# ALU Verification Plan

# CHAPTER 1 –PROJECT OVERVIEW AND SPECIFICATIONS

ALU:-

Arithmetic Logic Unit (ALU) is a simple combinational circuit, where simple arithmetic, as well as logic operations, are performed in a digital system. During design, the ALU is created with the assistance of a hardware description language, Verilog, typically used for modeling a digital system.

## ALU significance:-

- Everything your computer calculates goes through the ALU
- Computers cannot perform math nor make decisions without ALU
- Executes incredibly fast - millions of operations every second

## 1.1 Pros of ALU:

- Can do basic math like adding, subtracting, multiplying, and dividing
- Helps make logical choices like AND, OR, NOT
- Can compare numbers to check which is bigger, smaller, or equal
- It's fast, correct, and takes up little space in the processor

## 1.2 Cons of ALU:

a. **No Memory**
   - Can't remember past calculations

b. **Speed Limit**
   - Can't go faster than the processor itself

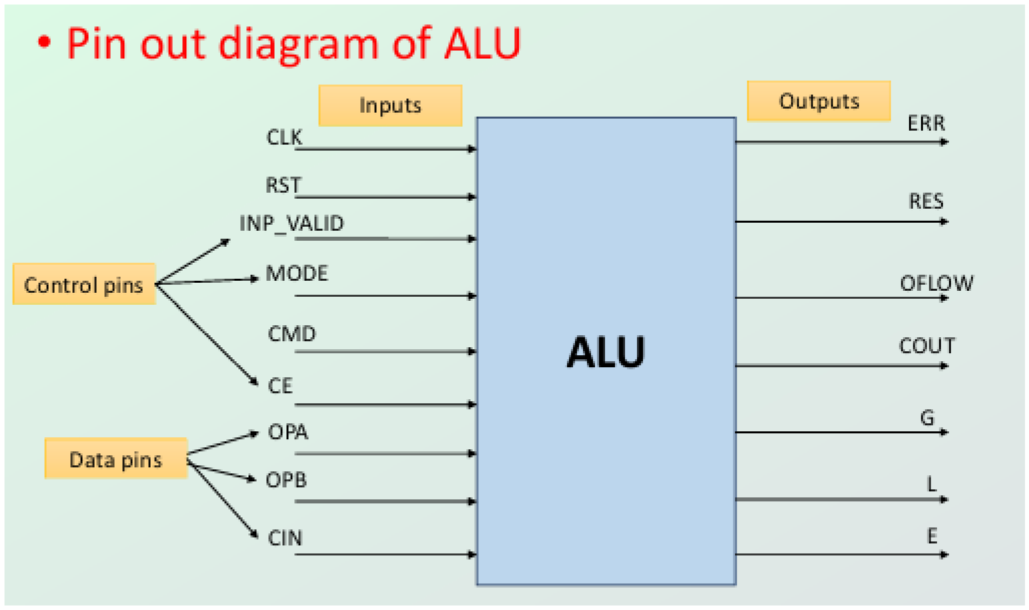
c. **High Power Use**
   - Uses more power when solving complex problems

## 1.3 Where ALU Is Used:

- **Microprocessors & Microcontrollers:** Does math and logic tasks
- **Digital Signal Processors (DSPs):** Used for tasks like filtering and sound/image changes
- **Embedded Systems:** Helps machines do control and automation work

- **Cryptography:** Used in data encryption with special math
- **AI & SIMD Units:** Helps with fast math in AI and graphics by working on many values at once

## 1.1.Project Overview of ALU:-



1. This project builds a flexible ALU using Verilog that can do many operations like math, logic, comparing values, and shifting/rotating bits. It is tested with System Verilog using a testbench that checks its behavior using coverage and rules (assertions).

2. The pin-out diagram shows how the ALU connects to other parts. It has separate input and output pins to help data move smoothly and control the operation. Control pins decide what the ALU should do and when, while data pins give the numbers to work with. The output shows the result and extra info like comparison results or errors.

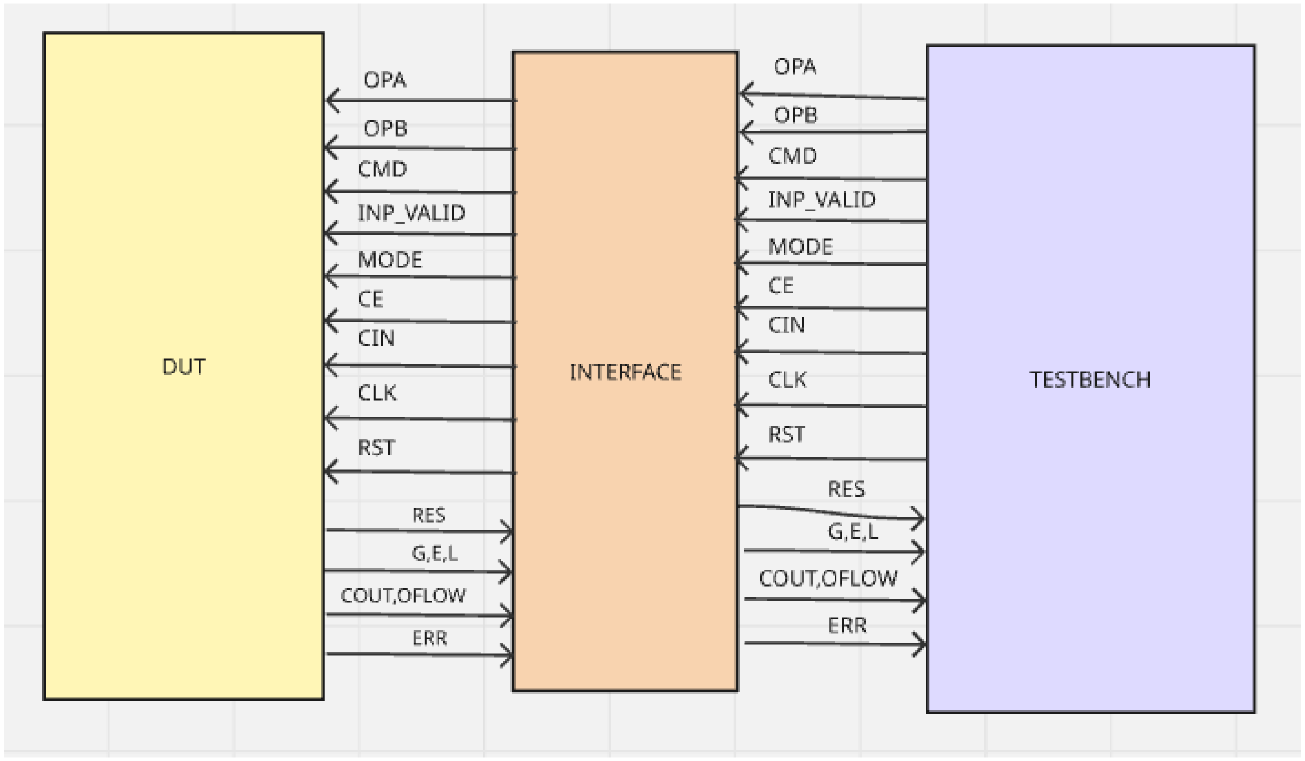
3. This kind of design makes the ALU easy to reuse, modify, and connect to bigger systems like CPUs and digital signal processors.

## 1.2.Design Features:-

| Serial no | Pin name | Direction | No of bits | Function |
|---|---|---|---|---|
| 1 | OPA | INPUT | Parametrized | Parameterized operand 1 |
| 2 | OPB | INPUT | Parametrized | Parameterized operand 2 |
| 3 | CIN | INPUT | 1 | This is the active high carry in input signal of 1-bit |
| 4 | CLK | INPUT | 1 | This is the clock signal to the design and it is edge sensitive |
| 5 | RST | INPUT | 1 | This is the active high asynchronous reset to the design |
| 6 | CE | INPUT | 1 | This is the active high clock enable signal 1-bit |
| 7 | MODE | INPUT | 1 | MODE signal 1 bit is high, then this is an Arithmetic Operation; otherwise, it is a Logical Operation |
| 8 | INP_VALID | INPUT | 2 | Operands are valid as per below table: 00: No operand is valid01: Operand A is valid10: Operand B is valid |
| 9 | RES | OUT | Parameterized +1 | This is the total parameterized plus 1 bits result of the instruction performed by the ALU |
| 10 | OFLOW | OUT | 1 | This 1-bit signal indicates an output overflow during Addition/Subtraction |
| 11 | COUT | OUT | 1 | This is the carry out signal of 1-bit during Addition/Subtraction |
| 12 | G | OUT | 1 | This is the comparator output of 1-bit which indicates that the value of OPA is greater than the value of OPB |
| 13 | L | OUT | 1 | This is the comparator output of 1-bit which indicates that the value of OPA is less than the value of OPB |
| 14 | E | OUT | 1 | This is the comparator output of 1-bit which indicates that the value of OPA is equal to the value of OPB |

|  |  |  |  |  |
|---|---|---|---|---|
| 15 | ERR | OUT | 1 | When CMD is selected as 12 or 13 and mode is logical operation, if 4th, 5th, 6th, and 7th bits of OPB are 1, then ERR bit will be 1; else it is high impedance |

## 1.3.Design diagram with interface signals:-



### 1.1.1. Driver Interface

| Signal Name | Direction (in clocking block) | Bit Width / Size | Description |
|---|---|---|---|
| OPA | output | `WIDTH` bits | Operand A |
| OPB | output | `WIDTH` bits | Operand B |
| CMD | output | `CMD_WIDTH` bits | Operation command |

|  |  |  |  |
|---|---|---|---|
| IN_VALID | output | 2 bits | Indicates which operands are valid |
| MODE | output | 1 bit | Arithmetic (1) or logical (0) mode |
| CE | output | 1 bit | Clock enable |
| CIN | output | 1 bit | Carry-in input |

### 1.1.2. Monitor Interface

| Signal Name | Direction (in clocking block) | Bit Width / Size | Description |
|---|---|---|---|
| RES | input | `WIDTH + 1` bits | Result of ALU operation |
| ERR | input | 1 bit | Error flag for illegal rotate etc. |
| OFLOW | input | 1 bit | Overflow flag |
| COUT | input | 1 bit | Carry-out flag |
| G | input | 1 bit | Comparator: OPA > OPB |
| E | input | 1 bit | Comparator: OPA == OPB |
| L | input | 1 bit | Comparator: OPA < OPB |

### 1.1.3. DUT Interface

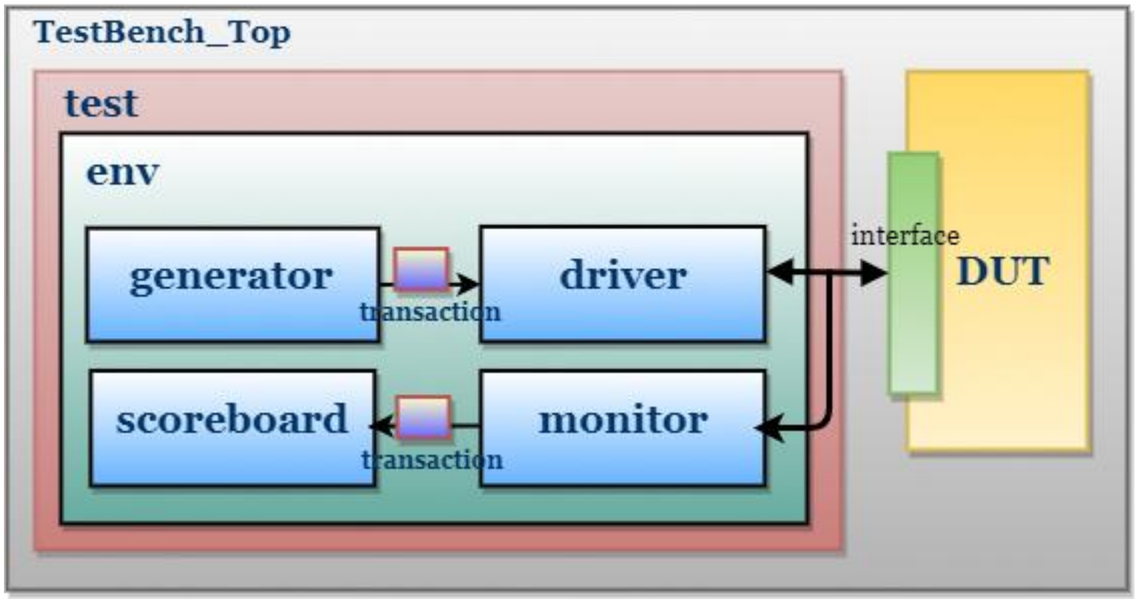
| Signal Name | Direction (in clocking block) | Bit Width / Size | Description |
|---|---|---|---|
| OPA | input | `WIDTH` bits | Operand A |
| OPB | input | `WIDTH` bits | Operand B |
| CMD | input | `CMD_WIDTH` bits | Operation command |
| IN_VALID | input | 2 bits | Indicates which operands are valid |
| MODE | input | 1 bit | Arithmetic (1) or logical (0) mode |
| CE | input | 1 bit | Clock enable |
| CIN | input | 1 bit | Carry-in input |

|  |  |  |  |
|---|---|---|---|
| RES | output | `WIDTH + 1` bits | Result of ALU operation |
| ERR | output | 1 bit | Error flag for illegal rotate etc. |
| OFLOW | output | 1 bit | Overflow flag |
| COUT | output | 1 bit | Carry-out flag |
| G | output | 1 bit | Comparator: OPA > OPB |
| E | output | 1 bit | Comparator: OPA == OPB |
| L | output | 1 bit | Comparator: OPA < OPB |

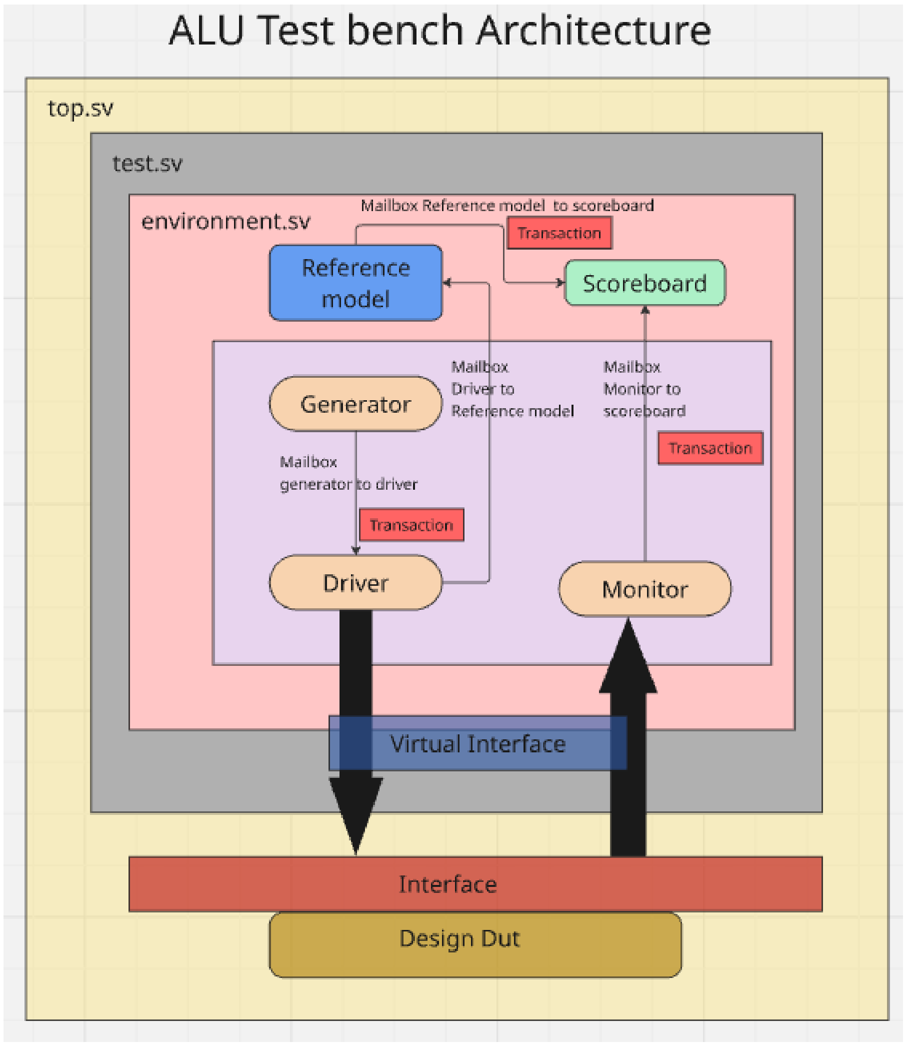
# CHAPTER 2 - Verification Architecture

## 2.1.Verification Architecture :-

General Testbench Architecture:

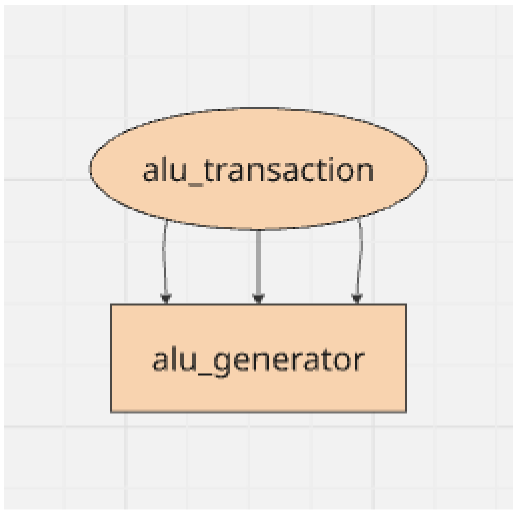


## 2.2.Component Details and Flowchart:


ALU Test bench Architecture
top.sv
test.sv
environment.sv
Mailbox Reference model to scoreboard
Transaction
Reference model
Scoreboard
Mailbox Driver to Reference model
Mailbox Monitor to scoreboard
Transaction
Generator
Mailbox generator to driver
Transaction
Driver
Monitor
Virtual Interface
Interface
Design Dut

Flow chart :

### 2.2.1. Transaction:



- In the transaction we have declared all the fields of the design both inputs and outputs and size of the signals .
- We also declared functions like copy that we use the deep copy to copy all the signals from the class transaction and sent that copied transaction to where we want using mailbox.
- We use this class to generate random stimulus in the generator.

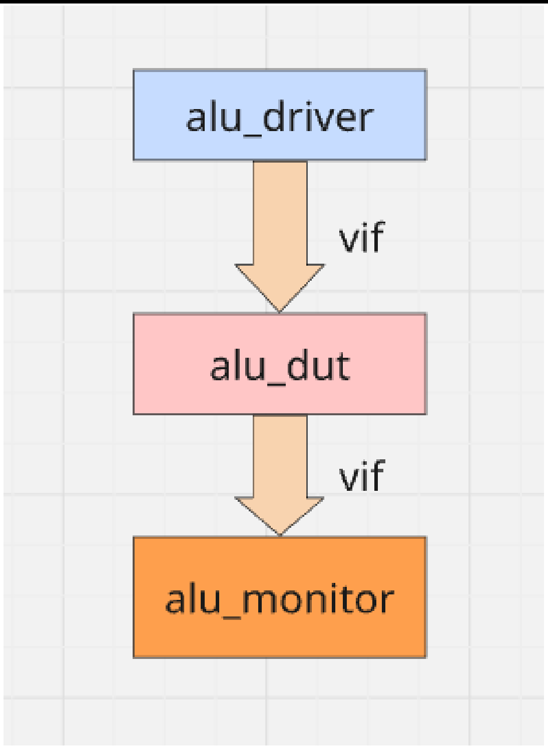
### 2.2.2. Interface:

- We use the interface to share common bunch of signals. And use them in between different components for connection.
- In this we declare interface between the driver, monitor and dut.
- We also wrote the mod ports to specify the direction of the signals and clocking block for synchronization between components.

2.2.2.1.1.1.1.

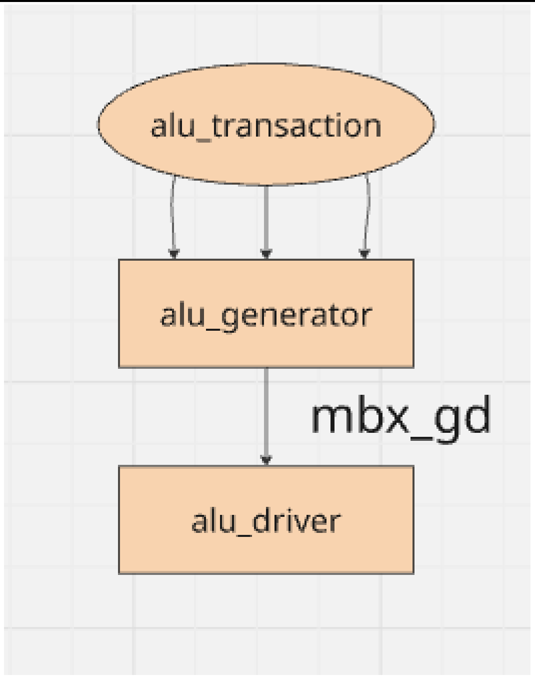
2.2.2.1.1.1.2.

2.2.2.1.1.1.3.

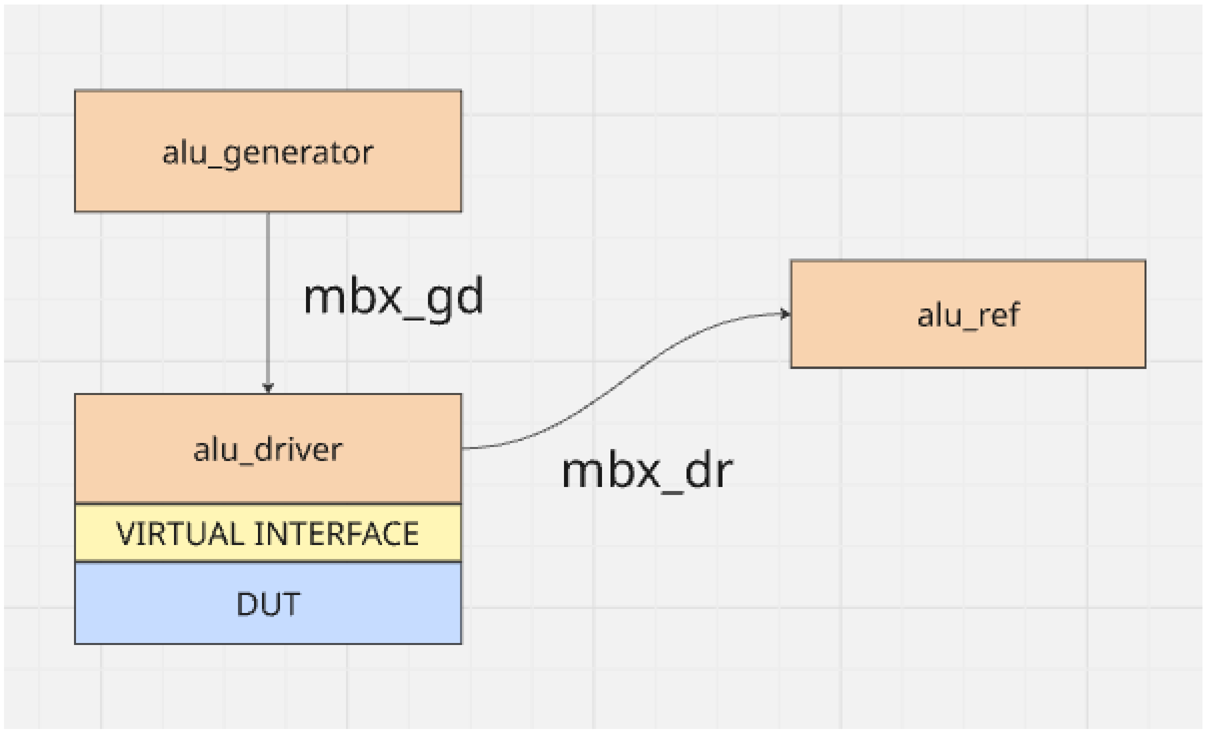


### 2.2.3. Generator:

- In generator we randomize all the stimulus and send to driver.
- According to design we want to keep the mode and ce and cmd same when we got two operand operation cmd and if the input valid is not 2’b11 then we have to wait for 16 clock cycles if it become 2’b11 before 16 cycles we will turn on the rand mode until we will turn off the rand mode of the ce,mode,cmd.
- The generated stimulus is sent to driver through the mbx_gd mailbox of type transaction class.
- I also used semaphore to synchronize the driver and the generator.



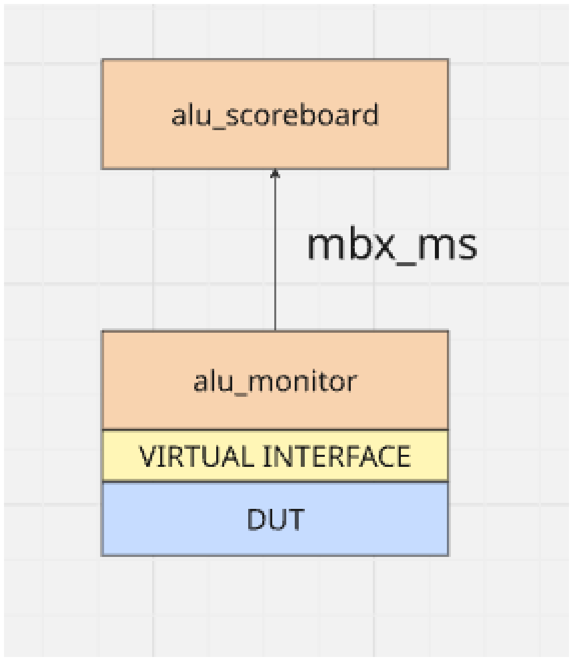
### 2.2.4. Driver:



- There are two mail boxes in the  driver one is to get the stimulus from the generator  and another one to send the stimulus from driver to reference model using the mail box mbx_dr.

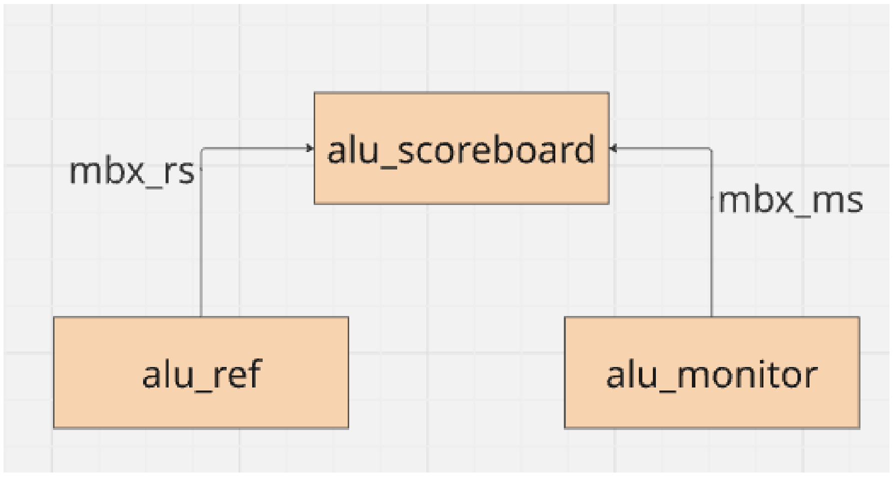
- We use the virtual interface to interact with DUT and send the signals to the design using interface only.

### 2.2.5. Monitor:



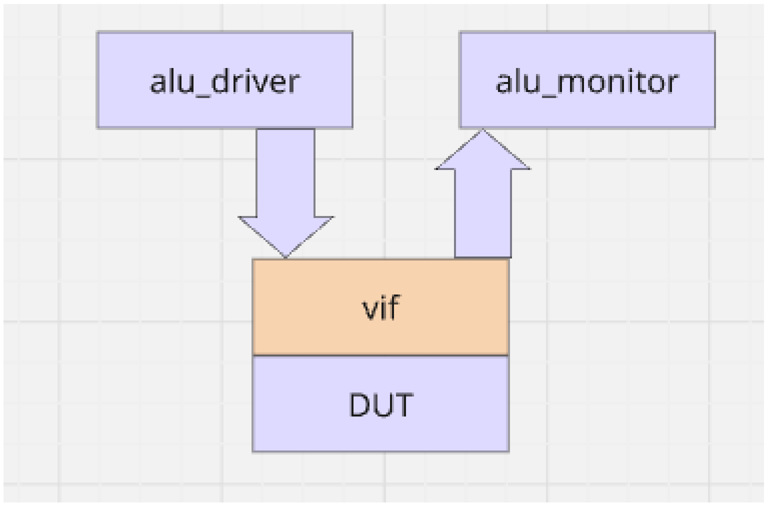
- In the monitor we have Interface where we will read the outputs of the design and send that to scoreboard using the mailbox mbx_ms.
- There is much logic in the monitor but to synchronize with the test driver make it difficult.

### 2.2.6. Scoreboard:



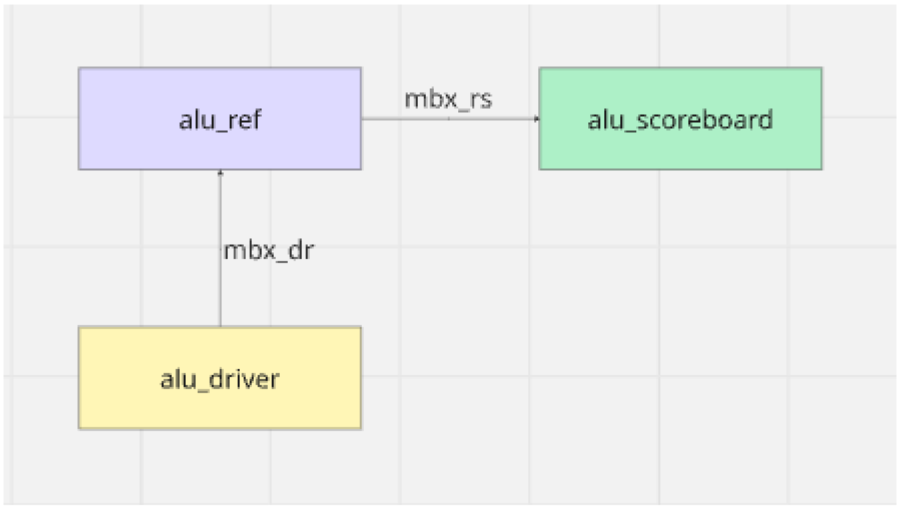
- In the score board we have 2 mailboxes(mbx_ms,mbx_rs).
- one mail box is used to get the outputs from the  monitor and another one is used  to get the output from the reference model .
- The main functionality of the scoreboard is to compare the values of the dut driven that we get from the monitor and the reference model give result according to that we evaluate our design.

### 2.2.7. DUT:



- The dut is the design instance created in the top module and it interacts with test bench through the virtual interface (vif) and the outputs are captured in the monitor and the inputs are sent to the dut using the driver.

### 2.2.8. Reference Model:



- The reference model has 2 mailboxes (mbx_rs ,mbx_dr).
- They are used to mbx_dr to drive the stimulus from the driver to the reference model and mbx_rs is used to send the result from the reference model to the scoreboard.
- The reference model is the golden model of the design.

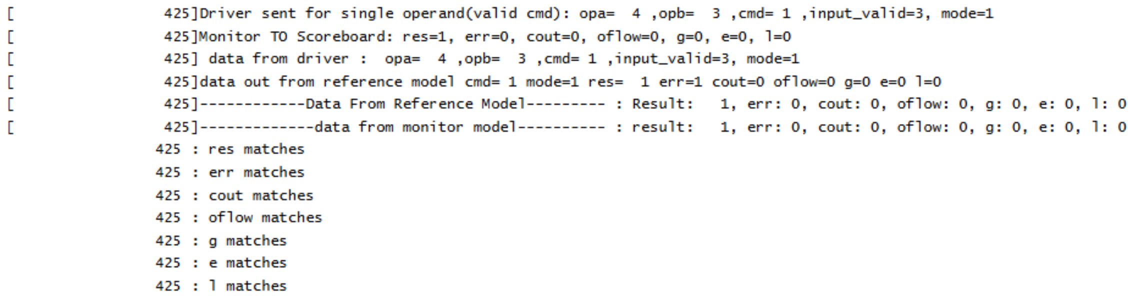
# Chapter 3 Outputs & Coverage

## 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 (IN_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.

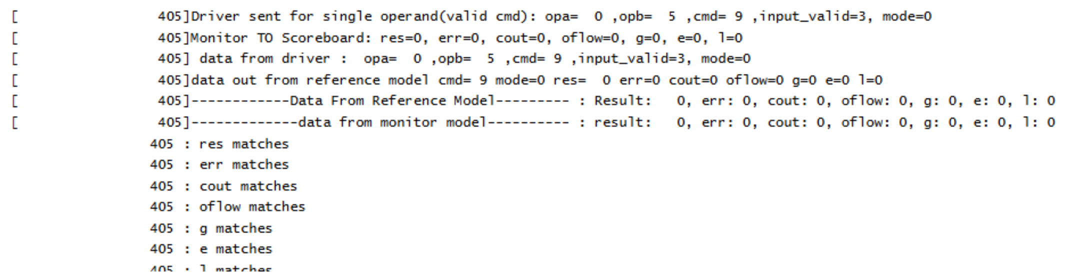
**Few examples: -**

- **For 2 clock cycle delay**-



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

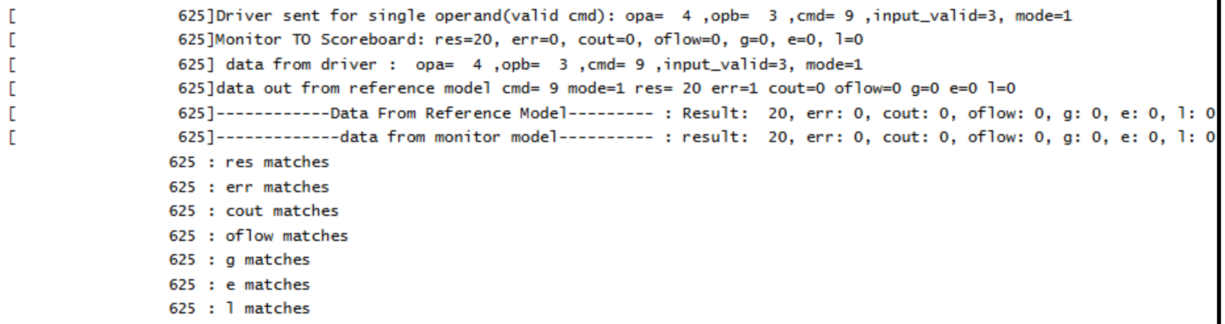
- 
  - This output is for the Subtraction operation, we can observe that the monitor has output 1(4 - 3) , and the reference model has also sent 1 as result .
  - The data is been driven at 405 from the driver to the DUT and we can observe at monitor and reference get the output at 425 which is 2 clock cycle delay.



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

- This output is for mode 0, Shift left A and we can observe that both the monitor and reference model are sending Result as 0 In the same clock cycle, so we can verify that DUT is working properly.

- **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
                625 : l matches
```

  - The above output is the output for multiplication which is a 3-cycle operation, in this we increment the operand A and B by 1 and then multiply.
  - We can observe that the output from monitor and reference model both are 20 which represents that the functionality of the DUT is correct.
  - The driver started sending he data at 595 and we can observe that at 625.i.e. after 3 clock cycles we are getting our expected output.

## 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.

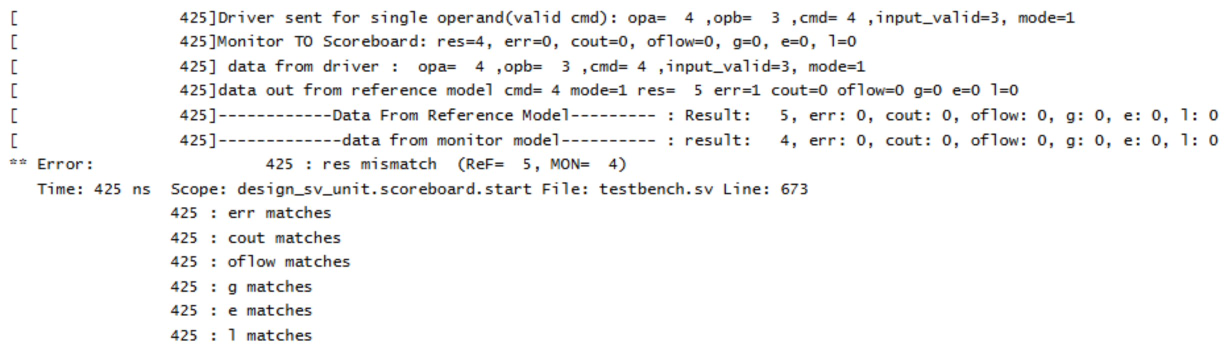
**Types of coverage relevant to this project:**

- **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:
  - **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 command range.
    - Operand values (OperandA and OperandB), categorized into zero, small, and large bins.
    - Control signals like CE (clock enable) and CIN (carry-in).
  - **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.

# Chapter 4 Design Errors

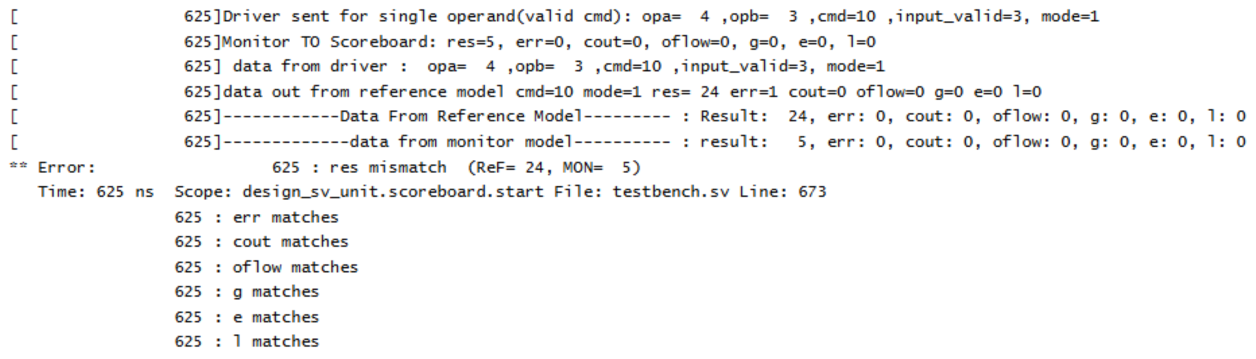
**During simulation and verification, we identified several design-related issues that were uncovered through functional coverage and driver stimulus. These include: -**

- 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
- The 16-clock cycle delay isn't working properly, it is not raising the flag high
- For MODE 1, command 4 the operation is increment 1 but the design is just latching the same value.



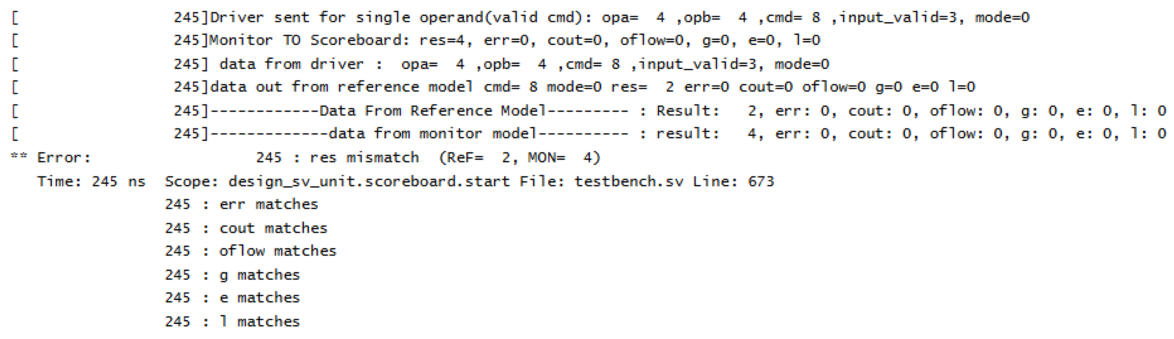
```
[                 425]Driver sent for single operand(valid cmd): opa=  4 ,opb=  3 ,cmd= 4 ,input_valid=3, mode=1
[                 425]Monitor TO Scoreboard: res=4, err=0, cout=0, oflow=0, g=0, e=0, l=0
[                 425] data from driver :  opa=  4 ,opb=  3 ,cmd= 4 ,input_valid=3, mode=1
[                 425]data out from reference model cmd= 4 mode=1 res=  5 err=1 cout=0 oflow=0 g=0 e=0 l=0
[                 425]------------Data From Reference Model--------- : Result:   5, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
[                 425]-------------data from monitor model---------- : result:   4, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
** Error:                  425 : res mismatch  (ReF=  5, MON=  4)
   Time: 425 ns  Scope: design_sv_unit.scoreboard.start File: testbench.sv Line: 673
                 425 : err matches
                 425 : cout matches
                 425 : oflow matches
                 425 : g matches
                 425 : e matches
                 425 : l matches
```

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



```
[                 625]Driver sent for single operand(valid cmd): opa=  4 ,opb=  3 ,cmd=10 ,input_valid=3, mode=1
[                 625]Monitor TO Scoreboard: res=5, err=0, cout=0, oflow=0, g=0, e=0, l=0
[                 625] data from driver :  opa=  4 ,opb=  3 ,cmd=10 ,input_valid=3, mode=1
[                 625]data out from reference model cmd=10 mode=1 res= 24 err=1 cout=0 oflow=0 g=0 e=0 l=0
[                 625]------------Data From Reference Model--------- : Result:  24, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
[                 625]-------------data from monitor model---------- : result:   5, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
** Error:                  625 : res mismatch  (ReF= 24, MON=  5)
   Time: 625 ns  Scope: design_sv_unit.scoreboard.start File: testbench.sv Line: 673
                 625 : err matches
                 625 : cout matches
                 625 : oflow matches
                 625 : g matches
                 625 : e matches
                 625 : l matches
```

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



```
[                 245]Driver sent for single operand(valid cmd): opa=  4 ,opb=  4 ,cmd= 8 ,input_valid=3, mode=0
[                 245]Monitor TO Scoreboard: res=4, err=0, cout=0, oflow=0, g=0, e=0, l=0
[                 245] data from driver :  opa=  4 ,opb=  4 ,cmd= 8 ,input_valid=3, mode=0
[                 245]data out from reference model cmd= 8 mode=0 res=  2 err=0 cout=0 oflow=0 g=0 e=0 l=0
[                 245]------------Data From Reference Model--------- : Result:   2, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
[                 245]-------------data from monitor model---------- : result:   4, err: 0, cout: 0, oflow: 0, g: 0, e: 0, l: 0
** Error:                  245 : res mismatch  (ReF=  2, MON=  4)
   Time: 245 ns  Scope: design_sv_unit.scoreboard.start File: testbench.sv Line: 673
                245 : err matches
                245 : cout matches
                245 : oflow matches
                245 : g matches
                245 : e matches
                245 : l matches
```

- For MODE 0, command 10 the specification has the command shift right B but the design is performing shift left for operand B.
- For MODE 0, command 2 is OR operation but the design has mentioned and logical AND, causing a mismatch.

# Chapter 5 Challenges & conclusions

## 3. 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.
- For certain commands, the DUT required the driver to wait 2 clock cycles after driving stimulus.
- For special commands (like CMD=9 or CMD=10), the DUT needed a 3-cycle delay to process correctly.
- 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.