RVX10-Pipeline: A Modular Five-Stage RISC-V Processor

Shreyas Sagar
Roll No: ECE - 230102118

Course: CS322M – Digital Logic and Computer Architecture
Indian Institute of Technology Guwahati

Supervisor: Dr. Satyajit Das

October 27, 2025

Contents

Abstract

This document presents a reworded and LaTeX-formatted version of the original project report describing a five-stage pipelined RISC-V core extended with custom ALU instructions (RVX10). The aim here is to preserve the organizational structure and the technical code excerpts while rewriting descriptive text to be original. The design covers pipeline stages (IF, ID, EX, MEM, WB), control pipeline, forwarding and hazard logic, memory models, and testbench guidance.

Chapter 1

Project Overview

This project implements a 32-bit RISC-V core using the classic five-stage pipeline organization. The core implements the RV32I base instruction set and adds a set of ten custom ALU operations grouped as RVX10. The five pipeline stages are:

- **IF (Instruction Fetch):** Fetch instruction from instruction memory and update the program counter.
- **ID (Instruction Decode):** Decode the instruction, read register file operands, and generate immediate values.
- **EX (Execute):** Perform ALU operations, calculate branch target, and evaluate branch conditions.
- **MEM (Memory):** Access data memory for loads and stores.
- **WB (Write-Back):** Commit results back to the register file.

Important building blocks include the pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB), a pipelined controller (main decoder + ALU decoder), an ALU supporting both base and custom operations, forwarding and hazard units to handle data/control hazards, and simple memory models for simulation.

Chapter 2

Top-Level Modules

2.1 top.sv

The top-level module integrates the CPU core with instruction and data memory models for simulation. It also exposes a few signals to the testbench so the bench can detect a prearranged success condition (for example, a particular memory write to a known address).

Key ports are clock and reset, and monitoring outputs like memory address, write-data, and write-enable signals used by the test harness.

2.2 riscvpipeline.sv

This file instantiates the datapath, the control pipeline, the forwarding unit, and the hazard detection logic. It also optionally includes simple performance counters (cycle and retired-instruction counters) to estimate CPI for a run.

Signals crossing module boundaries include control bundles (ALUSrc, PCSrc, ResultSrc, RegWrite), forwarding selects, stalls and flushes, and register identifiers passed between pipeline registers.

Chapter 3

Control Path Modules

3.1 controller.sv

The controller performs instruction decoding in the ID stage, produces control fields, and forwards them through pipeline control registers to later stages. It is split into a top-level controller that wires together two decoders: *maindec* (opcode-level policy) and *aludec* (fine ALU control).

The controller also computes a signal used to decide the next PC: typically `PCSrc_E = Branch_E && Zero_E` ORed with a jump indicator. Branch resolution happens in the EX stage.

3.1.1 maindec

The main decoder maps opcodes to a bundle of control outputs such as RegWrite, MemWrite, ALUSrc, ResultSrc (ALU vs Memory vs PC+4), ImmSrc, and an ALUOp field that hints the ALU decoder about the instruction group.

Common mapping examples:

- Load (opcode `0000011`): Set RegWrite, choose I-type immediate, and select memory result as writeback.
- Store (opcode `0100011`): Disable RegWrite and enable memory write.
- Register-register arithmetic (opcode `0110011`): Route to R-type ALU decoding.
- RVX10 group (e.g., opcode `0001011`): Select a custom ALUOp to let *aludec* decode extended instructions.

3.1.2 aludec

The ALU decoder translates ALUOp plus the instruction's `funct3` and `funct7` into a compact ALU control code. Unlike a base RV32I decoder, this design uses a 5-bit ALU control field to accommodate the custom RVX10 operations.

Chapter 4

Datapath and Pipeline Registers

4.1 datapath.sv

The datapath implements the datapath flow across pipeline stages:

- PC register and next-PC selection (PC+4 or branch target).
- IF/ID and ID/EX pipeline registers with enable (for stalls) and flush.
- Register file reads in ID and synchronous write in WB.
- Immediate generation unit for I/S/B/J types.
- Operand selection for the ALU, including forwarding multiplexers.
- EX/MEM and MEM/WB registers to carry ALU results and loaded data.

Forwarding is implemented using small 3-input multiplexers for each ALU input. These select between the value coming from ID/EX (no-forward), EX/MEM (alu result), or MEM/WB (writeback value) depending on the forwarding unit's outputs.

4.2 Pipeline register semantics

Each pipeline register supports reset, stall (enable), and flush control. On a control hazard or explicit flush request the register contents are replaced with NOP-equivalent values. For stalls the registers hold their previous contents, which freezes the pipeline at that stage.

Chapter 5

ALU, Register File, and Utilities

5.1 alu.sv

The ALU uses a 5-bit control code to select from a broad set of operations: standard arithmetic/-logical operations and a selection of RVX10 custom functions such as MIN/MAX, rotates, and bitwise-negated operations. The ALU also produces a Zero flag used for branch decisions.

A representative ALUControl mapping used in this implementation is:

00000	ADD
00001	SUB
00010	AND
00011	OR
00100	XOR
00101	SLT
00110	SLL
00111	SRL
01000	ANDN
01001	ORN
01010	XNOR
01011	MIN (signed)
01100	MAX (signed)
01101	MINU (unsigned)
01110	MAXU (unsigned)
01111	ROL
10000	ROR
10001	ABS (signed)

Design notes: shifts use the lower 5 bits of the second operand as the shift amount. Signed comparisons and min/max use two's complement interpretation when required.

5.2 regfile.sv and extend.sv

The register file implements two read ports (combinational) and one synchronous write port. Register x0 is hardwired to zero. The extension unit builds sign-extended immediates for I, S, B, and J formats based on a small selector signal (ImmSrc).

Chapter 6

Hazard and Forwarding Units

6.1 forwarding unit.sv

To avoid stalls for typical data dependencies, the forwarding unit monitors destination registers in the MEM and WB stages and, if they match EX stage source registers, it instructs the datapath to select a forwarded value. Priority is given to the value in MEM stage since it is the most recent.

Below is an exact excerpt from the original code implementing forwarding selection:

Listing 6.1: Forwarding unit (excerpt)

```
always_comb begin
  FwdSel_A = 2'b00; FwdSel_B = 2'b00;
  // Forward from WB if applicable
  if (RegWrite_W && (rd_W != 5'b0)) begin
    if (rd_W == rs1_E) FwdSel_A = 2'b01;
    if (rd_W == rs2_E) FwdSel_B = 2'b01;
  end
  // Forward from MEM if applicable (higher priority)
  if (RegWrite_M && (rd_M != 5'b0)) begin
    if (rd_M == rs1_E) FwdSel_A = 2'b10;
    if (rd_M == rs2_E) FwdSel_B = 2'b10;
  end
end
```

6.2 hazard unit.sv

The hazard unit detects load-to-use situations and control-flow changes that require pipeline stalls or flushes. A common pattern is that if the instruction in EX is a load and its destination matches an ID-stage source, the unit stalls IF and ID and injects a bubble into EX by flushing ID/EX.

The following code excerpt shows the central logic used to detect the load-use hazard and to generate stall/flush outputs (verbatim):

Listing 6.2: Load-use hazard detection (excerpt)

```
assign load_use_hazard = ResultSrc_E_0 & (rd_E != 5'b0)
  & ((rd_E == rs1_D) | (rd_E == rs2_D)
  );
assign stall_F = load_use_hazard;
```

```
5 assign stall_D = load_use_hazard;
6 assign flush_E = load_use_hazard | PCSrc_E;
7 assign flush_D = PCSrc_E;
```

Chapter 7

Memory Modules

7.1 imem.sv

Instruction memory is modeled as a simple word-indexed array initialized from an external file (read using `$readmemh` in simulation). Addressing uses word addressing (a[31:2]) to index rows.

7.2 dmem.sv

Data memory is modeled as a synchronous write, combinational read RAM. On a rising clock edge, if write-enable is asserted, the write data is stored. For simulation purposes the read port is kept combinational for simplicity.

Chapter 8

Auxiliary Modules and Utilities

Small utility modules used throughout the design include parameterized multiplexers (2-input and 3-input), a flop with enable (used for PC to support stalls), and simple adder modules to compute PC+4 and branch targets.

Chapter 9

Testbench and Simulation

9.1 tb_pipeline.sv

The testbench toggles the clock and reset, loads memory images into imem and runs the core until a known success pattern is observed. The success criterion used in the original testbench is writing a known value (e.g., 25) to a predetermined address (e.g., 100) which the test harness watches for.

9.2 Waveform checklist

When verifying the design with a waveform viewer, check for the following:

- PC progression (PC+4) and points where it is held due to stalls.
- Proper insertion of NOPs on flush events (branch taken or misfetch).
- Forwarding mux outputs toggle appropriately for back-to-back dependent instructions.
- The final memory write used by the testbench to detect success.

Chapter 10

Performance and Verification Notes

Basic performance counters track cycles and retired instructions, enabling a simple CPI computation (cycles / retired-instructions). When adding new instructions or tweaks, ensure the ALU encoding, control pipeline, and forwarding paths remain consistent.

A short verification checklist:

- Ensure register x0 remains constant zero and is never overwritten.
- Confirm ALU control encodings match both the decoder and the ALU.
- Validate ResultSrc propagation across pipeline control registers.
- Exercise load-use and control hazards to verify stall/flush behavior.

Chapter 11

Code Highlights

This chapter collects a few annotated examples for clarity. For the complete source, consult the project repository or the original source files.

Chapter 12

Conclusions and Suggested Extensions

The pipelined core demonstrates standard techniques for building a compact RISC-V processor with a small set of ISA extensions. Reasonable next steps include adding a simple branch predictor, exception support, compressed instructions support, or more detailed performance breaking-down of stalls and bubbles.

References

- David Harris and Sarah Harris, *Digital Design and Computer Architecture (RISC-V Edition).*
- Course materials: CS322M – Digital Logic and Computer Architecture, Indian Institute of Technology Guwahati.

This LaTeX file was prepared by paraphrasing the original project report to produce unique descriptive text while preserving technical structure and selected code excerpts.