

# **Early Implementation Status and Challenges Report**

## **RISC-V Simulator Pipelining**

Rishit Mittal  
Roll No: CS24BTECH11053

October 29, 2025

Project Status Report

## Abstract

This report covers my progress in changing the single-cycle RISC-V simulator into a 5-stage pipelined processor. I am rebuilding the core logic to run one "clock cycle" at a time, not one full instruction. This report explains the new design, what I have built so far, and the main problems I expect to face next. I also mention some extra challenges found during the work.

## 1 Project Overview & Design Plan

The main goal is to convert the simulator to a 5-stage pipeline (IF-ID-EX-MEM-WB). The old `Step()` command is being replaced by a `ClockTick()` function. This new function runs the pipeline stages backward (WB → IF) to simulate how real hardware works.

Data and control signals will now pass between stages using four new struct types, called pipeline registers.

**Figure 1: Example Pipeline Register (ID/EX)**

```
1 // In pipeline_registers.h
2
3 // Holds signals needed later (in EX, MEM, WB stages)
4 struct ControlSignals {
5     uint8_t alu_op = 0;      // What the ALU should do
6     bool    alu_src = false; // ALU input: register or immediate?
7     bool    mem_read = false;
8     bool    mem_write = false;
9     bool    reg_write = false; // Write back to register file?
10    bool    mem_to_reg = false; // Data from memory or ALU?
11 };
12
13 // Data passed from Decode (ID) to Execute (EX) stage
14 struct ID_EX_Register {
15     uint64_t rs1_data = 0;      // Value from register rs1
16     uint64_t rs2_data = 0;      // Value from register rs2
17     int64_t  immediate = 0;    // Sign-extended immediate value
18     uint8_t   rd_addr = 0;     // Destination register number
19     uint8_t   funct3 = 0;      // Needed for load/store types
20
21     ControlSignals control; // Pass control signals along
22     bool    valid = false;   // Is this a real instruction or a
23     bubble?
```

## 2 Implementation Status

I am currently making big changes to the simulator's core code. I am moving parts of the old, large functions (like the old `Execute()` and `WriteMemory()`) into new, smaller functions for each pipeline stage: `DoFetch()`, `DoDecode()`, `DoExecute()`, `DoMemory()`, and `DoWriteBack()`.

Each new stage function does three things:

1. Reads data and signals from the pipeline register before it.
2. Does only the work for that specific stage (like the ALU calculation in Execute, or memory access in Memory).

3. Writes its results and passes needed info to the next pipeline register for the next cycle.

Here is the new Decode stage function as an example. I have made similar changes to the other four stages, moving small pieces of the old code into the right place.

**Figure 2: Example Change - New ‘DoDecode()’ Stage**

```
1 // This function runs during the Decode stage of the pipeline
2 void RVSSVM::DoDecode() {
3     // Get instruction from the previous (Fetch) stage
4     const uint32_t instruction = if_id_reg_.instruction;
5
6     // If the instruction is not valid (a bubble), pass it on
7     if (!if_id_reg_.valid) {
8         id_ex_reg_ = {}; // Make the next stage get a NOP
9         id_ex_reg_.valid = false;
10        return;
11    }
12
13    // --- Logic moved from the old Execute() function ---
14    // Find the register numbers used by this instruction
15    uint8_t rs1_addr = (instruction >> 15) & 0b11111;
16    uint8_t rs2_addr = (instruction >> 20) & 0b11111;
17    uint8_t rd_addr = (instruction >> 7) & 0b11111;
18
19    // --- Logic moved from the old Decode() function ---
20    // Figure out the control signals (MemRead, RegWrite, etc.)
21    control_unit_.SetControlSignals(instruction);
22
23    // --- Logic moved from the old Execute() function ---
24    // Read the values from the register file
25    uint64_t rs1_data = registers_.ReadGpr(rs1_addr);
26    uint64_t rs2_data = registers_.ReadGpr(rs2_addr);
27
28    // --- Fill the ID/EX pipeline register for the next stage ---
29    id_ex_reg_.rs1_data = rs1_data;
30    id_ex_reg_.rs2_data = rs2_data;
31    id_ex_reg_.immediate = ImmGenerator(instruction); // Get
immediate value
32    id_ex_reg_.rd_addr = rd_addr;
33    id_ex_reg_.funct3 = (instruction >> 12) & 0b111; // Pass funct3
34
35    // Pass the control signals to the next stage
36    id_ex_reg_.control.alu_op = control_unit_.GetAluOp();
37    id_ex_reg_.control.alu_src = control_unit_.GetAluSrc();
38    id_ex_reg_.control.mem_read = control_unit_.GetMemRead();
39    id_ex_reg_.control.mem_write = control_unit_.GetMemWrite();
40    id_ex_reg_.control.reg_write = control_unit_.GetRegWrite();
41    id_ex_reg_.control.mem_to_reg = control_unit_.GetMemToReg();
42
43    id_ex_reg_.valid = true; // Mark this data as valid
44 }
```

## 3 Challenges Encountered & Expected

The simulator does not work correctly yet. I found several major problems during this work that I need to solve.

### 3.1 Core Pipeline Problems (Hazards)

- **Data Hazards (Load-Use):** Need a `HazardDetectionUnit` to check if an instruction tries to use data that is still being loaded by the instruction just before it (e.g., ‘add’ right after ‘lw’). The pipeline must be stopped (stalled) until the data is ready.
- **Control Hazards (Branches):** This is the trickiest part. Need logic to find branches (‘beq’, ‘bne’), guess if they will be taken, cancel the wrongly fetched instructions if the guess was wrong (flush), and tell the Fetch stage the correct address to fetch next.
- **Data Forwarding Unit:** To avoid unnecessary stalls, need a `ForwardingUnit` to send results from the EX and MEM stages directly back to the EX stage inputs if the next instruction needs them right away.

### 3.2 Problems Supporting Complex Instructions

While changing the code, I realized handling some instructions will be harder in a pipeline:

- **Multi-Cycle Instructions:** Instructions like multiply (‘mul’) and floating-point math (‘fadd.s’) take longer than one clock cycle in the Execute stage. I need to make the EX stage handle this and stall the pipeline correctly. This might also cause *structural hazards* if two long instructions try to use the same unit (like the multiplier) at the same time.
- **R4-Type Instructions (FMA):** Some instructions like Fused Multiply-Add (‘fmadd.s’) need to read three source registers (‘rs1’, ‘rs2’, ‘rs3’). My current pipeline registers (like `ID_EX_Register`) only have space for two. To support these, I would need to change the pipeline registers and the Decode stage logic.

### 3.3 Other Challenges

- **Undo/Redo Feature:** The old `Undo/Redo` saved the changes from one instruction. Now, up to 5 instructions are running at once. This feature won’t work anymore. It would need a total redesign (saving the whole pipeline state each cycle) or removal.

## 4 Results & Next Steps

### 4.1 Current Results

The simulator is not runnable yet because of these major changes. The main result so far is a new code structure that is organized into pipeline stages. This makes it ready for adding the hazard and forwarding logic.

### 4.2 Next Steps

My plan is to get a simple pipeline working first, then add the harder parts.

1. Finish moving the basic logic for ‘add’, ‘addi’, ‘lw’, and ‘sw’ into the new ‘Do...()’ stage functions. Test that data flows correctly through the pipeline registers (without hazard detection yet).

2. Build the ‘HazardDetectionUnit‘ to handle stalls for load-use hazards.
3. Build the ‘ForwardingUnit‘ to handle data forwarding.
4. Add basic logic for handling branches (like predict-not-taken and flushing).