Pipelined RISC-V Processor Implementation Report

Beyrem Hadj Fredj

January 11, 2026

Repository Link: `https://github.com/BeyremHF/RISC-V-Processor-in-Verilog/tree/RV-PL`

1 Implementation Overview

This report documents the implementation of a 5-stage pipelined RISC-V processor (RV32I subset) with comprehensive hazard detection and handling capabilities.

1.1 Implemented Instructions

The processor supports 24 RISC-V instructions across 6 instruction types:

- **R-type:** ADD, SUB, AND, OR, XOR, SLT, SLTU, SLL, SRL, SRA
- **I-type (arithmetic):** ADDI, ANDI, ORI, XORI, SLTI, SLTIU, SLLI, SRLI, SRAI
- **Load:** LW
- **Store:** SW
- **Branch:** BEQ
- **Jump:** JAL
- **Upper Immediate:** LUI

1.2 Test Results

The processor was validated using a comprehensive test program covering all 24 instructions with hazard scenarios; 38/38 tests passed.

2 System Architecture

2.1 Hazard Detection and Handling Logic

The hazard unit detects and resolves three types of hazards through the following logic:

2.1.1 RAW (Read After Write) Hazard - Data Forwarding

Detection Logic:

- Compare source register addresses in EX stage (`E_rf_a1`, `E_rf_a2`) with destination addresses in MA and WB stages (`M_rf_a3`, `W_rf_a3`)
- Verify that the instruction in MA/WB will write to register file (`M_we_rf`, `W_we_rf`)
- Ensure destination register is not x0 (hardwired to zero)

Resolution:

- Forward `M_alu_o` (MA stage result) or `W_alu_o` (WB stage result) to ALU inputs via 3-input MUXes
- Priority: MA stage forwarding > WB stage forwarding > normal pipeline value
- Register file writes on negative clock edge for same-cycle availability

2.1.2 Load-Use Hazard (LW) - Stall and Flush

Detection Logic:

- Detect LW instruction in EX stage by checking `E_opcode == 7'b0000011`
- Verify that instruction will write to register file (`E_we_rf == 1`)
- Compare source registers in ID stage (`D_rf_a1`, `D_rf_a2`) with LW destination register (`E_rf_a3`)

Resolution:

- Stall PC and PLR1 by setting `PC_en = 0` and `PLR1_en = 0`
- Flush PLR2 by setting `PLR2_clr = 1` (insert NOP/bubble in EX stage)
- Pipeline resumes after 1-cycle stall when data becomes available from memory

2.1.3 Control Hazard (Branch/Jump) - Flush

Detection Logic:

- Detect branch taken: `E_branch == 1` AND `E_zero == 1`
- Detect jump: `E_jump == 1`

Resolution:

- Flush PLR1 and PLR2 by setting `PLR1_clr = 1` and `PLR2_clr = 1`
- Discard two instructions fetched after branch/jump (converted to NOPs)
- PC loads new target address on next cycle

2.2 LW Hazard Flushing Analysis

In the LW hazard example from Figure 3b, instructions x2, x3, and x4 obtain correct values regardless of whether PLR2 is flushed. This occurs because:

- The stall forces these instructions to wait in ID stage until LW completes
- When they finally reach EX stage, the loaded value is already available in WB stage
- Data forwarding from WB stage provides the correct value
- The flush merely clears incorrect intermediate processing but doesn't affect final results

Demonstration Program: Flushing makes a critical difference when the dependent instruction performs a memory write:

```
lw x1, 0(x0)        # Load value into x1
sw x1, 4(x0)        # Store x1 to memory (depends on x1)
```

Without flushing PLR2, the SW instruction would use the old x1 value already in the pipeline, writing incorrect data to memory. Flushing ensures the SW instruction re-fetches x1 after LW completes.

2.3 Data Forwarding, Flushing, and Stalling Mechanisms

Data Forwarding:

- Circuit: 3-input MUXes before ALU inputs, controlled by hazard unit
- Effect: Bypasses register file by routing results directly from MA/WB stages to EX stage
- Eliminates RAW hazard delays (0 cycle penalty)

Flushing:

- Circuit: CLR signal forces pipeline register outputs to zero (NOP instruction)
- Effect: Converts incorrectly-fetched instructions into bubbles (no operation)
- Used for LW hazards (1 bubble) and control hazards (2 bubbles)

Stalling:

- Circuit: EN signal prevents pipeline registers from updating (holds current value)
- Effect: Freezes earlier pipeline stages while later stages complete
- Prevents new instructions from entering pipeline during LW hazard (1 cycle penalty)

2.4 Hazard Control Signals Table

Hazard Type	Signal Name	Conditions	Encoding
6*RAW	E_forward_alu_op1	(E_rf_a1 == M_rf_a3) AND M_we_rf AND (M_rf_a3 != x0)	2'b01: Forward from MA stage
	E_forward_alu_op1	(E_rf_a1 == W_rf_a3) AND W_we_rf AND (W_rf_a3 != x0)	2'b10: Forward from WB stage
	E_forward_alu_op1	No hazard detected	2'b00: Use normal pipeline value
	E_forward_alu_op2	Same conditions as above but for E_rf_a2	Same encoding as alu_op1
3*LW	PC_en	(E_opcode == LW) AND E_we_rf AND ((D_rf_a1 == E_rf_a3) OR (D_rf_a2 == E_rf_a3))	1'b0: Stall PC
	PLR1_en	Same as PC_en	1'b0: Stall PLR1
	PLR2_clr	Same as PC_en	1'b1: Flush PLR2
2*Control	PLR1_clr	(E_branch AND E_zero) OR E_jump	1'b1: Flush PLR1
	PLR2_clr	(E_branch AND E_zero) OR E_jump	1'b1: Flush PLR2

Table 1: Complete Hazard Control Signals

3 Performance Analysis

3.1 CPI Estimation

Based on the test program execution with 38 instructions:

- **Instructions without hazards:** 30 instructions × 1 cycle = 30 cycles
- **Instructions with RAW hazards:** 5 instructions × 1 cycle = 5 cycles (0 penalty with forwarding)
- **LW hazard:** 1 instruction × 2 cycles = 2 cycles (1 cycle stall penalty)
- **Branch hazard:** 1 instruction × 3 cycles = 3 cycles (2 cycle flush penalty)
- **Jump hazard:** 1 instruction × 3 cycles = 3 cycles (2 cycle flush penalty)

Total Cycles: 30 + 5 + 2 + 3 + 3 = 43 cycles
Average CPI: 43 ÷ 38 = 1.13

The pipeline achieves near-ideal CPI of 1, with only minor overhead from load-use and control hazards.

3.2 Critical Path Analysis

Using delay values from Table 2:

Single-Cycle Processor

Critical Path: Register clk-Q (40ps) + Register Read (100ps) + Mux (30ps) + ALU (120ps) + Mem Read (200ps) + Mux (30ps) + Register Setup
Total Delay: 40 + 100 + 30 + 120 + 200 + 30 = **520ps**
Maximum Frequency: $1 \div 520\text{ps} = 1.92$ GHz

Multi-Cycle Processor

Critical Path: Memory Read stage: Register clk-Q (40ps) + Mux (30ps) + Mem Read (200ps) + Mux (30ps)
Total Delay: 40 + 30 + 200 + 30 = **300ps**
Maximum Frequency: $1 \div 300\text{ps} = 3.33$ GHz

Pipelined Processor

Critical Path: Memory Access (MA) stage: Register clk-Q (40ps) + Mem Read (200ps)
Total Delay: 40 + 200 = **240ps**
Maximum Frequency: $1 \div 240\text{ps} = 4.17$ GHz

Pipeline Frequency Analysis

Question: For a 5-stage pipelined architecture, we expect that the max. frequency is 5× that of the single-cycle one. Is that true? Why?
Answer: No, this is not true. The actual frequency improvement is only 2.17× (4.17 GHz / 1.92 GHz), not 5×.
Reasons:

- Pipeline frequency is limited by the *slowest stage*, not average stage delay
- Register overhead (clk-Q delay) is added to every stage
- In our design, the MA stage (memory read) dominates with 240ps delay
- Ideal 5× speedup assumes all stages have equal delays of $520\text{ps} \div 5 = 104\text{ps}$
- Memory operations cannot be subdivided further, creating bottleneck

To achieve closer to 5× speedup, memory access would need to be split into multiple pipeline stages or use faster memory technology.

3.3 Performance Comparison

Architecture	CPI	Frequency (GHz)	Relative Performance
Single-Cycle	1.00	1.92	1.00×
Multi-Cycle	4.00	3.33	0.43×
Pipelined	1.13	4.17	**1.92**×

Table 2: Performance Comparison (Relative to Single-Cycle)

The pipelined processor achieves **1.92× better performance** than single-cycle through combined benefits of near-CPI=1 throughput and higher clock frequency.