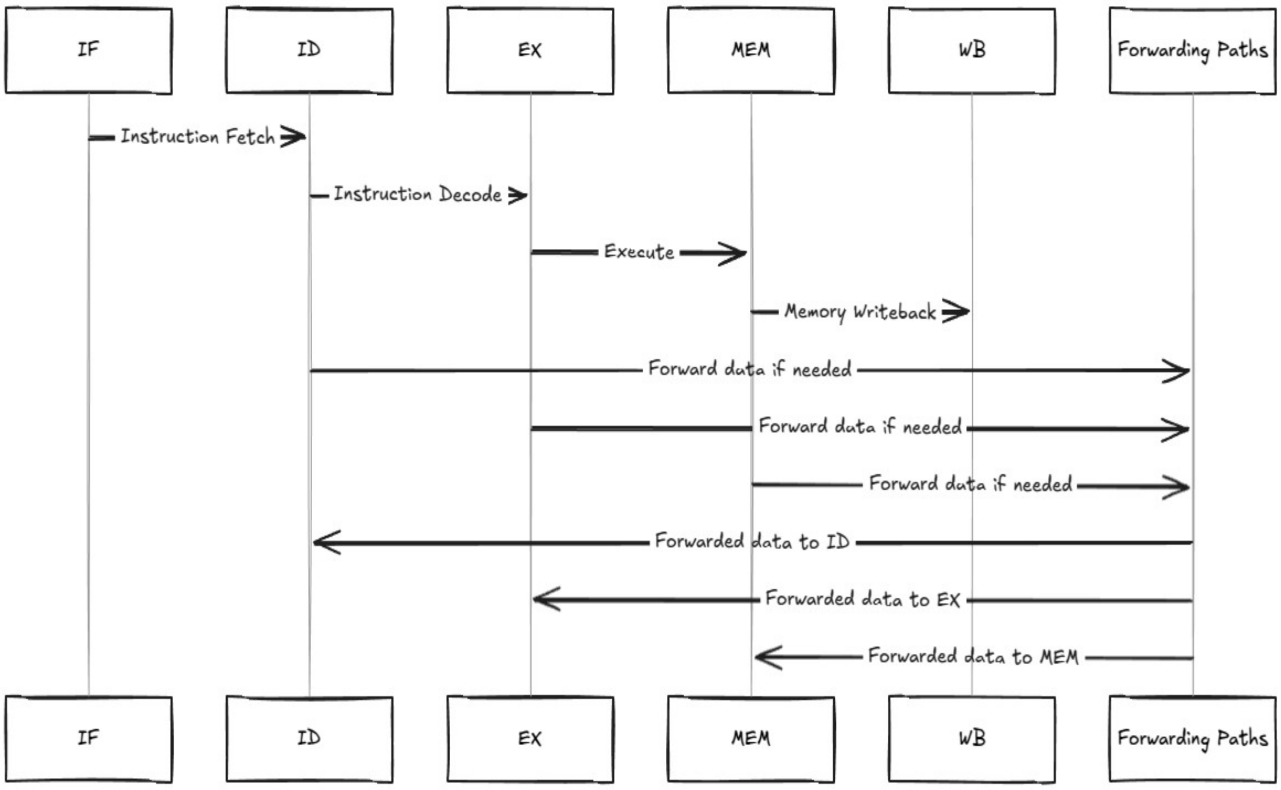
1.Forwarding: Results from the EX/MEM or MEM/WB stages are forwarded to the ID/EX stage to resolve RAW hazards.

2.Stalling: If forwarding cannot resolve a hazard, the pipeline is stalled by inserting NOPs.

5.4 Forwarding Paths in Action

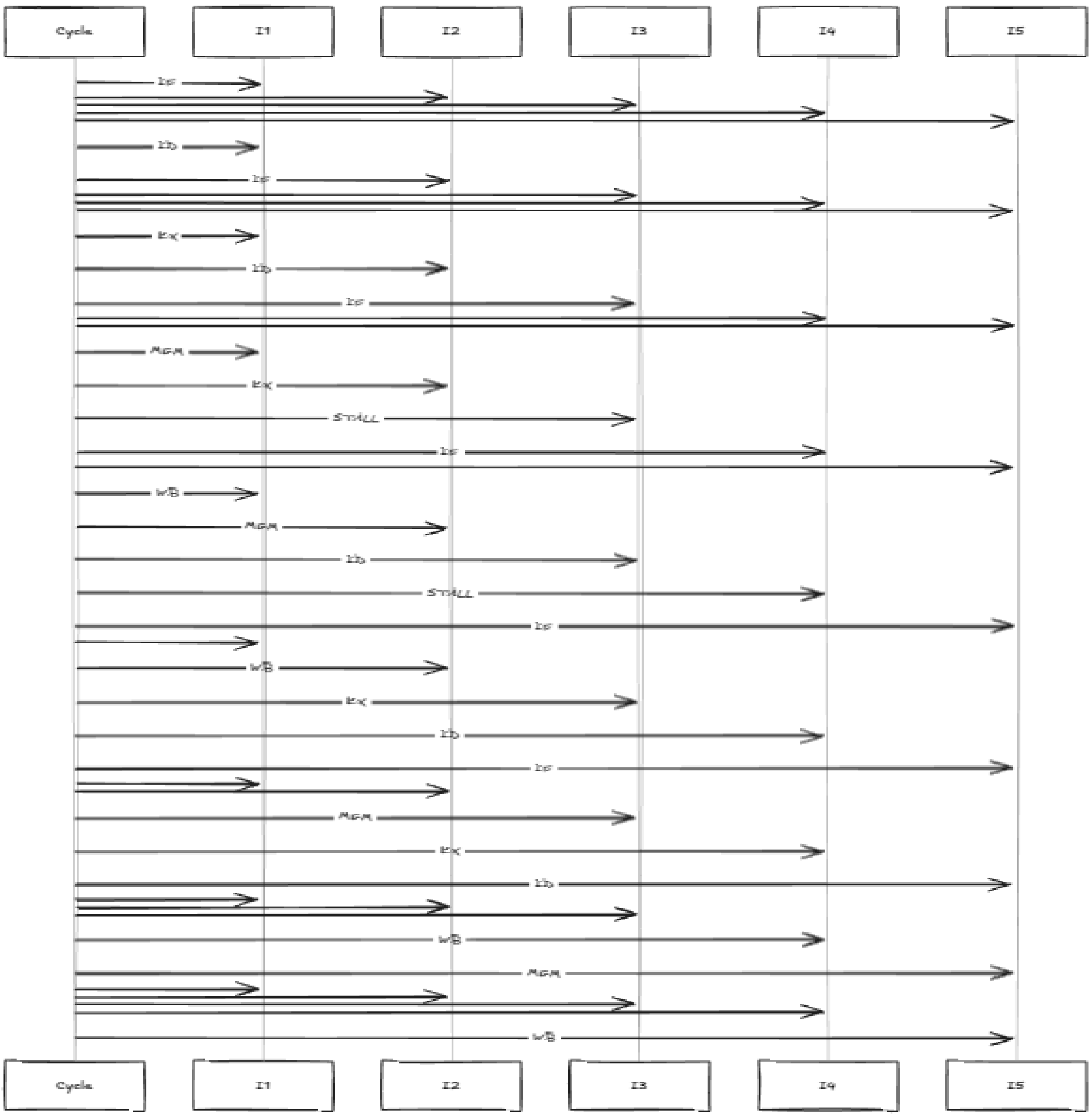
The following diagram shows how forwarding paths resolve data hazards:

•Forwarding Path 1: Data from the EX/MEM stage of I1 is forwarded to the EX stage of I2.
•Forwarding Path 2: Data from the MEM/WB stage of I2 is forwarded to the EX stage of I3.



5.5 Pipeline with Stalls (Example)

If a data hazard occurs between I2 and I3, the pipeline might need to stall for one cycle. Here’s how the diagram would look:



5.6 Hazard Detection Unit (HDU)

The hazard detection unit monitors dependencies and inserts stalls where necessary.

Logic for Data Hazard Detection:

1. If an instruction in EX/MEM stage writes to a register used in ID stage → Insert a stall.
2. If an instruction in MEM/WB stage writes to a register used in ID stage → Insert a stall if forwarding is unavailable.

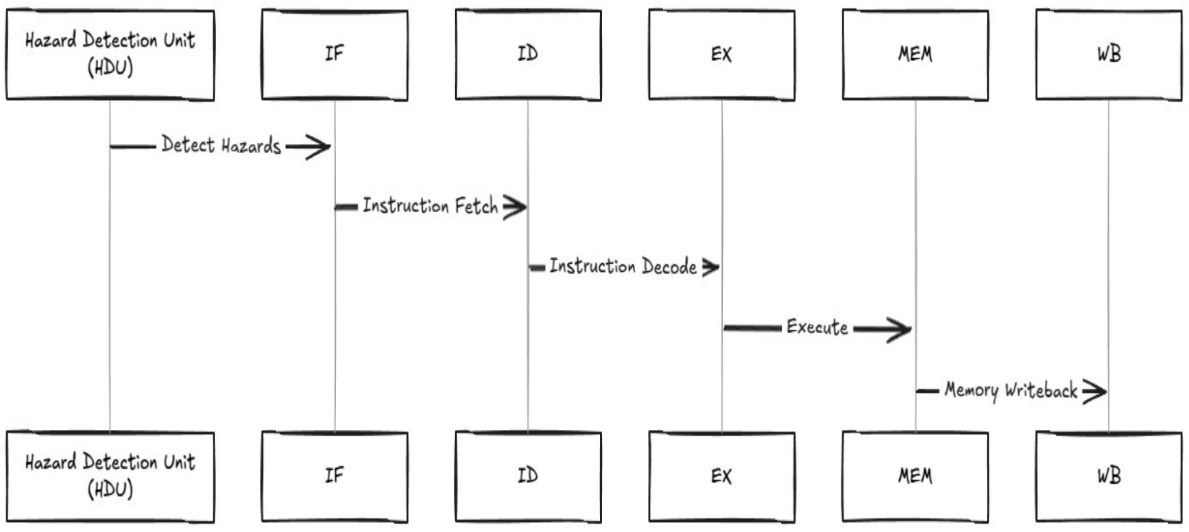
Pseudocode:

```
if (EX/MEM.RegisterRd == ID/EX.RegisterRs OR EX/MEM.RegisterRd ==
ID/EX.RegisterRt) then
    Stall Pipeline
```

Control Hazard Handling:

1. **Branch Prediction:** Assume taken or not taken and flush incorrect paths.
2. **Stall Cycles:** Delay execution until branch resolution.

This diagram shows the integration of the Hazard Detection Unit (HDU) into the pipeline:



5.7 Control Unit

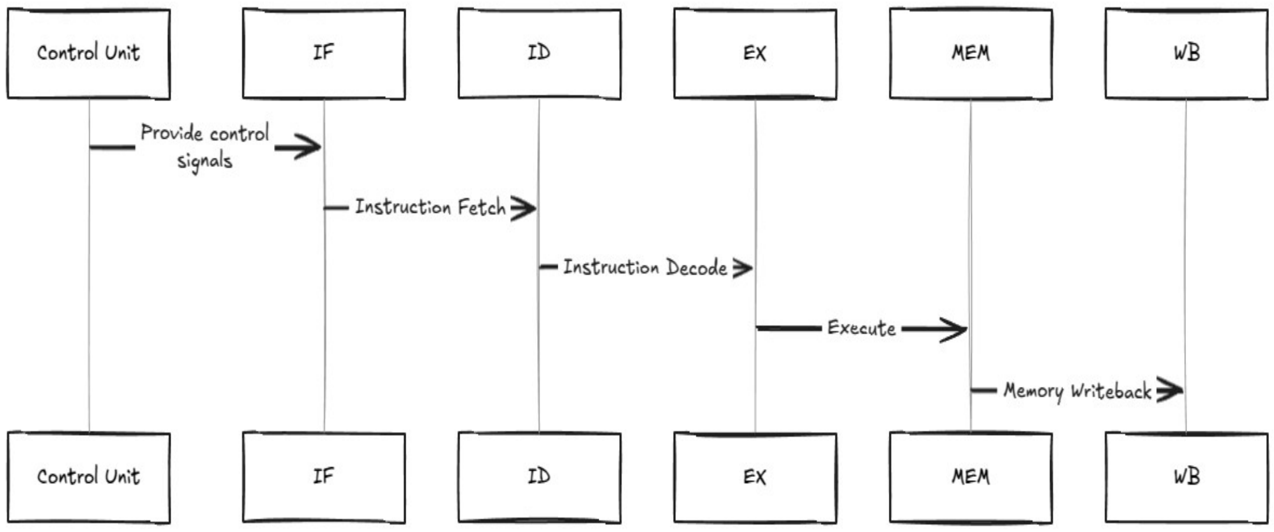
The Control Unit manages pipeline control signals, including:

•**Branch Prediction**: Predicts the outcome of branch instructions to minimize stalls.

•**Pipeline Flushing:** Flushes the pipeline if a branch prediction is incorrect.

•**Stall Signals:** Sends signals to stall the pipeline when necessary.

The diagram shows how the control signals flow through the stages, guiding the instruction through the pipeline:



6.Evaluation

The proposed pipelined processor design is evaluated based on its ability to handle data hazards, improve performance, and maintain correctness. Below is a detailed justification of the design decisions and their impact on the processor's performance and functionality.

6.1 Data Hazard Mitigation

- Implementing forwarding (data forwarding/bypassing) reduces stalls caused by dependencies between instructions.
- Utilizing pipeline interlocks helps detect and resolve hazards dynamically.
- Optimized instruction scheduling minimizes stalls by reordering instructions when possible.

6.2 Pipeline Efficiency and CPI Calculation

- **Ideal CPI Calculation:** In a fully pipelined processor with no hazards or stalls, the ideal CPI is **1**, meaning one instruction is completed per cycle.
- **Realistic CPI Analysis:** Considering data hazards and occasional stalls, the CPI is calculated using:

  Stall Cycles Instructions ExecutedCPI=1+Total Instructions ExecutedTotal Stall Cycles

  - If we assume a pipeline with an average of **30 stalls per 100 instructions**, then: CPI=1+10030=1.3
  - A lower CPI means better performance, and our hazard-handling techniques aim to minimize CPI close to **1**.

6.3 Branch Prediction and Performance Enhancement

- Implementing **static and dynamic branch prediction** minimizes control hazards.

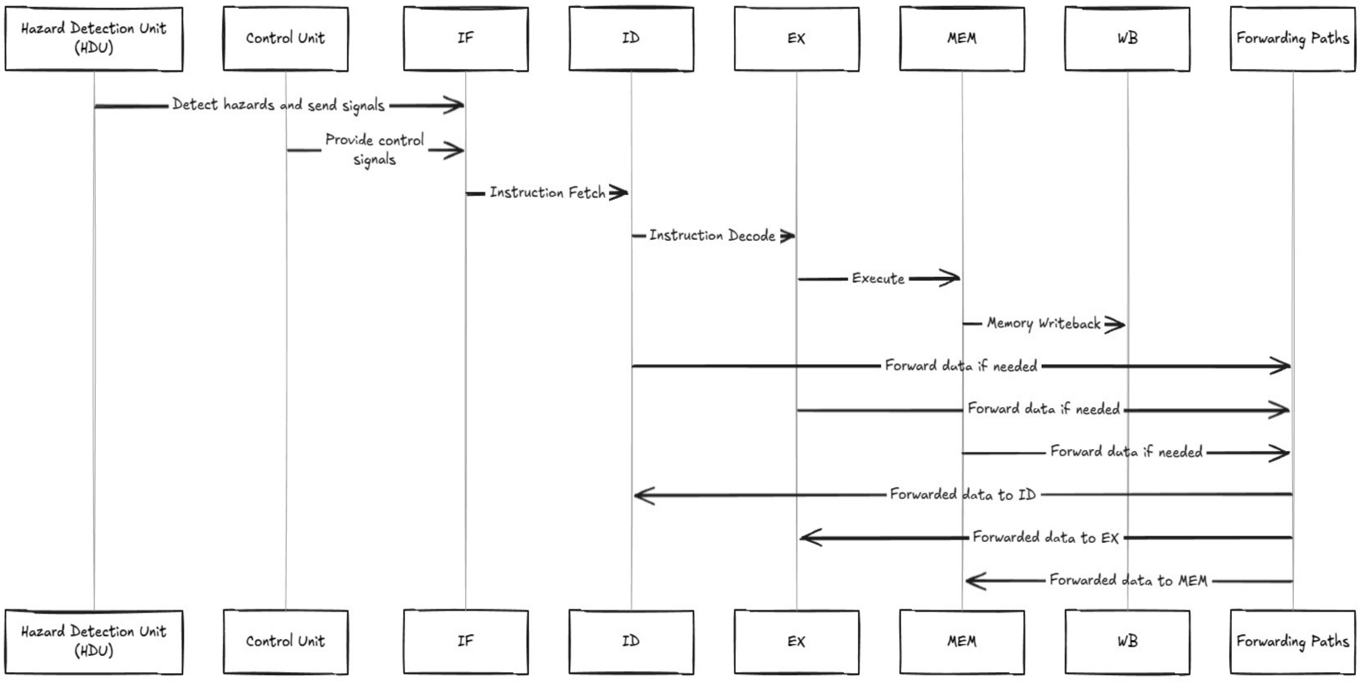
- A **2-bit saturating predictor** reduces mispredictions, thereby decreasing unnecessary stalls.
- Speculative execution further enhances instruction throughput.

6.4 Overall Performance Improvement

- The architecture ensures **higher instruction throughput** while reducing stalls due to hazards.
- Efficient **memory access optimization** and **cache utilization** improve performance by reducing memory latency.
- CPI is optimized close to **1**, ensuring near-maximum utilization of pipeline stages.

Final System Diagram:

The final system integrates all components, including the pipeline stages, forwarding paths, Hazard Detection Unit (HDU), and Control Unit-



7.Conclusion

The proposed pipelined processor design effectively addresses data hazards, improves performance, and maintains correctness. By leveraging pipelining, forwarding, and branch prediction, the design achieves a significant improvement in instruction throughput compared to a non-pipelined processor. The trade-offs between performance and complexity are carefully balanced to ensure an efficient and practical implementation. This design demonstrates a deep understanding of computer architecture principles and their application to real-world engineering problems.