ALU Verification Plan

N.Gowthaman

EMP ID:6086

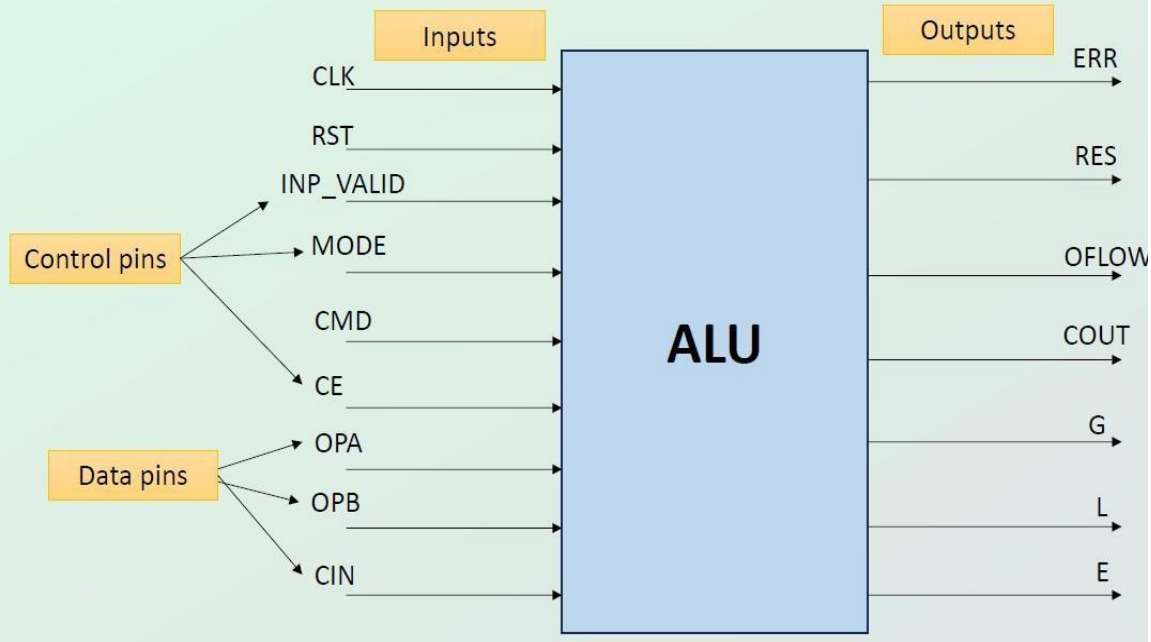
VERIFICATION DOCUMENT- ALU

CONTENTS	Page Number
Chapter -1	3-7
Chapter-2	7-15

CHAPTER 1 – DESIGN OVERVIEW

1. 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.



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.
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.

1.2 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.

1.3 Disadvantages of ALU:

- It has no memory and hence cannot remember previous calculations.
- Cannot work faster than the processor allows.
- Supports only specific bit widths such as 32-bit and 64-bit; larger data sizes are truncated during processing..
- Power consumption increases rapidly during high-complex calculations

1.4 Use cases of ALU:

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

- **Microprocessors and Microcontrollers** – Executes arithmetic and logic instructions (e.g., add, subtract, AND, OR).
- **DSPs (Digital Signal Processors)** – Used for filtering, FFT, and signal transformations.
- **Embedded Systems** – Performs calculations for automation and control logic.
- **Cryptographic Units** – Handles bitwise and modular arithmetic for encryption algorithms.
- **AI Accelerators / SIMD Units** – ALU arrays process matrix/vector operations efficiently.

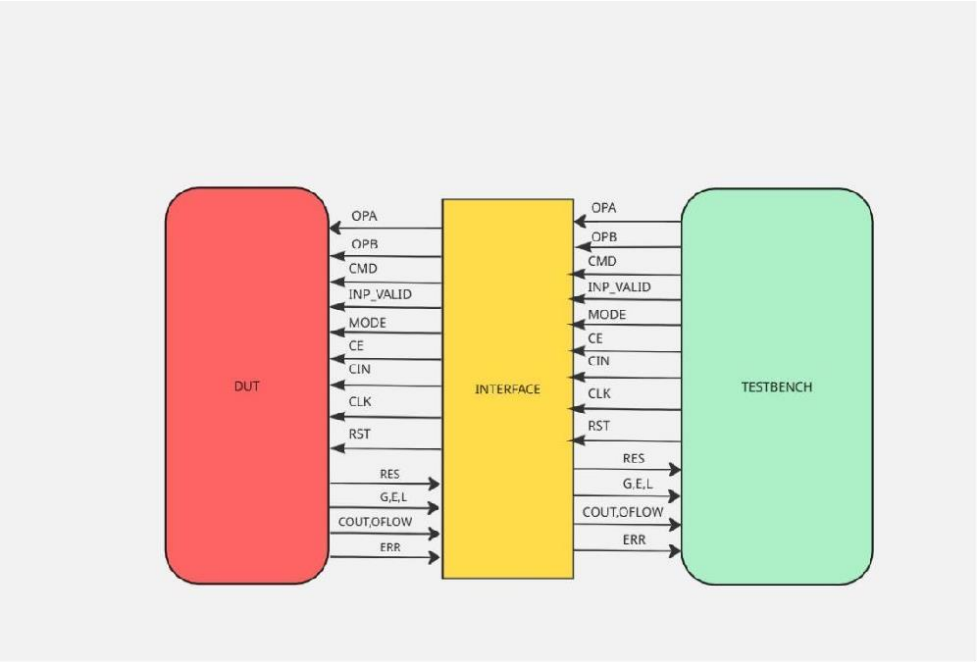
1.5 Project Overview of ALU:

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. the design ensures modularity, reusability, and ease of integration into larger digital systems such as CPUs and signal processors.

1.6 Design Features:

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

1.7 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

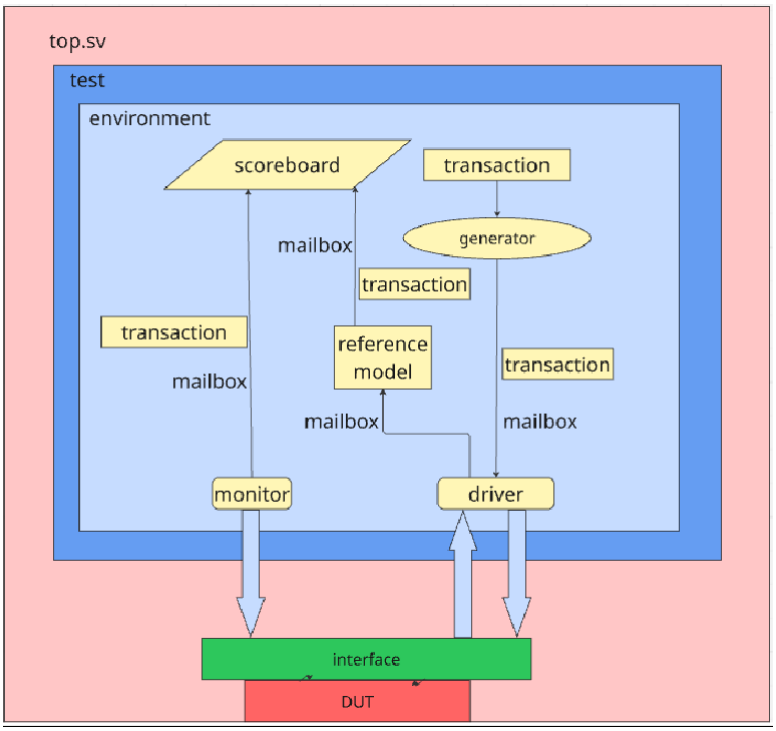
OPA	input	Operand A - first arithmetic/logic operand
OPB	input	Operand B - second arithmetic/logic operand
MODE	input	Mode selection signal - determines ALU operation mode
CE	input	Clock enable 1 indicates high
CIN	input	Carry In - input carry for arithmetic operations
CLK	input	Clock signal
RST	input	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

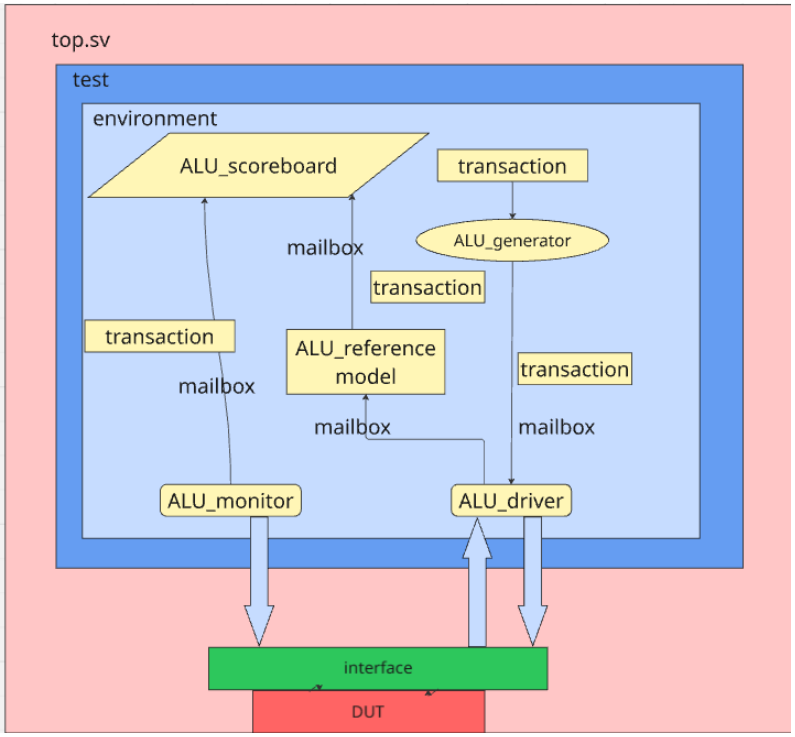
2.1 Verification Architecture :



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

- 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.

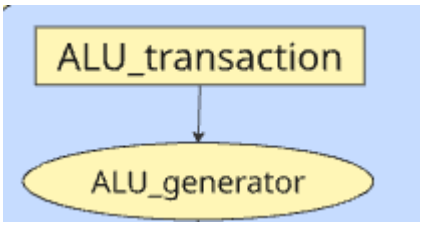
2.2 Verification Architecture for ALU:



2.3 FLOW CHART OF SV COMPONENTS :

1. Transaction class

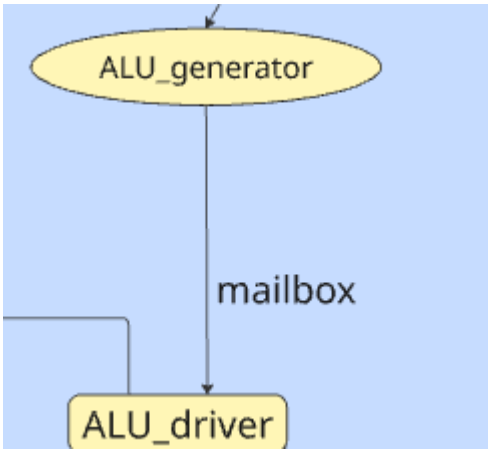
The transaction class encapsulates the input stimuli and output responses.



Deep Copy Method

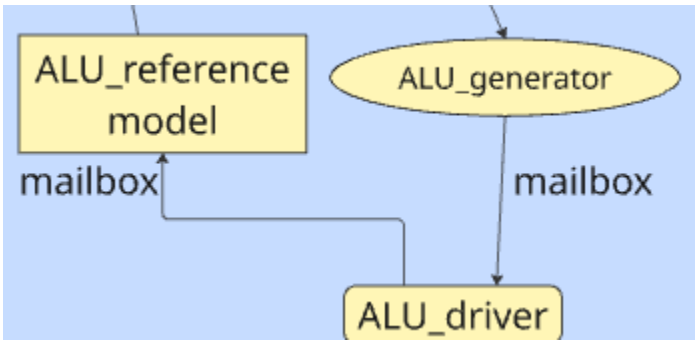
Implements a copy() function following the blueprint pattern for creating independent transaction copies used across testbench components.

2. Generator Class: -



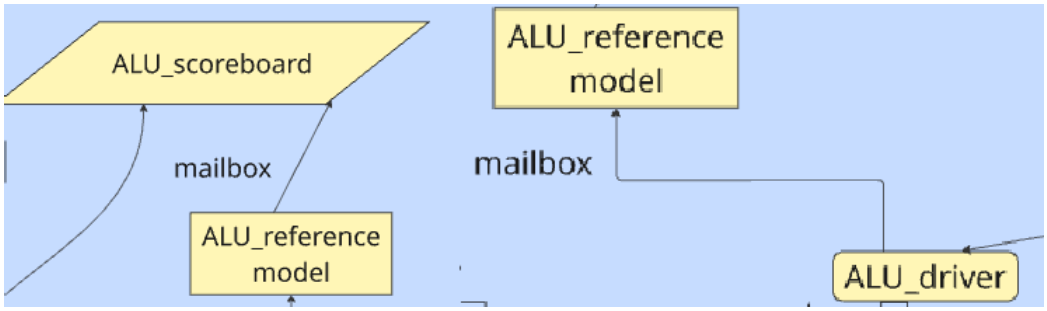
The generator consists of the ALU transaction class handle, the mailbox handle which connects to the driver, and the start() task which randomizes transactions and sends the randomized transactions to the driver through the mailbox.

3. Driver Class: -



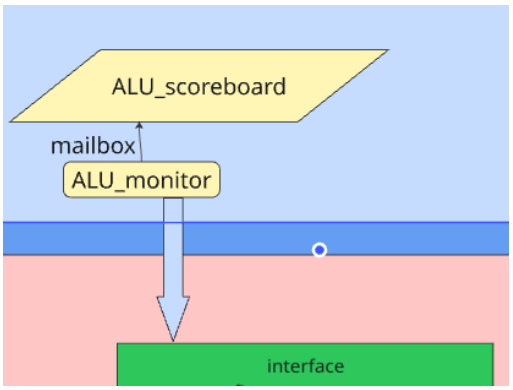
The driver class drives the input stimuli to the reference model.It receives the input stimuli generated by generator.It Communicates between testbench and DUV with proper synchronization.

4. Reference model



Replicates the DUT without delays. Continuously gets transactions from driver, computes expected results, and forwards to scoreboard.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.

6. Scoreboard:-

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

Chapter 3 Outputs & Coverage

1. 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.

```
# ----------------------------
# [GEN] data sent from generator
# time =375 opa=99,opb=99,cmd=12,cin=0,IN_VALID=11,ce=1,mode=0,RST=0|res=0,cout=0,oflow=0,g=0,e=0,l=0,err=0
# time = 375,cmd=12|mode=0|opa=234|opb=244|cin=1|inpvalid=3
# [MON] time = 375,res=174|err=1|cout=0|oflow=0|g=0|e=0|l=0
# [ref] time=375,cmd=12 | mode=0 | opa=234 | opb=244 | inp_valid=11 | res=0 | err=0
# [MON]time=375, data res=174|cout=0|oflow=0|err=1|g=0|e=0|l=0
# ----------------------------
# fail
# fail
# pass
# pass
# pass
# pass
# pass
# ----------------------------
```

```
# [GEN] data sent from generator
# time =30 opa=2,opb=1,cmd= 9,cin=0,IN_VALID=11,ce=1,mode=1,RST=0|res=0,cout=0,oflow=0,g=0,e=0,l=0,err=0
# [MON] time = 35,res=  0|err=0|cout=0|oflow=0|g=0|e=0|l=0
# [MON]time=35, data res=  0|cout=0|oflow=0|err=0|g=0|e=0|l=0
# ----------------------------
# pass
# pass
# pass
# pass
# pass
# pass
# pass
```

2 clock cycle

```
# ------------------------------
# time = 475,cmd=12|mode=0|opa= 99|opb= 99|cin=0|inpvalid=3
# [ref] time=475,cmd=12 | mode=0 | opa=99 | opb=99 | inp_valid=11 | res=0 | err=0
# time = 495,cmd=12|mode=0|opa=205|opb=130|cin=0|inpvalid=3
# [ref] time=495,cmd=12 | mode=0 | opa=205 | opb=130 | inp_valid=11 | res=0 | err=0
# time = 515,cmd=12|mode=0|opa=255|opb=207|cin=1|inpvalid=3
```

1.1. Coverage

Coverage is a critical metric in functional verification used to measure how thorough hly 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.

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

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.

- 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**.**

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.