ALU PROJECT

Introduction

This project focuses on building an Arithmetic Logic Unit (ALU) using Verilog HDL, covering its architecture and operational behavior. The ALU is a fundamental combinational module used to perform a broad set of arithmetic and logical operations on binary inputs. Developed as a modular and parameterized unit, the ALU supports operations such as addition, subtraction, bitwise logic (AND, OR, XOR, etc.), shifting, comparison, and extended functionality like signed arithmetic and multiplication with pipelined latency control.

Although the majority of operations are combinational, the design incorporates sequential logic to support multi-cycle operations (e.g., multiplication) by aligning results with the clock cycle requirements of synchronous systems. The ALU receives input operands, control signals, and operation codes, and generates outputs including the computation result, carry-out, overflow flag, and comparison flags (greater, equal, less).

This Verilog-based ALU plays an essential role in digital processor and embedded system design, providing a configurable and efficient computational core for various hardware applications.

Objectives

➢ To architect and implement a versatile ALU in Verilog HDL that performs a diverse set of arithmetic and logical operations.

➢ To incorporate both combinational and clocked control logic to support single-cycle and multi-cycle operations such as multiplication.

➢ To enable design scalability through parameterized data widths and flexible command decoding mechanisms.

➢ To ensure correct handling of signed and unsigned computations, along with accurate generation of carry, overflow, and comparison flags.

➢ To validate the ALU's functional accuracy using detailed simulation testbenches within a Verilog-based verification environment.

➢ To evaluate the ALU's performance under various operating scenarios, including boundary conditions and invalid input cases.

Architecture

ALU Design Architecture

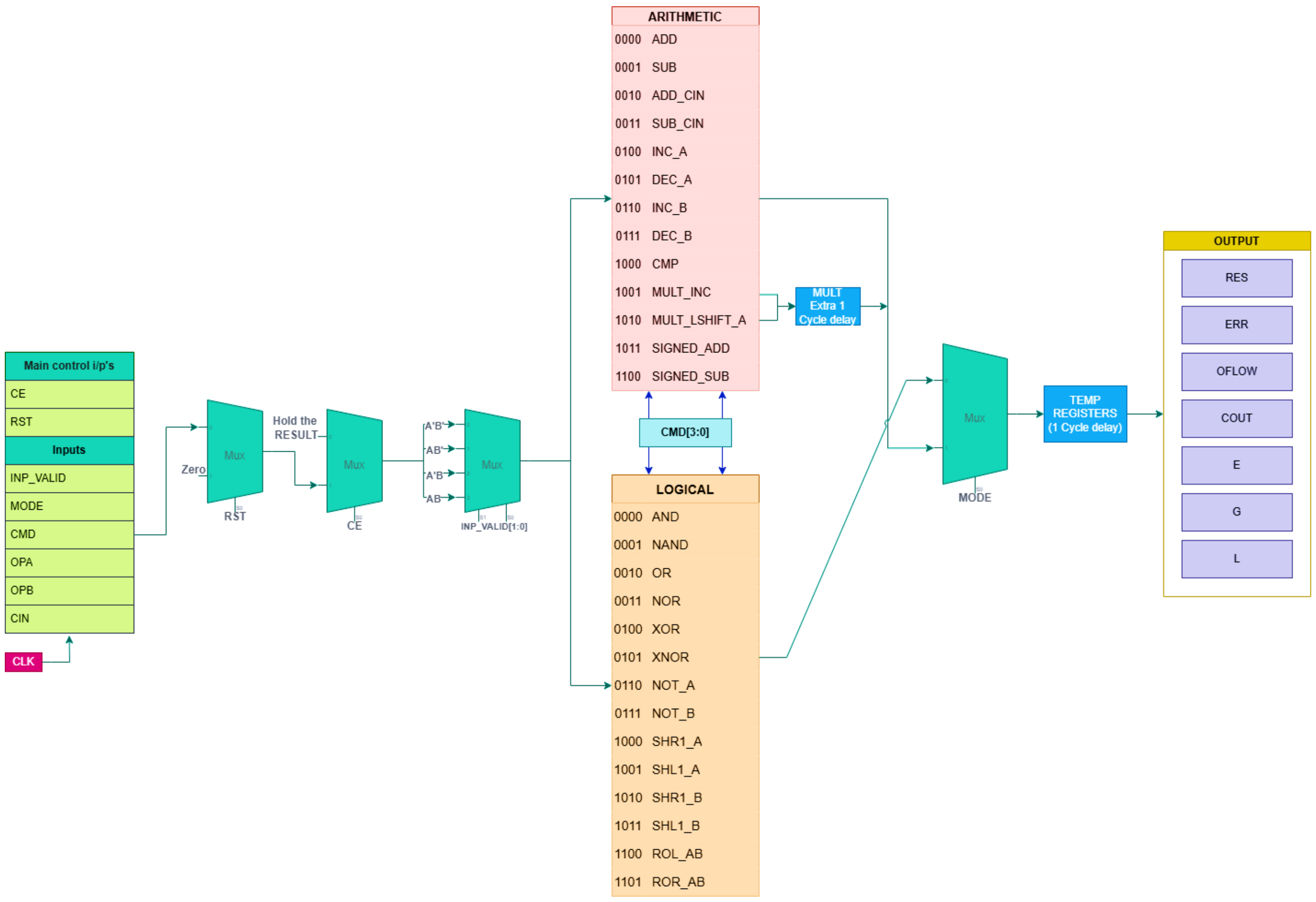


Fig 3.1 ALU Design Architecture

The ALU design in fig 3.1 integrates arithmetic and logical operations controlled by input command signals. Based on the MODE and CMD, the appropriate block (arithmetic or logical) is selected via multiplexers. It supports multi-cycle operations like multiplication and uses temporary registers to manage output delays. The architecture ensures modularity and timing control for accurate and efficient computation.

A more detailed explanation of the internal signal flow and operation selection is provided in the Working section.

Pin description for Fig 3.1

Inputs

- **OPA** & **OPB** are parameterized operands.
- **CIN** is 1-bit active high carry input signal.
- **CLK** is the clock to the design and it is edge sensitive.
- **RST** is the active high Asynchronous reset.
- **CE** is 1-bit active high clock enable signal.
- **MODE** is 1-bit operation control signal. Details specified in working part.
- **INP_VALID** controls the operand validity. Details specified in working part.
- **CMD** is a parameterized (4 bits) Command controller.

Outputs

- **RES** is the total (2 x parameterized) bit result.
- **OFLOW** indicates an output overflow during signed and unsigned add/sub.
- **COUT** indicates an output overflow during unsigned add/sub.
- **G, L and E** are the comparator outputs of 1-bit, which indicates that the value of OPA is greater than less than and equal to the value of OPB respectively.
- **ERR** is a 1-bit error signal which goes high whenever as specified in the working part.

ALU Testbench Architecture

This testbench architecture is designed to verify the functionality of an Arithmetic Logic Unit (ALU). It uses a modular approach where stimulus is read from a predefined test vector file, applied to the ALU, and the output is compared against expected results. The goal is to ensure the ALU behaves correctly for all possible inputs and edge cases

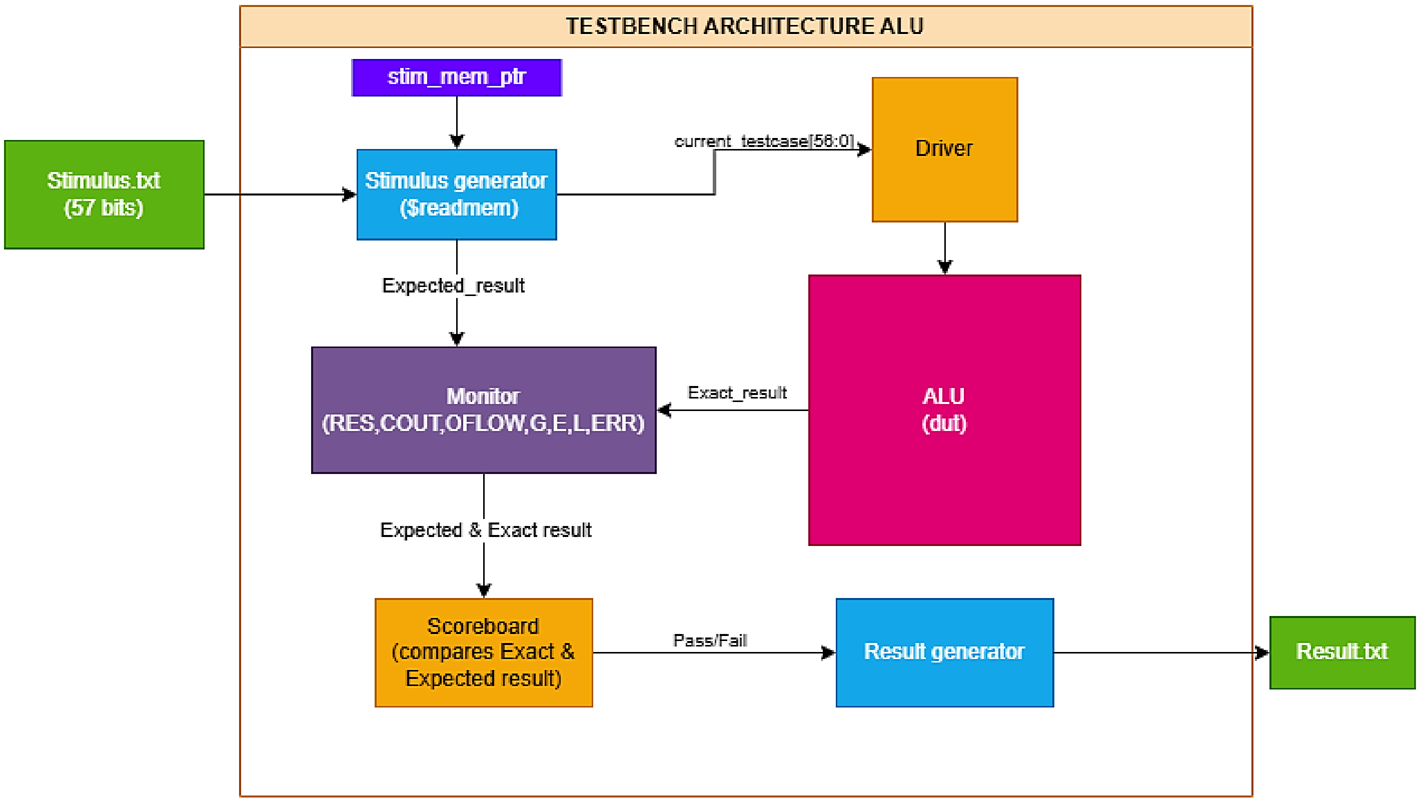


Fig 3.2 ALU Testbench Architecture

Stimulus.txt: Contains pre-generated 57-bit test cases including inputs and expected outputs.

Stimulus Generator: Reads test cases from the file and provides both the current input and the expected result.

Driver: Sends decoded input signals (operands, command) to the ALU.

ALU (DUT): The actual unit being tested. It computes the result and flags.

Monitor: Collects both expected and actual outputs for each test case.

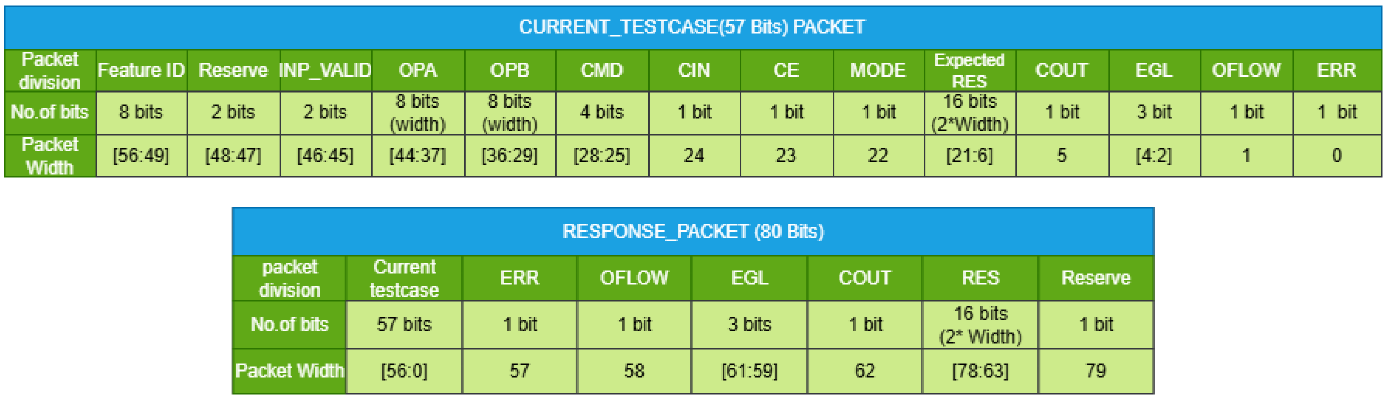
Scoreboard: Compares expected vs actual results and marks Pass/Fail.

Result Generator: Logs the outcomes into a human-readable result file.

Result.txt: Output file listing all test case results for final verification.

Test Packets

To verify the ALU functionality, each test case is packed into a structured 57-bit input packet and an 80-bit output response packet (this is for 8 bit operands). These packets encapsulate all necessary control signals, operands, and expected outcomes required for accurate testing. The stimulus file feeds these input packets into the testbench, and the monitor compares the ALU's output response against the expected values embedded in the packet structure. This approach ensures scalable, automated, and repeatable testing of the ALU across a wide range of functional scenarios.



CURRENT_TESTCASE(57 Bits) PACKET														
Packet division	Feature ID	Reserve	INP_VALID	OPA	OPB	CMD	CIN	CE	MODE	Expected RES	COUT	EGL	OFLOW	ERR
No.of bits	8 bits	2 bits	2 bits	8 bits (width)	8 bits (width)	4 bits	1 bit	1 bit	1 bit	16 bits (2*Width)	1 bit	3 bit	1 bit	1 bit
Packet Width	[56:49]	[48:47]	[46:45]	[44:37]	[36:29]	[28:25]	24	23	22	[21:6]	5	[4:2]	1	0

RESPONSE_PACKET (80 Bits)							
packet division	Current testcase	ERR	OFLOW	EGL	COUT	RES	Reserve
No.of bits	57 bits	1 bit	1 bit	3 bits	1 bit	16 bits (2* Width)	1 bit
Packet Width	[56:0]	57	58	[61:59]	62	[78:63]	79

Fig 3.3 Packet Distribution

Working

The ALU operates by interpreting control signals (command and mode) to select and perform the requested arithmetic or logical operation on the input operands. Depending on the operation type, the ALU either processes the inputs combinationally in a single clock cycle or uses internal registers to handle multi-cycle instructions, such as multiplication. Input validity signals ensure that only valid operand data triggers computations, while status flags provide real-time feedback on the result conditions like carry, overflow, and comparisons. This coordinated approach enables flexible and reliable ALU operation across a broad range of instructions.

Mode Control (MODE Signal):

- When MODE = 1, the ALU activates the arithmetic operation set, including signed/unsigned math and complex instructions like multiply-after-increment or shift-and-multiply.
- When MODE = 0, the ALU switches to logical operations such as AND, OR, NOT, XOR, and both logical shifts and rotational shifts.

Operand Validation (INP_VALID Signal):

- 2’b00: Both operands are invalid – operation is skipped, and an error flag is triggered.
- 2’b01 or 2’b10: One operand is valid – used for single-input operations like NOT or increment.
- 2’b11: Both operands are valid – required for two-operand operations.

Combinational vs Sequential Execution:

- Most operations (ADD, SUB, CMP, AND, OR, etc.) are computed purely combinationally.
- Results are latched to the output on the next clock cycle.
- For specific commands like multiplication (MULT_INC, MULT_SHIFT), intermediate results are stored in internal registers and forwarded to the output after one or two clock cycles, simulating pipeline behavior.

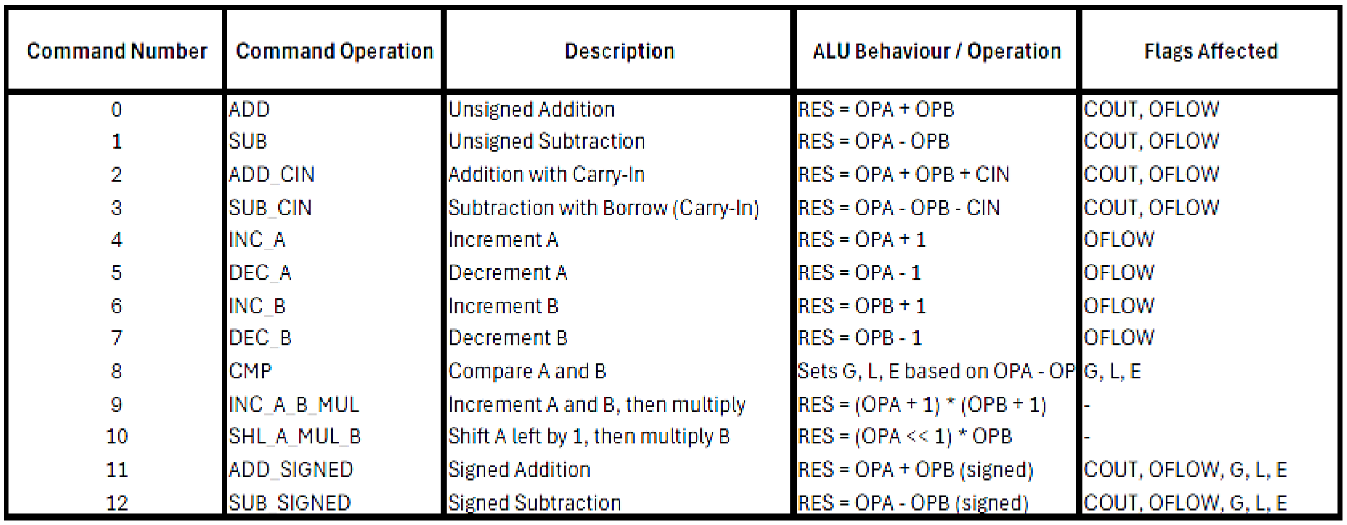
Status Flag Generation:

- COUT: Reflects carry-out from unsigned addition/subtraction.

- OFLOW: Indicates signed overflow.
- G, E, L: Signal greater-than, equal-to, and less-than conditions respectively during comparisons.
- ERR: Raised when an invalid command or operand condition is detected (e.g., illegal rotate shift) and when Cmd is selected as 12 or 13 and mode is logical operation , if 4th ,5th ,6th and 7th bit of OPB are 1.

Arithmetic Operations:

Arithmetic operations perform mathematical calculations like addition, subtraction, shifting, and rotation on binary inputs to manipulate numerical values.



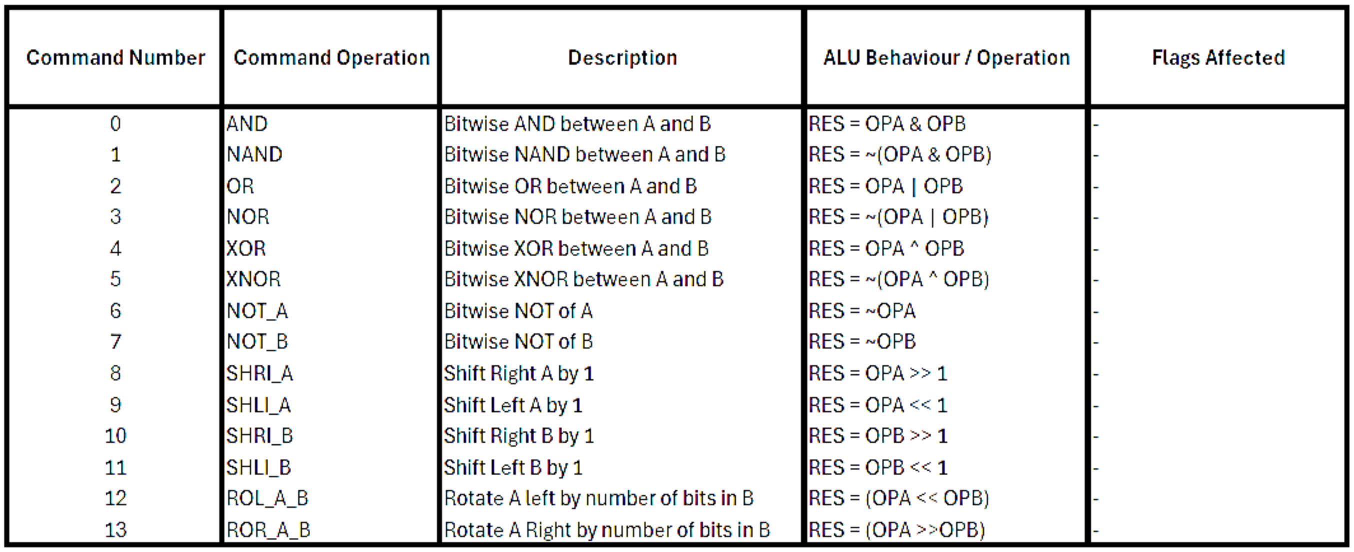
Command Number	Command Operation	Description	ALU Behaviour / Operation	Flags Affected
0	ADD	Unsigned Addition	RES = OPA + OPB	COUT, OFLOW
1	SUB	Unsigned Subtraction	RES = OPA - OPB	COUT, OFLOW
2	ADD_CIN	Addition with Carry-In	RES = OPA + OPB + CIN	COUT, OFLOW
3	SUB_CIN	Subtraction with Borrow (Carry-In)	RES = OPA - OPB - CIN	COUT, OFLOW
4	INC_A	Increment A	RES = OPA + 1	OFLOW
5	DEC_A	Decrement A	RES = OPA - 1	OFLOW
6	INC_B	Increment B	RES = OPB + 1	OFLOW
7	DEC_B	Decrement B	RES = OPB - 1	OFLOW
8	CMP	Compare A and B	Sets G, L, E based on OPA - OP	G, L, E
9	INC_A_B_MUL	Increment A and B, then multiply	RES = (OPA + 1) * (OPB + 1)	-
10	SHL_A_MUL_B	Shift A left by 1, then multiply B	RES = (OPA << 1) * OPB	-
11	ADD_SIGNED	Signed Addition	RES = OPA + OPB (signed)	COUT, OFLOW, G, L, E
12	SUB_SIGNED	Signed Subtraction	RES = OPA - OPB (signed)	COUT, OFLOW, G, L, E

Table 4.1 Arithmetic Operations

The table 4.1 presents the arithmetic command operations implemented in an ALU, associating each command number with a specific arithmetic task such as addition, subtraction, increment, decrement, and comparisons. It explains the behavior of each operation using expressions like RES = OPA + OPB or RES = OPA - OPB - CIN. Signed operations consider two complement arithmetic, while comparison commands set status flags like G, L, and E based on operand relationships. Flags such as COUT (carry out), OFLOW (overflow), and comparison indicators are updated accordingly. This structured mapping is crucial for testbench validation and verifying functional correctness of ALU outputs.

Logical Operations:

Logical operations perform bitwise comparisons (AND, OR, XOR, NOT) on binary inputs to produce true/false results at each bit of position.



Command Number	Command Operation	Description	ALU Behaviour / Operation	Flags Affected
0	AND	Bitwise AND between A and B	RES = OPA & OPB	-
1	NAND	Bitwise NAND between A and B	RES = ~(OPA & OPB)	-
2	OR	Bitwise OR between A and B	RES = OPA | OPB	-
3	NOR	Bitwise NOR between A and B	RES = ~(OPA | OPB)	-
4	XOR	Bitwise XOR between A and B	RES = OPA ^ OPB	-
5	XNOR	Bitwise XNOR between A and B	RES = ~(OPA ^ OPB)	-
6	NOT_A	Bitwise NOT of A	RES = ~OPA	-
7	NOT_B	Bitwise NOT of B	RES = ~OPB	-
8	SHRI_A	Shift Right A by 1	RES = OPA >> 1	-
9	SHLI_A	Shift Left A by 1	RES = OPA << 1	-
10	SHRI_B	Shift Right B by 1	RES = OPB >> 1	-
11	SHLI_B	Shift Left B by 1	RES = OPB << 1	-
12	ROL_A_B	Rotate A left by number of bits in B	RES = (OPA << OPB)	-
13	ROR_A_B	Rotate A Right by number of bits in B	RES = (OPA >>OPB)	-

Table 4.2 Logical Operations

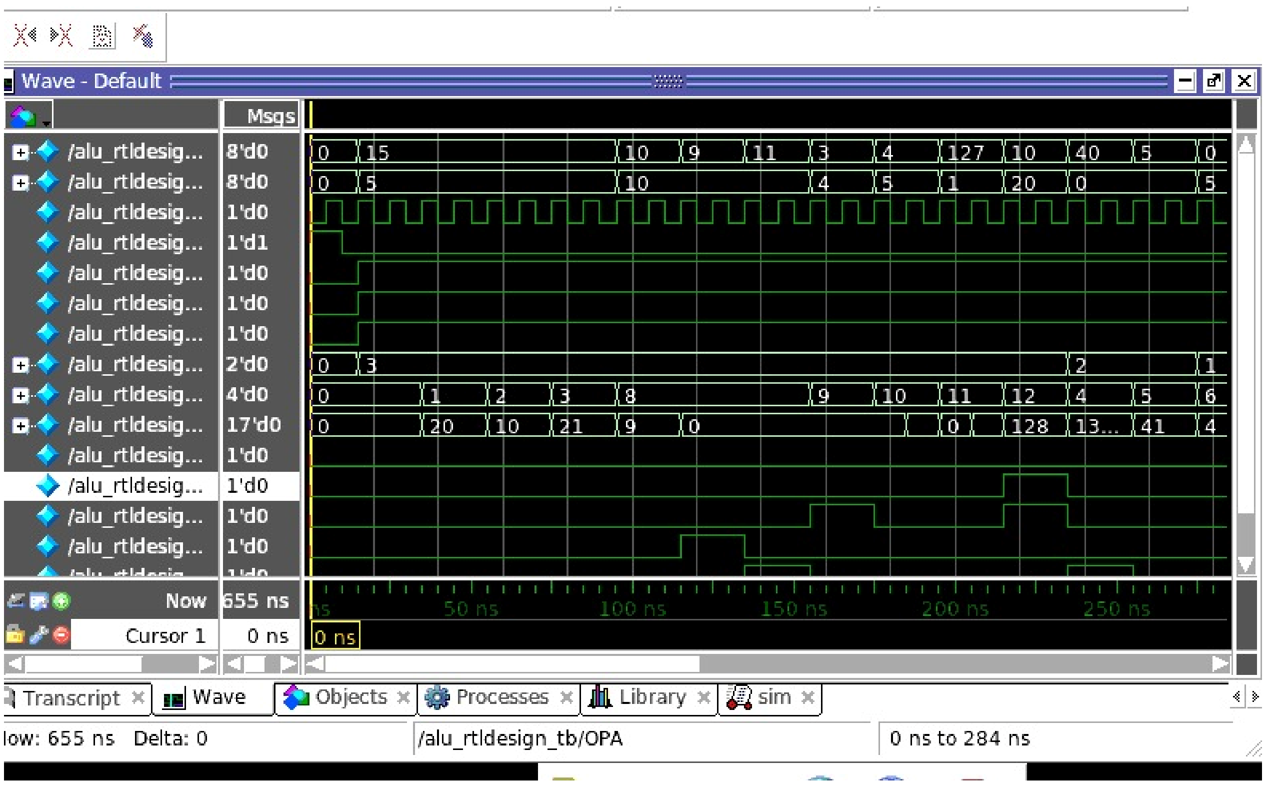
The table 4.2 presents the logical operation set of an ALU, detailing each command number with its corresponding logic based operation like AND, OR, XOR, NOT, and shifts. Each command manipulates binary inputs (OPA and OPB) using bitwise operations, as shown in the “ALU Behaviors” column. Commands such as SHRI_A, SHLI_A, and ROL_A_B involve bitwise shifting or rotation, influencing operand bit positions without arithmetic computation. Most operations do not affect status flags, but certain ones like rotation might trigger the ERR flag if operand width constraints are violated. This layout is essential for simulation, debugging, and validating functional logic behavior in digital designs.

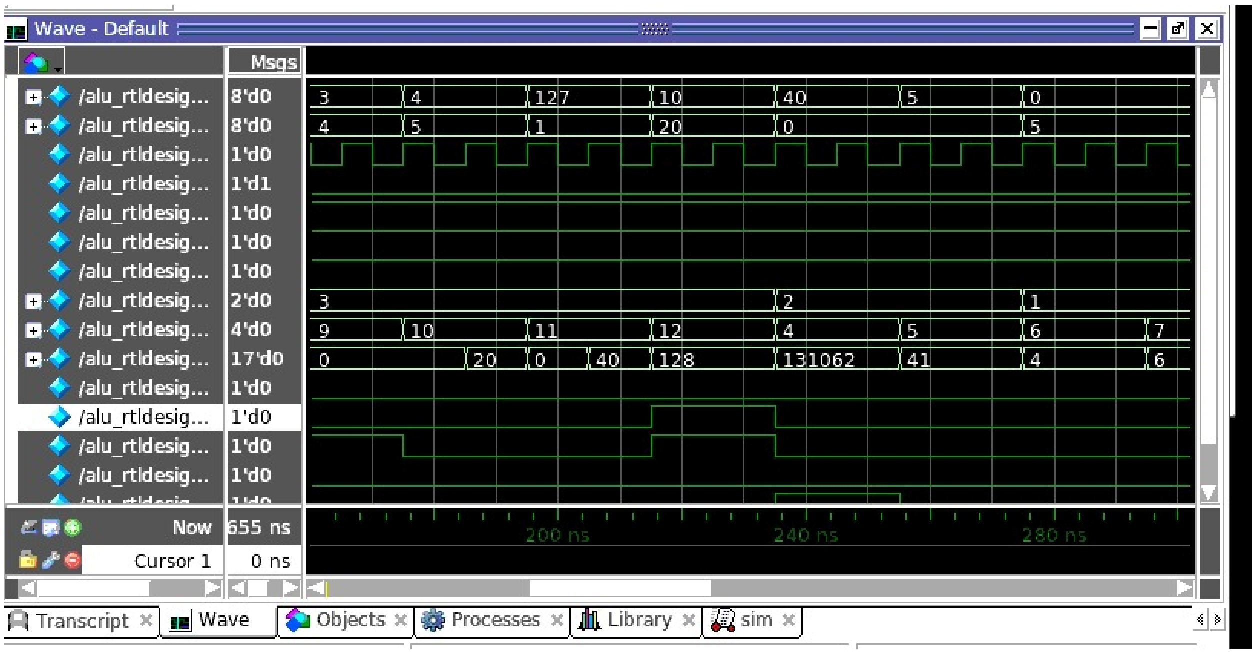
Result

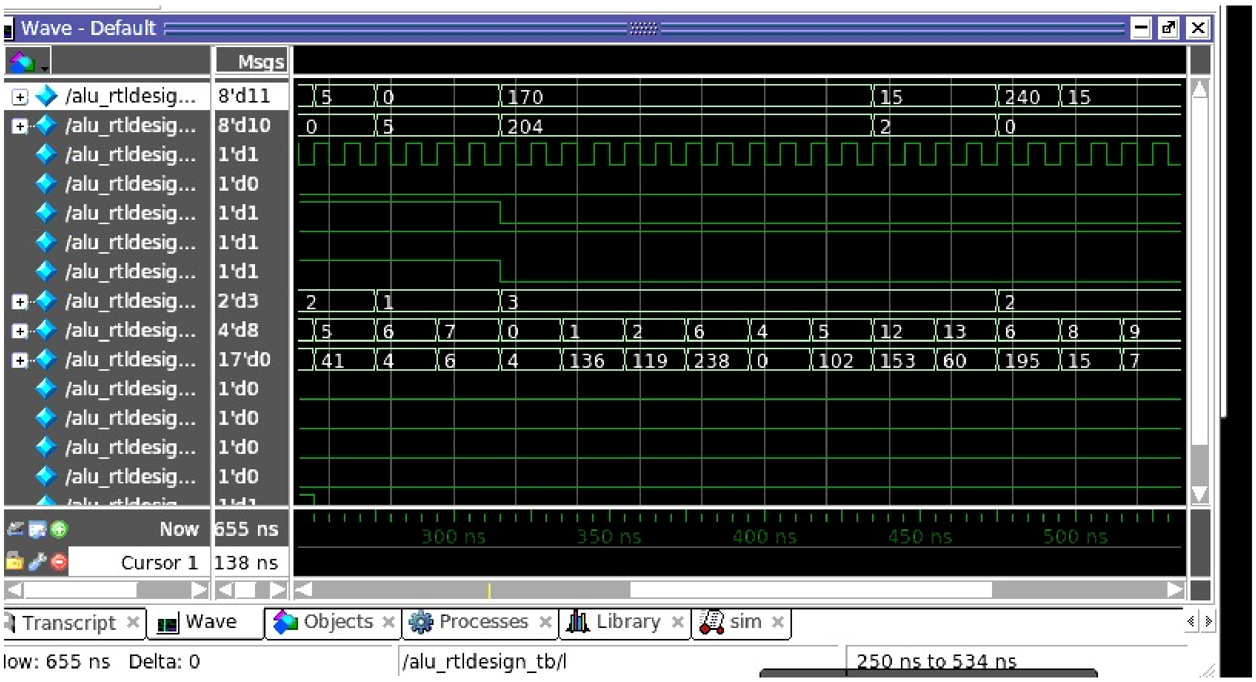
The ALU design was thoroughly simulated and tested, confirming its ability to accurately perform all supported arithmetic and logical operations. It handles addition, subtraction, signed and unsigned calculations, as well as bitwise operations like AND, OR, XOR, and shifts, producing correct results across a wide range of inputs, including boundary cases. The carry-out and overflow flags respond correctly to arithmetic conditions, while the comparison flags (greater-than, equal-to, less-than) reliably reflect operand relationships.

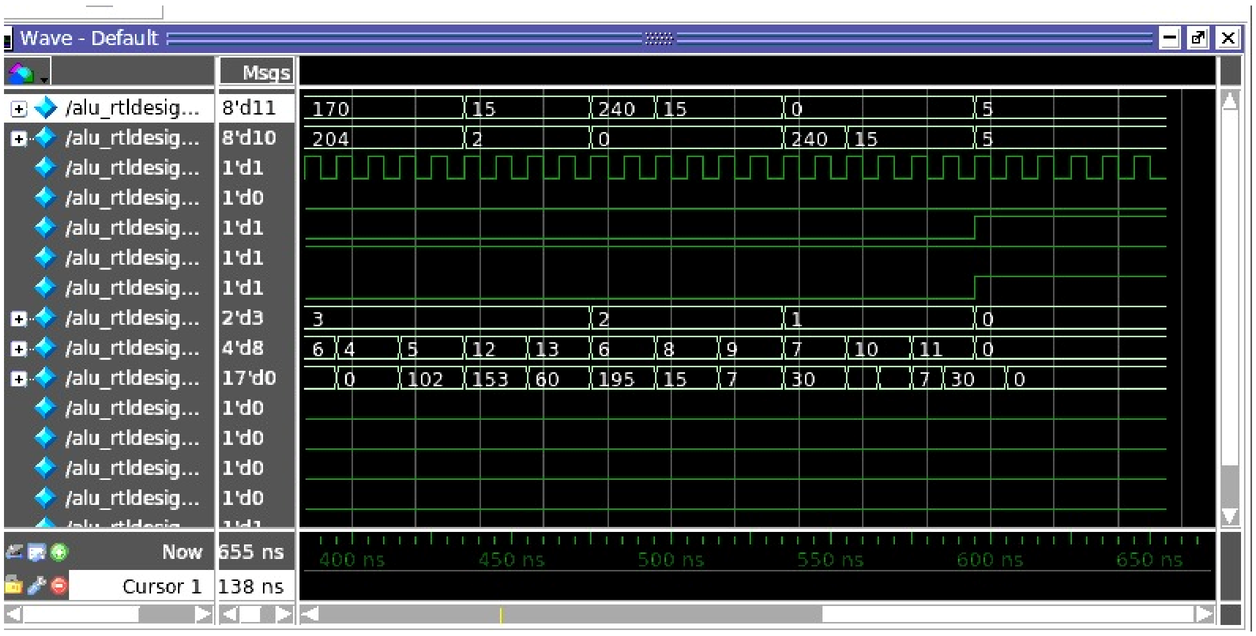
For multi-cycle instructions such as multiplication with increment or shift, the ALU demonstrates the intended latency of one to two clock cycles, validated through waveform analysis that highlights the internal pipelined stages. Invalid commands or operand states trigger the error flag, preventing incorrect outputs and ensuring robustness.

Overall, this confirms the ALU's functional correctness and the successful integration of pipeline techniques for delay-sensitive operations.

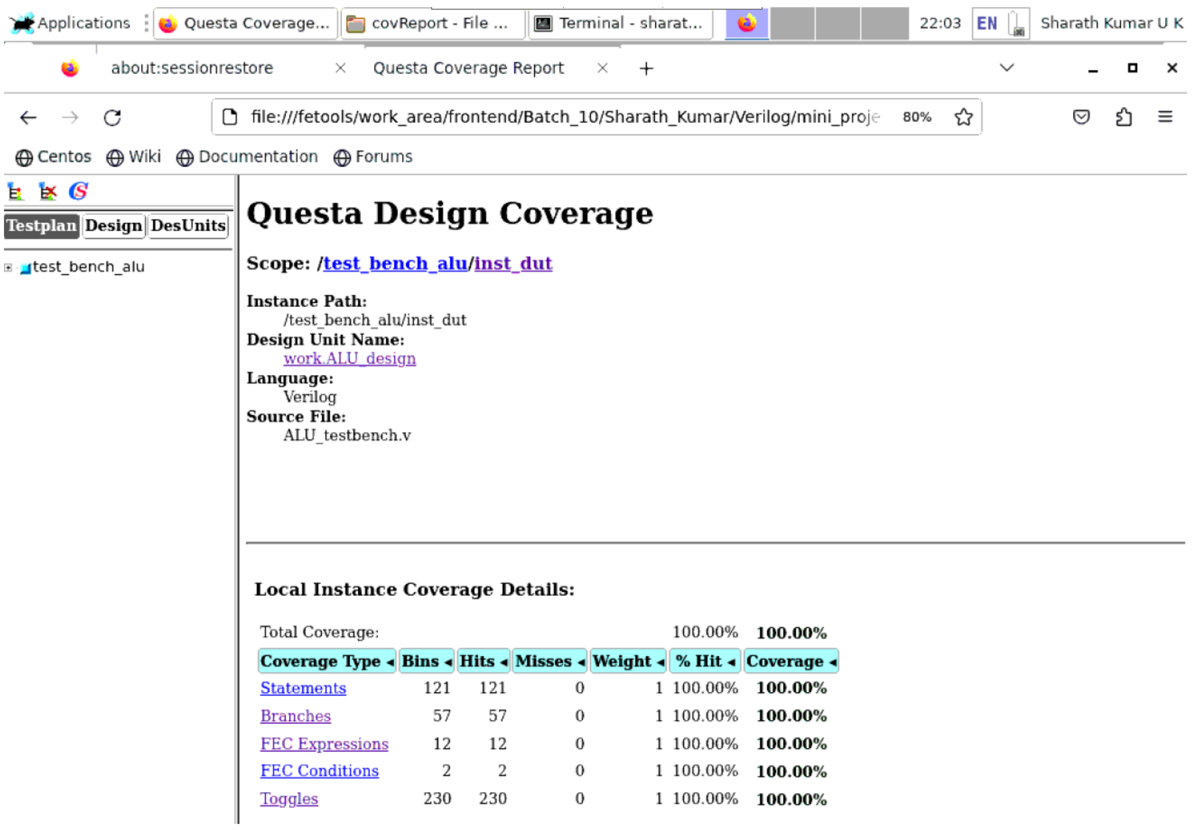



Wave - Default
Msgs
/alu_rtldesig... 8'd0
/alu_rtldesig... 8'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd1
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 2'd0
/alu_rtldesig... 4'd0
/alu_rtldesig... 17'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
3 4 127 10 40 5 0
4 5 1 20 0 5
3 2 1
9 10 11 12 4 5 6 7
0 20 0 40 128 131062 41 4 6
Now 655 ns
Cursor 1 0 ns
200 ns
240 ns
280 ns
Transcript
Wave
Objects
Processes
Library
sim


Wave - Default
Msgs
/alu_rtldesig... 8'd11
/alu_rtldesig... 8'd10
/alu_rtldesig... 1'd1
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd1
/alu_rtldesig... 1'd1
/alu_rtldesig... 1'd1
/alu_rtldesig... 2'd3
/alu_rtldesig... 4'd8
/alu_rtldesig... 17'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
/alu_rtldesig... 1'd0
5 0 170 15 240 15
0 5 204 2 0
2 1 3 2
5 6 7 0 1 2 6 4 5 12 13 6 8 9
41 4 6 4 136 119 238 0 102 153 60 195 15 7
Now 655 ns
Cursor 1 138 ns
300 ns
350 ns
400 ns
450 ns
500 ns
Transcript
Wave
Objects
Processes
Library
sim
Now: 655 ns Delta: 0
/alu_rtldesign_tb/l
250 ns to 534 ns



Code Coverage:



Questa Design Coverage

Scope: /test_bench_alu/inst_dut

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

Local Instance Coverage Details:

Total Coverage: 100.00% **100.00%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	121	121	0	1	100.00%	**100.00%**
Branches	57	57	0	1	100.00%	**100.00%**
FEC Expressions	12	12	0	1	100.00%	**100.00%**
FEC Conditions	2	2	0	1	100.00%	**100.00%**
Toggles	230	230	0	1	100.00%	**100.00%**

Conclusion

The Verilog-based ALU meets all its functional and timing requirements. It executes a wide range of arithmetic and logical operations reliably while maintaining clean modular boundaries. The inclusion of internal pipelining for specific multi-cycle operations such as multiplication adds to the robustness of the design.

Flag generation and result handling are implemented in a synchronized and logically separated manner, ensuring correctness and clarity. The design is fully synthesizable and structurally prepared for use in real-world digital systems like embedded processors and ASICs.

Future Improvement

To enhance the ALU's capabilities and performance, the following improvements are proposed:

Support Wider Data Widths:
Extend the design to support 16-bit, 32-bit, or even 64-bit operands for more complex computations.

Add More Arithmetic Operations:
Include multiplication with signed inputs, division, modulo, and floating-point operations.

Implement Pipelining:
Introduce pipeline stages to increase throughput and support higher clock frequencies.

Improve Error Handling:
Extend the error detection by introducing new unique error flags to handle invalid inputs, command conflicts, and timing violations.

Add Support for Signed Multiplication:
Implement signed multiplication and division to extend arithmetic capability.

Incorporate Interrupt or Exception Handling:

Add mechanisms to detect and report exceptional conditions during operation.

Expand Command Set:

Add more bitwise and arithmetic instructions like bit counting, population count, and conditional moves.

Integrate with Higher-Level Processor Designs:

Design an interface for the ALU to integrate seamlessly with registers, control units, and memory.