ALU Verification Plan

1. Project Overview

This verification project implements a comprehensive SystemVerilog testbench for an ALU design. The verification environment features a driver-controlled architecture with event-based synchronisation, ensuring precise timing alignment between stimulus generation and response monitoring. The testbench incorporates advanced specification requirements, including retry mechanisms for two-operand operations and parameterised functional coverage that scales with operand width. The verification methodology emphasises constraint-based randomisation, comprehensive error checking, and detailed coverage analysis to ensure design correctness across all operational scenarios.

1.1 Design Overview

1.1.1 Design Description

The Arithmetic Logic Unit (ALU) is a digital circuit that performs arithmetic and logical operations on binary data. The ALU operates in two distinct modes: arithmetic mode for mathematical operations and logical mode for bitwise operations.

1.1.2 Use cases of ALU:

- ❖ Microprocessor Cores: Central processing unit arithmetic and logical instruction execution
- ❖ Digital Signal Processing: Mathematical computations for filtering, transforms, and signal analysis
- ❖ Graphics Processing: Pixel manipulation, coordinate transformations, and rendering calculations
- ❖ Cryptographic Systems: Bitwise operations, modular arithmetic, and encryption algorithms
- ❖ Control Systems: Real-time calculations for feedback control and system monitoring

- Communication Protocols: Checksum calculations, error detection, and data encoding/decoding
- Scientific Computing: High-precision mathematical operations in research applications
- Embedded Systems: Resource-constrained applications requiring efficient arithmetic operations

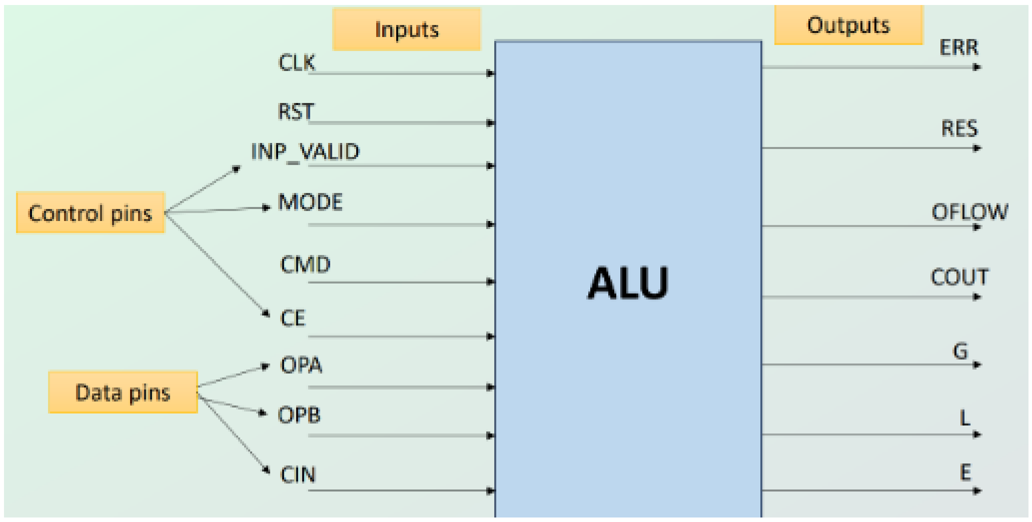
1.1.3 Key Features

- Parameterized Design: Configurable operand width (OP_WIDTH) supporting 8,16, 32,64,128 configurations
- Dual Operation Modes: Arithmetic mode (10 operations) and Logical mode (13 operations)
- Comprehensive Flag Generation: Six output flags (overflow, carry-out, greater, less, equal, error)
- Input Validation: Two-bit inp_valid signal controlling operand validity independently
- Clock Enable Control: Power management through ce signal
- Synchronous Reset: Deterministic state initialisation capability

2. Verification Objectives

- ❖ Operation Correctness: Verify all 27 arithmetic and logical operations produce correct results
- ❖ Timing Compliance: Verify 3-cycle latency for standard operations, 4-cycle for multiplication
- ❖ Flag Generation: Validate proper generation of overflow, carry, comparison, and error flags
- ❖ Input Validation: Ensure two-operand operations require inp_valid = 2'b11
- ❖ Reset Functionality: Confirm proper reset behavior and state initialization
- ❖ Clock Enable: Validate power management through selective operation enabling
- ❖ Functional Coverage: Above 95% coverage of all operation combinations
- ❖ Code Coverage:Good percentage of statement, branch, and condition coverage
- ❖ Corner Cases: Boundary value testing for all operand ranges
- ❖ Error Scenarios: Complete validation of error detection mechanisms

3. DUT Interfaces



Pin Name	Direction	No of Bits	Description
OPA	INPUT	Parameterized	Operand 1
OPB	INPUT	Parameterized	Operand 2
CIN	INPUT	1	Active high carry-in signal
CLK	INPUT	1	Clock signal
RST	INPUT	1	Active high asynchronous reset
CE	INPUT	1	Active high clock enable
MODE	INPUT	1	1: Arithmetic, 0: Logical operation
INP_VALID	INPUT	2	00: No operand valid, 01: A valid, 10: B valid(single operand operations), 11: Both valid(2 operand operations) **,Also single op Valid**
CMD (Arithmetic)	INPUT	4 (Default)	0: ADD, 1: SUB, 2: ADD_CIN, 3: SUB_CIN, 4: INC_A, 5: DEC_A, 6: INC_B, 7: DEC_B, 8: CMP, 9: A+1 * B+1,10: A<<1 * B
CMD (Logical)			0: AND, 1: NAND, 2: OR, 3: NOR, 4: XOR, 5: XNOR, 6: NOT_A, 7: NOT_B, 8: SHR1_A, 9: SHL1_A, 10: SHR1_B, 11: SHL1_B
CMD (ROL_A_B)			CMD:12 OP:0:OPA, 1: OPA << 1, 2: OPA << 2, ..., 7: OPA << 7 , | [7:4] high-ERR

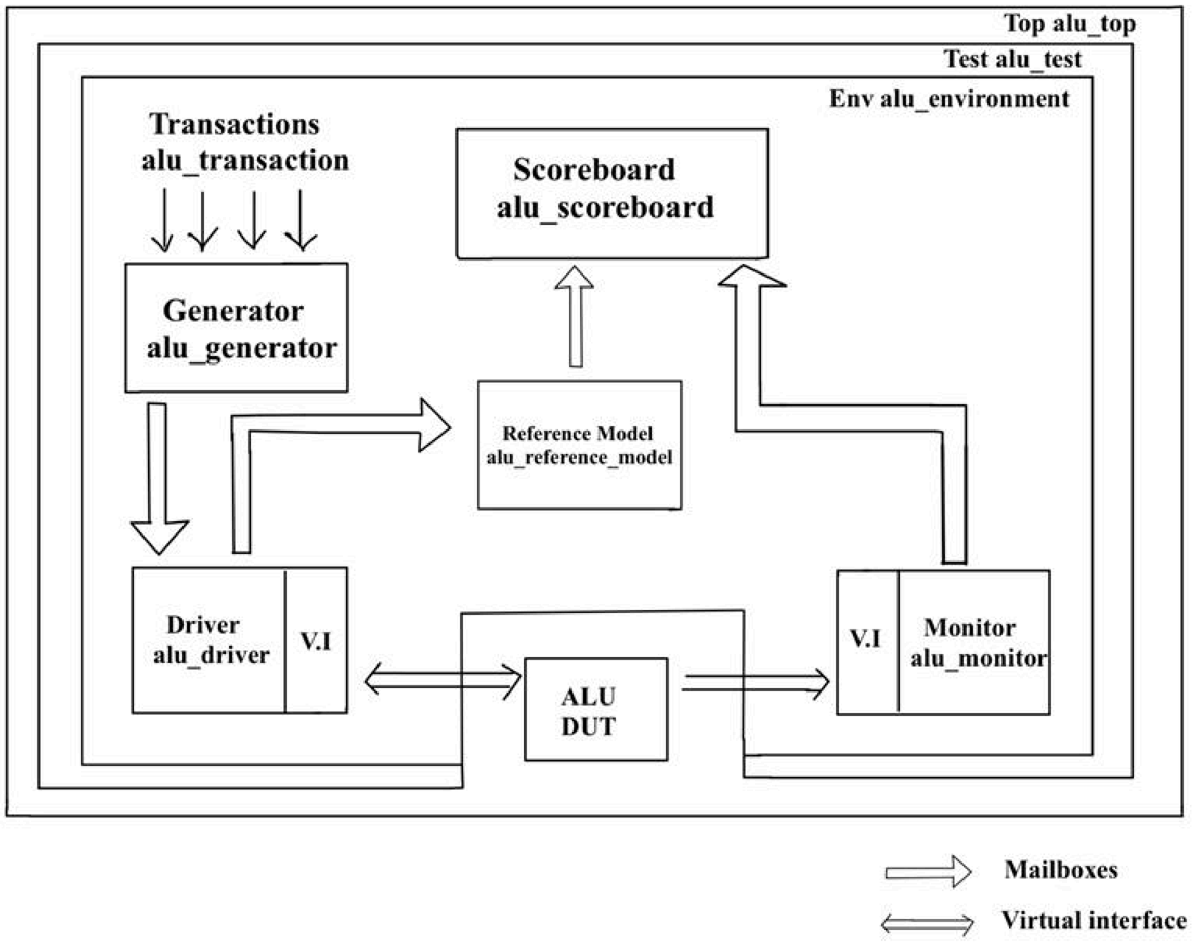
CMD (ROR_A_B)			CMD:13 0: OPA, 1: OPA >> 1, 2: OPA >> 2, ..., 7: OPA >> 7 , | [7:4] high-ERR
RES	OUTPUT	2*Parameterized (FOR MUL)	Result of the operation
OFLOW	OUTPUT	1	Overflow flag
COUT	OUTPUT	1	Carry out flag
G	OUTPUT	1	1 if OPA > OPB
L	OUTPUT	1	1 if OPA < OPB
E	OUTPUT	1	1 if OPA == OPB
ERR	OUTPUT	1	Error signal

4. Testbench Architecture

4.1 Architecture Overview

The testbench implements a driver-controlled verification environment with event-based synchronization, constraint-based randomization, and comprehensive coverage collection. The architecture features modular components communicating through mailboxes and events for precise timing coordination.

4.2 Architecture Diagram



4.3 Component Descriptions

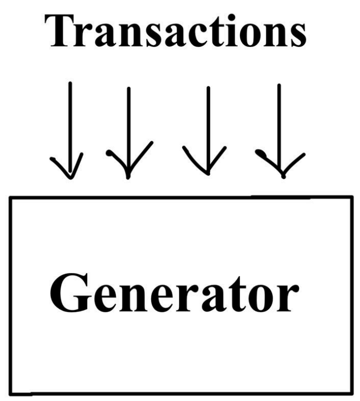
4.3.1 Transaction

Purpose: Fundamental data container encapsulating ALU input stimuli and output responses

Key Features:

- Parameterized: OP_WIDTH/CMD_WIDTH scalability
- Deep Copy Support: Blueprint pattern implementation

Flow:



Template:

```
None
class alu_transaction;
  // Randomizable Input Fields

  // Output Fields (Non-randomizable)

  // Constraints

  // Deep copy Method for blueprint pattern
  virtual function alu_transaction clone();
endclass
```

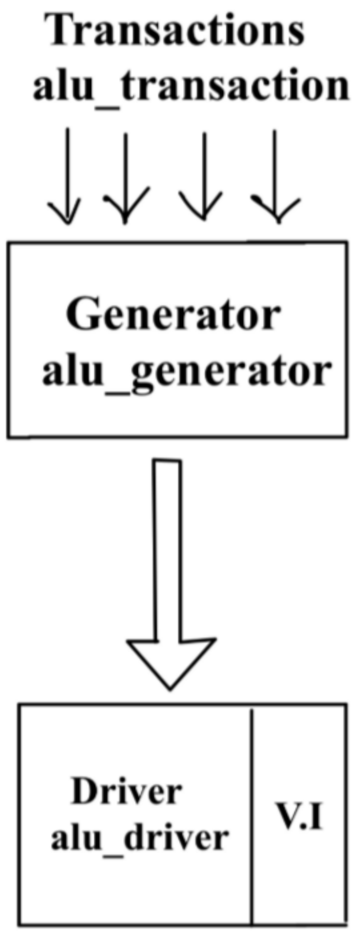
4.3.2 Generator

Purpose: Creates randomized transaction objects with constraints

Key Features:

- ❖ Constraint-based randomization for all input signals
- ❖ Parameterized operand generation based on OP_WIDTH
- ❖ Transaction count control via `no_of_trans macro

Flow:



- ❖ Generate random transaction
- ❖ Apply constraints
- ❖ Send transaction to driver via mbx_gd mailbox
- ❖ Repeat for specified number of transactions

Template:

```
None
class alu_generator;
  // Class Properties
  alu_transaction trans;
  mailbox #(alu_transaction) mbx_gd;

  // Constructor
  function new(mailbox #(alu_transaction) mbx_gd);

  task start();
    // Loop for specified transaction count
    // Create and randomize transactions
    // Send to driver via mailbox
  endtask
endclass
```

4.3.3 Driver

Purpose: Drives DUT inputs and manages timing synchronization

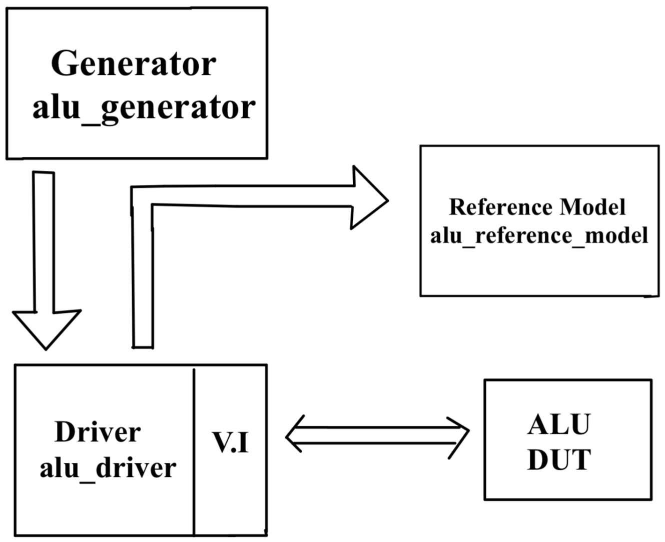
Key Features:

- ❖ Two-operand operation detection with is_two_op() function
- ❖ 16-cycle retry mechanism for inp_valid=2'b11 requirement
- ❖ Selective randomization control using rand_mode()
- ❖ Event-based monitor triggering after calculated delays
- ❖ Parameterized input functional coverage collection

Flow:

- ❖ Receive transaction from generator
- ❖ Check if two-operand operation using is_two_op()
    - ➢ For Two-Operand Operations:
        - ■ If inp_valid==2'b11: Drive DUT and reference model
        - ■ If inp_valid!=2'b11: Enter retry loop (max 16 cycles)
            - ● Drive DUT with current values
            - ● Disable cmd/mode randomization

        - Re-randomize other fields
        - Check inp_valid again
    - ➢ For Single-Operand Operations: Drive normally
- ❖ Calculate delay cycles (3 for standard, 4 for multiplication)
- ❖ Trigger monitor after appropriate delay
- ❖ Sample input functional coverage



Template:

```
None
class alu_driver;
  // Class Properties
  alu_transaction drv_trans;
  mailbox #(alu_transaction) mbx_gd;    // From generator
  mailbox #(alu_transaction) mbx_dr;    // To reference model
  virtual alu_if.DRV vif;
  event drv_trigger_event;              // To monitor synchronization

  // Input Functional Coverage
  covergroup drv_cg;

  // Constructor
  function new(mailbox #(alu_transaction) mbx_gd,
               mailbox #(alu_transaction) mbx_dr,
               virtual alu_if.DRV vif,
               event drv_trigger_event);
  // Utility Methods
  function int get_delay_cycles(input [`CMD_WIDTH-1:0] cmd, input mode);
  function bit is_two_op(input [`CMD_WIDTH-1:0] c, input m);

  // Drive Tasks
  task drive_dut_and_ref_model();       // Drive DUT and send to ref model
  task drive_dut();                     // Drive DUT only
  task trigger_monitor();               // Trigger monitor after delay

  task start();
    // Main transaction loop
    // Two-operand operation detection
    // 16-cycle retry mechanism for inp_valid=2'b11
    // Selective randomization control
    // Coverage sampling
    // Monitor triggering
  endtask
endclass
```

4.3.4 Monitor

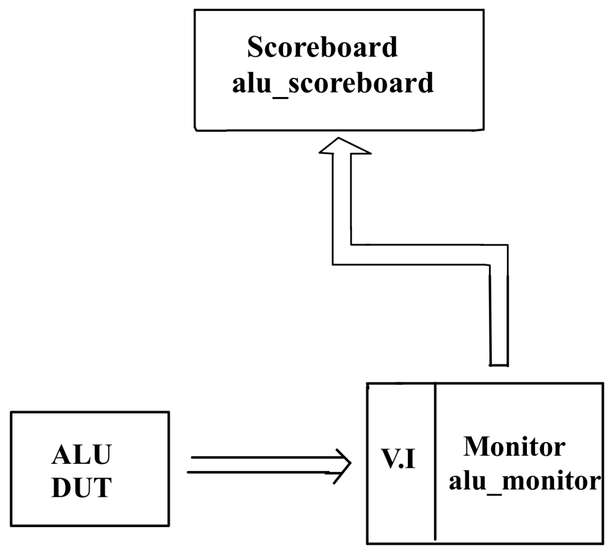
Purpose: Captures DUT outputs with precise timing coordination

Key Features:

- ❖ Event-based activation from driver trigger
- ❖ Parameterized output coverage with conditional width
- ❖ Transaction forwarding to scoreboard

Flow:

- ❖ Wait for driver trigger event (drv_trigger_event)
- ❖ Capture all DUT outputs at triggered time
- ❖ Create monitor transaction with captured data
- ❖ Sample output functional coverage
- ❖ Send transaction to scoreboard via mbx_ms



Template:

```
None
class alu_monitor;
  // Class Properties
  alu_transaction mon_trans;
  mailbox #(alu_transaction) mbx_ms;    // To scoreboard
  virtual alu_if.MON vif;
  event drv_trigger_event;              // From driver synchronization

  // Output Functional Coverage
  covergroup mon_cg;

  // Constructor
  function new(mailbox #(alu_transaction) mbx_ms,
               virtual alu_if.MON vif,
               event drv_trigger_event);

  task start();
    // Wait for driver trigger events
    // Capture DUT outputs at precise timing
    // Create monitor transactions
    // Sample output coverage
    // Send to scoreboard
  endtask
endclass
```

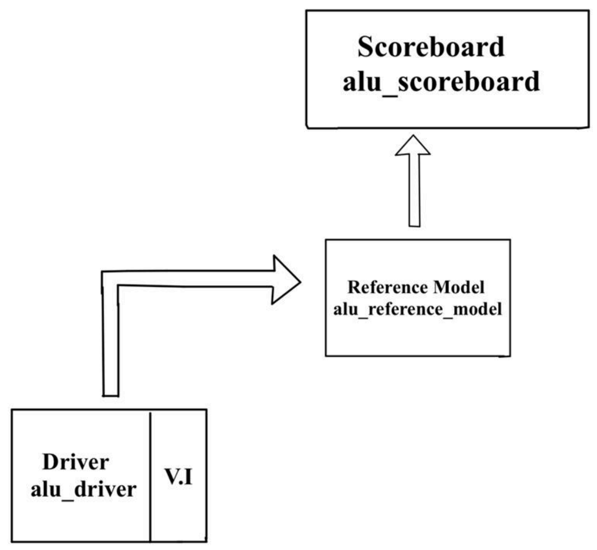
4.3.5 Reference Model

Purpose: Provides expected results for comparison

Key Features:

- ❖ Comprehensive operation implementation
- ❖ Reset and clock enable logic
- ❖ Error condition detection

Flow:



- ❖ Receive transaction from driver via mbx_dr
- ❖ Check reset and clock enable conditions
- ❖ Implement operation based on mode and command
- ❖ Send expected results to scoreboard via mbx_rm

Template:

```
None
class alu_reference_model;
  // Class Properties
  alu_transaction ref_trans;
  mailbox #(alu_transaction) mbx_dr;    // From driver
  mailbox #(alu_transaction) mbx_rm;    // To scoreboard
  virtual alu_if.REF_SB vif;

  // Constructor
  function new(mailbox #(alu_transaction) mbx_dr,
               mailbox #(alu_transaction) mbx_rm,
               virtual alu_if.REF_SB vif);

  // Golden Reference Calculation
  task calculate_expected(alu_transaction trans);
    // Reset and Clock Enable Logic
    // Error condition detection
    // Mode-based operation implementation
  endtask

  // Main Reference Model Task
  task start();
    // Receive transactions from driver
    // Calculate expected results
    // Send to scoreboard
  endtask
endclass
```

4.3.6 Scoreboard

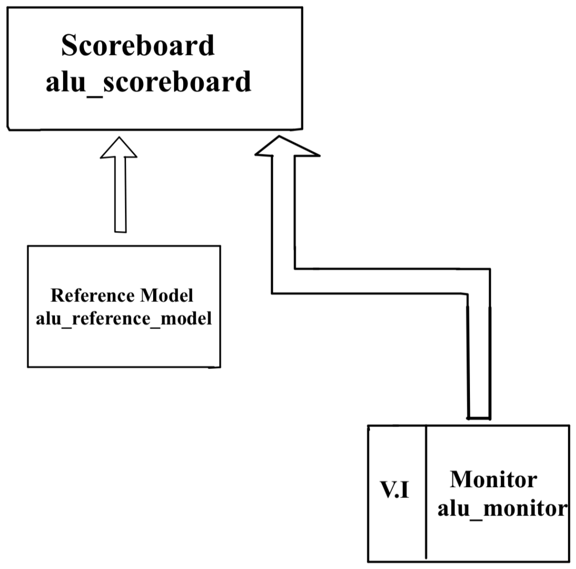
Purpose: Compares actual vs expected results

Key Features:

- ❖ Transaction matching
- ❖ Detailed mismatch reporting
- ❖ Pass/fail determination

Flow:

- ❖ Receive actual results from monitor
- ❖ Receive expected results from reference model
- ❖ Compare all fields (result, flags, error conditions)
- ❖ Report mismatches with detailed information



Template:

```
None
class alu_scoreboard;
  // Class Properties
  alu_transaction actual_trans;
  alu_transaction expected_trans;
  mailbox #(alu_transaction) mbx_ms;    // From monitor
  mailbox #(alu_transaction) mbx_rm;    // From reference model

  int pass_count;
  int fail_count;

  // Constructor
  function new(mailbox #(alu_transaction) mbx_ms,
               mailbox #(alu_transaction) mbx_rm);

  // Comparison Methods
  function bit compare_transactions();   // Overall comparison

  // Main Scoreboard Task
  task start();
    // Receive actual and expected transactions
    // Perform detailed comparisons
    // Track matches & mismatches
  endtask
endclass
```

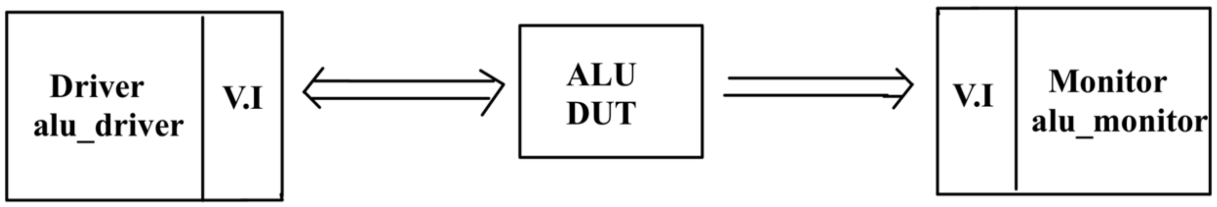
4.3.7 Interface

Purpose: Provides modular connection between testbench and DUT

Key Features:

- ❖ Separate modports for driver, monitor,reference model
- ❖ Clocking blocks for synchronous operation and direction control

Flow:



Template:

```
None
interface alu_if(input bit clk, input bit rst);
  // Signal Declarations - Parameterized Widths
  // Conditional Result Width
  // Output Flags

  // Clocking Blocks - Synchronous Operation
  clocking drv_cb @(posedge clk);
    // Driver output timing
  endclocking

  clocking mon_cb @(posedge clk);
    // Monitor input timing
  endclocking

  clocking ref_sb_cb @(posedge clk);
    // Reference model input timing
  endclocking

  // Modports - Access Control
  modport DRV (clocking drv_cb);
  modport MON (clocking mon_cb);
  modport REF_SB (clocking ref_sb_cb);
endinterface
```

4.3.8 Environment

Purpose: Orchestrates all verification components and manages their interactions

Key Features:

- ❖ Component instantiation and configuration
- ❖ Mailbox creation and connection
- ❖ Event for synchronization
- ❖ Parallel task execution control

Template:

```
None
class alu_environment;
  // Component Instances
  alu_generator gen;
  alu_driver drv;
  alu_monitor mon;
  alu_reference_model ref_model;
  alu_scoreboard sb;

  // Communication Channels
  mailbox #(alu_transaction) mbx_gd;    // Generator to Driver
  mailbox #(alu_transaction) mbx_dr;    // Driver to Reference Model
  mailbox #(alu_transaction) mbx_ms;    // Monitor to Scoreboard
  mailbox #(alu_transaction) mbx_rm;    // Reference Model to Scoreboard

  // Synchronization Events
  event drv_trigger_event;              // Driver to Monitor sync

  // Virtual Interface Handles
  virtual alu_if.DRV vif_drv;
  virtual alu_if.MON vif_mon;
  virtual alu_if.REF_SB vif_ref;

  // Constructor

  // Build Phase - Component Creation and Connection
  function void build();
    // Create communication mailboxes
    // Instantiate all verification components
    // Connect virtual interfaces and events
  endfunction

  // Run Phase - Parallel Execution
  task run();
    // Start all components in parallel
  endtask
endclass
```

4.3.9 Test

Purpose: Defines specific test scenarios and configuration parameters

Key Features:

- ❖ Environment instantiation and configuration
- ❖ Test-specific transaction blueprint assignment

Template:

```
None
class alu_test;
  // Class Properties
  alu_environment env;

  // Virtual Interface Handles
  virtual alu_if.DRV vif_drv;
  virtual alu_if.MON vif_mon;
  virtual alu_if.REF_SB vif_ref;

  // Constructor
  function new(virtual alu_if.DRV vif_drv,
               virtual alu_if.MON vif_mon,
               virtual alu_if.REF_SB vif_ref);

  // Test Execution Methods
  task run();
    // Initialize test environment
    // Configure test parameters
    // Call build and start tasks of env
  endtask
endclass
```

4.3.10 Top Module

Purpose: Serves as simulation harness and DUT instantiation point

Key Features:

- ❖ Clock and reset generation
- ❖ DUT instantiation and connection
- ❖ Interface instantiation
- ❖ Test execution control

Template:

```
None
  `include "defines.sv"      // Parameter definitions
  `include "alu_pkg.sv"      // Verification package
  `include "alu_if.sv"       // Interface definition
  `include "alu_top.v"       // DUT implementation
module top;
  // Package Imports
  import alu_pkg::*;

  // Clock and Reset Signals
  bit clk;
  bit rst;
  // Clock Generation
  // Reset Initialization

  initial begin
    // Initialize clock and reset states
  end

  // Interface Instantiation
  alu_if alu_intf(clk, rst);

  // DUT Instantiation and Port Mapping

  // Test Execution Block
  initial begin
    // Test instantiation with interface handles
    // Main test execution
  end
endmodule
```