UVM Based Verification for 5- Stage Pipelined RISC-V Processor

Project Report

February 2025

MOHAMED EHAB

Contents

Abstract

The objective of this project is to verify the functionality of a 5-stage pipelined RISC-V processor using Universal Verification Methodology (UVM). The verification focuses on functional correctness, identifying hazards, testing corner cases, and achieving comprehensive coverage. This report outlines the verification strategy, implementation details, and results obtained, including bugs found in the RTL.

Introduction

Pipelining is a powerful way to improve the throughput of a digital system. The pipelined RISC-V processor divides instruction execution into five stages: Fetch, Decode, Execute, Memory Access, and Writeback. By overlapping the execution of multiple instructions, the pipeline achieves higher throughput compared to a single-cycle processor, where only one instruction is executed at a time. This architectural advancement makes the pipelined RISC-V faster and more efficient for modern applications.

However, pipelining introduces challenges such as data hazards (when one instruction depends on the result of a previous instruction) and control hazards (occurring during branch instructions). These challenges necessitate robust handling mechanisms, such as forwarding, stalling, and branch prediction. Due to its performance and flexibility, the pipelined RISC-V processor is widely used in commercial applications.

The processor design used in this project supports 27 instructions, categorized as follows:

Instruction Type	Instruction
I-type	lw, addi, slli, slti, xori, srli, srai, ori, andi, jalr
R-type	add, sub, sll, slt, xor, srl, sra, or, and
B-type	beq, bne, bge, blt
S-type	sw
J-type	Jal

Test Plan

The verification plan targets the following functionalities and scenarios:

- Instruction Execution: Ensuring all RISC-V instructions are executed correctly.
- Pipeline Hazards: Verifying data and control hazard handling.
- Corner Cases: Testing edge conditions, such as:
    - All Ones: Validating the addition and subtraction of operands with all bits set to 1.
    - All Zeros: Testing addition and subtraction of operands with all bits set to 0.
    - Toggling Ones and Zeros: Ensuring correct behavior when alternating patterns (e.g., 101010... and 010101...) are used as operands.
- Instruction Transitions: Evaluating transitions between different instruction types, especially when crossing with hazards occurring.
- Memory and Register file Operations: Validating memory and register file read/write operations using UVM RAL.
- Internal Signals: Monitoring control signals for compliance with expected behavior.

The test plan includes directed and random tests, ensuring high coverage of both typical and edge cases. functional and code coverage are employed to validate critical design behaviors.

Methodology

Universal Verification Methodology (UVM) was chosen for its modularity, scalability, and reusability as it support OOP. Key benefits of UVM for this project include:

- Structured Testbench: Facilitates separation of stimulus generation, DUT interaction, and result checking.
- Randomization: Enables efficient testing of various scenarios, including corner cases.
- RAL Integration: Simplifies register-level operations for memory and register file Read/Write operation testing.
- Reusable Components: Reduces development time by reusing sequencers, monitors, and scoreboards.

The UVM environment consists of a testbench with components such as an agent, monitor, driver, sequencer, scoreboard, and coverage collector which will be discussed in details in the next section.

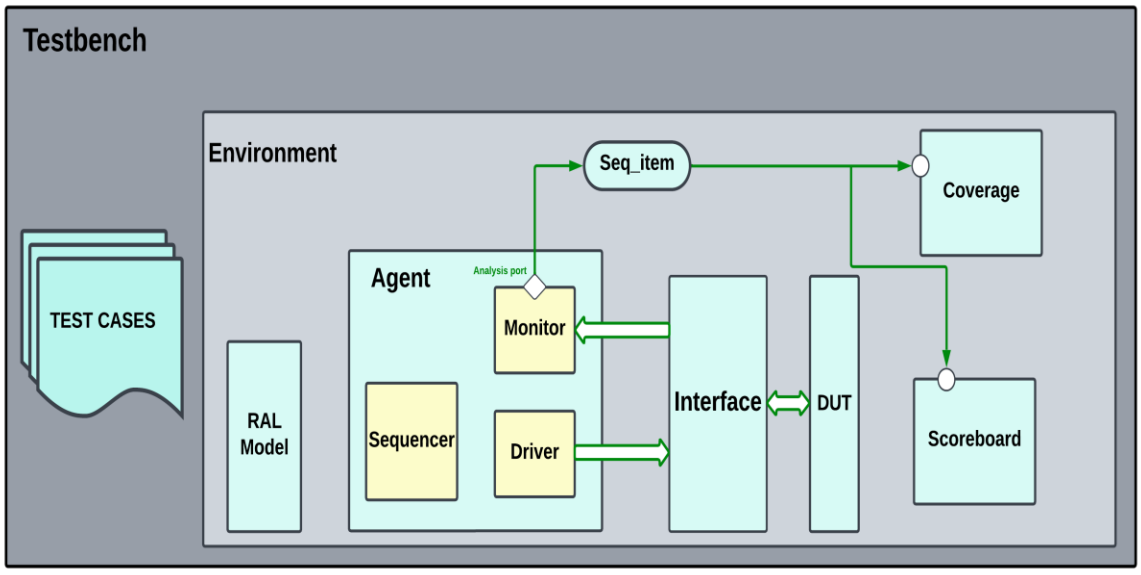
Implementation

- **Design**

The RTL design used in this project was obtained from a publicly available GitHub repository [Repo Link] This repository provided the baseline RTL, which was used as the DUT in the verification environment.

- **TestBench**

The UVM testbench was implemented to mimic the pipeline behavior of the RISC-V processor



fig(1) Testbench Block Diagram

- **Testbench Component**

1. Interface

The interface serves as a bridge between the testbench and the DUT, encapsulating all input and output signals. It abstracts the complexity of signal-level communication, ensuring a structured and efficient connection between testbench components and the DUT.

```
interface intf (input logic clk) ;

  import risc_pkg::* ;

  logic        reset      ;
  logic [31:0] InstrF     ;
  logic [31:0] PCF        ;
  logic [31:0] WriteDataM ;
  logic [31:0] DataAdrM   ;
  logic [31:0] ReadDataM  ;
  logic        MemWriteM  ;
  instr_type   inst_type  ;
```

2. Sequence Item

The sequence item defines the fundamental transaction structure for stimulus generation. It includes:

- Declaration of input and output signals for the DUT, including a 32-bit randomized instruction and auxiliary signals for hazard control.
- Basic constraints to ensure that only supported instruction formats are generated.
- Essential utility functions such as:
  - Reset function for initializing the transaction state.
  - Get_type function to determine the instruction type.
  - Extend function for extending 12-bit immediate

```
function [31:0] Extend ;

    case(InstrF[6:0])
      // I-type
      imm, lw, jalr : Extend = {{20{InstrF[31]}}, InstrF[31:20]};

      // S-type (stores)
      sw : Extend = {{20{InstrF[31]}}, InstrF[31:25], InstrF[11:7]};

      // B-type (branches)
      brnch : Extend = {{20{InstrF[31]}}, InstrF[7], InstrF[30:25], InstrF[11:8], 1'b0};

      // J-type (jal)
      jal : Extend = {{12{InstrF[31]}}, InstrF[19:12], InstrF[20], InstrF[30:21], 1'b0};

      // U-type
      lui, auipc : Extend = {InstrF[31:12], 12'b0};

      default: Extend = 32'bx; // undefined
  endcase
endfunction : Extend
```

3. Sequence

Sequences define the test scenarios and generate transaction items to stimulate the DUT. This includes:

- Basic random sequences for generalized instruction execution testing.
- Reset sequence to initialize the DUT state.
- Directed sequences for individual instruction types, ensuring deterministic verification of each operation.

```
class rand_seq extends uvm_sequence #(seq_item);
  `uvm_object_utils(rand_seq)

  seq_item rand_item ;

  function  new(string name = "rand_seq");
    super.new(name);
  endfunction: new

  task body() ;
    rand_item = seq_item::type_id::create("rand_item") ;
    start_item(rand_item) ;
    assert(rand_item.randomize() )
    else
    `uvm_error(get_type_name(),"randomization failed in rand_sequence")
    rand_item.Get_type ;
    finish_item(rand_item);
  endtask

endclass: rand_seq
```

4. Driver

The driver is responsible for delivering input transactions from the sequencer to the DUT via the interface. It operates on the positive edge of the clock, ensuring proper synchronization with the DUT's execution cycle.

```
// Run Phase
 task run_phase (uvm_phase phase);
   super.run_phase(phase);
        forever begin
          driver_item = seq_item::type_id::create("driver_item");
          seq_item_port.get_next_item(driver_item) ;
          drive(driver_item) ;
          seq_item_port.item_done() ;
        end
 endtask : run_phase


// drive task
     task drive (seq_item RISC_item) ;
       @(posedge driver_intf.clk) ;
        driver_intf.reset  <= RISC_item.reset  ;
        driver_intf.InstrF <= RISC_item.InstrF ;
        driver_intf.inst_type <= RISC_item.inst_type ;
     endtask : drive
```

5. Monitor

The monitor passively collects DUT output transactions through the interface, observing signal behavior without modifying it. Like the driver, it operates at the positive edge of the clock and forwards extracted transactions for further processing.

```
// Run Phase
task  run_phase(uvm_phase phase);
  super.run_phase(phase);

  forever begin
       monitor_item = seq_item::type_id::create("monitor_item");
       @(posedge monitor_intf.clk) ;
       monitor_item.reset      = monitor_intf.reset      ;
       monitor_item.InstrF     = monitor_intf.InstrF     ;
       monitor_item.inst_type  = monitor_intf.inst_type  ;
       monitor_item.PCF        = monitor_intf.PCF        ;
       monitor_item.WriteDataM = monitor_intf.WriteDataM ;
       monitor_item.DataAdrM   = monitor_intf.DataAdrM   ;
       monitor_item.MemWriteM  = monitor_intf.MemWriteM  ;
       monitor_item.ReadDataM  = monitor_intf.ReadDataM  ;
    #1 ;
       monitor_ap.write(monitor_item) ;
  end

endtask : run_phase
```

6. Sequencer

The sequencer controls the flow of transactions between the sequence and driver. It arbitrates and schedules transaction execution, ensuring smooth stimulus delivery.

7. Agent

The agent encapsulates key verification components and manages their interactions. It:

- Instantiates the sequencer, driver, and monitor.
- Establishes a connection between the monitor, scoreboard, and coverage collector to enable functional checking and coverage tracking.

```
// Build Phase
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  risc_driver    = driver::type_id::create("risc_driver",this);
  risc_monitor = monitor::type_id::create("risc_monitor",this);
  risc_sequencer = sequencer::type_id::create("risc_sequencer",this);
endfunction : build_phase



// Connect Phase
function void connect_phase (uvm_phase phase);
  super.connect_phase(phase);
   risc_driver.seq_item_port.connect(risc_sequencer.seq_item_export); // connect driver to sequencer
endfunction :connect_phase
```

8. RAL Model

The UVM Register Abstraction Layer (RAL) model is implemented for:

- Data memory and register file modeling, ensuring efficient backdoor access.
- Synchronization with actual memory and register files, allowing for register-level testing and validation.

```
class reg32 extends uvm_reg;
  `uvm_object_utils(reg32)

  uvm_reg_field field;  // register is one field


  function new(string name = "reg32");
    super.new(name, 32, UVM_NO_COVERAGE); // constructor (name , size , coverage)
  endfunction


  virtual function void build();
    field = uvm_reg_field::type_id::create("field");

    //function void configure( uvm_reg  parent,
    // int  unsigned    size,
    // int  unsigned    lsb_pos,
    // string   access,
    // bit  volatile,
    // uvm_reg_data_t   reset,
    // bit  has_reset,
    // bit  is_rand,
    // bit  individually_accessible )

    field.configure(this, 32, 0 , "RW", 0, 32'b0, 1, 1, 1);
```

```
class data_mem extends uvm_mem;
  `uvm_object_utils(data_mem)
  function new(string name = "data_mem");
    super.new(name, // Name of the memory
              256,  // Number of words (1 KB = 256 words of 4 bytes each)
              32,   // Data width (32 bits)
              "RW", // Access rights
              UVM_NO_COVERAGE); // Coverage (optional)
  endfunction
endclass
```

9. Scoreboard

The scoreboard acts as the reference model for verification. It:

- Mimics the pipelining behavior of the RISC-V processor.
- Detects hazard occurrences, ensuring correct handling of data and control dependencies.
- Validates memory and register file operations using UVM RAL backdoor access.
- Predicts expected transactions and compares them against actual DUT outputs.

```
// Instruction pipelining
wr_seq.copy(mem_seq) ;
if(ex_seq.lwstall == 0)
  mem_seq.copy(ex_seq) ;
else
mem_seq.lwstall = ex_seq.lwstall ;
ex_seq.copy(dec_seq) ;
dec_seq.copy(fetch_seq) ;
fetch_seq.copy(sb_item) ;
```

```
SW : begin
  p_item.PCF = dec_seq.lwstall ? dec_seq.PCF : dec_seq.PCF + 4 ;
  p_item.MemWriteM = (ex_seq.beqflush || mem_seq.beqflush || ex_seq.lwstall) ? 0 : 1 ;
  p_item.DataAdrM = (ex_seq.beqflush || mem_seq.beqflush || ex_seq.lwstall) ? 0 : srcA + mem_seq.Extend ;
  p_item.WriteDataM = srcB ;
end

ADD : begin
  p_item.PCF = dec_seq.lwstall ? dec_seq.PCF : dec_seq.PCF + 4 ;
  p_item.MemWriteM = 0 ;
  p_item.DataAdrM = (ex_seq.beqflush || mem_seq.beqflush || ex_seq.lwstall) ? 0 : srcA + srcB ;
end

SUB : begin
  p_item.PCF = dec_seq.lwstall ? dec_seq.PCF : dec_seq.PCF + 4 ;
  p_item.MemWriteM = 0 ;
  p_item.DataAdrM = (ex_seq.beqflush || mem_seq.beqflush || ex_seq.lwstall) ? 0 : srcA - srcB ;
end
```

10. Coverage

Functional coverage is collected to ensure thorough verification. The coverage model includes:

- Instruction coverage: Ensuring all supported instruction types are exercised.
- Transition coverage: Capturing instruction transitions and verifying interactions across pipeline stages.
- Hazard coverage: Ensuring correct handling of data hazards, control hazards, and structural hazards.
- Corner case coverage: Testing edge cases such as register overflows, minimum/maximum operand values, and instruction dependencies.
- Memory and register file coverage: Tracking all read/write operations to verify correct memory interactions.

```
covergroup Instruction_cg ; // To Cover All Instructions and crosses among them
Instructions_cp :       coverpoint instruction {
                        bins I_Type[] = {LW, ADDI, SLLI, SLTI, XORI, SRLI, SRAI, ORI, ANDI, JALR} ;
                        bins R_Type[] = {ADD, SUB, SLL, SLT, XOR, SRL, SRA, OR, AND} ;
                        bins B_Type[] = {BEQ, BNE, BLT, BGE} ;
                        bins U_Type[] = {LUI, AUIPC} ;
                        bins S_Type[] = {SW} ;
                        bins J_Type[] = {JAL} ;
                        bins  Reset[] = {RESET} ;
                        illegal_bins unknow = {UNKNOWN} ;
                        }

Imm_Transitions_cp :    coverpoint instruction {
                        bins I_R[]     = (SLTI, SRLI => SLL, SLT) ;
                        bins I_B[]     = (SLTI => BEQ, BNE, BLT, BGE) ;
                        bins I_SW[]    = (SLTI => SW) ;
                        bins I_SW_SW[] = (SLTI => SW[*2]) ;
                        bins I_LW[]    = (SLTI => LW) ;
                        bins I_LW_LW[] = (SLTI => LW[*2]) ;
                        bins I_J[]     = (ADDI, SLTI => JAL,JALR) ;
                        bins I_J_J[]   = (ADDI, SLTI => JAL[*2]) ;
                        bins I_JR_JR[] = (ADDI, SLTI => JALR[*2]) ;
                        }
```

11. Environment (env)

The UVM environment (env) is responsible for integrating all verification components. It:

- Instantiates the agent, scoreboard, coverage model, and RAL model.
- Passes the RAL model to the scoreboard using config_db.
- Establishes connections between the agent, scoreboard, and coverage collector for seamless data flow.

```
// Build Phase
  function void build_phase(uvm_phase phase);
   super.build_phase(phase);

    ral_model_h   = ral_model::type_id::create("ral_model_h");
    ral_model_h.build();
    ral_model_h.lock_model();
    ral_model_h.reset();
    uvm_config_db#(ral_model)::set(null, "*", "ral_model_h", ral_model_h);

    risc_agent = agent::type_id::create("risc_agent",this);
    risc_scoreboard = scoreboard::type_id::create("risc_scoreboard",this);
    risc_coverage = coverage::type_id::create("risc_coverage",this);
  endfunction :build_phase
```

12. Test

The test layer defines the overall verification strategy. It includes:

- Randomized tests to uncover unexpected design behaviors.
- Directed tests to validate specific scenarios and corner cases.
- A structured approach to ensure comprehensive coverage of all RISC-V instruction types, hazards, and memory operations.

```
class rand_test extends base_test ;
 `uvm_component_utils(rand_test)

  rand_seq rand_s ;

  function new(string name = "rand_test",uvm_component parent=null);
    super.new(name,parent);
  endfunction


  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    rand_s = rand_seq::type_id::create("rand_s");
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    repeat (2) reset_s.start(risc_env.risc_agent.risc_sequencer);
    repeat (10000) rand_s.start(risc_env.risc_agent.risc_sequencer);
    repeat (5) reset_s.start(risc_env.risc_agent.risc_sequencer);
    repeat (10000) rand_s.start(risc_env.risc_agent.risc_sequencer);

    phase.drop_objection(this);
  endtask

endclass
```

- **Run file**

A script automates the compilation, simulation, and coverage collection . It performs the following tasks:

- Compilation: Uses VCS to compile the design with UVM support and coverage metrics enabled.
- Test Execution: Iterates through a test list, running each test separately while generating individual coverage databases.
- Coverage Merging: Aggregates coverage data from all test runs using urg, producing a comprehensive coverage report.
- Logging & Cleanup: Logs results for analysis and removes temporary files after execution.

```
    echo "Running test: $test_name"
    test_vdb="${test_name}.vdb"
    vdb_files+=("-dir $test_vdb") # Add to list of VDB directories

    # Run the test and generate a unique VDB directory
    $sim_exe simv.log +UVM_TESTNAME="$test_name" +covoverwrite -cm_dir $test_vdb
    if [ $? -ne 0 ]; then
        echo "Test $test_name failed! Check simv.log for details." | tee -a $log_file
    else
        echo "Test $test_name passed!" | tee -a $log_file
    fi
done < "$test_list"

# Merge all coverage databases
echo "Merging coverage data from all tests..."
urg "${vdb_files[@]}" -format both -dbname MERGE_COV -report $coverage_dir
if [ $? -ne 0 ]; then
    echo "Coverage merge failed! Check for issues with the VDB directories."
    exit 1
fi

echo "Coverage merge successful! Report available in $coverage_dir"
echo "All tests completed. Check $log_file for summary."

#clean on finish
rm  -rf  *~  csrc  simv*  vc_hdrs.h  ucli.key *.log  novas.* *.fsdb* verdiLog  64* DVEfiles
```

Results

The verification process uncovered several RTL issues, including:

- Control signals: A cycle advance in two signals (funct3 and branchOP)
- Shift Instructions: wrong section of immediate field was pathed to the Extend function
- Memory Alignment: Incorrect handling of unaligned memory accesses.

After debugging and fixing these issues, the design achieved 100% functional coverage, confirming its correctness.

Data from the following tests was used to generate this report

rand_test/test
imm_test/test
branch_test/test
jump_test/test

Total groups in report: 4

NAME	SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
$unit::coverage::Registers_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Instruction_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Reset_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Memory_cg	100.00	1	100	1	0	64	64	

Variables for Group $unit::coverage::Instruction_cg

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEI
Instructions_cp	28	0	28	100.00	100	
Imm_Transitions_cp	20	0	20	100.00	100	
Branch_Transitions_cp	60	0	60	100.00	100	
Jump_Transitions_cp	29	0	29	100.00	100	
SW_Transitions_cp	9	0	9	100.00	100	
LW_Transitions_cp	8	0	8	100.00	100	

Variables for Group $unit::coverage::Registers_cg

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT	AT LEAST
Rs1_cp	32	0	32	100.00	100	1	1
Rs2_cp	32	0	32	100.00	100	1	1
Rd_cp	31	0	31	100.00	100	1	1

Variables for Group $unit::coverage::Memory_cg

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
DataAdr_cp	256	0	256	100.00	100	1
MemWrite_cp	2	0	2	100.00	100	1

Crosses for Group $unit::coverage::Memory_cg

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
Memory_Address_cp	257	0	257	100.00	100	1

Future Work

Future enhancements to this project may include:

- Performance Testing: Analyzing the processor's performance under heavy workloads.
- Verification of Multi-Core Systems: Extending the verification to a multi-core RISC-V processor.
- Enhanced Coverage: Including additional coverage points for power and timing analysis.

Reference

- Harris, Sarah, and David Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 2021.
- Salemi, Ray. *The UVM Primer: An Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.
- Marconi, S., et al. "IEEE Standard for Universal Verification Methodology Language Reference Manual." (2017).
- ChipVerify