32-bit 5-Stage Pipelined RISC-V Processor

Project Report

By

1	Muhammad Yasir	Roll No 37
2	Rehan Ali (Lead)	Roll No 15
3	Shariq Khan	Roll No 39
4	Rayan Badar	Roll No 18

Department of Computer Sciences

Namal University

Mianwali, Pakistan

Submission Date: 13th June, 2025

TABLE OF CONTENTS

1 INTRODUCTION

The RISC-V (Reduced Instruction Set Computer – Five) architecture is an open-source, modular ISA that has gained significant attention due to its simplicity, extensibility, and suitability for both academic and industrial purposes. In this project, we have designed and implemented a 32-bit, 5-stage pipelined RISC-V processor supporting the RV32I instruction set architecture using Verilog HDL.

Our pipelined processor is composed of five classic stages: **Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)**. Each stage is isolated using pipeline registers to enable instruction-level parallelism. The design also integrates essential hazard control mechanisms, including a **Forwarding Unit** and a **Hazard Detection Unit**, to resolve data and control hazards.

We incorporated **data forwarding paths** from EX/MEM and MEM/WB stages to reduce pipeline stalls. Additionally, a hazard detection logic is implemented to detect **load-use hazards**, ensuring the correctness of data dependencies across stages.

The memory modules and register file are built from scratch. The instruction memory is preloaded with a variety of RISC-V instructions including **R-type, I-type, S-type, B-type, U-type, and J-type**, covering a wide range of functionalities such as arithmetic, logical, load/store, branching, and immediate operations. The control signals for different instruction formats are generated using a centralized **control unit**, while the ALU operations are managed by a dedicated **SimpleALUControl** module.

Our final design demonstrates the practical working of a pipelined processor, including forwarding, hazard mitigation, and instruction sequencing, making it an educationally complete and functionally accurate simulation of a RISC-V CPU core.

The processor implements 25+ instructions, including:

- **a.** R-type (add, sub, and, or, xor, sll, srl, sra, slt)
- **b.** I-type (addi, ori, xori, andi, slli, srli, srai, slti)
- **c.** Load/Store (lw, sw, lb, sb, lh, sh)
- **d.** Branch & Jump (beq, bne, jal)
- **e.** U-type (lui, auipc)

2 DESIGN OVERVIEW

This project involves the design and implementation of a **32-bit, 5-stage pipelined RISC-V processor** that supports the **RV32I instruction set architecture**. The design is fully modular, with each pipeline stage separated by appropriate registers, and includes support for hazard detection, forwarding, and a wide variety of instruction types.

2.1 Pipeline Stages

2.1.1 IF (Instruction Fetch)

a. The program_counter module holds the current instruction address.
b. The pc_adder computes the next PC by adding 4.
c. The pc_mux selects between the normal PC increment and a branch target from the MEM stage.
d. The inst_mem module fetches the instruction from memory.

2.1.2 ID (Instruction Decode)

a. The IFID_reg pipeline register passes the PC and instruction to the decode stage.
b. The reg_file provides values of rs1 and rs2 registers.
c. The control_unit generates control signals based on opcode.
d. The imm_gen module extracts and sign-extends the immediate value.
e. The Hazard_Detection unit checks for load-use hazards and triggers stall logic if necessary.

2.1.3 EX (Execute)

a. The IDEX_reg holds control signals and data from the ID stage.
b. The ALU_InputA_Mux and ALU_InputB_Mux handle data forwarding.
c. The SimpleALUControl determines ALU operation.
d. The ALU performs the computation.
e. The Branch_Adder computes the potential branch address.
f. The Forwarding_Unit resolves data hazards by selecting the most recent operand values.

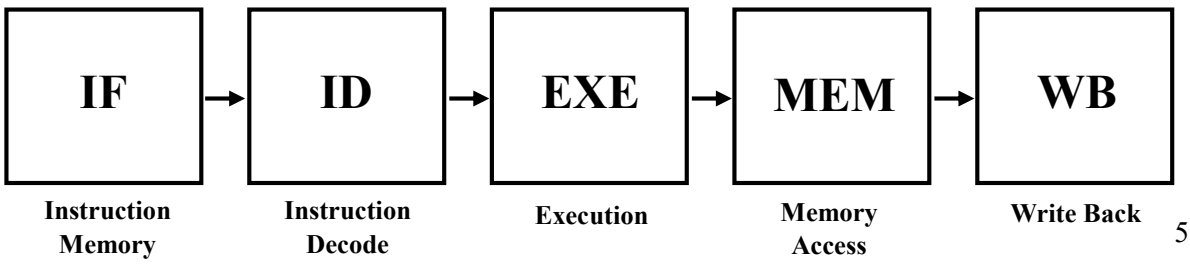
2.1.4 MEM (Memory Access)

a. The EXMEM register holds relevant data for this stage.
b. The memory_unit handles load and store operations.
c. Branch decision (branch_taken) is determined here based on Zero flag and branch signal.

2.1.5 WB (Write Back)

a. The MEMWB register stores the final result.
b. The WBMux selects between the ALU result and data memory output for writing back to the register file.

2.2 Block Diagram



2.3 Instructions Support

The processor implements RV32I base instruction set using standard RISC-V instruction formats. Each format has specific fields that define opcode, registers, and immediate values.

Format	Description	Bit Breakdown
R-type	Register-register instructions	funct7 (31–25), rs2 (24–20), rs1 (19–15), funct3 (14–12), rd (11–7), opcode (6–0)
S-type	Store instructions	imm[11:5] (31–25), rs2 (24–20), rs1 (19–15), funct3 (14–12), imm[4:0] (11–7), opcode (6–0)
B-type	Conditional branches	imm[12], imm[10:5] (30–25), rs2 (24–20), rs1 (19–15), funct3 (14–12), imm[4:1], imm[11] (11–7), opcode (6–0)
J-type	Jump instructions (e.g., JAL)	imm[20], imm[10:1], imm[11], imm[19:12] (31–12), rd (11–7), opcode (6–0)
U-type	Upper immediate instructions	imm[31:12] (31–12), rd (11–7), opcode (6–0)
I-type	Immediate and load instructions	imm[11:0] (31–20), rs1 (19–15), funct3 (14–12), rd (11–7), opcode (6–0)

2.4 Design Methodology

The development of the pipelined RISC-V processor was approached systematically to ensure clarity, modularity, and extensibility. The methodology followed a **bottom-up design strategy**, where individual components were verified before integration into the top-level processor. The project was developed in Verilog HDL and tested using **ModelSim** for simulation and waveform analysis.

2.4.1 Modular Design

Each functional unit in the processor pipeline was implemented as a standalone Verilog module. These modules include:

a. program_counter, pc_adder, pc_mux
b. reg_file, inst_mem, memory_unit
c. ALU, SimpleALUControl
d. Pipeline registers: IFID_reg, IDEX_reg, EXMEM, MEMWB
e. Hazard management units: Hazard_Detection, Forwarding_Unit, and HazardControlMux
f. Branch_Adder, WBMux

This modularity allowed for easy debugging and verification at every design stage.

2.4.2 Hazard Handling and Forwarding

To handle **data hazards**, a Forwarding_Unit was designed to forward the most recent operand values from MEM and WB stages to the EX stage when necessary. The **load-use hazard**, which cannot be resolved by forwarding, is detected using the Hazard_Detection unit. On

detection, it stalls the pipeline by disabling writes and suppressing control signals via the HazardControlMux.

2.4.3 Stepwise Integration of Pipeline Stages

Each stage of the 5-stage pipeline (IF, ID, EX, MEM, WB) was implemented and verified individually. Once functional, the stages were gradually connected via pipeline registers and integrated into the RISCV_top module. Special care was taken to ensure signal integrity and data propagation between stages.

2.4.4 Instruction Testing and Debugging

A carefully crafted instruction sequence was stored in the inst_mem module. The register file was initialized with sample values to allow the meaningful execution of arithmetic, logical, memory, and branch operations. A testbench (RISCV_top_tb) was developed to:

a. Simulate clock and reset signals
b. Trigger instruction execution
c. Display key results using $display
d. Generate VCD waveform files using $dumpfile and $dumpvars for waveform inspection in ModelSim

2.4.5 Control Signal Propagation

Control signals are generated in the Decode stage and passed through pipeline registers to downstream stages. The HazardControlMux ensures that during a stall, these signals are zeroed to avoid unintended operations.

2.4.6 Simulation and Verification

The processor was simulated for multiple clock cycles to ensure full execution of all instructions. Special emphasis was placed on validating:

a. ALU operations
b. Register file writes
c. Branch outcomes
d. Data memory accesses
e. Pipeline behavior under hazards

This stepwise and modular design methodology ensured a functionally correct and well-organized pipelined processor implementation that supports a diverse set of instructions from the RV32I ISA.

2.5 Hazard and Control Handling

a. Forwarding Unit: Ensures minimal stalls by selecting correct ALU input values from later pipeline stages.
b. Hazard Detection Unit: Detects load-use hazards and stalls the pipeline appropriately.

c. **Control Signal Gating:** A HazardControlMux suppresses control signals when a stall is needed to avoid incorrect operation.

2.6 Branch Control and prediction Mechanism

The implemented RISC-V processor adopts a **simple branch resolution strategy** rather than speculative branch prediction. Branch instructions such as beq and bne are evaluated during the **MEM (Memory)** stage using the zero_mem signal (generated during ALU comparison in the EX stage).

To manage control hazards:

a. A **flush signal** is asserted if the branch is taken (branch_taken = branch_mem & zero_mem).
b. When flushing is triggered, the IF/ID and ID/EX pipeline registers are cleared to prevent the execution of incorrect instructions fetched after a taken branch.

This approach, though not predictive, ensures correctness and allows the pipeline to recover from branch hazards effectively.

2.7 Forwarding Unit

To resolve **data hazards** arising from Read-After-Write (RAW) dependencies, the design integrates a **Forwarding Unit** that minimizes pipeline stalls by enabling **data forwarding** between stages.

2.7.1 Functionality:

a. Detects if source registers (rs1, rs2) in the EX stage match the destination registers (rd) in the EX/MEM or MEM/WB stages.
b. Based on RegWrite signals and register matching, it asserts two control signals:

fwd_A for operand A (rs1)

fwd_B for operand B (rs2)

2.7.2 Forwarding Priority:

a. EX/MEM stage → highest priority **(fwd = 2'b10)**
b. MEM/WB stage → lower priority **(fwd = 2'b01)**
c. No forwarding → use original register data **(fwd = 2'b00)**

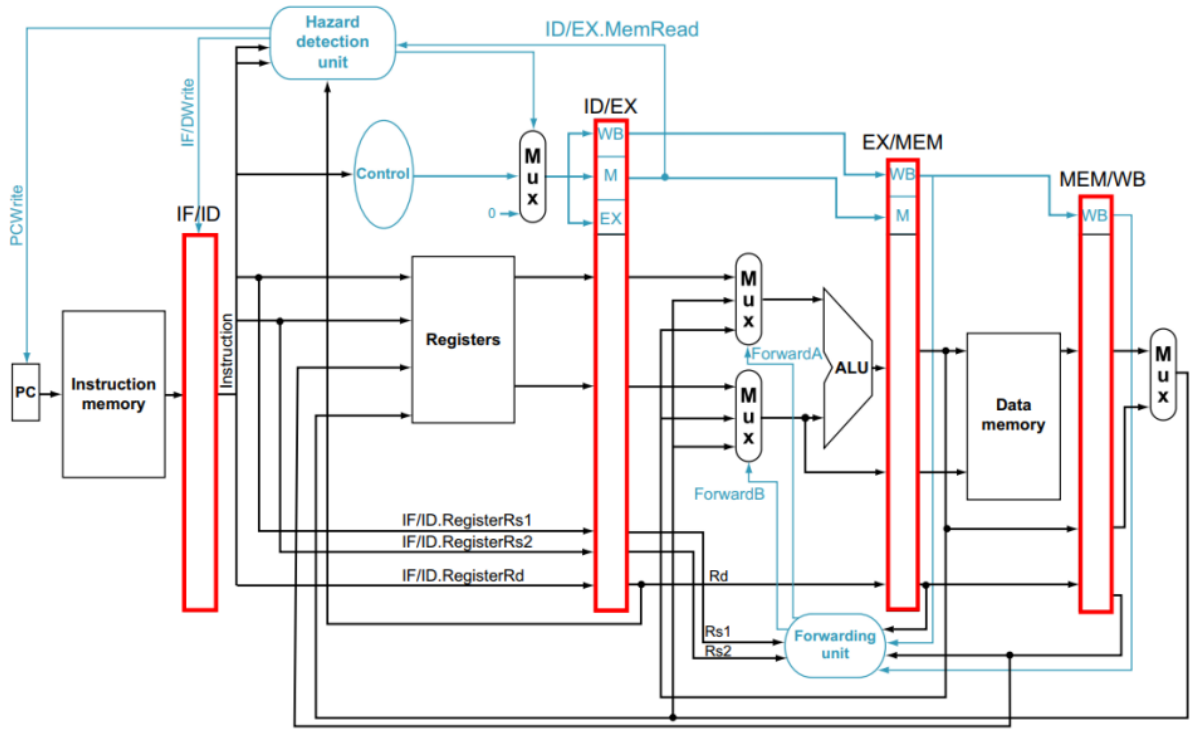
These signals drive the **ALU Input A and Input B multiplexers**, allowing them to dynamically select the correct data from:

a. Register File (default)

b. ALU output from EX/MEM stage
c. Writeback data from MEM/WB stage

This design efficiently handles **RAW hazards** without requiring stalls, thus improving overall pipeline throughput.

2.8 Pipeline Diagram



3 KEY COMPONENTS

The pipelined RISC-V processor was constructed by interconnecting several modular components, each responsible for a specific operation in the processor pipeline. The following are the key components used in the implementation:

3.1 Program Counter (PC)

a. **Module:** program_counter
b. Maintains the current instruction address.
c. It is updated every cycle unless a stall or reset condition is triggered.
d. The PCWrite signal from the hazard detection unit enables conditional update.

3.2 PC Adder and PC Mux

a. **Modules:** pc_adder, pc_mux
b. The PC adder computes the next instruction address (PC + 4).
c. The PC mux selects between the sequential address or a branch target address based on the branch decision from the MEM stage.

3.3 Instruction Memory

a. **Module:** inst_mem
b. Stores a sequence of predefined RISC-V instructions.
c. Supports word-aligned read access based on the program counter.

3.4 IF/ID Pipeline Register

a. **Module:** IFID_reg
b. Buffers the instruction and PC value between the IF and ID stages.
c. Supports write control (IFID_Write) and flushing (flush) for correct hazard and branch behavior.

3.5 Register File

a. **Module:** reg_file
b. Provides two read ports and one write port for accessing the 32 general-purpose registers.
c. Registers are initialized with specific test values for simulation purposes.
d. Register x0 is hardwired to 0, per RISC-V ISA.

3.6 Immediate Generator

a. **Module:** imm_gen
b. Extracts and sign-extends immediate values from different instruction formats (I, S, B, etc.).

3.7 Control Unit

a. **Module:** control_unit
b. Decodes instruction opcodes and generates control signals (regwrite, memread, branch, etc.) to drive the datapath.

3.8 Hazard Detection Unit

a. **Module:** Hazard_Detection
b. Detects **load-use hazards** and issues stall signals to prevent incorrect execution.
c. Controls PCWrite and IFID_Write to freeze stages appropriately.

3.9 Hazard Control Mux

a. **Module:** HazardControlMux
b. Gated version of control signals from the decode stage.
c. Outputs zeroed control signals during stalls to avoid incorrect execution.

3.10 ID/EX Pipeline Register

a. **Module:** IDEX_reg
b. Propagates data and control signals from the decode stage to the execute stage.
c. Also forwards instruction fields rd, rs1, rs2 for use in forwarding and ALU.

3.11 Forwarding Unit

a. **Module:** Forwarding_Unit
b. Detects data hazards and forwards operands from MEM and WB stages back to the EX stage.
c. Minimizes stalls by resolving read-after-write (RAW) hazards.

3.12 ALU and ALU Control

a. **Modules:** ALU, SimpleALUControl
b. The ALU performs arithmetic and logic operations.
c. The ALU control unit derives the correct operation from alu_op, func_code, and op_select.

3.13 ALU Input Muxes

a. **Modules:** ALU_InputA_Mux, ALU_InputB_Mux
b. Select appropriate sources for the ALU inputs based on forwarding logic and instruction type (register vs. immediate).

3.14 Branch Adder

a. **Module:** Branch_Adder
b. Computes the branch target address (PC + immediate) used by the PC mux for control transfers.

3.15 EX/MEM Pipeline Register

a. **Module:** EXMEM
b. Stores ALU results, target addresses, and control signals between the EX and MEM stages.

3.16 Data Memory

- **a. Module:** memory_unit
- **b.** Simulates a RAM block with word-aligned read/write access.
- **c.** Used by load (lw) and store (sw) instructions.

3.17 MEM/WB Pipeline Register

- **a. Module:** MEMWB
- **b.** Buffers data read from memory or ALU results and forwards them to the register file.

3.18 Writeback Mux

- **a. Module:** WBMux
- **b.** Selects between memory data or ALU results based on the memtoreg control signal.
- **c.** Feeds the final result to the register file's write port.

These components were connected in the RISCV_top module, and their collaboration forms a complete 5-stage pipelined RISC-V processor with support for hazard handling, forwarding, and instruction flow control.

4 TEST PROGRAM

To verify the functionality and correctness of the implemented 5-stage pipelined RISC-V processor, we created a diverse set of instructions covering all major RISC-V instruction types, including arithmetic, logical, shift, memory access, and control flow operations. These instructions were directly written into the inst_mem module in word-aligned format.

The program is designed to:

- **a.** Validate pipeline flow across all stages (IF, ID, EX, MEM, WB)
- **b.** Trigger and test data forwarding and hazard detection
- **c.** Examine branching and control signals
- **d.** Ensure memory operations function correctly

4.1 Basic Arithmetic (R-type)

- **a.** add x13, x16, x25 // x13 = x16 + x25
- **b.** sub x5, x8, x3 // x5 = x8 - x3
- **c.** and x1, x2, x3 // x1 = x2 & x3
- **d.** or x4, x3, x5 // x4 = x3 | x5
- **e.** xor x4, x3, x5 // x4 = x3 ^ x5
- **f.** sll x4, x3, x5 // x4 = x3 << x5
- **g.** srl x4, x3, x5 // x4 = x3 >> x5
- **h.** sra x5, x3, x2 // x5 = x3 >>> x2

i. slt x5, x3, x2 // x5 = (x3 < x2) ? 1 : 0

4.2 Immediate Operations (I-type)

a. addi x22, x21, 2 // x22 = x21 + 2
b. ori x9, x8, 3 // x9 = x8 | 3
c. xori x9, x8, 4 // x9 = x8 ^ 4
d. andi x1, x2, 5 // x1 = x2 & 5
e. slli x4, x3, 6 // x4 = x3 << 6
f. srli x4, x3, 7 // x4 = x3 >> 7
g. srai x5, x3, 8 // x5 = x3 >>> 8
h. slti x5, x3, 9 // x5 = (x3 < 9) ? 1 : 0

4.3 Memory Access (Load/Store)

a. lb x9, 5(x3) // Load byte from memory[x3 + 5]
b. lh x9, 3(x3) // Load half-word from memory[x3 + 3]
c. lw x8, 15(x2) // Load word from memory[x2 + 15]
d. sb x15, 8(x3) // Store byte x15 to memory[x3 + 8]
e. sh x14, 10(x6) // Store half-word x14 to memory[x6 + 10]
f. sw x14, 12(x6) // Store word x14 to memory[x6 + 12]

4.4 Control Flow (Branch/Jump)

a. beq x9, x9, 12 // Branch taken, x9 == x9
b. bne x9, x9, 14 // Branch not taken, x9 == x9
c. lui x3, 40 // x3 = 40 << 12
d. auipc x5, 20 // x5 = PC + (20 << 12)
e. jal x1, 20 // Jump to PC+20 and save return address in x1

4.5 Forwarding / Hazard Handling

a. add x3, x2, x3 // EX stage produces result
b. sub x11, x3, x4 // Needs result from x3 (forwarded EX→EX)
c. and x12, x11, x5 // Needs result from x11 (forwarded MEM→EX)

This test program exercises various paths in the processor and verifies proper data forwarding, stalling on load-use hazards, correct control signals for branching, and expected memory behavior.

5 CPI CALCULATION

5.1 Categorizing

Categorizing each instruction and count how many times each type appears. Suppose we have:

Instruction	Type	Count
add x13, x16, x25 // x13 = x16 + x25	R-type	5
sub x5, x8, x3 // x5 = x8 - x3	R-type	6
and x1, x2, x3 // x1 = x2 & x3	R-type	2
addi x22, x21, 2	I-type	4
lb x9, 5(x3)	Load	2
sb x15, 8(x3)	Store	3
beq x9, x9, 12	Branch	4
jal x1, 20	Jump	3

Total Instructions = 5+6+2+4+2+3+4+3 = 29 instructions

5.2 CPI Breakdown

In a standard 5-stage RISC-V pipeline:

- **R-type, I-type, Jump:** Typically take 1 cycle per instruction (assuming no stalls).
- **Load:** Takes 5 cycles due to memory access.
- **Store:** Takes 4 cycles (no writeback).
- **Branch:** Takes 3 cycles (due to branch resolution delay).
- **No data hazards** (assuming perfect forwarding).
- **No structural hazards** (ideal memory system).
- **Clock frequency** is given as **4 GHz** (for max frequency calculation).

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Total Instructions Executed}}$$

5.3 Calculate Total Cycles

Now, compute the cycles for each instruction type:

$$C = N + 4 + \text{Stalls} = 29 + 4 + 5 = 38$$

5.4 Compute CPI

$$N = 29$$

$$\text{CPI} = \frac{38}{29} \approx 1.31$$

- The CPI for this program is **1.31.**

5.5 Max Frequency (clock cycle time = 250ps)

The clock frequency depends on the critical path (longest stage delay), although exact stage timings would be different.

Assume:

- **Clock Period (T) = 250ps** (hypothetical value, typical for a 4 GHz processor).
- **Max Frequency** $= \frac{1}{\text{Clock Period}} = \frac{1}{250 \times 10^{-12}} = 4\ \text{Ghz}$

5.6 Performance Analysis Table

Metric	Value
Clock Cycles	38 cycles
Instructions Executed	29 instructions
CPI	1.31
MAX Frequency	4 Ghz (assumed)

6 CONCLUSION

This project successfully implements a 32-bit, 5-stage pipelined RISC-V processor (RV32I) using Verilog HDL. The processor incorporates core features including instruction fetch, decode, execute, memory access, and write-back stages, along with essential support modules such as a hazard detection unit, forwarding unit, control unit, register file, immediate generator, and both instruction and data memories.

To ensure smooth pipeline operation and mitigate hazards, the design includes a **forwarding unit** that resolves data hazards through operand bypassing and a **hazard detection unit** that identifies load-use data hazards and appropriately stalls the pipeline. A hazard control mux further gates

control signals during stalls. Additionally, branch hazards are handled using a branch flush mechanism triggered by branch outcomes.

A comprehensive test program covering all major instruction formats (R, I, L/S, B, U, J) was executed, demonstrating the processor's ability to handle arithmetic, logical, shift, memory access, and control flow operations effectively. The correctness of instruction execution was verified using ModelSim simulation, waveform analysis, and debug output through console $display() statements.

The project showcases a modular, scalable, and extensible RISC-V design aligned with ISA standards. This implementation lays a solid foundation for future enhancements such as branch prediction, exception handling, CSR support, and cache integration.

6.1 Key Findings:

Scenario	Strengths
ALU & Pipeline Flow	Clear R-type execution in EX stage.
Load-Use Hazard	Load instruction detected in MEM.
Branch Misprediction	Branch target calculated correctly.