ALU Verification Plan

CHAPTER 1 – DESIGN OVERVIEW

1.0 **ALU:-**

The Arithmetic Logic Unit (ALU) is a fundamental combinational circuit that performs essential arithmetic and logical operations within a digital system. In this design, the ALU is implemented using Verilog, a hardware description language widely used for modelling digital systems.

Why is ALU important:-

- Every calculation your computer does goes through the ALU
- Without ALU, computers cannot do math or make decisions
- It works very fast - millions of operations per second

1.1 **Advantages of ALU:-**

- Does math problems like add, subtract, multiply, divide.
- Used to make logical decisions like AND, OR, NOT.
- Used to compares numbers which is bigger, smaller, or equal.
- It is fast accurate and occupies less space inside processor.

1.2 **Disadvantages of ALU:-**

1.**No Memory**

- Cannot remember previous calculations.

2.**speed limit**

Cannot work faster than the processor allows.

3.Supports only specific bit widths such as 32-bit and 64-bit; larger data sizes are truncated during processing..

4.Power consumption increases rapidly during high-complex calculations

1.3 **Use cases of ALU:-**

The ALU is a core component in digital systems and processors. Key use cases include:

1.Microprocessors and Microcontrollers – Executes arithmetic and logic instructions (e.g., add, subtract, AND, OR).

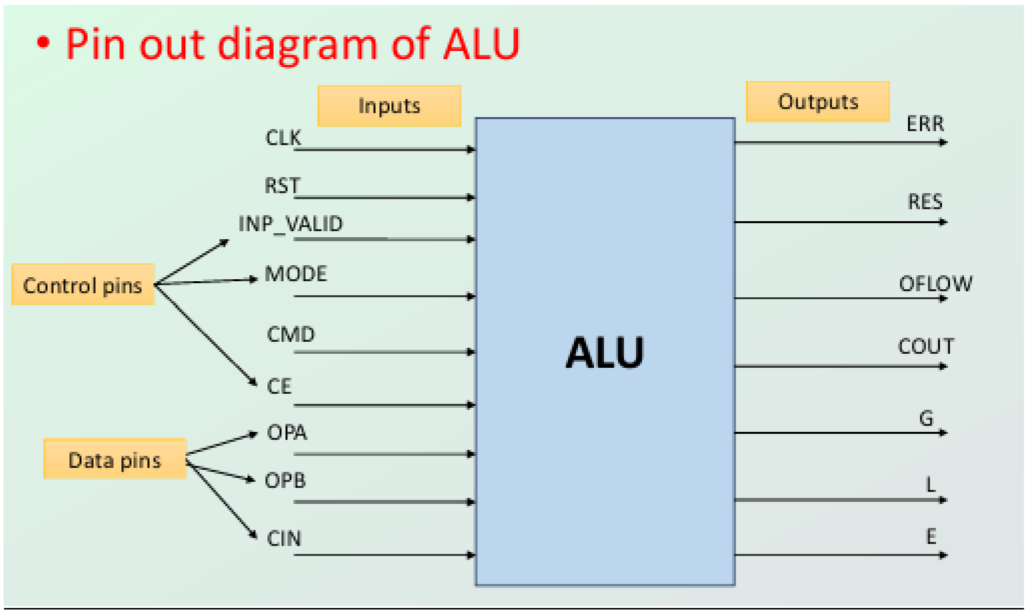
2.DSPs (Digital Signal Processors) – Used for filtering, FFT, and signal transformations.

3.Embedded Systems – Performs calculations for automation and control logic.

4.Cryptographic Units – Handles bitwise and modular arithmetic for encryption algorithms.

5.AI Accelerators / SIMD Units – ALU arrays process matrix/vector operations efficiently.

1.4 **Project Overview of ALU:-**



1.This project implements a parameterized ALU in Verilog that supports a variety of operations including arithmetic, logical, comparison, and bitwise shift/rotation. The ALU design is tested using a SystemVerilog testbench with functional coverage and assertions.

2.This pin-out diagram provides a structural overview of how the ALU interfaces with external components. The design features clearly categorized inputs and outputs to enable smooth data flow and operation control. Control pins manage the operation mode and synchronization,while data pins provide the operands required for computation. The outputs reflect the results of the operations along with relevant status indicators such as comparison and error flags.

3.By structuring the ALU in this way, the design ensures modularity, reusability, and ease of integration into larger digital systems such as CPUs and signal processors.

1.5 **Design Features:-**

• Arithmetic
  Addition ,subtraction ,Comparision ,Increment,Decrement,Multiplication
• Logic
  AND,NAND,OR,NOR, XOR,XNOR, NOT
• Others
  Rotate right/left , Shift right/left

1.6 Verification Objectivies

• Functional Verification:
1. Arithmetic Operations.
2. Logical Operations.
3. Control and Interface: - MODE switching between arithmetic and logical modes. - inp_valid combinations (00, 01, 10, 11) impact verification. - Clock enable and reset behavior testing. - 16-cycle timeout mechanism for missing operands.

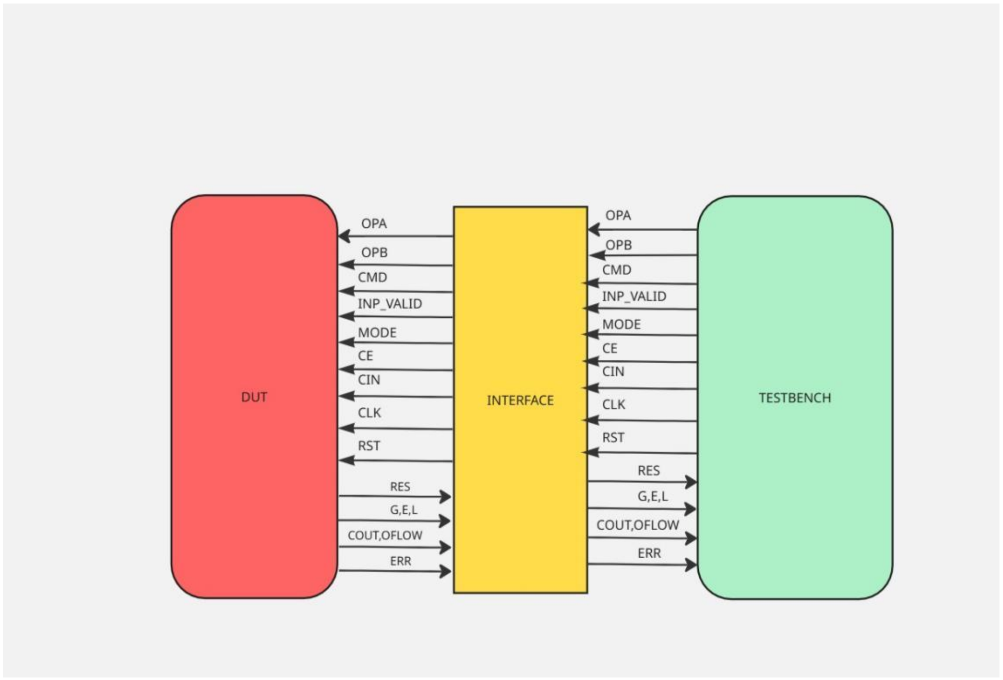
4. Error Handling:

• Functional Coverage: - 100% command coverage across both modes - All inp_valid state transitions - Comprehensive error condition covera

• Verification Stratergy: -

- Constrained random stimulus generation
- Self-checking testbenches with reference models
- Directed tests for corner cases
- Assertion-based protocol verification
- Coverage-driven methodology

1.7 **Design diagram with interface signals:**-



* **INPUT PORTS:-**

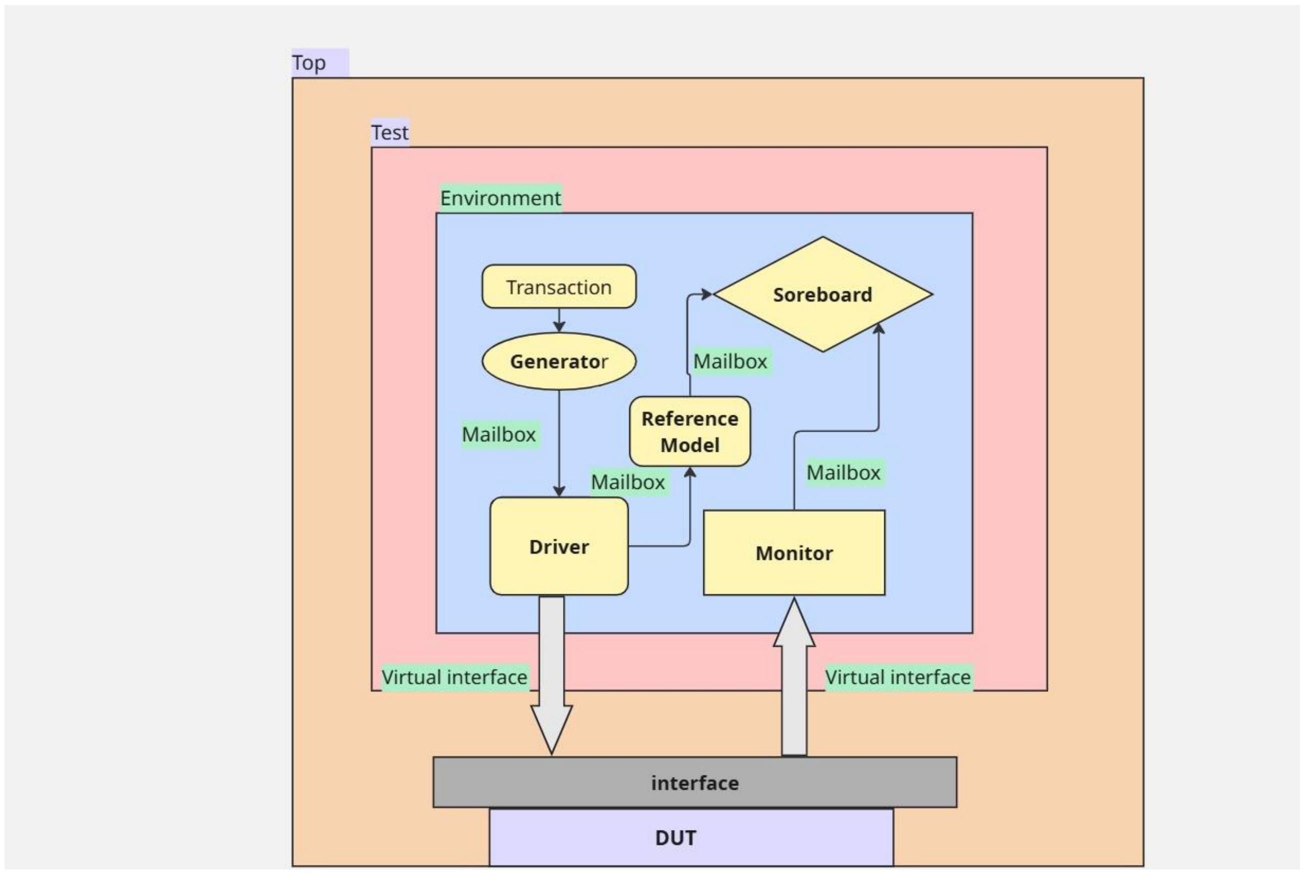
Signal Name	**Type**	**Description**
Inp_valid	input	Input valid signal - indicates when input data is valid
CMD	input	Command signal - specifies the specific ALU operation
OPA	input	Operand A - first arithmetic/logic operand
OPB	input	Operand B - second arithmetic/logic operand
MODE	input	Mode selection signal - determines ALU operation mode
CE	input	Clock enable 1 indicates high
CIN	input	Carry In - input carry for arithmetic operations
CLK	input	Clock signal
RST	input	Rst signal

Output port:-

Signal	**Type**	**Description**
RES	Output	Result - the output result of the ALU operation
ERR	Output	Error signal - indicates if an error occurred during operation
G	Output	Greater than - comparison result flag
E	Output	Equal - comparison result flag
L	Output	Less than - comparison result flag
COUT	Output	Carry Out - output carry from arithmetic operations
OFLOW	Output	Overflow - indicates arithmetic overflow condition

CHAPTER 2 - Verification Architecture

2.1 **Verification Architecture :-**



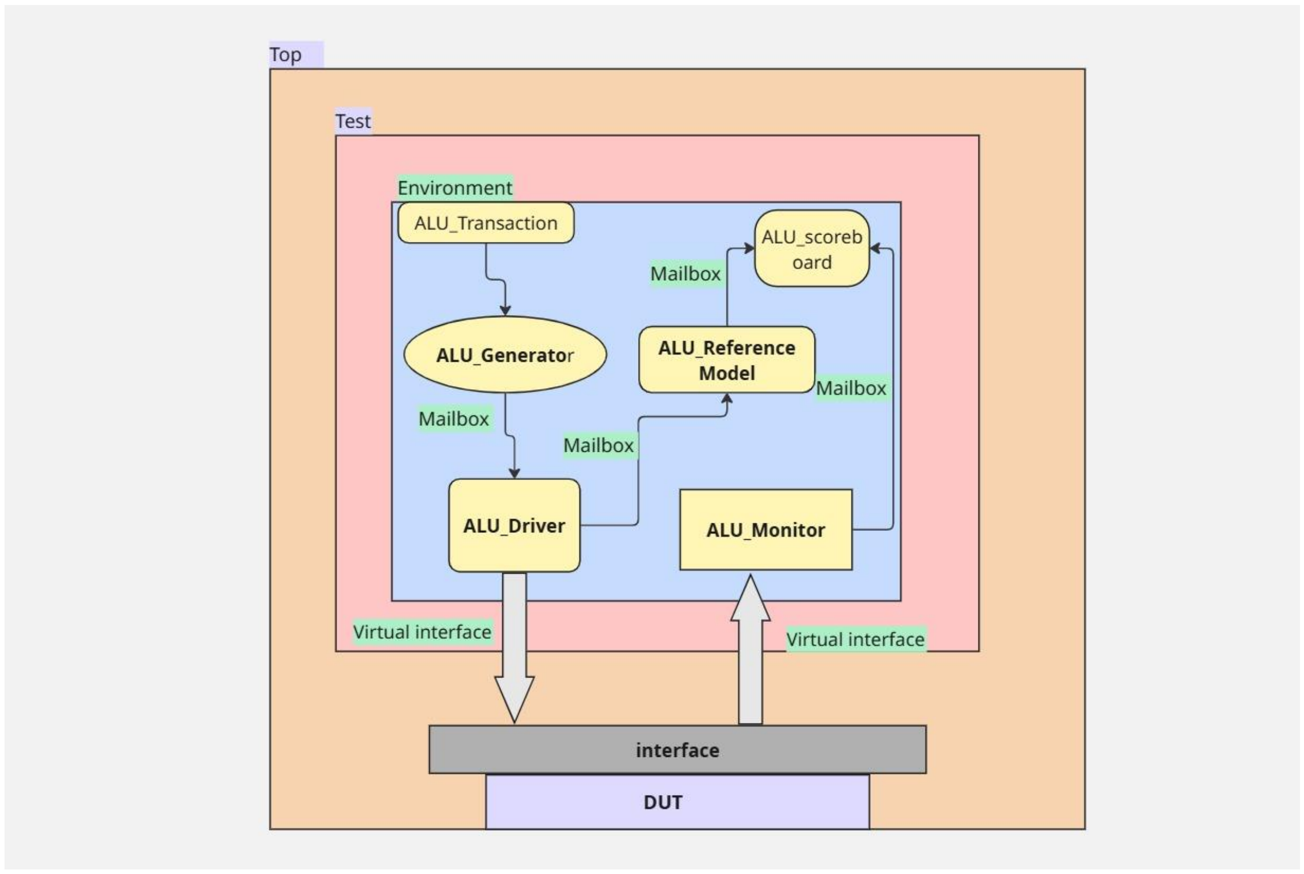
This diagram shows a typical testbench setup used to verify digital designs.

- At the center is the Design Under Verification (DUV).
- The testbench environment includes several parts:
- A generator creates input test cases (transactions), often using random values.
- These inputs go to the driver, which converts them into signals and sends

them to the DUV through a virtual interface.

- The DUV's output is captured by a monitor, which passes the data to the scoreboard.
- The scoreboard checks if the output is correct by comparing it to the expected results from a reference model.
- All these components are organized within the environment block, which manages their interaction.
- This setup helps verify that the DUV works correctly by comparing actual and expected behavior.

2.2 Verification Architecture of ALU:-



The figure illustrates the Verification Architecture for an Arithmetic Logic Unit (ALU) using a modular testbench environment.

Key Components:

- **ALU_transaction:**

Defines the input and output of transactions (e.g., operands and operations) exchanged between components.

- **ALU_generator:**

Randomly creates test scenarios by generating transactions. These are sent to the driver for processing.

- **ALU_driver:**

Receives transactions from the generator and drives them to the DUV and reference model using a virtual interface (VI)

- **DUV (Design Under Verification):**

The actual ALU design that receives inputs from the driver and generates outputs for validation.

- **ALU_monitor:**

Observes and captures the output signals from the DUV through the virtual interface. It converts them back into transaction format and forwards them to the scoreboard.

- **ALU_reference_model:**

Serves as a golden model that receives the same input as the DUV and produces the expected output for comparison.

- **ALU_Scoreboard:**

Compares the output from the DUV (via the monitor) with the expected output from the reference model. Any mismatches are flagged as functional errors.

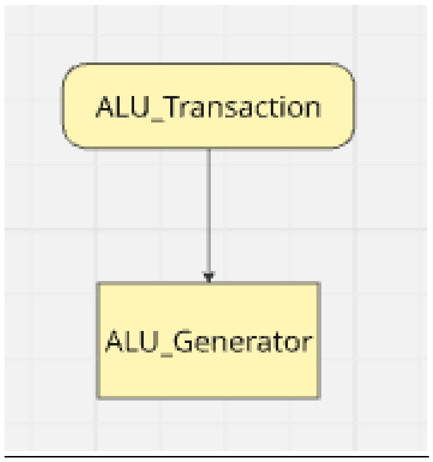
• Mailboxes:

Facilitate communication between generator, driver, monitor, reference model, and scoreboard by passing transactions.

2.3 FLOW CHART OF SV COMPONENTS :-

1. Transaction Class:

- The alu_transaction class encapsulates all ALU input stimuli and output responses for verification purposes.



Components

Randomized Input Stimuli: The transaction contains ALU input signals declared with the rand keyword for automatic randomization:

• inp_valid, MODE, CMD, OPA, OPB, CIN: Input signals that will be randomized by the generator

Non-Randomized Output Signals

Output signals are declared without rand as they represent the ALU's response:

• ERR, RES, OFLOW, COUT, G, L, E: Output signals monitored for verification

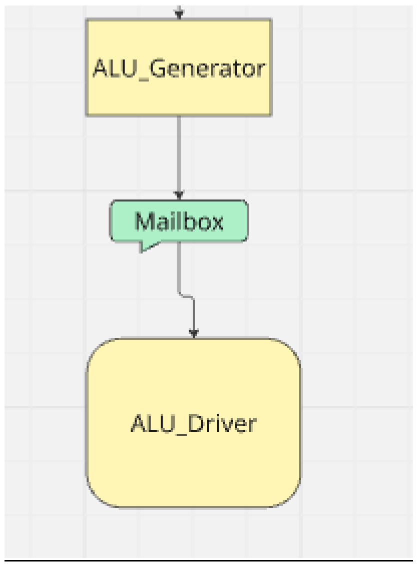
Constraints

The class implements mode-dependent constraints for realistic test scenarios:

Deep Copy Method

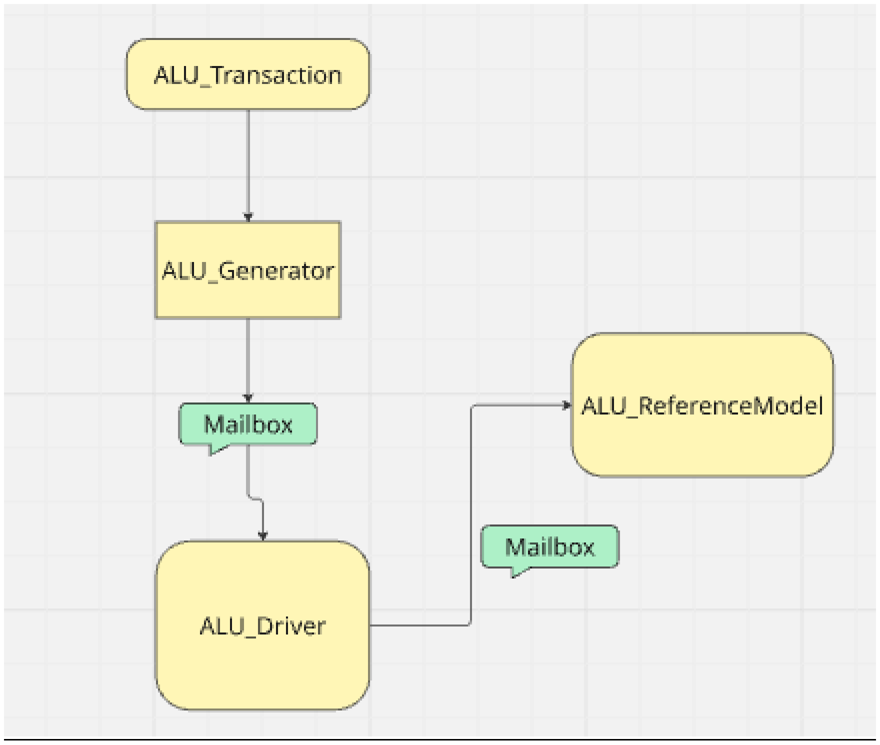
Implements a copy() function following the blueprint pattern for creating independent transaction copies used across testbench components.

2.Generator Class: -



The generator consists of the ALU transaction class handle, the mailbox handle which connects to the driver, and the start() task which randomizes transactions and sends the randomized transactions to the driver through the mailbox.

3.Driver class:-



Key Features Implemented:

1. Two Mailboxes

• mbox_d: Transfers transactions from generator to driver

• mbo_r: Sends transactions from driver to reference model

2. Virtual Interface (Dynamic)

• intf: SystemVerilog interface with clocking blocks

• alu_if: Virtual interface for dynamic access

• Communicates between testbench and DUV with proper synchronization

3. Functional Coverage Group

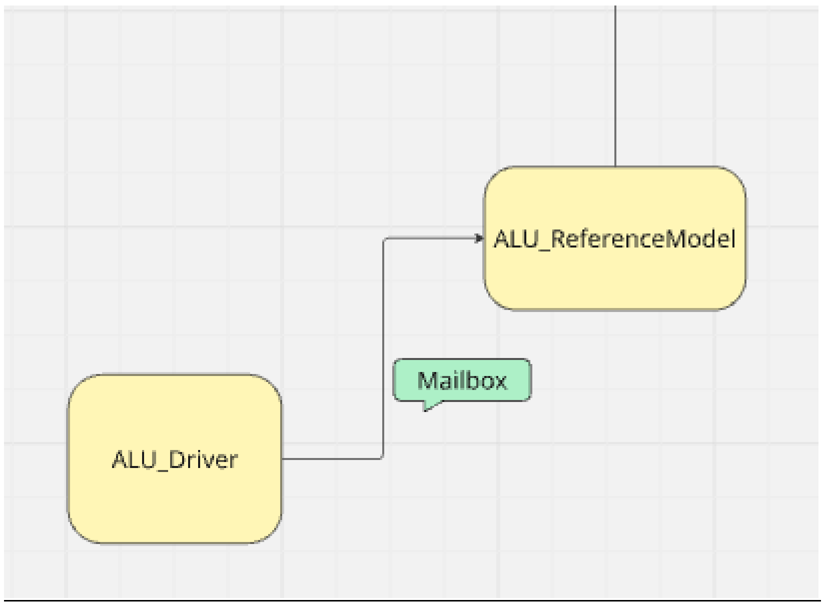
• Covers all input combinations (operands a, b, operation).

• Edge case coverage (zero results, overflow, carry conditions).

4. Drive Task in ALU Driver

• transaction(): Drives stimuli to the DUV.

4. Reference Model:-



Serves as a golden model that receives the same input as the DUV and produces the expected output for comparison.

Key Features:

1. Two Mailboxes

1. mbox_r: Receives transactions from the driver with input stimuli

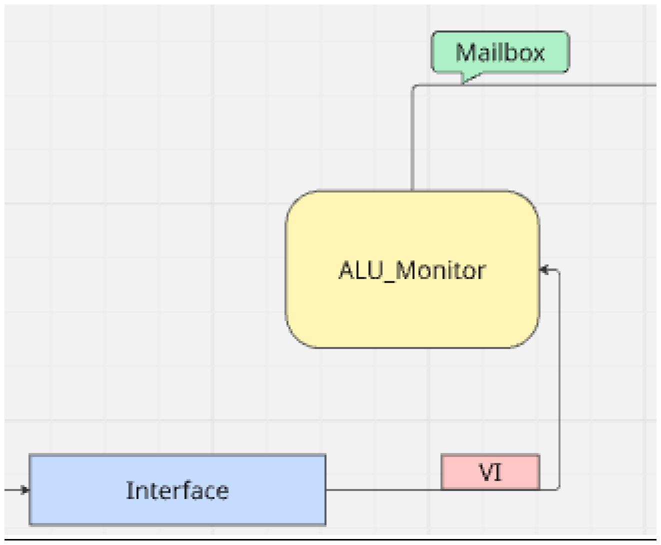
2. mbox_s: Sends processed transactions with expected results to the scoreboard

2. Functionality

• Task : run(), Continuously gets transactions from driver, computes expected results, and forwards to scoreboard

• Task to permorm operations: Implements golden reference for all ALU operations .

5.Monitor:-



The ALU Monitor captures transactions from the DUV interface and forwards them to the scoreboard for

result comparison.

Mailbox Communication

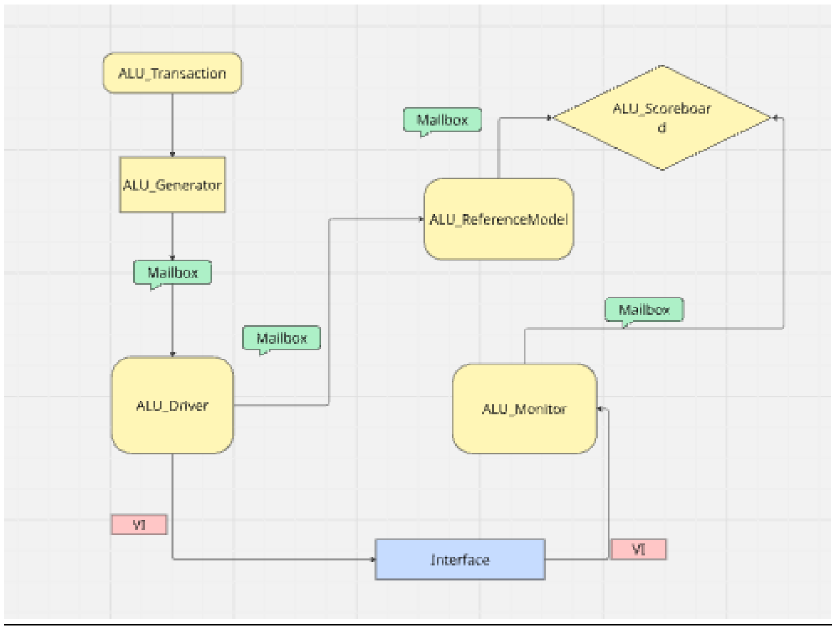
• mbox_m: Sends captured DUV transactions to the scoreboard

Functionality

• run(): Continuously monitors DUV interface and captures completed transactions

• Samples DUV outputs when valid transaction completes and creates transaction objects

6.scoreboard:-



The ALU Scoreboard compares actual DUV results against expected results from the reference model to determine test pass/fail status.

Mailbox Communication

• mbox_m: Receives expected results from the reference model

• mbox_r: Receives actual results from the monitor

Functionality

- Compares expected vs actual results and reports mismatches

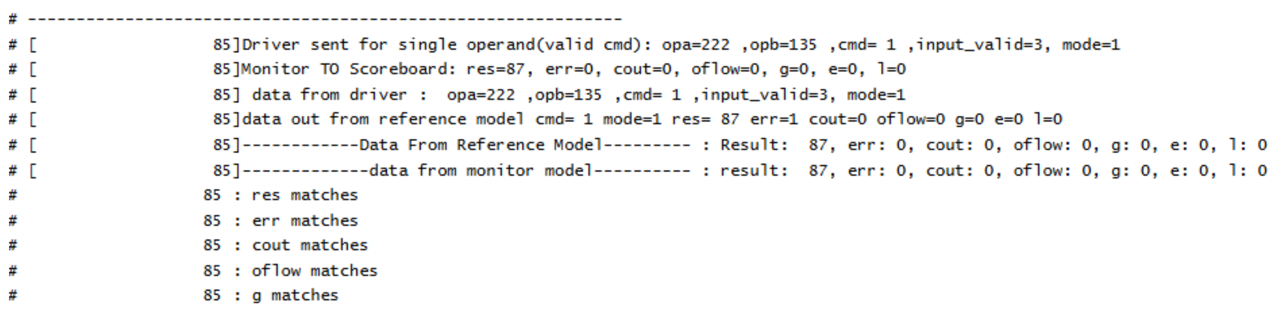
Chapter 3 Outputs & Coverage

3. Outputs

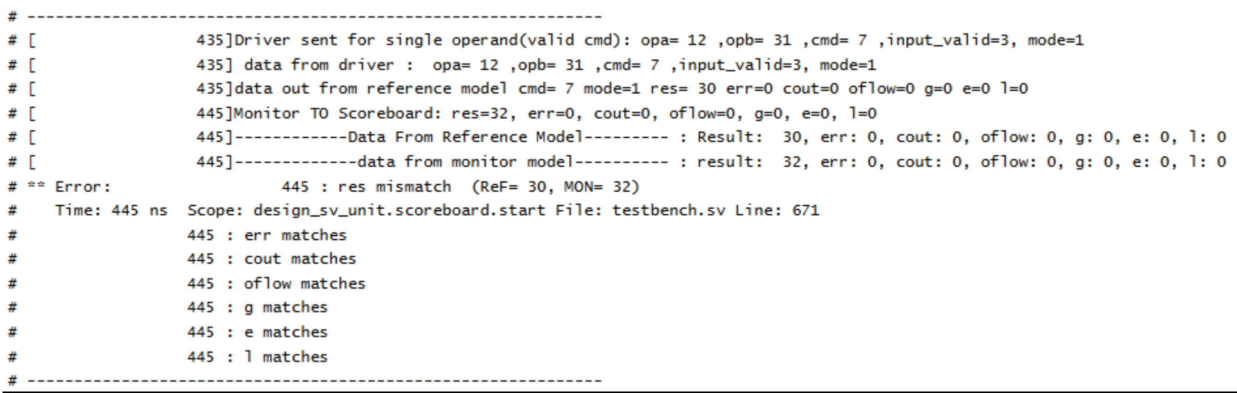
- Successfully captured the outputs from the Design Under Test (DUT) during simulation.
- Compared these outputs against the results generated by the reference model, ensuring functional correctness.
- Verified the DUT behaviour across multiple scenarios:
- Different command values and operating modes.
- Partial input validity cases (IN_VALID = 2'b10 / 2'b01) and full validity (inp_valid= 2'b11).
- Boundary and corner cases using randomized operands and control signals.
- Identified and analysed design bugs and unexpected DUT responses during simulation,helping refine the DUT logic.
- Achieved significant functional coverage through systematic stimulus generation and cross-coverage of operands, commands, and control signals.
- Ensured the DUT meets design specifications by correlating waveform observations,simulation logs, and coverage data.

Few examples: -

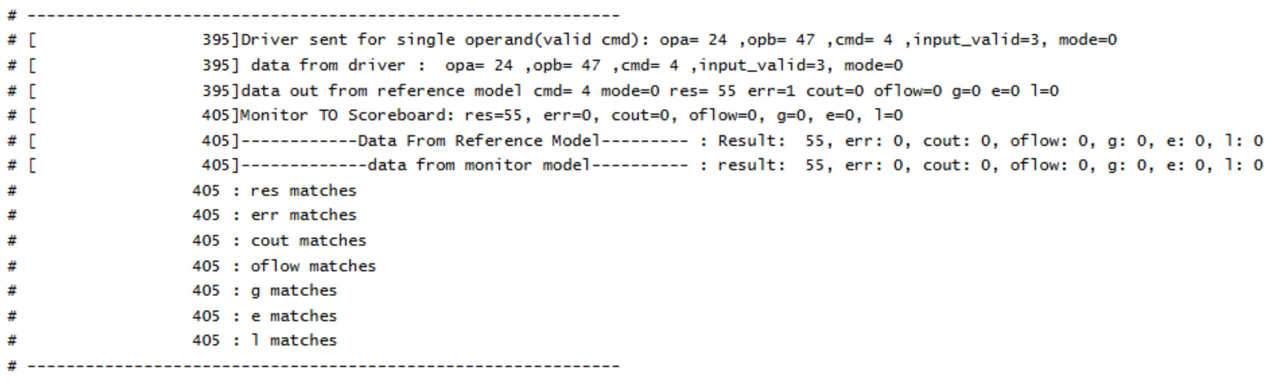
• For 2 clock cycle delay-



```
# -------------------------------------------------------------
# [                   85]Driver sent for single operand(valid cmd): opa=222 ,opb=135 ,cmd= 1 ,input_valid=3, mode=1
# [                   85]Monitor TO Scoreboard: res=87, err=0, cout=0, oflow=0, g=0, e=0, l=0
# [                   85] data from driver :  opa=222 ,opb=135 ,cmd= 1 ,input_valid=3, mode=1
# [                   85]data out from reference model cmd= 1 mode=1 res= 87 err=1 cout=0 oflow=0 g=0 e=0 l=0
# [                   85]------------Data From Reference Model--------- : Result:  87, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# [                   85]-------------data from monitor model---------- : result:  87, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
#                    85 : res matches
#                    85 : err matches
#                    85 : cout matches
#                    85 : oflow matches
#                    85 : g matches
```

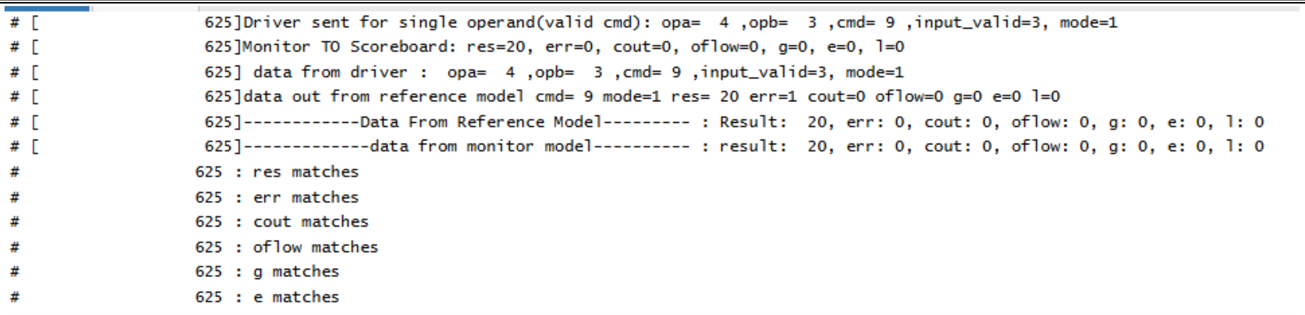


```
# -------------------------------------------------------------
# [                  435]Driver sent for single operand(valid cmd): opa= 12 ,opb= 31 ,cmd= 7 ,input_valid=3, mode=1
# [                  435] data from driver :  opa= 12 ,opb= 31 ,cmd= 7 ,input_valid=3, mode=1
# [                  435]data out from reference model cmd= 7 mode=1 res= 30 err=0 cout=0 oflow=0 g=0 e=0 l=0
# [                  445]Monitor TO Scoreboard: res=32, err=0, cout=0, oflow=0, g=0, e=0, l=0
# [                  445]------------Data From Reference Model--------- : Result:  30, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# [                  445]-------------data from monitor model---------- : result:  32, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# ** Error:                  445 : res mismatch  (ReF= 30, MON= 32)
#    Time: 445 ns  Scope: design_sv_unit.scoreboard.start File: testbench.sv Line: 671
#                  445 : err matches
#                  445 : cout matches
#                  445 : oflow matches
#                  445 : g matches
#                  445 : e matches
#                  445 : l matches
# -------------------------------------------------------------
```

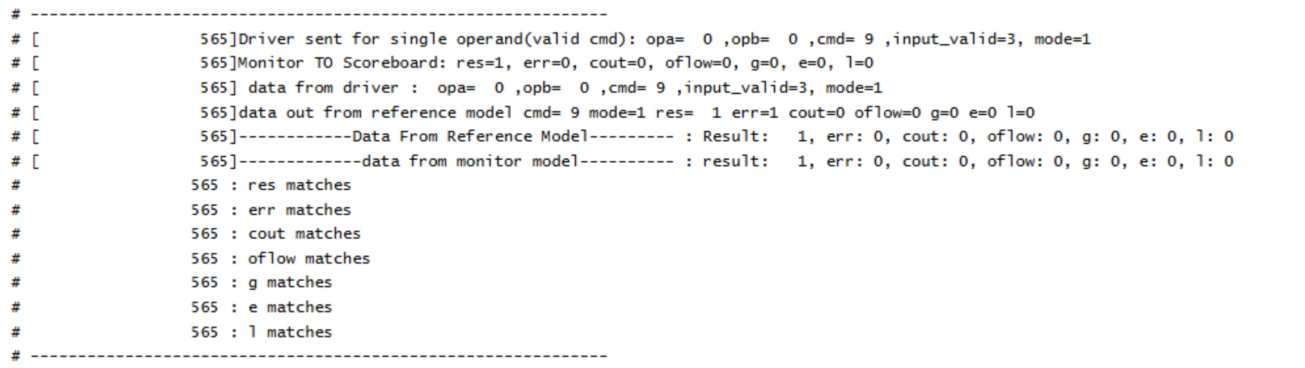


```
# -------------------------------------------------------------
# [                  395]Driver sent for single operand(valid cmd): opa= 24 ,opb= 47 ,cmd= 4 ,input_valid=3, mode=0
# [                  395] data from driver :  opa= 24 ,opb= 47 ,cmd= 4 ,input_valid=3, mode=0
# [                  395]data out from reference model cmd= 4 mode=0 res= 55 err=1 cout=0 oflow=0 g=0 e=0 l=0
# [                  405]Monitor TO Scoreboard: res=55, err=0, cout=0, oflow=0, g=0, e=0, l=0
# [                  405]------------Data From Reference Model--------- : Result:  55, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# [                  405]-------------data from monitor model---------- : result:  55, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
#                  405 : res matches
#                  405 : err matches
#                  405 : cout matches
#                  405 : oflow matches
#                  405 : g matches
#                  405 : e matches
#                  405 : l matches
# -------------------------------------------------------------
```

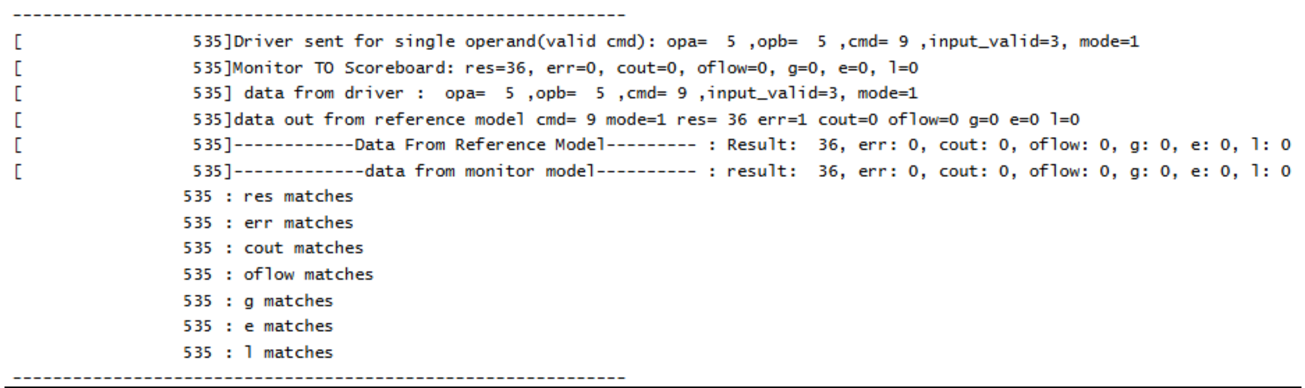
• For 3 clock cycle delay-



```
# [                  625]Driver sent for single operand(valid cmd): opa=  4 ,opb=  3 ,cmd= 9 ,input_valid=3, mode=1
# [                  625]Monitor TO Scoreboard: res=20, err=0, cout=0, oflow=0, g=0, e=0, l=0
# [                  625] data from driver :  opa=  4 ,opb=  3 ,cmd= 9 ,input_valid=3, mode=1
# [                  625]data out from reference model cmd= 9 mode=1 res= 20 err=1 cout=0 oflow=0 g=0 e=0 l=0
# [                  625]------------Data From Reference Model--------- : Result:  20, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# [                  625]-------------data from monitor model---------- : result:  20, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
#                   625 : res matches
#                   625 : err matches
#                   625 : cout matches
#                   625 : oflow matches
#                   625 : g matches
#                   625 : e matches
```



```
# ------------------------------------------------------------
# [                  565]Driver sent for single operand(valid cmd): opa=  0 ,opb=  0 ,cmd= 9 ,input_valid=3, mode=1
# [                  565]Monitor TO Scoreboard: res=1, err=0, cout=0, oflow=0, g=0, e=0, l=0
# [                  565] data from driver :  opa=  0 ,opb=  0 ,cmd= 9 ,input_valid=3, mode=1
# [                  565]data out from reference model cmd= 9 mode=1 res=  1 err=1 cout=0 oflow=0 g=0 e=0 l=0
# [                  565]------------Data From Reference Model--------- : Result:   1, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
# [                  565]-------------data from monitor model---------- : result:   1, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
#                   565 : res matches
#                   565 : err matches
#                   565 : cout matches
#                   565 : oflow matches
#                   565 : g matches
#                   565 : e matches
#                   565 : l matches
# ------------------------------------------------------------
```



```
 ------------------------------------------------------------
 [                  535]Driver sent for single operand(valid cmd): opa=  5 ,opb=  5 ,cmd= 9 ,input_valid=3, mode=1
 [                  535]Monitor TO Scoreboard: res=36, err=0, cout=0, oflow=0, g=0, e=0, l=0
 [                  535] data from driver :  opa=  5 ,opb=  5 ,cmd= 9 ,input_valid=3, mode=1
 [                  535]data out from reference model cmd= 9 mode=1 res= 36 err=1 cout=0 oflow=0 g=0 e=0 l=0
 [                  535]------------Data From Reference Model--------- : Result:  36, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
 [                  535]-------------data from monitor model---------- : result:  36, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
                   535 : res matches
                   535 : err matches
                   535 : cout matches
                   535 : oflow matches
                   535 : g matches
                   535 : e matches
                   535 : l matches
 ------------------------------------------------------------
```

3.1. Coverage

Coverage is a critical metric in functional verification used to measure how thoroughly the design has been tested. In the context of our UVM-based verification project, coverage helps ensure that the stimulus generated by the driver adequately exercises different parts of the Design Under Test (DUT), uncovering hidden bugs and corner cases.

Types of coverage relevant to this project:

• Functional coverage:

Focuses on verifying what has been tested, rather than how the DUT is implemented. In our project, functional coverage was captured through a dedicated covergroup (driver_cover) in the driver. It included:

o Coverpoints on key DUT inputs and control signals:

▪ IN_VALID to capture all possible input validity states.

▪ CMD divided into meaningful bins (cmd_first and cmd_second) based on comand range.

▪ Operand values (OperandA and OperandB), categorized into zero, small,and large bins.

▪ Control signals like CE (clock enable) and CIN (carry-in).

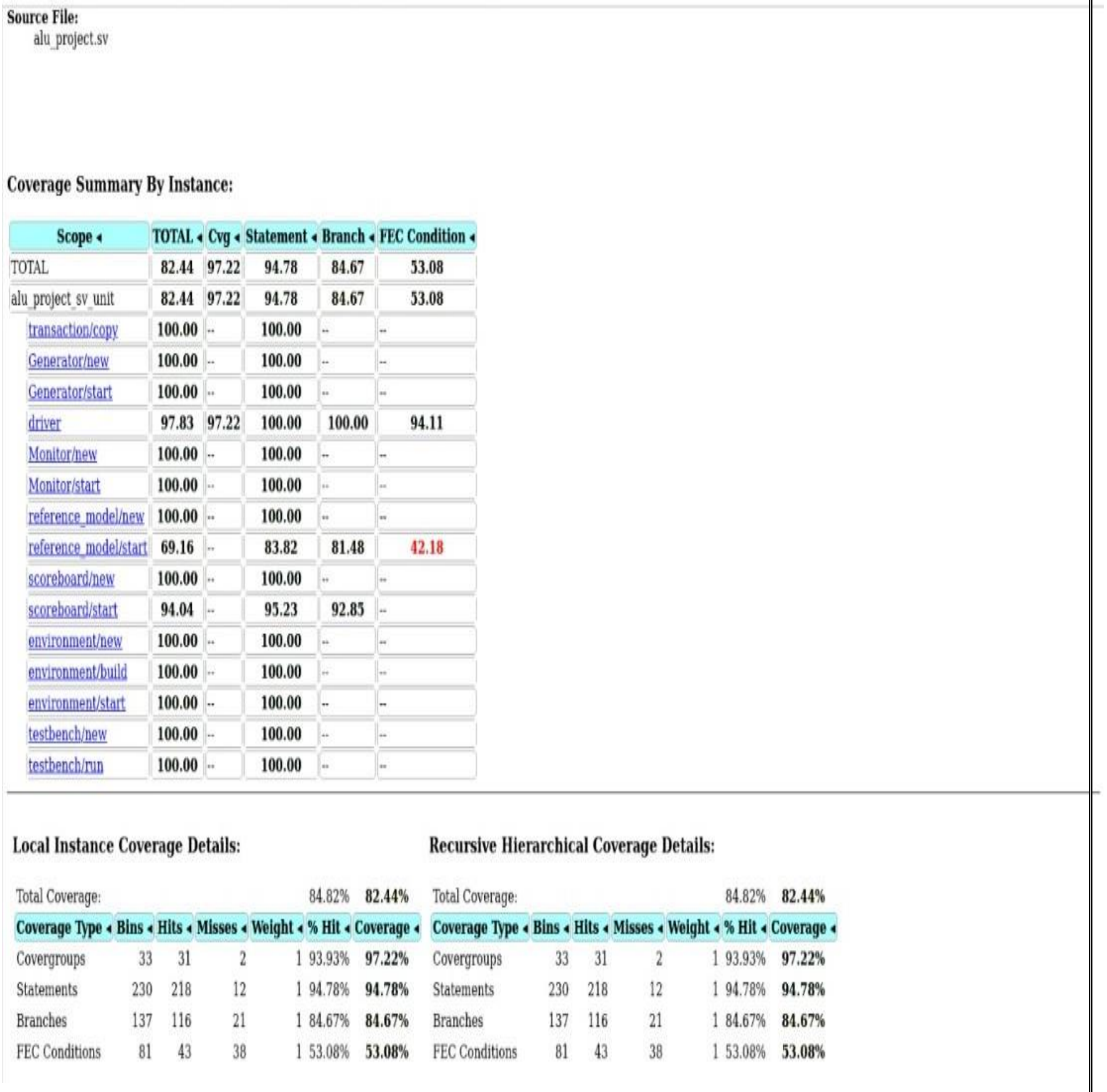
o Cross coverage:

▪ OperandA x OperandB to verify combinations of operands.

▪ Command x Input_Valid to ensure different commands are exercised under various input validity conditions.

• Code coverage (optional, usually reported by simulators):

Measures how much of the HDL code was actually executed during simulation,including statement, branch, and toggle coverage.While our main focus was functional coverage, code coverage complements it by identifying untested parts of the design implementation.



Source File:
alu_project.sv

Coverage Summary By Instance:

Scope	TOTAL	Cvg	Statement	Branch	FEC Condition
TOTAL	82.44	97.22	94.78	84.67	53.08
alu_project_sv_unit	82.44	97.22	94.78	84.67	53.08
transaction/copy	100.00	--	100.00	--	--
Generator/new	100.00	--	100.00	--	--
Generator/start	100.00	--	100.00	--	--
driver	97.83	97.22	100.00	100.00	94.11
Monitor/new	100.00	--	100.00	--	--
Monitor/start	100.00	--	100.00	--	--
reference_model/new	100.00	--	100.00	--	--
reference_model/start	69.16	--	83.82	81.48	42.18
scoreboard/new	100.00	--	100.00	--	--
scoreboard/start	94.04	--	95.23	92.85	--
environment/new	100.00	--	100.00	--	--
environment/build	100.00	--	100.00	--	--
environment/start	100.00	--	100.00	--	--
testbench/new	100.00	--	100.00	--	--
testbench/run	100.00	--	100.00	--	--

Local Instance Coverage Details:

Total Coverage: 84.82% **82.44%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	33	31	2	1	93.93%	**97.22%**
Statements	230	218	12	1	94.78%	**94.78%**
Branches	137	116	21	1	84.67%	**84.67%**
FEC Conditions	81	43	38	1	53.08%	**53.08%**

Recursive Hierarchical Coverage Details:

Total Coverage: 84.82% **82.44%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	33	31	2	1	93.93%	**97.22%**
Statements	230	218	12	1	94.78%	**94.78%**
Branches	137	116	21	1	84.67%	**84.67%**
FEC Conditions	81	43	38	1	53.08%	**53.08%**

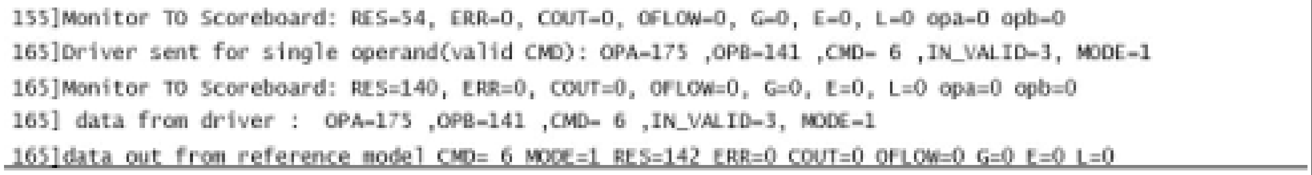
Chapter 4 Design Errors

4. During simulation and verification, we identified several design-related issues that were uncovered through functional coverage and driver stimulus. These include:

o The WIDTH of the result is 10 bits, but according to the specification the result should be 1 more than the WIDTH which is 8, due to this for multiplication of larger numbers is giving a Mismatch as 1 bit more in design

o The 16-clock cycle delay isn't working properly, it is not raising the flag high

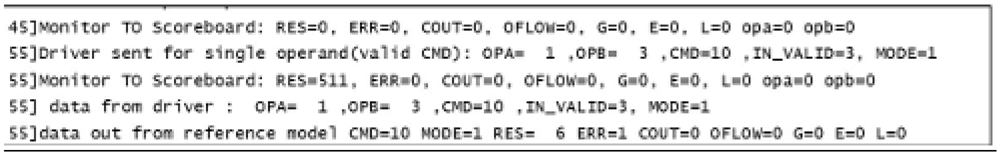
o For MODE 1, command 4 the operation is increment 1 but the design is just latching the same value.



```
155]Monitor TO Scoreboard: RES=54, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
165]Driver sent for single operand(valid CMD): OPA=175 ,OPB=141 ,CMD= 6 ,IN_VALID=3, MODE=1
165]Monitor TO Scoreboard: RES=140, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
165] data from driver :  OPA=175 ,OPB=141 ,CMD= 6 ,IN_VALID=3, MODE=1
165]data out from reference model CMD= 6 MODE=1 RES=142 ERR=0 COUT=0 OFLOW=0 G=0 E=0 L=0
```

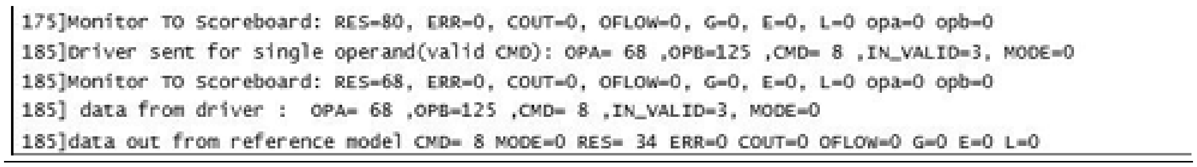
o For MODE 1, command 6 operation is incrementing operand B and command 7 is decrement operand B but outputs of both the operations are vice versa.

o The multiplication operation, command 10 the multiplication operator is replaced by subtraction operator, getting a output mismatch**.**



```
45]Monitor TO Scoreboard: RES=0, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
55]Driver sent for single operand(valid CMD): OPA=  1 ,OPB=  3 ,CMD=10 ,IN_VALID=3, MODE=1
55]Monitor TO Scoreboard: RES=511, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
55] data from driver :  OPA=  1 ,OPB=  3 ,CMD=10 ,IN_VALID=3, MODE=1
55]data out from reference model CMD=10 MODE=1 RES=  6 ERR=1 COUT=0 OFLOW=0 G=0 E=0 L=0
```

o For MODE 0, command 8 states as right shift of operand B but the design latches the same value the output is mismatch.



```
175]Monitor TO Scoreboard: RES=80, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
185]Driver sent for single operand(valid CMD): OPA= 68 ,OPB=125 ,CMD= 8 ,IN_VALID=3, MODE=0
185]Monitor TO Scoreboard: RES=68, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
185] data from driver :  OPA= 68 ,OPB=125 ,CMD= 8 ,IN_VALID=3, MODE=0
185]data out from reference model CMD= 8 MODE=0 RES= 34 ERR=0 COUT=0 OFLOW=0 G=0 E=0 L=0
```

o For MODE 0, command 10 the specification has the command shift right B but the design is performing shift left for operand B.

o For MODE 0, command 2 is OR operation but the design has mentioned and logical AND, causing a mismatch

Project result:

• Achieved a functional coverage of 82%, indicating that most of the intended stimulus space was exercised.

• The coverage model helped confirm that:

o All command ranges and input validity scenarios were stimulated.

o Combinations of operand values and control signals were tested.

• The remaining uncovered bins helped identify which rare scenarios were not hit, guiding further test enhancement.

Chapter 5 Challenges & conclusions

5. Challenges

• Timing delays for synchronization:

Implementing the correct wait logic in the driver to match the DUT's internal pipeline and operation latency was challenging.

o For certain commands, the DUT required the driver to wait 2 clock cycles after driving stimulus.

o For special commands (like CMD=9 or CMD=10), the DUT needed a 3-cycle delay to process correctly.

o When IN_VALID was initially partial (2'b10 or 2'b01), the driver had to wait and randomize for up to 16 cycles until valid input (2'b11) was achieved, making timing management complex.

• Achieving sufficient functional coverage:

Despite randomization, some bins (especially in cross coverage and rare operand combinations) remained uncovered, requiring additional directed stimulus.

• Debugging mismatches:

Tracing why DUT outputs occasionally differed from the reference model demanded detailed waveform analysis and careful comparison cycle by cycle.

• Handling partial input validity scenarios:

Ensuring the driver correctly detected partial validity and held or updated control signals properly until fully valid conditions were achieved.

• Balancing random vs. directed stimulus:

Relying solely on random transactions risked missing corner cases; adding directed tests increased coverage but added complexity to the testbench.

5.1. Conclusion

• Through systematic verification, we successfully tested the functionality of the DUT across a wide range of scenarios, including corner and boundary cases.By comparing DUT outputs against the reference model, we uncovered and documented several design errors, leading to improvements in the design logic.

• The project achieved a functional coverage of 82%, demonstrating that most intended stimulus scenarios were exercised, while also highlighting areas for further testing.

• This work not only validated the correctness of the DUT but also strengthened our understanding of driver implementation, functional coverage design, and the practical challenges of constrained-random verification.

• Overall, the project demonstrates the importance of a structured UVM approach in improving design quality and reliability before tape-out.