ALU (ARITHMATIC AND LOGIC UNIT)
MINI PROJECT REPORT

Submitted By:

Name: Rakshith S Shetty

Emp.ID: 6100

Date: 08/06/2025

ALU PROJECT

Introduction

This project comprises the design as well as the implementation of an Arithmetic Logic Unit by means of Verilog HDL. The ALU is a fundamental combinational circuit that carries out arithmetic and logical operations over binary data. Due to its modular and parameterizable design, it can perform operations such as add, subtract, bitwise operations (AND, OR, XOR, etc.), Shifts, and combinations, including more advanced functions like signed arithmetic and multiplication with latency management.

Most of these are purely combinational operations, while clocked logic is incorporated within the design only to allow pipelining and multi-cycle operations (such as multiplication); hence, it can work in synchronous digital systems. Inputs into the ALU include commands and control signals that place it in a select mode of operation, with outputs produced comprising result, carry out, overflow, and condition flags (greater than, equal to, less than).

The Verilog-based ALU hereby designed represents an essential building block in processor as well as embedded system architectures for a strong and flexible solution to execute a core computational operation efficiently.

Objectives

- To design and implement an ALU in Verilog HDL, capable of performing many arithmetic and logic operations.
- To verify that the ALU functions correctly, and according to specification, by fully simulating it through Verilog testbenches of all supported operations.
- To provide both combinational and multi-cycle functions such as multiplication with latency through appropriate control integration.
- To support signed and unsigned operations properly, as well as produce status flags correctly, i.e., carry-out, overflow, and comparisons (greater, equal, less).
- The goal was to utilize pipelining techniques for enhanced performance in multi-cycle operations, to dovetail with clocked digital systems, and to improve throughput in processor-oriented applications.
- To parameterize data width and decode commands so that the ALU can be scaled up/down or converted for use in different designs.
- To establish and prove that the ALU performs well in various input situations from normal to edge cases and even invalid inputs, confirming the design's reliability and fault tolerance.

Architecture

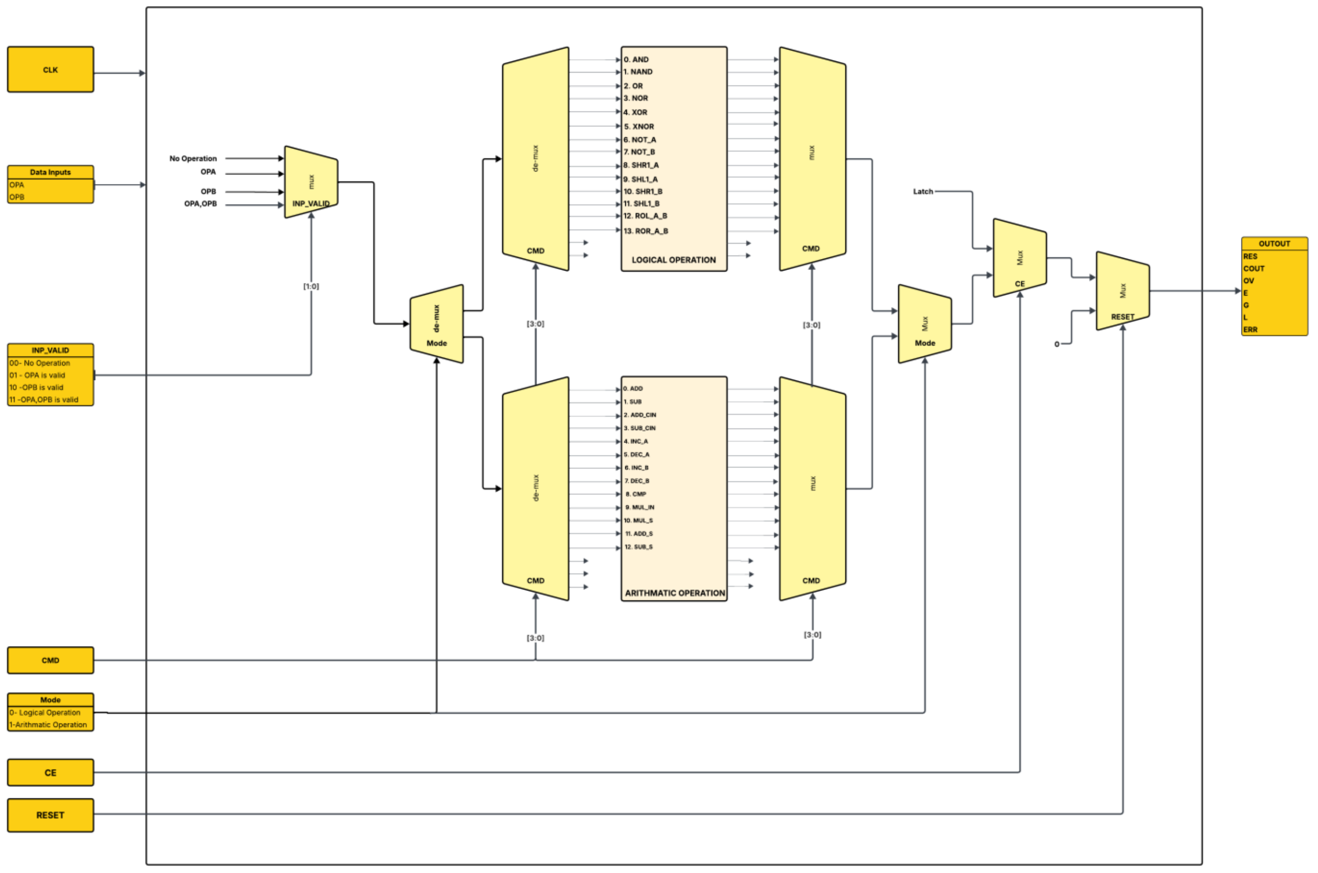


Figure 1: ALU architecture

ALU architecture is a parameterized modular design developed using Verilog HDL. It performs various arithmetic and logical operations on data depending on control signals set externally. It uses input validation, mode selection, and command decoding to route input operand data dynamically through the designated arithmetic or logic units of operation. Inclusion of multiplexers and de-multiplexers in the architecture makes data flow flexible among operation units and enables efficient selection of operation. Clock-enabled output controls and reset options have been incorporated in this architecture for ease of use in synchronous digital systems. The design has flexibility coupled with clarity and hence is an ideal choice for scalable and reusable ALU implementations.

ALU Design Architecture

- **CLK:**

  It is the main clock that drives the ALU. All the internal operations are synchronized with the rising edge of the clock signal.

❖ **Inputs:**

- **OPA, OPB:**

  Two operand inputs with a configurable bit-width (WIDTH). Used for all arithmetic and logical operations.

- **INP_VALID (2 bit):**

  Shows which inputs are valid:

  00: No operation (ALU idle)

  01: Only OPB is valid

  10: Only OPA is valid

  11: Both OPA and OPB are valid (normal operation)

- **CMD (4-bit):**

  Specifies the operation to be performed, e.g., ADD, SUB, XOR, etc.

- **Mode (1 bit):**

  Selects type of operation:

  0 - Logical operations

  1 - Arithmetic operations

- **CE (Clock Enable):**

  If high, output will be updated; else output will be latched.

- **RESET:**

  Clear internal state, output register, and flags.

❖ **Internal Control Logic:**

- **Demux (CMD demux):**

  This logic decodes the 4-bit CMD to specific control signals for individual operations in the arithmetic and logical blocks.

- **Mux (Mode selector):**

  Based on the Mode bit, selects among the outputs from the logical and arithmetic operation units.

- **INP_VALID decoder & muxes:**

  Checks operand inputs and only enables correct data paths. Muxes on the operand input path control which operand (OPA or OPB or both) are passed to operation units depending on INP_VALID.

- **Output Mux:**

  Final multiplexer which chooses between the logical or arithmetic result before being latched depending on Mode.

- **Latch Control:**

  Driven by CE, ensures output (RES and flags) gets updated only when CE is high.

ALU Testbench Architecture

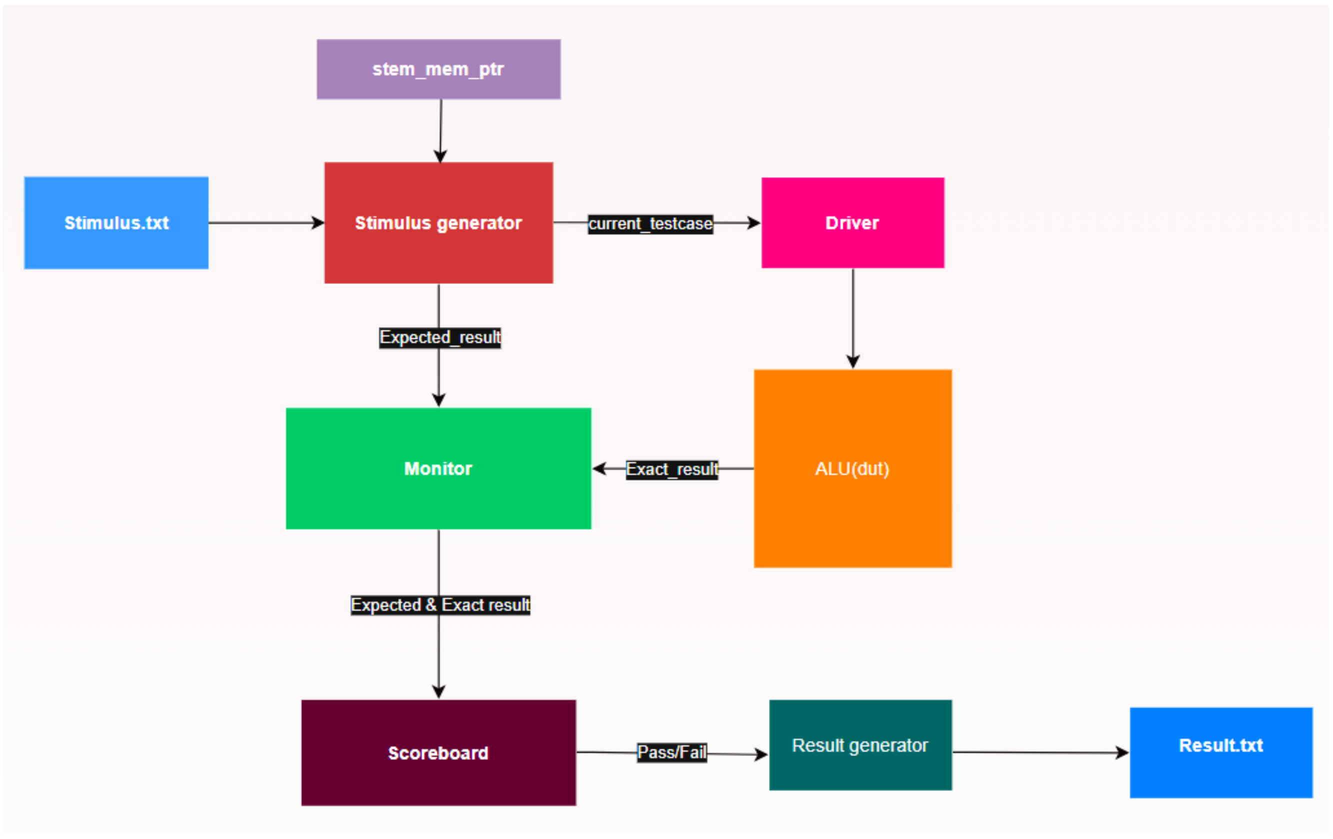


Figure 2: ALU Testbench Architecture

The ALU testbench architecture is designed to verify the functionality of the Arithmetic Logic Unit. It automates input generation, signal driving, and output validation. Firstly, a stimulus file contains a predetermined set of test cases, which are then applied to the ALU through stimuli generator and driver components. Input and corresponding outputs are processed by the ALU, and the outputs are observed.

These outputs are then compared against expected outcomes using a scoreboard, with result logs available for analysis. Due to its modular construction, it allows for the methodical and efficient testing of ALU operations under numerous kinds of inputs.

1. Stimulus Generator

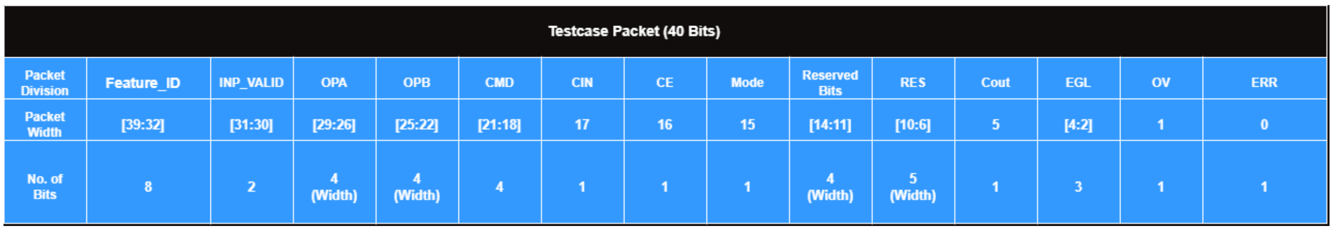
- Extracts fields from each 40-bit packet from Stimulus.txt.
- Sends control/data inputs to driver and expected outputs to monitor.

2. stem_mem_ptr

- Pointer interface against stimulus memory.
- Feeds the stimulus one line of test case at a time into the Stimulus Generator.

3. Stimulus.txt

- Encodes 40-bit testcases with fixed field format.
- Acts as input database for test execution.



Testcase Packet (40 Bits)														
Packet Division	Feature_ID	INP_VALID	OPA	OPB	CMD	CIN	CE	Mode	Reserved Bits	RES	Cout	EGL	OV	ERR
Packet Width	[39:32]	[31:30]	[29:26]	[25:22]	[21:18]	17	16	15	[14:11]	[10:6]	5	[4:2]	1	0
No. of Bits	8	2	4 (Width)	4 (Width)	4	1	1	1	4 (Width)	5 (Width)	1	3	1	1

Figure 3: Testcase Packet (40 bits)

4. Driver

- Drives OPA, OPB, CMD, CIN, CE, and Mode into the ALU.
- Translates test case data into ALU pin-level inputs.

5. ALU (DUT)

- Carries out the operation depending on the command and operand inputs.
- Outputs results and flags: RES, Cout, EGL, OV, ERR.

6. Monitor

- Grabs the actual output of ALU and expected out.
- Forwards them to the Scoreboard for validation.

7. Scoreboard

- Compares actual versus expected output.
- Declares Pass or Fail for each test case.

8. Result Generator

- Formats the Pass and Fail results with test case ID.
- Logs them to output files.

9. Result.txt

- Stores the final test report with comparisons.
- Used as verification analysis and debugging.

Working

- **ALU Modes of Operation and Control Signals**

Arithmetic Logic Unit (ALU) is programmed to perform a large variety of arithmetic and logical operations depending upon the control inputs received. Two basic operands, OPA and OPB, are employed along with a 4-bit command signal (CMD) and a 1-bit mode signal (MODE) to decide precisely what operation needs to be carried out. When MODE is 1, the ALU goes into Arithmetic Mode, allowing operations such as addition, subtraction, increment, decrement, and both signed and unsigned multiplication. When MODE is 0, the ALU goes into Logical Mode, doing operations such as AND, OR, XOR, NAND, NOR, XNOR, NOT, and a lot of shifting and rotations.

- **Input Validity and Operand Selection**

Input validity is checked using the INP_VALID signal, a 2-bit control that determines which operand(s) are valid. When it is 2'b00, both operands are invalid and no operation is performed, setting the ERR flag. When it is 2'b01 or 2'b10, only one of the operands is valid and single-operand operations can be performed, and when it is 2'b11, both operands are valid and full binary operations are permitted.

- **Synchronous Operation and Result Timing**

Internally, the ALU performs all operations combinationally. The immediate result of any operation is temporarily stored in an intermediate result register and on the subsequent rising edge of the clock (CLK) moved to the output register (RES). This creates a 1-cycle latency for the majority of operations, providing synchronous.

- **Pipelined Multiplication Management**

For multiply operations, because they are complex and to replicate real-world delays, internal pipelining is applied in the architecture. Partial results of multiplication are temporarily stored in a register (mul_res). A 1-cycle delay is added using a single stage of registers, whereas a 2-cycle delay uses two registers to imitate a multi-stage pipeline. Through pipelining, the results of multiplication are glitch-free and stable prior to loading into the ultimate output (RES).

- **Output Width and Data Integrity**

RES output is parameterized as WIDTH + 1 bits to allow carry and overflow values from arithmetic operations. For the case of multiplication, this bit-width increase prevents data loss by making use of reserved bits in the RES register to store the extended result. This is a design choice that allows for safe and accurate representation of wider multiplication outputs.

- **Status Flags and Error Detection**

The ALU maintains a number of flags determined by the operations and operands. COUT shows the existence of a carry-out on unsigned addition. OVFLOW captures overflow situations that arise in signed addition and subtraction, as well as in unsigned subtraction. G, E, and L represent the conditions of comparison: one operand is larger, equal, or smaller than the other. In all situations when an illegal operation or operand setting is found, the ERR flag is set to signify the abnormality.

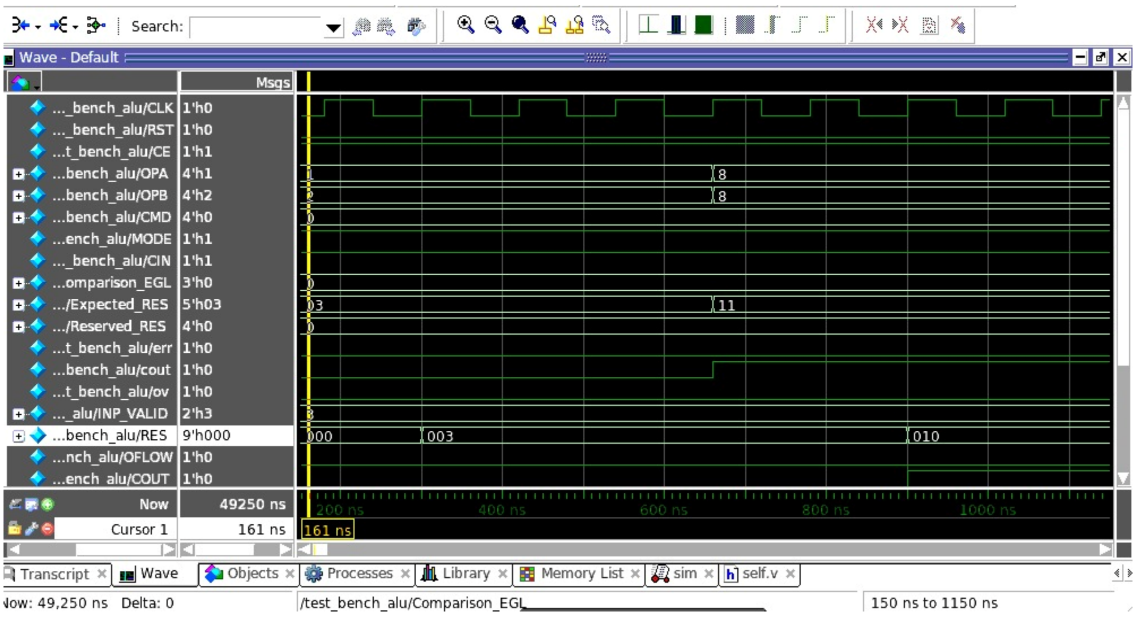
Results

The design of the Arithmetic Logic Unit (ALU) was comprehensively validated using simulation with a large testbench. The functional correctness was proven on a broad set of operations, such as arithmetic (add, subtract, increment, decrement), logical (AND, OR, XOR, NOT, etc.), and special functions like rotation and signed arithmetic. For every operation, various patterns of input were tested, including corner cases that cause overflow, underflow, or error conditions.

Waveform analysis during simulation verified the desired operation of the ALU. In particular, for pipelined operations like regular arithmetic, the result (res) is generated after one clock cycle, as desired. For delayed operations like the multiplication instruction (CMD 9), the ALU delivers the result after three clock cycles.

The verification testbench was solid and modular. It was structured to read input vectors from a formatted stimulus.txt file and verify the DUT (Design Under Test) outputs with expected values. Each test was assessed for pass/fail and written to a results file. The testbench automatically monitored not only output correctness, but also the state of flags like COUT, OFLOW, G, L, E, and ERR.

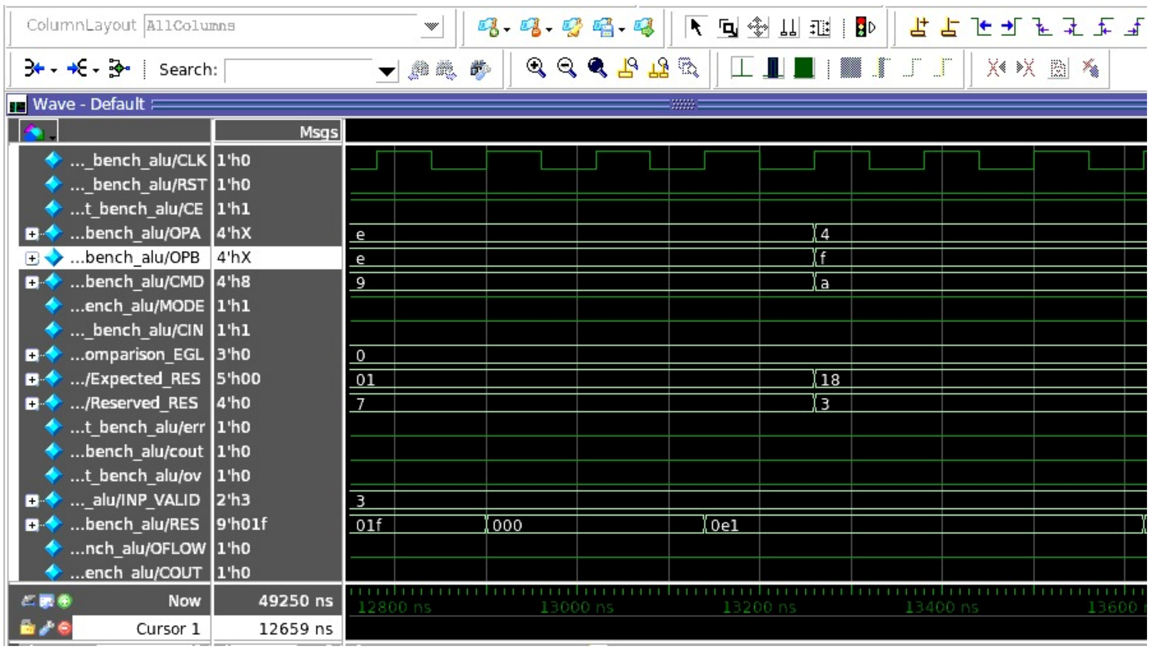
Special care was taken during testing for flag behavior. Overflow (OFLOW) and carry-out (COUT) flags were properly set in boundary arithmetic operations, like in adding two large unsigned numbers or signed operation close to the boundary of 4-bit representation. Comparison flags (G, L, E) were verified to accurately indicate the relational status of operands, and the ERR flag was found to be set correctly during erroneous operations (e.g., improper INP_VALID or unproper rotate counts).



```
Wave - Default                      Msgs
..._bench_alu/CLK       1'h0
..._bench_alu/RST       1'h0
...t_bench_alu/CE       1'h1
...bench_alu/OPA        4'h1
...bench_alu/OPB        4'h2
...bench_alu/CMD        4'h0
...ench_alu/MODE        1'h1
..._bench_alu/CIN       1'h1
...omparison_EGL        3'h0
.../Expected_RES        5'h03
.../Reserved_RES        4'h0
...t_bench_alu/err      1'h0
...bench_alu/cout       1'h0
...t_bench_alu/ov       1'h0
..._alu/INP_VALID       2'h3
...bench_alu/RES        9'h000
...nch_alu/OFLOW        1'h0
...ench_alu/COUT        1'h0
Now                     49250 ns
Cursor 1                161 ns
```

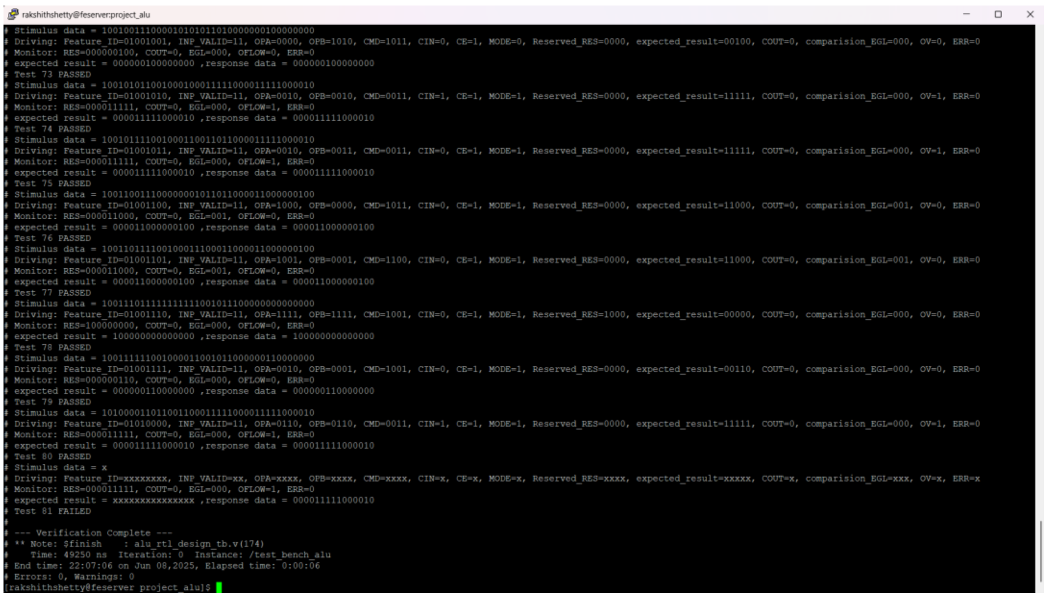
Now: 49,250 ns   Delta: 0   /test_bench_alu/Comparison_EGL   150 ns to 1150 ns

Figure 4: Output wavefrom



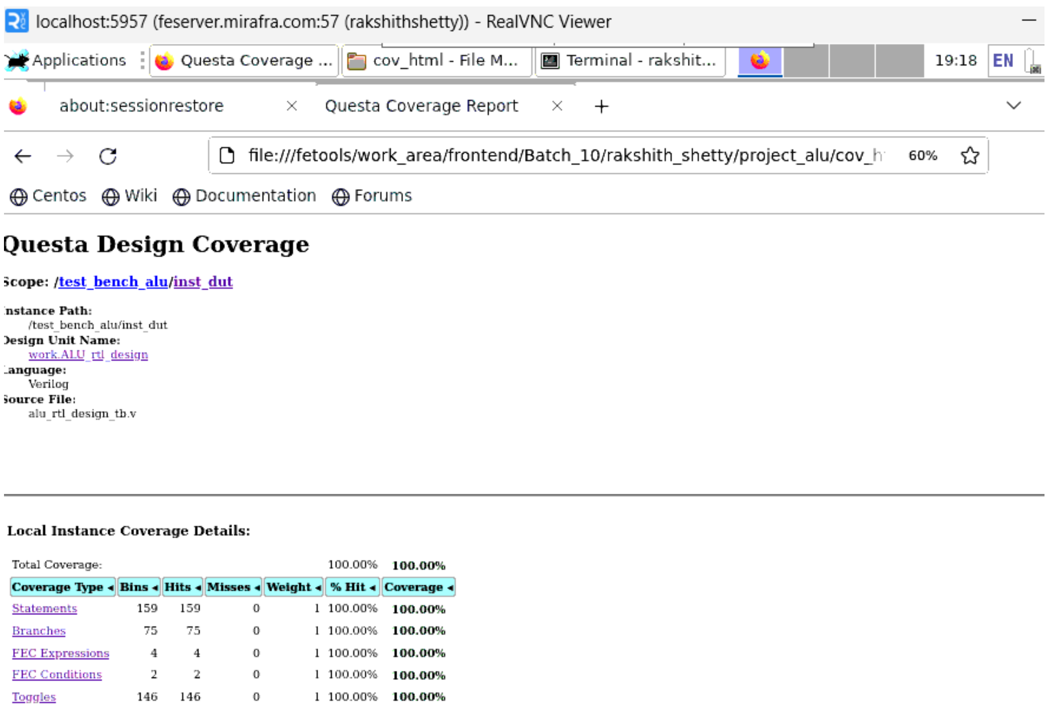
```
Wave - Default                      Msgs
..._bench_alu/CLK       1'h0
..._bench_alu/RST       1'h0
...t_bench_alu/CE       1'h1
...bench_alu/OPA        4'hX
...bench_alu/OPB        4'hX
...bench_alu/CMD        4'h8
...ench_alu/MODE        1'h1
..._bench_alu/CIN       1'h1
...omparison_EGL        3'h0
.../Expected_RES        5'h00
.../Reserved_RES        4'h0
...t_bench_alu/err      1'h0
...bench_alu/cout       1'h0
...t_bench_alu/ov       1'h0
..._alu/INP_VALID       2'h3
...bench_alu/RES        9'h01f
...nch_alu/OFLOW        1'h0
...ench_alu/COUT        1'h0
Now                     49250 ns
Cursor 1                12659 ns
```

Figure 5: Output wavefrom for MUL_INoperation



```
# Stimulus data = 10010011100001010101101000000001000000000
# Driving: Feature_ID=01001001, INP_VALID=11, OPA=0000, OPB=1010, CMD=1011, CIN=0, CE=1, MODE=0, Reserved_RES=0000, expected_result=00100, COUT=0, comparision_EGL=000, OV=0, ERR=0
# Monitor: RES=000000100, COUT=0, EGL=000, OFLOW=0, ERR=0
# expected result = 000000100000000 ,response data = 000000100000000
# Test 73 PASSED
# Stimulus data = 10010101100100010001111100001111000010
# Driving: Feature_ID=01001010, INP_VALID=11, OPA=0010, OPB=0010, CMD=0011, CIN=1, CE=1, MODE=1, Reserved_RES=0000, expected_result=11111, COUT=0, comparision_EGL=000, OV=1, ERR=0
# Monitor: RES=000011111, COUT=0, EGL=000, OFLOW=1, ERR=0
# expected result = 000011111000010 ,response data = 000011111000010
# Test 74 PASSED
# Stimulus data = 10010111100100011001101100001111000010
# Driving: Feature_ID=01001011, INP_VALID=11, OPA=0010, OPB=0011, CMD=0011, CIN=0, CE=1, MODE=1, Reserved_RES=0000, expected_result=11111, COUT=0, comparision_EGL=000, OV=1, ERR=0
# Monitor: RES=000011111, COUT=0, EGL=000, OFLOW=1, ERR=0
# expected result = 000011111000010 ,response data = 000011111000010
# Test 75 PASSED
# Stimulus data = 10011001110000000101101100001100000000100
# Driving: Feature_ID=01001100, INP_VALID=11, OPA=1000, OPB=0000, CMD=1011, CIN=0, CE=1, MODE=1, Reserved_RES=0000, expected_result=11000, COUT=0, comparision_EGL=001, OV=0, ERR=0
# Monitor: RES=000011000, COUT=0, EGL=001, OFLOW=0, ERR=0
# expected result = 000011000000100 ,response data = 000011000000100
# Test 76 PASSED
# Stimulus data = 10011011110010001110001100001100000000100
# Driving: Feature_ID=01001101, INP_VALID=11, OPA=1001, OPB=0001, CMD=1100, CIN=0, CE=1, MODE=1, Reserved_RES=0000, expected_result=11000, COUT=0, comparision_EGL=001, OV=0, ERR=0
# Monitor: RES=000011000, COUT=0, EGL=001, OFLOW=0, ERR=0
# expected result = 000011000000100 ,response data = 000011000000100
# Test 77 PASSED
# Stimulus data = 10011101111111111100101110000000000000000
# Driving: Feature_ID=01001110, INP_VALID=11, OPA=1111, OPB=1111, CMD=1001, CIN=0, CE=1, MODE=1, Reserved_RES=1000, expected_result=00000, COUT=0, comparision_EGL=000, OV=0, ERR=0
# Monitor: RES=100000000, COUT=0, EGL=000, OFLOW=0, ERR=0
# expected result = 100000000000000 ,response data = 100000000000000
# Test 78 PASSED
# Stimulus data = 10011111100100001100101100000011000000000
# Driving: Feature_ID=01001111, INP_VALID=11, OPA=0010, OPB=0001, CMD=1001, CIN=0, CE=1, MODE=1, Reserved_RES=0000, expected_result=00110, COUT=0, comparision_EGL=000, OV=0, ERR=0
# Monitor: RES=000000110, COUT=0, EGL=000, OFLOW=0, ERR=0
# expected result = 000000110000000 ,response data = 000000110000000
# Test 79 PASSED
# Stimulus data = 10100001101100110001111100001111000010
# Driving: Feature_ID=01010000, INP_VALID=11, OPA=0110, OPB=0110, CMD=0011, CIN=1, CE=1, MODE=1, Reserved_RES=0000, expected_result=11111, COUT=0, comparision_EGL=000, OV=1, ERR=0
# Monitor: RES=000011111, COUT=0, EGL=000, OFLOW=1, ERR=0
# expected result = 000011111000010 ,response data = 000011111000010
# Test 80 PASSED
# Stimulus data = x
# Driving: Feature_ID=xxxxxxxx, INP_VALID=xx, OPA=xxxx, OPB=xxxx, CMD=xxxx, CIN=x, CE=x, MODE=x, Reserved_RES=xxxx, expected_result=xxxxx, COUT=x, comparision_EGL=xxx, OV=x, ERR=x
# Monitor: RES=000011111, COUT=0, EGL=000, OFLOW=1, ERR=0
# expected result = xxxxxxxxxxxxxxx ,response data = 000011111000010
# Test 81 FAILED
#
# --- Verification Complete ---
# ** Note: $finish    : alu_rtl_design_tb.v(174)
#    Time: 49250 ns  Iteration: 0  Instance: /test_bench_alu
# End time: 22:07:06 on Jun 08,2025, Elapsed time: 0:00:06
# Errors: 0, Warnings: 0
[rakshithshetty@feserver project_alu]$
```

Figure 6: Output from self_testbench



Questa Design Coverage

Scope: /test_bench_alu/inst_dut

Instance Path:
/test_bench_alu/inst_dut
Design Unit Name:
work.ALU_rtl_design
Language:
Verilog
Source File:
alu_rtl_design_tb.v

Local Instance Coverage Details:

Total Coverage: 100.00% **100.00%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	159	159	0	1	100.00%	**100.00%**
Branches	75	75	0	1	100.00%	**100.00%**
FEC Expressions	4	4	0	1	100.00%	**100.00%**
FEC Conditions	2	2	0	1	100.00%	**100.00%**
Toggles	146	146	0	1	100.00%	**100.00%**

Figure 7: Functional coverage report

Code coverage was monitored with coverage tools, and a score of around 100% was early on attained, mostly covering valid operations. Additional corner case stimulus vectors—like signed overflow, rotate invalidity, and mismatched flag combinations—were included, and the better coverage resulted.

Additionally, the ALU was synthesizable and parameterized, with it being scalable and reusable. The op width and cmd width parameters enable designers to easily change the operand size (e.g., from 8 bits to 16 or 32 bits) and accommodate future command extensions without changing the basic logic.

Synthesizability was guaranteed by excluding non-synthesizable constructs and having clean reset conditions and clocked processes.

The design is modular, clear, and hardy—attributes that are crucial to digital system design. The simulation results and waveform proof confirm that the ALU functions correctly for all tested conditions and is prime for FPGA implementation or further development.

Conclusion

The ALU based on Verilog successfully meets all functional specifications defined, providing uniform and reliable performance over a wide variety of arithmetic and logical functions. Modular design encourages straightforward decoupling of functionalities, making debugging easier, facilitating greater scalability, and allowing code reuse. Pipelining is achieved by using internal registers, which maintain proper timing for multi-cycle operations such as multiplication, while maintaining single-cycle execution for combinational instructions alone.

Simulation outcome confirms the accuracy of output values as well as correct status flag updates under different test cases, affirming the strength of the design. Further, the ALU code is completely synthesizable and hardware-efficient optimized, and thus suitable for inclusion in highly complex digital systems.

Future Improvement

To build on the existing ALU design, the following improvements are suggested:

- **Multi-Stage Pipelining:** Apply sophisticated pipelined stages to enhance instruction level and enable concurrent execution of multiple operations.
- **Extended Operation Set:** Add more sophisticated arithmetic operations like division, modulus, and barrel shifting to make the ALU more versatile.
- **Scalable Data Widths:** Make the design flexible enough to accommodate configurable data widths (e.g., 16-bit, 32-bit) to facilitate higher-precision calculations and wider usage.
- **Formal Verification:** Utilize formal verification methods to strictly confirm the correctness of the ALU for all input combinations, with greater reliability than from simulation alone.
- **Optimized Multiplication Unit:** Substituting the existing register-based delay approach with a dedicated pipelined multiplier for more efficient multi-cycle multiplication.