Instruction Pipeline Simulation

Detecting Data Dependencies and Visualizing Stalls using Strategy Design Pattern

Seif Eldin Mohamed Mohamed Essam Mohamed Medhat Saeed Waleed
Saeed Mahmoud

Faculty of Computers and Information
HPC Course

Supervised by: Prof. Dr. Hossam Reda Mohamed

November 2025

Tools: Python • Google Colab

Project Overview

- This project simulates a 5-stage instruction pipeline similar to what is used inside a CPU.
- The goal is to show how data dependencies and forwarding affect instruction execution.
- The simulation is written in Python, and results are visualized to make the concept clear and educational.

Why We Did This

- The pipeline concept is essential in computer architecture courses (like HPC).
- By simulating it, we can see what happens cycle-by-cycle instead of only learning it theoretically.
- The project helps understand:
  - How CPU instructions overlap in execution.
  - What causes stalls and how forwarding fixes them.

What is an Instruction Pipeline?

- A pipeline divides instruction execution into multiple stages:
- **IF** $\rightarrow$ **ID** $\rightarrow$ **EX** $\rightarrow$ **MEM** $\rightarrow$ **WB**
- Each stage handles one part of the instruction while others work on different ones.
- This parallelism increases performance and CPU throughput.

Example

While instruction **I1** is executing (EX), instruction **I2** can be decoding (ID), and instruction **I3** can be fetched (IF).

Stages Explained

Stage	Full Name	Description
IF	Instruction Fetch	Get instruction from memory
ID	Instruction Decode	Decode operation and registers
EX	Execute	Perform arithmetic or logic operation
MEM	Memory Access	Read/write from memory
WB	Write Back	Write result back to register file

The Problem: Data Hazard

Example: RAW (Read After Write) Hazard

```
I1: R1 = R2 + R3
I2: R4 = R1 + R5
```
$\leftarrow$ *depends on R1*

- Here, **I2** needs **R1** before **I1** finishes writing it.
- This causes a RAW (Read After Write) hazard.
- The CPU must either:
  - **Wait (stall)** until R1 is ready, or
  - **Forward** the result early from an intermediate stage.

How We Handle Hazards

We implemented two strategies:

- **No Forwarding:**
  - Waits until the first instruction reaches the Write Back stage.
  - More stalls $\rightarrow$ slower execution.
- **Forwarding:**
  - Passes the result directly from EX or MEM to the next instruction.
  - Fewer stalls $\rightarrow$ faster performance.

Strategy Design Pattern

- We used the Strategy Pattern to make the hazard-handling logic flexible.
- **Structure:**
  - `HazardResolver` (abstract)
    - `NoForwarding`
    - `Forwarding`
- This allows us to easily switch between strategies and compare results in one simulation.

Implementation Overview

- Implemented a `PipelineSimulator` class handling 5 stages.
- Each instruction passes through these stages every cycle.
- The simulator:
    - Detects dependencies.
    - Inserts NOPs (stalls) automatically.
    - Displays results in a per-cycle table.
- Visualization created using Matplotlib to show timing per instruction.

Example Program

Instructions with RAW Hazards

```
I1: R1 = R2 + R3
I2: R4 = R1 + R5
I3: R6 = R4 + R1
I4: R7 = R8 + R9
I5: R1 = R10 + R11
```

- These instructions contain data dependencies that cause RAW hazards.
- Our simulator detects them and adjusts execution timing automatically.

Simulation Results

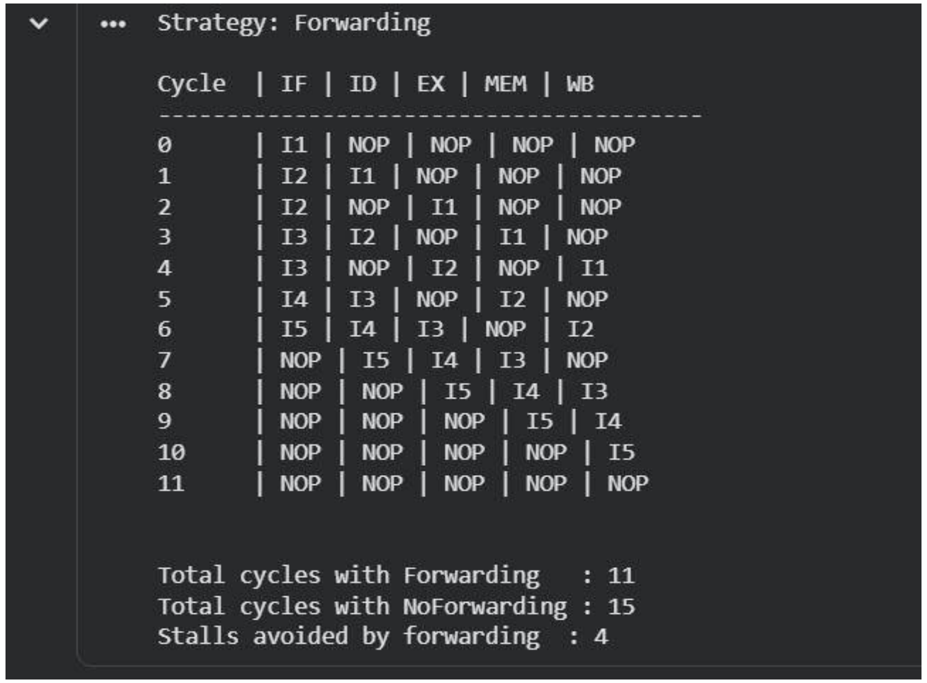
Mode	Total Cycles	Stalls Avoided
Without Forwarding	15	—
With Forwarding	11	4 cycles saved

Performance Benefit

Forwarding reduced total cycles by 4, proving its performance benefit.

Results: Forwarding Strategy (Cycle by Cycle)

- This output from our simulator shows the "Forwarding" strategy in action.
- The pipeline correctly inserts **NOPs** (stalls) when a dependency is detected but cannot be forwarded (e.g., in cycle 2, 4).
- **Total cycles: 11**.

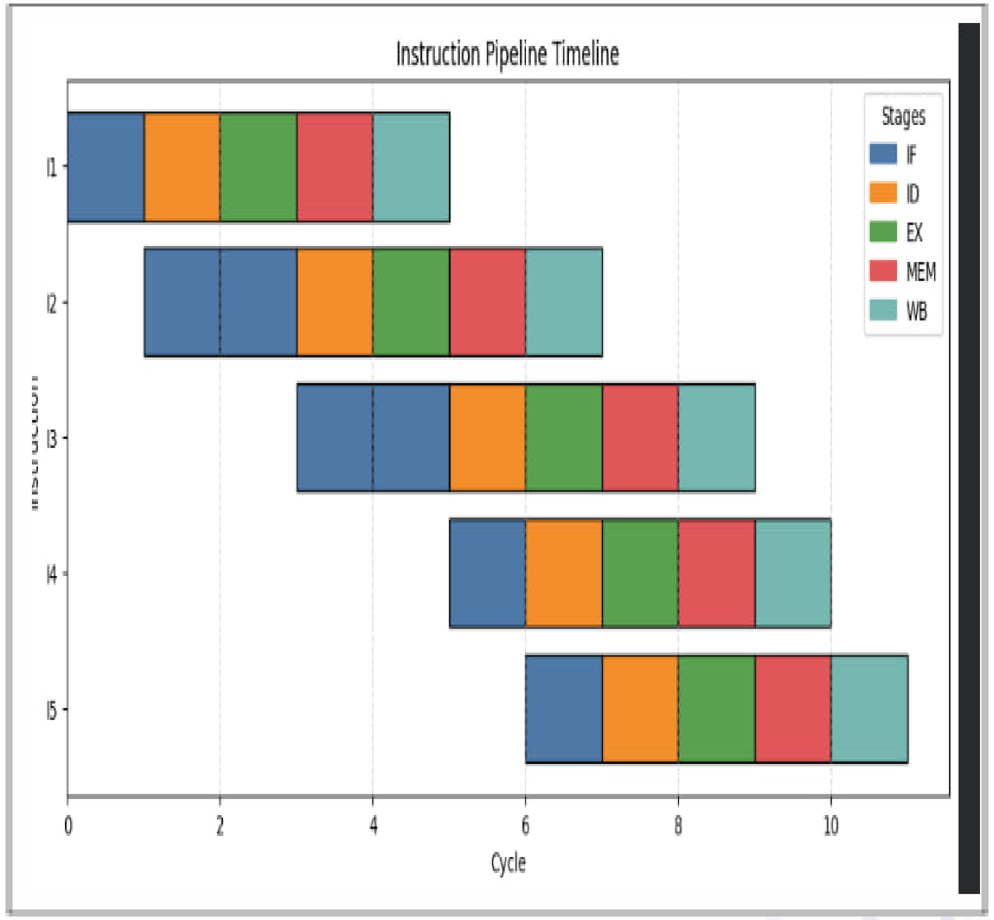


```
Strategy: Forwarding

Cycle  | IF | ID | EX | MEM | WB
----------------------------------------
0      | I1 | NOP | NOP | NOP | NOP
1      | I2 | I1 | NOP | NOP | NOP
2      | I2 | NOP | I1 | NOP | NOP
3      | I3 | I2 | NOP | I1 | NOP
4      | I3 | NOP | I2 | NOP | I1
5      | I4 | I3 | NOP | I2 | NOP
6      | I5 | I4 | I3 | NOP | I2
7      | NOP | I5 | I4 | I3 | NOP
8      | NOP | NOP | I5 | I4 | I3
9      | NOP | NOP | NOP | I5 | I4
10     | NOP | NOP | NOP | NOP | I5
11     | NOP | NOP | NOP | NOP | NOP


Total cycles with Forwarding   : 11
Total cycles with NoForwarding : 15
Stalls avoided by forwarding   : 4
```

Figure: Console output for "Forwarding" mode.

Visualization: Timeline (With Forwarding)

- Timeline chart shows each instruction per stage across cycles.
- Colors represent stages:
  - Blue = IF
  - Orange = ID
  - Green = EX
  - Red = MEM
  - Teal = WB
- Gaps represent stalls.



Key Insights

- Forwarding significantly improves instruction flow.
- The Strategy Pattern made the implementation clean and flexible.
- Simulation helped visualize what really happens inside the CPU pipeline.
- Understanding these concepts is crucial for systems design and optimization.

Future Work

- Add Load/Store and Branch instructions.
- Simulate Control Hazards (like branch misprediction).
- Model latency for different instruction types.
- Add GUI or animation for real-time visualization.

Conclusion

- We built a complete Instruction Pipeline Simulation in Python.
- It demonstrates data hazards, stalls, and forwarding visually.
- It applies object-oriented design and the Strategy pattern effectively.
- The project enhances both understanding and practical skills in computer architecture.

Team:
Seif Eldin Mohamed • Mohamed Essam • Mohamed Medhat • Saeed Waleed Saeed Mahmoud

Thank You

Questions?

Appendix: HazardResolver Strategy Pattern

```python
from abc import ABC, abstractmethod
class HazardResolver(ABC):
    @abstractmethod
    def should_stall(self, pipeline: Dict[str, Instruction], incoming: Instruction) -> int
    :
        pass

class NoForwarding(HazardResolver):
    def should_stall(self, pipeline, incoming):
        if incoming is None or incoming == NOP or not incoming.srcs:
            return 0
        max_stall = 0
        stage_order = ["IF","ID","EX","MEM","WB"]
        for stage_idx, stage in enumerate(stage_order):
            instr = pipeline.get(stage, NOP)
            if instr and instr != NOP and instr.dest:
                for s in incoming.srcs:
                    if s == instr.dest:
                        remaining = (len(stage_order)-1) - stage_idx
                        if remaining > max_stall:
                            max_stall = remaining
        return max_stall
```