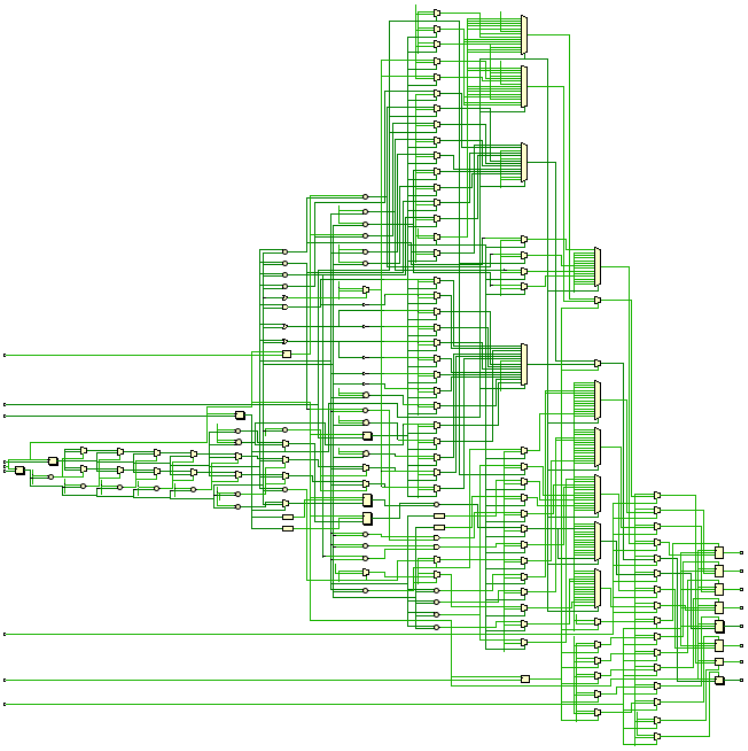


ALU PROJECT

ABSTRACT

This project presents the design and implementation of a versatile arithmetic logic unit (alu) capable of performing a wide range of arithmetic and logical operations. The alu accepts parameterized inputs and is controlled via a structured command set that allows dynamic operation selection.

Vadali Kirti Krishna Vamsi

EMP ID: - 6120

Design Documentation for ALU Project

1. Introduction:

An arithmetic logic unit (ALU) is a part of a central processing unit (CPU) that carries out arithmetic and logic operations. The ALU takes the input operands and an instruction and outputs the result**.** The purpose of an ALU is to speed up a CPU's overall processing by performing math and logic functions. By splitting out these functions, the different portions of the CPU can be more specialized and perform different operations simultaneously.

This project involves the design and implementation of a **multi-functional Arithmetic Logic Unit (ALU)** using **Verilog HDL**. The ALU serves as a core computational component capable of executing a wide range of arithmetic, logical, comparison, and shift operations. It is structured around a comprehensive input-output interface, including control signals, data buses, and status flags.

The design supports **parameterized operands** and a **rich instruction set**, including:

- Arithmetic operations
- Logical operations

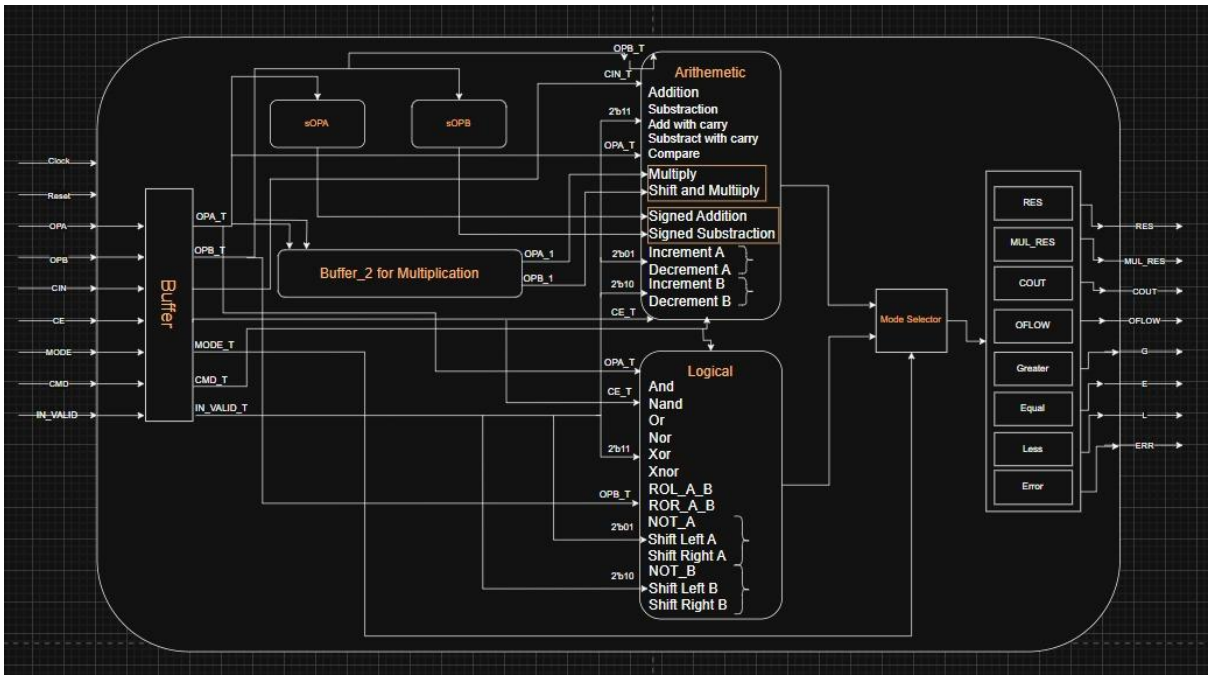
The ALU responds to a parameterized bit command signal (`CMD`) and a 1-bit mode signal (`MODE`) that determines whether the operation is arithmetic or logical. Additionally, it generates flags such as carry-out (`COUT`), overflow (`OFLOW`), error (`ERR`), and comparison outputs (`G`, `L`, `E`). The design is validated using simulation waveforms and timing diagrams that demonstrate correct operation under various scenarios. All the operations give the result on the next clock cycle where as the multiplication operations gives the result on the 3rd posedge of clock.

2. Objectives: -

- To design a versatile and parameterized ALU using Verilog.
- To support multiple arithmetic operations:
    - Addition
    - Subtraction
    - Add with carry
    - Subtract with carry
    - Decrement A
    - Decrement B
    - Compare
    - Increment and multiply
    - Shift and multiply
    - Signed Addition
    - Signed Subtraction
- To implement logical operations such as:
    - And
    - Nand
    - Or
    - Nor
    - Xor
    - Xnor
    - Not A
    - Not B
    - Shift right A
    - Shift Left A
    - Shift right B
    - Shift left B
    - Rotate right A by B
    - Rotate left A by B
- To support shift and rotate functions with operand-based control.
- To incorporate flag generation logic including carry-out, overflow, and comparator flags (greater, less, equal).
- To validate functionality through simulation using testbenches and waveform analysis.
- The delay of all the operations except multiply and shift and multiply should have a cycle delay whereas the multiply operations should have 3 clock cycle delay.
- The design and the test bench both should be able to perform the functions for any size of the parameterized input.

3. Architecture: -

- Design Architecture:



The architecture of the ALU is divided into the following modules:

3.1. Input Modules:

- **OPA, OPB:** Parameterized input operands.
- **CIN:** Carry input bit.
- **CLK, RST:** Clock and asynchronous reset.
- **CE:** Clock enable signal.
- **MODE:** Determines whether the operation is arithmetic (1) or logical (0).
- **INP_VALID:** Indicates which operands are valid.
- **CMD:** Operation selector encoded with various command values.

3.2. Control Logic:

- Decodes the CMD input to select the correct operation.
- Based on the mode, it differentiates between arithmetic and logical instructions.

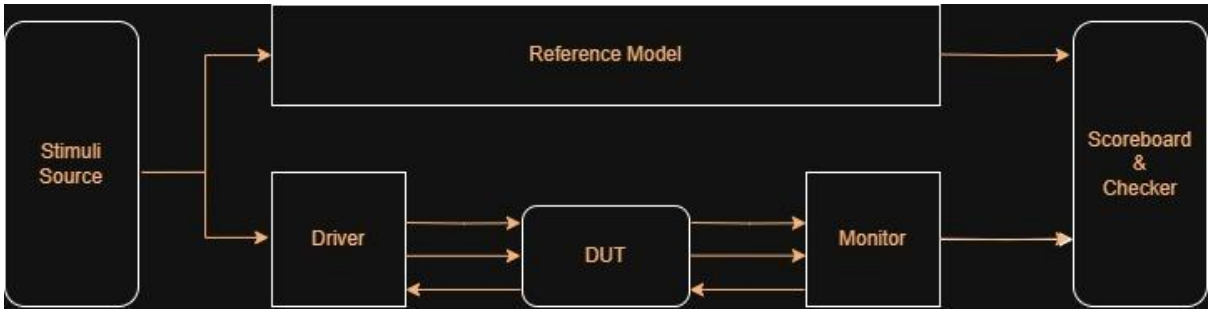
3.3. Operation Units:

- Arithmetic Unit handles addition, subtraction, increment, decrement, compare, etc.
- Logic Unit performs NOT, AND, OR, XOR, etc.
- Shift/Rotate Unit supports logical and circular shifts based on operand B.

3.4. Output Logic and Flags:

- **RES:** Result of operation.
- **COUT:** Carry-out flag for addition/subtraction.
- **OFLOW:** Overflow flag.
- **G, L, E:** Flags for Greater than, less than, and Equal comparison.
- **ERR:** Indicates invalid or illegal operations.

Testbench Architecture



Stimuli Source

- **Function:** Generates a sequence of input vectors or test cases.
- **Role:** Acts as the initiator in the verification process. It sends input data and control signals to both the **Driver** and the **Reference Model**.
- **Contents:** Could include random test generators, directed test cases, or constrained-random stimulus.

Driver

- **Function:** Takes the abstract inputs from the **Stimuli Source** and converts them into **pin-level signals** suitable for the **Design Under Test (DUT)**.
- **Role:** Acts as an interface that translates the high-level input to signals that the DUT understands.
- **Example for ALU:** It drives operands (`OPA`, `OPB`), operation command (`CMD`), mode, and clock/reset signals to the ALU.

DUT (Design Under Test)

- **Function:** The actual hardware module being verified — in this case, the **ALU**.
- **Role:** Receives input from the **Driver** and produces output results like `RES`, `COUT`, `OFLOW`, `G/L/E`, etc.

Monitor

- **Function:** Observes the DUT outputs non-intrusively.
- **Role:** It collects outputs from the DUT and sends them to the **Scoreboard and Checker**.
- **Additional Note:** It may also log the DUT response for debugging or waveform analysis.

Reference Model

- **Function:** Implements a golden model or behavioural version of the DUT.
- **Role:** Given the same inputs as the DUT, it computes the **expected output**.
- **Purpose:** Acts as a correctness oracle — used to compare against DUT outputs.

Scoreboard & Checker

- **Function:** Core of the verification engine.
- **Role:**
    - **Checker:** Compares the DUT output (via Monitor) with the **Reference Model** output.
    - **Scoreboard:** Keeps a record of all transactions and verifies functional coverage.
- **Result:** Flags mismatches or confirms correct behaviour.

4. Working:

The ALU design in Verilog uses a combination of combinational and sequential logic to handle various arithmetic and logical instructions. Here's a breakdown of how the working is implemented in code:

4.1 Operation Processing Logic

- When an operation is triggered via the `CMD` and `MODE` inputs, the operation type (arithmetic/logical) is decoded by control logic.
- The **valid input flags (`INP_VALID`)** ensure operands `OPA` and/or `OPB` are correctly used based on the type of operation.
- Based on the `CMD`, a corresponding logic block is executed using **case statements** in the Verilog code.

4.2 Temporary Registers and Delays

To simulate real hardware behaviour and control timing:

- **Temporary Registers (1-cycle delay)**:
  - For most arithmetic and logical operations, **one clock cycle** delay is introduced using **intermediate temporary registers**.
  - These registers store input values to temporary registers (`OPA_T`, `OPB_T`, IN_VALID_T etc.) and pass them to the final output after a clock edge.
- **Multiplication Delay (3-cycle delay)**:
  - Multiplication operations are more complex and use **two additional temporary registers** for the input operands (`OPA_1`, `OPA_2`) to introduce a **3-clock cycle delay,** in the first register the direct values are copied in the second posedge the values are updated on the basis of command and in the third clock cycle the results are passed.
  - This mimics the longer latency of multiplication hardware and helps in aligning output with pipeline designs in larger processors.
- **Flag Registers**:
  - Flags such as `COUT`, `OFLOW`, `G`, `L`, and `E` are also updated using clocked registers to ensure stable output after operation completes.

4.3 Flag Behaviour

- `COUT` is set if there's a carry out in addition/subtraction.
- `OFLOW` is set based on sign bit overflow detection.
- `G`, `L`, and `E` are set based on operand comparison.
- `ERR` is raised for invalid rotate/shift commands or unsupported modes.

4.4. Testbench Architecture

The ALU was verified using a **modular testbench**

4.4.1 Stimuli Source

- Generates randomized or directed input vectors for all signals (`OPA`, `OPB`, `CMD`, `MODE`, `CIN`, etc.).
- Can simulate edge cases like signed overflow, carry propagation, or invalid input formats.

4.4.2 Driver

- Applies the generated input vectors to the DUT (ALU) by driving them on the ALU interface.
- Ensures timing and control signals (`CLK`, `RST`, `CE`, `INP_VALID`) are also correctly applied.

4.4.3 DUT (Design Under Test)

- The instantiated ALU module.
- Receives the inputs and produces outputs which are then captured for comparison.

4.4.4 Monitor

- Observes and records DUT outputs (`RES`, `COUT`, `OFLOW`, etc.).
- Feeds these to the scoreboard for result validation.

4.4.5 Reference Model

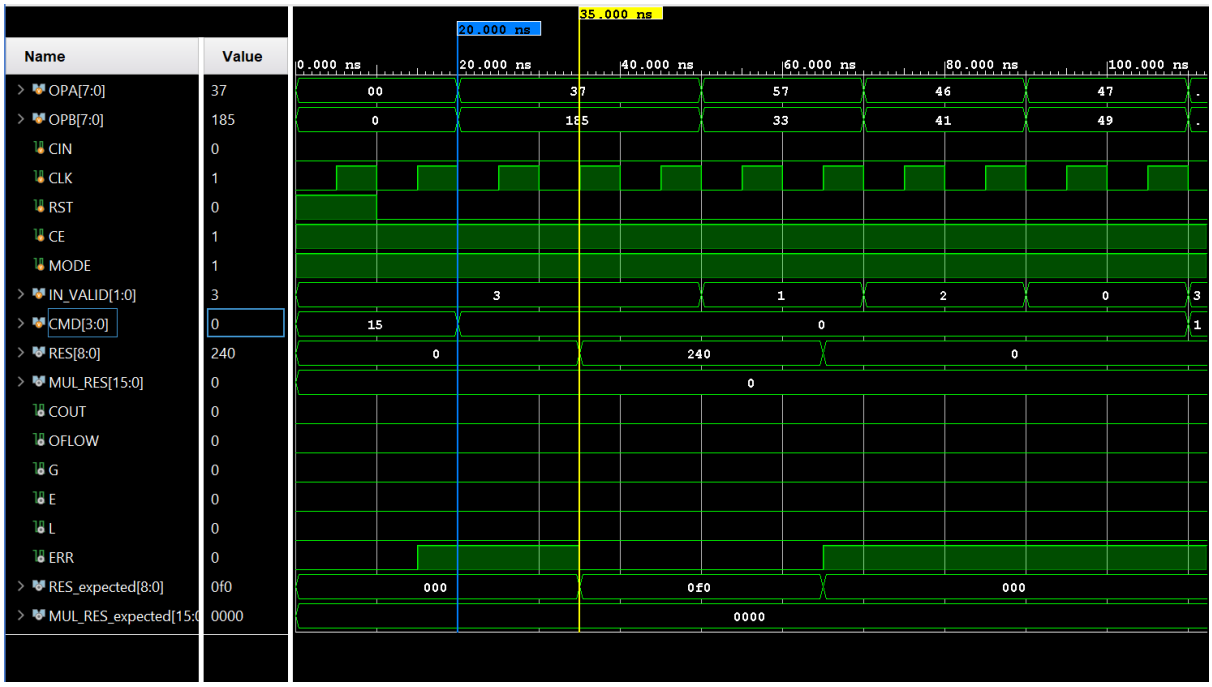
- A behavioural model of the ALU written in the testbench environment.
- Simulates the same operation independently to produce expected results for each test case.

4.4.6 Scoreboard & Checker

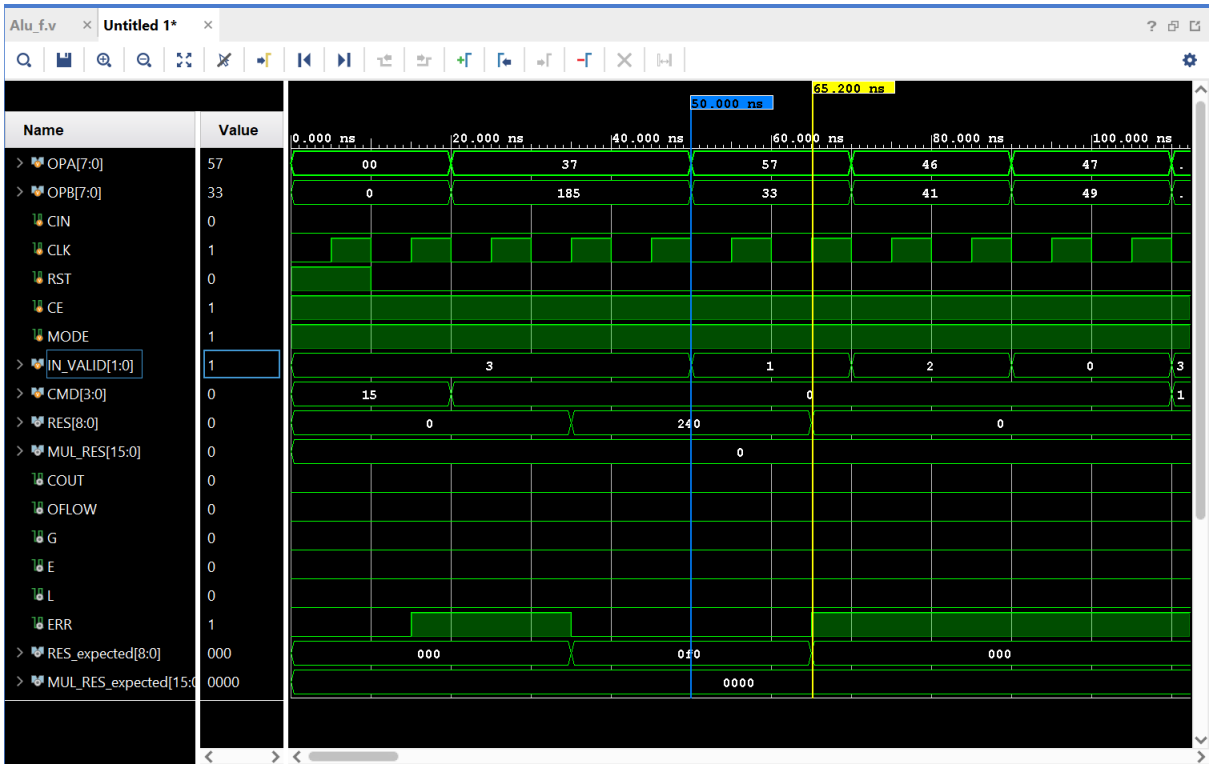
- Compares the **DUT outputs** with those from the **Reference Model**.
- Logs pass/fail status for each operation.
- Reports mismatches with debug information including input vectors, expected vs actual output.

5. Result: -

- The above timing diagram showcases the working of multiplication operation in which we can observe that when the command asserts binary – 1001 i.e. 9, the result is changed to 5180, this is because on the first posedge processing starts, on the second posedge the OPA and OPB increments it's value and on the third posedge RES gets the value.

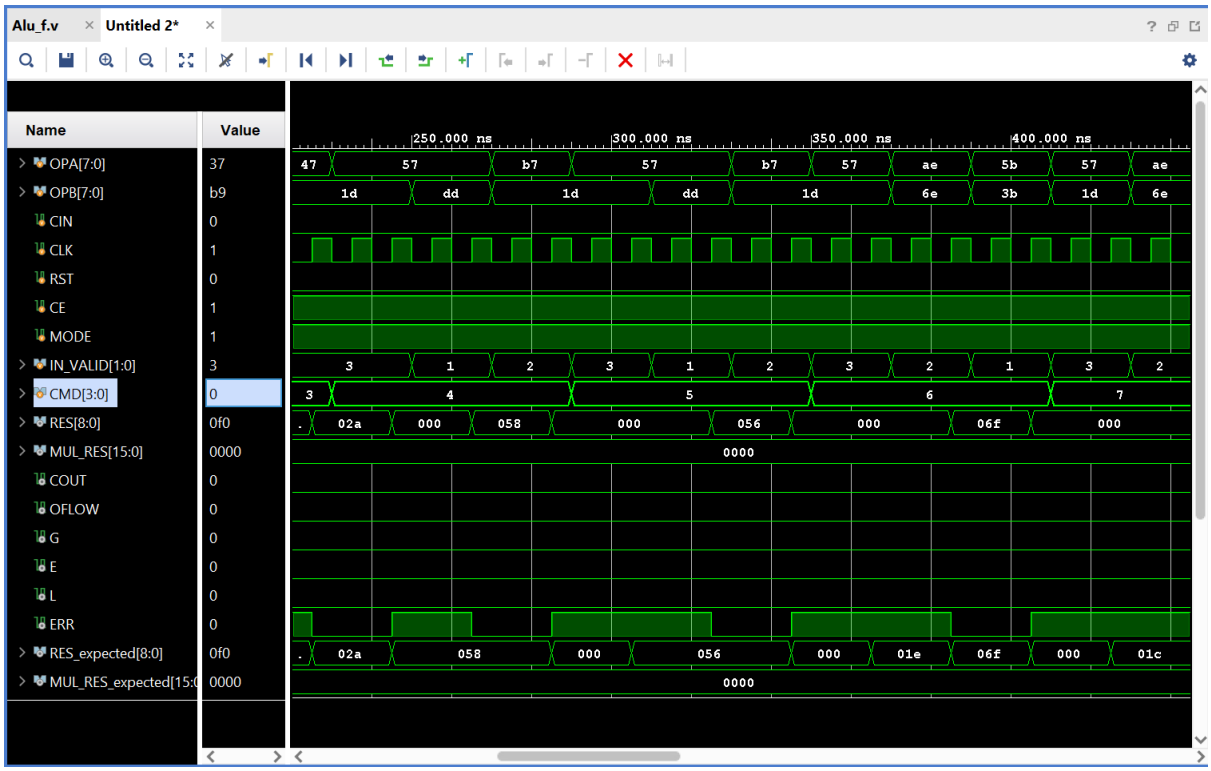


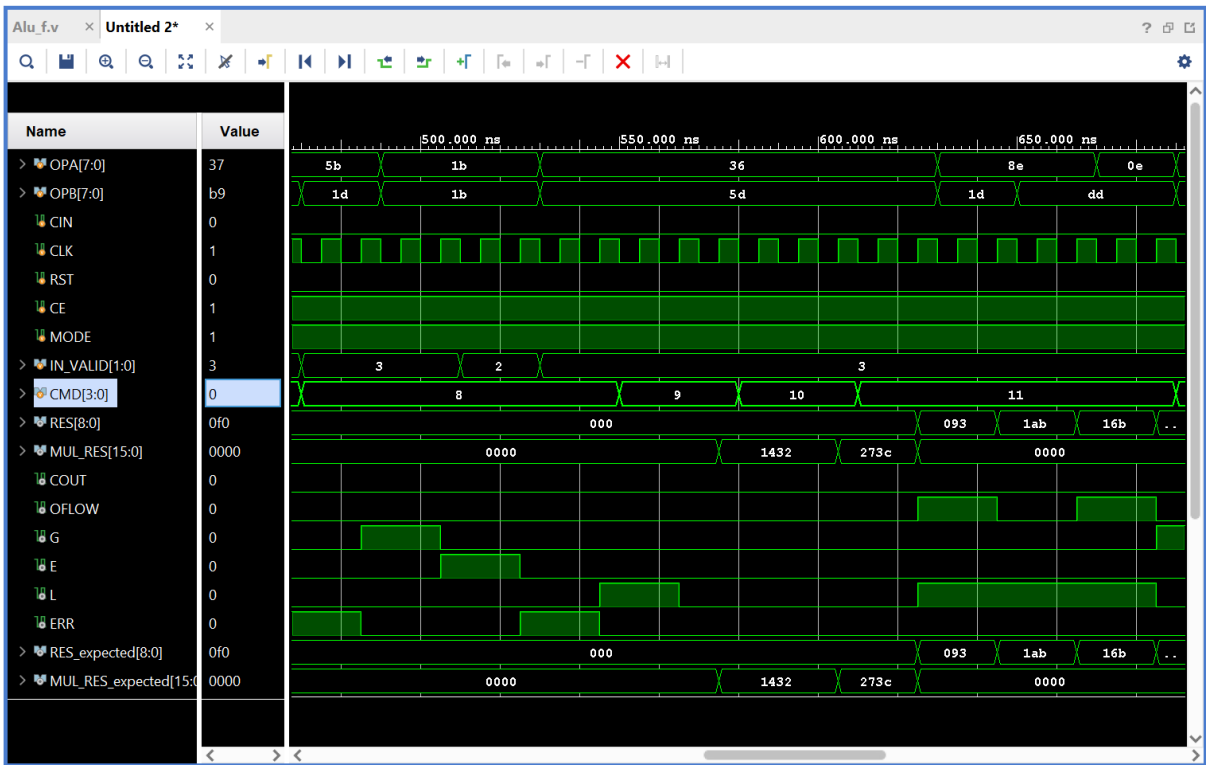
- The above timing diagram depicts the functionality of command “0” which is used for addition, when the “0” assets in the command on the negedge the operation starts, we get the values at the first posedge of the clock and on the second posedge we get the output in the RES.

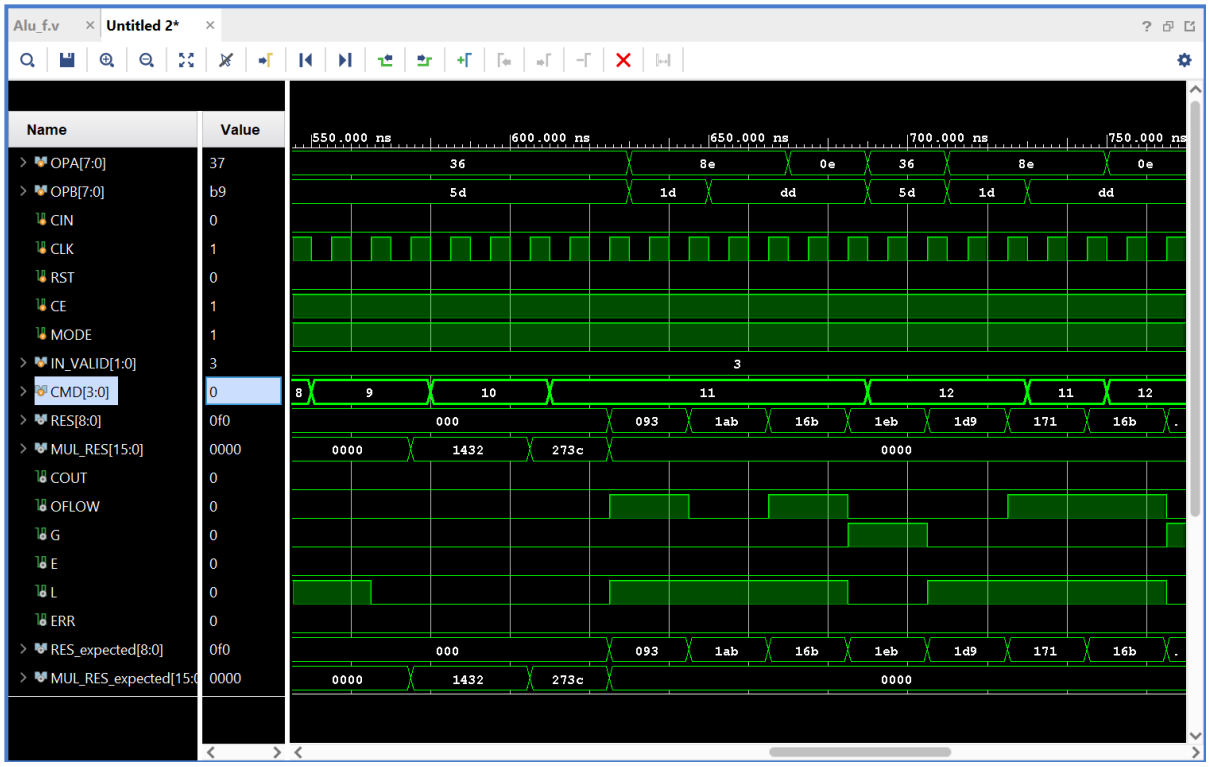


- In the above timing diagram we can observe that for the command “0” that is addition, when the IN_VALID is “01”,the operation won’t take place and hence getting the output as ”0” and the error is “1”. It will continue until either the CMD is changed or the IN_VALID is changed.

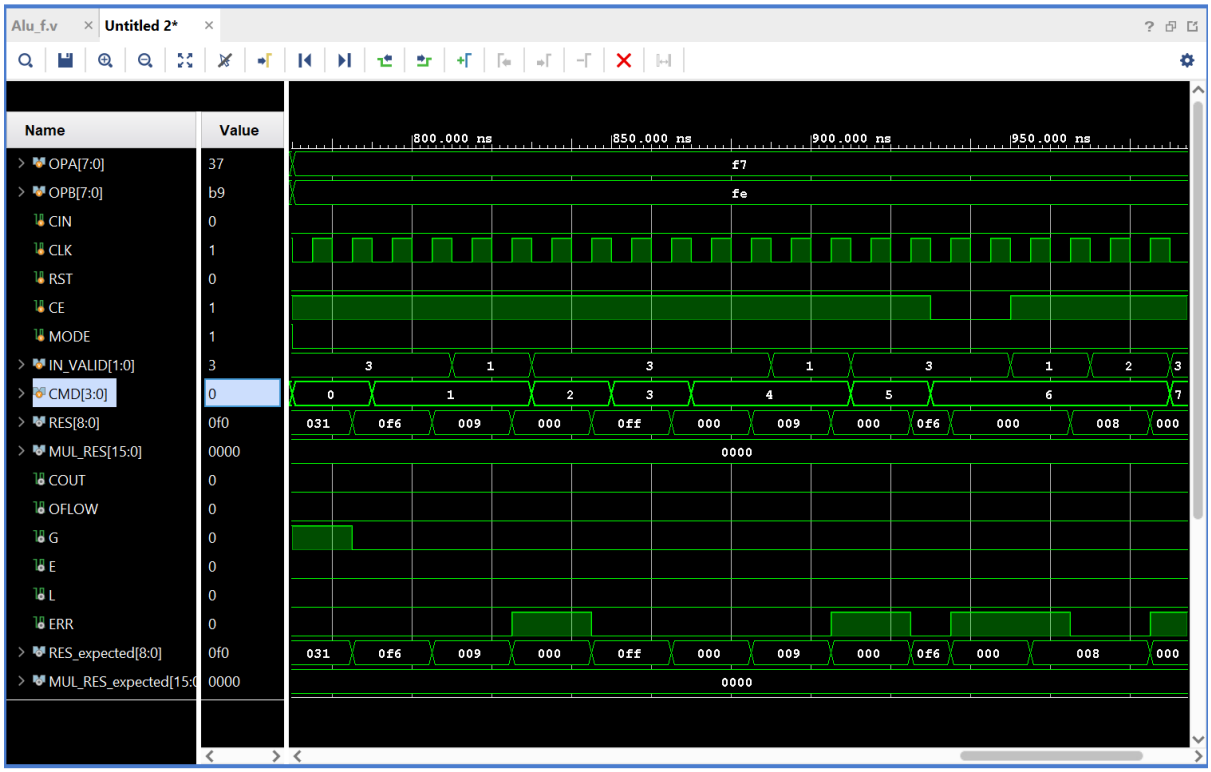
Output of few arithmetic operations are: -

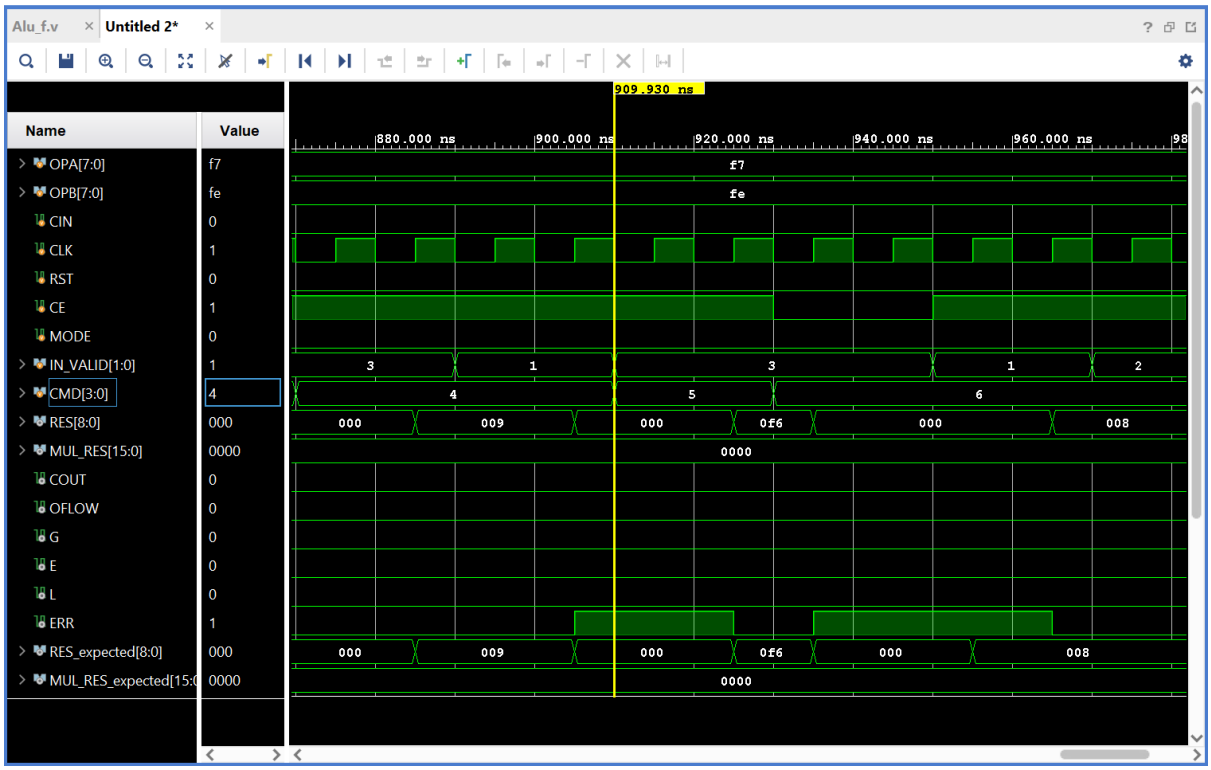






Output of few logical operations are: -





- Here for the above timing diagram, we have also tried making the CE (clock enable) as “0”, to test the DUT as well as the reference model and we can see that when the CE turns “0” in the next posedge the ERR flag as 1.

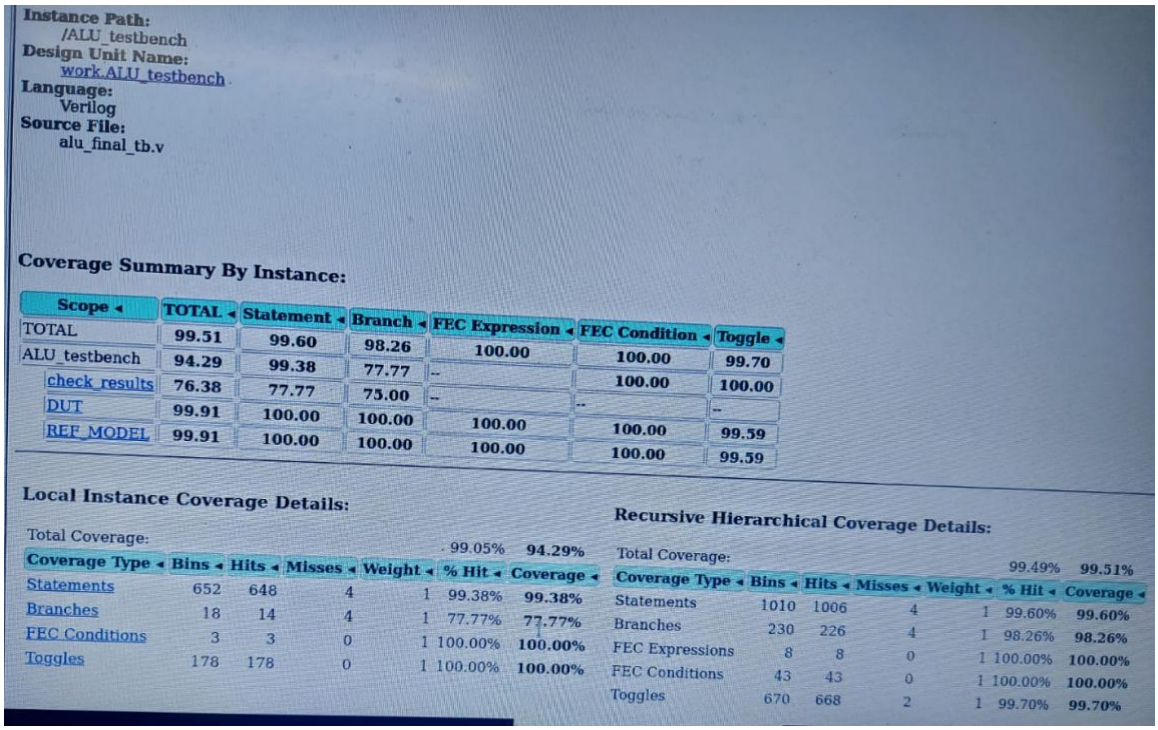
6. Challenges:

- Faced challenges in adjusting the number of clock cycles delay for the operations specially for multiply operations.
- Figuring out the size of the output was complex.
- Rotate right and rotate left was continuously giving error results, finding the logic that fits for all the possible widths was quite challenging.
- Adjusting the simulation time in the driver module to get the expected result.
- Comparing the result with expected result and printing “Pass” or “Fail” was giving “Fail” every time, as it was giving the present operation result in the next operation’s result.

7. Conclusions: -

The designed ALU meets all specified functional and structural requirements. With temporary register staging, it efficiently manages single and multi-cycle operations. It handles a wide range of arithmetic and logical tasks and correctly generates status flags, making it suitable for embedded and processor systems. The modular, synthesizable nature of the Verilog code ensures easy integration and future extensibility. The testbench framework provides robust coverage and validation, ensuring reliability under diverse conditions.

Code Coverage: -



Instance Path: /ALU_testbench
Design Unit Name: work.ALU_testbench
Language: Verilog
Source File: alu_final_tb.v

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle
TOTAL	99.51	99.60	98.26	100.00	100.00	99.70
ALU_testbench	94.29	99.38	77.77	--	100.00	100.00
check_results	76.38	77.77	75.00	--	--	--
DUT	99.91	100.00	100.00	100.00	100.00	99.59
REF_MODEL	99.91	100.00	100.00	100.00	100.00	99.59

Local Instance Coverage Details:

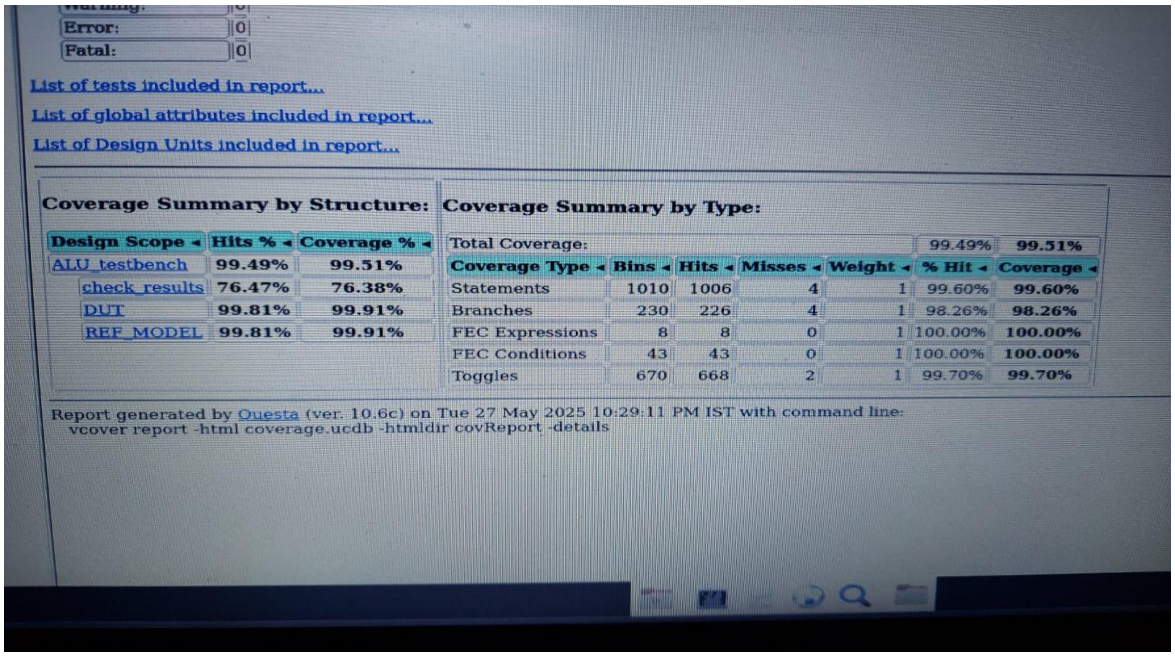
Total Coverage: 99.05% 94.29%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	652	648	4	1	99.38%	99.38%
Branches	18	14	4	1	77.77%	77.77%
FEC Conditions	3	3	0	1	100.00%	100.00%
Toggles	178	178	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage: 99.49% 99.51%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	1010	1006	4	1	99.60%	99.60%
Branches	230	226	4	1	98.26%	98.26%
FEC Expressions	8	8	0	1	100.00%	100.00%
FEC Conditions	43	43	0	1	100.00%	100.00%
Toggles	670	668	2	1	99.70%	99.70%



Error: 0
Fatal: 0

List of tests included in report...
List of global attributes included in report...
List of Design Units included in report...

Coverage Summary by Structure:

Design Scope	Hits %	Coverage %
ALU_testbench	99.49%	99.51%
check_results	76.47%	76.38%
DUT	99.81%	99.91%
REF_MODEL	99.81%	99.91%

Coverage Summary by Type:

Total Coverage:					99.49%	99.51%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	1010	1006	4	1	99.60%	99.60%
Branches	230	226	4	1	98.26%	98.26%
FEC Expressions	8	8	0	1	100.00%	100.00%
FEC Conditions	43	43	0	1	100.00%	100.00%
Toggles	670	668	2	1	99.70%	99.70%

Report generated by Questa (ver. 10.6c) on Tue 27 May 2025 10:29:11 PM IST with command line:
vcover report -html coverage.ucdb -htmldir covReport -details

8. Design Code & Self Checking Testbench code

https://github.com/vamsi1004/Dynamic-ALU-Project.git