Milestone 4: UVM Testbench

ECE-593: Fundamentals of Pre-Silicon Validation

Maseeh College of Engineering and Computer Science

Winter, 2025

Project Name: RISC-V ALU

Team#: 3

Members: Angelo Maldonado-Liu,Saishree Lnu, Niko Nikolov

Date:02/26/2025

Introduction

The UVM testbench for verifying the RV32I ALU is structured around key components that drive, monitor, and validate the design under test (DUT).

The alu_agent is the main component of the design, set in active mode to generate stimuli and observe DUT behavior. It includes a sequencer that produces transactions, a driver that converts these transactions into DUT input signals via a virtual interface, and a monitor that captures both input and output signals, creating transactions for analysis.

The alu_driver connects directly to the DUT through the virtual interface, translating sequencer transactions into precise signal-level activity to stimulate the DUT.

The alu_monitor observes the DUT"s input and output signals in real-time, forming transactions from them and sending them via an analysis port to other components for verification.

The alu_scoreboard checks the DUT"s correctness by comparing its actual outputs, received from the monitor, with expected outputs, which may come from a reference model or transaction calculations, focusing on signals like the program counter, flush, stall, and register write enables.

The alu_coverage collector assesses test completeness by sampling monitor transactions, tracking metrics such as ALU operations, opcodes, branch and jump behaviors, pipeline control signals, and exceptions, using cross-coverage to verify interactions.

The alu_sequence defines the test stimuli, combining directed test cases for critical scenarios with constrained-random transactions to cover a wide range of conditions.

The alu_env integrates all components, configuring the alu_agent as active and linking the monitor"s analysis port to the alu_scoreboard and alu_coverage collector.

The alu_base_test class manages the verification process, instantiating the alu_env, setting it up, and running the alu_sequence on the agent"s sequencer while handling UVM simulation phases.

The transaction class captures all input and output signals for stimulus generation and response checking, and the alu_if virtual interface ensures proper signal communication between the testbench and DUT.

ALU Agent

The ALU agent is implemented as the "alu_agent" class, tasked with stimulus generation and response monitoring for the Device Under Test (DUT).

This component integrates the sequencer, driver, and monitor.

The "alu_agent" class is declared as a subclass of "uvm_agent", a UVM base class designed for encapsulating verification components.

Its instantiation occurs within the "alu_env" class during the "build_phase" function, where it is created unconditionally with the line "agent = alu_agent::type_id::create("agent", this);".

It uses the UVM factory"s "type_id::create" method, assigning the agent the name "agent" and setting its parent to the environment instance ("this").

The agent"s role as a container for subcomponents—sequencer, driver, and monitor—is established through its member declarations: "uvm_sequencer #(transaction) sequencer;", "alu_driver driver;", and "alu_monitor monitor;". These members are defined to work with the "transaction" class.

The agent's internal configuration and instantiation of its subcomponents are handled within its own "build_phase" function, defined as "function void build_phase(uvm_phase phase);".

The monitor is always created with "monitor = alu_monitor::type_id::create("monitor", this);" to ensure continuous observation of the DUT, regardless of the agent"s mode.

If the agent is configured as active, determined by "get_is_active() == UVM_ACTIVE", the sequencer and driver are instantiated with "sequencer = uvm_sequencer#(transaction)::type_id::create("sequencer", this);" and "driver = alu_driver::type_id::create("driver", this);".

The agent"s has a dual capability: active mode for both driving and monitoring, or passive mode for monitoring only.

The active/passive status is set in the "alu_env" and "build_phase" using "uvm_config_db#(uvm_active_passive_enum)::set(this, "agent", "is_active", UVM_ACTIVE);", configuring the agent to generate stimuli via the sequencer and driver.

The connections between the agent subcomponents are established in its "connect_phase" function, implemented as "function void connect_phase(uvm_phase phase);".

When in active mode, the driver is linked to the sequencer with "driver.seq_item_port.connect(sequencer.seq_item_export);", enabling the driver to fetch transactions from the sequencer through a UVM port-based handshake.

The connection ensures that transactions generated by the sequencer—such as ALU operations—are delivered to the driver for application to the DUT. The monitor"s connections to the scoreboard and coverage collector are handled at the environment level in "alu_env""s "connect_phase" with "agent.monitor.mon2scb.connect(scoreboard.scb_port);" and "agent.monitor.mon2scb.connect(coverage.analysis_export);", allowing the monitor"s captured transactions to be broadcast for analysis, though this is outside the "alu_agent""s direct scope.

When the test starts, the agent's sequencer executes the "alu_sequence", generating transactions that the driver retrieves and applies to the DUT"s inputs via the virtual interface "alu_if".

Concurrently, the monitor observes the DUT"s inputs and outputs, packaging them into transactions and sending them via its analysis port ("mon2scb") for further processing by the scoreboard and coverage collector.

The agent"s role is to coordinate these activities, ensuring that stimulus generation (via the sequencer and driver) and response capture (via the monitor) happens. The phases—"build_phase" for instantiation and "connect_phase" for linking—along with factory creation and port-based communication, aligns the implementation with UVM standards.

Monitor

The monitor (alu_monitor) is responsible for capturing the DUT"s behavior in real time, ensuring that the verification process remains independent and reliable.

It monitors both the input stimuli provided to the ALU and the resulting output responses, using a virtual interface to sample these signals. This sampling typically occurs on clock edges, allowing the monitor to accurately record the timing and values of the ALU"s activities.

Once the signals are captured, the monitor organizes them into transaction objects.

These transactions are structured data packets that encapsulate both the inputs and outputs of each ALU operation, creating a detailed snapshot of what occurred during a given cycle.

The monitor then transmits these transaction objects to other parts of the testbench through an analysis port.

The primary recipients of these transactions are the scoreboard and the coverage collector. The scoreboard uses the data to check the correctness of the ALU"s outputs by comparing them against expected results, ensuring that the DUT performs as intended. Meanwhile, the coverage collector leverages the transactions to assess whether the testbench has exercised all necessary scenarios.

Scoreboard

The scoreboard is implemented as the "alu_scoreboard" class. Integrated within the "alu_env" environment, this scoreboard leverages Universal Verification Methodology (UVM) conventions to process transactions from the monitor and ensure the ALU’s functionality aligns with its design specifications.

The "alu_scoreboard" class is declared as a subclass of "uvm_scoreboard", a UVM base class designed for self-checking verification components.

Its instantiation occurs within the "alu_env" class during the "build_phase" function, where it is created unconditionally with the line "scoreboard = alu_scoreboard::type_id::create("scoreboard", this);".

It uses the UVM factory’s "type_id::create" method, assigning the scoreboard the name "scoreboard" and setting its parent to the environment instance ("this").

The scoreboard does not require a transaction type parameter since it uses an analysis import to receive data, defined as "uvm_analysis_imp #(transaction, alu_scoreboard) scb_port;".

The scoreboard’s connection to the testbench is established in the "alu_env" "connect_phase" function with the statement "agent.monitor.mon2scb.connect(scoreboard.scb_port);".

The "mon2scb" is the monitor’s analysis port, and "scb_port" is the scoreboard’s analysis import, instantiated in the "build_phase" with "scb_port = new("scb_port", this);". The connection enables the scoreboard to receive transactions broadcast by the "alu_monitor", which captures the DUT’s input and output signals.

The analysis import automatically invokes the scoreboard’s "write" function for each transaction received, a UVM mechanism that simplifies data collection without requiring explicit polling or handshakes, contrasting with the sequencer-driver interaction.

The core functionality of the "alu_scoreboard" is implemented through its "build_phase", "run_phase", and supporting functions. In the "build_phase", defined as "function void build_phase(uvm_phase phase);", the scoreboard initializes its analysis import and logs a debug message with ""uvm_info("SCB", "build phase", UVM_HIGH)"".

The primary verification logic resides in the "run_phase" task, implemented as "task run_phase(uvm_phase phase);", where it runs a "forever" loop to process transactions. Inside the loop, it waits for transactions to accumulate in a queue ("wait (tx.size() > 0);"), pops the front transaction with "curr_tx = tx.pop_front();", increments a counter, and calls "compare_transactions(curr_tx);" to check the DUT’s outputs.

Transactions are added to the queue via the "write" function, "function void write(transaction item);", which pushes each received transaction into the queue with "tx.push_back(item);".

The "compare_transactions" function, defined as "function void compare_transactions(transaction curr_tx);", performs the actual verification by comparing key DUT outputs against expected behaviors.

For example, it checks the program counter ("if (curr_tx.i_pc !== curr_tx.o_pc)"), flags an error with ""uvm_error("SCB", $sformatf("PC mismatch: Expected %h, Got %h", curr_tx.i_pc, curr_tx.o_pc))"" if mismatched, and verifies conditions like no PC change during flush or stall ("if (curr_tx.i_flush || curr_tx.i_stall || curr_tx.i_force_stall)").

It also confirms register write signals and addresses, reporting successful checks with "uvm_info("SCB", $sformatf("Transaction %d successfully verified!", this.count), UVM_LOW)". This self-checking logic ensures the ALU’s outputs—such as computation results or control signals—are correct for each transaction.

During simulation, the "alu_scoreboard" operates within the testbench’s flow, initiated by the "alu_base_test"’s "run_phase".

As the sequencer generates transactions and the driver stimulates the DUT, the monitor captures the resulting behavior and broadcasts transactions via its analysis port to the scoreboard. The scoreboard collects these transactions in its queue, processes them in the "run_phase", and performs comparisons to validate the ALU’s functionality. Its use of UVM phases ("build_phase" for setup, "run_phase" for execution), factory instantiation, and analysis import integration .

Driver

The driver is implemented as the "alu_driver" class, translating high-level transactions into signal-level stimuli for the Device Under Test (DUT).

The "alu_driver" class is declared as a subclass of "uvm_driver", parameterized with the "transaction" type to handle ALU-specific transactions.

Its instantiation occurs within the "alu_agent" class during the "build_phase" function, where it is created conditionally if the agent is in active mode, determined by "get_is_active() == UVM_ACTIVE".

The instantiation is executed with the line "driver = alu_driver::type_id::create("driver", this);", utilizing the UVM factory"s "type_id::create" method. This assigns the driver the name "driver" and sets its parent to the agent instance ("this"). The factory-based approach allows for potential overrides, while the parameterization ensures compatibility with the "transaction" class, which includes fields like "i_alu", "i_rs1", and "i_opcode" specific to ALU operations.

The driver"s connection to the sequencer is established in the "alu_agent""s "connect_phase" function with the statement "driver.seq_item_port.connect(sequencer.seq_item_export);", executed only in active mode.

The "seq_item_port" is the driver"s port, inherited from "uvm_driver", and it connects to the sequencer"s "seq_item_export". The port-based connection sets up a handshake mechanism, allowing the driver to request transactions from the sequencer using "get_next_item" and signal their completion with "item_done".

The core functionality of the "alu_driver" is implemented across its "build_phase", "run_phase", and "drive_item" task.

In the "build_phase", defined as "function void build_phase(uvm_phase phase);", the driver retrieves a virtual interface to the DUT with "uvm_config_db#(virtual alu_if)::get(this, "alu_vif", drv_if)".

If the interface isn"t found, it triggers a fatal error with "uvm_fatal("NO_VIF", "Virtual interface not found in config db with key "alu_vif"")", ensuring the testbench fails early if misconfigured.

The UVM configuration database decouples the driver from the DUT"s physical interface, enhancing reusability across different test setups.

The driver"s main operation occurs in the "run_phase" task, implemented as "task run_phase(uvm_phase phase);".

It runs an infinite loop with "forever begin", where it retrieves a transaction using "seq_item_port.get_next_item(tx)", calls "drive_item(tx)" to apply the transaction to the DUT, and signals completion with "seq_item_port.item_done()".

This loop keeps the driver active throughout the simulation, processing transactions as they arrive from the sequencer. The "drive_item" task, defined as "virtual task drive_item(transaction tx);", translates the transaction into DUT signals.

It logs the action with ""uvm_info("DRV", $sformatf("Driving transaction: %s", tx.convert2string()), UVM_HIGH)"" for debugging, then checks the reset signal "drv_if.i_rst_n".

If reset is active ("!drv_if.i_rst_n"), it drives default values (e.g., "drv_if.i_alu <= 0;"), otherwise it assigns transaction values (e.g., "drv_if.i_alu <= tx.i_alu;") using a clocking block ("drv_if.cb_input") for timing precision, falling back to direct assignments if unavailable. The task synchronizes with the DUT clock using "@(posedge drv_if.i_clk)", ensuring proper signal timing.

As the "alu_base_test" initiates the "run_phase", the sequencer generates transactions via the "alu_sequence", and the driver retrieves these transactions through its connected port. It then drives the DUT"s inputs—such as operands and operation codes—while the monitor captures responses for analysis by the scoreboard and coverage collector.

The driver"s use of UVM phases ("build_phase" for setup, "run_phase" for execution), the configuration database for interface retrieval, and port-based communication .

Sequencer

The sequencer is implemented as a specialized component named alu_sequencer", integrated within the "alu_agent" class to manage the generation and delivery of test transactions to the driver.

The sequencer is declared and instantiated within the "alu_agent" class as a parameterized type based on the "uvm_sequencer" base class, to handle transactions of type "transaction".

In the "alu_agent""s "build_phase" function, the sequencer is created conditionally if the agent is set to active mode, which is determined by the "get_is_active()" method returning "UVM_ACTIVE".

The instantiation occurs with the line "sequencer uvm_sequencer#(transaction)::type_id::create("sequencer", this);".

This uses the UVM factory"s "type_id::create" method, passing a name ("sequencer") and a parent pointer ("this", referring to the agent).

The factory-based creation allows for potential overrides, enhancing flexibility, while the parameterization with "transaction" ensures the sequencer works with the specific transaction class defined for ALU operations, such as those containing fields like "i_alu", "i_rs1", and "i_opcode".

Once instantiated, the sequencer must be connected to the driver to enable transaction flow, which is handled in the "alu_agent"s "connect_phase" function.

The connection is established with the statement "driver.seq_item_port.connect(sequencer.seq_item_export);", executed only if the agent is active. This line links the driver"s "seq_item_port"—a UVM port inherited from the "uvm_driver" base class—to the sequencer"s "seq_item_export", an export port provided by the "uvm_sequencer" class.

This port-based connection sets up a handshake mechanism where the driver can request transactions from the sequencer using "get_next_item" and signal completion with "item_done", ensuring controlled and synchronized delivery of test stimuli to the ALU DUT.\

An instance of the "alu_sequence" class is created with "seq = alu_sequence::type_id::create("seq");", again using the UVM factory for instantiation.

The sequence is then started on the sequencer with "seq.start(env.agent.sequencer);", where "env.agent.sequencer" references the sequencer instance within the agent, itself contained in the "alu_env" environment.

This call triggers the sequence"s "body" task, which defines the test scenario—such as generating a series of transactions with randomized or directed ALU operations—and interacts with the sequencer to send those transactions to the driver.

While the sequencer itself does not define the transaction content (that"s handled by the "alu_sequence" class), its role is to manage the execution of the sequence and facilitate transaction delivery.

When the "run_phase" begins, the test raises an objection to keep the simulation active, starts the sequence on the sequencer, and allows it to generate transactions—say, 20 randomized ALU operations as specified in the "alu_sequence""s default configuration ("num_transactions = 20").

The sequencer queues these transactions and delivers them to the driver, which drives the DUT"s inputs, while the monitor captures responses, and the scoreboard and coverage collector analyze the results.

Subscriber

The subscriber is implemented as the "alu_coverage" class, designed to collect and analyze coverage data to evaluate the thoroughness of the verification process.

It is integrated within the "alu_env" environment. It samples transactions from the monitor and tracks key metrics about the ALU"s behavior.

The "alu_coverage" class is declared as a subclass of "uvm_subscriber", parameterized with the "transaction" type to process ALU-specific transactions.

Its instantiation occurs within the "alu_env" class during the "build_phase" function, where it is created unconditionally with the line "coverage = alu_coverage::type_id::create("coverage", this);".

It uses the UVM factory"s "type_id::create" method, assigning it the name "coverage" and setting the parent to the environment instance ("this").

Unlike the sequencer, which is tied to an active agent, the subscriber"s role is passive, focusing solely on receiving and analyzing data, so no active/passive condition is needed. The factory-based creation ensures flexibility, allowing potential overrides, while the parameterization ensures compatibility with the "transaction" class containing fields like "i_opcode", "i_alu", and "o_change_pc".

The subscriber"s connection to the testbench is established in the (environment) "alu_env""s "connect_phase" function, where it is linked to the "alu_monitor""s analysis port with the statement "agent.monitor.mon2scb.connect(coverage.analysis_export);".

The "mon2scb" is the analysis port of the monitor, and "analysis_export" is the subscriber"s export port, inherited from "uvm_subscriber".

This connection enables the "alu_coverage" class to receive transactions broadcast by the monitor, which captures the DUT"s input and output signals.

Unlike the sequencer-driver relationship, which involves a request-response handshake, the subscriber operates as a listener, passively collecting data pushed to it via the analysis port, a standard UVM mechanism for one-to-many communication.

The core functionality of the "alu_coverage" subscriber is defined in its implementation, particularly through the "write" function and a coverage group ("alu_cg").

The "write" function, inherited from "uvm_subscriber", is automatically called whenever a transaction is received via the analysis port. In the code, it is implemented as "function void write(transaction t);", where it assigns the incoming transaction to a class variable "tr = t;" and samples it with the covergroup if certain conditions are met ("if (tr.i_ce || !tr.i_rst_n) alu_cg.sample();").

Coverage is collected only during active clock cycles or reset states, aligning with the ALU"s operational context. The "alu_cg" covergroup defines bins for key signals—like opcodes (e.g., R-type, I-type), ALU operations (e.g., ADD, SUB), reset states, and pipeline controls—along with cross-coverage to track interactions, such as opcodes with exceptions.

In the "extract_phase" function, where the subscriber reports the achieved coverage percentage with the line ""uvm_info("COVERAGE", $sformatf("Functional Coverage: %0.2f%%", alu_cg.get_coverage()), UVM_LOW)"".

This phase runs after the main simulation, providing a summary of how comprehensively the testbench exercised the ALU"s functionality. The "alu_coverage" class also includes a constructor ("function new(string name, uvm_component parent);"), which initializes the covergroup with "alu_cg = new();", ensuring it"s ready to sample data as soon as transactions arrive.

During simulation, the subscriber operates within the broader testbench flow. As the "alu_base_test" initiates the "run_phase" and the sequencer generates transactions, the driver stimulates the DUT, and the monitor captures the resulting signals.

These transactions are broadcast via the monitor"s analysis port to both the scoreboard and the "alu_coverage" subscriber.

The subscriber collects and analyzes this data throughout the simulation, incrementing coverage bins as it encounters various opcodes, operations, and conditions. By the end of the run, it provides a quantitative measure of test completeness, ensuring critical aspects of the ALU—like all 32 registers or branch behaviors—are adequately tested.

UVM Base Test (test.sv)

In the UVM testbench for the RV32I ALU, the "alu_base_test" class serves as the top-level test managing the entire verification process, setting up the environment, executing test sequences, and managing simulation flow.

The "alu_base_test" class is defined as a subclass of "uvm_test", a UVM base class designed for top-level test components.

Its instantiation occurs implicitly when the simulation invokes it via "run_test("alu_base_test")" in the top module, as specified in the provided "top.sv". The class is registered with the UVM factory using "uvm_component_utils(alu_base_test)".

Within the class, key members are declared: "alu_env env" to hold the testbench environment (agent, scoreboard, and coverage collector), and "virtual alu_if vif" to provide a connection to the DUT's interface. A file handle "integer log_file" is also declared to manage UVM logging output, ensuring simulation messages are captured for debugging.

The configuration and setup of the testbench occur in the "build_phase" function, implemented as "function void build_phase(uvm_phase phase);". It starts by calling "super.build_phase(phase)" to execute the parent class's initialization. It then opens a log file with "log_file = $fopen("uvm_log.txt", "w")", and if successful, configures UVM reporting with commands like "set_report_severity_action_hier(UVM_INFO, UVM_DISPLAY | UVM_LOG)" to display and log informational messages, and similar settings for warnings and errors.

These messages are directed to the file using "set_report_severity_file_hier", and a confirmation is logged with "uvm_info("TEST", "Log file opened and report server configured", UVM_LOW)". If the file fails to open, a ""uvm_fatal"" halts the simulation with an error message. Next, the virtual interface is retrieved from the UVM configuration database with "uvm_config_db#(virtual alu_if)::get(this, "", "alu_vif", vif)", matching the key "alu_vif" set in the top module. A failure here triggers another ""uvm_fatal"", ensuring the test stops if the interface isn't available. Finally, the environment is instantiated with "env =

alu_env::type_id::create("env", this);", creating the agent, scoreboard, and coverage components within the test hierarchy.

At the "end_of_elaboration_phase" function, it cleans up all the resources it uses. Defined as "function void end_of_elaboration_phase(uvm_phase phase);". This phase runs after the testbench is fully constructed but before the main simulation begins.

It closes the log file with "$fclose(log_file)" if it was opened, logging a confirmation message with ""uvm_info("TEST", "Log file closed", UVM_LOW)"". This step prevents resource leaks, ensuring the file handle is released cleanly before the simulation proceeds to the execution phase.

The execution of the test occurs in the "run_phase" task, implemented as "task run_phase(uvm_phase phase);".

This phase starts by raising an objection with "phase.raise_objection(this, "Starting ALU test sequence")", signaling that the simulation should remain active while the test runs. It then creates an instance of the "alu_sequence" class with "seq = alu_sequence::type_id::create("seq");", using the UVM factory for instantiation.

The sequence is started on the agent's sequencer with "seq.start(env.agent.sequencer);", triggering the generation of transactions—such as randomized ALU operations—that the driver applies to the DUT.

A delay of "#10000" provides sufficient time for the sequence to complete its default 20 transactions (as set by "num_transactions = 20" in "alu_sequence"), after which the objection is dropped with "phase.drop_objection(this, "ALU test sequence completed")", allowing the simulation to end gracefully.

Initiated by the top module's "run_test("alu_base_test")", it constructs the testbench in the "build_phase", retrieves the DUT interface set by the top module ("uvm_config_db#(virtual alu_if)::set(null, "*", "alu_vif", dut_if)"), and instantiates the environment. In the "run_phase", it launches the sequence, which generates transactions that flow through the sequencer to the driver, stimulating the ALU DUT.

The monitor captures responses, broadcasting them to the scoreboard for correctness checks and the coverage collector for completeness analysis, while UVM logs are written to "uvm_log.txt". The "end_of_elaboration_phase" ensures cleanup, and the simulation runs for 10,000 time units.

Current Errors:

Currently there are some errors with the scoreboard verifying the DUT, these errors require more time than we had to fix them, but they will be fixed by the project deadline.

We were unable to get the logging to work properly. I tried many different methods and tried the example given in the lecture and it didn't work for me. Will try to have it working by the end of the project. Currently we can write to the terminal, but when it comes to writing to the log file, it writes only locally, after it was defined and set in the test.sv file, it doesn't go down the hierarchy like it is supposed to. I managed to cobble together something that sort of works, where every single build phase has the exact same code that connects their output to the uvm_log.txt, but unfortunately this means sometimes things write concurrently and it ruins the formatting.

Appendix:

Repository at branch m4:

- https://github.com/nnikolov3/RISC-V-ALU-Design-and-Verification/tree/m4

Coverage runs:

AI Help:

We used Grok (grok.com) for debugging, helping with writing this report, and getting example code.