ALU UVM

Omar Magdy

ALU UVM

The verification environment is compiled and run using **VCS.**

Run make to compile and run the test bench and generate the coverage reports.

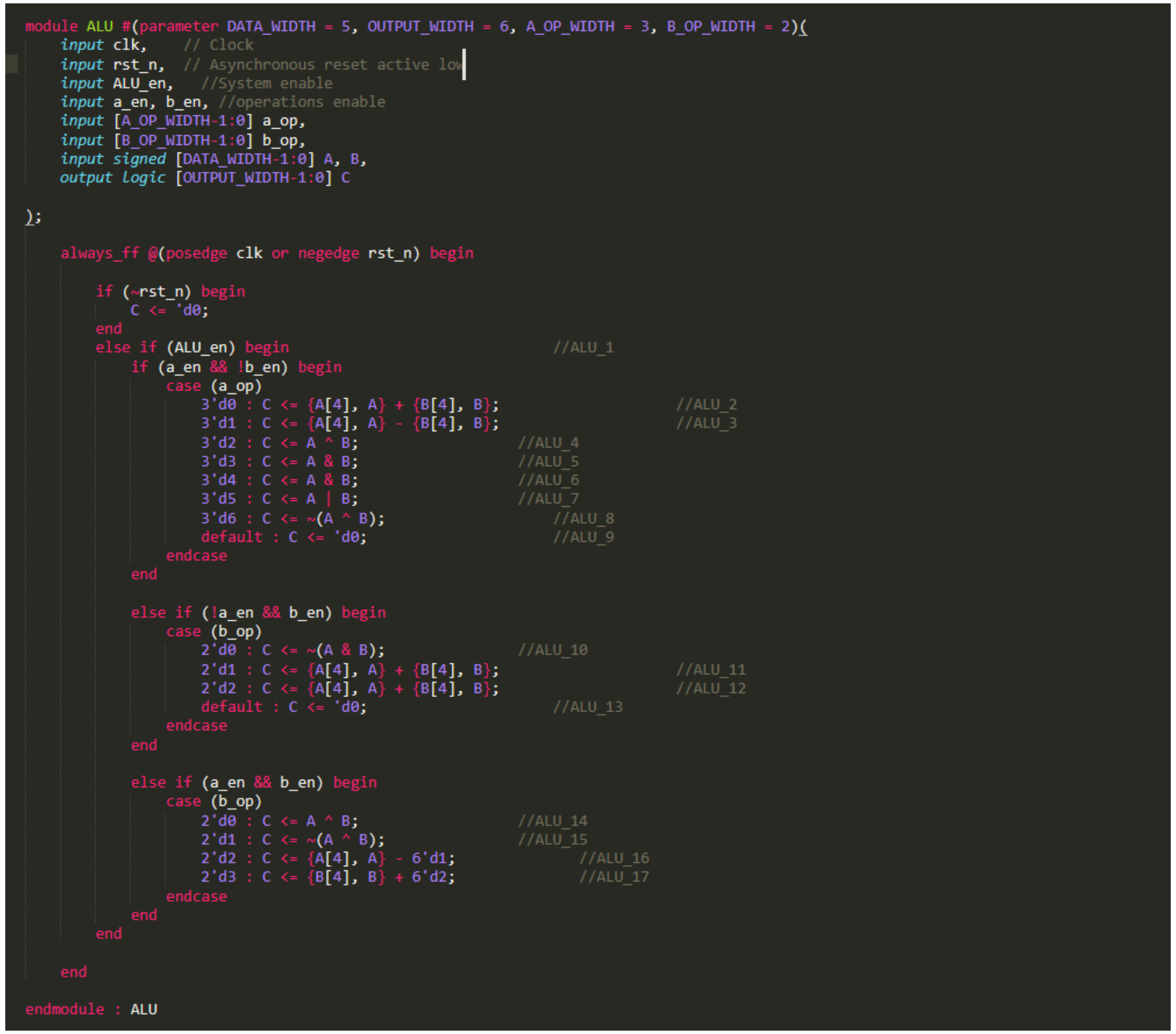
AGENDA

ALU

The verification environment is designed to fully verify ALU.

It has 10 ports, 9 input ports and 1 output port.

There are three operation sets, each of them are used depending on the different combinations of “a_en” and “b_en”.

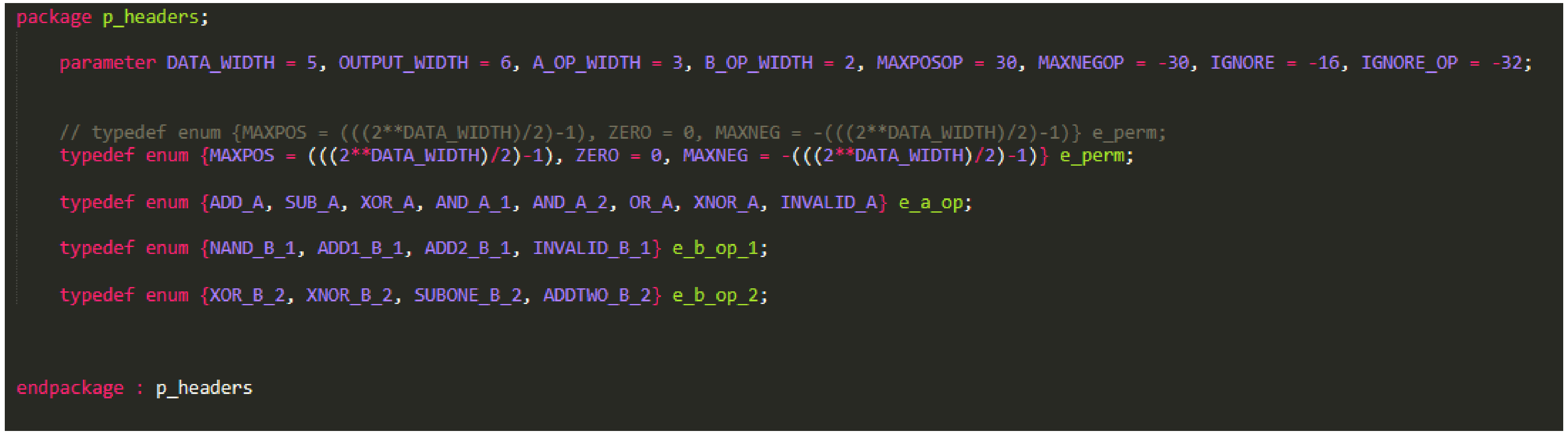


```
module ALU #(parameter DATA_WIDTH = 5, OUTPUT_WIDTH = 6, A_OP_WIDTH = 3, B_OP_WIDTH = 2)(
    input clk,    // Clock
    input rst_n,  // Asynchronous reset active low
    input ALU_en,   //System enable
    input a_en, b_en, //operations enable
    input [A_OP_WIDTH-1:0] a_op,
    input [B_OP_WIDTH-1:0] b_op,
    input signed [DATA_WIDTH-1:0] A, B,
    output logic [OUTPUT_WIDTH-1:0] C

);

    always_ff @(posedge clk or negedge rst_n) begin

        if (~rst_n) begin
            C <= 'd0;
        end
        else if (ALU_en) begin                             //ALU_1
            if (a_en && !b_en) begin
                case (a_op)
                    3'd0 : C <= {A[4], A} + {B[4], B};                    //ALU_2
                    3'd1 : C <= {A[4], A} - {B[4], B};                    //ALU_3
                    3'd2 : C <= A ^ B;                  //ALU_4
                    3'd3 : C <= A & B;                  //ALU_5
                    3'd4 : C <= A & B;                  //ALU_6
                    3'd5 : C <= A | B;                  //ALU_7
                    3'd6 : C <= ~(A ^ B);                  //ALU_8
                    default : C <= 'd0;                    //ALU_9
                endcase
            end

            else if (!a_en && b_en) begin
                case (b_op)
                    2'd0 : C <= ~(A & B);               //ALU_10
                    2'd1 : C <= {A[4], A} + {B[4], B};                    //ALU_11
                    2'd2 : C <= {A[4], A} + {B[4], B};                    //ALU_12
                    default : C <= 'd0;                    //ALU_13
                endcase
            end

            else if (a_en && b_en) begin
                case (b_op)
                    2'd0 : C <= A ^ B;                  //ALU_14
                    2'd1 : C <= ~(A ^ B);               //ALU_15
                    2'd2 : C <= {A[4], A} - 6'd1;             //ALU_16
                    2'd3 : C <= {B[4], B} + 6'd2;             //ALU_17
                endcase
            end
        end

    end

endmodule : ALU
```

PACKAGE

Package file is created to group common parameters that are used in multiple classes like input bit width, maximum positive, and negative values.

Also, it contains defined enums to facilitate dealing with the operations.

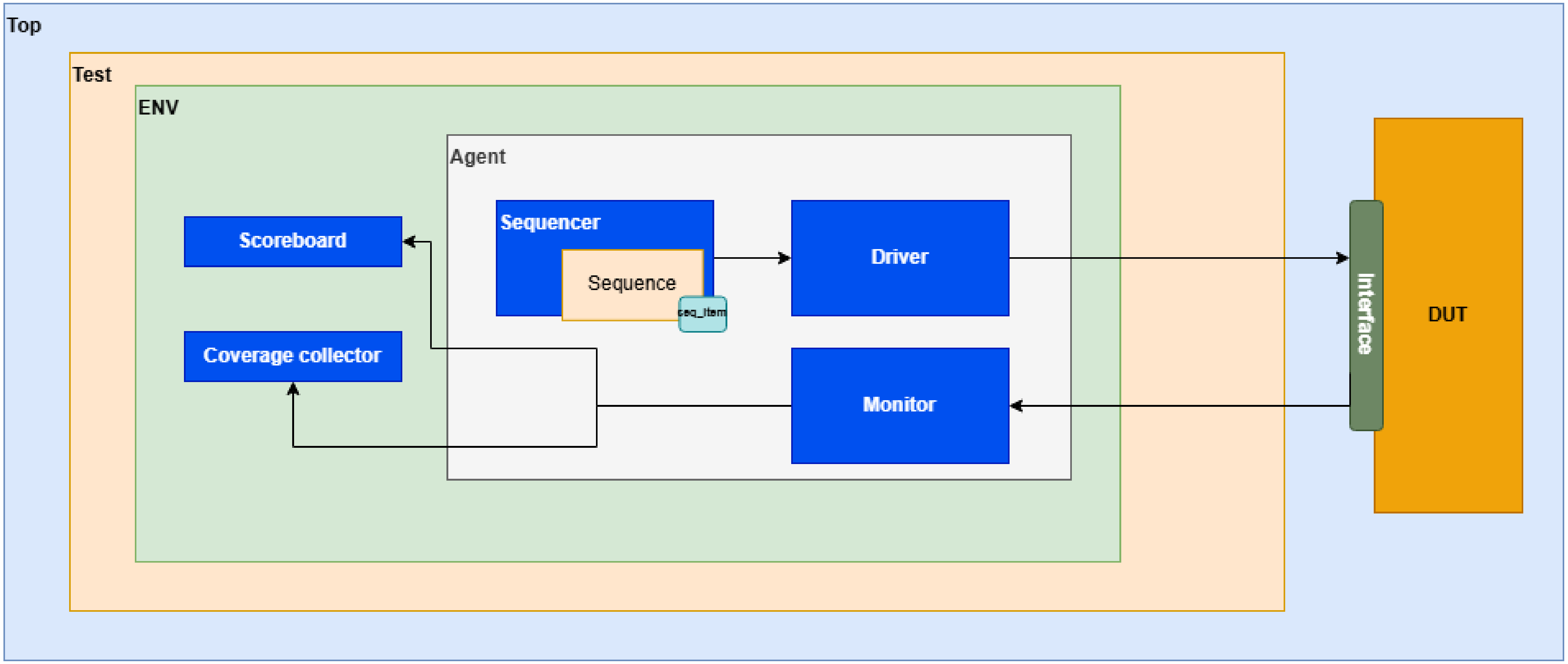


```
package p_headers;

    parameter DATA_WIDTH = 5, OUTPUT_WIDTH = 6, A_OP_WIDTH = 3, B_OP_WIDTH = 2, MAXPOSOP = 30, MAXNEGOP = -30, IGNORE = -16, IGNORE_OP = -32;


    // typedef enum {MAXPOS = (((2**DATA_WIDTH)/2)-1), ZERO = 0, MAXNEG = -(((2**DATA_WIDTH)/2)-1)} e_perm;
    typedef enum {MAXPOS = (((2**DATA_WIDTH)/2)-1), ZERO = 0, MAXNEG = -(((2**DATA_WIDTH)/2)-1)} e_perm;

    typedef enum {ADD_A, SUB_A, XOR_A, AND_A_1, AND_A_2, OR_A, XNOR_A, INVALID_A} e_a_op;

    typedef enum {NAND_B_1, ADD1_B_1, ADD2_B_1, INVALID_B_1} e_b_op_1;

    typedef enum {XOR_B_2, XNOR_B_2, SUBONE_B_2, ADDTWO_B_2} e_b_op_2;


endpackage : p_headers
```

ENVIRONMENT ARCHITECTURE

The UVM environment consists of:

- Sequence item: required to have the data to be randomized.
- Sequence: responsible for defining the scenario in which the sequence item randomization depends on.
- Sequencer: used to drive sequence item from sequence to driver.
- Driver: receives data from sequencer and drives the interface signals.
- Monitor: samples the DUT signals and convert them to transaction level.
- Agent: encapsulates driver, monitor, and sequencer.
- Scoreboard: checks the transaction coming from monitor.
- Coverage Collector: samples the incoming transaction to calculate functional coverage.
- Environment: encapsulates scoreboard, coverage collector.
- Test: it determine which sequence will be run.
- Top: it connects the DUT to the testbench.

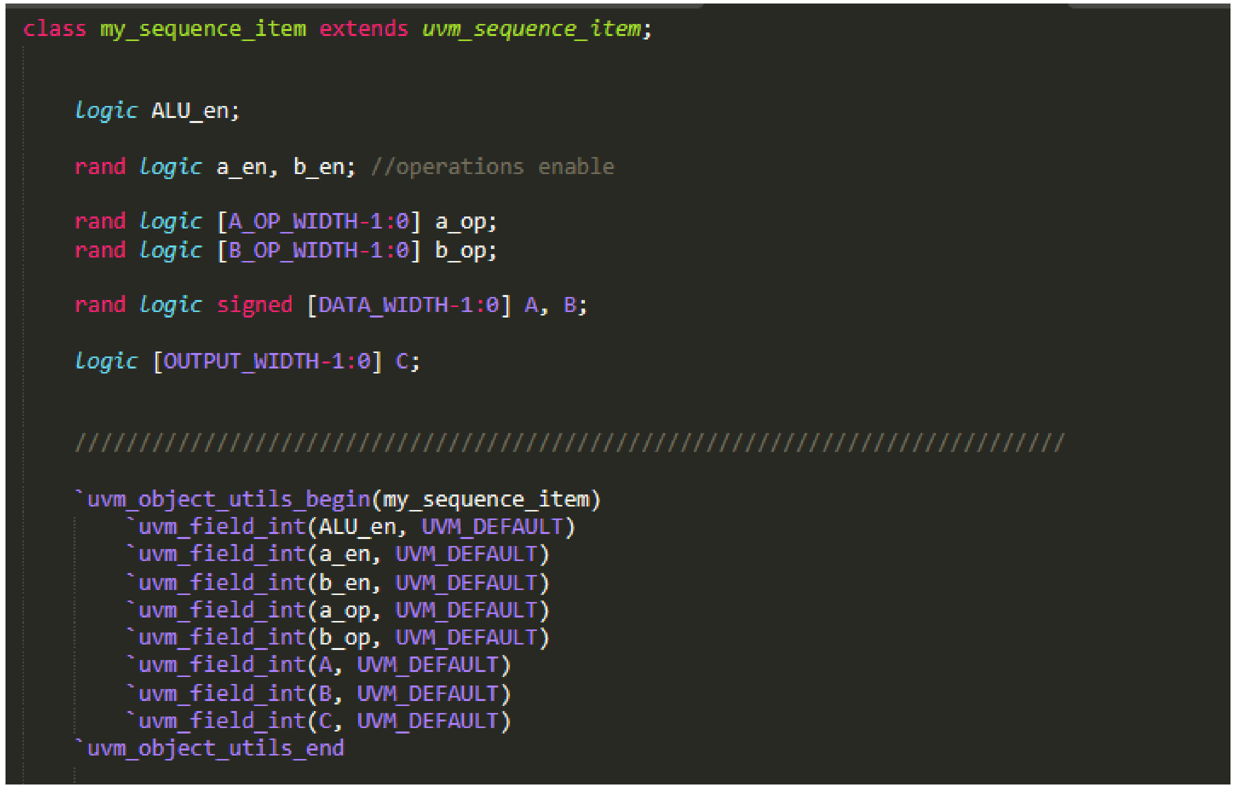

Top
Test
ENV
Agent
Sequencer
Sequence
seq_item
Driver
Monitor
Scoreboard
Coverage collector
Interface
DUT

SEQUENCE ITEM

Sequence item contains the data that is going to be randomized.

Also, it has the constraints in which the randomization will use to achieve the highest functional coverage.

Field macro is used to determine which data is going to be printed.



```
class my_sequence_item extends uvm_sequence_item;


    logic ALU_en;

    rand logic a_en, b_en; //operations enable

    rand logic [A_OP_WIDTH-1:0] a_op;
    rand logic [B_OP_WIDTH-1:0] b_op;

    rand logic signed [DATA_WIDTH-1:0] A, B;

    logic [OUTPUT_WIDTH-1:0] C;


    ////////////////////////////////////////////////////////////////////////////////


    `uvm_object_utils_begin(my_sequence_item)
        `uvm_field_int(ALU_en, UVM_DEFAULT)
        `uvm_field_int(a_en, UVM_DEFAULT)
        `uvm_field_int(b_en, UVM_DEFAULT)
        `uvm_field_int(a_op, UVM_DEFAULT)
        `uvm_field_int(b_op, UVM_DEFAULT)
        `uvm_field_int(A, UVM_DEFAULT)
        `uvm_field_int(B, UVM_DEFAULT)
        `uvm_field_int(C, UVM_DEFAULT)
    `uvm_object_utils_end
```

SEQUENCE

The sequence determine the scenario for testing.

The sequence class has 4 scenarios:

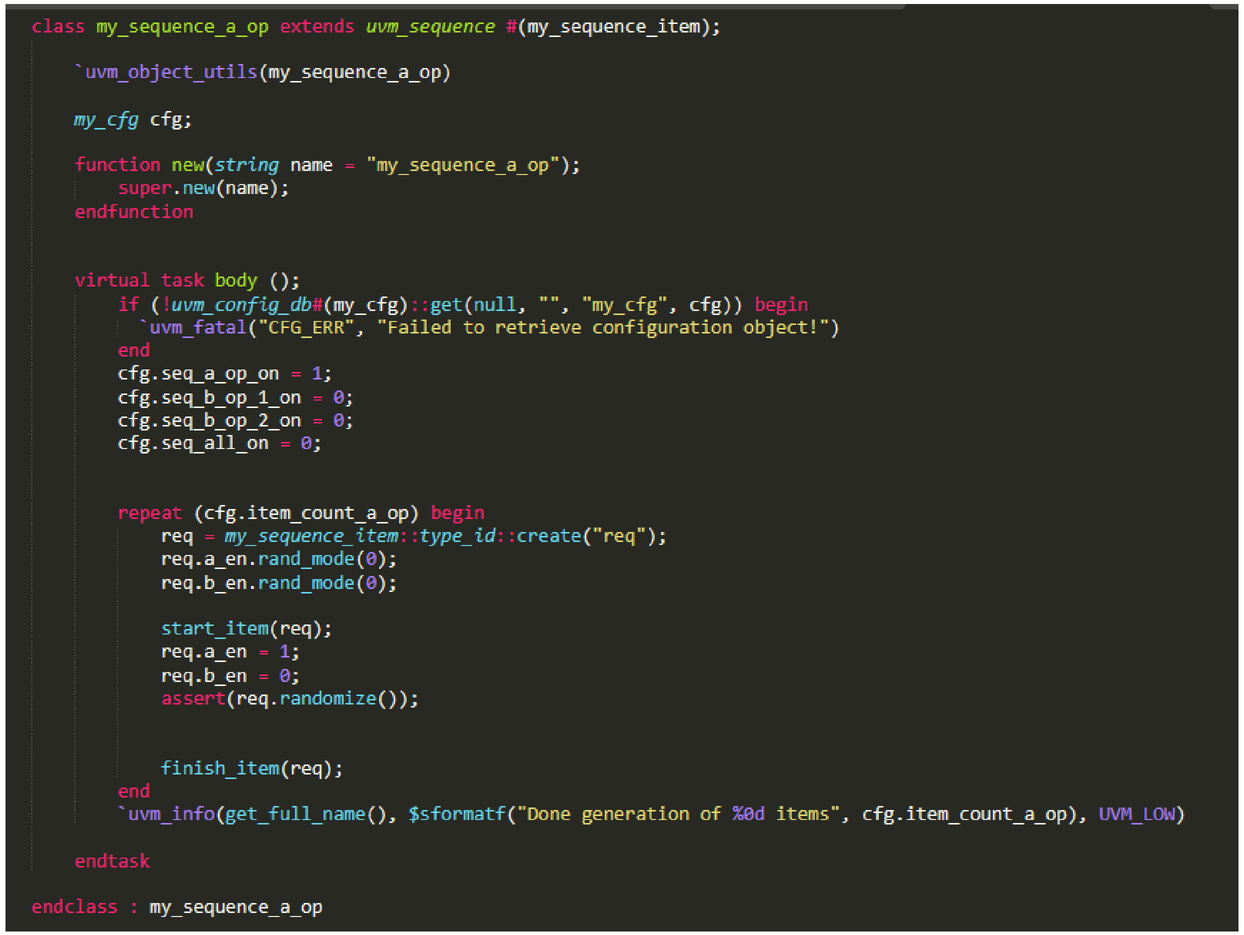
- One for a_op set.
- One for the first b_op set.
- One for the second b_op set.
- One is a randomization sequence.

Final sequence is used to wrap and all sequences using uvm_do macro.

SEQUENCE

Sequence a_op

- The sequence turn off randomization for a_en and b_en and sets them to 1 and 0 respectively to enable the a_op set.
- It get the item count from configuration object set in the test.

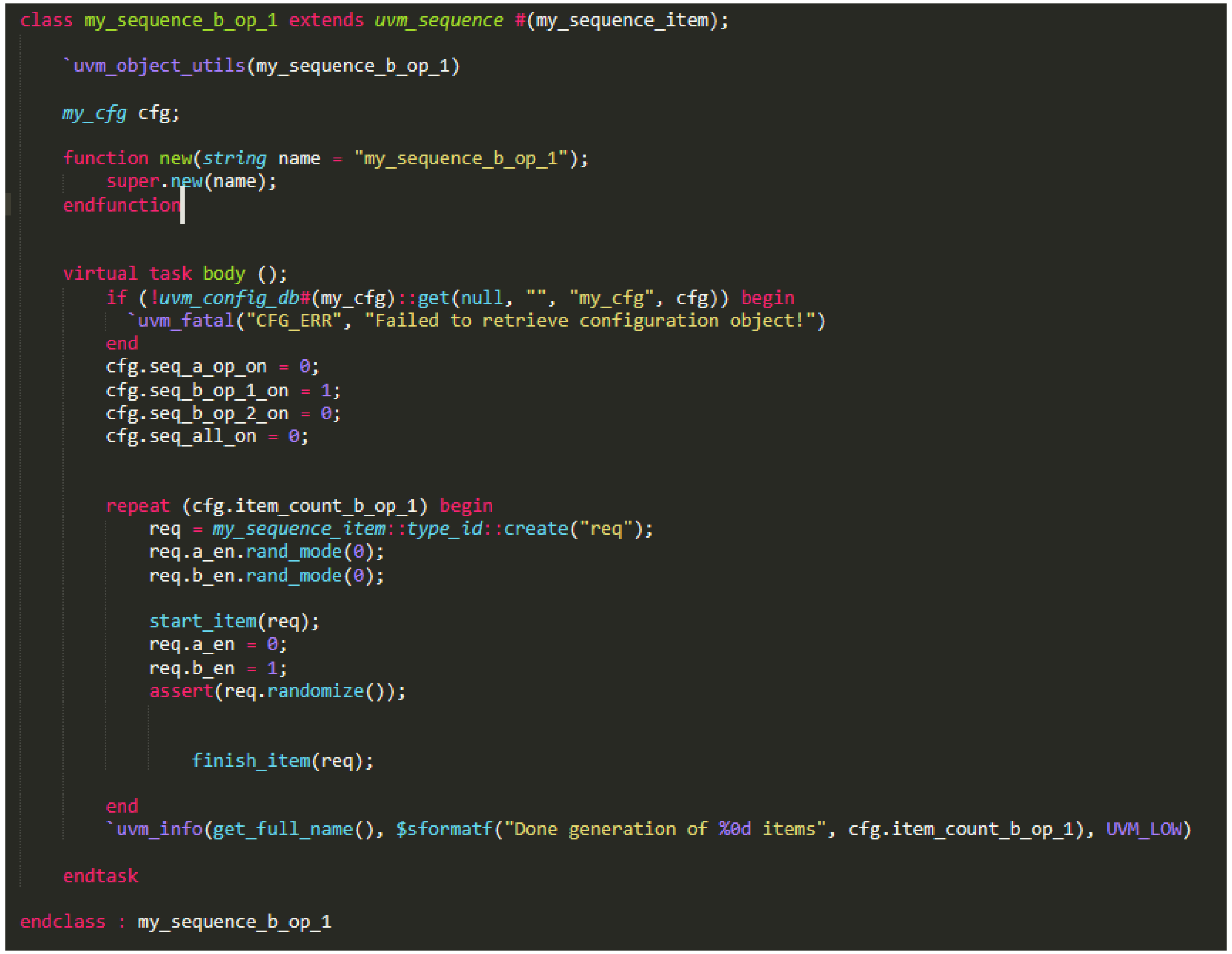


```
class my_sequence_a_op extends uvm_sequence #(my_sequence_item);

    `uvm_object_utils(my_sequence_a_op)

    my_cfg cfg;

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


    virtual task body ();
        if (!uvm_config_db#(my_cfg)::get(null, "", "my_cfg", cfg)) begin
          `uvm_fatal("CFG_ERR", "Failed to retrieve configuration object!")
        end
        cfg.seq_a_op_on = 1;
        cfg.seq_b_op_1_on = 0;
        cfg.seq_b_op_2_on = 0;
        cfg.seq_all_on = 0;


        repeat (cfg.item_count_a_op) begin
            req = my_sequence_item::type_id::create("req");
            req.a_en.rand_mode(0);
            req.b_en.rand_mode(0);

            start_item(req);
            req.a_en = 1;
            req.b_en = 0;
            assert(req.randomize());


            finish_item(req);
        end
        `uvm_info(get_full_name(), $sformatf("Done generation of %0d items", cfg.item_count_a_op), UVM_LOW)

    endtask

endclass : my_sequence_a_op
```

SEQUENCE

Sequence b_op

- The sequence turn off randomization for a_en and b_en and sets them to 0 and 1 respectively to enable the b_op set.
- It get the item count from configuration object set in the test.

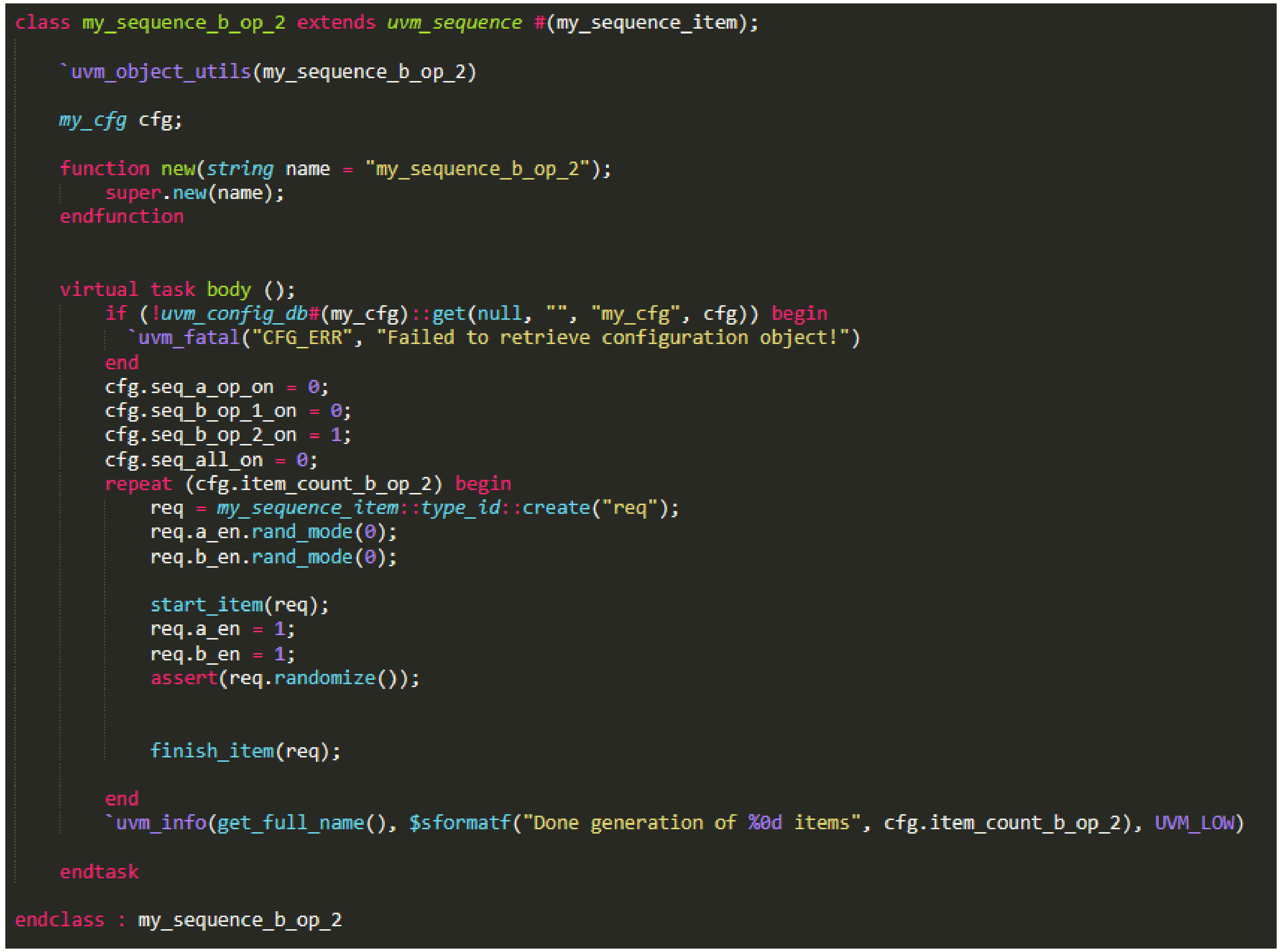


```
class my_sequence_b_op_1 extends uvm_sequence #(my_sequence_item);

    `uvm_object_utils(my_sequence_b_op_1)

    my_cfg cfg;

    function new(string name = "my_sequence_b_op_1");
        super.new(name);
    endfunction


    virtual task body ();
        if (!uvm_config_db#(my_cfg)::get(null, "", "my_cfg", cfg)) begin
          `uvm_fatal("CFG_ERR", "Failed to retrieve configuration object!")
        end
        cfg.seq_a_op_on = 0;
        cfg.seq_b_op_1_on = 1;
        cfg.seq_b_op_2_on = 0;
        cfg.seq_all_on = 0;


        repeat (cfg.item_count_b_op_1) begin
            req = my_sequence_item::type_id::create("req");
            req.a_en.rand_mode(0);
            req.b_en.rand_mode(0);

            start_item(req);
            req.a_en = 0;
            req.b_en = 1;
            assert(req.randomize());


                finish_item(req);

        end
        `uvm_info(get_full_name(), $sformatf("Done generation of %0d items", cfg.item_count_b_op_1), UVM_LOW)

    endtask

endclass : my_sequence_b_op_1
```

SEQUENCE

Sequence b_op

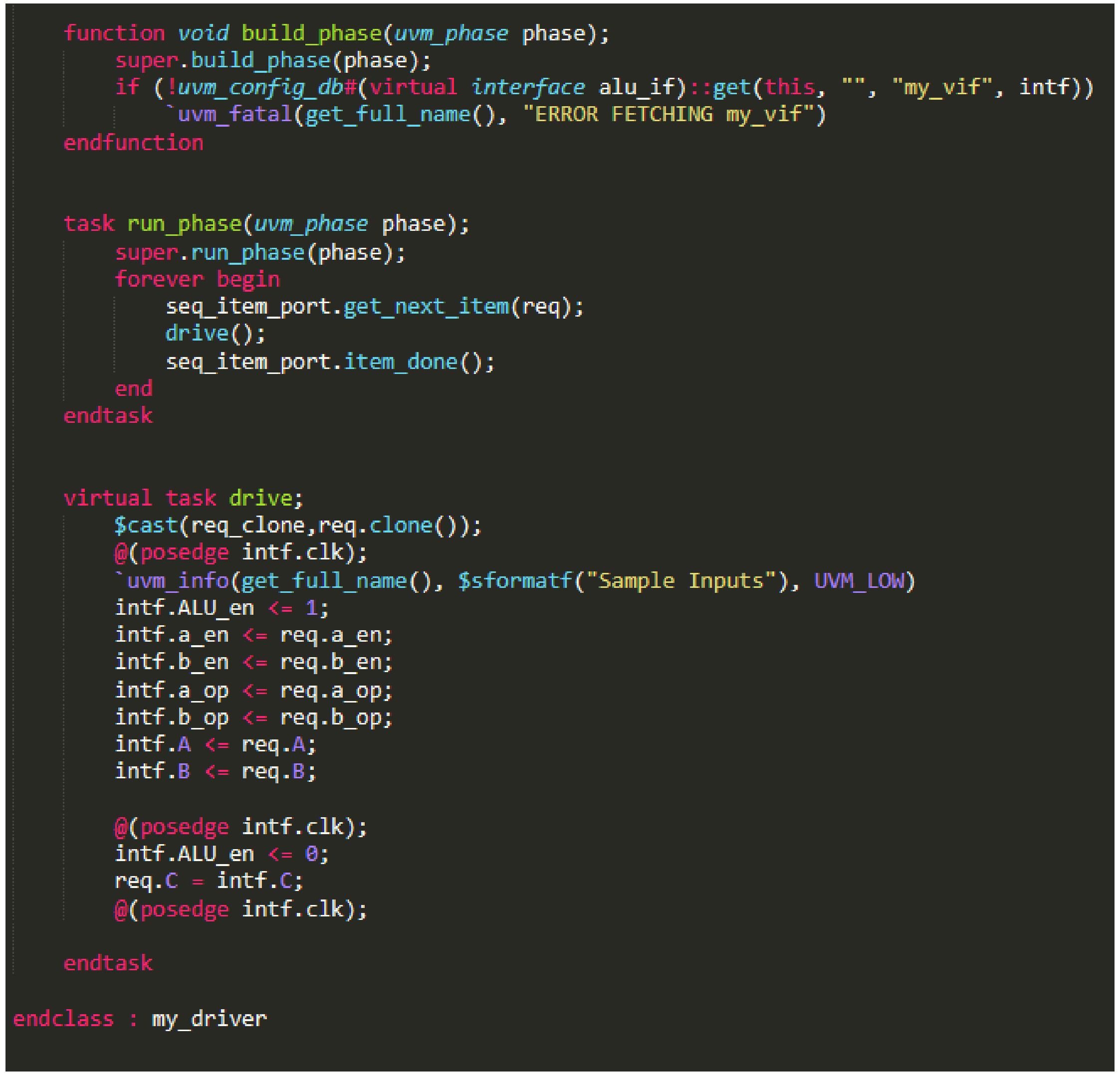
- The sequence turn off randomization for a_en and b_en and sets them to 1 and 1 respectively to enable the b_op set.
- It get the item count from configuration object set in the test.



```
class my_sequence_b_op_2 extends uvm_sequence #(my_sequence_item);

    `uvm_object_utils(my_sequence_b_op_2)

    my_cfg cfg;

    function new(string name = "my_sequence_b_op_2");
        super.new(name);
    endfunction


    virtual task body ();
        if (!uvm_config_db#(my_cfg)::get(null, "", "my_cfg", cfg)) begin
          `uvm_fatal("CFG_ERR", "Failed to retrieve configuration object!")
        end
        cfg.seq_a_op_on = 0;
        cfg.seq_b_op_1_on = 0;
        cfg.seq_b_op_2_on = 1;
        cfg.seq_all_on = 0;
        repeat (cfg.item_count_b_op_2) begin
            req = my_sequence_item::type_id::create("req");
            req.a_en.rand_mode(0);
            req.b_en.rand_mode(0);

            start_item(req);
            req.a_en = 1;
            req.b_en = 1;
            assert(req.randomize());


            finish_item(req);

        end
        `uvm_info(get_full_name(), $sformatf("Done generation of %0d items", cfg.item_count_b_op_2), UVM_LOW)

    endtask

endclass : my_sequence_b_op_2
```

DRIVER

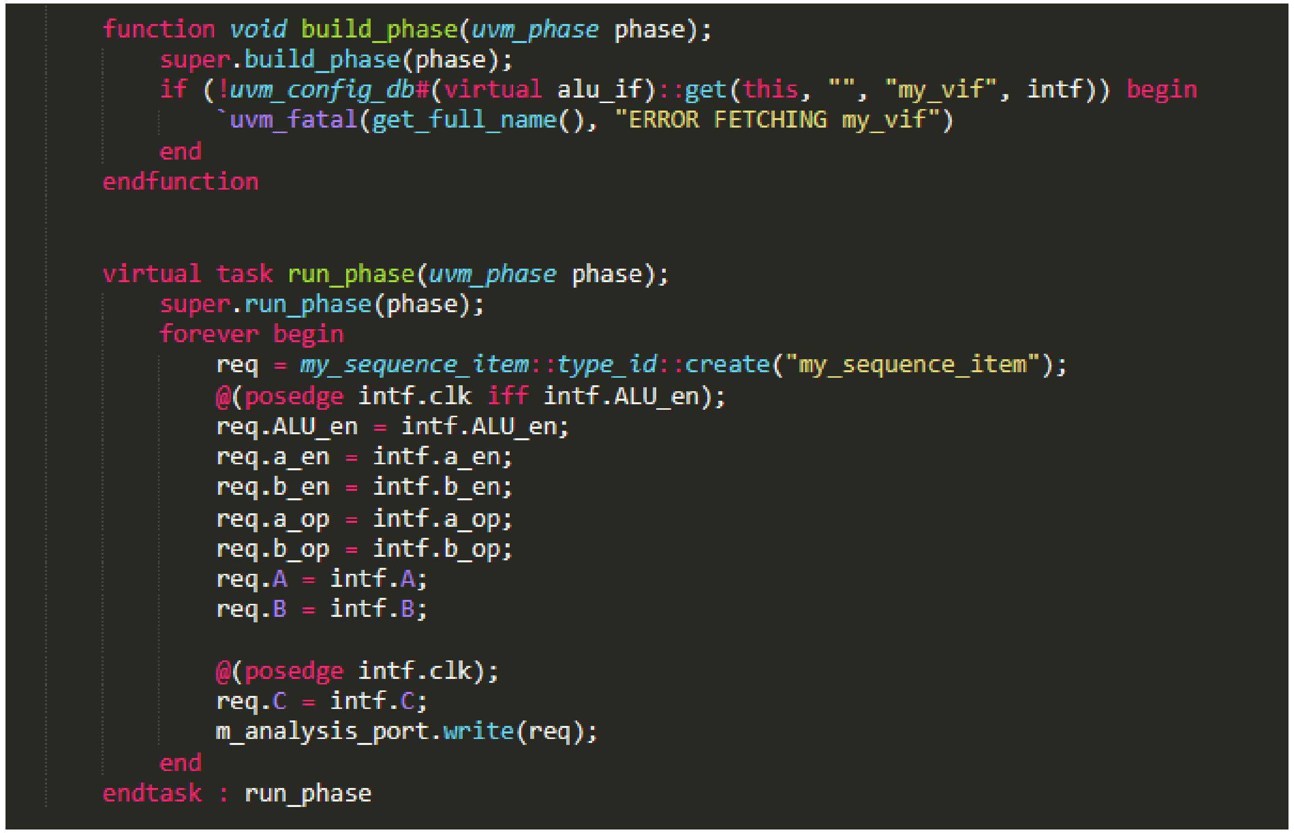
The driver transforms from transaction level to pin level.



```
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual interface alu_if)::get(this, "", "my_vif", intf))
            `uvm_fatal(get_full_name(), "ERROR FETCHING my_vif")
    endfunction


    task run_phase(uvm_phase phase);
        super.run_phase(phase);
        forever begin
            seq_item_port.get_next_item(req);
            drive();
            seq_item_port.item_done();
        end
    endtask


    virtual task drive;
        $cast(req_clone,req.clone());
        @(posedge intf.clk);
        `uvm_info(get_full_name(), $sformatf("Sample Inputs"), UVM_LOW)
        intf.ALU_en <= 1;
        intf.a_en <= req.a_en;
        intf.b_en <= req.b_en;
        intf.a_op <= req.a_op;
        intf.b_op <= req.b_op;
        intf.A <= req.A;
        intf.B <= req.B;

        @(posedge intf.clk);
        intf.ALU_en <= 0;
        req.C = intf.C;
        @(posedge intf.clk);

    endtask

endclass : my_driver
```

MONITOR

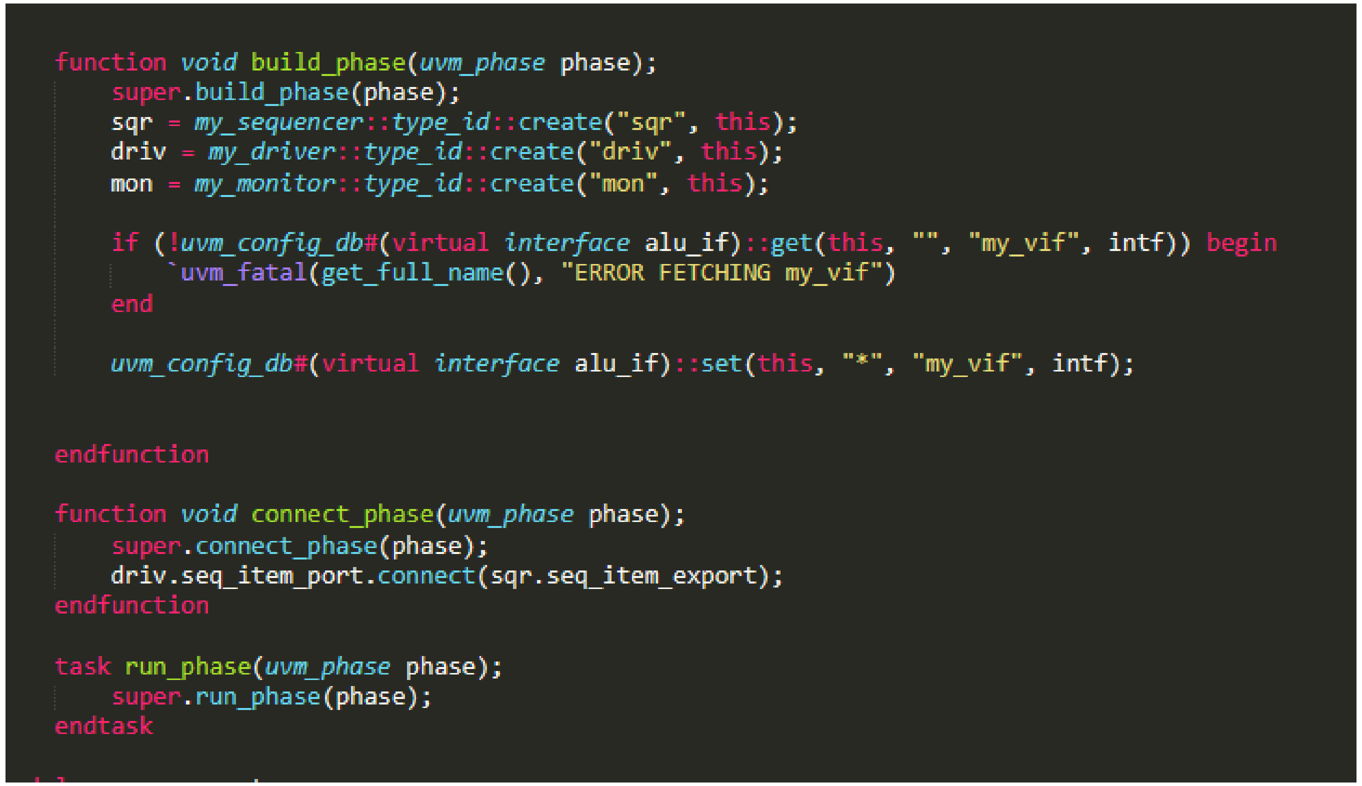
The Monitor is responsible for sampling interface values and assigning them to sequence item which will be sent to Scoreboard and Coverage collector classes.



```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual alu_if)::get(this, "", "my_vif", intf)) begin
        `uvm_fatal(get_full_name(), "ERROR FETCHING my_vif")
    end
endfunction


virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        req = my_sequence_item::type_id::create("my_sequence_item");
        @(posedge intf.clk iff intf.ALU_en);
        req.ALU_en = intf.ALU_en;
        req.a_en = intf.a_en;
        req.b_en = intf.b_en;
        req.a_op = intf.a_op;
        req.b_op = intf.b_op;
        req.A = intf.A;
        req.B = intf.B;

        @(posedge intf.clk);
        req.C = intf.C;
        m_analysis_port.write(req);
    end
endtask : run_phase
```

AGENT

It encapsulates driver, monitor, and sequencer.



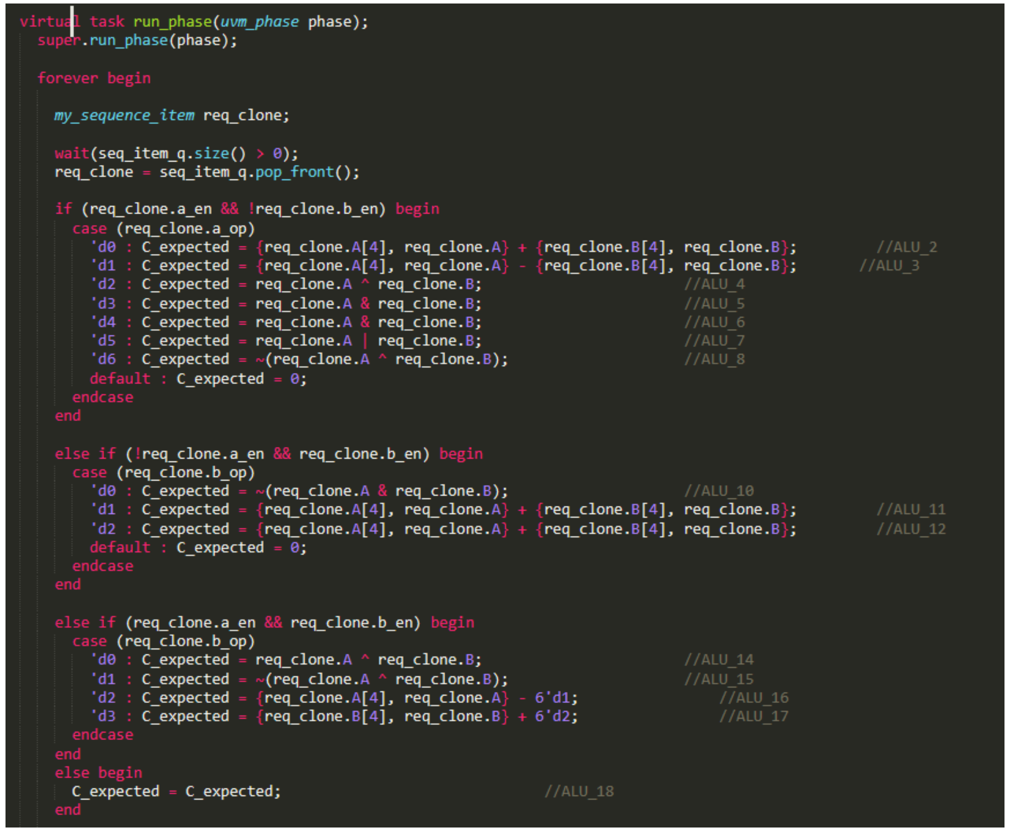
```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    sqr = my_sequencer::type_id::create("sqr", this);
    driv = my_driver::type_id::create("driv", this);
    mon = my_monitor::type_id::create("mon", this);

    if (!uvm_config_db#(virtual interface alu_if)::get(this, "", "my_vif", intf)) begin
        `uvm_fatal(get_full_name(), "ERROR FETCHING my_vif")
    end

    uvm_config_db#(virtual interface alu_if)::set(this, "*", "my_vif", intf);


endfunction

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    driv.seq_item_port.connect(sqr.seq_item_export);
endfunction

task run_phase(uvm_phase phase);
    super.run_phase(phase);
endtask
```

SCOREBOARD

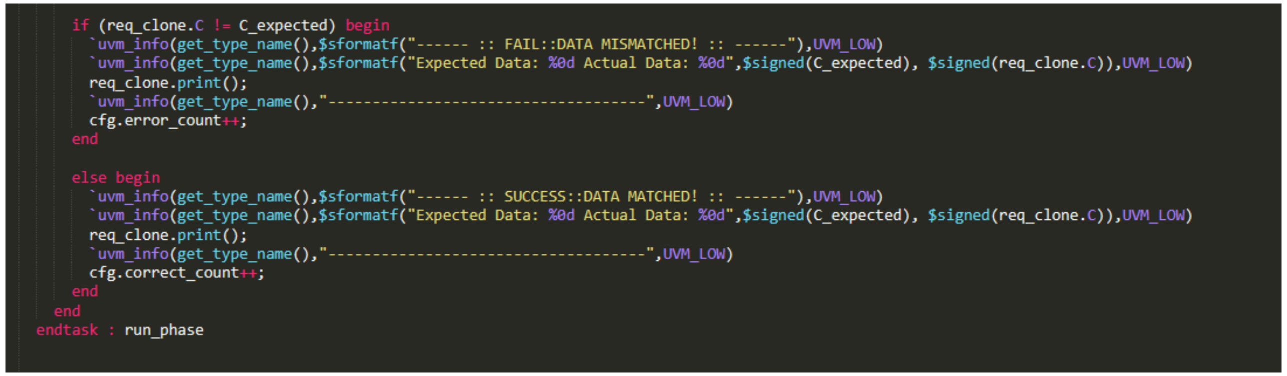
Scoreboard receives the transactions from Monitor class.

It samples transaction values to evaluate the expected output and check the actual output value sampled from the same transaction.

SCOREBOARD



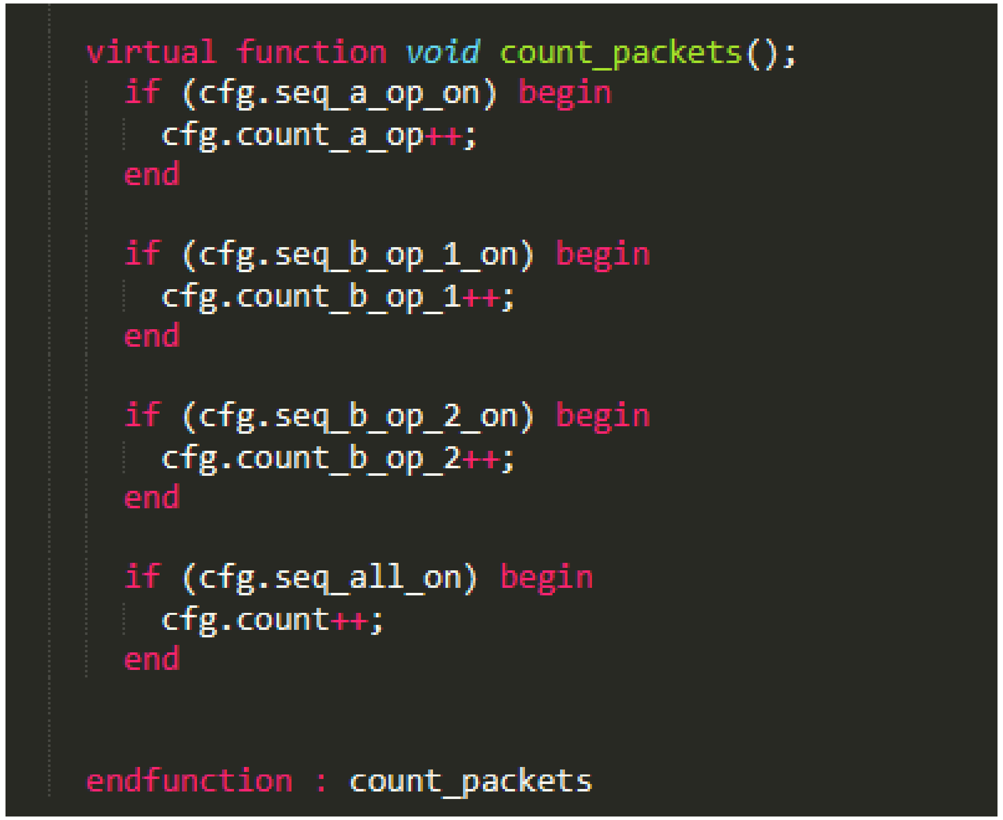
```
virtual task run_phase(uvm_phase phase);
  super.run_phase(phase);

  forever begin

    my_sequence_item req_clone;

    wait(seq_item_q.size() > 0);
    req_clone = seq_item_q.pop_front();

    if (req_clone.a_en && !req_clone.b_en) begin
      case (req_clone.a_op)
        'd0 : C_expected = {req_clone.A[4], req_clone.A} + {req_clone.B[4], req_clone.B};         //ALU_2
        'd1 : C_expected = {req_clone.A[4], req_clone.A} - {req_clone.B[4], req_clone.B};       //ALU_3
        'd2 : C_expected = req_clone.A ^ req_clone.B;                        //ALU_4
        'd3 : C_expected = req_clone.A & req_clone.B;                        //ALU_5
        'd4 : C_expected = req_clone.A & req_clone.B;                        //ALU_6
        'd5 : C_expected = req_clone.A | req_clone.B;                        //ALU_7
        'd6 : C_expected = ~(req_clone.A ^ req_clone.B);                     //ALU_8
        default : C_expected = 0;
      endcase
    end

    else if (!req_clone.a_en && req_clone.b_en) begin
      case (req_clone.b_op)
        'd0 : C_expected = ~(req_clone.A & req_clone.B);                     //ALU_10
        'd1 : C_expected = {req_clone.A[4], req_clone.A} + {req_clone.B[4], req_clone.B};         //ALU_11
        'd2 : C_expected = {req_clone.A[4], req_clone.A} + {req_clone.B[4], req_clone.B};         //ALU_12
        default : C_expected = 0;
      endcase
    end

    else if (req_clone.a_en && req_clone.b_en) begin
      case (req_clone.b_op)
        'd0 : C_expected = req_clone.A ^ req_clone.B;                        //ALU_14
        'd1 : C_expected = ~(req_clone.A ^ req_clone.B);                     //ALU_15
        'd2 : C_expected = {req_clone.A[4], req_clone.A} - 6'd1;                 //ALU_16
        'd3 : C_expected = {req_clone.B[4], req_clone.B} + 6'd2;                 //ALU_17
      endcase
    end
    else begin
      C_expected = C_expected;                                //ALU_18
    end
```



```
    if (req_clone.C != C_expected) begin
      `uvm_info(get_type_name(),$sformatf("------ :: FAIL::DATA MISMATCHED! :: ------"),UVM_LOW)
      `uvm_info(get_type_name(),$sformatf("Expected Data: %0d Actual Data: %0d",$signed(C_expected), $signed(req_clone.C)),UVM_LOW)
      req_clone.print();
      `uvm_info(get_type_name(),"------------------------------------",UVM_LOW)
      cfg.error_count++;
    end

    else begin
      `uvm_info(get_type_name(),$sformatf("------ :: SUCCESS::DATA MATCHED! :: ------"),UVM_LOW)
      `uvm_info(get_type_name(),$sformatf("Expected Data: %0d Actual Data: %0d",$signed(C_expected), $signed(req_clone.C)),UVM_LOW)
      req_clone.print();
      `uvm_info(get_type_name(),"------------------------------------",UVM_LOW)
      cfg.correct_count++;
    end
  end
endtask : run_phase
```

SCOREBOARD

The scoreboard increments the number of incoming packets from each sequence to verify all packets are received by the scoreboard.



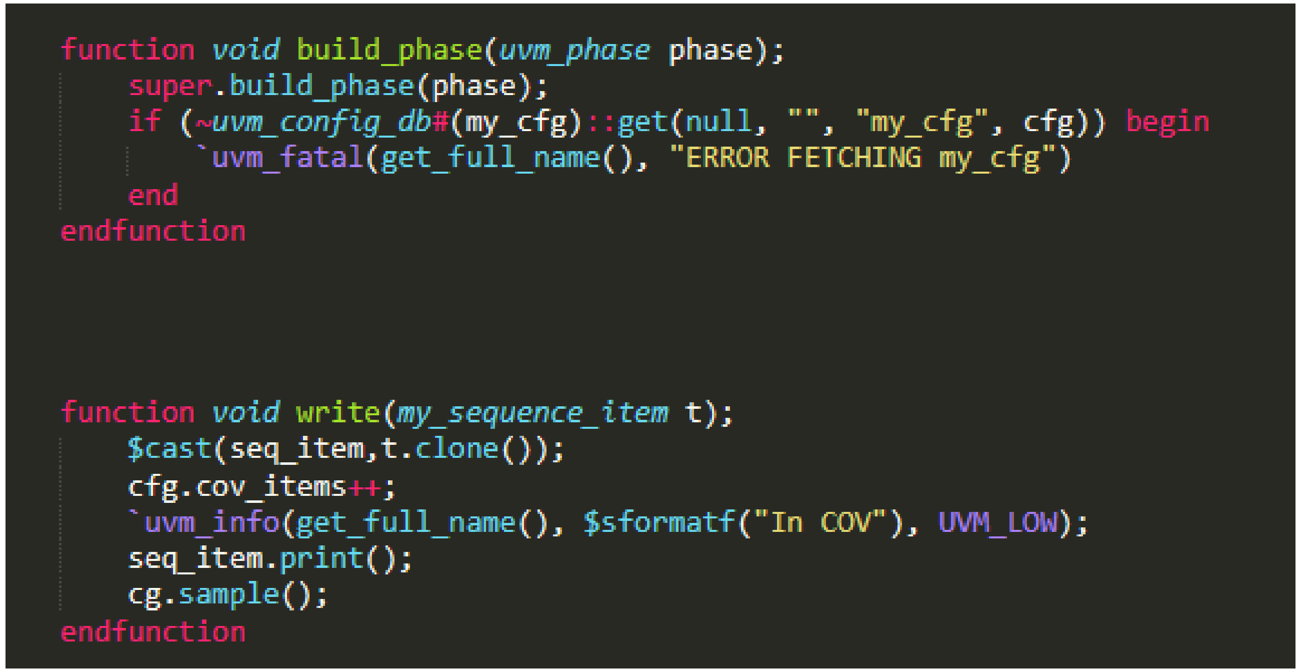
```
virtual function void count_packets();
  if (cfg.seq_a_op_on) begin
    cfg.count_a_op++;
  end

  if (cfg.seq_b_op_1_on) begin
    cfg.count_b_op_1++;
  end

  if (cfg.seq_b_op_2_on) begin
    cfg.count_b_op_2++;
  end

  if (cfg.seq_all_on) begin
    cfg.count++;
  end


endfunction : count_packets
```

COVERAGE COLLECTOR

The class is responsible for grouping the cover points and cross coverage for the transaction values.

This ensures ALU functionality is fully checked.

The cover group is sampled when collector class receives the transaction.



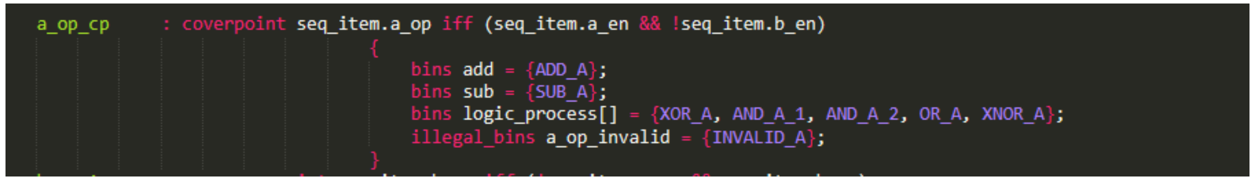
```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (~uvm_config_db#(my_cfg)::get(null, "", "my_cfg", cfg)) begin
        `uvm_fatal(get_full_name(), "ERROR FETCHING my_cfg")
    end
endfunction



function void write(my_sequence_item t);
    $cast(seq_item,t.clone());
    cfg.cov_items++;
    `uvm_info(get_full_name(), $sformatf("In COV"), UVM_LOW);
    seq_item.print();
    cg.sample();
endfunction
```

COVERAGE COLLECTOR

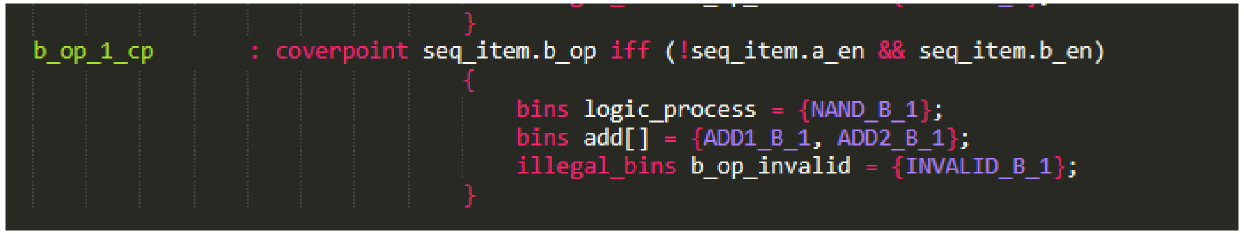
Illegal bins are defined for the prohibited operations in set A and set B1.

Furthermore, illegal bins are defined for "-32" for "C" and "-16" for both "A" and "B".

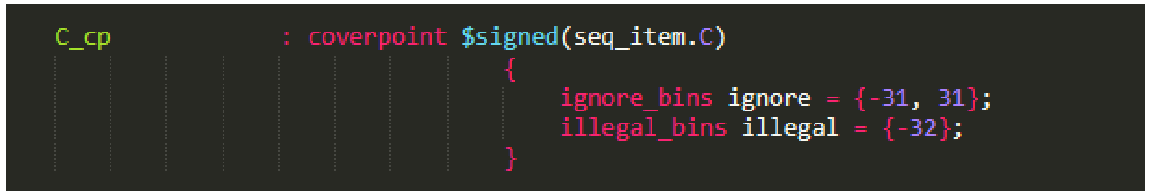
Ignore bins are defined for "-31,31" as they can not be achieved given the range of the inputs [-15, 15].



```
a_op_cp     : coverpoint seq_item.a_op iff (seq_item.a_en && !seq_item.b_en)
                        {
                            bins add = {ADD_A};
                            bins sub = {SUB_A};
                            bins logic_process[] = {XOR_A, AND_A_1, AND_A_2, OR_A, XNOR_A};
                            illegal_bins a_op_invalid = {INVALID_A};
                        }
```



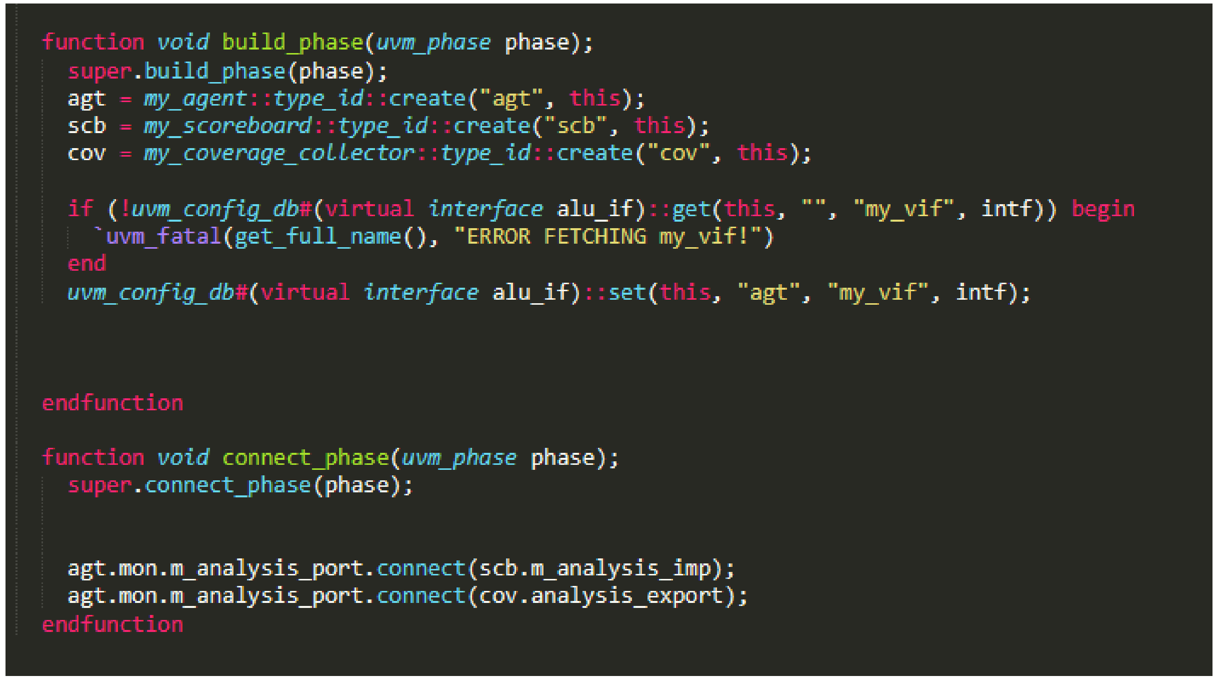
```
                        }
b_op_1_cp       : coverpoint seq_item.b_op iff (!seq_item.a_en && seq_item.b_en)
                        {
                            bins logic_process = {NAND_B_1};
                            bins add[] = {ADD1_B_1, ADD2_B_1};
                            illegal_bins b_op_invalid = {INVALID_B_1};
                        }
```



```
C_cp            : coverpoint $signed(seq_item.C)
                        {
                            ignore_bins ignore = {-31, 31};
                            illegal_bins illegal = {-32};
                        }
```

ENVIRONMENT

Environment class encapsulates Agent, Scoreboard, and Coverage Collector classes.

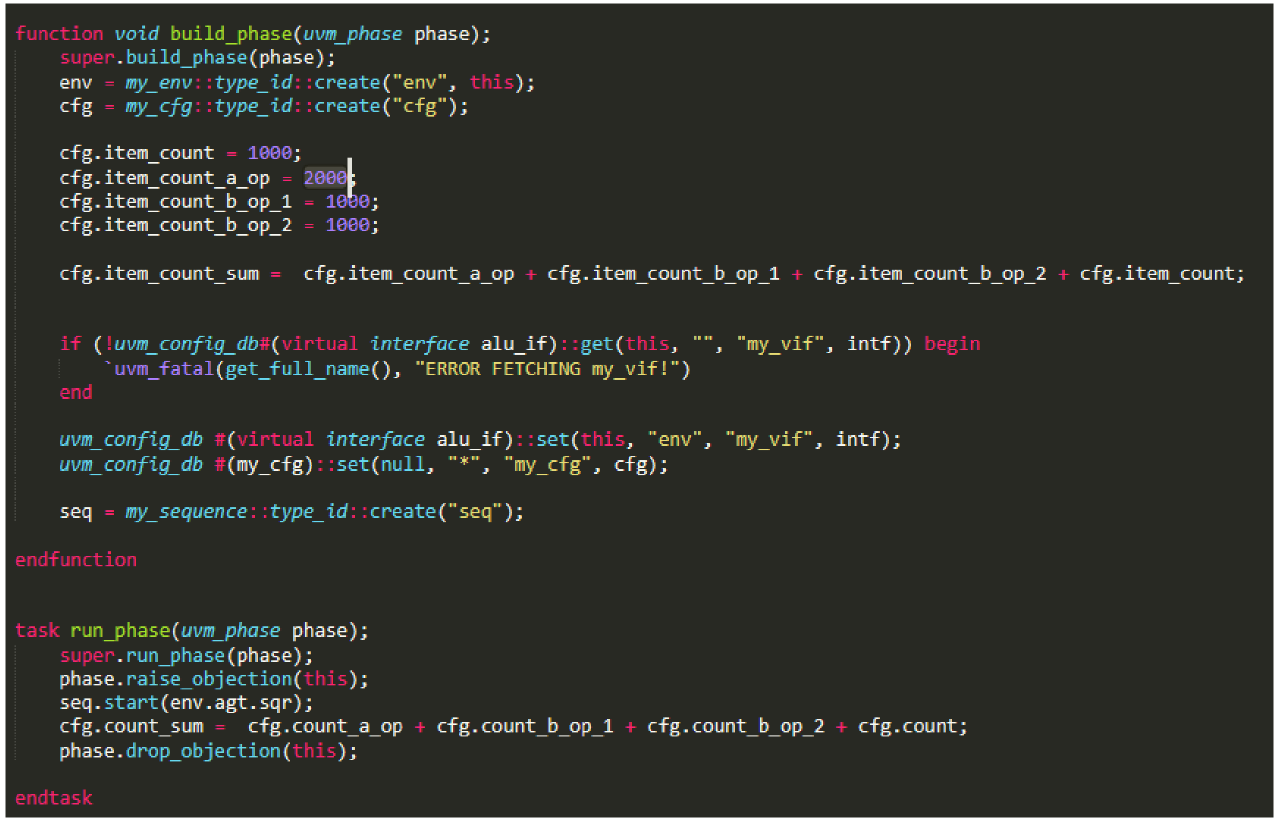


```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  agt = my_agent::type_id::create("agt", this);
  scb = my_scoreboard::type_id::create("scb", this);
  cov = my_coverage_collector::type_id::create("cov", this);

  if (!uvm_config_db#(virtual interface alu_if)::get(this, "", "my_vif", intf)) begin
    `uvm_fatal(get_full_name(), "ERROR FETCHING my_vif!")
  end
  uvm_config_db#(virtual interface alu_if)::set(this, "agt", "my_vif", intf);



endfunction

function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);


  agt.mon.m_analysis_port.connect(scb.m_analysis_imp);
  agt.mon.m_analysis_port.connect(cov.analysis_export);
endfunction
```

TEST

Test program creates the Environment class and starts the intended sequence.

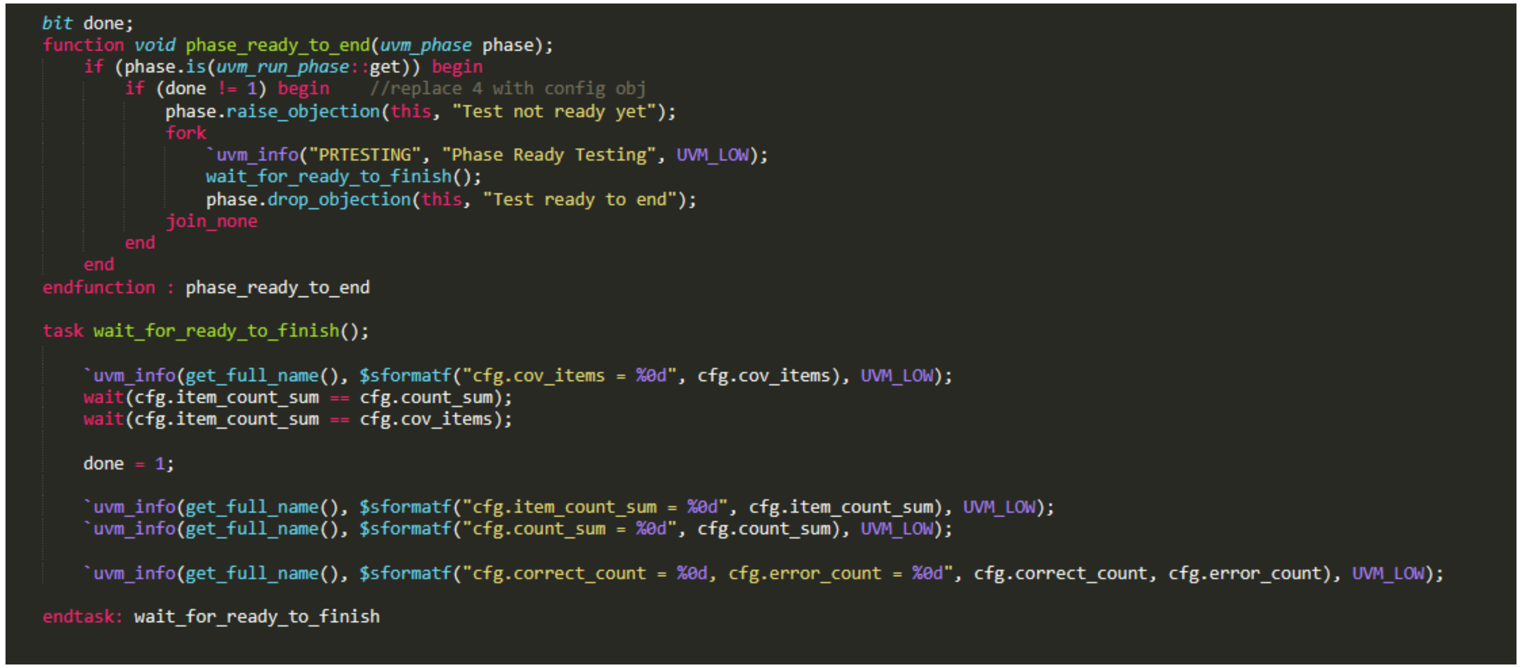
The number of packets in each sequence are determined in build phase.



```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = my_env::type_id::create("env", this);
    cfg = my_cfg::type_id::create("cfg");

    cfg.item_count = 1000;
    cfg.item_count_a_op = 2000;
    cfg.item_count_b_op_1 = 1000;
    cfg.item_count_b_op_2 = 1000;

    cfg.item_count_sum =  cfg.item_count_a_op + cfg.item_count_b_op_1 + cfg.item_count_b_op_2 + cfg.item_count;


    if (!uvm_config_db#(virtual interface alu_if)::get(this, "", "my_vif", intf)) begin
        `uvm_fatal(get_full_name(), "ERROR FETCHING my_vif!")
    end

    uvm_config_db #(virtual interface alu_if)::set(this, "env", "my_vif", intf);
    uvm_config_db #(my_cfg)::set(null, "*", "my_cfg", cfg);

    seq = my_sequence::type_id::create("seq");

endfunction


task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this);
    seq.start(env.agt.sqr);
    cfg.count_sum =  cfg.count_a_op + cfg.count_b_op_1 + cfg.count_b_op_2 + cfg.count;
    phase.drop_objection(this);

endtask
```

TEST

Phase ready to end function is used to ensure the test does not end before all packets are processed.



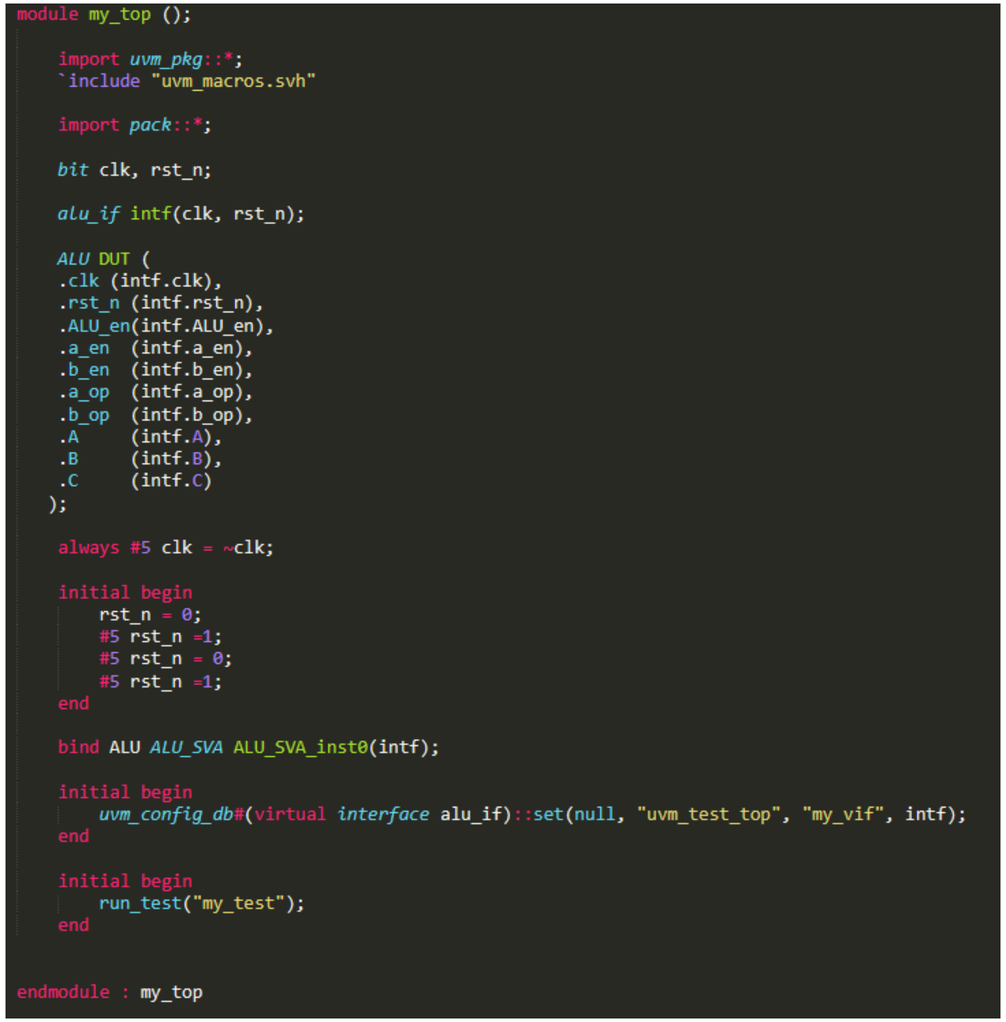
```
bit done;
function void phase_ready_to_end(uvm_phase phase);
    if (phase.is(uvm_run_phase::get)) begin
        if (done != 1) begin    //replace 4 with config obj
            phase.raise_objection(this, "Test not ready yet");
            fork
                `uvm_info("PRTESTING", "Phase Ready Testing", UVM_LOW);
                wait_for_ready_to_finish();
                phase.drop_objection(this, "Test ready to end");
            join_none
        end
    end
endfunction : phase_ready_to_end

task wait_for_ready_to_finish();

    `uvm_info(get_full_name(), $sformatf("cfg.cov_items = %0d", cfg.cov_items), UVM_LOW);
    wait(cfg.item_count_sum == cfg.count_sum);
    wait(cfg.item_count_sum == cfg.cov_items);

    done = 1;

    `uvm_info(get_full_name(), $sformatf("cfg.item_count_sum = %0d", cfg.item_count_sum), UVM_LOW);
    `uvm_info(get_full_name(), $sformatf("cfg.count_sum = %0d", cfg.count_sum), UVM_LOW);

    `uvm_info(get_full_name(), $sformatf("cfg.correct_count = %0d, cfg.error_count = %0d", cfg.correct_count, cfg.error_count), UVM_LOW);

endtask: wait_for_ready_to_finish
```

TOP

Top module initiates DUT, Test, and interface.

System clock is generated and passed to the interface as an input.

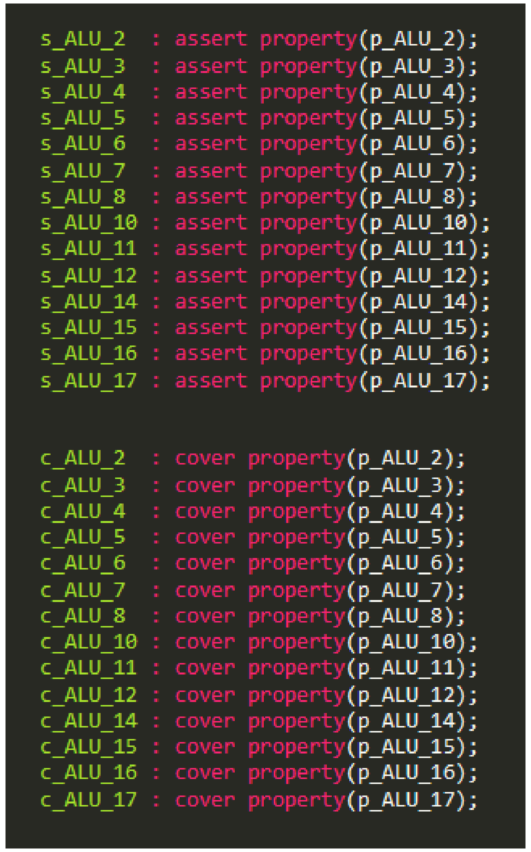
Bind is used to initiate ALU_SVA module in ALU module.



```
module my_top ();

    import uvm_pkg::*;
    `include "uvm_macros.svh"

    import pack::*;

    bit clk, rst_n;

    alu_if intf(clk, rst_n);

    ALU DUT (
    .clk (intf.clk),
    .rst_n (intf.rst_n),
    .ALU_en(intf.ALU_en),
    .a_en  (intf.a_en),
    .b_en  (intf.b_en),
    .a_op  (intf.a_op),
    .b_op  (intf.b_op),
    .A     (intf.A),
    .B     (intf.B),
    .C     (intf.C)
   );

    always #5 clk = ~clk;

    initial begin
        rst_n = 0;
        #5 rst_n =1;
        #5 rst_n = 0;
        #5 rst_n =1;
    end

    bind ALU ALU_SVA ALU_SVA_inst0(intf);

    initial begin
        uvm_config_db#(virtual interface alu_if)::set(null, "uvm_test_top", "my_vif", intf);
    end

    initial begin
        run_test("my_test");
    end


endmodule : my_top
```

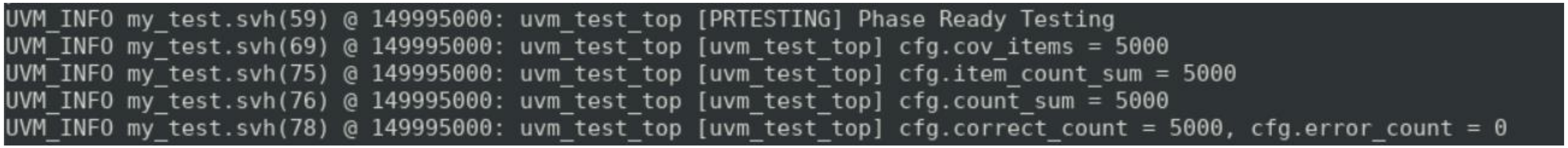
ASSERTIONS

Assertions module is created to group multiple properties of the design to fully ensure the properties of the design are covered.



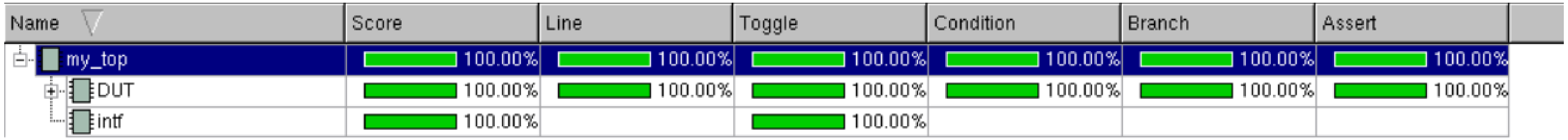
```
s_ALU_2  : assert property(p_ALU_2);
s_ALU_3  : assert property(p_ALU_3);
s_ALU_4  : assert property(p_ALU_4);
s_ALU_5  : assert property(p_ALU_5);
s_ALU_6  : assert property(p_ALU_6);
s_ALU_7  : assert property(p_ALU_7);
s_ALU_8  : assert property(p_ALU_8);
s_ALU_10 : assert property(p_ALU_10);
s_ALU_11 : assert property(p_ALU_11);
s_ALU_12 : assert property(p_ALU_12);
s_ALU_14 : assert property(p_ALU_14);
s_ALU_15 : assert property(p_ALU_15);
s_ALU_16 : assert property(p_ALU_16);
s_ALU_17 : assert property(p_ALU_17);


c_ALU_2  : cover property(p_ALU_2);
c_ALU_3  : cover property(p_ALU_3);
c_ALU_4  : cover property(p_ALU_4);
c_ALU_5  : cover property(p_ALU_5);
c_ALU_6  : cover property(p_ALU_6);
c_ALU_7  : cover property(p_ALU_7);
c_ALU_8  : cover property(p_ALU_8);
c_ALU_10 : cover property(p_ALU_10);
c_ALU_11 : cover property(p_ALU_11);
c_ALU_12 : cover property(p_ALU_12);
c_ALU_14 : cover property(p_ALU_14);
c_ALU_15 : cover property(p_ALU_15);
c_ALU_16 : cover property(p_ALU_16);
c_ALU_17 : cover property(p_ALU_17);
```

SIMULATION RESULTS



```
UVM_INFO my_test.svh(59) @ 149995000: uvm_test_top [PRTESTING] Phase Ready Testing
UVM_INFO my_test.svh(69) @ 149995000: uvm_test_top [uvm_test_top] cfg.cov_items = 5000
UVM_INFO my_test.svh(75) @ 149995000: uvm_test_top [uvm_test_top] cfg.item_count_sum = 5000
UVM_INFO my_test.svh(76) @ 149995000: uvm_test_top [uvm_test_top] cfg.count_sum = 5000
UVM_INFO my_test.svh(78) @ 149995000: uvm_test_top [uvm_test_top] cfg.correct_count = 5000, cfg.error_count = 0
```

ASSERTIONS AND COVERAGE CLOSURE



Name	Score	Line	Toggle	Condition	Branch	Assert
my_top	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
DUT	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
intf	100.00%		100.00%			