ALU Verification Document

Index

Introduction

This document outlines the verification process for an Arithmetic Logic Unit (ALU) design, ensuring that the implemented functionality adheres to the specified requirements. The ALU is a critical component in digital systems, responsible for performing arithmetic and logical operations, and its correct operation is essential for overall system reliability.

The verification process involves a thorough review of the design specification, development of a comprehensive testbench, and execution of test cases to validate the ALU's functionality under various scenarios. The testbench will include randomized tests to cover all conditions.

Key Objectives

Key objectives of the project are :-

- Understanding the ALU design specification, including supported operations (e.g., addition, subtraction, AND, OR, XOR, shifts) and control signals.
- Developing a structured testbench using a hardware verification language (e.g., SystemVerilog) with assertions and coverage metrics.
- Executing functional verification to confirm that all operations produce expected results.
- Analyzing coverage (code, functional, and assertion coverage) to ensure all design aspects are thoroughly tested.

DUT INTERFACES

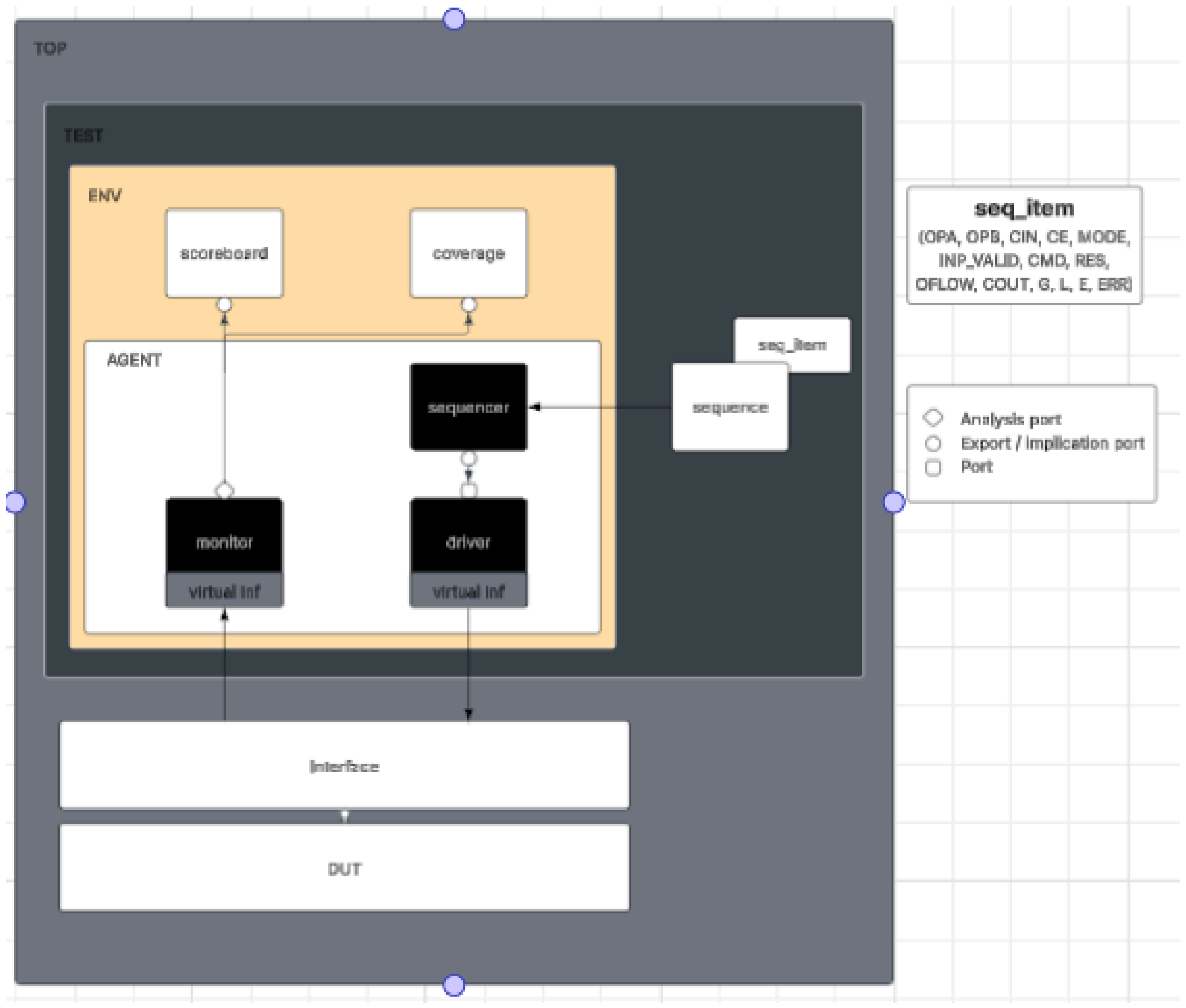
The DUT consists of the following input and output pins.

PIN name	Direction	Width	Description
OPA	INPUT	Parameterized	Parameterized operand 1
OPB	INPUT	Parameterized	Parameterized operand 2
CIN	INPUT	1	This is the active high carry in input signal of 1-bit.
CLK	INPUT	1	This is the clock signal to the design and it is edge

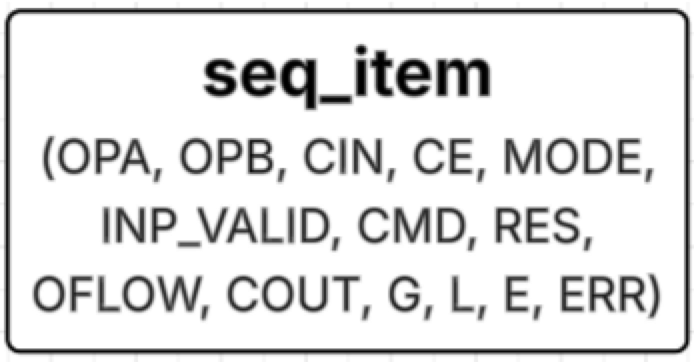
			sensitive.
RST	INPUT	1	This is the active high asynchronous reset to the design.
CE	INPUT	1	This is the active high clock enable signal 1 bit.
MODE	INPUT	1	MODE signal 1 bit is high, then this is an Arithmetic Operation otherwise it is logical operation
INP_VALID INPUT	INPUT	2	Operands are valid as per below table :- 00 : No operand is valid. 01: Operand A is valid. 10: Operand B is valid. 11: Both Operands are valid.
CMD	INPUT	4	Selects the command to be executed.
RES	OUTPUT	Parameterized + 1	This is the total parameterized plus 1 bits result of the instruction performed by the ALU.
OFLOW	OUTPUT	1	This 1-bit signal indicates an output overflow, during Addition/Subtraction
COUT	OUTPUT	1	This is the carry out signal of 1-bit, during Addition/Subtraction
G	OUTPUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is greater than the value of OPB.

L	OUTPUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is lesser than the value of OPB/
E	OUTPUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is equal to the value of OPB.
ERR	OUTPUT	1	When Cmd is selected as 12 or 13 and mode is logical operation , if 4th ,5th ,6th and 7th bit of OPB are 1, then ERR bit will be 1 else it is high impedance .

Testbench Architecture



Sequence_Items (seq_item)

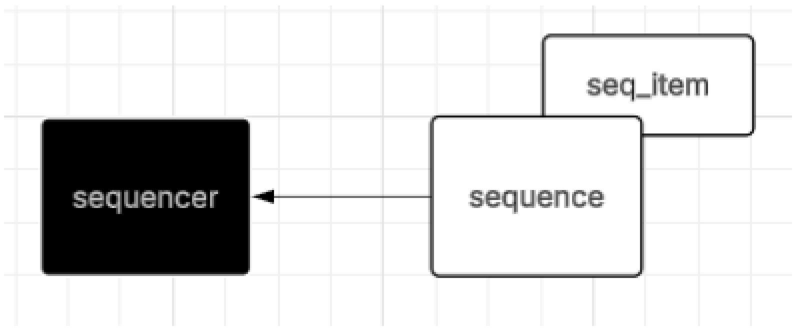

seq_item
(OPA, OPB, CIN, CE, MODE, INP_VALID, CMD, RES, OFLOW, COUT, G, L, E, ERR)

Sequence items (seq_item) are user defined class objects that encapsulate all input and output signals of the ALU. They serve as the primary data structure passed between testbench components (via TLM ports).
They consist of the following signals.

1. Inputs
   - ➢ OPA, OPB (Operands)
   - ➢ CIN (Carry-in)
   - ➢ CE (Clock Enable)
   - ➢ MODE (Operation Mode)
   - ➢ CMD (Command/Opcode)

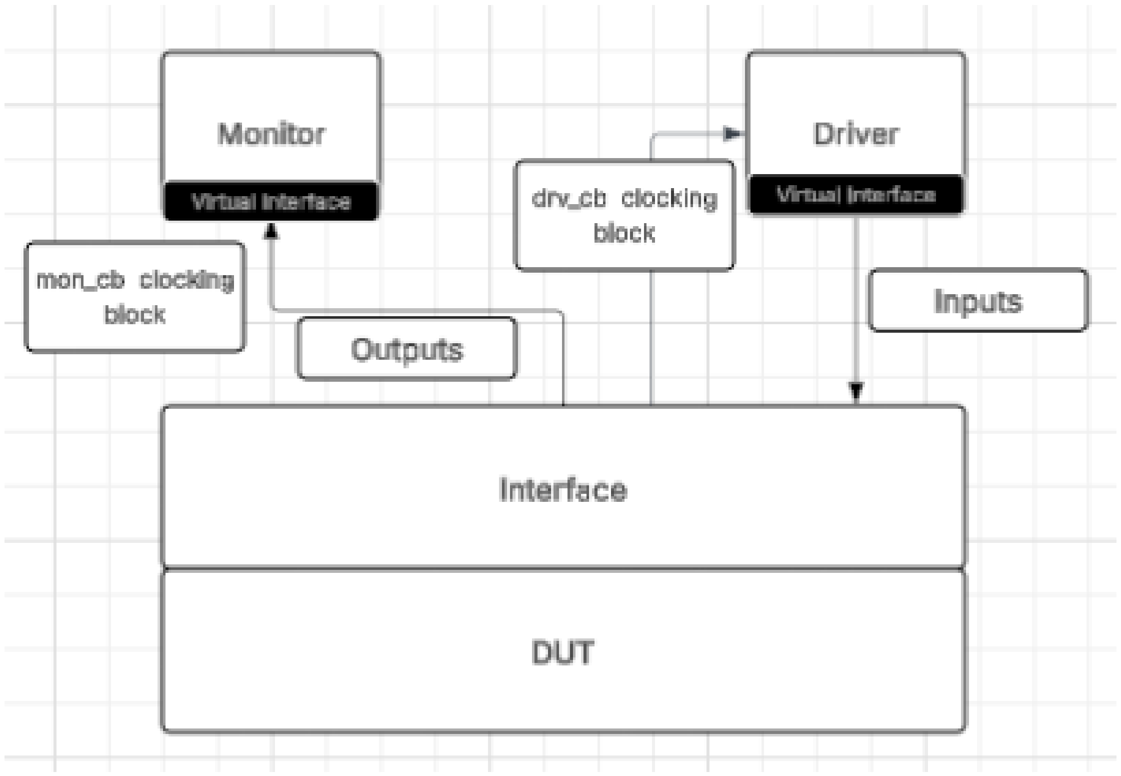
2. Outputs
   - ➢ RES (Result)
   - ➢ OFLOW (Overflow)
   - ➢ COUT (Carry-out)
   - ➢ G, L, E (Greater, Less, Equal flags)
   - ➢ ERR (Error flag

Sequence



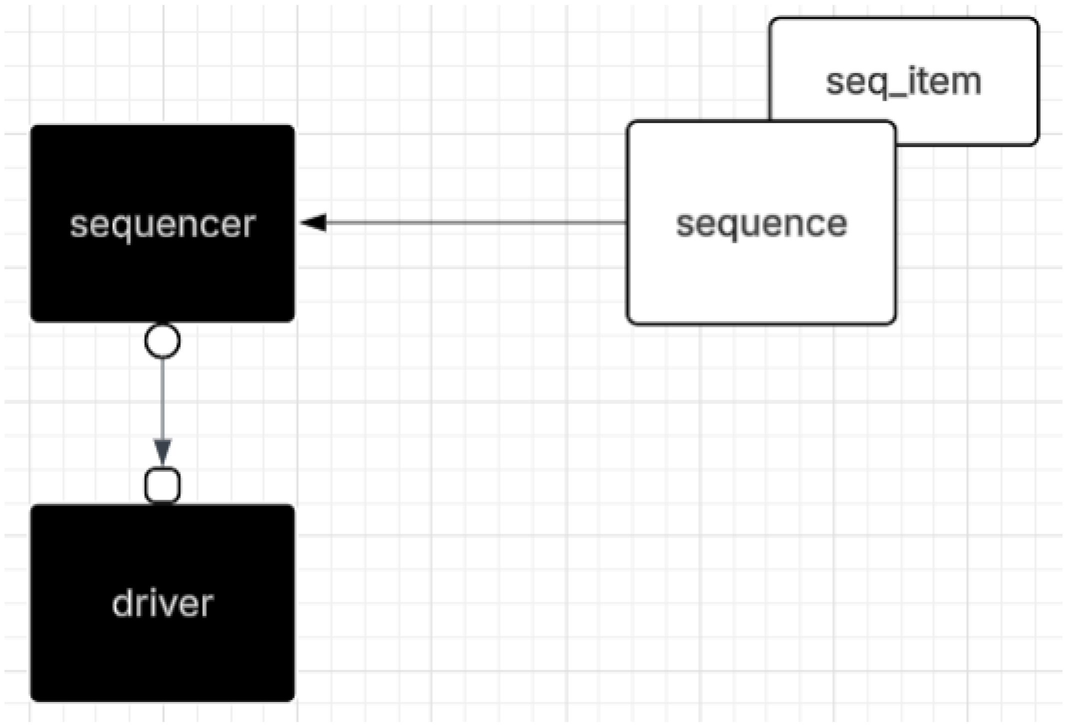
- Generates the stimulus containing randomized ALU inputs (OPA, OPB, CMD, CIN, MODE, CMD) by randomizing the seq_item's.
- Controls test variability by applying constraints to randomization (e.g., valid opcodes, corner-case operands).
- These seq_items are sent to the sequencer req fifo where they are stored.

Interface



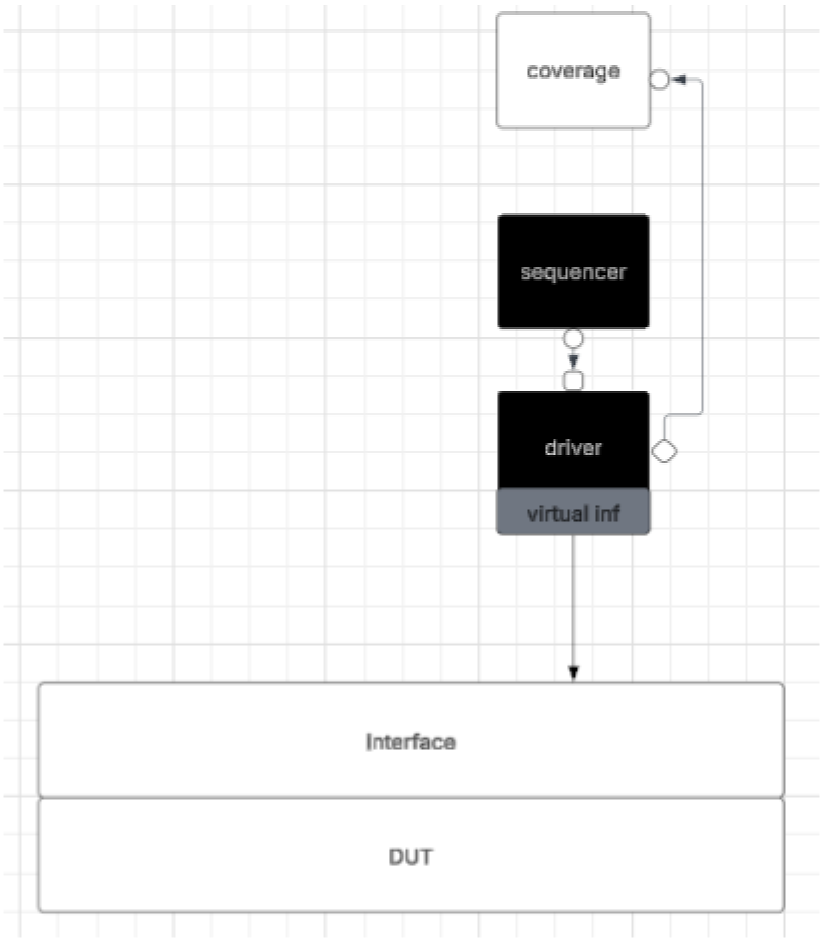
- It is used to bundle all inputs and outputs of DUT, so we don't have to instantiate DUT every time and can directly use an interface.
- It is used to synchronize different components of the testbench to work together via clocking blocks namely drv_cb (for driver), mon_cb(for monitor).
- It is used to drive the inputs from the driver to the DUT, and used to access output values from DUT to the monitor.

Sequencer



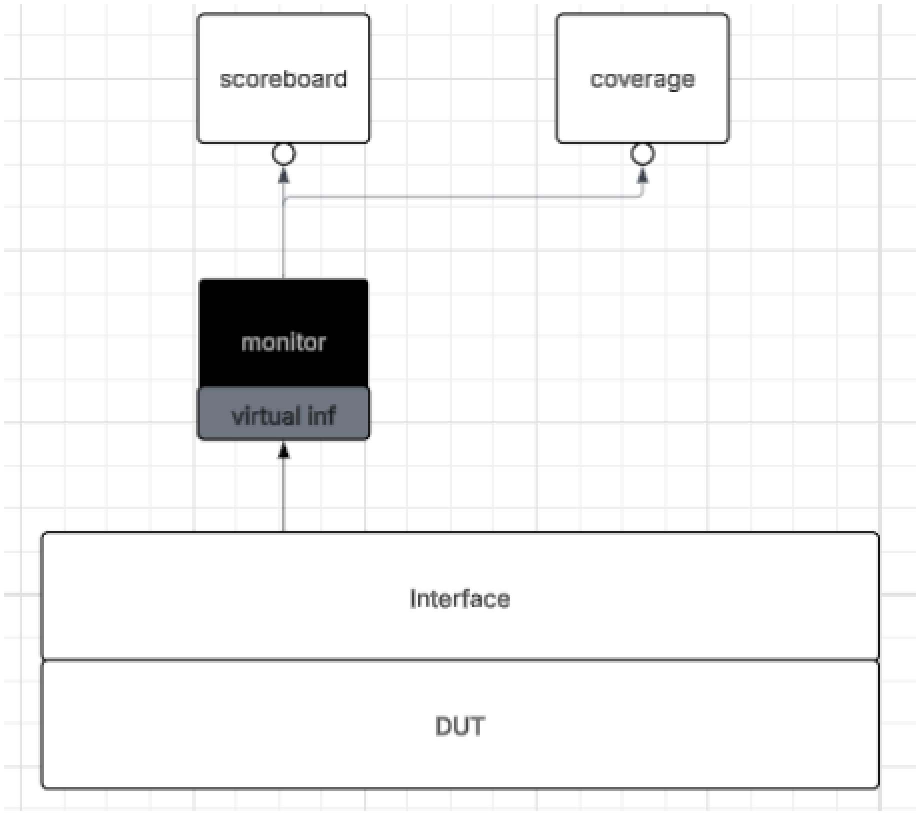
- Controls the transaction level communication between sequence and driver by establishing a connection between them.
- Ultimately, it passes transactions or sequence items to the driver so that they can be driven to the DUT.
- Sequencer is connected to sequence in the test class.
  <sequence_handle>.start(<env_handle>.<agent_handle>.<sequencer_handle>)
- The sequencer is connected to the driver with the help of TLM ports in the agent class.
  <driver_handle>.seq_item_port>.connect(<sequencer_handle>.seq_item_export).
  (seq_item_port in an in-built method of uvm_driver, which is used to request items from sequencer. seq_item_export is an inbuilt method of uvm_sequencer which is used to send seq_items to the driver.)

Driver



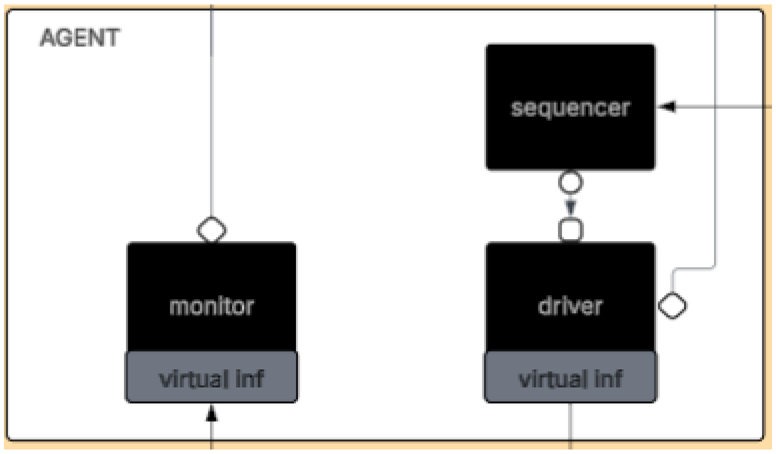
- Driver receives transactions from Generator via TLM port of sequencer.
- Driver retrieves the virtual interface from uvm_config_db. Syntax is uvm_config_db #(seq_item) ::get(this, “”, “<virtual_interface_handle>”, <virtual_interface_handle in database>).
- Drives inputs to the DUT through the virtual interface.
- Handles special cases (single operand commands, multi operand commands with wrong inp_valid, multiplication operations).
- Also has a TLM port connection to coverage component where input coverage will be measured. It is declared as such in driver class :-
  uvm_analysis_port #(<sequence_item class name>) <driver_port_handle>
  It is then connected to the coverage component’s implication port in connect_phase of env (environment class).
  <agent_handle>.<driver_handle>.<driver_port_handle>.connect(<coverage_component_handle>.<driver_coverage_port_handle>).

Monitor



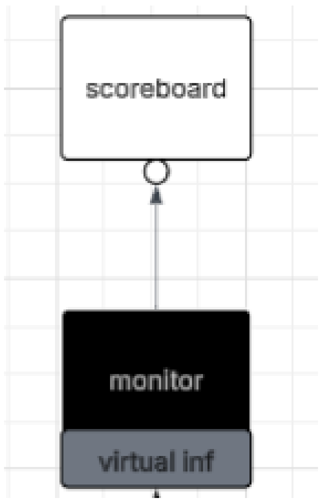
- Monitor as the name states is used to monitor the outputs from the DUT via the interface and store it in a seq_item packet (mon_item here).
- The Monitor then sends these sequence_items (seq_item) to scoreboard and coverage via TLM ports. Syntax :-
  uvm_analysis_port #(seq_item) <monitor_port_handle>
  We call a write function to send it to the scoreboard queue.
  <monitor_port_handle>.write(mon_item).
  Connected in the agent class.
- Also has a TLM connection to the coverage component which is used to track output coverage. The same port which is sending to the scoreboard is sending to the coverage component as well. It is connected in the connect phase of environment class.
  <agent_handle>.<monitor_handle>.<monitor_analysis_port_handle>.connect(<coverage_handle>.<monitor_coverage_handle>)

Agent



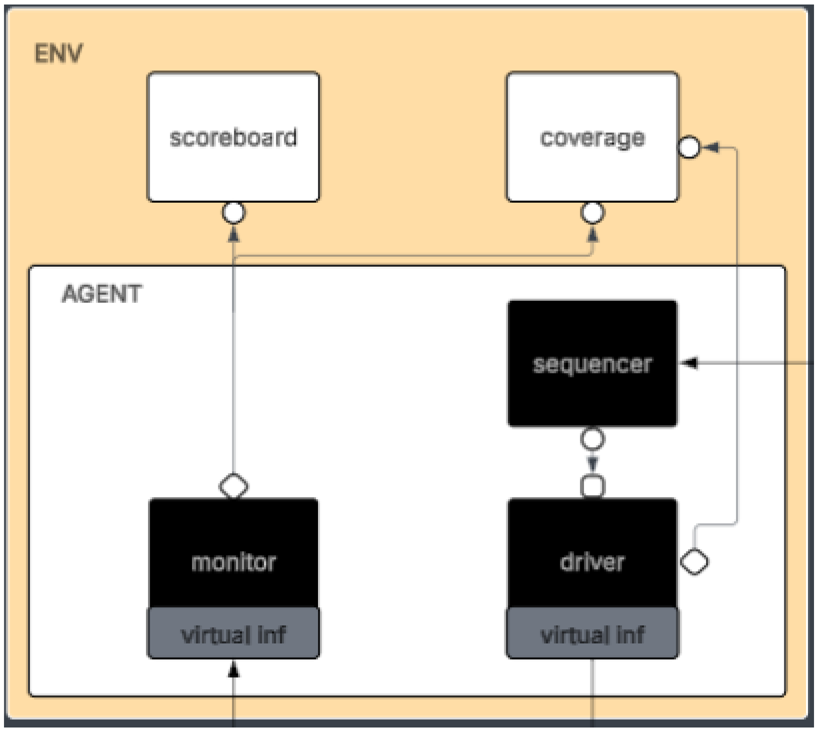
- An agent is a container that holds and connects the driver, monitor, and sequencer instances.
- The agent develops a structured hierarchy based on the protocol or interface requirement.
- Our testbench is using an active agent (driver, monitor, sequencer are all present).
- In build_phase, the agent constructs sequencer, driver and monitor.
- In connect_phase, the agent connects sequencer and driver's TLM ports.
  <driver_handle>.<seq_item_port>.connect(<sequencer_handle>.<seq_item_export>
)

Scoreboard



- Scoreboard has a dual function in uvm, it also has reference model functionality. It compares the outputs from the internal reference model with the actual outputs produced in the DUT.
- It calculates how many test cases have passed and how many have failed.
- It receives via TLM connection to the monitor. The packets received are stored in a queue. Syntax :-
  seq_item q[$];
  uvm_port_imp #(seq_item) <scoreboard_port_handle>
  We have to implement write function in scoreboard
  function void write(seq_item pkt);
      q.push_back(pkt);
  Endfunction
  In run_phase we check whether there is any packet received from monitor (q.size > 0) then calculate using internal reference model and compare both.

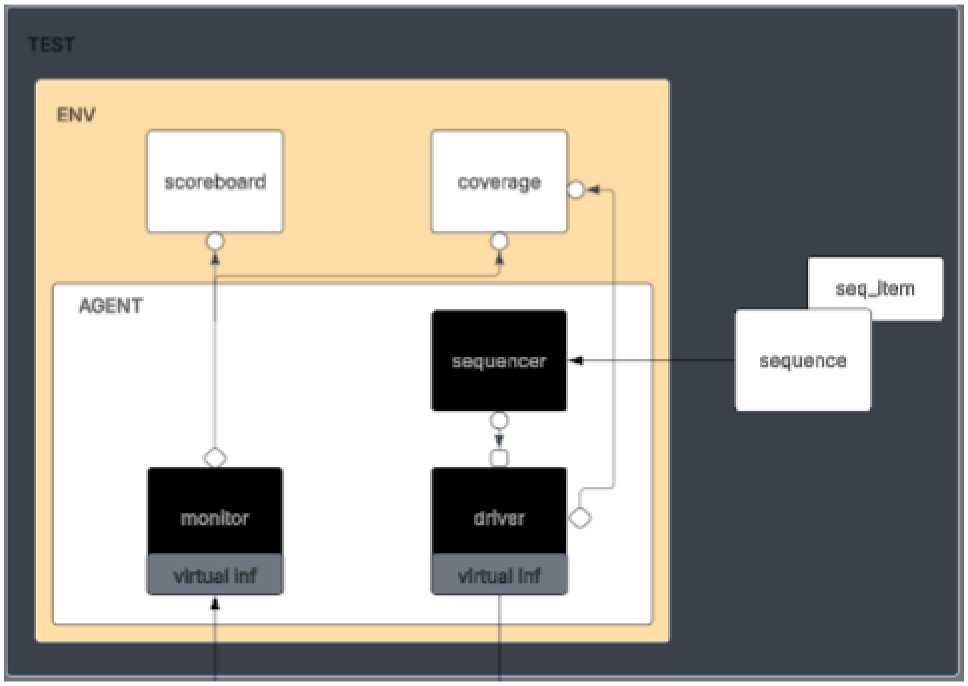
Environment (env)



- Contains the agent, scoreboard and coverage components.
- In the build_phase creates the agent, scoreboard and coverage components.
- In the connect phase, connect the TLM ports of monitor - scoreboard, driver - coverage, monitor - coverage. Syntax :-

<agent_handle>.<driver_handle>.<analysis_port_handle>.connect(<coverage_handle>.<coverage_driver_handle>)
<agent_handle>.<monitor_handle>.<analysis_port_handle>.connect(<coverage_handle>.<coverage_monitor_handle>)
<agent_handle>.<monitor_handle>.<analysis_port_handle>.connect(<scoreboard_handle>.<scoreboard_port_handle>)

Test



- Test class contains environment class and the relevant sequence class.
- In build_phase, env class is constructed.
- In run_phase, the sequence is connected to the sequencer, and generation and application of stimulus to DUT starts.

Errors in DUT Functionality

Sno	Operation	Errors
1	Add Unsigned	-
2	Subtraction Unsigned	-
3	Addition (Cin)	Cin arriving at the next clock cycle.
4	Subtraction (Cin)	Cin arriving at the next clock cycle.
5	Increment A	Wrong operation logic, Not working for inp_valid = 2’b01.
6	Decrement A	Not working for inp_valid = 2’b01
7	Increment B	Wrong operation logic, Not working for inp_valid = 2’b10.
8	Decrement B	Wrong operation logic. Not working for inp_valid = 2’b10
9	Comparision	-
10	Increment Multiplication	-
11	Shift Multiplication	Wrong operation logic
12	AND	-
13	NAND	-
14	OR	Wrong operation logic
15	NOR	-
16	XOR	-
17	XNOR	-
18	NOT of A	Not working for inp_valid = 2’b01
19	NOT of B	Not working for inp_valid = 2’b10

20	Shift Right A by 1 bit	Wrong Operation Logic, Not working for inp_valid = 2’b01.
21	Shift Left A by 1 bit	Not working for inp_valid = 2’b01.
22	Shift Right B by 1 bit	Wrong Operation Logic, Not working for inp_valid = 2’b10.
23	Shift Left B by 1 bit	Not working for inp_valid = 2’b10.
24	Rotate A left by B bits	-
25	Rotate A right by B bits	Error not being asserted if OPA[7:4] > 0.
26	Invalid Inputs	Error not being asserted when inp_valid = 2’b00.
27	16 Clock Cycle Timeout	Error not being asserted if inp_valid = 2’b11 is not received even after waiting 16 clock cycles.

Coverage

Coverage is a metric used in verification to measure how thoroughly a design has been tested. It provides quantitative data on:

- Which parts of the design have been exercised.
- Which test scenarios have been executed.
- Whether all functional requirements have been verified.

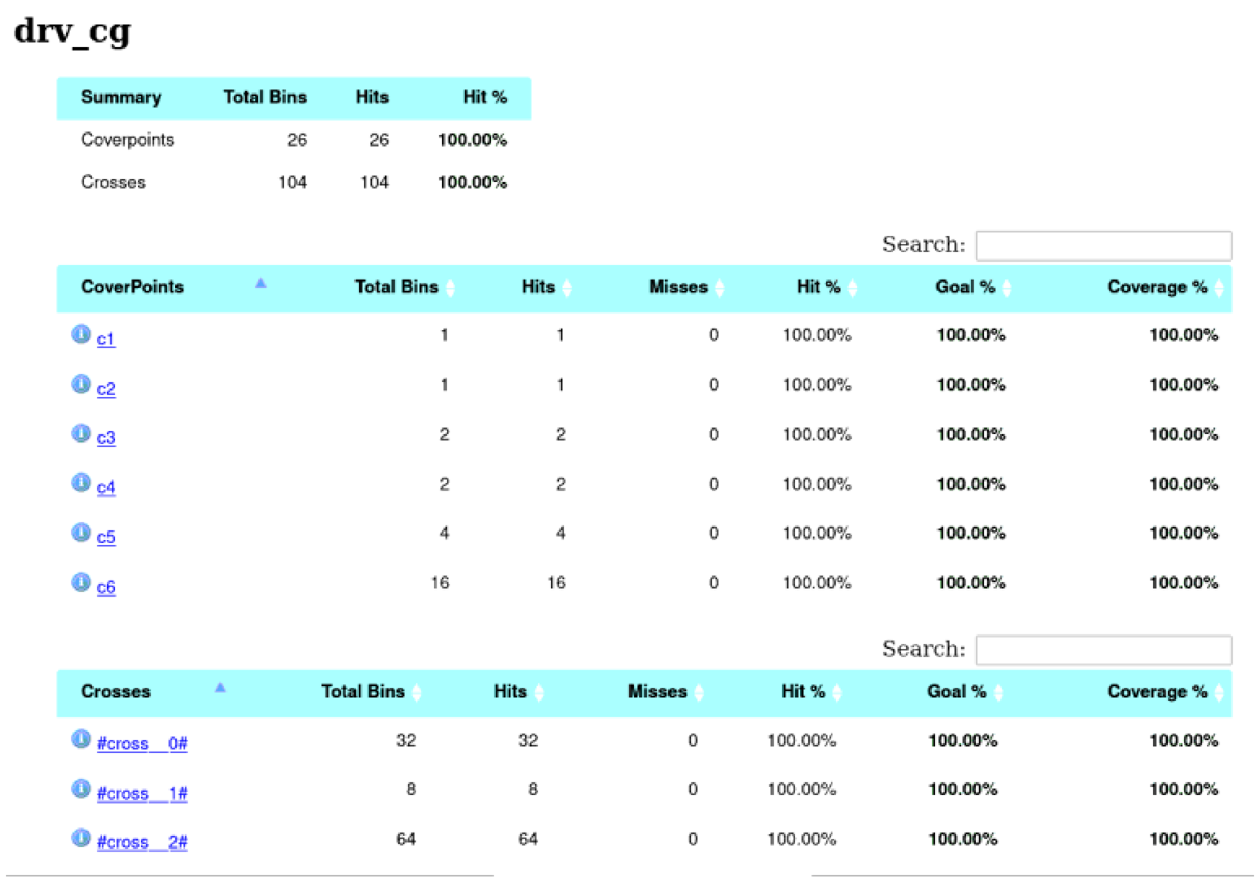
In digital design verification, coverage ensures that:

- All features of the ALU are tested.
- Corner cases (edge conditions) are exercised.

We are using two major covergroups inside our coverage component to measure our coverage :- cg(covergroup for inputs) and mon_cg(covergroup for outputs).

Input Coverage (drv_cg)

The covergroup cg is used to measure the range of inputs we have provided to the DUT.



drv_cg

Summary	Total Bins	Hits	Hit %
Coverpoints	26	26	**100.00%**
Crosses	104	104	**100.00%**

Search:

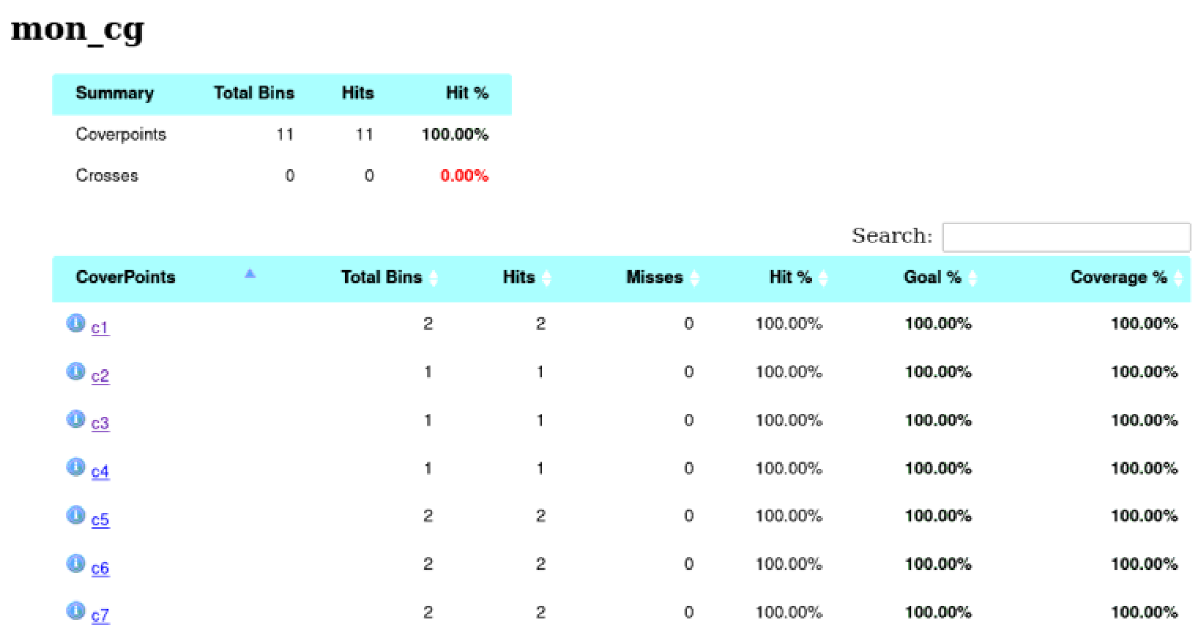
CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
c1	1	1	0	100.00%	**100.00%**	**100.00%**
c2	1	1	0	100.00%	**100.00%**	**100.00%**
c3	2	2	0	100.00%	**100.00%**	**100.00%**
c4	2	2	0	100.00%	**100.00%**	**100.00%**
c5	4	4	0	100.00%	**100.00%**	**100.00%**
c6	16	16	0	100.00%	**100.00%**	**100.00%**

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
#cross__0#	32	32	0	100.00%	**100.00%**	**100.00%**
#cross__1#	8	8	0	100.00%	**100.00%**	**100.00%**
#cross__2#	64	64	0	100.00%	**100.00%**	**100.00%**

Output Coverage (mon_cg)

This covergroup is used to measure the range of inputs we have received from the DUT.



mon_cg

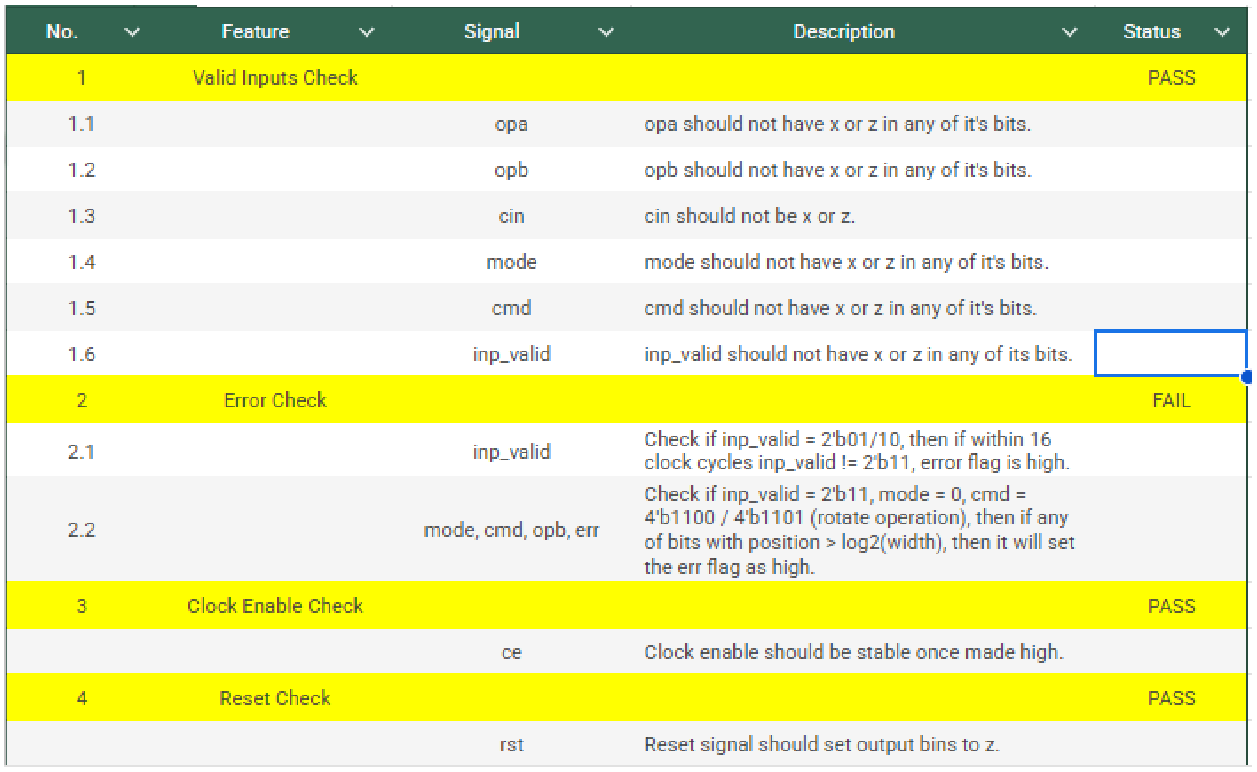
Summary	Total Bins	Hits	Hit %
Coverpoints	11	11	100.00%
Crosses	0	0	0.00%

Search:

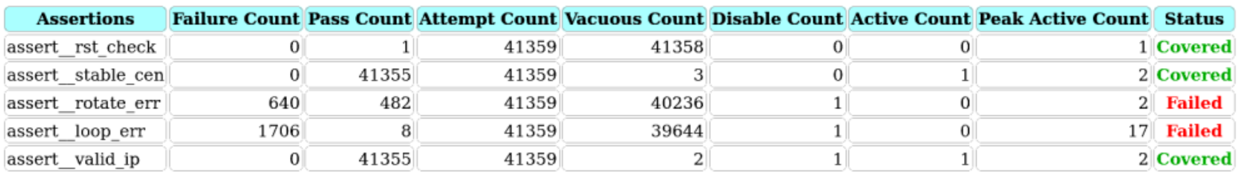
CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
c1	2	2	0	100.00%	100.00%	100.00%
c2	1	1	0	100.00%	100.00%	100.00%
c3	1	1	0	100.00%	100.00%	100.00%
c4	1	1	0	100.00%	100.00%	100.00%
c5	2	2	0	100.00%	100.00%	100.00%
c6	2	2	0	100.00%	100.00%	100.00%
c7	2	2	0	100.00%	100.00%	100.00%

Assertion Coverage

They are used to check whether the DUT is functioning as specified.

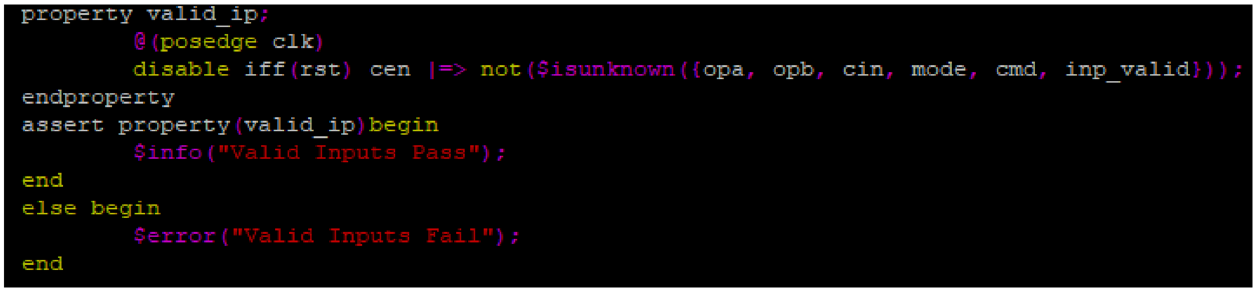


No.	Feature	Signal	Description	Status
1	Valid Inputs Check			PASS
1.1		opa	opa should not have x or z in any of it's bits.	
1.2		opb	opb should not have x or z in any of it's bits.	
1.3		cin	cin should not be x or z.	
1.4		mode	mode should not have x or z in any of it's bits.	
1.5		cmd	cmd should not have x or z in any of it's bits.	
1.6		inp_valid	inp_valid should not have x or z in any of its bits.	
2	Error Check			FAIL
2.1		inp_valid	Check if inp_valid = 2'b01/10, then if within 16 clock cycles inp_valid != 2'b11, error flag is high.	
2.2		mode, cmd, opb, err	Check if inp_valid = 2'b11, mode = 0, cmd = 4'b1100 / 4'b1101 (rotate operation), then if any of bits with position > log2(width), then it will set the err flag as high.	
3	Clock Enable Check			PASS
		ce	Clock enable should be stable once made high.	
4	Reset Check			PASS
		rst	Reset signal should set output bins to z.	



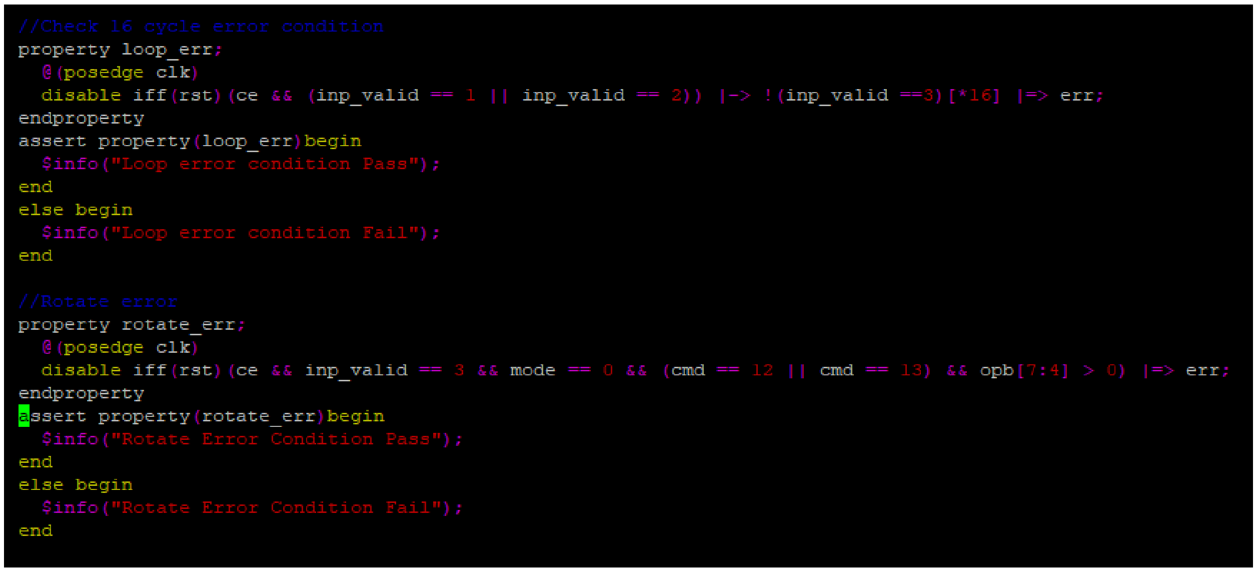
Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
assert__rst_check	0	1	41359	41358	0	0	1	Covered
assert__stable_cen	0	41355	41359	3	0	1	2	Covered
assert__rotate_err	640	482	41359	40236	1	0	2	Failed
assert__loop_err	1706	8	41359	39644	1	0	17	Failed
assert__valid_ip	0	41355	41359	2	1	1	2	Covered

1) Valid Inputs Check



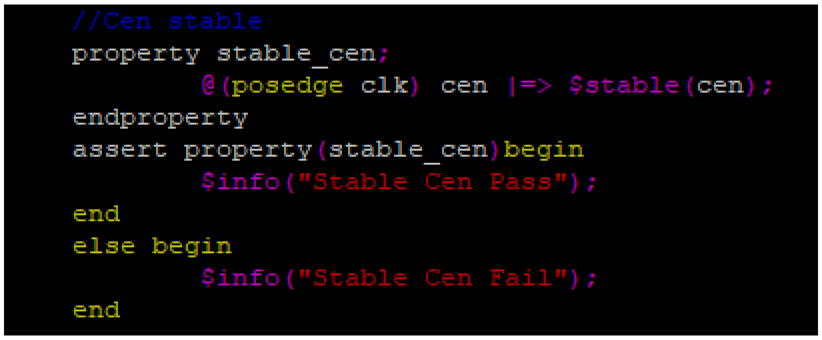
```
property valid_ip;
        @(posedge clk)
        disable iff(rst) cen |=> not($isunknown({opa, opb, cin, mode, cmd, inp_valid}));
endproperty
assert property(valid_ip)begin
        $info("Valid Inputs Pass");
end
else begin
        $error("Valid Inputs Fail");
end
```

2) Error Check



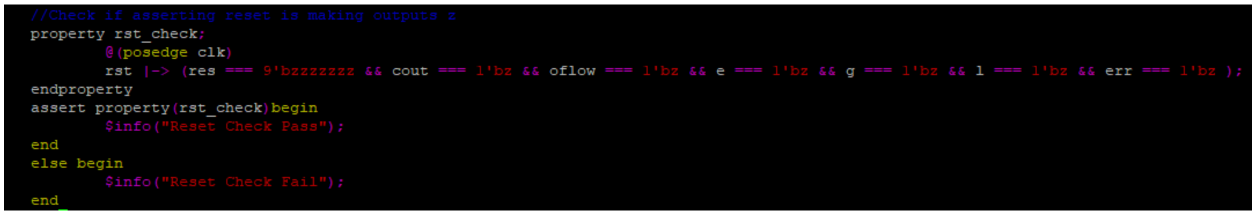
```
//Check 16 cycle error condition
property loop_err;
  @(posedge clk)
  disable iff(rst)(ce && (inp_valid == 1 || inp_valid == 2)) |-> !(inp_valid ==3)[*16] |=> err;
endproperty
assert property(loop_err)begin
  $info("Loop error condition Pass");
end
else begin
  $info("Loop error condition Fail");
end

//Rotate error
property rotate_err;
  @(posedge clk)
  disable iff(rst)(ce && inp_valid == 3 && mode == 0 && (cmd == 12 || cmd == 13) && opb[7:4] > 0) |=> err;
endproperty
assert property(rotate_err)begin
  $info("Rotate Error Condition Pass");
end
else begin
  $info("Rotate Error Condition Fail");
end
```

3) Clock Enable Check



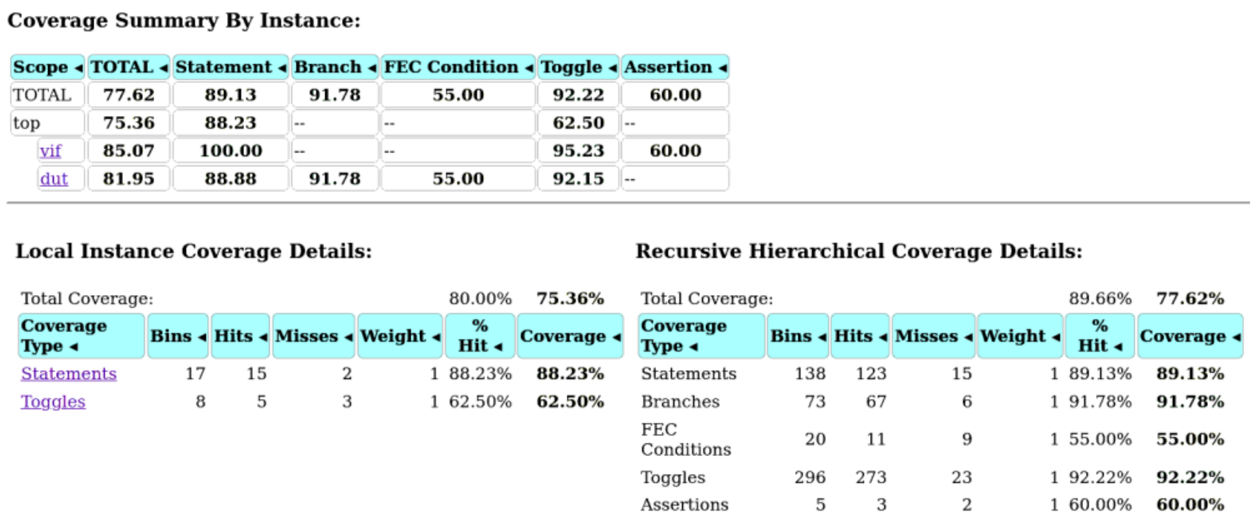
```
//Cen stable
property stable_cen;
        @(posedge clk) cen |=> $stable(cen);
endproperty
assert property(stable_cen)begin
        $info("Stable Cen Pass");
end
else begin
        $info("Stable Cen Fail");
end
```

4) Reset Check



```
//Check if asserting reset is making outputs z
property rst_check;
        @(posedge clk)
        rst |-> (res === 9'bzzzzzzz && cout === 1'bz && oflow === 1'bz && e === 1'bz && g === 1'bz && l === 1'bz && err === 1'bz );
endproperty
assert property(rst_check)begin
        $info("Reset Check Pass");
end
else begin
        $info("Reset Check Fail");
end
```

Overall Coverage



Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	Assertion
TOTAL	77.62	89.13	91.78	55.00	92.22	60.00
top	75.36	88.23	--	--	62.50	--
vif	85.07	100.00	--	--	95.23	60.00
dut	81.95	88.88	91.78	55.00	92.15	--

Local Instance Coverage Details:

Total Coverage: 80.00% **75.36%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	17	15	2	1	88.23%	**88.23%**
Toggles	8	5	3	1	62.50%	**62.50%**

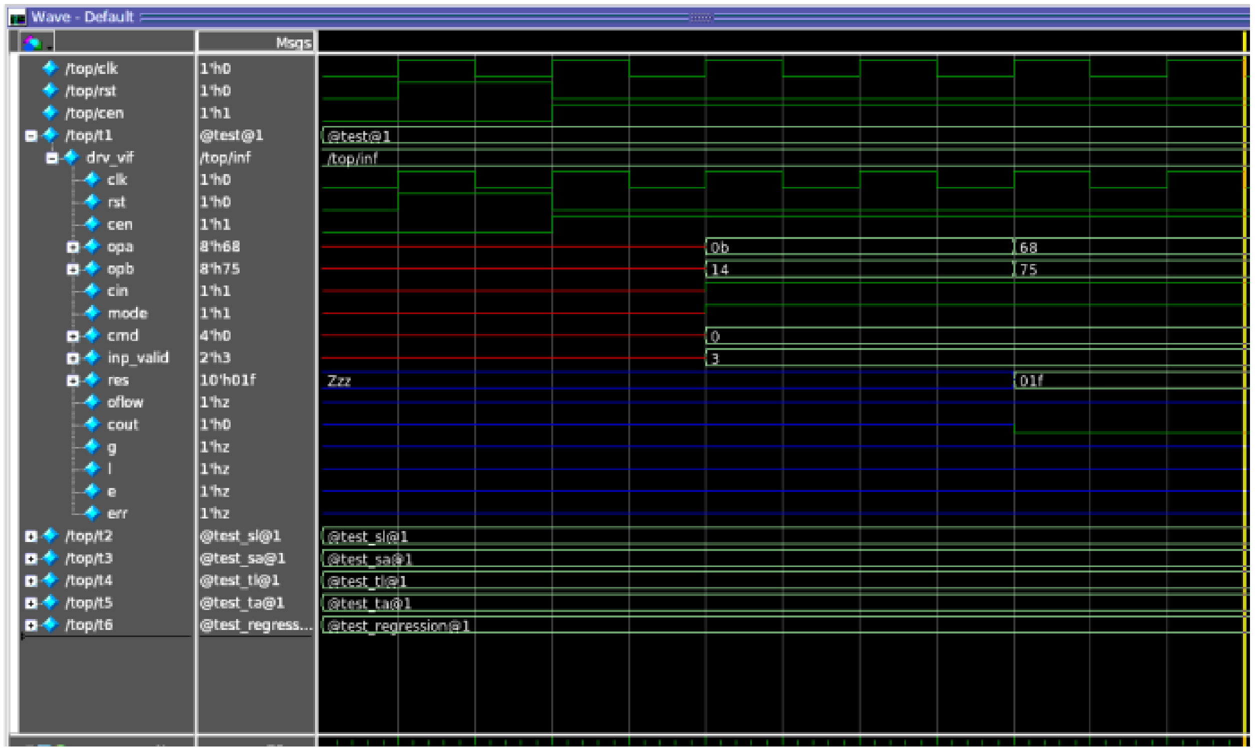
Recursive Hierarchical Coverage Details:

Total Coverage: 89.66% **77.62%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	138	123	15	1	89.13%	**89.13%**
Branches	73	67	6	1	91.78%	**91.78%**
FEC Conditions	20	11	9	1	55.00%	**55.00%**
Toggles	296	273	23	1	92.22%	**92.22%**
Assertions	5	3	2	1	60.00%	**60.00%**

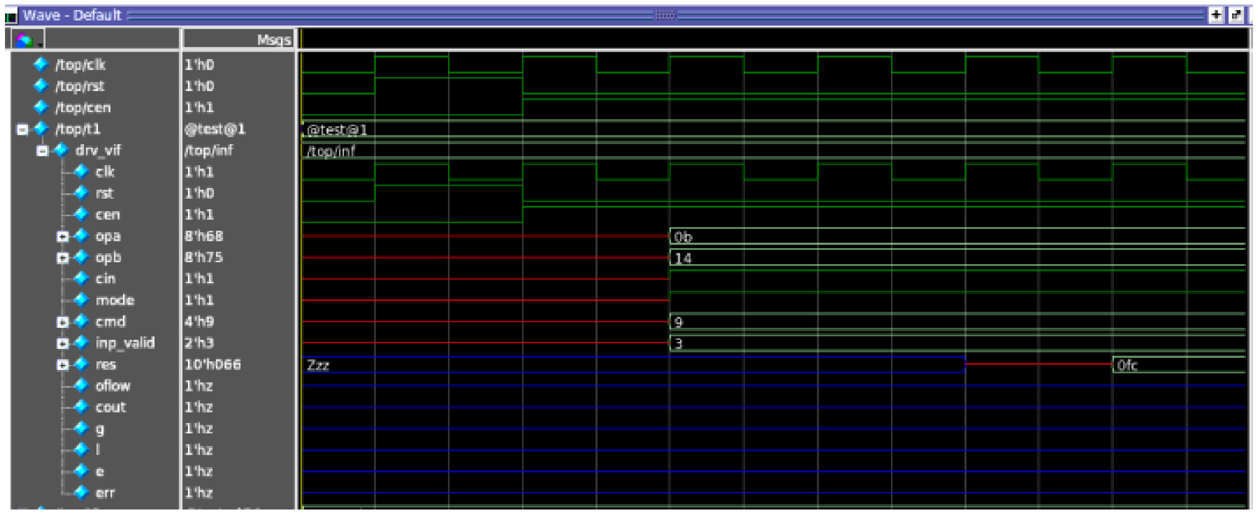
Output Waveform

2 Cycle Operation (Unsigned Addition here) Waveform



```
/top/clk        1'h0
/top/rst        1'h0
/top/cen        1'h1
/top/t1         @test@1
  drv_vif       /top/inf
    clk         1'h0
    rst         1'h0
    cen         1'h1
    opa         8'h68
    opb         8'h75
    cin         1'h1
    mode        1'h1
    cmd         4'h0
    inp_valid   2'h3
    res         10'h01f
    oflow       1'hz
    cout        1'h0
    g           1'hz
    l           1'hz
    e           1'hz
    err         1'hz
/top/t2         @test_sl@1
/top/t3         @test_sa@1
/top/t4         @test_tl@1
/top/t5         @test_ta@1
/top/t6         @test_regress...
```

3 Cycle Operation (Increment and Multiply) Waveform



```
/top/clk        1'h0
/top/rst        1'h0
/top/cen        1'h1
/top/t1         @test@1
  drv_vif       /top/inf
    clk         1'h1
    rst         1'h0
    cen         1'h1
    opa         8'h68
    opb         8'h75
    cin         1'h1
    mode        1'h1
    cmd         4'h9
    inp_valid   2'h3
    res         10'h066
    oflow       1'hz
    cout        1'hz
    g           1'hz
    l           1'hz
    e           1'hz
    err         1'hz
```