RVX10-P: Five-Stage Pipelined RISC-V Core

Detailed Code Explanation and Module Walkthrough

Course: CS322M – Digital Logic and Computer Architecture

Indian Institute of Technology, Guwahati

Submitted by:
Shreyas Sagar
Roll No: ECE - 230102118

Under the guidance of: Dr. Satyajit Das

October 27, 2025

Contents

Abstract

This report explains the SystemVerilog implementation of the RVX10-P five-stage pipelined RISC-V core. The goal is to present a clear, module-by-module description of how each file in the project implements parts of the processor: top-level glue, pipelined datapath, control, ALU with RVX10 extensions, forwarding, hazard detection, pipeline registers, memory, and register file. Each section describes the module's port interface, internal behavior, and how it integrates into the overall pipeline.

Chapter 1

Project Overview

RVX10-P is a five-stage pipelined implementation of an RV32I core extended with ten custom ALU operations (RVX10). The five classic stages are:

- IF — Instruction Fetch
- ID — Instruction Decode / Register Read
- EX — Execute (ALU, branch decision)
- MEM — Data Memory access
- WB — Write Back to register file

Key functional blocks:

- Pipeline registers: IF/ID, ID/EX, EX/MEM, MEM/WB
- Control path: main decoder (maindec), ALU decoder (aludec), controller pipeline registers (ctrl_ID_EX, ctrl_EX_MEM, ctrl_MEM_WB)
- Datapath: datapath.sv (PC, ALU, MUXes, register file integration)
- ALU: supports RV32I operations + RVX10 custom ops
- Hazard management: forwarding_unit, hazard_unit
- Memories: imem (instruction memory), dmem (data memory)

Chapter 2

Top-Level Modules

2.1 top.sv

Purpose: Integrates the processor core with instruction memory and data memory for simulation / testbench. Also exposes monitoring ports used by the testbench to detect pass/fail conditions.

Important ports:

- Inputs: `clk`, `reset`
- Outputs for TB: `WriteData`, `DataAdr`, `MemWrite`
- Wires connecting core ↔ memory: `PC_F`, `Instr_F`, `MemReadData_M`, etc.

Behavior: Instantiates:

- `riscvpipeline` (top-level core)
- `imem` (instruction memory)
- `dmem` (data memory)

The data memory write port is monitored by the testbench — when the known success pattern is stored to a predetermined address (address 100), the testbench reports success.

Notes for students:

- Keep `imem` and `dmem` file paths correct so the simulator can load the memory image (e.g., using `$readmemh`).
- The PC is word aligned: `imem` uses `a[31:2]` indexing.

2.2 riscvpipeline.sv

Purpose: Top-level module for the pipelined core. Connects datapath, controller, forwarding unit, and hazard detection unit. Also contains optional performance counters.

Key signals between modules:

- Control signals: `ResultSrc_W`, `PCSrc_E`, `ALUSrc_E`, `RegWrite_W`, `ImmSrc_D`, `ALUControl_E`
- Hazard signals: `stall_F`, `stall_D`, `flush_D`, `flush_E`, `FwdSel_A`, `FwdSel_B`
- Datapath feedback: `Instr_D`, `rs1_D`, `rs2_D`, `rs1_E`, `rs2_E`, `rd_E`, `rd_M`, `rd_W`

Instantiated submodules:

1. `datapath` — the entire pipelined datapath (IF–WB)
2. `controller` — control signal generator with pipeline registers
3. `forwarding_unit` — decides if ALU inputs in EX should be forwarded
4. `hazard_unit` — detects load-use hazards and handles stalls/flushes

Performance counters: Two counters `total_cycles` and `retired_instructions` count cycles and the number of instructions retired (uses `RegWrite_W` to approximate retirement). CPI can be computed from these.

Integration notes:

- `PCSrc_E` is computed in the controller using EX-stage signals (branch/jump resolved in EX).
- The hazard unit receives `ResultSrc_E_0` (low-level indicator that EX is a load instruction) to detect load-use.

Chapter 3

Control Path Modules

3.1 controller.sv

Purpose: Implements pipelined control: decodes instructions in ID stage and pipelines control signals to EX, MEM, and WB stages. Also computes primary signals like `PCSrc_E`, `ALUSrc_E`, `ALUControl_E`.

Structure:

- `maindec` (main decoder) — provides high-level control fields: `ResultSrc`, `MemWrite`, `Branch`, `ALUSrc`, `RegWrite`, `Jump`, `ImmSrc`, `ALUOp`.
- `aludec` (ALU decoder) — translates ALUOp, funct3, and funct7 (including RVX10 funct7) to a 5-bit `ALUControl` signal.
- Pipeline control registers:
  - `ctrl_ID_EX` — latches control from ID to EX (supports flush).
  - `ctrl_EX_MEM` — latches control from EX to MEM.
  - `ctrl_MEM_WB` — latches control from MEM to WB.

Notes:

- The controller outputs `PCSrc_E = (Branch_E & Zero_E) | Jump_E`. Branch decision is taken in EX stage using ALU zero flag.
- The ALU decode is extended for RVX10 operations: `ALUControl` is 5 bits, giving room to encode custom operations.

3.1.1 maindec

Function: For each opcode field, sets a bundle of controls. Example encodings:

- `0000011` (lw): RegWrite=1, ImmSrc=I-type, ALUSrc=1, ResultSrc=MEM
- `0100011` (sw): RegWrite=0, MemWrite=1, ALUSrc=1
- `0110011` (R-type): ALUOp selects R-type decode
- `0001011` (RVX10): custom opcode case added — ALUOp selects RVX10 group

3.1.2 aludec

Function: Produces a 5-bit ALUControl. Logic:

- ALUOp=00: addition
- ALUOp=01: subtraction
- ALUOp=10: R/I-type decoding by funct3/funct7 bits (regular RV32I)
- ALUOp=11: RVX10 custom group — use full funct7 and funct3 to select operations

Chapter 4

Datapath and Pipeline Registers

4.1 datapath.sv

Purpose: Implements the pipeline datapath: PC logic, instruction fetch and pipeline registers, register file reads, ALU inputs (with forwarding), branch target calculation, EX/MEM and MEM/WB pipeline registers, and the writeback multiplexer.

Key datapath elements:

- **PC management:** `pc_register` (flopren) holds PC. A mux selects between PC+4 and branch target `pc_target_E` based on `PCSrc_E`.
- **IF/ID register:** `pipe_IF_ID` saves fetched instruction and PC-related signals; supports stall (enable) and flush.
- **Register file:** Combinational reads in ID stage, synchronous write in WB stage (`RegWrite_W` + `rd_W` + data `R_W`).
- **Extend unit:** `extend.sv` computes immediates (I, S, B, J types) based on `ImmSrc_D`.
- **ID/EX register:** `pipe_ID_EX` passes operands, immediates, PC, and register addresses to EX stage; flushable to insert NOPs when needed.
- **Forwarding muxes:** Two 3-input muxes (`mux3`) select ALU operands from ID/EX, MEM/WB (R_W), or EX/MEM (ALUResultM) depending on forwarding unit outputs (`FwdSel_A`, `FwdSel_B`).
- **ALU:** `alu.sv` performs arithmetic/logic based on 5-bit `ALUControl_E` and sets `Zero_E`.

- **EX/MEM register:** `pipe_EX_MEM` holds ALU result, WriteData (for stores), PCPlus4, RdE -¿ RdM.
- **MEM/WB register:** `pipe_MEM_WB` holds memory read data, ALU result, PC+4, RdM -¿ RdW.
- **Writeback mux:** `mux3` at WB stage selects between ALU result, memory read data, or PC+4 (for JAL).

Important signals mapping:

- `MemAddr_M` serves as ALUResultM output to data memory.
- `MemWriteData_M` is the forwarded register value for store.
- `R_W` is the final writeback data selected by `ResultSrc_W`.

4.2 Pipeline Register Modules

Your code implements pipeline registers carefully with reset, enable (stall), and flush options:

4.2.1 pipe_IF_ID

Ports: `enable`, `flush`, `InstrF`, `PCF`, `PCPlus4F` $\rightarrow$ `InstrD`, `PCD`, `PCPlus4D`.
Behavior:

- On reset: clears registers.
- If `enable==0`: holds previous values (stall).
- If `flush==1`: writes a NOP instruction (e.g., `32'h00000033`) into `InstrD` and zeroes PC fields.
- Else: latches the fetched instruction and PC values.

4.2.2 pipe_ID_EX

Ports: Latches RD1D, RD2D, PCD, ImmExtD, PCPlus4D, Rs1D, Rs2D, RdD into EX-side signals. Supports `flush` to insert NOPs.

4.2.3 pipe_EX_MEM

and `pipe_MEM_WB` **Function:** Standard registers capturing ALU result, write data, PC+4, Rd for next stage; ensure synchronous transfer on clock edges and reset behavior.

Implementation note: All these registers use `always_ff` with asynchronous reset to ensure deterministic state after reset.

Chapter 5

ALU, Register File, and Supporting Modules

5.1 alu.sv

Purpose: Implements arithmetic and logic operations for both base RV32I and RVX10 custom operations. ALU control is a 5-bit code; the ALU uses this to choose an operation.

Supported operations (by ALUControl code):

- `00000` — ADD
- `00001` — SUB
- `00010` — AND
- `00011` — OR
- `00100` — XOR
- `00101` — SLT
- `00110` — SLL
- `00111` — SRL
- `01000` — ANDN (a & b)
- `01001` — ORN (a — b)
- `01010` — XNOR ((a ^{b}))
- `01011` — MIN (signed)

- `01100` — MAX (signed)
- `01101` — MINU (unsigned)
- `01110` — MAXU (unsigned)
- `01111` — ROL (rotate left)
- `10000` — ROR (rotate right)
- `10001` — ABS (absolute value signed)

Implementation details:

- The ALU computes `condinvb = alucontrol[0] ?  b :  b` and `sum = a + condinvb + alucontrol[0]`, enabling reuse of an add/sub shape.
- For shift/rotate, the ALU uses `b[4:0]` as shift amount (`shamt`).
- Zero flag is asserted if the result equals zero; used for branch decisions (e.g., BEQ).
- Handles signed comparisons by casting to `signed` integers where required (MIN, MAX, ABS).

Caveat: Overflow logic (`v`) is computed but not used for trap — consistent with RV32I behavior in this design.

5.2 regfile.sv

Purpose: Implements a 32x32 register file with two asynchronous read ports and one synchronous write port.

Behavior:

- Reads (rd1, rd2) are combinational and return 0 for address 0 (x0 must be hardwired to 0).
- Write occurs on rising clock edge if `we3` is asserted, writing `wd3` into `rf[a3]`.

Notes: Ensure that forwarding logic prevents hazards for reads needed in the immediate next cycle after a write (if write-back happens later).

5.2.1 extend.sv

Purpose: Sign/zero extension of immediate fields based on `ImmSrc`.

Behavior:

- `00` — I-type: bits [31:20]
- `01` — S-type: bits [31:25] + bits [11:7]
- `10` — B-type: branch immediate arrangement with LSB zero
- `11` — J-type: jal immediate arrangement

Chapter 6

Hazard and Forwarding Units

6.1 forwarding_unit.sv

Purpose: Resolve data hazards by selecting the correct source for ALU inputs in the EX stage from:

- ID/EX (no forwarding)
- MEM/WB (writeback result)
- EX/MEM (ALU result currently in MEM stage)

Output selects:

- `FwdSel_A` — select for ALU operand A
- `FwdSel_B` — select for ALU operand B

Priority: MEM forwarding takes precedence over WB forwarding (implemented by checking WB first and then overwriting with MEM check). This ensures the most recent available result is used.

Key conditions:

- If `RegWrite_M rd_M != 0 rd_M == rs1_E` ⇒ forward from MEM to A
- Similar checks for B and for WB forwarding.

6.2 hazard_unit.sv

Purpose: Detect load-use hazards and control hazards and generate stall/flush signals.

Inputs: `rs1_D`, `rs2_D`, `rd_E`, `ResultSrc_E_0`, `PCSrc_E`
Outputs: `stall_F`, `stall_D`, `flush_D`, `flush_E`
Logic:

- **Load-use hazard:** If EX-stage instruction is a load (`ResultSrc_E_0`) and `rd_E == rs1_D || rd_E == rs2_D` then: stall IF and ID (freeze PC and IF/ID), and flush ID/EX (insert NOP in EX). This creates a one-cycle pipeline bubble.
- **Control hazard:** If branch/jump is taken (`PCSrc_E==1`), then flush IF/ID and ID/EX (since the next instructions fetched are wrong).

Effect: Prevents use of stale data and ensures correct program flow on branches.

Chapter 7

Memory Modules

7.1 imem.sv

Purpose: Simple word-addressable instruction memory. The module instantiates a small RAM array and uses `$readmemh` to initialize content from a '.mem' file.

Important: Address indexing uses `a[31:2]` because memory is word-aligned.

7.2 dmem.sv

Purpose: Simple synchronous data memory (RAM). On rising clock edge, when `we` is asserted, the memory writes `wd` into RAM[`a[31:2]`]; read is combinational output of RAM addressed by `a[31:2]`.

Simulation note: In actual hardware memory writes/reads may have timing differences; for simulation this simple RAM model is sufficient.

Chapter 8

Auxiliary Modules and Utilities

8.1 mux2, mux3

Simple parameterized multiplexers used widely across datapath:

- `mux2`: width param, selects between two inputs using single-bit select.
- `mux3`: 3-input selector using 2-bit select (implemented as nested selection).

8.2 flopren

A parameterized register with asynchronous reset and synchronous enable — used for PC register (`pc_register`) to support stalls by gating enable.

8.3 adder

Simple 32-bit adder module used for PC+4 and PC+imm computations.

Chapter 9

Testbench and Simulation

9.1 Testbench (tb_pipeline.sv)

Role: Drives clock and reset, instantiates `top.sv`, and monitors the memory write-back. A passing test writes 25 into memory location 100 and asserts `MemWrite` when this happens.

Recommended checks:

- After simulation completes, assert that `MemWrite==1`, `DataAdr==100`, and `WriteData==25`.
- Dump waveform (`.vcd`) for inspection in GTKWave to check pipeline overlaps, stalls, forwarding signals.

9.2 Waveform checks (suggested)

Open the waveform and verify:

- PC increments by 4 when not stalled; check when stalls occur PC holds its value.
- IF/ID and ID/EX registers are flushed to NOP on branch/jump or load-use hazard clique.
- Forwarding mux selects (FwdSel_A/B) toggle when dependent instructions are back-to-back.
- MemWrite and DataAdr show the final store of 25 at address 100.

Chapter 10

Performance and Observations

10.1 Performance counters

Your design includes cycle and instruction counters:

- `total_cycles` incremented every clock
- `retired_instructions` incremented when `RegWrite_W` asserted

Compute CPI = `total_cycles / retired_instructions`. Example measured CPI for sample workload was 1.28 (improved from single-cycle).

10.2 Common pitfalls and verification checklist

- Ensure x0 is never overwritten — regfile returns zero for address 0 and forwarding must not attempt to write to x0.
- Confirm ALU encodings in `aludec` match the ALU's implemented control table.
- Verify `ResultSrc` encodings for load, ALU, and JAL are correctly routed through control pipeline registers to select writeback data.
- Ensure `pipe_IF_ID` and `pipe_ID_EX` flush/enable semantics are correct — a stale enable/flush ordering can cause wrong instructions to execute.
- When adding new custom instructions, maintain consistent funct7/funct3 encoding between assembler (memory image) and `aludec` matching the ALU.

Chapter 11

Annotated Example — Key Code Snippets

Below are short annotated excerpts illustrating key behaviors. (For full code, refer to the `src/` folder.)

Forwarding Unit (annotated)

Listing 11.1: forwarding_unit.sv (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
```

Explanation: Default no-forward. If register written in WB stage matches EX stage source, forward from WB (01). If MEM stage has the destination, overwrite with MEM forward (10) — MEM is newer so higher priority.

Load-Use Hazard (annotated)

Listing 11.2: hazard_unit.sv (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;
assign stall_D = load_use_hazard;
assign flush_E = load_use_hazard | PCSrc_E;
assign flush_D = PCSrc_E;
```

Explanation: If EX-stage is a load and its rd is needed by ID-stage sources, stall IF ID to prevent incorrect operand read; flush ID/EX to insert bubble into EX.

Chapter 12

Conclusion and Extensions

This document explained the entire codebase module-by-module, showing how the pipeline is structured, how hazards are handled, and how forwarding and control flow are implemented. The implementation achieves correct behavior for the RV32I base plus RVX10 custom ALU operations.

12.1 Possible extensions

- Add full branch prediction (e.g., 1-bit/2-bit predictor) to reduce branch flushes.
- Add exceptions/interrupt handling.
- Add support for compressed instructions or additional ISA extensions.
- Implement cycle-accurate performance counters that report CPI per program and per-stall breakdown.

References

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