ALU VERIFICATION DOCUMENT

Verification Plan

JULY 28, 2025

MIRAFRA TECHNOLOGY

INDEX

CHAPTER 1 – DESIGN OVERVIEW

1. ALU:

An **Arithmetic Logic Unit (ALU)** is a crucial digital circuit within a processor that performs both arithmetic operations, such as addition and subtraction, and logic operations like AND, OR, and XOR on binary data. Acting as the "computational core" of the CPU, the ALU directly executes the mathematical and logical instructions required by programs, enabling the processor to process and manipulate data. Its design typically includes multiplexers and logic gates that select and implement the desired operation based on control signals. The efficiency and capability of the ALU largely influence the overall speed and performance of a computing system.

1.1. Advantages of ALU

- Enables high-speed execution of arithmetic and logic operations.
- Supports multiple operations (addition, subtraction, bitwise logic) within a single unit.
- Compact design allows integration into CPUs, microcontrollers, and digital systems.
- Contributes to the programmability and flexibility of digital devices.
- Reduces the need for external computation hardware.

1.2. Disadvantages of ALU

- Limited to fixed, pre-designed operations and cannot easily adapt to specialized tasks.
- Complex ALU designs increase area and power consumption in hardware.
- Higher bit-width ALUs add latency, affecting overall system speed.
- Some arithmetic operations like division or floating-point computation require additional dedicated units.

1.3. Use cases of ALU

- Central part of CPUs and digital signal processors (DSPs) for data processing.
- Used in embedded systems and microcontrollers for sensor data computation.
- Key component in programmable logic controllers (PLCs) and automation systems.
- Employed in cryptography and graphics processors for fast bit manipulation.
- Integral to scientific calculators and digital measurement instruments.

1.4. Project Overview of ALU

- This project focuses on designing a fully parameterized, multi-functional Arithmetic Logic Unit (ALU) that can perform a wide range of arithmetic and logical operations based on configurable command inputs. The ALU accepts two parameterized-width operands (OPA and OPB), and supports addition, subtraction, increment, decrement, comparison, multiplication with pre-processing (e.g., increment or shift before multiply), as well as advanced logical functions including AND, OR, NAND, NOR, XOR, XNOR, bitwise NOT, and shifting operations.
- It includes specialized rotate operations (ROL_A_B and ROR_A_B) whose behavior depends on specific patterns in OPB, and incorporates built-in error detection when illegal operand patterns are encountered. The design features synchronous operation controlled by a clock (CLK), asynchronous reset (RST), and an active-high clock enable (CE) to manage data flow. The ALU differentiates between arithmetic and logical operations using the MODE input and validates operands based on INP_VALID.
- The outputs include the computed result (RES), status flags for overflow (OFLOW), carry out (COUT), and comparator flags (G, L, E), along with an error flag (ERR) to indicate invalid rotate conditions. This design demonstrates the integration of arithmetic computation, logical operations, comparator functions, and error handling into a single configurable ALU module suitable for digital processing applications, verification exercises, or as a reusable IP block in complex SoC designs.

1.5. Design Features

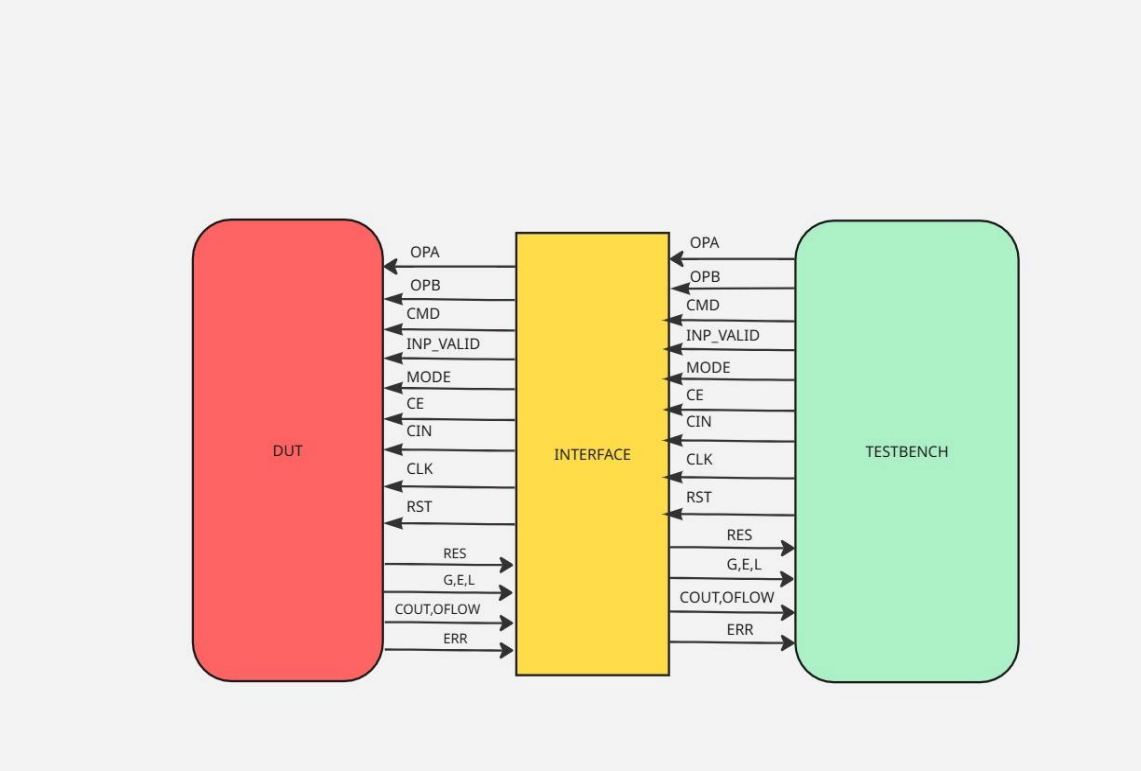
Serial no	Pin name	Direction	No of bits	Function
1	OPA	INPUT	Parametrized	Parameterized operand 1
2	OPB	INPUT	Parametrized	Parameterized operand 2
3	CIN	INPUT	1	This is the active high carry in input signal of 1-bit
4	CLK	INPUT	1	This is the clock signal to the design and it is edge sensitive
5	RST	INPUT	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

Serial no	Pin name	Direction	No of bits	Function
10	RES	OUT	Parameterized +1	This is the total parameterized plus 1 bits result of the instruction performed by the ALU
11	OFLOW	OUT	1	This 1-bit signal indicates an output overflow during Addition/Subtraction
12	COUT	OUT	1	This is the carry out signal of 1-bit during Addition/Subtraction
13	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
14	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
15	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
16	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

1.6. Design Limitation

- Supports only a fixed set of predefined arithmetic and logical operations; adding new operations requires design modification.
- Limited operand width defined by parameterization; very wide operands may increase area and reduce speed.
- Rotate operations (`ROL_A_B` and `ROR_A_B`) depend on specific operand patterns, and invalid patterns produce an error instead of a predictable fallback result.
- Comparator outputs (`G`, `L`, `E`) are based on unsigned comparison only; signed comparison is not supported.
- No built-in support for complex operations like division, modulo, or full multiplication beyond pre-processed multiply.
- Error flag (`ERR`) is specific to rotate commands and does not capture other types of functional errors or overflow in logical mode.
- The design is synchronous and edge-sensitive, so glitches or metastability issues may occur if inputs violate setup/hold requirements.
- No pipelining or parallel execution; performance may be limited in high-speed applications where multiple operations must be processed simultaneously.

1.7. Interfaces



-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

-Monitor Interface

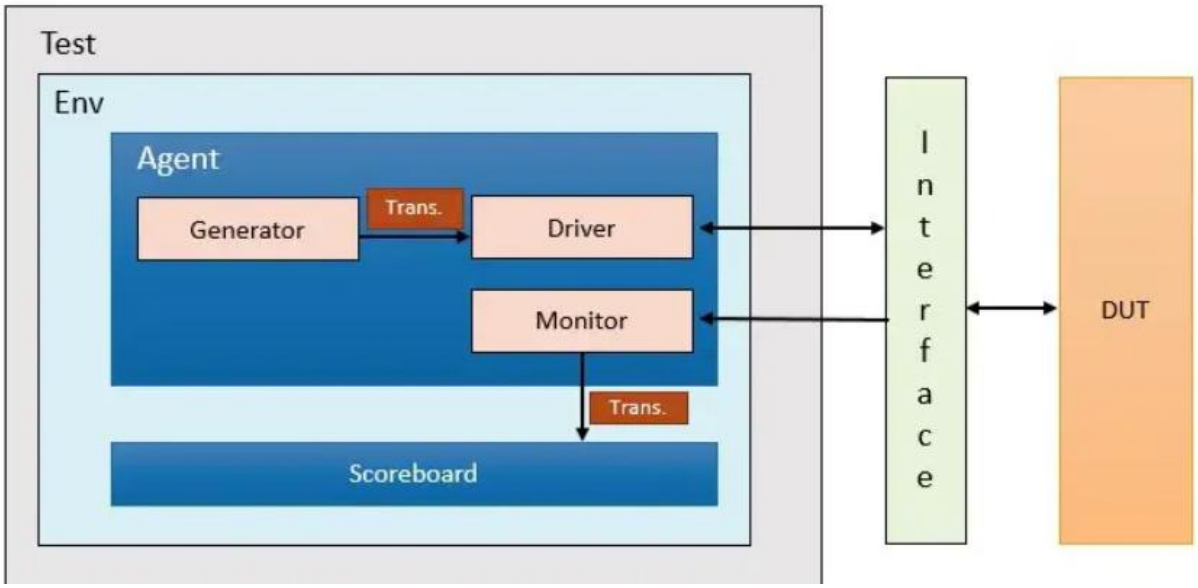
Signal Name	Direction (in clocking block)	Bit Width / Size	Description
RES	input	`WIDTH + 1` bits	Result of ALU operation
ERR	input	1 bit	Error flag for illegal rotate etc.
OFLOW	input	1 bit	Overflow flag
COUT	input	1 bit	Carry-out flag
G	input	1 bit	Comparator: OPA > OPB
E	input	1 bit	Comparator: OPA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

-Reference Interface

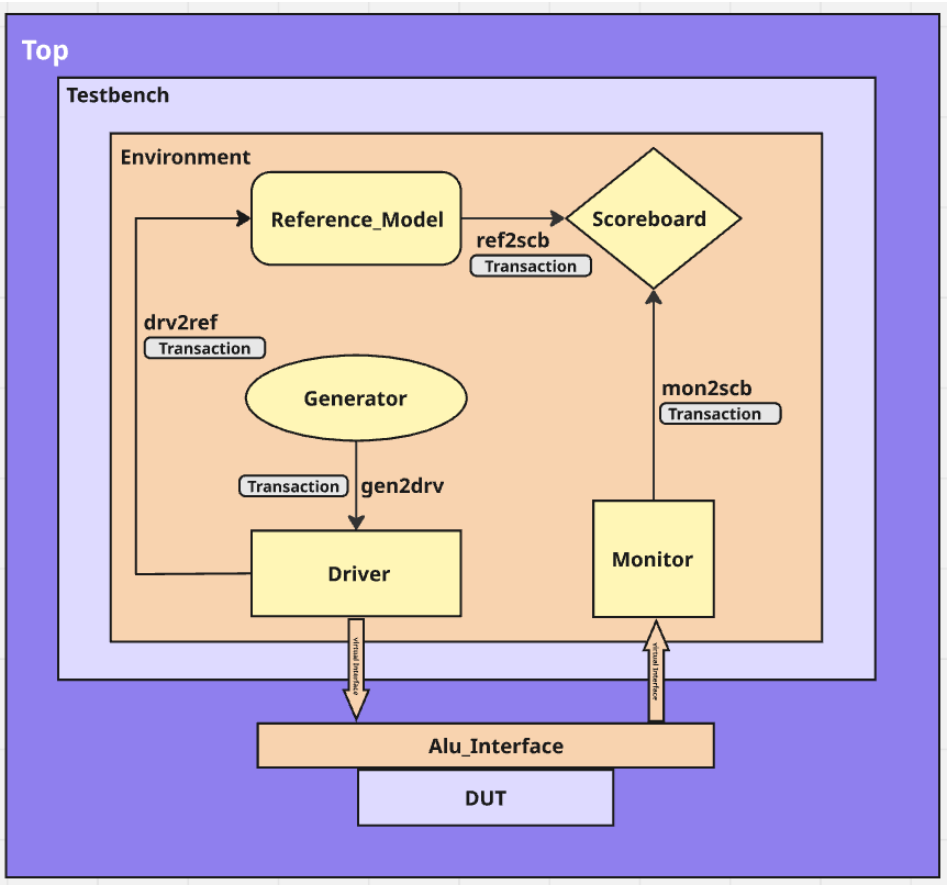
Signal Name	Direction (in clocking block)	Bit Width / Size	Description
OPA	input	`WIDTH` bits	Operand A
OPB	input	`WIDTH` bits	Operand B
CMD	input	`CMD_WIDTH` bits	Operation command
IN_VALID	input	2 bits	Indicates which operands are valid
MODE	input	1 bit	Arithmetic or logical mode
CE	input	1 bit	Clock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

Chapter 2 – VERIFICATION ARCHITECTURE

2. Verification Architecture

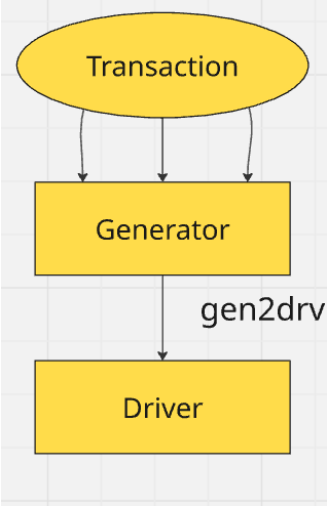


2.1. Verification Architecture for ALU



2.2. Flow-Chart for each component

i. Transaction

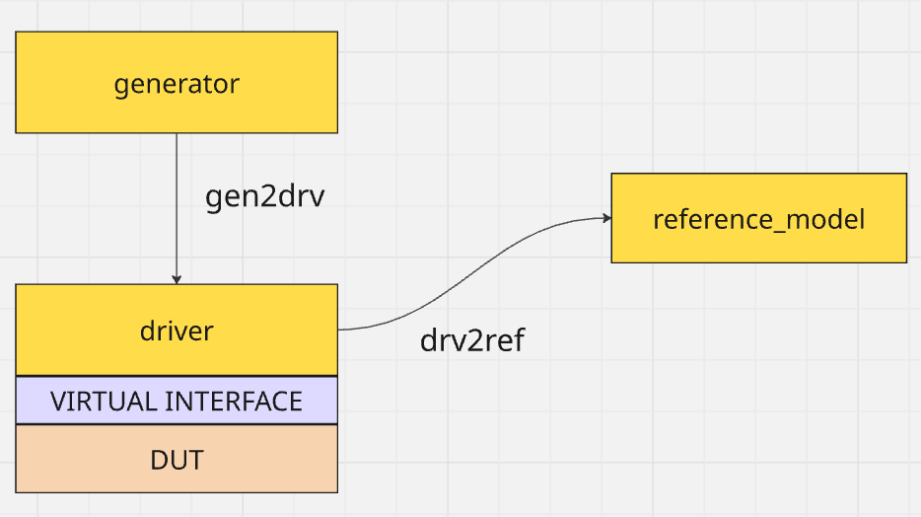


- Represents a single set of inputs and expected outputs for the ALU.
- Typically includes fields like OPA, OPB, CMD, MODE, IN_VALID, etc.
- Acts as the *data carrier* between generator, driver, monitor, and scoreboard.
- Designed as a System Verilog class with randomization support for constrained-random testing.
- Generator generates all the randomized fields from transaction

ii. Generator

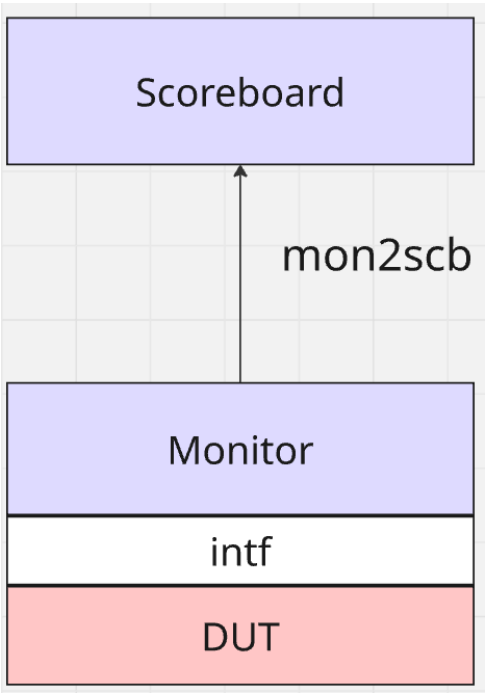
- Generates randomized or directed transaction objects.
- Controls the sequence of inputs to cover different ALU operations (add, sub, logical ops, rotate, etc.).
- May implement constraints to focus on interesting corner cases like overflow, carry, and invalid rotate conditions.
- Feeds transactions to the driver through a mailbox.

iii. Driver



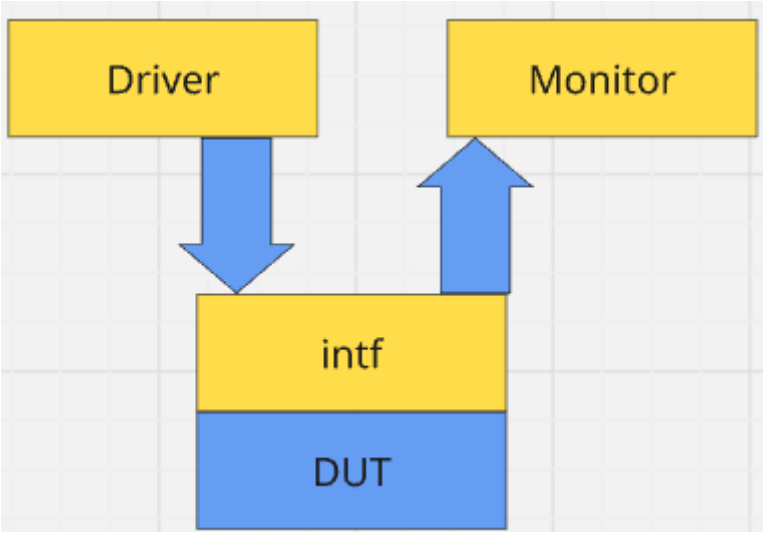
- Drives input signals from a transaction onto the DUT interface (via virtual interface).
- Reads fields like OPA, OPB, CMD, etc., from the transaction and assigns them to interface signals.
- Synchronizes with the DUT using clocking blocks and follows proper handshake protocols.
- Translates abstract transactions into actual pin-level activity on the DUT.
- Driver sends the data to reference model using the mailbox.

iv. Monitor



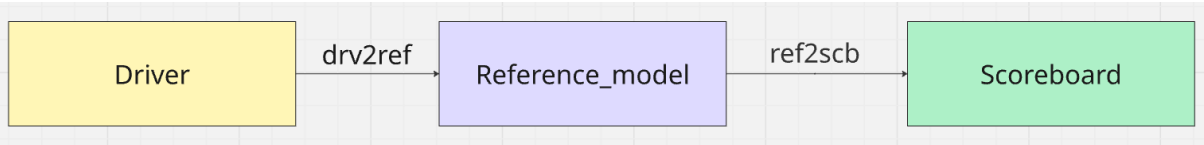
- Passively observes DUT inputs and outputs via the interface.
- Captures actual values of result (RES), flags (COUT, OFLOW, ERR), and comparator outputs (G, E, L).
- Creates new transactions or analysis data based on observed DUT behaviour.
- Sends captured data to the scoreboard for checking.

v. Interface



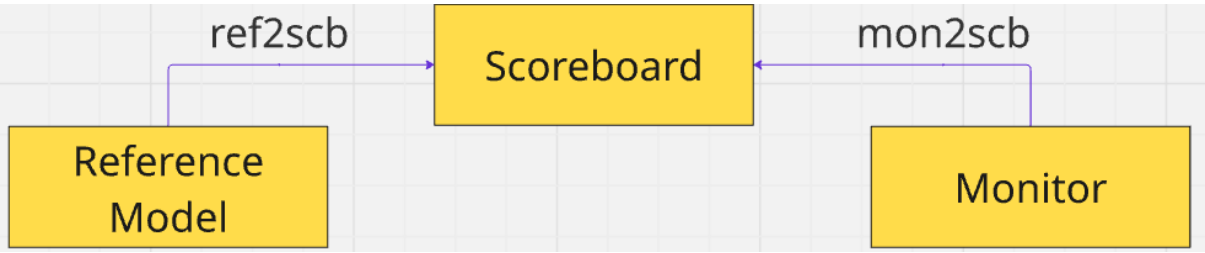
- The Alu_interface groups all DUT input and output signals into a single structured block, improving readability and maintainability.
- Clocking blocks (Drv_cb, Mon_cb, Ref_cb) inside the interface define timing and direction for driver, monitor, and reference, ensuring correct sampling and driving with respect to the clock.
- The **virtual interface** acts as a dynamic handle, allowing verification components (driver, monitor, reference model) to access the DUT signals indirectly.
- Using a virtual interface keeps the testbench modular and reusable, making it easier to swap DUT instances or change the environment without modifying component code.
- It enables consistent and synchronized communication between the DUT and verification components, simplifying the design of complex testbenches.

vi. Reference Model



- The reference model is a *golden model* that predicts the correct output of the ALU based on given inputs (OPA, OPB, CMD, MODE, etc.).
- It usually runs in parallel with the DUT during simulation, receiving the same inputs via the interface or virtual interface.
- The predicted results from the reference model are sent to the scoreboard to be compared against the actual DUT outputs (RES, COUT, OFLOW, etc.).
- Helps ensure functional correctness by detecting mismatches between expected and actual behavior.
- Acts as an independent, high-level behavioral model, making it easier to debug and isolate DUT design bugs.

vii. Scoreboard



- Receives expected transactions from generator (or reference model) and actual results from the monitor.
- Compares DUT outputs with expected values, considering corner cases like overflow, carry, or illegal commands.
- Logs mismatches and issues error reports if DUT behaviour doesn't match expectations.
- Helps measure functional correctness and coverage of the ALU.

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**-

```
45]Driver sent for single operand(valid CMD): OPA= 56 ,OPB= 10 ,CMD= 1 ,IN_VALID=3, MODE=1
45]Monitor TO Scoreboard: RES=46, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
45] data from driver :  OPA= 56 ,OPB= 10 ,CMD= 1 ,IN_VALID=3, MODE=1
45]data out from reference model CMD= 1 MODE=1 RES= 46 ERR=1 COUT=0 OFLOW=0 G=0 E=0 L=0
45]------------Data From Reference Model--------- : Result:  46, ERR: 0, COUT: 0, OFLOW: 0, G: 0, E: 0, L: 0
45]------------Data From Monitor Model---------- : Result:  46, ERR: 0, COUT: 0, OFLOW: 0, G: 0, E: 0, L: 0
```

  - This output is for the Subtraction operation, we can observe that the monitor has output 46 (56 - 10) , and the reference model has also sent 46 as result .
  - The data is been driven at 25 from the driver to the DUT and we can observe at monitor and reference get the output at 45 which is 2 clock cycle delay.

```
65]Driver sent for single operand(valid CMD): OPA=156 ,OPB= 45 ,CMD= 9 ,IN_VALID=3, MODE=0
65]Monitor TO Scoreboard: RES=56, ERR=0, COUT=0, OFLOW=0, G=0, E=0, L=0 opa=0 opb=0
65] data from driver :  OPA=156 ,OPB= 45 ,CMD= 9 ,IN_VALID=3, MODE=0
65]data out from reference model CMD= 9 MODE=0 RES= 56 ERR=0 COUT=0 OFLOW=0 G=0 E=0 L=0
```

- This output is for mode 0, Shift left A and we can observe that both the monitor and reference model are sending Result as 56 In the same clock cycle, so we can verify that DUT is working properly.

- **For 3 clock cycle delay-**

```
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= 9 ,IN_VALID=3, MODE=1
55]Monitor TO Scoreboard: RES=8, 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= 9 ,IN_VALID=3, MODE=1
55]data out from reference model CMD= 9 MODE=1 RES=  8 ERR=1 COUT=0 OFLOW=0 G=0 E=0 L=0
```

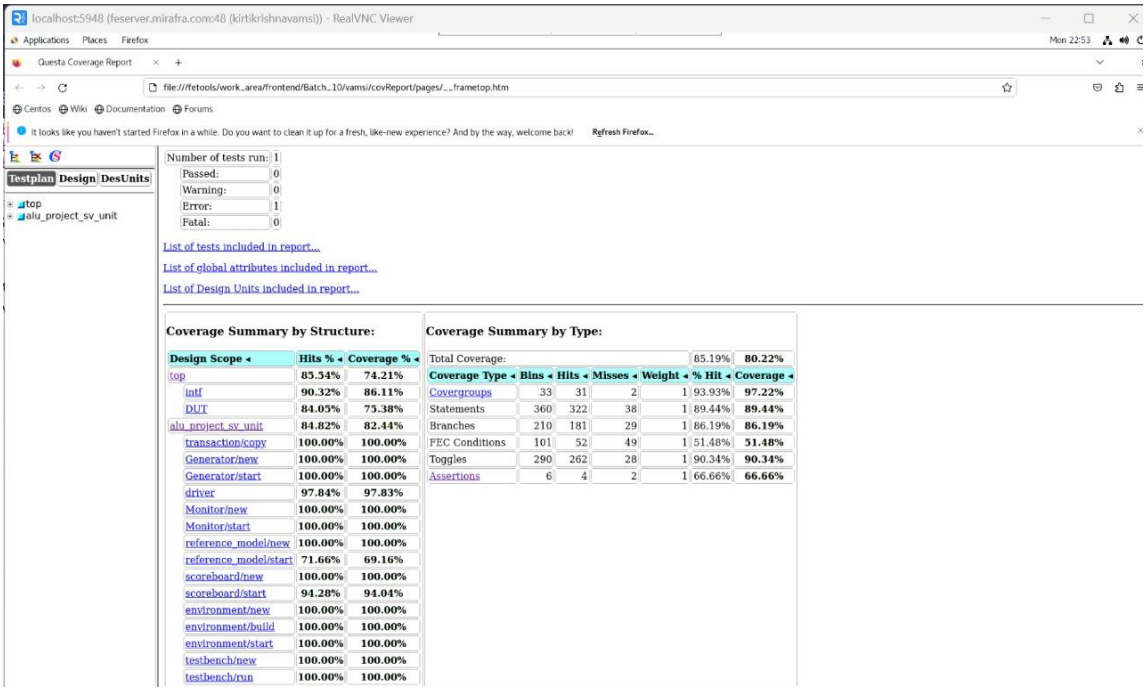
  - The above output is the output for multiplication which is a 3-cycle operation, in this we increment the operand A and B by 1 and then multiply.
  - We can observe that the output from monitor and reference model both are 8 which represents that the functionality of the DUT is correct.
  - The driver started sending he data at 25 and we can observe that at 55.i.e. after 3 clock cycles we are getting our expected output.

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.

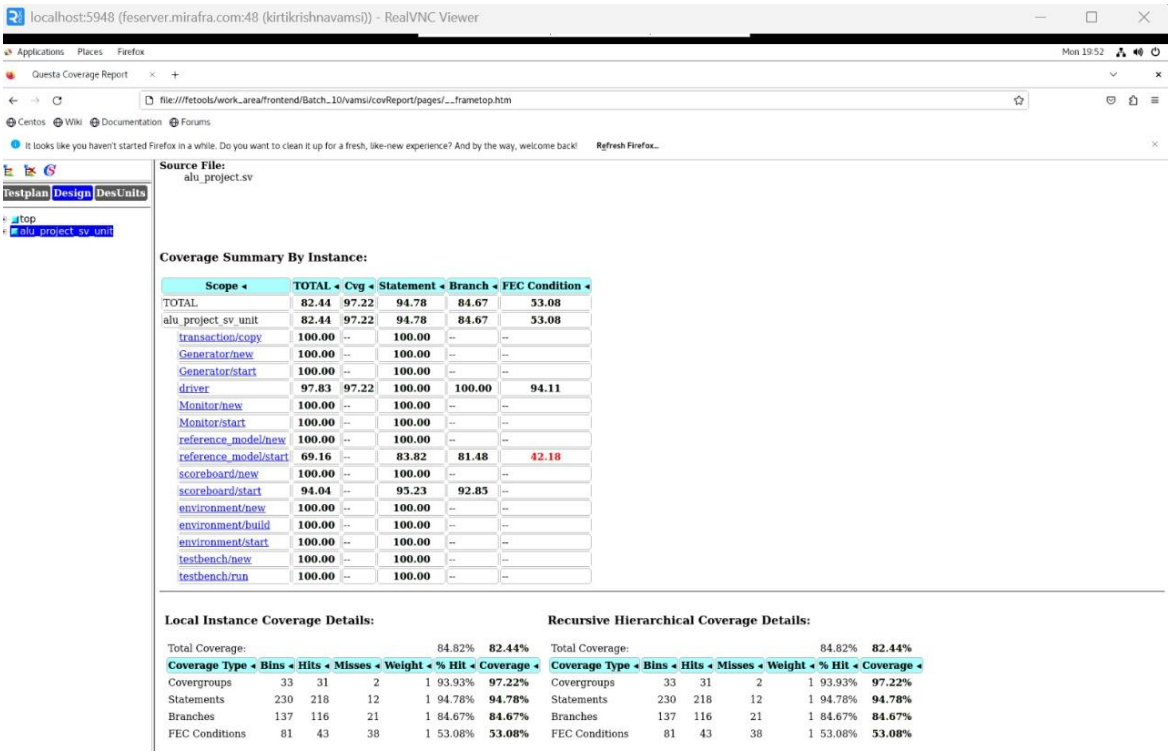
Types of coverage relevant to this project:

- **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:
  - **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.



Design Scope	Hits %	Coverage %
top	85.54%	74.21%
intf	90.32%	86.11%
DUT	84.05%	75.38%
alu_project_sv_unit	84.82%	82.44%
transaction/copy	100.00%	100.00%
Generator/new	100.00%	100.00%
Generator/start	100.00%	100.00%
driver	97.84%	97.83%
Monitor/new	100.00%	100.00%
Monitor/start	100.00%	100.00%
reference_model/new	100.00%	100.00%
reference_model/start	71.66%	69.16%
scoreboard/new	100.00%	100.00%
scoreboard/start	94.28%	94.04%
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%

Total Coverage:					85.19%	80.22%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	33	31	2	1	93.93%	97.22%
Statements	360	322	38	1	89.44%	89.44%
Branches	210	181	29	1	86.19%	86.19%
FEC Conditions	101	52	49	1	51.48%	51.48%
Toggles	290	262	28	1	90.34%	90.34%
Assertions	6	4	2	1	66.66%	66.66%



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%

Project result:

- Achieved a functional coverage of **82%**, indicating that most of the intended stimulus space was exercised.
- The coverage model helped confirm that:
    - All command ranges and input validity scenarios were stimulated.
    - 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 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: -**

- 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.

```
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
```

- 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.

```
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
```

- 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
```

- 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

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.
    - 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.

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.