Milestone 2+3: Class-based 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:01/31/2025

Testbench

Designed to verify the rv32i_alu module, and also serves as the top module in our design.

The testbench interfaces with the ALU through an alu_if instance (interface.sv), a SystemVerilog interface that provides a structured way to combine all input and output signals.

On the input side, signals such as i_clk, i_rst_n, i_alu, i_rs1, i_rs2, i_imm, i_opcode, and various control signals like i_ce, i_stall, and i_flush are driven into the ALU.

On the output side, it captures signals including the computation result o_y, bypassed operands (o_rs1, o_rs2), program counter updates (o_next_pc, o_change_pc), and pipeline control signals (o_stall_from_alu, o_ce, o_flush).

The testbench uses components instantiated within the initial block, such as generator, a driver, two monitors (input and output), a scoreboard, and a coverage collector. These components are launched concurrently using a fork construct with join_none, enabling parallel execution to maximize efficiency and emulate real-time DUT interaction.

The generator class is responsible for creating the test stimuli.

It constructs a comprehensive set of transaction objects, each representing a test scenario with fields such as ALU operation (i_alu), operands (i_rs1, i_rs2, i_imm), opcode (i_opcode), and control signals etc.

The transactions are dispatched to the driver via the interface mailbox (driver_mb), creating a decoupled producer-consumer relationship.

The driver, instantiated from the driver class (driver.sv), translates these high-level transactions into low-level signal manipulations on the alu_if interface (interface.sv). For each transaction received from the mailbox, it assigns values to DUT inputs—such as setting i_alu for the desired operation or i_rs1 and i_imm for operands—and synchronizes these updates with the positive edge of i_clk.

Essentially, trying to mimic the timing of a real pipeline.

Concurrently, two monitors observe the DUT's behavior.

The input monitor (monitor_in) samples the ALU's input signals on every positive clock edge when i_ce is asserted, capturing data such as i_alu, i_rs1, and i_opcode into a transaction object.

This transaction is then sent to the scoreboard via the mon_in2scb mailbox.

The output monitor (monitor_out) performs a similar role for the ALU's outputs, waiting for o_ce to indicate valid data before sampling signals like o_y, o_next_pc, and o_rd_valid. These output transactions are forwarded to the scoreboard through the mon_out2scb mailbox.

Together, these monitors provide a complete record of the ALU's input-output behavior.

The scoreboard, an instance of the scoreboard class (scoreboard.sv), receives input transactions from mon_in2scb and buffers their signals in FIFO arrays (e.g., i_alu_fifo, i_rs1_fifo), preserving the sequence of operations.

As output transactions arrive from mon_out2scb, it retrieves the corresponding input data from these FIFOs, computes the expected ALU result using the alu_operation function, and compares it against the actual outputs.

The alu_operation function, defined within the testbench, mirrors the ALU's logic by interpreting the one-hot encoded i_alu signal to perform operations such as addition (a + b), subtraction (a - b), or comparisons (e.g., a < b). It supports the full range of RV32I ALU operations, including SLT, SLL, and GEU.

The scoreboard checks the computation result (o_y), control signals (e.g., o_change_pc for branches, o_wr_rd for writeback) and pipeline states (e.g., o_stall). Any mismatch triggers an error message.

The coverage collector class (coverage.sv) monitors the DUT signals via the same alu_if interface (interface.sv).

It defines a covergroup (alu_cg) that samples key signals—such as i_opcode, i_alu, o_change_pc, and pipeline controls—on clock edges when i_ce is high. This covergroup includes coverpoints for individual signal ranges (e.g., all 32 destination register addresses) and cross coverage for signal interactions (e.g., opcodes with ALU operations or stalls).

After some number of time units, it reports coverage percentages.

However, we had some trouble reaching beyond 40% the functional coverage at this stage. We will continue to improve our coverage as we progress through UVM.

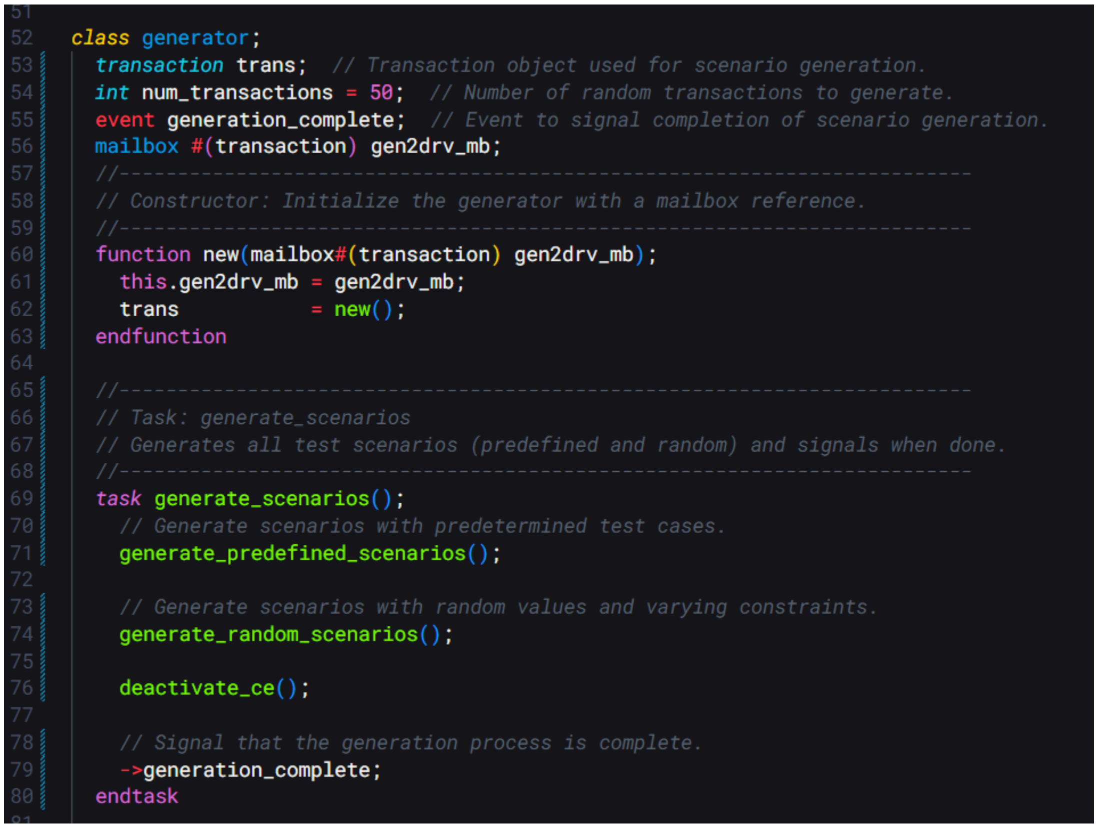
The simulation runs for 10,000 time units before terminating with $finish, allowing for the generator to exhaust its predefined and random scenarios.

Generator

This class is responsible for generating both predefined and randomized transactions to verify the RISC-V ALU design.

It creates test scenarios for arithmetic, logical, shift, comparison, memory, branch, jump, upper immediate, system, and fence operations.

Each generated transaction is sent to the driver via mailbox.



```
class generator;
  transaction trans;  // Transaction object used for scenario generation.
  int num_transactions = 50;  // Number of random transactions to generate.
  event generation_complete;  // Event to signal completion of scenario generation.
  mailbox #(transaction) gen2drv_mb;
  //----------------------------------------------------------------------
  // Constructor: Initialize the generator with a mailbox reference.
  //----------------------------------------------------------------------
  function new(mailbox#(transaction) gen2drv_mb);
    this.gen2drv_mb = gen2drv_mb;
    trans           = new();
  endfunction

  //----------------------------------------------------------------------
  // Task: generate_scenarios
  // Generates all test scenarios (predefined and random) and signals when done.
  //----------------------------------------------------------------------
  task generate_scenarios();
    // Generate scenarios with predetermined test cases.
    generate_predefined_scenarios();

    // Generate scenarios with random values and varying constraints.
    generate_random_scenarios();

    deactivate_ce();

    // Signal that the generation process is complete.
    ->generation_complete;
  endtask
```

Figure 1 Generator class, mailbox of type transaction

The initialization routine derives a seed from configuration parameters, establishing a nontrivial state that is immediately processed to ensure unique, predictable state transitions.

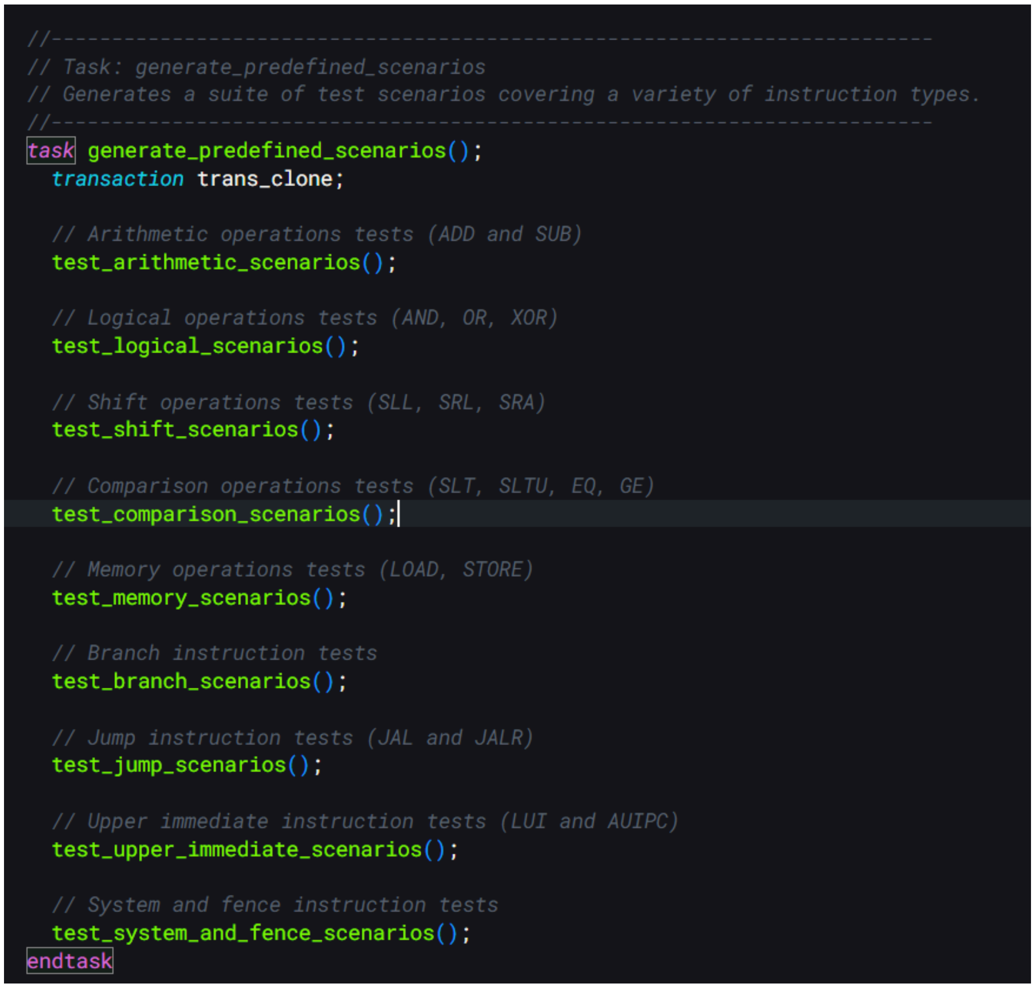
The core loop applies a recurrence relation that combines bitwise operators with modular arithmetic, thereby minimizing computational overhead and enforcing strict determinism.

Each transaction represents a distinct test scenario.

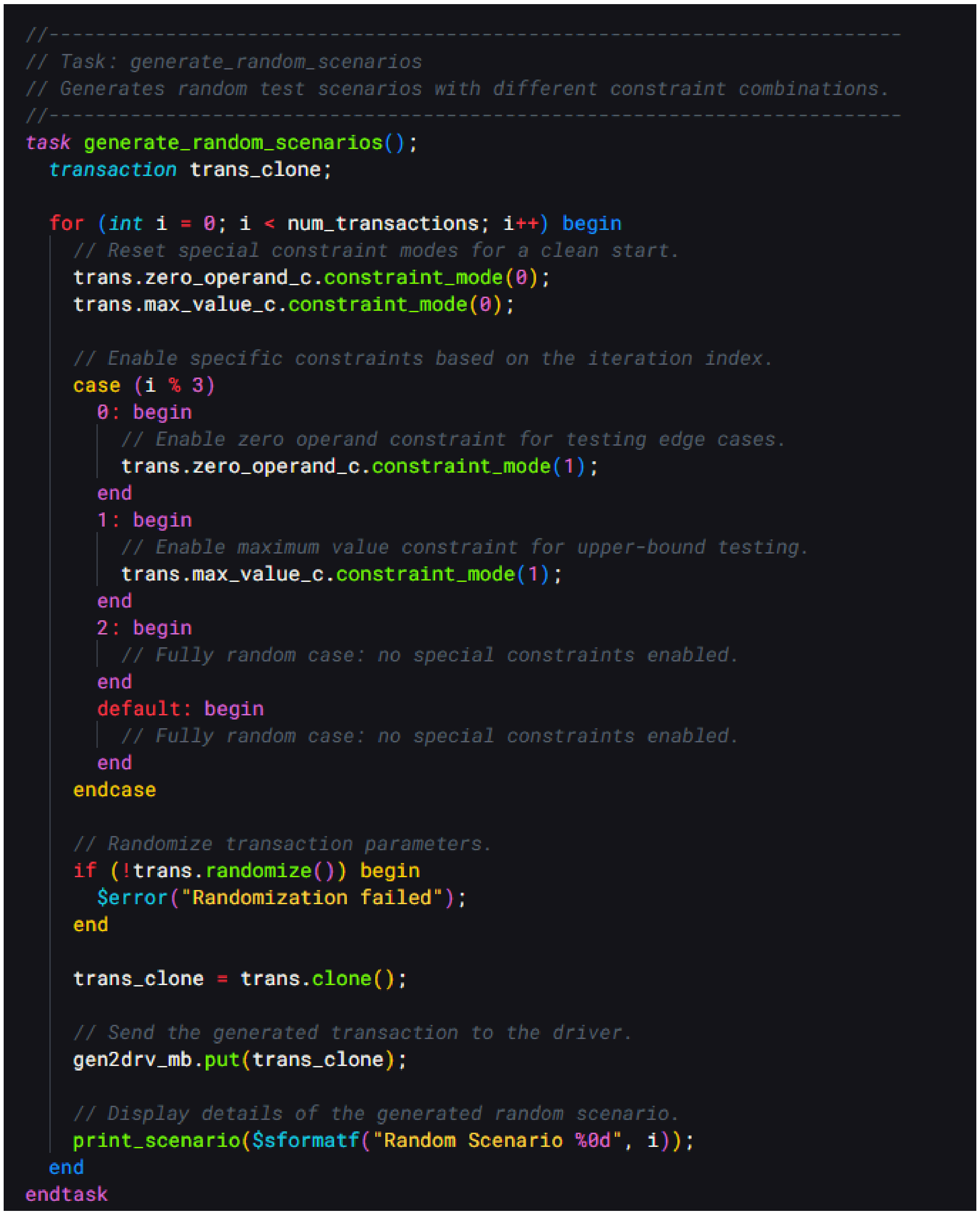
The module defines two enumerated types, one specifies the full set of ALU operations—including arithmetic, logical, shift, and comparison functions—while the other employs one-hot encoding to represent various RISC-V instruction formats.

Transaction generation is done in two ways. One method uses predefined tasks that generate test scenarios targeting specific instruction categories, ensuring that edge conditions such as overflow, underflow, and boundary constraints are rigorously exercised.

The other method employs a constraint-based random testing strategy that rotates among different modes—testing zero operands, maximum values, and fully randomized inputs. This dual approach provides both directed coverage of known critical cases and the opportunity to uncover unexpected issues.



```
//----------------------------------------------------------------------
// Task: generate_predefined_scenarios
// Generates a suite of test scenarios covering a variety of instruction types.
//----------------------------------------------------------------------
task generate_predefined_scenarios();
  transaction trans_clone;

  // Arithmetic operations tests (ADD and SUB)
  test_arithmetic_scenarios();

  // Logical operations tests (AND, OR, XOR)
  test_logical_scenarios();

  // Shift operations tests (SLL, SRL, SRA)
  test_shift_scenarios();

  // Comparison operations tests (SLT, SLTU, EQ, GE)
  test_comparison_scenarios();

  // Memory operations tests (LOAD, STORE)
  test_memory_scenarios();

  // Branch instruction tests
  test_branch_scenarios();

  // Jump instruction tests (JAL and JALR)
  test_jump_scenarios();

  // Upper immediate instruction tests (LUI and AUIPC)
  test_upper_immediate_scenarios();

  // System and fence instruction tests
  test_system_and_fence_scenarios();
endtask
```



```
//--------------------------------------------------------------------------
// Task: generate_random_scenarios
// Generates random test scenarios with different constraint combinations.
//--------------------------------------------------------------------------
task generate_random_scenarios();
  transaction trans_clone;

  for (int i = 0; i < num_transactions; i++) begin
    // Reset special constraint modes for a clean start.
    trans.zero_operand_c.constraint_mode(0);
    trans.max_value_c.constraint_mode(0);

    // Enable specific constraints based on the iteration index.
    case (i % 3)
      0: begin
        // Enable zero operand constraint for testing edge cases.
        trans.zero_operand_c.constraint_mode(1);
      end
      1: begin
        // Enable maximum value constraint for upper-bound testing.
        trans.max_value_c.constraint_mode(1);
      end
      2: begin
        // Fully random case: no special constraints enabled.
      end
      default: begin
        // Fully random case: no special constraints enabled.
      end
    endcase

    // Randomize transaction parameters.
    if (!trans.randomize()) begin
      $error("Randomization failed");
    end

    trans_clone = trans.clone();

    // Send the generated transaction to the driver.
    gen2drv_mb.put(trans_clone);

    // Display details of the generated random scenario.
    print_scenario($sformatf("Random Scenario %0d", i));
  end
endtask
```

Figure 3 Constraint based random test generation.

Generator AI Summary:

Class: generator
Purpose
The generator class creates a comprehensive set of test scenarios to exercise the RV32I ALU. It generates both predefined test cases targeting specific operations and random scenarios to cover

a broader range of conditions. These scenarios are encapsulated as transaction objects, which are sent to the driver through a mailbox for execution.

Key Components

- trans: A transaction object that defines the parameters of each test scenario (e.g., operands, operation type, immediate values).
- num_transactions: An integer specifying the number of random transactions to generate (default: 50).
- generation_complete: An event triggered to indicate that all test scenarios have been generated.
- gen2drv_mb: A mailbox used to pass transaction objects from the generator to the driver.

Core Functions

- **Constructor (**new**)**
  Initializes the generator with a reference to the gen2drv_mb mailbox and instantiates a new transaction object for scenario generation.
- **Task:** generate_scenarios
  The main entry point for test generation. It orchestrates the execution of:
  - generate_predefined_scenarios(): Produces targeted test cases.
  - generate_random_scenarios(): Produces randomized test cases.
  - deactivate_ce(): Deactivates the clock enable signal (i_ce = 0) after generation.
  - Signals completion via the generation_complete event.
- **Task:** generate_predefined_scenarios
  Generates a suite of predefined test cases covering various RISC-V instruction categories:
  - **Arithmetic**: Tests ADD (e.g., overflow) and SUB (e.g., underflow).
  - **Logical**: Tests AND, OR, and XOR with specific bit patterns.
  - **Shift**: Tests SLL, SRL, and SRA with edge cases (e.g., maximum shift).
  - **Comparison**: Tests SLT, SLTU, EQ, and GE with boundary conditions.
  - **Memory**: Tests LOAD and STORE for address handling.
  - **Branch**: Tests branch taken/not-taken conditions.
  - **Jump**: Tests JAL and JALR with forward jumps and return addresses.
  - **Upper Immediate**: Tests LUI and AUIPC with large immediates.
  - **System/Fence**: Tests CSR operations and memory ordering (FENCE).
- **Task:** generate_random_scenarios
  Produces num_transactions random test cases with varied constraints:
  - **Zero Operands**: Forces one operand to zero (enabled every third iteration).
  - **Max Values**: Uses maximum operand values (enabled every third iteration).
  - **Fully Random**: No special constraints (default case).
  - Randomizes transaction parameters and sends each to the driver via the mailbox.

- **Task:** deactivate_ce
  Sets the clock enable (i_ce) to 0, sends the final transaction, and prepares the system for shutdown.
- **Function:** print_scenario
  Displays detailed information about each test scenario, including:
  - Operation type (e.g., ADD, SUB).
  - Instruction type (e.g., R_TYPE, I_TYPE).
  - Operands (i_rs1, i_rs2), immediate (i_imm), and clock enable (i_ce).
  - Expected result based on the ALU operation.

Behavior

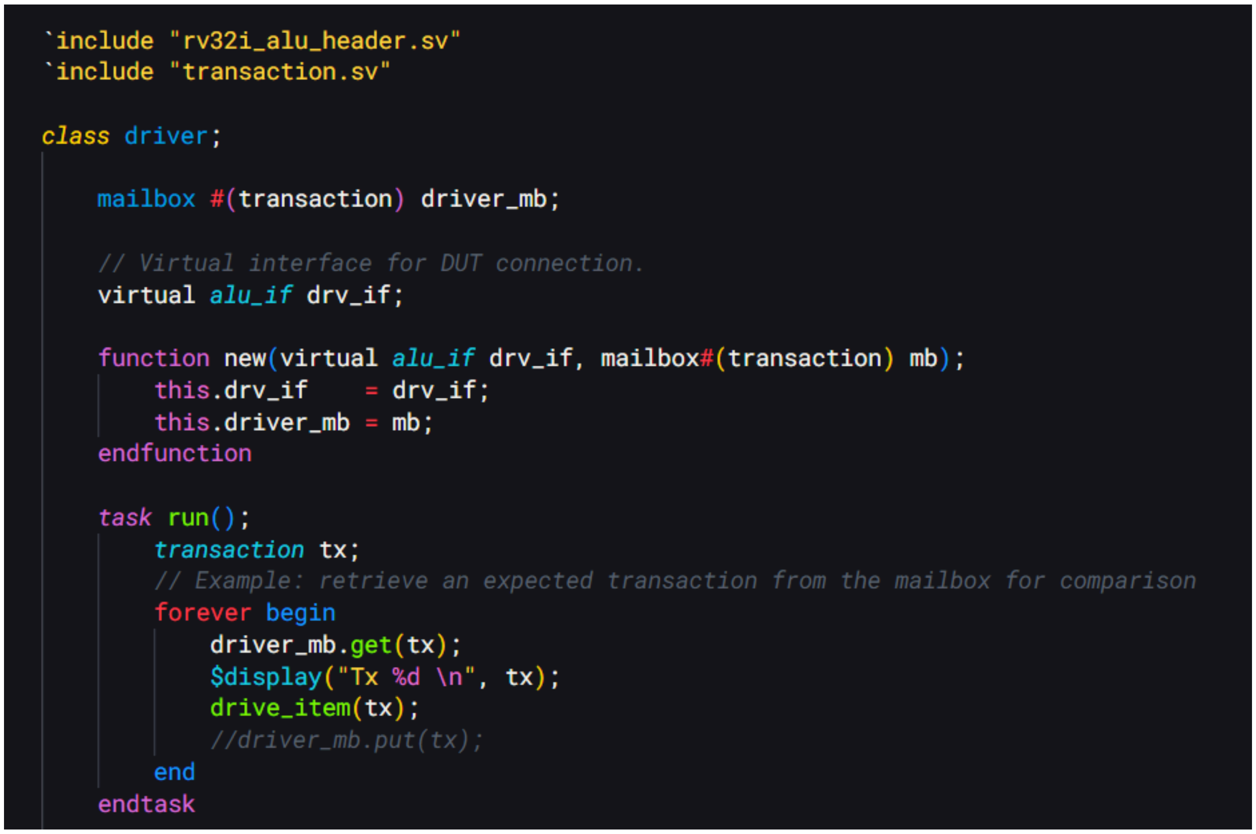
The generator class ensures thorough verification of the RV32I ALU by:

1. Creating predefined scenarios to test specific functionality and edge cases (e.g., overflow, sign preservation).
2. Generating random scenarios to explore a wide range of inputs and conditions.
3. Using a structured approach to send transactions to the driver and signal completion.

This combination of targeted and random testing provides robust coverage of the ALU's arithmetic, logical, shift, comparison, and control operations, making it an essential component in the verification process.

Driver

The driver module (driver.sv) implements a communication mechanism between the generator, the monitor and the interface.

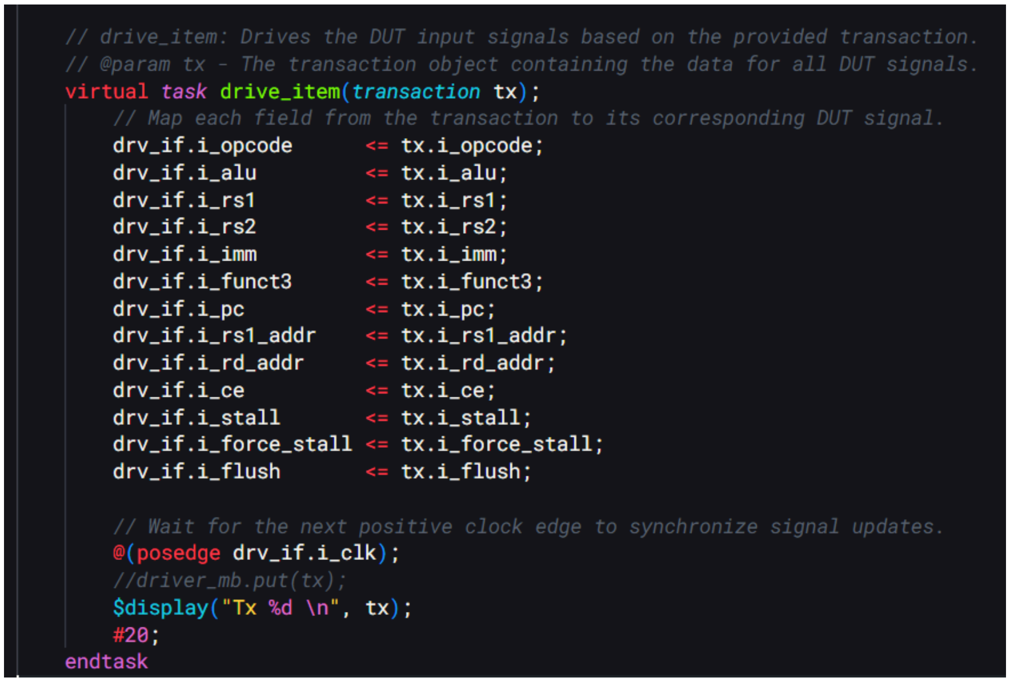


```
`include "rv32i_alu_header.sv"
`include "transaction.sv"

class driver;

    mailbox #(transaction) driver_mb;

    // Virtual interface for DUT connection.
    virtual alu_if drv_if;

    function new(virtual alu_if drv_if, mailbox#(transaction) mb);
        this.drv_if    = drv_if;
        this.driver_mb = mb;
    endfunction

    task run();
        transaction tx;
        // Example: retrieve an expected transaction from the mailbox for comparison
        forever begin
            driver_mb.get(tx);
            $display("Tx %d \n", tx);
            drive_item(tx);
            //driver_mb.put(tx);
        end
    endtask
```

Figure 4 Mailbox and object instantiation

The driver monitors a mailbox interface for incoming transaction objects.

Once a transaction is detected, a non-blocking read mechanism retrieves the object and transfers its contents into a set of signals destined for the design under test.



```
// drive_item: Drives the DUT input signals based on the provided transaction.
// @param tx - The transaction object containing the data for all DUT signals.
virtual task drive_item(transaction tx);
    // Map each field from the transaction to its corresponding DUT signal.
    drv_if.i_opcode      <= tx.i_opcode;
    drv_if.i_alu         <= tx.i_alu;
    drv_if.i_rs1         <= tx.i_rs1;
    drv_if.i_rs2         <= tx.i_rs2;
    drv_if.i_imm         <= tx.i_imm;
    drv_if.i_funct3      <= tx.i_funct3;
    drv_if.i_pc          <= tx.i_pc;
    drv_if.i_rs1_addr    <= tx.i_rs1_addr;
    drv_if.i_rd_addr     <= tx.i_rd_addr;
    drv_if.i_ce          <= tx.i_ce;
    drv_if.i_stall       <= tx.i_stall;
    drv_if.i_force_stall <= tx.i_force_stall;
    drv_if.i_flush       <= tx.i_flush;

    // Wait for the next positive clock edge to synchronize signal updates.
    @(posedge drv_if.i_clk);
    //driver_mb.put(tx);
    $display("Tx %d \n", tx);
    #20;
endtask
```

Figure 5 Driver task that routes the update to the mailbox transaction of transaction type

The implementation separates the communication logic from the signal-driving routines, resulting in a modular structure that enhances both maintainability and scalability.

Driver AI Summary:

Class: driver

- **Purpose**:
  Drives the RV32I ALU DUT by converting transaction objects into low-level signal manipulations on the DUT's interface (alu_if).
- **Key Components**:
  - driver_mb: Mailbox to receive transaction objects from the sequencer.
  - drv_if: Virtual interface (alu_if) to connect to the DUT.
- **Core Functions**:
  - **Constructor** (new):
    Initializes the driver with the virtual interface and mailbox.
  - **Task:** run:
    Runs in a infinite loop, fetching transactions from driver_mb and calling drive_item to drive the DUT.

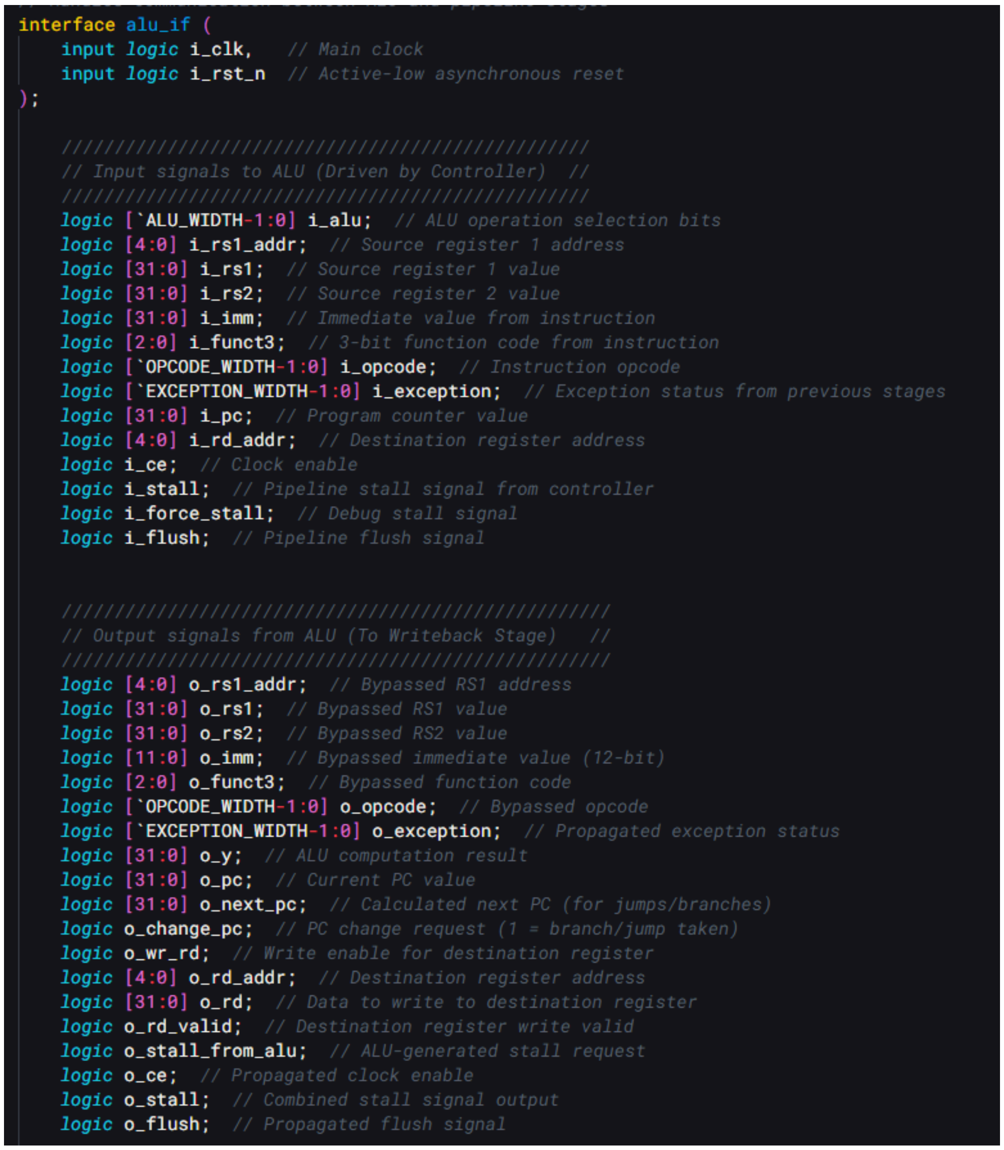
    - **Task:** drive_item:
      Maps transaction fields (e.g., i_opcode, i_rs1) to DUT signals via drv_if. Updates signals at the positive edge of i_clk, followed by a #20 delay for simulation timing.

- **Behavior**:
  Continuously drives DUT inputs with transaction data, synchronized to the clock, ensuring controlled stimulus application.

Interface

The interface (interface.sv), provides a structured way for signal exchange between verification components and the design under test by grouping in one place all of the signals.

It specifies the essential signals, such as the clock, reset, valid, ready, and data signals, and establishes a coherent protocol for transaction communication.



```
interface alu_if (
    input logic i_clk,   // Main clock
    input logic i_rst_n  // Active-low asynchronous reset
);

    //////////////////////////////////////////////////
    // Input signals to ALU (Driven by Controller)  //
    //////////////////////////////////////////////////
    logic [`ALU_WIDTH-1:0] i_alu;  // ALU operation selection bits
    logic [4:0] i_rs1_addr;  // Source register 1 address
    logic [31:0] i_rs1;  // Source register 1 value
    logic [31:0] i_rs2;  // Source register 2 value
    logic [31:0] i_imm;  // Immediate value from instruction
    logic [2:0] i_funct3;  // 3-bit function code from instruction
    logic [`OPCODE_WIDTH-1:0] i_opcode;  // Instruction opcode
    logic [`EXCEPTION_WIDTH-1:0] i_exception;  // Exception status from previous stages
    logic [31:0] i_pc;  // Program counter value
    logic [4:0] i_rd_addr;  // Destination register address
    logic i_ce;  // Clock enable
    logic i_stall;  // Pipeline stall signal from controller
    logic i_force_stall;  // Debug stall signal
    logic i_flush;  // Pipeline flush signal


    ////////////////////////////////////////////////////
    // Output signals from ALU (To Writeback Stage)   //
    ////////////////////////////////////////////////////
    logic [4:0] o_rs1_addr;  // Bypassed RS1 address
    logic [31:0] o_rs1;  // Bypassed RS1 value
    logic [31:0] o_rs2;  // Bypassed RS2 value
    logic [11:0] o_imm;  // Bypassed immediate value (12-bit)
    logic [2:0] o_funct3;  // Bypassed function code
    logic [`OPCODE_WIDTH-1:0] o_opcode;  // Bypassed opcode
    logic [`EXCEPTION_WIDTH-1:0] o_exception;  // Propagated exception status
    logic [31:0] o_y;  // ALU computation result
    logic [31:0] o_pc;  // Current PC value
    logic [31:0] o_next_pc;  // Calculated next PC (for jumps/branches)
    logic o_change_pc;  // PC change request (1 = branch/jump taken)
    logic o_wr_rd;  // Write enable for destination register
    logic [4:0] o_rd_addr;  // Destination register address
    logic [31:0] o_rd;  // Data to write to destination register
    logic o_rd_valid;  // Destination register write valid
    logic o_stall_from_alu;  // ALU-generated stall request
    logic o_ce;  // Propagated clock enable
    logic o_stall;  // Combined stall signal output
    logic o_flush;  // Propagated flush signal
```

Figure 6ALU Interface and signal definitions

Interface AI Summary:

This interface is designed to facilitate communication between the RV32I ALU and pipeline stages (e.g., controller and writeback) in a RISC-V processor design.

Interface: alu_if

- **Purpose**:
  Enables structured data and control signal exchange between the ALU and other pipeline components in an RV32I-based RISC-V processor.
- **Clock and Reset**:
  - i_clk: Main clock signal for synchronization.
  - i_rst_n: Active-low asynchronous reset.

Input Signals (Driven by Controller)

These signals are provided to the ALU to specify operations, operands, and control:

- i_alu: ALU operation selection bits (width: ALU_WIDTH).
- i_rs1_addr: Source register 1 address (5 bits).
- i_rs1: Source register 1 value (32 bits).
- i_rs2: Source register 2 value (32 bits).
- i_imm: Immediate value from the instruction (32 bits).
- i_funct3: 3-bit function code from the instruction (3 bits).
- i_opcode: Instruction opcode (width: OPCODE_WIDTH).
- i_exception: Exception status from prior stages (width: EXCEPTION_WIDTH).
- i_pc: Program counter value (32 bits).
- i_rd_addr: Destination register address (5 bits).
- i_ce: Clock enable signal.
- i_stall: Pipeline stall signal from the controller.
- i_force_stall: Debug-specific stall signal.
- i_flush: Pipeline flush signal.

Output Signals (To Writeback Stage)

These signals are generated by the ALU and passed to the writeback stage:

- o_rs1_addr: Bypassed source register 1 address (5 bits).
- o_rs1: Bypassed source register 1 value (32 bits).

- o_rs2: Bypassed source register 2 value (32 bits).
- o_imm: Bypassed immediate value (12 bits).
- o_funct3: Bypassed 3-bit function code (3 bits).
- o_opcode: Bypassed opcode (width: OPCODE_WIDTH).
- o_exception: Propagated exception status (width: EXCEPTION_WIDTH).
- o_y: ALU computation result (32 bits).
- o_pc: Current program counter value (32 bits).
- o_next_pc: Calculated next PC for jumps/branches (32 bits).
- o_change_pc: PC change request (1 = branch/jump taken).
- o_wr_rd: Write enable for the destination register.
- o_rd_addr: Destination register address (5 bits).
- o_rd: Data to write to the destination register (32 bits).
- o_rd_valid: Indicates valid write data for the destination register.
- o_stall_from_alu: Stall request generated by the ALU.
- o_ce: Propagated clock enable signal.
- o_stall: Combined stall signal output.
- o_flush: Propagated flush signal.

Behavior

- **Data Flow**:
  Inputs like operands (i_rs1, i_rs2, i_imm), operation type (i_alu), and instruction details (i_funct3, i_opcode) drive the ALU computation. Outputs include the result (o_y), bypassed data, and control signals for pipeline management.
- **Pipeline Control**:
  Supports stalls (i_stall, o_stall_from_alu), flushes (i_flush, o_flush), and clock gating (i_ce, o_ce).
- **Branch/Jump Handling**:
  Provides PC-related signals (o_pc, o_next_pc, o_change_pc) for control flow changes.
- **Exception Handling**:
  Propagates exception status (i_exception → o_exception) through the pipeline.

Usage

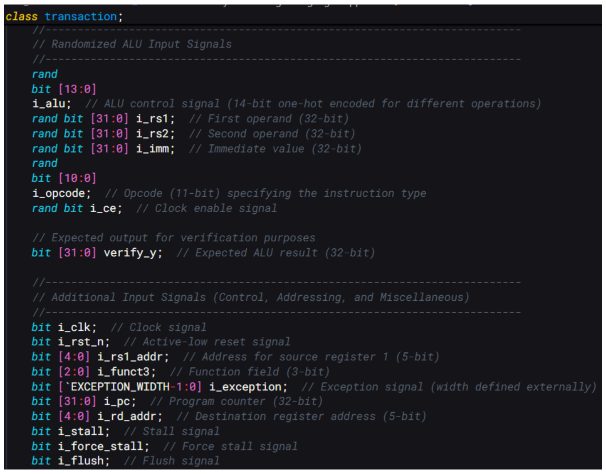
- **Context**:
  Used in a testbench or processor design to connect the ALU module with pipeline stages like the controller and writeback.

- **Verification**:
  Enables precise observation and control of ALU inputs/outputs, supporting functional verification of the RV32I ALU.

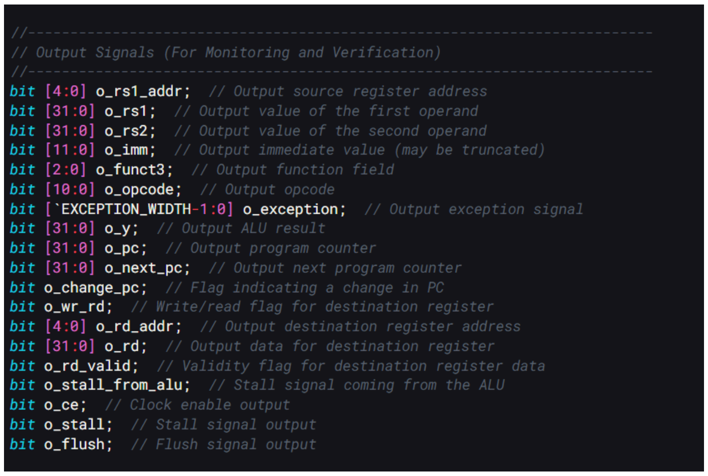
Transaction

The transaction class is used to generate and encapsulate stimulus for a RISC-V ALU verification environment. The mailboxes in the testbench are of this type/instance.

The class contains a set of randomized input signals such as a 14‑bit one-hot encoded ALU control, two 32-bit operands, an immediate value, an 11-bit opcode, and a clock enable signal, all of which model the various inputs to the ALU.



```
class transaction;
    //--------------------------------------------------------------------------
    // Randomized ALU Input Signals
    //--------------------------------------------------------------------------
    rand
    bit [13:0]
    i_alu;  // ALU control signal (14-bit one-hot encoded for different operations)
    rand bit [31:0] i_rs1;  // First operand (32-bit)
    rand bit [31:0] i_rs2;  // Second operand (32-bit)
    rand bit [31:0] i_imm;  // Immediate value (32-bit)
    rand
    bit [10:0]
    i_opcode;  // Opcode (11-bit) specifying the instruction type
    rand bit i_ce;  // Clock enable signal

    // Expected output for verification purposes
    bit [31:0] verify_y;  // Expected ALU result (32-bit)

    //--------------------------------------------------------------------------
    // Additional Input Signals (Control, Addressing, and Miscellaneous)
    //--------------------------------------------------------------------------
    bit i_clk;  // Clock signal
    bit i_rst_n;  // Active-low reset signal
    bit [4:0] i_rs1_addr;  // Address for source register 1 (5-bit)
    bit [2:0] i_funct3;  // Function field (3-bit)
    bit [`EXCEPTION_WIDTH-1:0] i_exception;  // Exception signal (width defined externally)
    bit [31:0] i_pc;  // Program counter (32-bit)
    bit [4:0] i_rd_addr;  // Destination register address (5-bit)
    bit i_stall;  // Stall signal
    bit i_force_stall;  // Force stall signal
    bit i_flush;  // Flush signal
```

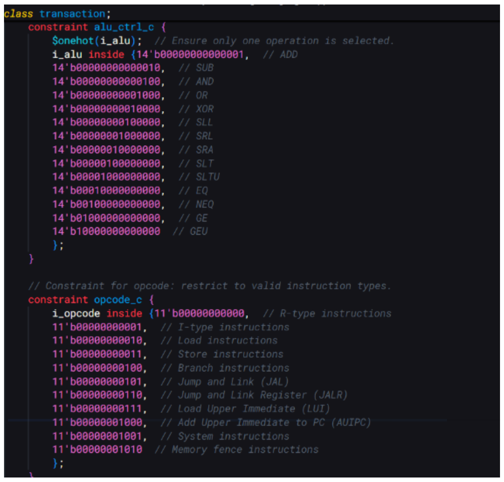


```
//--------------------------------------------------------------------------
// Output Signals (For Monitoring and Verification)
//--------------------------------------------------------------------------
bit [4:0] o_rs1_addr;  // Output source register address
bit [31:0] o_rs1;  // Output value of the first operand
bit [31:0] o_rs2;  // Output value of the second operand
bit [11:0] o_imm;  // Output immediate value (may be truncated)
bit [2:0] o_funct3;  // Output function field
bit [10:0] o_opcode;  // Output opcode
bit [`EXCEPTION_WIDTH-1:0] o_exception;  // Output exception signal
bit [31:0] o_y;  // Output ALU result
bit [31:0] o_pc;  // Output program counter
bit [31:0] o_next_pc;  // Output next program counter
bit o_change_pc;  // Flag indicating a change in PC
bit o_wr_rd;  // Write/read flag for destination register
bit [4:0] o_rd_addr;  // Output destination register address
bit [31:0] o_rd;  // Output data for destination register
bit o_rd_valid;  // Validity flag for destination register data
bit o_stall_from_alu;  // Stall signal coming from the ALU
bit o_ce;  // Clock enable output
bit o_stall;  // Stall signal output
bit o_flush;  // Flush signal output
```

Figure 8 Signals to be randomized

In addition to these randomized inputs, the class holds signals for expected output verification, including the expected ALU result, and many other signals that simulate control, addressing, and miscellaneous behaviors such as clock, reset, register addresses, function codes, exception signals, program counter, stall, and flush signals.

The code also defines constraints that govern the randomization process.



```
class transaction;
    constraint alu_ctrl_c {
        $onehot(i_alu);  // Ensure only one operation is selected.
        i_alu inside {14'b00000000000001,  // ADD
        14'b00000000000010,  // SUB
        14'b00000000000100,  // AND
        14'b00000000001000,  // OR
        14'b00000000010000,  // XOR
        14'b00000000100000,  // SLL
        14'b00000001000000,  // SRL
        14'b00000010000000,  // SRA
        14'b00000100000000,  // SLT
        14'b00001000000000,  // SLTU
        14'b00010000000000,  // EQ
        14'b00100000000000,  // NEQ
        14'b01000000000000,  // GE
        14'b10000000000000  // GEU
        };
    }

    // Constraint for opcode: restrict to valid instruction types.
    constraint opcode_c {
        i_opcode inside {11'b00000000000,  // R-type instructions
        11'b00000000001,  // I-type instructions
        11'b00000000010,  // Load instructions
        11'b00000000011,  // Store instructions
        11'b00000000100,  // Branch instructions
        11'b00000000101,  // Jump and Link (JAL)
        11'b00000000110,  // Jump and Link Register (JALR)
        11'b00000000111,  // Load Upper Immediate (LUI)
        11'b00000001000,  // Add Upper Immediate to PC (AUIPC)
        11'b00000001001,  // System instructions
        11'b00000001010  // Memory fence instructions
        };
```

One constraint ensures that the ALU control signal is one-hot encoded and only takes one of the predetermined 14-bit values, each corresponding to a specific operation like ADD, SUB, AND, OR, XOR, various shift operations, and comparison operations.

There is also a constraint to limit the opcode to a valid set representing different instruction types, and a distribution constraint that biases the clock enable signal to be active most of the time.

Additional optional constraints force at least one operand to be either zero or its maximum value, or to fall near a sign boundary, which is useful for stress-testing sign-related behavior.

The constructor of the class initializes all input and output signals to their default values. There is a helper function that allows the user to set specific values for the ALU operation by clearing any previous ALU control setting, selecting the appropriate bit for the desired operation, and assigning the corresponding operands and opcode. Another function computes the expected result of the ALU operation based on the current operands, immediate, and ALU control, using a case statement that matches each one-hot encoded operation to its corresponding arithmetic or logical function.

Finally, a cloning function is provided to create an exact copy of a transaction, duplicating its key input values and expected result, which is useful for maintaining stimulus consistency during verification runs.

Transaction AI Summary:

Class: transaction

- **Purpose**: Represents a data packet for RV32I ALU verification, containing randomized inputs, control signals, expected results, and output signals for stimulus generation and monitoring.
- **Key Features**:
    1. **Randomized Inputs**:
        - i_alu (14-bit): One-hot encoded ALU control (e.g., ADD, SUB).
        - i_rs1, i_rs2, i_imm (32-bit): Operands and immediate value.
        - i_opcode (11-bit): Instruction type.

    - i_ce: Clock enable.

  2. **Expected Output**:

    - verify_y (32-bit): Expected ALU result for verification.

  3. **Additional Inputs**:

    - Control signals (e.g., i_clk, i_rst_n, i_stall) and addressing (e.g., i_rs1_addr, i_rd_addr).

  4. **Output Signals**:

    - ALU outputs (e.g., o_y, o_rd) and control (e.g., o_ce, o_rd_valid).

  5. **Constraints**:

    - alu_ctrl_c: Enforces one-hot i_alu for valid operations (e.g., ADD, XOR).
    - opcode_c: Restricts i_opcode to valid RV32I types (e.g., R-type, JAL).
    - ce_c: i_ce is 90% enabled.
    - operand_ranges: Full 32-bit range for operands.
    - Optional: zero_operand_c, max_value_c, sign_boundary_c for specific test cases.

- **Methods**:

  1. **Constructor (**new**)**:

    - Initializes all signals to 0 (or 1 for i_rst_n).

  2. set_values:

    - Configures inputs (e.g., i_alu, i_rs1) for a specific operation.

  3. alu_operation:

    - Computes verify_y based on i_alu, operands, and i_opcode (e.g., ADD, SLT).

  4. clone:

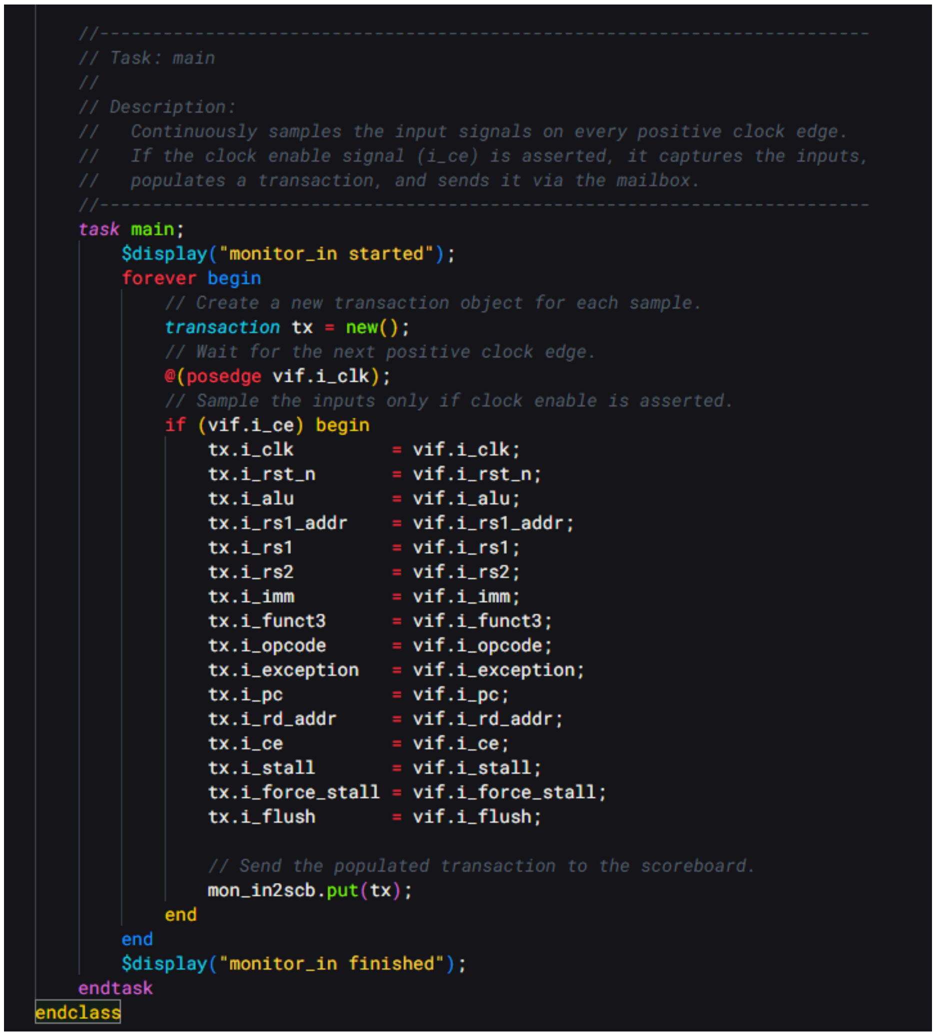
    - Returns a copy of the transaction with input values preserved.

- **Behavior**: Generates constrained-random stimuli for ALU testing, computes expected results, and supports input/output monitoring in a testbench.

Monitor

We designed two monitor classes for the verification of the RISC-V ALU.

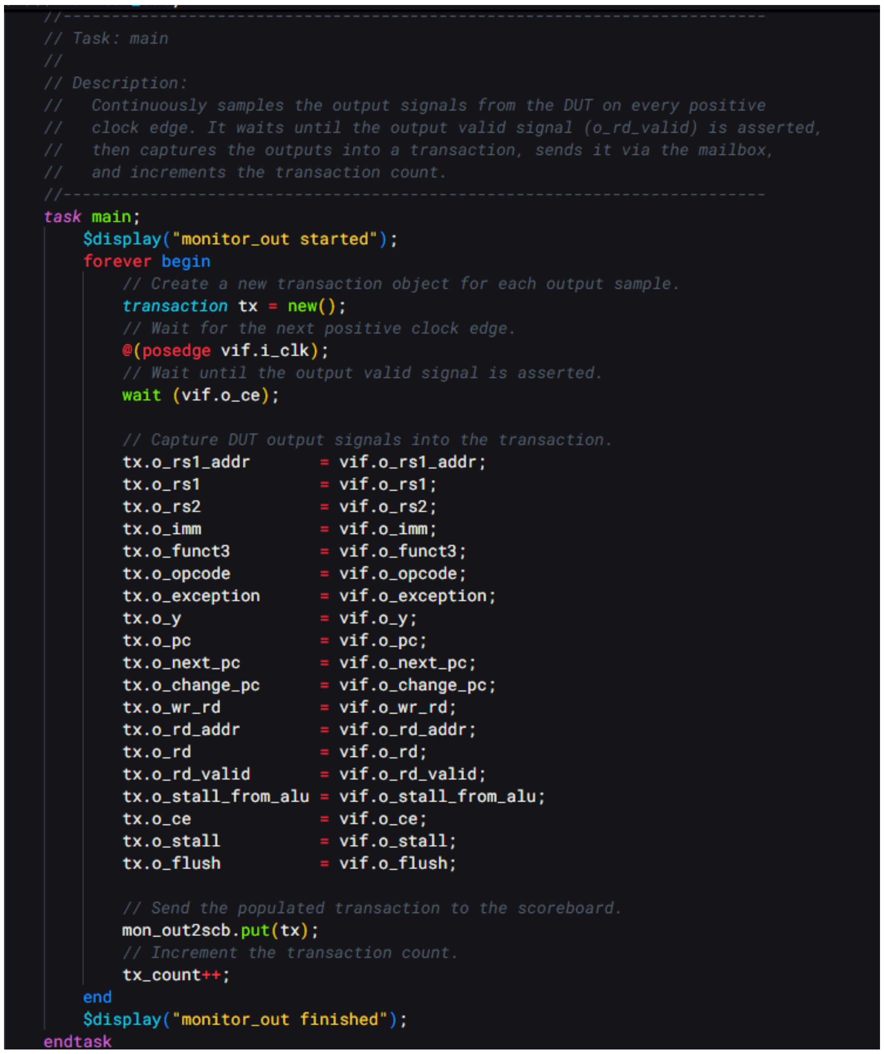
The first class, `monitor_in`, is responsible for sampling input signals from the interface we implemented. It operates by continuously monitoring the clock signal and, when the clock enable is asserted, captures all relevant input signals such as reset, ALU controls, register addresses, immediate values, and control flags. These captured values are packaged into transaction objects and sent to a scoreboard for verification via a transaction type mailbox.



```
    //----------------------------------------------------------------------
    // Task: main
    //
    // Description:
    //   Continuously samples the input signals on every positive clock edge.
    //   If the clock enable signal (i_ce) is asserted, it captures the inputs,
    //   populates a transaction, and sends it via the mailbox.
    //----------------------------------------------------------------------
    task main;
        $display("monitor_in started");
        forever begin
            // Create a new transaction object for each sample.
            transaction tx = new();
            // Wait for the next positive clock edge.
            @(posedge vif.i_clk);
            // Sample the inputs only if clock enable is asserted.
            if (vif.i_ce) begin
                tx.i_clk         = vif.i_clk;
                tx.i_rst_n       = vif.i_rst_n;
                tx.i_alu         = vif.i_alu;
                tx.i_rs1_addr    = vif.i_rs1_addr;
                tx.i_rs1         = vif.i_rs1;
                tx.i_rs2         = vif.i_rs2;
                tx.i_imm         = vif.i_imm;
                tx.i_funct3      = vif.i_funct3;
                tx.i_opcode      = vif.i_opcode;
                tx.i_exception   = vif.i_exception;
                tx.i_pc          = vif.i_pc;
                tx.i_rd_addr     = vif.i_rd_addr;
                tx.i_ce          = vif.i_ce;
                tx.i_stall       = vif.i_stall;
                tx.i_force_stall = vif.i_force_stall;
                tx.i_flush       = vif.i_flush;

                // Send the populated transaction to the scoreboard.
                mon_in2scb.put(tx);
            end
        end
        $display("monitor_in finished");
    endtask
endclass
```

Figure 10 monitor_in class' main task

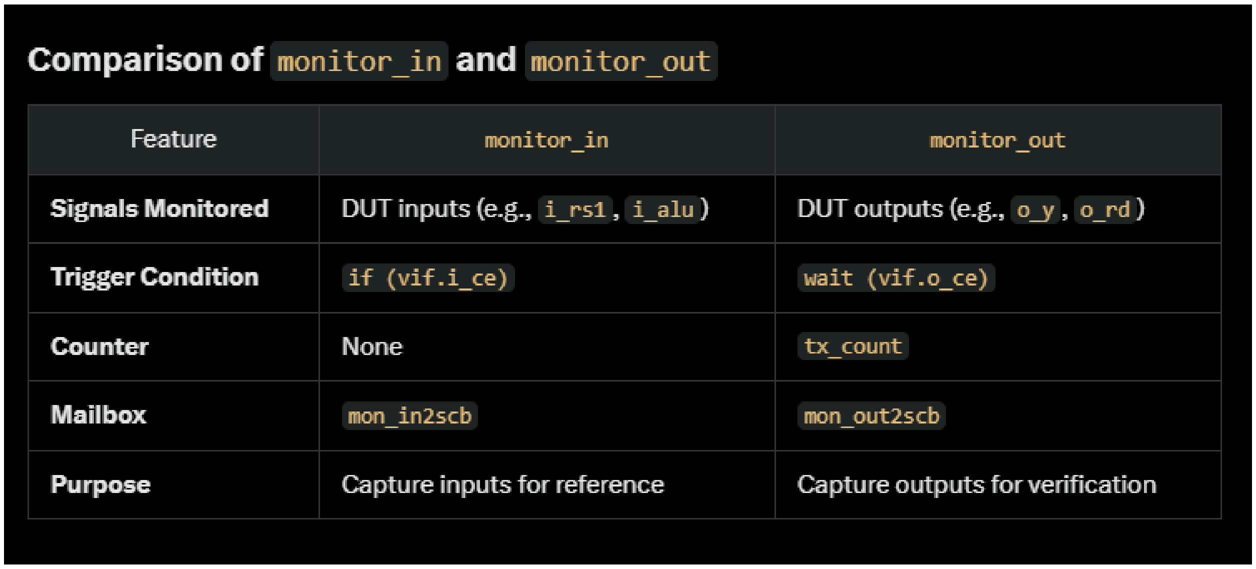
The second class, `monitor_out`, serves a complementary role by monitoring the DUT's output signals. It maintains a count of processed transactions and waits for the output valid signal (o_ce) before capturing the output data. When valid output is detected, it samples various signals including register values, computation results, program counter values, and control flags. Like its input counterpart, it packages these outputs into transaction objects and sends them to the scoreboard through a dedicated mailbox.



```
//-----------------------------------------------------------------------
// Task: main
//
// Description:
//   Continuously samples the output signals from the DUT on every positive
//   clock edge. It waits until the output valid signal (o_rd_valid) is asserted,
//   then captures the outputs into a transaction, sends it via the mailbox,
//   and increments the transaction count.
//-----------------------------------------------------------------------
task main;
    $display("monitor_out started");
    forever begin
        // Create a new transaction object for each output sample.
        transaction tx = new();
        // Wait for the next positive clock edge.
        @(posedge vif.i_clk);
        // Wait until the output valid signal is asserted.
        wait (vif.o_ce);

        // Capture DUT output signals into the transaction.
        tx.o_rs1_addr       = vif.o_rs1_addr;
        tx.o_rs1            = vif.o_rs1;
        tx.o_rs2            = vif.o_rs2;
        tx.o_imm            = vif.o_imm;
        tx.o_funct3         = vif.o_funct3;
        tx.o_opcode         = vif.o_opcode;
        tx.o_exception      = vif.o_exception;
        tx.o_y              = vif.o_y;
        tx.o_pc             = vif.o_pc;
        tx.o_next_pc        = vif.o_next_pc;
        tx.o_change_pc      = vif.o_change_pc;
        tx.o_wr_rd          = vif.o_wr_rd;
        tx.o_rd_addr        = vif.o_rd_addr;
        tx.o_rd             = vif.o_rd;
        tx.o_rd_valid       = vif.o_rd_valid;
        tx.o_stall_from_alu = vif.o_stall_from_alu;
        tx.o_ce             = vif.o_ce;
        tx.o_stall          = vif.o_stall;
        tx.o_flush          = vif.o_flush;

        // Send the populated transaction to the scoreboard.
        mon_out2scb.put(tx);
        // Increment the transaction count.
        tx_count++;
    end
    $display("monitor_out finished");
endtask
```

Figure 11 monitor_out class' main task

Both monitors are essential components of the verification environment, working together to ensure the ALU's behavior matches its specification. They use virtual interfaces to access the DUT signals and implement a standard constructor pattern for initialization.



Comparison of `monitor_in` and `monitor_out`

Feature	`monitor_in`	`monitor_out`
Signals Monitored	DUT inputs (e.g., `i_rs1`, `i_alu`)	DUT outputs (e.g., `o_y`, `o_rd`)
Trigger Condition	`if (vif.i_ce)`	`wait (vif.o_ce)`
Counter	None	`tx_count`
Mailbox	`mon_in2scb`	`mon_out2scb`
Purpose	Capture inputs for reference	Capture outputs for verification

Figure 12 Comparison of monitor_in and monitor_out

Monitor AI Summary:

Class: monitor_in

- **Purpose**: Samples DUT input signals via a virtual interface (vif), creates transaction objects, and sends them to the scoreboard through a mailbox (mon_in2scb).
- **Key Features**:
  - Uses virtual alu_if to access inputs (e.g., i_clk, i_rs1, i_alu).
  - Constructor initializes vif and mon_in2scb.
  - main task: Runs forever, samples on posedge i_clk when i_ce is high, populates transaction, and sends it via mailbox.
- **Behavior**: Synchronous, conditional sampling of RV32I ALU inputs.

Class: monitor_out

- **Purpose**: Samples DUT output signals via vif, creates transaction objects, sends them to the scoreboard via mon_out2scb, and tracks count with tx_count.
- **Key Features**:
  - Uses virtual alu_if for outputs (e.g., o_y, o_rd, o_rd_valid).
  - Constructor initializes vif and mon_out2scb.
  - main task: Runs forever, samples on posedge i_clk when o_ce is high, populates transaction, sends it, and increments tx_count.
- **Behavior**: Synchronous, waits for valid outputs, tracks transactions.

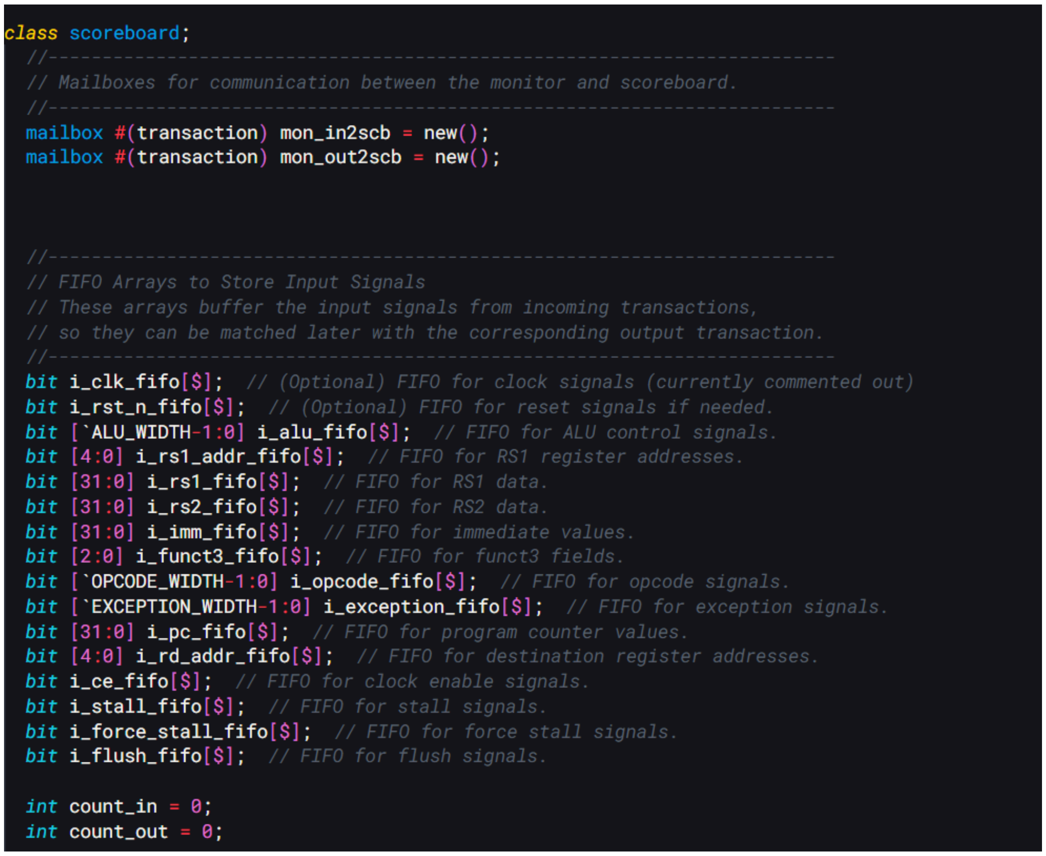
Overview

- **Context**: Part of an RV32I ALU testbench; monitor_in captures inputs, monitor_out captures outputs for scoreboard verification.
- **Operation**: Both use mailboxes for asynchronous communication and run continuously, aligned with clock edges.

Scoreboard

The scoreboard functions as a checker that compares the ALU's outputs with the expected results derived from input transactions.

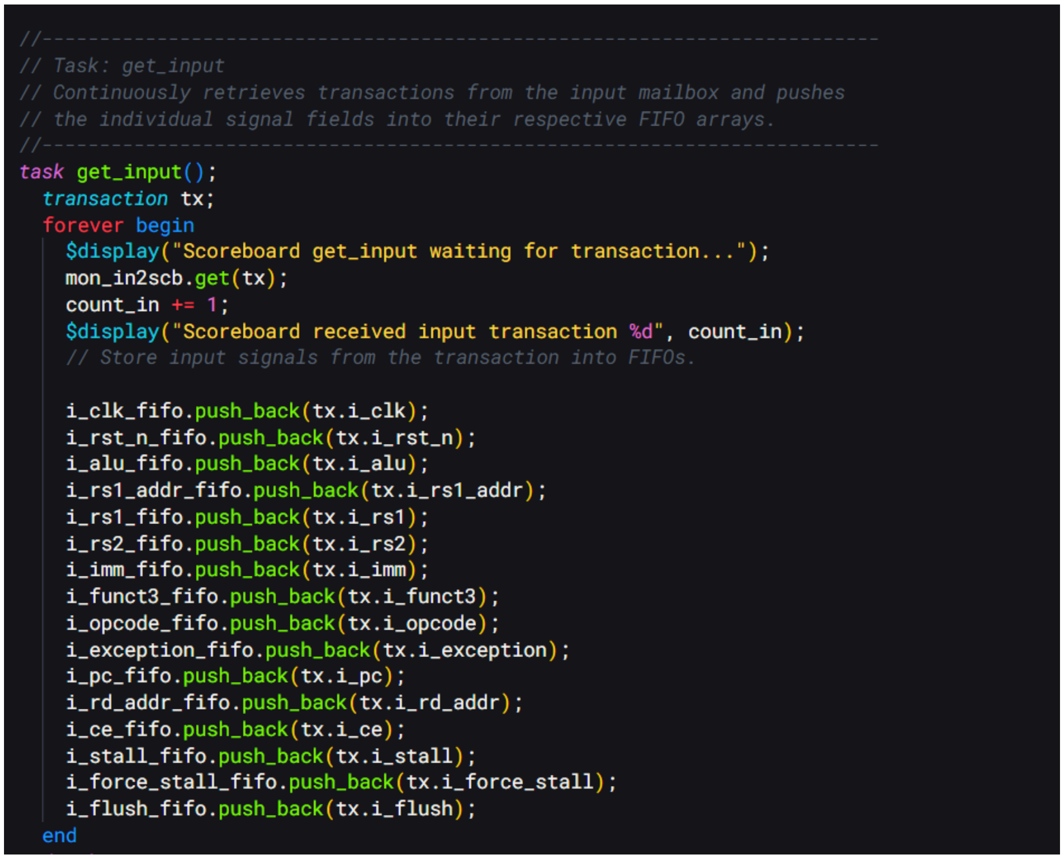
The scoreboard receives two streams of transactions via mailboxes: one carrying the inputs and the other the outputs.



```
class scoreboard;
  //----------------------------------------------------------------------
  // Mailboxes for communication between the monitor and scoreboard.
  //----------------------------------------------------------------------
  mailbox #(transaction) mon_in2scb = new();
  mailbox #(transaction) mon_out2scb = new();



  //----------------------------------------------------------------------
  // FIFO Arrays to Store Input Signals
  // These arrays buffer the input signals from incoming transactions,
  // so they can be matched later with the corresponding output transaction.
  //----------------------------------------------------------------------
  bit i_clk_fifo[$];  // (Optional) FIFO for clock signals (currently commented out)
  bit i_rst_n_fifo[$];  // (Optional) FIFO for reset signals if needed.
  bit [`ALU_WIDTH-1:0] i_alu_fifo[$];  // FIFO for ALU control signals.
  bit [4:0] i_rs1_addr_fifo[$];  // FIFO for RS1 register addresses.
  bit [31:0] i_rs1_fifo[$];  // FIFO for RS1 data.
  bit [31:0] i_rs2_fifo[$];  // FIFO for RS2 data.
  bit [31:0] i_imm_fifo[$];  // FIFO for immediate values.
  bit [2:0] i_funct3_fifo[$];  // FIFO for funct3 fields.
  bit [`OPCODE_WIDTH-1:0] i_opcode_fifo[$];  // FIFO for opcode signals.
  bit [`EXCEPTION_WIDTH-1:0] i_exception_fifo[$];  // FIFO for exception signals.
  bit [31:0] i_pc_fifo[$];  // FIFO for program counter values.
  bit [4:0] i_rd_addr_fifo[$];  // FIFO for destination register addresses.
  bit i_ce_fifo[$];  // FIFO for clock enable signals.
  bit i_stall_fifo[$];  // FIFO for stall signals.
  bit i_force_stall_fifo[$];  // FIFO for force stall signals.
  bit i_flush_fifo[$];  // FIFO for flush signals.

  int count_in = 0;
  int count_out = 0;
```

Figure 13 Scoreboard mailboxes and FIFO Arrays

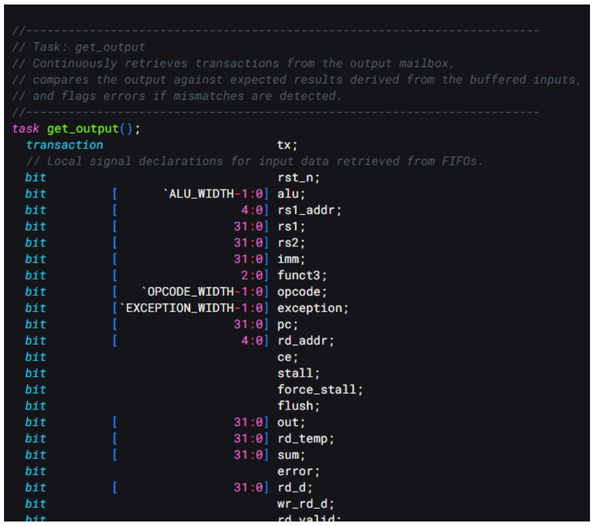
Each input transaction, which contains all the signal fields such as clock, reset, ALU control, register addresses and data, immediate values, function codes, opcodes, exceptions, and other control signals, is stored in corresponding FIFO arrays.



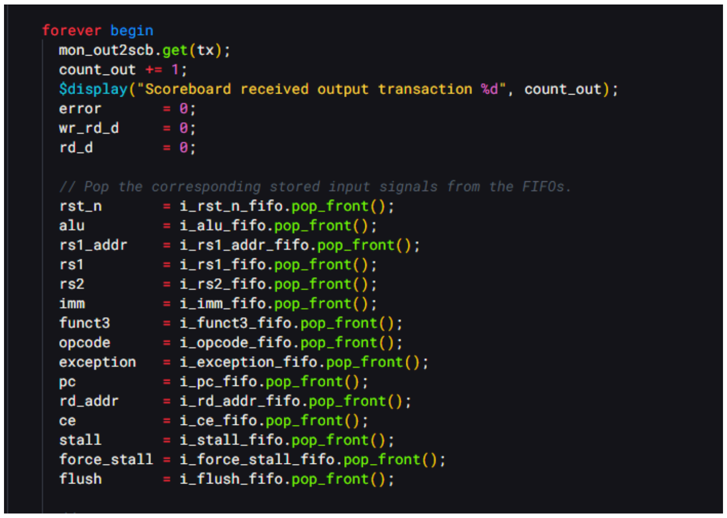
```
//-----------------------------------------------------------------------
// Task: get_input
// Continuously retrieves transactions from the input mailbox and pushes
// the individual signal fields into their respective FIFO arrays.
//-----------------------------------------------------------------------
task get_input();
  transaction tx;
  forever begin
    $display("Scoreboard get_input waiting for transaction...");
    mon_in2scb.get(tx);
    count_in += 1;
    $display("Scoreboard received input transaction %d", count_in);
    // Store input signals from the transaction into FIFOs.

    i_clk_fifo.push_back(tx.i_clk);
    i_rst_n_fifo.push_back(tx.i_rst_n);
    i_alu_fifo.push_back(tx.i_alu);
    i_rs1_addr_fifo.push_back(tx.i_rs1_addr);
    i_rs1_fifo.push_back(tx.i_rs1);
    i_rs2_fifo.push_back(tx.i_rs2);
    i_imm_fifo.push_back(tx.i_imm);
    i_funct3_fifo.push_back(tx.i_funct3);
    i_opcode_fifo.push_back(tx.i_opcode);
    i_exception_fifo.push_back(tx.i_exception);
    i_pc_fifo.push_back(tx.i_pc);
    i_rd_addr_fifo.push_back(tx.i_rd_addr);
    i_ce_fifo.push_back(tx.i_ce);
    i_stall_fifo.push_back(tx.i_stall);
    i_force_stall_fifo.push_back(tx.i_force_stall);
    i_flush_fifo.push_back(tx.i_flush);
  end
```

Figure 14 Scoreboard getting input taks

Meanwhile, the output transactions are fetched, and for each one, the scoreboard pops the associated input values from the FIFO arrays.



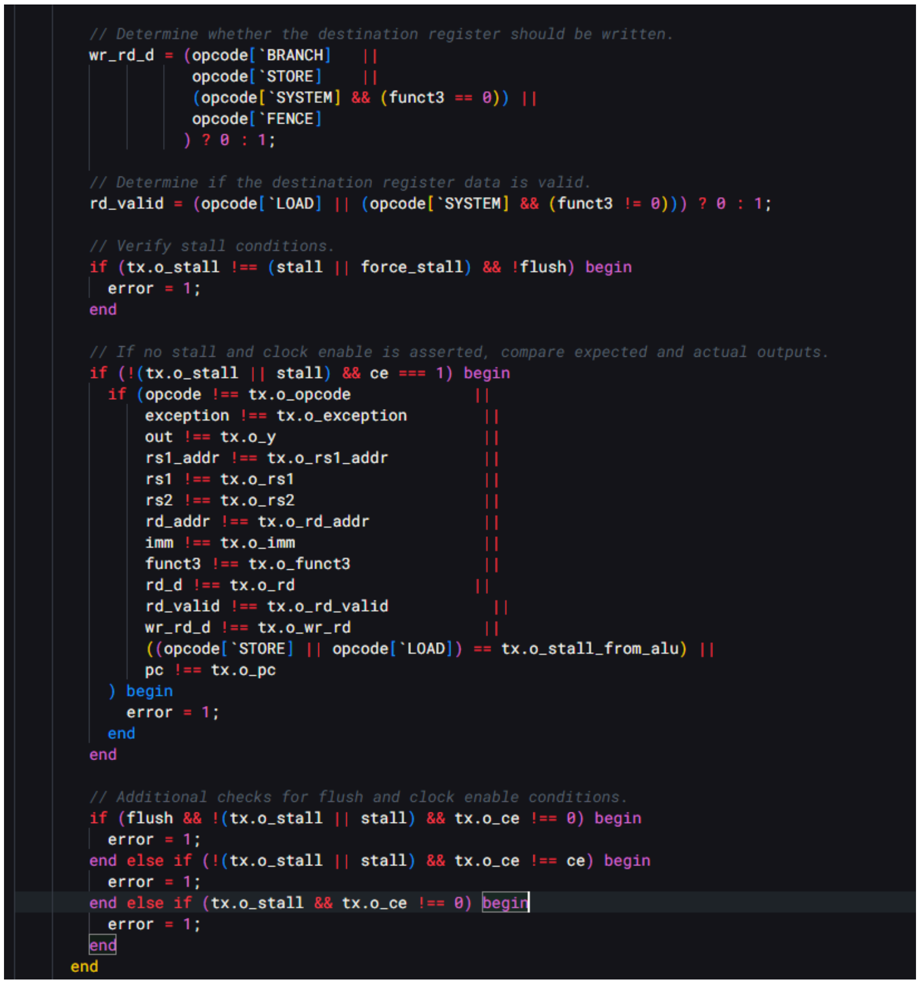
```
//-----------------------------------------------------------------------
// Task: get_output
// Continuously retrieves transactions from the output mailbox,
// compares the output against expected results derived from the buffered inputs,
// and flags errors if mismatches are detected.
//-----------------------------------------------------------------------
task get_output();
  transaction                         tx;
  // Local signal declarations for input data retrieved from FIFOs.
  bit                                 rst_n;
  bit         [       `ALU_WIDTH-1:0] alu;
  bit         [                  4:0] rs1_addr;
  bit         [                 31:0] rs1;
  bit         [                 31:0] rs2;
  bit         [                 31:0] imm;
  bit         [                  2:0] funct3;
  bit         [    `OPCODE_WIDTH-1:0] opcode;
  bit         [`EXCEPTION_WIDTH-1:0] exception;
  bit         [                 31:0] pc;
  bit         [                  4:0] rd_addr;
  bit                                 ce;
  bit                                 stall;
  bit                                 force_stall;
  bit                                 flush;
  bit         [                 31:0] out;
  bit         [                 31:0] rd_temp;
  bit         [                 31:0] sum;
  bit                                 error;
  bit         [                 31:0] rd_d;
  bit                                 wr_rd_d;
  bit                                 rd_valid;
```



```
  forever begin
    mon_out2scb.get(tx);
    count_out += 1;
    $display("Scoreboard received output transaction %d", count_out);
    error       = 0;
    wr_rd_d     = 0;
    rd_d        = 0;

    // Pop the corresponding stored input signals from the FIFOs.
    rst_n       = i_rst_n_fifo.pop_front();
    alu         = i_alu_fifo.pop_front();
    rs1_addr    = i_rs1_addr_fifo.pop_front();
    rs1         = i_rs1_fifo.pop_front();
    rs2         = i_rs2_fifo.pop_front();
    imm         = i_imm_fifo.pop_front();
    funct3      = i_funct3_fifo.pop_front();
    opcode      = i_opcode_fifo.pop_front();
    exception   = i_exception_fifo.pop_front();
    pc          = i_pc_fifo.pop_front();
    rd_addr     = i_rd_addr_fifo.pop_front();
    ce          = i_ce_fifo.pop_front();
    stall       = i_stall_fifo.pop_front();
    force_stall = i_force_stall_fifo.pop_front();
    flush       = i_flush_fifo.pop_front();
```

Figure 16 Scoreboard getting output task

It then computes the expected ALU result using a dedicated function that interprets a one-hot encoded operation signal to decide between operations like addition, subtraction, logical comparisons, bitwise operations, and various shift operations.



```
    // Determine whether the destination register should be written.
    wr_rd_d = (opcode[`BRANCH]   ||
               opcode[`STORE]    ||
               (opcode[`SYSTEM] && (funct3 == 0)) ||
               opcode[`FENCE]
              ) ? 0 : 1;

    // Determine if the destination register data is valid.
    rd_valid = (opcode[`LOAD] || (opcode[`SYSTEM] && (funct3 != 0))) ? 0 : 1;

    // Verify stall conditions.
    if (tx.o_stall !== (stall || force_stall) && !flush) begin
      error = 1;
    end

    // If no stall and clock enable is asserted, compare expected and actual outputs.
    if (!(tx.o_stall || stall) && ce === 1) begin
      if (opcode !== tx.o_opcode            ||
          exception !== tx.o_exception       ||
          out !== tx.o_y                     ||
          rs1_addr !== tx.o_rs1_addr         ||
          rs1 !== tx.o_rs1                   ||
          rs2 !== tx.o_rs2                   ||
          rd_addr !== tx.o_rd_addr           ||
          imm !== tx.o_imm                   ||
          funct3 !== tx.o_funct3             ||
          rd_d !== tx.o_rd                  ||
          rd_valid !== tx.o_rd_valid          ||
          wr_rd_d !== tx.o_wr_rd             ||
          ((opcode[`STORE] || opcode[`LOAD]) == tx.o_stall_from_alu) ||
          pc !== tx.o_pc
      ) begin
        error = 1;
      end
    end

    // Additional checks for flush and clock enable conditions.
    if (flush && !(tx.o_stall || stall) && tx.o_ce !== 0) begin
      error = 1;
    end else if (!(tx.o_stall || stall) && tx.o_ce !== ce) begin
      error = 1;
    end else if (tx.o_stall && tx.o_ce !== 0) begin
      error = 1;
    end
  end
```

Figure 17 Scoreboard handling different instruction types

The code includes logic to handle different instruction types, such as R-type, I-type, branch, jump, LUI, and AUIPC instructions, and also checks conditions related to stalls, flushes, and resets.

Scoreboard AI Summary:

Overview of the scoreboard **Class**

Purpose

The scoreboard class is a critical verification component designed to ensure the correctness of the RV32I ALU's outputs. It achieves this by:

Collecting input transactions from the DUT (Device Under Test) via a monitor.

Storing these inputs in FIFO (First-In-First-Out) buffers.

Retrieving corresponding output transactions from the DUT.

Comparing the actual outputs against expected results computed from the buffered inputs.

The scoreboard handles a variety of conditions, including normal ALU operations, reset states, stalls, flushes, and instruction-specific behaviors (e.g., branches, jumps), and provides detailed error reporting when discrepancies are detected.

Key Components

Mailboxes

mon_in2scb: A mailbox that receives input transactions from the monitor capturing the DUT's input signals.

mon_out2scb: A mailbox that receives output transactions from the monitor capturing the DUT's output signals.

Role: These mailboxes enable communication between the scoreboard and the monitors, ensuring that input and output data are properly synchronized for verification.

FIFO Arrays

The scoreboard uses FIFO arrays to buffer individual signals from input transactions until their corresponding outputs are available for comparison. Each FIFO corresponds to a specific input signal:

i_clk_fifo[$]: Stores clock signals (currently optional and could be commented out).

i_rst_n_fifo[$]: Stores reset signals (rst_n).

i_alu_fifo[$]: Stores ALU control signals (one-hot encoded, width: ALU_WIDTH).

i_rs1_addr_fifo[$]: Stores RS1 register addresses (5 bits).

i_rs1_fifo[$]: Stores RS1 data (32 bits).

i_rs2_fifo[$]: Stores RS2 data (32 bits).

i_imm_fifo[$]: Stores immediate values (32 bits).

i_funct3_fifo[$]: Stores funct3 fields (3 bits).

i_opcode_fifo[$]: Stores opcode signals (width: OPCODE_WIDTH).

i_exception_fifo[$]: Stores exception signals (width: EXCEPTION_WIDTH).

i_pc_fifo[$]: Stores program counter values (32 bits).

i_rd_addr_fifo[$]: Stores destination register addresses (5 bits).

i_ce_fifo[$]: Stores clock enable signals.

i_stall_fifo[$]: Stores stall signals.

i_force_stall_fifo[$]: Stores forced stall signals.

i_flush_fifo[$]: Stores flush signals.

Counters

count_in: Tracks the number of input transactions received.

count_out: Tracks the number of output transactions processed.

Role: These counters help monitor the progress of verification and ensure that inputs and outputs are processed in sync.

Class Methods and Tasks

Constructor

Signature: function new(mailbox#(transaction) mon_in2scb, mailbox#(transaction) mon_out2scb)

Functionality:

Initializes the scoreboard with references to the input and output mailboxes (mon_in2scb and mon_out2scb).

Establishes the communication pathways for receiving transactions from the monitors.

Task: main

Purpose: Launches the primary concurrent operations of the scoreboard.

Implementation:

```
systemverilog

task main;

  fork

    get_input();  // Collect input transactions

    get_output(); // Collect and verify output transactions

  join_none

endtask
```

Behavior: Uses a fork block with join_none to run get_input() and get_output() in parallel, allowing continuous and non-blocking processing of input and output transactions.

Task: get_input

Purpose: Continuously retrieves input transactions from mon_in2scb and buffers their signals in the FIFOs.

Implementation:

Waits for a transaction using mon_in2scb.get(tx).

Increments count_in.

Pushes each signal from the transaction (tx) into its corresponding FIFO (e.g., i_alu_fifo.push_back(tx.i_alu)).

Key Steps:

Logs receipt of the transaction with a display message.

Stores all input signals (e.g., i_clk, i_rst_n, i_alu, etc.) into their respective FIFOs.

Behavior: Ensures that all input data is preserved in order until the corresponding output is available for comparison.

Task: get_output

Purpose: Retrieves output transactions from mon_out2scb, computes expected results from buffered inputs, and verifies the DUT outputs.

Implementation:

Waits for a transaction using mon_out2scb.get(tx).

Increments count_out.

Pops input signals from the FIFOs (e.g., rst_n = i_rst_n_fifo.pop_front()).

Performs detailed verification based on the opcode, ALU operation, and control signals.

Key Verification Steps:

Reset Check: If rst_n == 0, ensures o_exception, o_ce, and o_stall_from_alu are 0. Flags an error if not.

ALU Operation: Calls alu_operation to compute the expected result (out) based on:

First operand: pc for JAL/AUIPC, otherwise rs1.

Second operand: rs2 for RTYPE/BRANCH, otherwise imm.

ALU control signal: alu.

Instruction-Specific Checks:

Branches: Verifies o_next_pc, o_change_pc, and o_flush match expected values when out is true.

Jumps (JAL/JALR): Computes sum (e.g., rs1 + imm for JALR) and checks o_next_pc, o_change_pc, and o_flush. Sets rd_d = pc + 4.

LUI: Sets rd_d = imm.

AUIPC: Sets rd_d = pc + imm.

Register Writeback: Determines wr_rd_d (write enable) and rd_valid based on the opcode (e.g., 0 for BRANCH, STORE).

Stall/Flush Handling: Verifies o_stall matches (stall || force_stall) unless flushed.

Output Comparison: When no stall and ce == 1, compares actual outputs (e.g., o_y, o_rd, o_rd_valid) with expected values (out, rd_d, etc.).

Error Reporting: If any mismatch occurs (error == 1), prints a detailed log of all input and output signals for debugging.

Function: alu_operation

Signature: function automatic logic [31:0] alu_operation(input logic [31:0] a, input logic [31:0] b, input logic [ALU_WIDTH-1:0] op)`

Purpose: Computes the expected ALU result based on operands a and b and the one-hot encoded operation op.

Implementation:

Uses $clog2(op) to identify the active operation (assuming one-hot encoding).

Supports operations like:

0: a + b (ADD)

1: a - b (SUB)

2: (a < b) ? 1 : 0 (SLT)

3: (unsigned'a < unsigned'b) ? 1 : 0 (SLTU)

4: a ^ b (XOR)

5: a | b (OR)

6: a & b (AND)

7: a << b[4:0] (SLL)

8: a >> b[4:0] (SRL)

9: $signed(a) >>> b[4:0] (SRA)

10: (a == b) ? 1 : 0 (EQ)

11: (a != b) ? 1 : 0 (NEQ)

12: (a >= b) ? 1 : 0 (GE)

13: (unsigned’a >= unsigned’b) ? 1 : 0 (GEU)

Default: Returns 0 for invalid operations.

Behavior: Provides a reference model for ALU operations, enabling the scoreboard to predict correct outputs.

Key Behaviors

Input Buffering: FIFOs store input signals in sequence, aligning them with pipelined outputs for accurate comparison.

Error Detection: Comprehensive checks cover ALU results, control signals, and pipeline states, flagging any deviations.

Reset Handling: Ensures proper reset behavior by validating control signals during rst_n == 0.

Instruction-Specific Logic: Adapts verification logic to handle different RV32I instruction types (e.g., arithmetic, branches, jumps).

Stall and Flush Management: Verifies correct propagation of stall and flush signals, ensuring pipeline integrity.

Interaction with Other Components

Monitors: Receives input and output transactions via mon_in2scb and mon_out2scb.

Driver and Generator: Indirectly verifies the DUT’s response to inputs generated and driven by these components.

Coverage: Complements coverage analysis by confirming that exercised scenarios produce correct results.

Coverage

Coverage primarily focuses on functional coverage data for an ALU design.

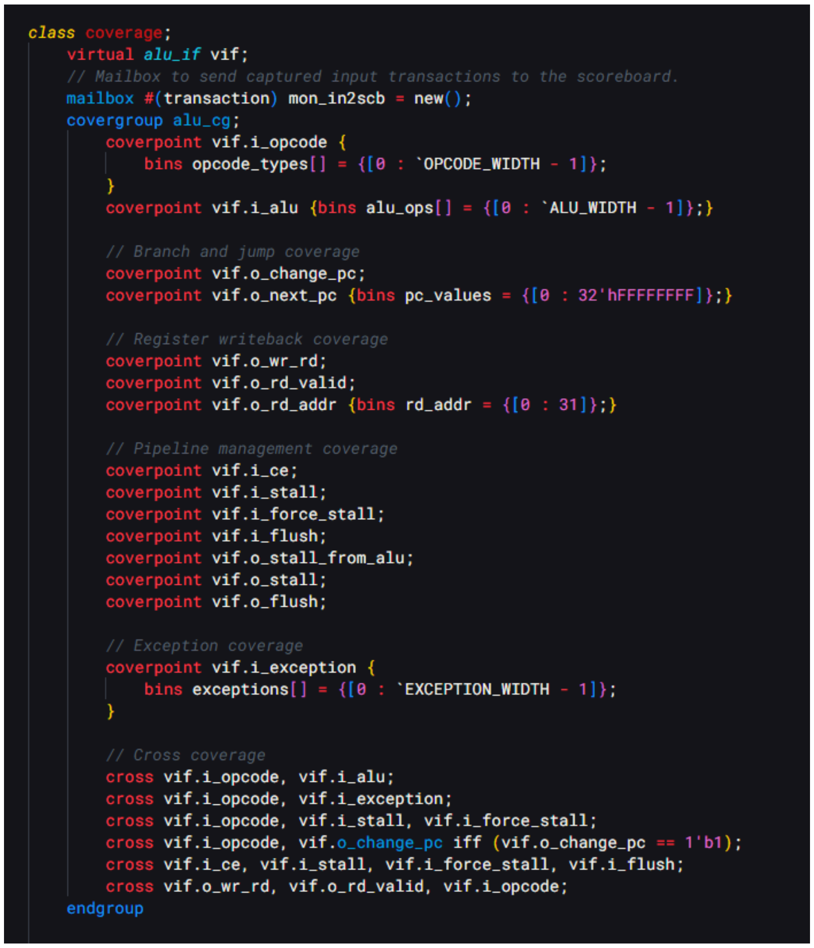
The intent is as we progress to add more coverage and to get at 100% for code coverage.

The class uses the custom virtual interface (inteface.sv) to sample different signals from the design under test.

It sets up a covergroup that monitors a variety of signals, including those related to ALU operations like the opcode and ALU control, branch and jump behavior such as the next program counter and control signals, as well as signals used for managing pipeline behavior like stalls, flushes, and clock enables.

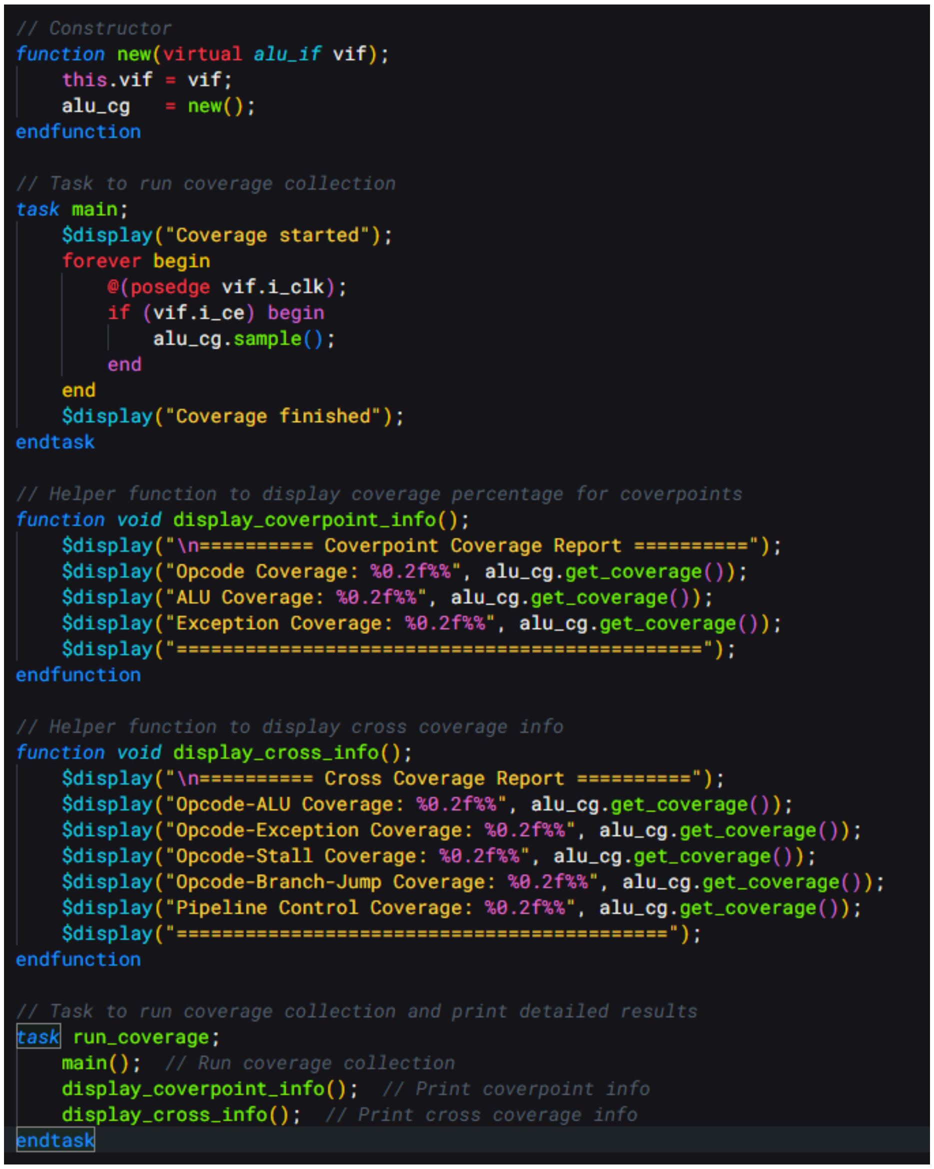
Additionally, it tracks exception signals and defines cross coverage among several signal groups to ensure that their interactions are thoroughly exercised.

The class includes a main task that, on every rising clock edge, samples the coverage data when the clock enable is active, and it provides helper functions to print out the coverage percentages for both individual coverpoints and their crosses.



```
class coverage;
    virtual alu_if vif;
    // Mailbox to send captured input transactions to the scoreboard.
    mailbox #(transaction) mon_in2scb = new();
    covergroup alu_cg;
        coverpoint vif.i_opcode {
            bins opcode_types[] = {[0 : `OPCODE_WIDTH - 1]};
        }
        coverpoint vif.i_alu {bins alu_ops[] = {[0 : `ALU_WIDTH - 1]};}

        // Branch and jump coverage
        coverpoint vif.o_change_pc;
        coverpoint vif.o_next_pc {bins pc_values = {[0 : 32'hFFFFFFFF]};}

        // Register writeback coverage
        coverpoint vif.o_wr_rd;
        coverpoint vif.o_rd_valid;
        coverpoint vif.o_rd_addr {bins rd_addr = {[0 : 31]};}

        // Pipeline management coverage
        coverpoint vif.i_ce;
        coverpoint vif.i_stall;
        coverpoint vif.i_force_stall;
        coverpoint vif.i_flush;
        coverpoint vif.o_stall_from_alu;
        coverpoint vif.o_stall;
        coverpoint vif.o_flush;

        // Exception coverage
        coverpoint vif.i_exception {
            bins exceptions[] = {[0 : `EXCEPTION_WIDTH - 1]};
        }

        // Cross coverage
        cross vif.i_opcode, vif.i_alu;
        cross vif.i_opcode, vif.i_exception;
        cross vif.i_opcode, vif.i_stall, vif.i_force_stall;
        cross vif.i_opcode, vif.o_change_pc iff (vif.o_change_pc == 1'b1);
        cross vif.i_ce, vif.i_stall, vif.i_force_stall, vif.i_flush;
        cross vif.o_wr_rd, vif.o_rd_valid, vif.i_opcode;
    endgroup
```

Figure 18 Coverage mailbox,interface, covergroup and coverpoint



```
// Constructor
function new(virtual alu_if vif);
    this.vif = vif;
    alu_cg   = new();
endfunction

// Task to run coverage collection
task main;
    $display("Coverage started");
    forever begin
        @(posedge vif.i_clk);
        if (vif.i_ce) begin
            alu_cg.sample();
        end
    end
    $display("Coverage finished");
endtask

// Helper function to display coverage percentage for coverpoints
function void display_coverpoint_info();
    $display("\n========== Coverpoint Coverage Report ==========");
    $display("Opcode Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("ALU Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Exception Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("==============================================");
endfunction

// Helper function to display cross coverage info
function void display_cross_info();
    $display("\n========== Cross Coverage Report ==========");
    $display("Opcode-ALU Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Exception Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Stall Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Branch-Jump Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Pipeline Control Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("===========================================");
endfunction

// Task to run coverage collection and print detailed results
task run_coverage;
    main();  // Run coverage collection
    display_coverpoint_info();  // Print coverpoint info
    display_cross_info();  // Print cross coverage info
endtask
```

Figure 19 Coverage main task invoked in `run_coverage`

Finally, a dedicated task runs the entire coverage collection process and displays the detailed reports, helping verify that the simulation exercises all aspects of the ALU and related pipeline controls.

Coverage AI Summary

Purpose

The coverage class collects functional coverage data to verify the RV32I ALU design. It samples various signals from the Device Under Test (DUT) through a virtual interface (vif) and uses a covergroup to track the coverage of ALU operations, branch/jump handling, register writeback, pipeline control signals, and exceptions. This ensures that all specified scenarios, including normal operations and edge cases, are exercised during simulation.

Key Components

- vif: A virtual interface (alu_if) that connects to the DUT's signals, allowing the class to sample them for coverage analysis.
- mon_in2scb: A mailbox for sending input transactions to the scoreboard. While declared, it is not used in the provided code snippet.
- alu_cg: A covergroup that defines the coverage model, including individual coverpoints and cross coverage points for key signals.

Covergroup: alu_cg

The alu_cg covergroup is the core of the coverage collection, specifying what signals to monitor and how to bin their values. It includes:

Coverpoints

Coverpoints define the individual signals to be sampled and the ranges or states to be covered:

- vif.i_opcode:
  - **Bins**: opcode_types[] = {[0 : OPCODE_WIDTH - 1]}`
  - Covers all possible opcode values, ensuring every opcode type is tested.
- vif.i_alu:
  - **Bins**: alu_ops[] = {[0 : ALU_WIDTH - 1]}`
  - Covers all ALU operation types, ensuring each operation (e.g., add, subtract) is exercised.
- vif.o_change_pc:
  - Covers whether the program counter changes (e.g., for branches or jumps), typically a 1-bit signal with two states (0 or 1).
- vif.o_next_pc:
  - **Bins**: pc_values = {[0 : 32'hFFFFFFFF]}
  - Covers the full 32-bit range of next program counter values, though this broad range may need refinement in practice.
- vif.o_wr_rd:
  - Covers whether a write to the register file is enabled (e.g., 0 for no write, 1 for write).
- vif.o_rd_valid:

  - Covers the validity of the register write data (e.g., 0 for invalid, 1 for valid).
- vif.o_rd_addr:
  - **Bins**: rd_addr = {[0 : 31]}
  - Covers all 32 possible destination register addresses in the RISC-V register file.
- **Pipeline Control Signals**:
  - vif.i_ce: Clock enable signal.
  - vif.i_stall: Stall signal from the pipeline.
  - vif.i_force_stall: Forced stall signal.
  - vif.i_flush: Flush signal to clear the pipeline.
  - vif.o_stall_from_alu: Stall signal originating from the ALU.
  - vif.o_stall: Overall stall output.
  - vif.o_flush: Flush output signal.
  - These ensure all pipeline control states are tested.
- vif.i_exception:
  - **Bins**: exceptions[] = {[0 : EXCEPTION_WIDTH - 1]}`
  - Covers all possible exception types, ensuring they are triggered during testing.

Cross Coverage

Cross coverage ensures that combinations of signals are tested together:

- cross vif.i_opcode, vif.i_alu:
  - Verifies that every opcode is tested with every ALU operation.
- cross vif.i_opcode, vif.i_exception:
  - Ensures exceptions are tested across different opcodes.
- cross vif.i_opcode, vif.i_stall, vif.i_force_stall:
  - Covers stall conditions (normal and forced) with various opcodes.
- cross vif.i_opcode, vif.o_change_pc iff (vif.o_change_pc == 1'b1):
  - Focuses on branch/jump scenarios (PC changes) for different opcodes, only sampling when o_change_pc is 1.
- cross vif.i_ce, vif.i_stall, vif.i_force_stall, vif.i_flush:
  - Ensures all combinations of pipeline control signals are tested.
- cross vif.o_wr_rd, vif.o_rd_valid, vif.i_opcode:

- Verifies register write scenarios (write enable and data validity) across different opcodes.

Core Functions and Tasks

- **Constructor** (new):
    - **Inputs**: virtual alu_if vif
    - Initializes the virtual interface (vif) and creates an instance of the alu_cg covergroup.
    - **Code**:

```
function new(virtual alu_if vif);
    this.vif = vif;
    alu_cg   = new();
endfunction

```

- **Task:** main:
    - Runs an infinite loop to collect coverage data.
    - Samples the covergroup on every positive edge of vif.i_clk when vif.i_ce (clock enable) is high, ensuring coverage is collected only when the ALU is active.
    - **Code**:

```
task main;
    $display("Coverage started");
    forever begin
        @(posedge vif.i_clk);
        if (vif.i_ce) begin
            alu_cg.sample();
        end
    end
    $display("Coverage finished");
endtask

```

- **Function:** display_coverpoint_info:
    - Displays coverage percentages for key coverpoints (opcode, ALU operations, exceptions).
    - Uses alu_cg.get_coverage() to retrieve overall coverage data, though individual coverpoint percentages could be refined with more specific methods in a real implementation.
    - **Code**:

```
function void display_coverpoint_info();
    $display("\n========== Coverpoint Coverage Report ==========");
    $display("Opcode Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("ALU Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Exception Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("==============================================");
```

```
endfunction

```

- **Function:** display_cross_info:
    - Displays coverage percentages for cross coverage points (e.g., opcode-ALU, opcode-exception).
    - Similarly uses alu_cg.get_coverage() for simplicity, though cross-specific reporting could be enhanced.
    - **Code**:

```
function void display_cross_info();
    $display("\n========== Cross Coverage Report ==========");
    $display("Opcode-ALU Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Exception Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Stall Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Opcode-Branch-Jump Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("Pipeline Control Coverage: %0.2f%%", alu_cg.get_coverage());
    $display("===========================================");
endfunction

```

- **Task:** run_coverage:
    - Executes the main task to collect coverage and then calls the display functions to print detailed results.
    - **Code**:

```
task run_coverage;
    main();  // Run coverage collection
    display_coverpoint_info();  // Print coverpoint info
    display_cross_info();  // Print cross coverage info
endtask

```

Behavior

The coverage class operates as follows:

1. **Initialization**: The constructor sets up the virtual interface and instantiates the covergroup.
2. **Sampling**: The main task continuously samples the covergroup on clock edges when the ALU is enabled (vif.i_ce == 1).
3. **Coverage Collection**: Coverpoints and crosses track the occurrence of signal values and combinations, ensuring comprehensive testing of:
    - ALU operations (opcodes and ALU types).
    - Branch/jump scenarios (PC changes).
    - Register writeback (write enable, validity, addresses).
    - Pipeline control (stalls, flushes).

   - Exceptions.

4. **Reporting**: The display functions provide a summary of coverage percentages, helping engineers assess the completeness of the test suite.

Comprehensive Verification

This class ensures thorough verification by:

- Sampling key signals across all operational aspects of the ALU.
- Using cross coverage to verify interactions between signals (e.g., opcodes with stalls or exceptions).
- Collecting data only when the ALU is active, focusing on relevant scenarios.
- Providing detailed reports to identify untested conditions.

Appendix:

Repository at branch m2:

- RISC-V-ALU-Design-and-Verification/team_3_RiscV_ALU/M2 at m2 · nnikolov3/RISC-V-ALU-Design-and-Verification

Coverage runs:

- RISC-V-ALU-Design-and-Verification/team_3_RiscV_ALU/M2/docs/report_50.txt at m2 · nnikolov3/RISC-V-ALU-Design-and-Verification
- RISC-V-ALU-Design-and-Verification/team_3_RiscV_ALU/M2/docs/report_1000.txt at m2 · nnikolov3/RISC-V-ALU-Design-and-Verification