ALU Verification Plan

By Ashok D'sa EMP ID:6070

1. Project Overview

1.1 Arithmetic Logic Unit (ALU)

An Arithmetic Logic Unit (ALU) is a core component of a computer's central processing unit (CPU) responsible for carrying out arithmetic operations (such as addition, subtraction, multiplication and division) and logic operations (such as AND, OR, NOT and XOR). It is one of the most essential parts of the CPU, enabling it to process data and make decisions. The performance and efficiency of the ALU directly impact the overall speed and capability of a computer system.

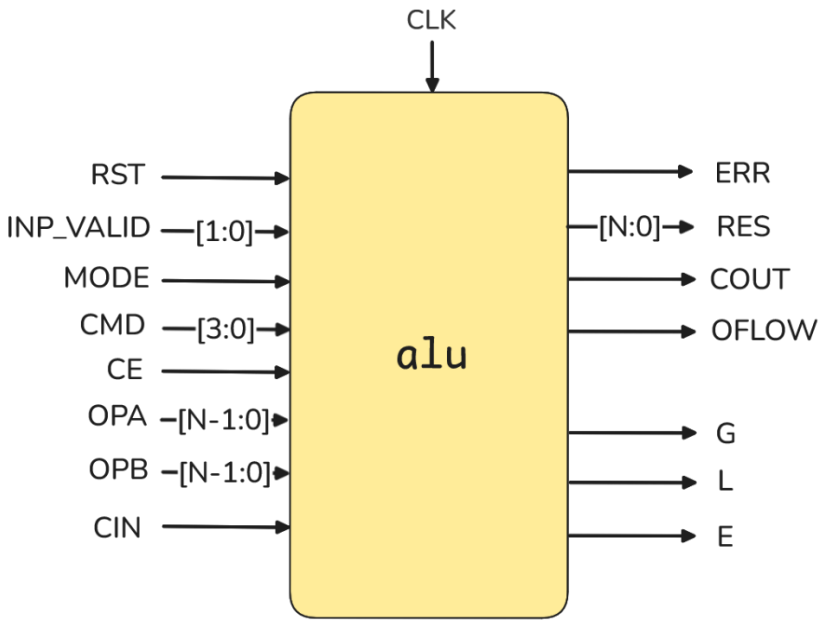


Figure 1.1: ALU Block

1.2 Advantage of ALU

- Integrates arithmetic and logic functions into a single unit, reducing the need for additional components.
- Facilitates advanced operations needed in scientific, engineering and real-time applications.
- Acts as the core computational unit, enhancing the speed and capability of the processor.

1.3 Disadvantage of ALU

- ALUs typically perform only basic arithmetic and logic operations, more complex tasks require additional hardware or processing units.
- ALU's rely on registers and memory units for input and output handling.
- An ALU needs instructions from the control unit to operate, making it incapable of functioning independently.
- ALUs are general purpose, they may not be optimized for specialized tasks.

1.4 Use cases of ALU

- Used in CPUs to execute basic arithmetic operations like addition, subtraction, multiplication and division.
- Decision Making in Programs using logical operations (e.g., AND, OR, NOT) essential for conditional branching and comparisons in software.
- Found in microcontrollers used in appliances, automobiles and industrial machines for real-time control and processing.
- Supports calculations needed for rendering graphics, physics simulations and game logic.
- Used in digital signal processors (DSPs) for fast, repetitive arithmetic operations on audio, video, or image data.

2. Verification Objectives

- Review design documentation: analyze functional requirements, supported operations, inputs/outputs, operating modes and edge-case behaviors.
- Construct the verification plan.
- Develop functional coverage and assertion plan.
- Frame the testbench architecture for the alu design.
- Creation of template codes for testbench components
- Implement and enhance testbench for coverage, integrating the functional coverage and SystemVerilog assertions.
- Validating the functional correctness of the ALU, considering the corner cases as well.
- Validating the timing of the operations.
- Checking its robustness against errors.

3. DUT Interfaces

These are the signals present in the interface which is to be shared between the Test and Design.

Signal	Direction	Size(bits)	Description
CLK	INPUT	1	Clock Signal
RST	INPUT	1	Active High Asynchronous Reset
INP_VALID	INPUT	2	Shows the Validity of the Operands(active high) MSB shows the validity of the OPB and LSB shows the validity for OPA.
MODE	INPUT	1	If the value is 1 the ALU is in Arithmetic Mode else it is in Logical Mode
CMD	INPUT	4	Commands for the Operation
CE	INPUT	1	Active high clock enable signal
OPA	INPUT	Parameterized	Operand 1
OPB	INPUT	Parameterized	Operand 2
CIN	INPUT	1	Carry In signal
ERR	OUTPUT	1	Active High Error Signal
RES	OUTPUT	Parameterized + 1	Result of the instruction performed by the ALU.
COUT	OUTPUT	1	Carry out signal, updated during Addition/Subtraction
G	OUTPUT	1	Comparator output which indicates that the value of OPA is greater than the value of OPB
L	OUTPUT	1	Comparator output which indicates that the value of OPA is lesser than the value of OPB
E	OUTPUT	1	Comparator output which indicates that the value of OPA is equal than the value of OPB
OFLOW	OUTPUT	1	Indicates output overflow, during Addition/Subtraction

4. Testbench Architecture

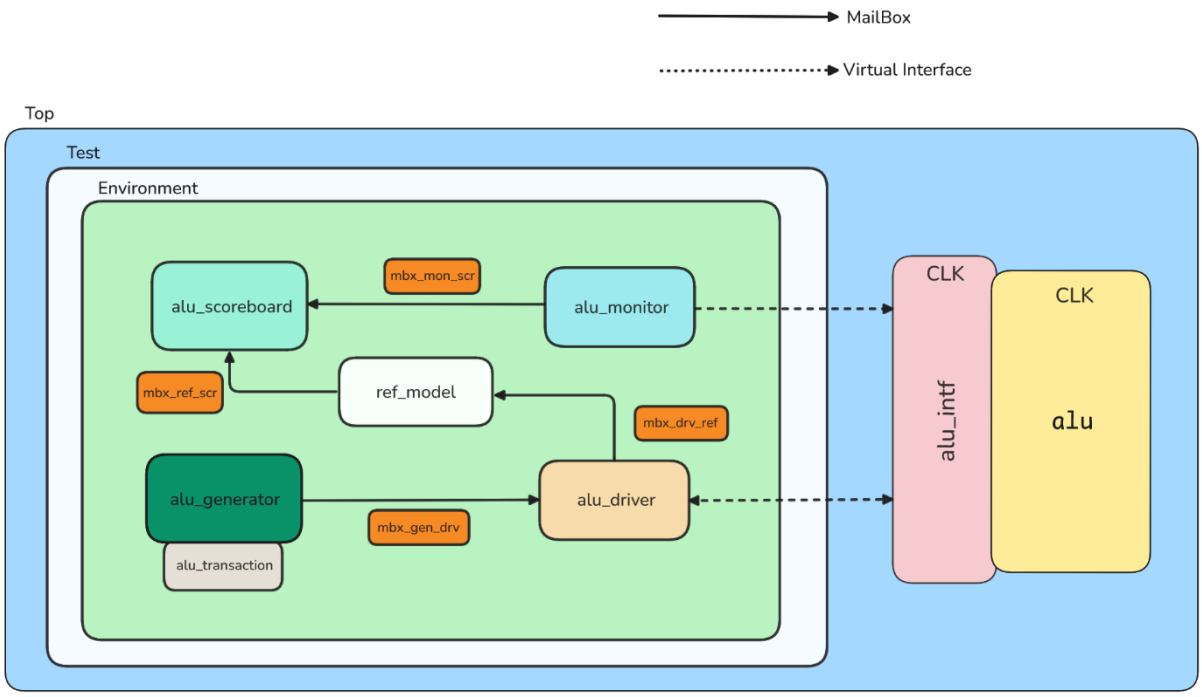
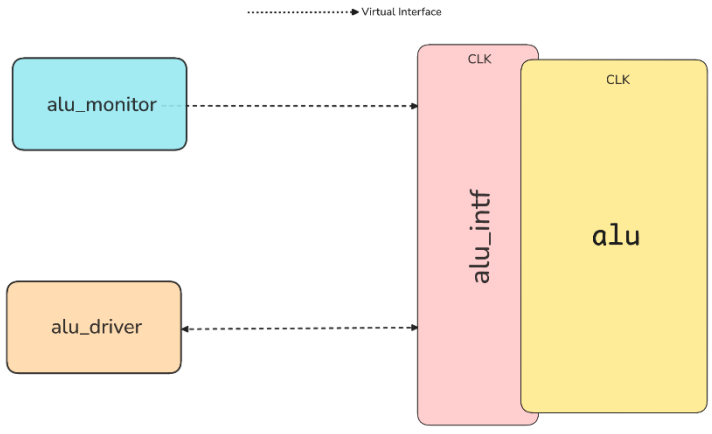


Figure 2: Test Bench Architecture for ALU

4.1 Flow Chart of each component with explanation

Interface

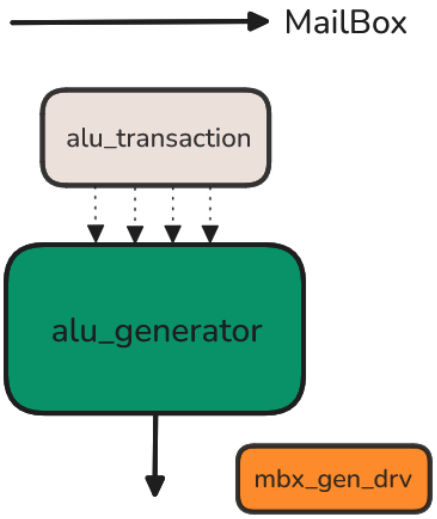
Bundled set of signals connecting testbench modules to the DUT.



```
class alu_intf(input clk);
  logic rst;
  logic [1:0]inp_valid;
  logic mode;
  logic [3:0] cmd;
  logic ce;
  logic [(`WIDTH - 1) : 0]opa,opb;
  logic cin;
  logic err;
  logic [`WIDTH : 0]res;
  logic oflow,cout;
  logic g,l,e;

  clocking drv_cb @(posedge clk);
    default input #0 output #0;
    output rst;
    output inp_valid;
    output mode;
    output cmd;
    output ce;
    output opa;
    output opb;
    output cin;
  endclocking

  clocking mon_cb @(posedge clk);
    default input #0 output #0;
    input err;
    input res;
    input oflow;
    input cout;
    input g,l,e;
  endclocking

  clocking ref_cb @(posedge clk);
  endclocking

  modport DRV (clocking drv_cb);
  modport MON (clocking mon_cb);
  modport REF (clocking ref_cb);
endclass
```

Transaction

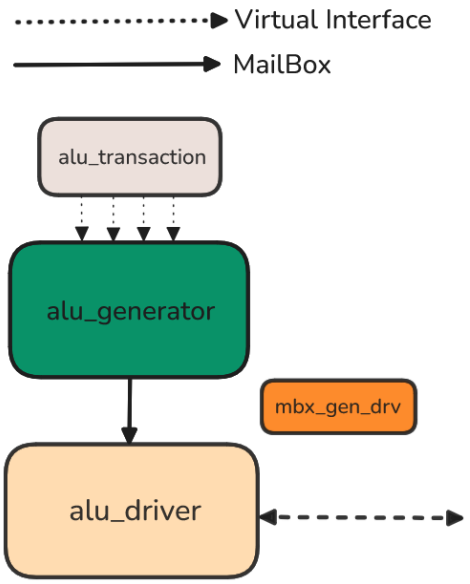
Object that encapsulates the stimulus exchanged between testbench components, containing all inputs and outputs of the DUT. excluding the clock signal, which is generated separately in the top module. The input stimulus fields can be declared as rand to enable randomization in the generator and can be constrained to target specific test scenarios.




```
`include "defines.svh"
class base_transaction; //proper working
  rand logic rst;
  rand logic[1:0] inp_valid;
  randc logic mode;
  randc logic [3:0] cmd;
  rand logic ce;
  rand logic [(`WIDTH-1):0]opa,opb;
  rand logic cin;
  logic err;
  logic [`WIDTH : 0] res;
  logic cout;
  logic g,l,e;
  logic oflow;
  //constraints

  virtual function base_transaction copy();
    copy = new();
    //copy the properties from this class over to object copy
    return copy;
  endfunction
endclass
```

Generator

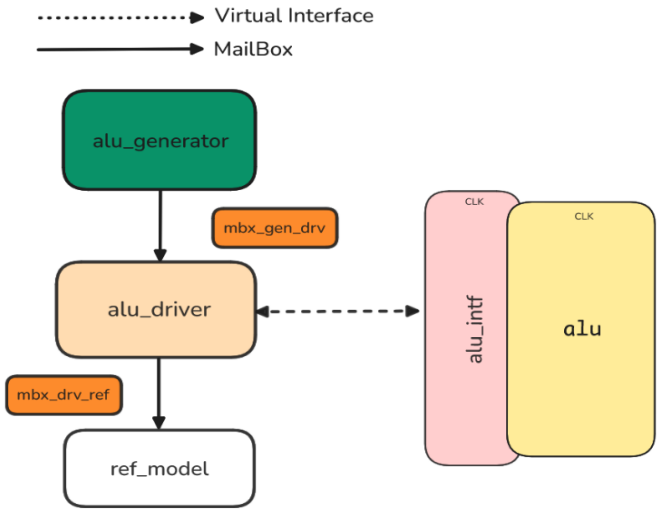
This is a component of the testbench which generates constrained random stimuli (transactions) for the DUT. The generator then sends the generated stimuli to the driver through a mailbox(mbx_gen_drv).




```
class alu_generator;
  base_transaction trans;
  mailbox #(base_transaction) mbx_gen_drv;

  function new(mailbox #(base_transaction) mbx_gen_drv);
    this.mbx_gen_drv = mbx_gen_drv;
    trans = new();
  endfunction

  task start();
    repeat() begin
      assert(trans.randomize);
      mbx_gen_drv.put(trans.copy);
    end
  endtask
endclass
```

Driver

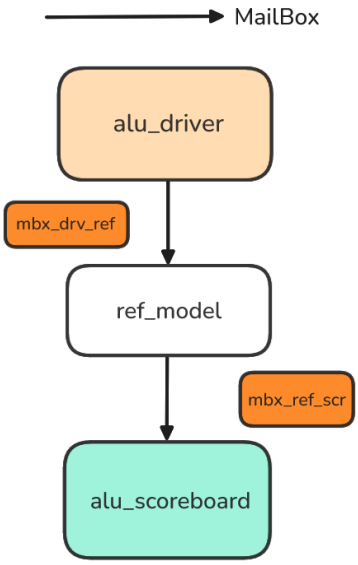
This component converts high-level transactions into pin-level activity at the DUT inputs. It receives transactions from the generator via a mailbox(mbx_gen_drv) and drives them to the DUT using a virtual interface. Also, it forwards the received transactions to the reference model through another mailbox (mbx_drv_ref) for result prediction and comparison.




```
class alu_driver;
  base_transaction trans;

  mailbox #(base_transaction) mbx_gen_drv;
  mailbox #(base_transaction) mbx_drv_ref;

  virtual alu_intf.DRV vif;

  function new(mailbox #(base_transaction) mbx_gen_drv,mailbox
#(base_transaction) mbx_drv_ref,virtual alu_intf.DRV vif);
  this.mbx_gen_drv = mbx_gen_drv;
  this.mbx_drv_ref = mbx_drv_ref;
  this.vif = vif;
  endfunction

  task start();
    repeat()
    begin
      trans = new();
      mbx_gen_drv.get(trans);
      //drive to vif
      mbx_drv_ref.put(trans);
    end
  endtask
endclass
```

Reference Model

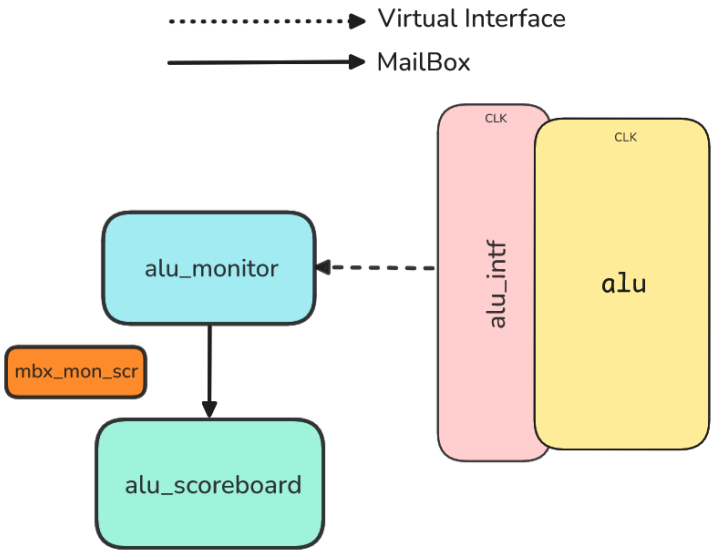
Serves as golden implementation/expected output for output prediction, validation and evaluation of the actual output.




```
class alu_reference;
  base_transaction trans;

  mailbox #(base_transaction) mbx_drv_ref;
  mailbox #(base_transaction) mbx_ref_scr;

  virtual alu_intf.REF vif;

  function new(mailbox #(base_transaction) mbx_drv_ref, mailbox
#(base_transaction) mbx_ref_scr,virtual alu_intf.REF vif);
  this.mbx_drv_ref = mbx_drv_ref;
  this.mbx_ref_scr = mbx_ref_scr;
  this.vif = vif;
  endfunction

  task start();
    repeat()
    begin
      trans = new();
      mbx_drv_ref.get(trans);
      //reference code
      @(vif.ref_cb);
      mbx_ref_scr.put(trans);
    end
  endtask
endclass
```

Monitor

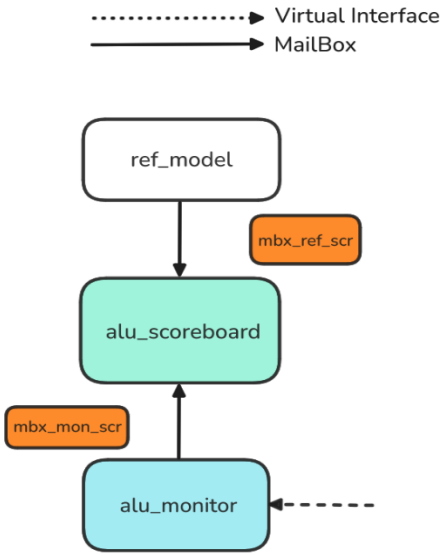
Captures the DUT outputs and sends it to the scoreboard.




```
class alu_monitor;

  base_transaction trans;

  mailbox #(base_transaction) mbx_mon_scr;

  virtual alu_intf.MON vif;

  function new(mailbox #(base_transaction) mbx_mon_scr,virtual alu_intf.MON
vif);
  this.mbx_mon_scr = mbx_mon_scr;
  this.vif = vif;
  endfunction

  task start();
    repeat()
    begin
      trans = new();
      @(vif.mon_cb);
      //read from vif
      mbx_mon_scr.put(trans);
    end
  endtask
endclass
```

Scoreboard

Compares the actual outputs of the DUT against expected results (reference model).

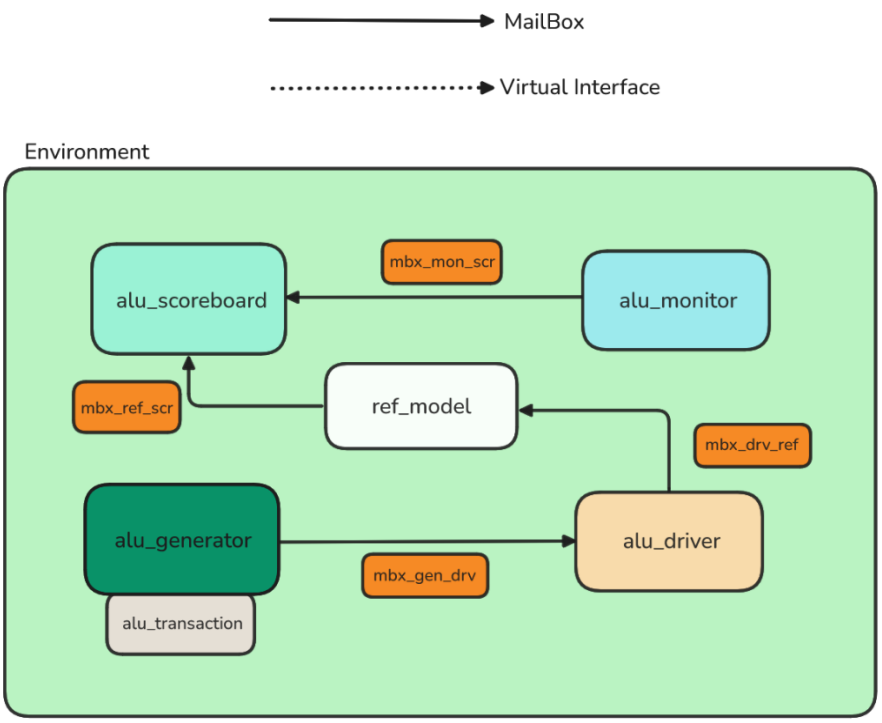



```
class alu_scoreboard;
  base_transaction dut_trans;
  base_transaction ref_trans;

  mailbox #(base_transaction) mbx_ref_scr;
  mailbox #(base_transaction) mbx_mon_scr;

  function new(mailbox #(base_transaction) mbx_ref_scr,mailbox
#(base_transaction) mbx_mon_scr);
    this.mbx_ref_scr = mbx_ref_scr;
    this.mbx_mon_scr = mbx_mon_scr;
  endfunction

  task start();
    repeat(`no_of_trans)
    begin
      dut_trans = new();
      ref_trans = new();
      fork
        begin
          mbx_mon_scr.get(dut_trans);
        end
        begin
          mbx_ref_scr.get(ref_trans);
        end
      join
      compare();
    end
  endtask
```

```
  task compare();
     //compares transaction from monitor and reference model
  endtask
endclass
```

Environment

Integrates all the test components and connects them using mailboxes.

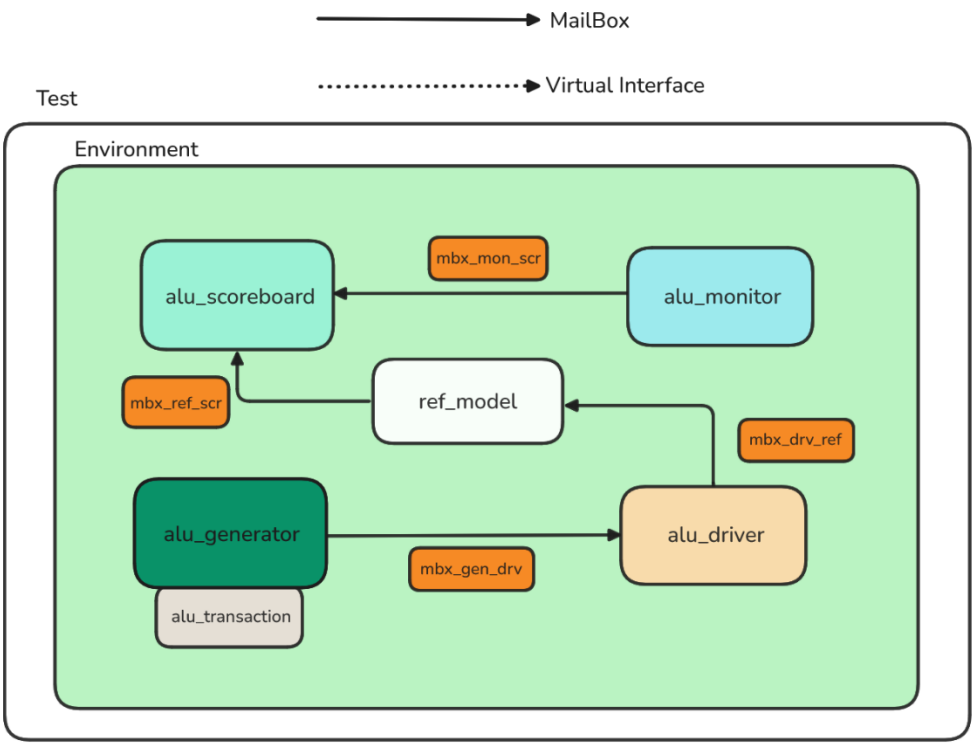



```
class alu_environment;
  alu_generator gen;
  alu_driver drv;
  alu_monitor mon;
  alu_scoreboard scr;
  alu_reference_model refe;

  mailbox #(base_transaction) mbx_gen_drv;
  mailbox #(base_transaction) mbx_drv_ref;
  mailbox #(base_transaction) mbx_mon_scr;
  mailbox #(base_transaction) mbx_ref_scr;

  virtual alu_intf vif;

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

```
  task build();
    mbx_gen_drv = new();
    mbx_drv_ref = new();
    mbx_mon_scr = new();
    mbx_ref_scr = new();

    gen = new(mbx_gen_drv);
    drv = new(mbx_gen_drv,mbx_drv_ref,vif);
    mon = new(mbx_mon_scr,vif);
    scr = new(mbx_ref_scr,mbx_mon_scr);
    refe = new(mbx_drv_ref,mbx_ref_scr,vif);
  endtask

  task start();
    fork
      gen.start();
      drv.start();
      mon.start();
      scr.start();
      refe.start();
    join
endclass
```

Test

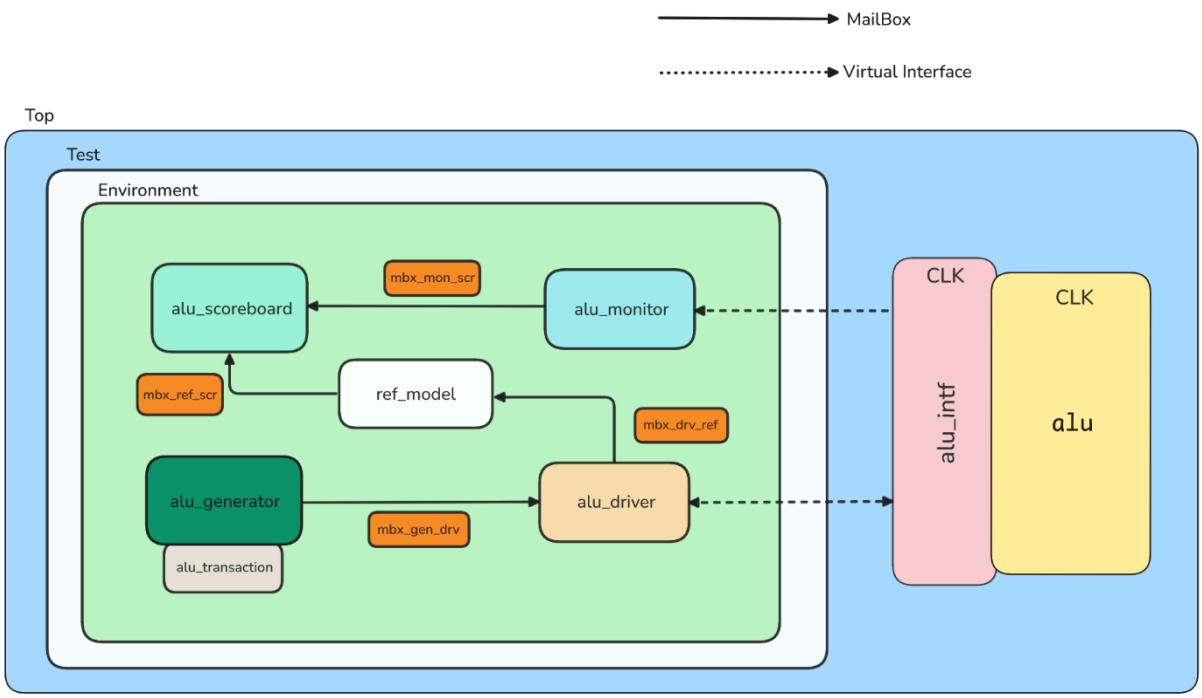
Used to configure and coordinate all testbench components.




```
class test;
  alu_environment env;
  virtual alu_intf vif;
  function new(virtual alu_intf vif);
    this.vif = vif;
  endfunction

  task start();
    env = new(vif);
    env.build();
    begin
      env.start();
    end
  endtask
endclass
```

Top

The top-level module that instantiates all components (DUT, interface and test)




```
module top;
  alu_pkg::*;
  bit clk;

  alu_intf intf(clk);

  ALU DUT(intf);

  always #5 clk = ~clk;

  test1 tb = new(intf);

  initial begin //initialize the reset
    @(negedge clk);
    intf.reset = 1;
  end

  initial begin //MAIN
    tb.start();
    $finish;
  end
endmodule
```

3. Test Plan

3.1 Test Scenarios

Test Plan

3.2 Functional Coverage Plan

Functional Coverage Plan

3.3 Assertions

Assertion Plan