ALU Verification Document

# Index

[**Index**](#_y27d6258yc4x) **1**

[**Introduction**](#_emdai76m2zwk) **2**

[**Key Objectives**](#_szxwugm1h1id) **2**

[**DUT INTERFACES**](#_jospna4cq6tc) **2**

[**Testbench Architecture**](#_izuyn5j2ol1h) **5**

[Sequence\_Items (seq\_item)](#_y6tiuxsnjg91) 5

[Sequence](#_bpuhslrh54ml) 6

[Interface](#_qmw6mn98csmi) 7

[Sequencer](#_ojht7de6m96) 8

[Driver](#_f1ix6rhcbdzl) 9

[Monitor](#_rpnvsdvebm7b) 10

[Agent](#_2voavoqxoq6i) 11

[Scoreboard](#_8zchdvotbhle) 11

[Environment (env)](#_vc02aekfjotc) 12

[Test](#_7q4h7zwksf44) 13

[**Errors in DUT Functionality**](#_i2rjss19zniy) **14**

[**Coverage**](#_beklmvgvqsox) **15**

[Input Coverage (drv\_cg)](#_vg6j2ti131f2) 15

[Output Coverage (mon\_cg)](#_gml2s8wpkqkp) 16

[Assertion Coverage](#_z6rraaqi30ig) 17

[Overall Coverage](#_osa24tjavitm) 19

[**Output Waveform**](#_43mpb2olmtc9) **20**

[2 Cycle Operation (Unsigned Addition here) Waveform](#_wllmwxbacksj) 20

[3 Cycle Operation (Increment and Multiply) Waveform](#_qagh2y4tksu2) 20

# 

# 

# 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 :-

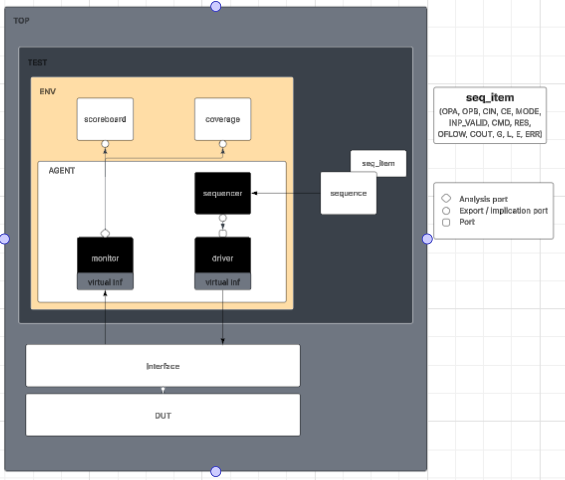
* Understanding the ALU design specification, including supported operations (e.g., addition, subtraction, AND, OR, XOR, shifts) and control signals.
* Developing a structured testbench using a hardware verification language (e.g., SystemVerilog) with assertions and coverage metrics.
* Executing functional 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

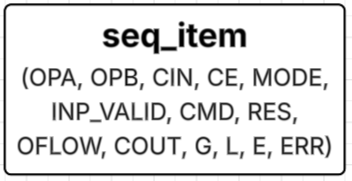
The DUT consists of the following input and output pins.

| 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)



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.

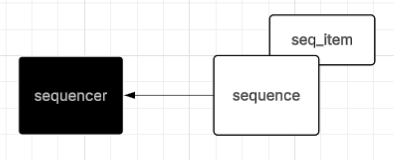
1. Inputs

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

2. Outputs

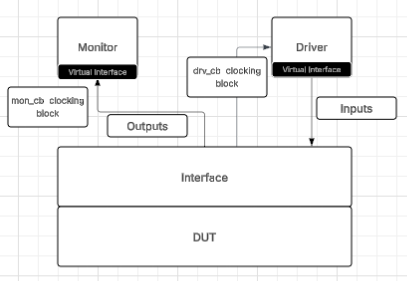
* + RES (Result)
  + OFLOW (Overflow)
  + COUT (Carry-out)
  + G, L, E (Greater, Less, Equal flags)
  + ERR (Error flag

## Sequence



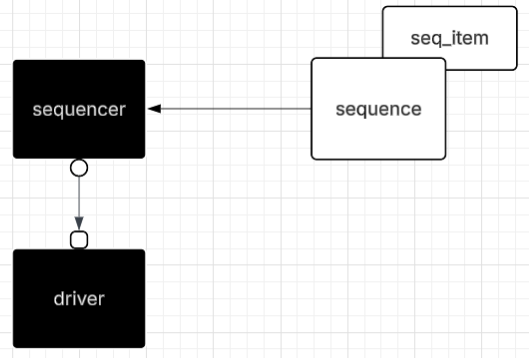
* Generates the stimulus containing randomized ALU inputs (OPA, OPB, CMD, CIN, MODE, CMD) by randomizing the seq\_item’s.
* Controls test variability by applying constraints to randomization (e.g., valid opcodes, corner-case operands).
* These seq\_items are sent to the sequencer req fifo where they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

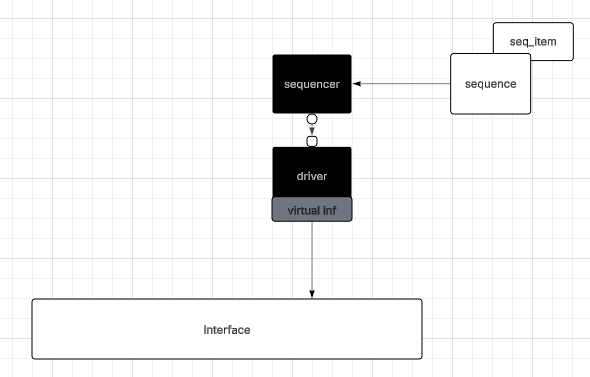
## Sequencer



* 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)*.

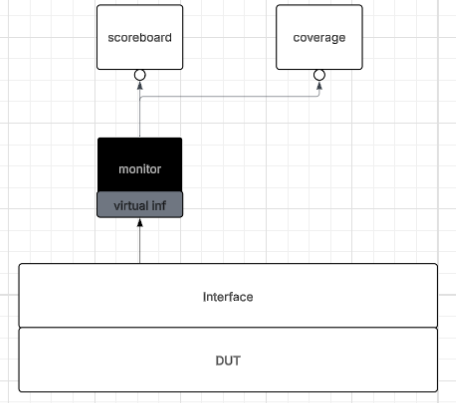
(seq\_item\_port in an in-built method of uvm\_driver, which is used to request items from sequencer. seq\_item\_export is an inbuilt method of uvm\_sequencer which is used to send seq\_items to the driver.)

## 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

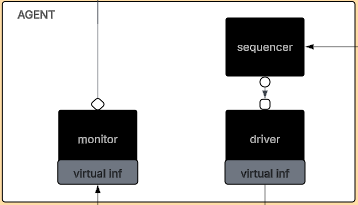


* Monitor as the name states is used to monitor the outputs from the DUT via the interface and store it in a seq\_item packet (mon\_item here).
* The Monitor then sends these sequence\_items (seq\_item) to scoreboard and coverage via TLM ports. Syntax :-   
  *uvm\_analysis\_port #(seq\_item) <monitor\_port\_handle>*  
  We call a write function to send it to the scoreboard queue.   
  *<monitor\_port\_handle>.write(mon\_item).*

Connected in the agent class.

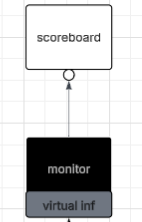
* Also has a TLM connection to the coverage component which is used to track output coverage. The same port which is sending to the scoreboard is sending to the coverage component as well. It is connected in the connect phase of environment class.  
  *<agent\_handle>.<monitor\_handle>.<monitor\_analysis\_port\_handle>.connect(<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>)

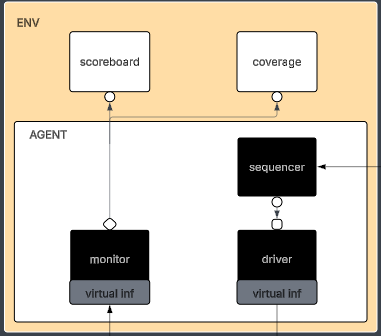
## Scoreboard



* 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[$];

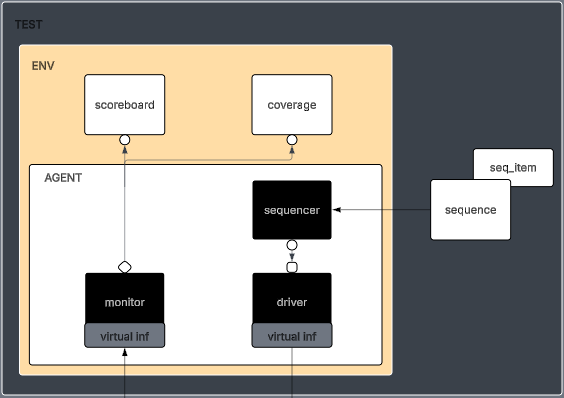
uvm\_port\_imp #(seq\_item) <scoreboard\_port\_handle>  
We have to implement write function in scoreboard  
function void write(seq\_item pkt);  
 q.push\_back(pkt);  
Endfunction  
In run\_phase we check whether there is any packet received from monitor (q.size > 0) then calculate using internal reference 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.

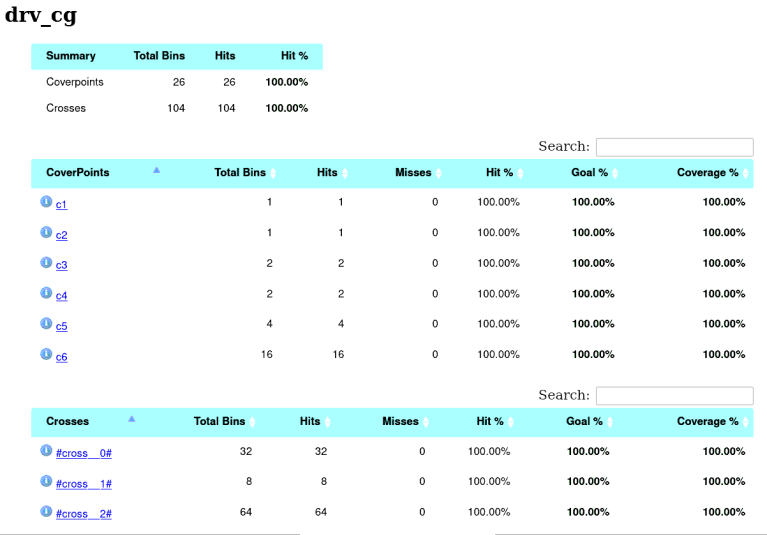
In digital design verification, coverage ensures that:

* 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).

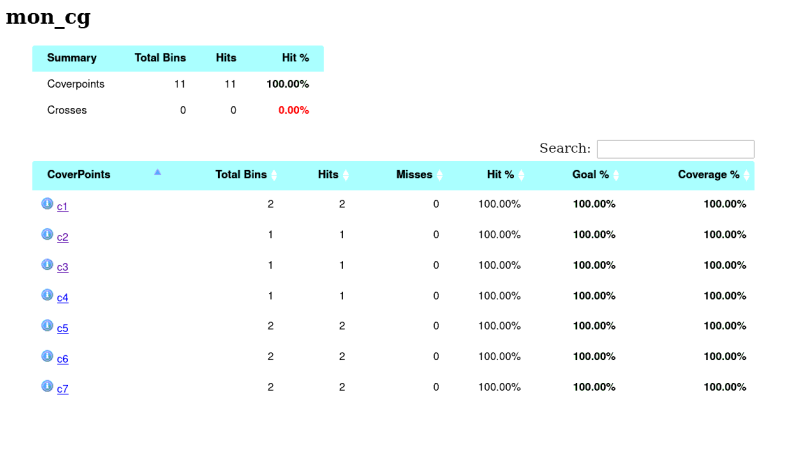
## Input Coverage (drv\_cg)

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

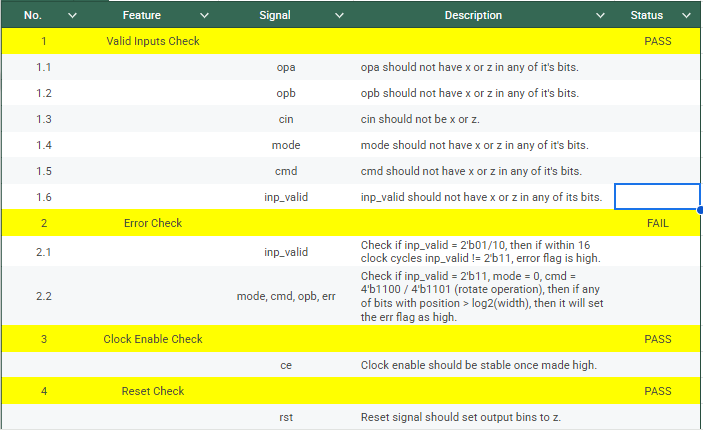


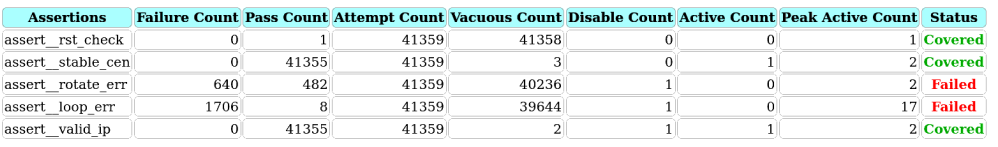
## Output Coverage (mon\_cg)

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

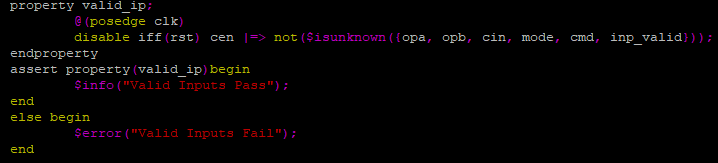


## Assertion Coverage

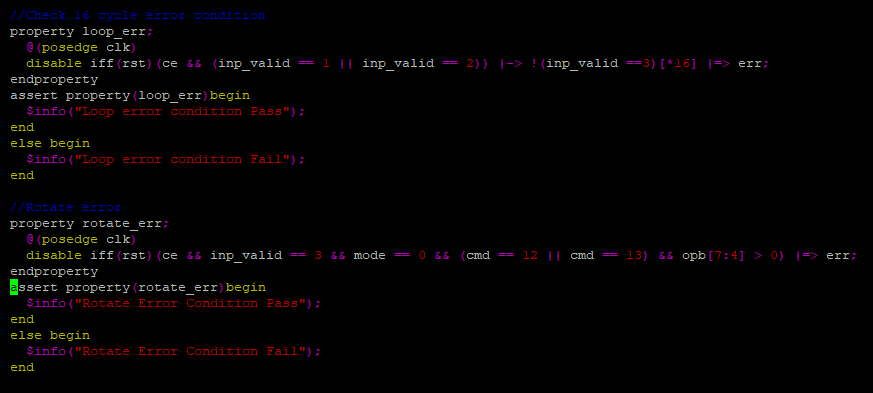
They are used to check whether the DUT is functioning as specified.



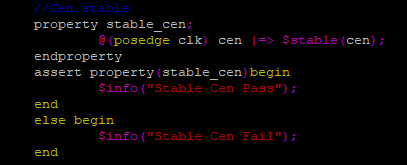
1. Valid Inputs Check



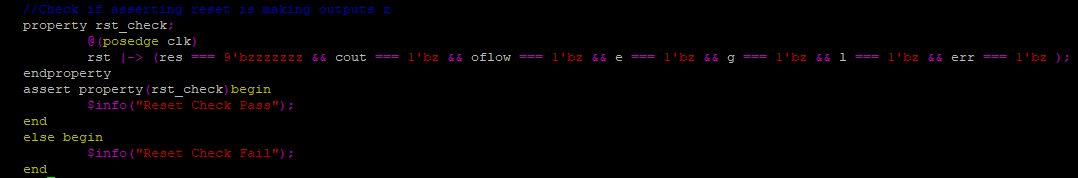
1. Error Check



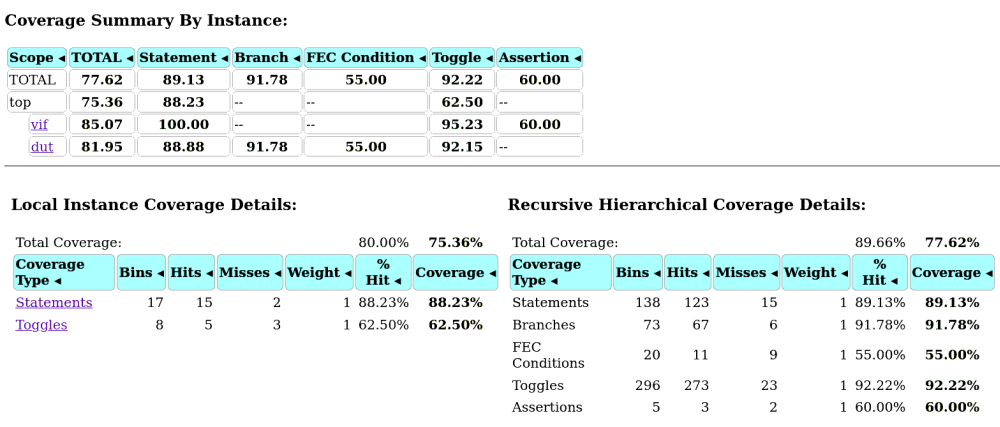
1. Clock Enable Check



1. Reset Check



## Overall Coverage



# Output Waveform

## 2 Cycle Operation (Unsigned Addition here) Waveform

## 3 Cycle Operation (Increment and Multiply) Waveform

