Verification Plan for RISC-V ALU and Execute Stage

Shengjie Chen

1 Verification Scope and Objectives

This verification plan targets two cascaded modules in the RISC-V pipeline:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic, logic, shift, and comparison operations based on the 4-bit control signal.
- **Execute Stage:** Integrates the ALU, operand forwarding logic, and branch handling. It computes results and forwarding data for subsequent pipeline stages.

Objectives:

1. Verify functional correctness of ALU operations and flag generation (zero, overflow).
2. Verify operand selection and forwarding paths within the execute stage.
3. Ensure correct generation of control signals (`pc_src`, `jalr_flag`, etc.).
4. Achieve functional and code coverage closure with UVM-based verification.

2 Design Interface Summary

ALU

Table 1: Interface Summary of the ALU

Signal	Direction	Description
`control[3:0]`	Input	Operation select code controlling the arithmetic, logical, shift, or comparison behavior of the ALU.
`left_operand[31:0]`	Input	Left-hand operand used as the first input to the operation.
`right_operand[31:0]`	Input	Right-hand operand or shift amount, depending on operation type.
`result[31:0]`	Output	Computed result of the selected ALU operation.
`zero_flag`	Output	Asserted high when `result` equals zero; used for branch decision logic.
`overflow`	Output	Indicates a signed arithmetic overflow during `ADD` or `SUB` operations.

Execute Stage

Table 2: Interface Summary of the Execute Stage

Signal	Direction	Description
`data1`, `data2`	Input	Source register operands from the decode stage.
`immediate_data`	Input	Immediate value decoded from instruction.
`pc_in`	Input	Current program counter value entering the execute stage.
`control_in`	Input	Control structure containing ALU operation type, instruction encoding, and pipeline control signals.
`wb_forward_data`	Input	Data forwarded from the write-back stage for hazard resolution.
`mem_forward_data`	Input	Data forwarded from the memory stage for hazard resolution.
`forward_rs1`, `forward_rs2`	Input	Multiplexer selectors determining which data source to forward for operands 1 and 2.
`control_out`	Output	Propagated control signals to the next pipeline stage.
`alu_data`	Output	Final ALU computation result.
`memory_data`	Output	Data prepared for memory write operations (valid for S-type instructions).
`pc_src`	Output	Branch decision flag indicating whether the next PC should be redirected.
`jalr_target_offset`	Output	Calculated target address offset for JALR instructions.
`jalr_flag`	Output	Indicates that a JALR-type instruction is being executed.
`pc_out`	Output	Program counter value passed to the next stage.
`overflow`	Output	Overflow indicator propagated from the ALU operation.

3 Verification Environment Overview

3.1 Testbench Architecture

A UVM-based testbench will be built using Easier UVM generator, including:

- **Interface:** Connects testbench to DUT (ALU or execute_stage).
- **Driver:** Drives randomized transaction stimuli.

- **Monitor:** Captures DUT outputs for checking and coverage.
- **Scoreboard:** Compares DUT outputs with reference model.
- **sequence item:** The data generated from the randomization.
- **Sequence/Sequencer:** Generates constrained random operations and operand patterns.
- **Coverage Collector:** Records coverage metrics.

4 Functional Verification Plan

4.1 ALU Verification Items

Operation Coverage:

Category	Operation	Verification Goal
Logic	AND, OR, XOR	Verify bitwise correctness.
Arithmetic	ADD, SUB	Verify signed arithmetic and overflow.
Comparison	SLT, SLTU	Verify signed/unsigned comparison correctness.
Shift	SLL, SRL, SRA	Verify correct shift amount and direction.
Immediate	LUI	Verify correct upper-immediate load.
Branch	Bxx (BEQ, BNE, etc.)	Verify zero-flag generation correctness.

Flag Verification:

- `zero_flag` = 1 iff result == 0.
- `overflow` = 1 iff ADD/SUB crosses signed boundary.

4.2 Execute Stage Verification Items

Forwarding Logic:

- Verify correct selection from `data1`, `data2`, `mem_forward_data`, and `wb_forward_data`.
- Check data hazards resolution in cases FORWARD_FROM_EX and FORWARD_FROM_MEM.

Control and Branch:

- Verify `pc_src` asserted only for valid branch.
- Verify `jalr_flag` and `jalr_target_offset` for jump instructions.

ALU Integration:

- Cross-check ALU output consistency with Execute Stage's `alu_data`.
- Verify propagation of overflow and zero_flag signals.

5 Bug Detection Strategy

- Use **scoreboard** to compare DUT outputs with a SystemVerilog reference model.
- Add **assertions (SVA)**:
  - `assert property (result == 0 -> zero_flag == 1);`
  - `assert property (control inside {ADD,SUB} |-> overflow == expected);`
  - `assert property (forwarding path matches selected signal);`
- Use directed tests for corner cases (e.g., overflow, max/min shift).

6 Coverage Plan

6.1 Code Coverage

Collected automatically by simulator:

- Statement coverage
- Branch coverage
- Toggle coverage

6.2 Functional Coverage

A covergroup will be defined on the transaction level to track ALU operation types and key functional signals:

```
// Covergroup definition for ALU operations and overflow conditions
covergroup alu_cg @(posedge vif.clk);

  // Control operation type classification
  coverpoint tr.control {
    // Logical operations
    bins logic_ops[]  = {ALU_AND, ALU_OR, ALU_XOR};

    // Arithmetic operations with possible overflow
    bins arith_ops[]  = {ALU_ADD, ALU_SUB};

    // Shift operations
    bins shift_ops[]  = {ALU_SLL, ALU_SRL, ALU_SRA};

    // Comparison operations
    bins cmp_ops[]    = {ALU_SLT, ALU_SLTU};

    // Illegal or default operation (unexpected control)
    bins illegal_ops  = default;
  }
```

```
  // Capture overflow condition
  coverpoint tr.overflow;

  // Ensure overflow only occurs in ADD or SUB operations
  cross control, overflow;

endgroup
```

Execute Stage coverage will cross:

- Operand forwarding paths × control type. → *Ensures all instruction types are tested under every forwarding condition.*
- Branch decisions (*pc_src*) × zero_flag. → *Confirms branch logic behaves correctly for both taken and not-taken cases.*
- Instruction encoding types (*encoding_t*). → *Verifies that all instruction formats (R/I/S/B/U/J) are executed in simulation.*
- Immediate data extension cases. → *Ensures both sign-extension and zero-extension immediates are covered.*
- Control signal forwarding (*control_in* × *control_out*). → *Confirms control signals are properly propagated between pipeline stages.*
- JALR branch target computation ($jalr\_flag \times jalr\_target\_offset$). → *Validates that JALR-type branch targets are calculated correctly.*

7 Verification Closure Criteria

- All functional coverage points reached ($\geq 95\%$).
- Code coverage $\geq 90\%$.
- All assertions pass without failure.
- No mismatches reported by scoreboard.
- All directed corner cases verified.

8 Deliverables

- UVM environment source files.
- Verification plan and coverage report.
- Simulation waveforms for representative cases.
- Summary report of detected bugs and fixes.