Design and Implementation of a Five-Stage Pipelined RISC-V Processor with Hazard Handling

Muhammad Qasim
Dept. of Computer Engineering
Ghulam Ishaq Khan Institute
of Eng. Sciences and Technology
Pakistan
u2023488@giki.edu.pk

Muhammad Hammad
Dept. of Computer Engineering
Ghulam Ishaq Khan Institute
of Eng. Sciences and Technology
Pakistan
u2023420@giki.edu.pk

***Abstract*—This project presents the design and implementation of a five-stage pipelined processor based on the RISC-V instruction set architecture (ISA). The processor follows the classical pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The design supports a subset of RISC-V instructions including arithmetic, logical, load/store, and branch operations. To ensure correct execution and improved performance, critical data and control hazards are handled using a comprehensive set of techniques including data forwarding, hazard detection for load-use stalls, and pipeline flushing for control instructions. The processor is implemented in the hardware description language Verilog HDL and functionally verified through rigorous simulation using representative instruction sequences that demonstrate correct hazard detection and resolution.**

***Index Terms*—RISC-V, Pipeline Processor, Hazard Detection, Forwarding Unit, Verilog HDL, Computer Architecture.**

I. Introduction

A. *Background*

Pipelining is a fundamental technique used in modern processors to improve instruction throughput by overlapping the execution of multiple instructions [1]. This concurrency, achieved by dividing instruction processing into discrete stages, significantly reduces the average Cycles Per Instruction (CPI) towards the ideal value of 1. The challenge in pipelining lies in maintaining correct program semantics in the presence of hazards, which are structural, data, or control dependencies that disrupt the smooth flow of the pipeline.

B. *Project Objectives*

The primary objective of this project is to design and implement a five-stage pipelined processor conforming to the RISC-V ISA [2]. Specifically, the goals are to:

- Implement the five classical pipeline stages (IF, ID, EX, MEM, WB).
- Integrate a complete **Hazard Detection Unit** and a **Forwarding Unit** to correctly resolve data hazards.
- Implement control logic for pipeline flushing to manage branch (control) hazards.
- Verify the design using Verilog HDL to confirm functional correctness and hazard handling integrity.

C. *Scope of the Report*

This report details the architectural design and functional implementation of the pipelined datapath and its associated control unit. The implementation targets the RV32I base integer instruction set subset. It strictly focuses on the core pipeline structure and hazard management. Advanced microarchitectural features, such as cache memory (L1/L2), complex branch prediction schemes (e.g., two-bit predictor), and advanced exceptions/interrupts, are beyond the scope of this project [3].

II. Theoretical Background: Pipeline Concepts

A. *Basic Pipeline Structure*

The processor utilizes the classic five-stage organization:

1) **Instruction Fetch (IF):** Fetches the instruction from instruction memory at the address specified by the Program Counter (PC) and calculates $PC + 4$.
2) **Instruction Decode (ID):** Decodes the instruction, reads required source operands from the Register File, and calculates the sign-extended immediate value.
3) **Execute (EX):** The ALU performs the specified arithmetic/logical operation. Branch target addresses are calculated, and branch conditions are evaluated.
4) **Memory Access (MEM):** Accesses the data memory for load (`lw`) and store (`sw`) instructions.
5) **Write Back (WB):** Writes the final result (from the ALU or memory) back to the Register File.

Pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) are crucial components that isolate the stages, ensuring data integrity and synchronization across clock cycles.

B. *Pipelining Hazards*

Pipeline hazards degrade the ideal CPI by requiring instructions to pause or execute incorrectly.

1) Structural Hazards: Structural hazards occur when two instructions require the same hardware resource simultaneously. In this design, hazards are mitigated by using **separate instruction and data memories** (Harvard architecture style) and reading/writing the Register File in different half-cycles [4].

2) Data Hazards: Data hazards arise when an instruction attempts to use data that has not yet been written back by a preceding instruction. The primary concern is the **Read After Write (RAW)** hazard. The design uses the in-order pipeline structure, which inherently avoids Write After Write (WAW) and Write After Read (WAR) hazards.

3) Control Hazards: Control hazards, or branch hazards, occur when the pipeline fetches instructions sequentially but a branch instruction changes the program control flow (e.g., `beq`). The penalty is incurred when instructions fetched *after* the branch are flushed because the branch is taken.

C. *Hazard Mitigation Techniques*

1) Forwarding (Bypassing): The **Forwarding Unit** is a critical component that resolves most RAW data hazards. It checks if the required source operands (read in ID) are being written by instructions currently in the EX, MEM, or WB stages. If so, the required result is forwarded (bypassed) directly to the ALU inputs in the EX stage, eliminating the need for a stall.

2) Stall and Bubble Insertion: Forwarding cannot resolve **Load-Use Hazards** (e.g., `lw` followed immediately by an instruction using the loaded value). The **Hazard Detection Unit** detects this specific condition and generates control signals to stall the IF and ID stages and insert a **Bubble** (a NOP instruction) into the ID/EX pipeline register [1].

3) Control Hazard Handling (Flushing): If a branch is taken, the control unit asserts a **Flush** signal, which turns the instruction in the IF/ID register into a NOP, thereby flushing the erroneously fetched instruction. A branch penalty of one cycle is incurred when a branch is taken.

III. Processor Design and Implementation

A. *Supported Instruction Set Summary*

Table I summarizes the subset of the RISC-V ISA implemented in this project.

TABLE I
Supported RISC-V Instructions

Type	Instruction	Format
R-type	`add`, `sub`, `and`, `or`, `slt`	R
I-type	`addi`	I
I-type	`lw`	I
S-type	`sw`	S
B-type	`beq`	B

B. *Datapath Design*

The datapath adheres to the standard MIPS/RISC-V five-stage pipeline. The complexity is primarily introduced by the feedback paths required for forwarding. Key components include the ALU, Register File, separate Instruction and Data Memories, and the five inter-stage pipeline registers. The Register File is designed to allow a write in the first half of the clock cycle (WB stage) and a read in the second half (ID stage) to prevent specific RAW hazards.

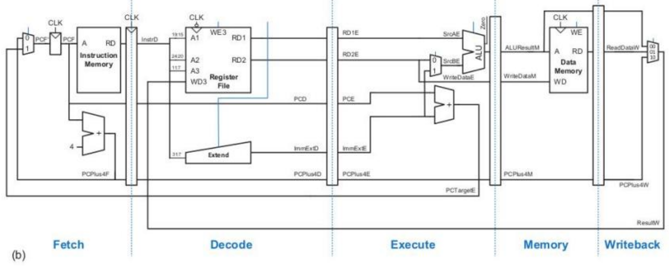

Fig. 1. Five-Stage Pipelined Datapath implementation.

C. *Control Unit Design*

The control unit is centralized and generates all necessary control signals:

- **Main Control:** Generates basic signals (`RegWrite`, `MemRead`, `MemWrite`, `ALUSrc`, `Branch`) based on the instruction type in the ID stage.
- **Hazard Unit:** Generates the **Stall/Bubble** signal (for load-use hazards) and the **Flush** signal (for taken branches).
- **Forwarding Unit:** Generates 2-bit multiplexer control signals (`ForwardA` and `ForwardB`) in the EX stage, directing data from the EX/MEM or MEM/WB registers to the ALU inputs when a hazard is detected.

D. *Implementation Platform*

The processor is modeled using **Verilog HDL**. Functional simulation and verification were performed using the **ModelSim** or **Vivado Simulator** environment [5].

IV. Results and Analysis

A. *Verification and Testing*

The processor's functional correctness was established using directed test programs designed to stress the hazard resolution logic. Table II outlines the specific instruction sequence used to demonstrate hazard handling.

TABLE II
Verification Test Program Demonstrating Hazards

Instruction	Purpose
`addi x7, x0, 2`	Initialize register x7
`addi x6, x0, 0x123`	Load base memory address
`sw x7, 4(x6)`	Store value to memory
`lw x8, 4(x6)`	Load value from memory
`add x9, x8, x7`	**Load-Use Hazard** (Requires stall)
`addi x4, x0, 10`	**Data Hazard** (Resolved by forwarding)
`srli x9, x4, 1`	Shift Right Logical Immediate

The design is verified using test programs that include arithmetic operations, load/store instructions, and branch instructions. Special emphasis is placed on programs with data dependencies to validate the forwarding and stalling mechanisms.

B. *Simulation Analysis*

Simulation waveforms confirm that:

1) Forwarding correctly resolves most RAW data dependencies.
2) The detection unit correctly inserts a single-cycle stall (bubble) for the critical Load-Use Hazard.
3) Branch instructions are evaluated in the EX-stage, and the IF/ID register is flushed upon a taken branch, minimizing the control hazard penalty.

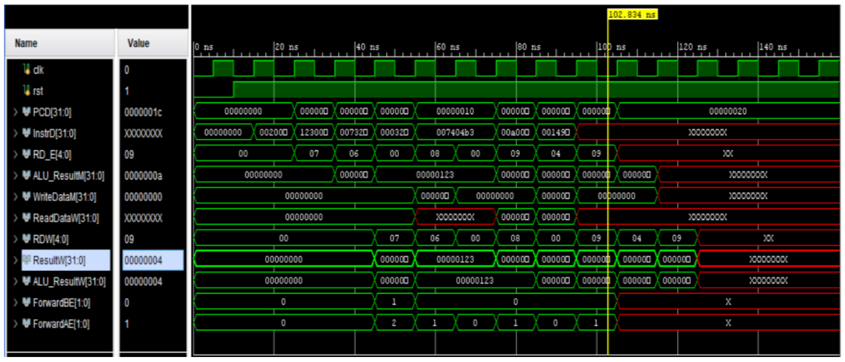

Fig. 2. Simulation Waveforms Demonstrating Hazard Resolution.

V. Conclusion

This project successfully achieved the design and functional implementation of a five-stage pipelined RISC-V processor featuring robust hazard handling. The core objectives of enhancing instruction throughput while ensuring correct program execution were met through the systematic application of data forwarding, hazard detection, and pipeline flushing techniques. The project illustrates the necessary architectural trade-offs required to manage dependencies in pipelined processor design, laying a strong foundation for understanding more complex, high-performance architectures.

VI. Future Work

Several enhancements can significantly improve the performance and capability of this processor:

- **Branch Prediction:** Implementation of an advanced branch prediction scheme (e.g., two-bit saturating counter predictor) to reduce the branch penalty.
- **Memory Hierarchy:** Integration of instruction and data caches (L1) to address memory access latency.
- **ISA Extension:** Expanding support for the full RV32I base integer instruction set, as well as the 'M' extension (multiplication and division).
- **Exception Handling:** Adding comprehensive support for exceptions and interruptions.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, RISC-V Edition. Morgan Kaufmann, 2017.

[2] RISC-V Foundation, "The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA," Version 2.2, 2017.

[3] D. Money Harris and S. L. Harris, *Digital Design and Computer Architecture*, RISC-V Edition. Morgan Kaufmann, 2021.

[4] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Addison-Wesley, 2011.

[5] T. R. Padmanabhan and B. B. Bala, *Design Through Verilog HDL*. IEEE Press, 2003.