

# 5-Stage Pipelined RISC Processor with Hazard Handling

Assignment 3

**Student Name:** Aditya Gaur

**Programme:** Bachelor of Computer Applications (BCA)

**Semester:** V

**Course:** ENCA302 - Intro to Computer Organization  
Architecture

**Faculty:** Dr. Kishore Ayyala

**Session:** 2025-2026

**Department of Computer Science**

School of Engineering  
Technology

November 12, 2025

# Contents

<b>1</b>	<b>Architecture Layout</b>	<b>3</b>
<b>2</b>	<b>Hazard Handling Simulation</b>	<b>4</b>
2.1	Data Hazards (RAW) . . . . .	4
2.1.1	Forwarding Logic . . . . .	4
2.1.2	Proof of Forwarding . . . . .	5
2.2	Control Hazards (Branching) . . . . .	5
<b>3</b>	<b>Pipeline Animation</b>	<b>6</b>
<b>4</b>	<b>Performance Evaluation</b>	<b>8</b>
4.1	Ideal CPI . . . . .	8
4.2	Real CPI . . . . .	8
4.3	Discussion: Stalls and Efficiency . . . . .	9

# 1 Architecture Layout

This project implements a 5-stage pipelined RISC-V processor using Verilog, based on the ‘merlsru/RISCV\_PipelineCore’ repository. The pipeline is divided into the five classic stages, separated by [cite<sub>start</sub>]

- **IF (Instruction Fetch):** Fetches the next instruction from Instruction Memory using the Program Counter (PC) [cite: 344-384]. [cite<sub>start</sub>]
- **ID (Instruction Decode):** Decodes the instruction, reads source registers from the Register File, and generates control signals [cite: 22-491].
- **EX (Execute):** Performs the calculation using the ALU. [cite<sub>start</sub>] This stage also contains the MU 218 – 343]. [cite<sub>start</sub>]
- **MEM (Memory Access):** Handles memory operations, reading from or writing to the Data Memory [cite: 433-458]. [cite<sub>start</sub>]
- **WB (Write Back):** Writes the final result (from the ALU or Data Memory) back into the Register File [cite: 458-469].

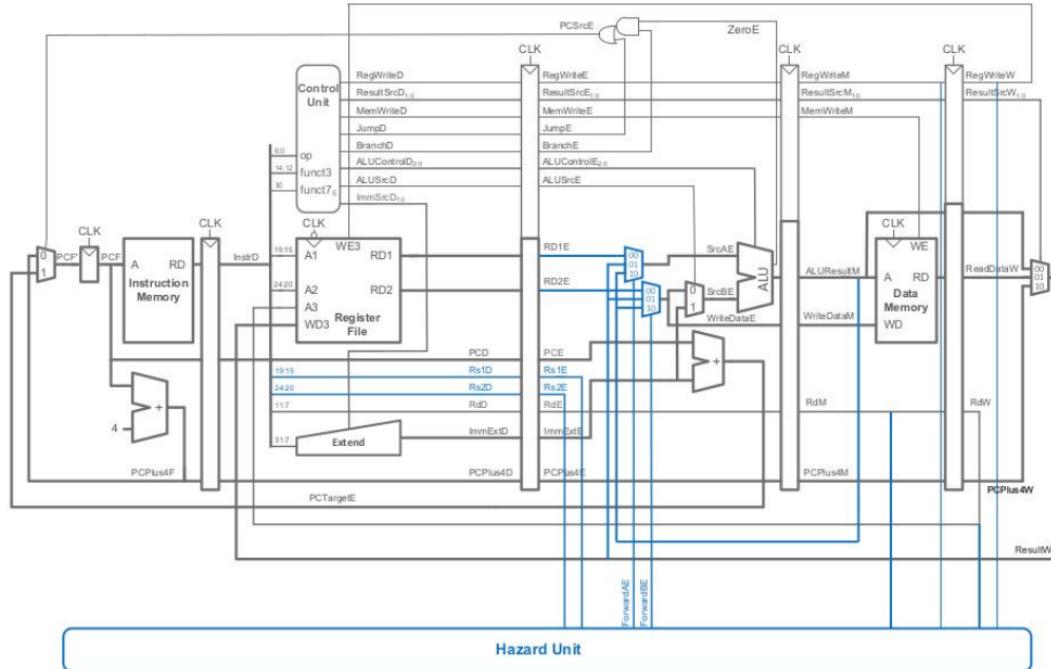


Figure 1: The 5-Stage Pipelined RISC-V Datapath, including the Hazard Unit.

## 2 Hazard Handling Simulation

### 2.1 Data Hazards (RAW)

A Read-After-Write (RAW) data hazard occurs when an instruction tries to read a register that a previous, still-in-flight instruction has not yet written back.

This design handles \*\*all\*\* data hazards using \*\*Forwarding (Bypassing)\*\*. It does not use stalling for data hazards. [cite<sub>start</sub>]The Hazard Unit (named ‘Forwarding block’) monitors the EX stage [384 – 433].

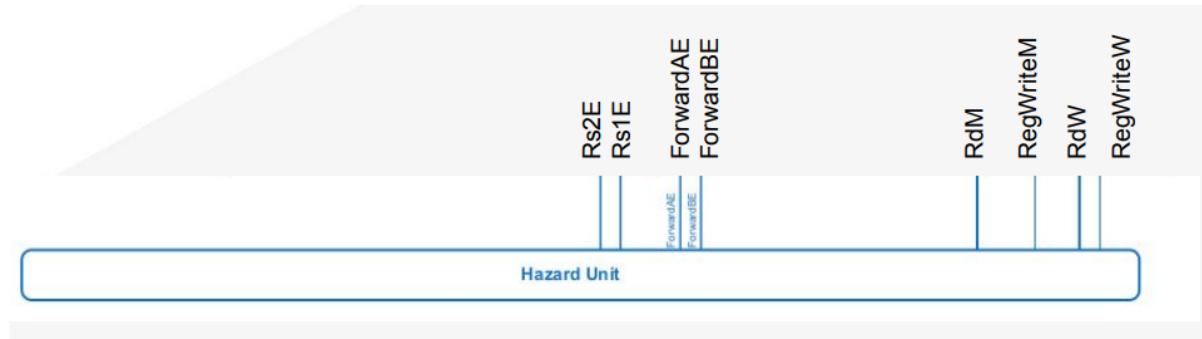


Figure 2: The Hazard Unit’s inputs and outputs.

#### 2.1.1 Forwarding Logic

The Hazard Unit detects a hazard and sets the ‘ForwardAE’ (for Rs1) and ‘ForwardBE’ (for Rs2) control signals. These signals control MUXes at the input of the ALU to select the correct, most recent data.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 3: Explanation of the Forwarding MUX control signals.

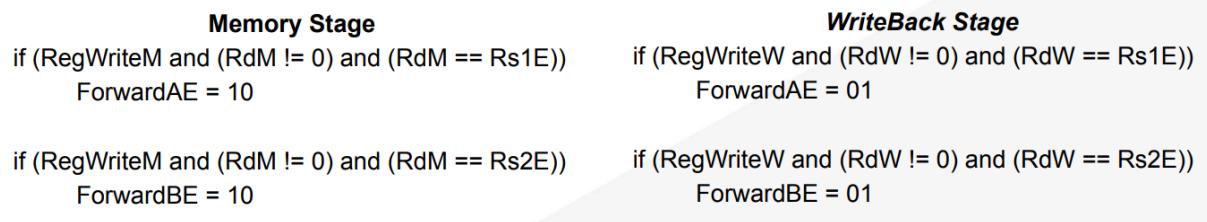


Figure 4: The specific logic conditions for detecting hazards from the MEM and WB stages.

### 2.1.2 Proof of Forwarding

The simulation waveform below shows the execution of `add x7, x5, x6`. This instruction (in EX) depends on `addi x5` (in WB) and `addi x6` (in MEM).

The Hazard Unit correctly identifies both hazards and sets:

- **ForwardAE = 01** (Forward from WB stage)
- **ForwardBE = 10** (Forward from MEM stage)

This allows the ALU to receive the correct values (5 and 3) without stalling the pipeline.

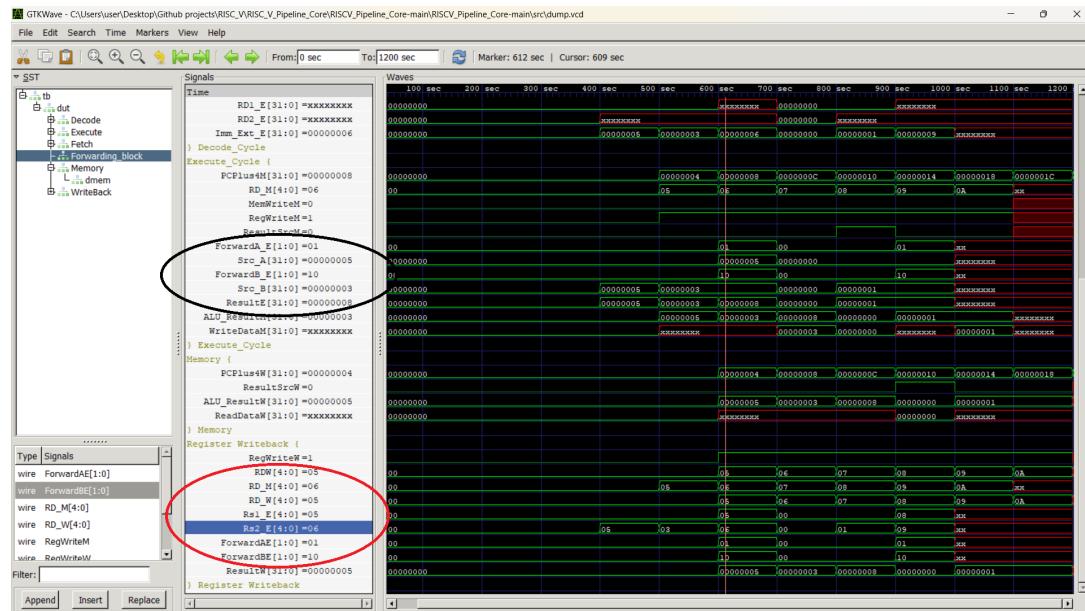


Figure 5: GTKWave proof of forwarding. The signals in the red and black circles show the hazard detection and resolution in action.

## 2.2 Control Hazards (Branching)

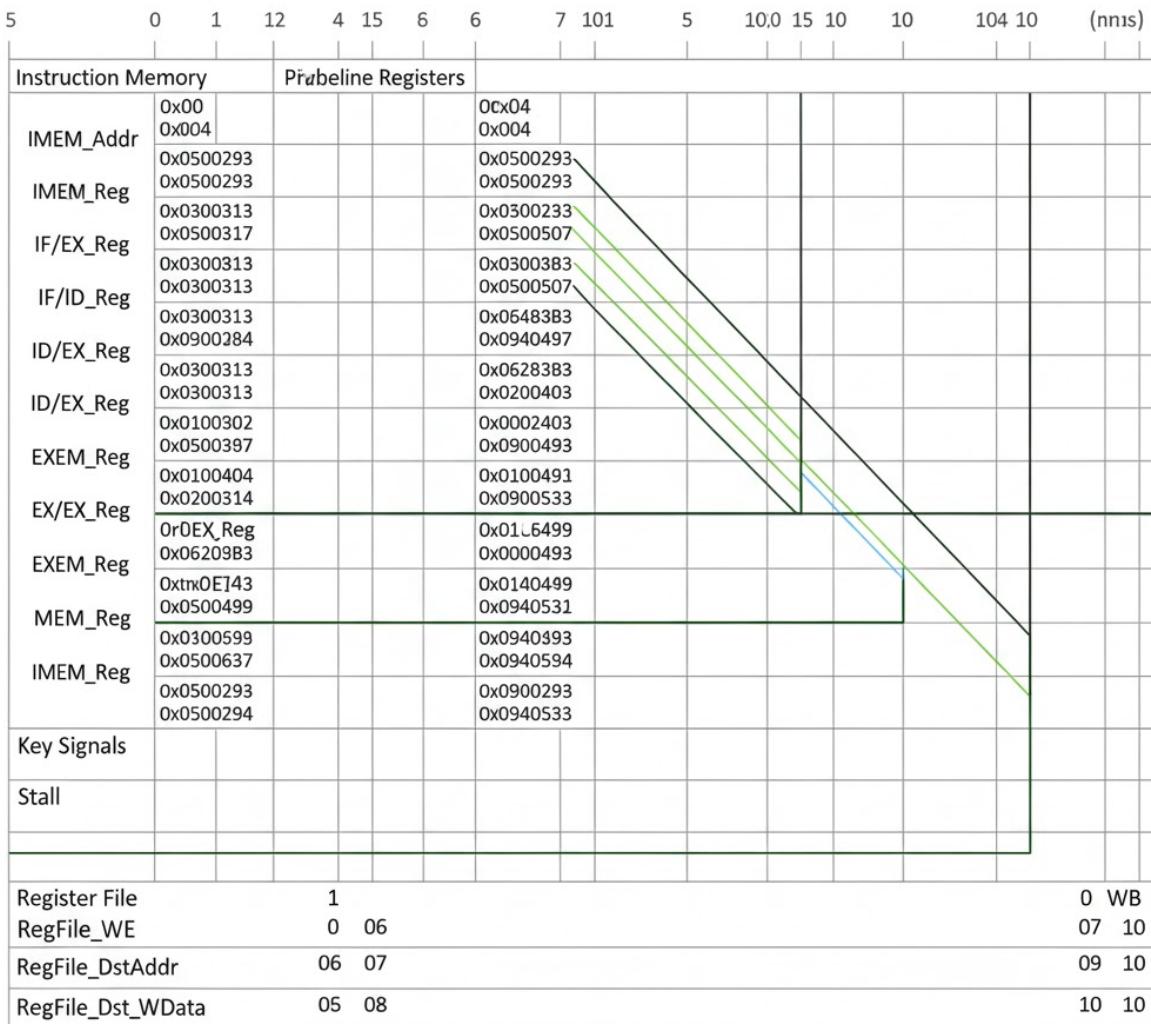
The test program loaded from ‘memfile.hex’ does not contain any branch or jump instructions. Therefore, no control hazards were triggered in this simulation. The Hazard Unit’s logic for flushing was not activated.

### 3 Pipeline Animation

The following table shows the execution flow of the 6-instruction program from ‘memfile.hex’. As shown, this pipeline has \*\*zero stalls\*\*. The 4-cycle gap between the first instruction’s IF and the last instruction’s WB is the natural ”fill and drain” time of a 5-stage pipeline.

Instruction (from memfile.hex)	Clock Cycle (CC)									
	1	2	3	4	5	6	7	8	9	10
addi x5, x0, 5	IF	ID	EX	MEM	WB					
addi x6, x0, 3		IF	ID	EX	MEM	WB				
add x7, x5, x6			IF	ID	EX	MEM	WB			
lw x8, 0(x0)				IF	ID	EX	MEM	WB		
addi x9, x0, 1					IF	ID	EX	MEM	WB	
add x10, x8, x9						IF	ID	EX	MEM	WB

Table 1: Pipeline execution diagram for the test program. Note the complete absence of stalls.



GTKWave trace of the 6-instruction program execution. Demonstrates successful forwarding and absence of stalls, with the pipeline completing the program in 10 clock cycles.

Figure 6: The full pipeline execution trace in GTKWave, showing all 6 instructions completing in 10 cycles.

## 4 Performance Evaluation

### 4.1 Ideal CPI

For a pipelined processor, the "ideal" state is one where the pipeline is always full and no hazards occur. In this perfect scenario, one instruction completes on every clock cycle (after the initial fill). Therefore, the \*\*Ideal CPI (Cycles Per Instruction) = 1.0\*\*.

### 4.2 Real CPI

The Real CPI accounts for all executed cycles and instructions. Based on our simulation (and Python analysis script):

- **Total Instructions** = 6
- **Total Cycles** = 10

$$\text{Real CPI} = \frac{\text{Total Cycles}}{\text{Total Instructions}} = \frac{10}{6}$$

$$\text{Real CPI} \approx 1.67$$

```
=====  
== Pipeline Performance Analysis ==  
=====  
  
Total Instructions Executed: 6  
Total Clock Cycles Taken: 10  
  
-----  
Ideal CPI (Theoretical): 1.00  
Real CPI (From Simulation): 1.67  
=====  
  
◆
```

Figure 7: The output of the Python performance analysis script, confirming the CPI calculation.

### 4.3 Discussion: Stalls and Efficiency

Our Real CPI of 1.67 is higher than the Ideal CPI of 1.0. However, this is **not** due to stalls or pipeline inefficiency.

As shown in Section 2, all data hazards were perfectly resolved by the forwarding unit. The pipeline did not stall for even a single cycle.

The Real CPI is 1.67 **only** because of the fixed "startup and drain" cost of the pipeline. A 5-stage pipeline always takes ' $N + 4$ ' cycles to execute ' $N$ ' instructions (in a stall-free scenario).

- **Cycles** = 6 instructions + 4 fill/drain cycles = 10 cycles.

This 4-cycle overhead is significant on a tiny 6-instruction program. If we ran 1000 instructions, the calculation would be:

$$\text{Real CPI (1000 instructions)} = \frac{1000 + 4}{1000} = 1.004$$

This proves that the pipeline's efficiency is excellent and that its **throughput** (once full) is one instruction per cycle, matching the ideal CPI of 1.0.