ALU PROJECT

Introduction:

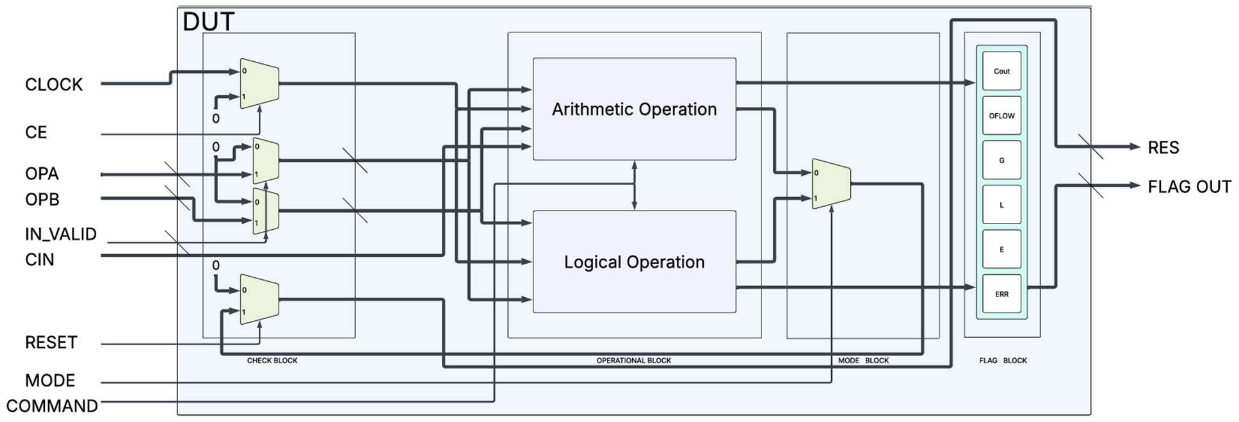
This project focuses on designing and implementing an Arithmetic Logic Unit (ALU) using Verilog HDL. As a core component of digital systems, the ALU plays a vital role in handling various arithmetic and logical operations on binary data. Built with a modular and parameterized approach, it efficiently performs tasks such as addition, subtraction, bitwise operations (AND, OR, XOR, etc.), shifts, comparisons, and even specialized functions like signed arithmetic and multiplication with latency control.

While many of its operations are purely combinational, the design also incorporates clocked logic to support pipelining and multi-cycle tasks like multiplication, ensuring synchronization with digital system timing. The ALU responds to command inputs that determine the operation, along with mode and control signals, and generates outputs such as results, carry-out, overflow, and condition flags (greater, equal, less). This Verilog-based ALU is a crucial element in processor and embedded system design, providing a flexible and efficient foundation for essential computational processes.

Objectives:

1. Design a synthesizable ALU in Verilog that can be implemented on hardware.
2. Allow the input, output, and command widths to be set through parameters for flexibility.
3. Support both signed and unsigned addition and subtraction operations.
4. Correctly generate carry-out and overflow flags based on the operation and input types.
5. Ensure all operations except multiplication are complete with a one-clock-cycle delay.
6. Ensure multiplication is complete with a 2-clock-cycle delay.
7. Implement multiplication so that its result is available after three clock cycles.
8. Raise an error flag whenever an invalid command or input condition occurs.

Architecture:



Inputs:

- CLOCK (CLK): Synchronizes sequential operations.
- CE (Clock Enable): Controls when the ALU processes inputs.
- OPA, OPB: Input operands for arithmetic/logical operations.
- IN_VALID: Indicates when input data is ready.
- CIN: Carry-in for arithmetic operations (e.g., adder circuits).
- RESET: Asynchronous reset signal.
- MODE: Selects between operation modes (e.g., arithmetic vs. logical).
- COMMAND (Opcode): Specifies the operation (e.g., ADD, AND SHIFT).

Outputs:

- RES: Result of the ALU operation.
- FLAG OUT: Status flags (e.g., Carry, Overflow, Zero).

Functional Blocks:

1. Arithmetic Operation Block: Handles additions, subtractions, etc.
2. Logical Operation Block: Performs AND, OR, XOR, etc.
3. Flag Block: Generates condition flags (C, OVFLOW, G, L, E, ERR).
4. Check Block (RESET, MODE, CLOCK EN, INPUT VALID): Manages clock in, mode selection, reset output and operation sequencing.

Working:

The Arithmetic Logic unit is designed to process digital signals through various functional blocks, performing arithmetic and logical operations based on user-defined inputs and control signals. It efficiently handles data computation while ensuring error detection and output validation, making it a fundamental component for digital processing applications.

Key Control Signals

Reset:

Reset Control the boot up and initial assignment of the output data. If the reset is high, the out resets its initial data.

Clock Enable:

The clock is the main input in a sequential circuit, enabling it to change the output based on current input or previous output.

Input Valid:

Input valid is used to check if the given input is valid or not. If valid, accept the input or else raise the error flag.

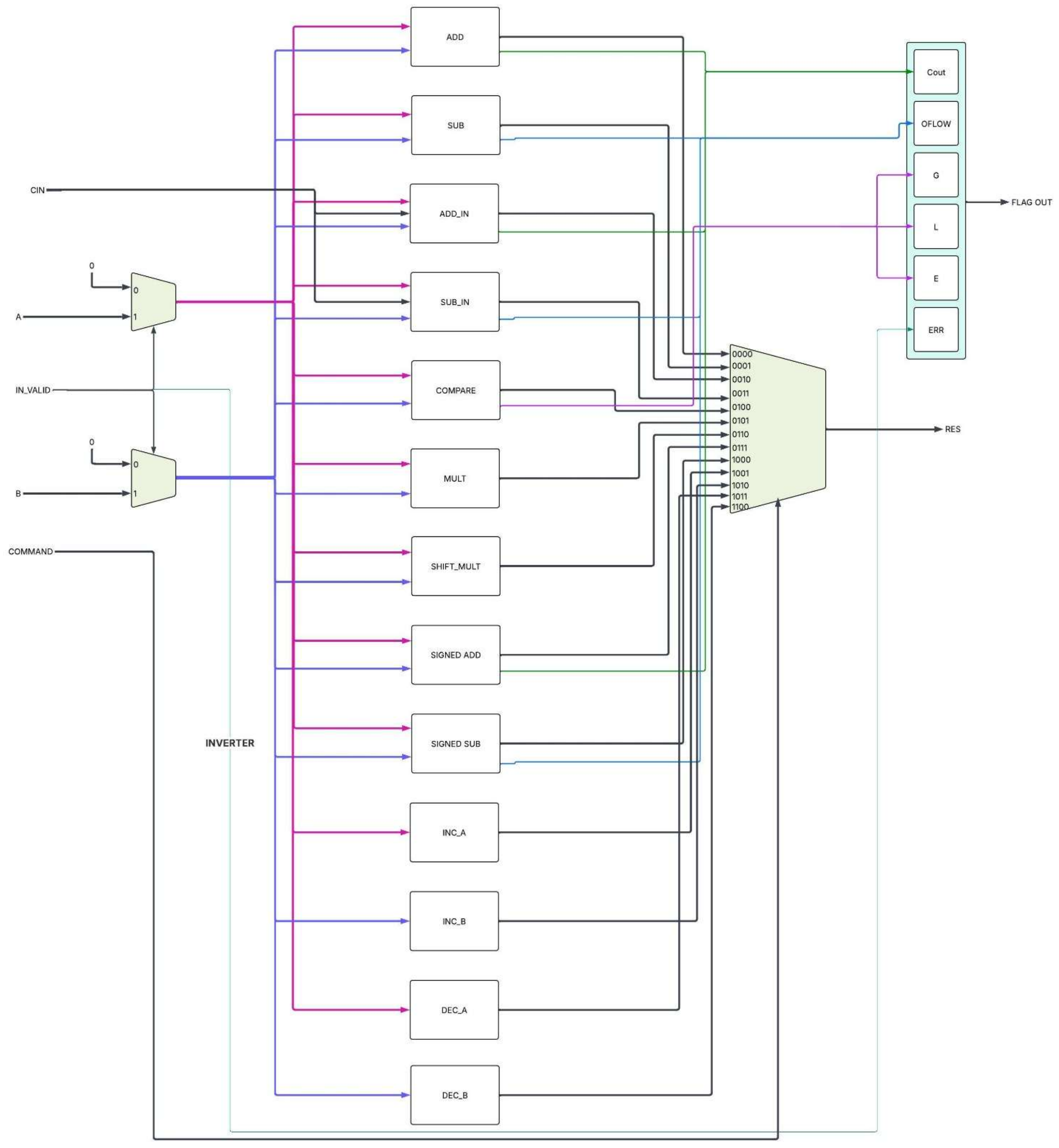
Mode Selection:

Mode is used to select arithmetic or logical operations. If the Mode is high, it selects the arithmetic operation and if it is low then it selects the Logical operation.

Command:

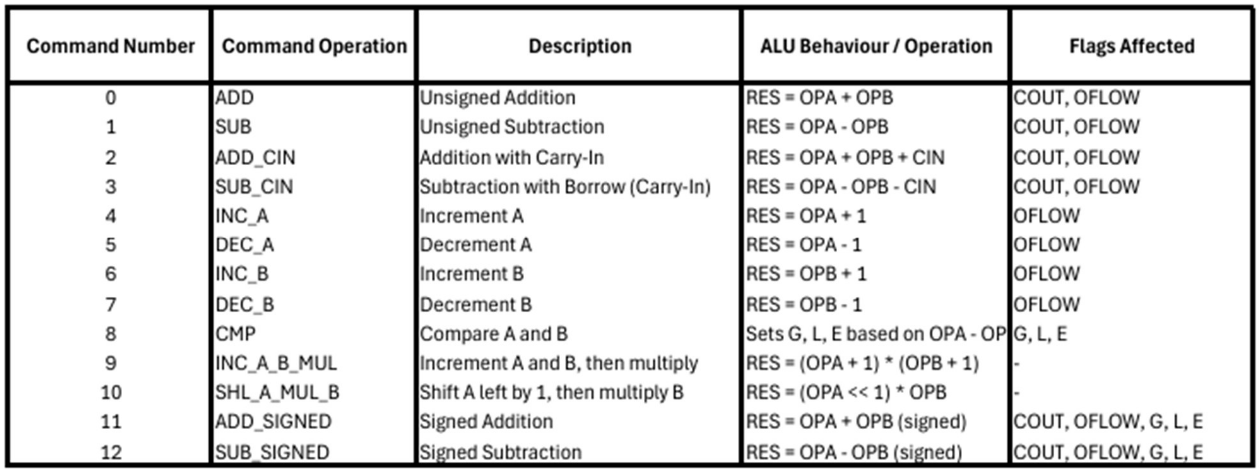
COMMAND is a 4-bit control signal, meaning it can represent values from 0000 (0) to 1111 (15). Each value corresponds to one of the 16 ALU operations.

Arithmetic Operation:



Arithmetic Operations:

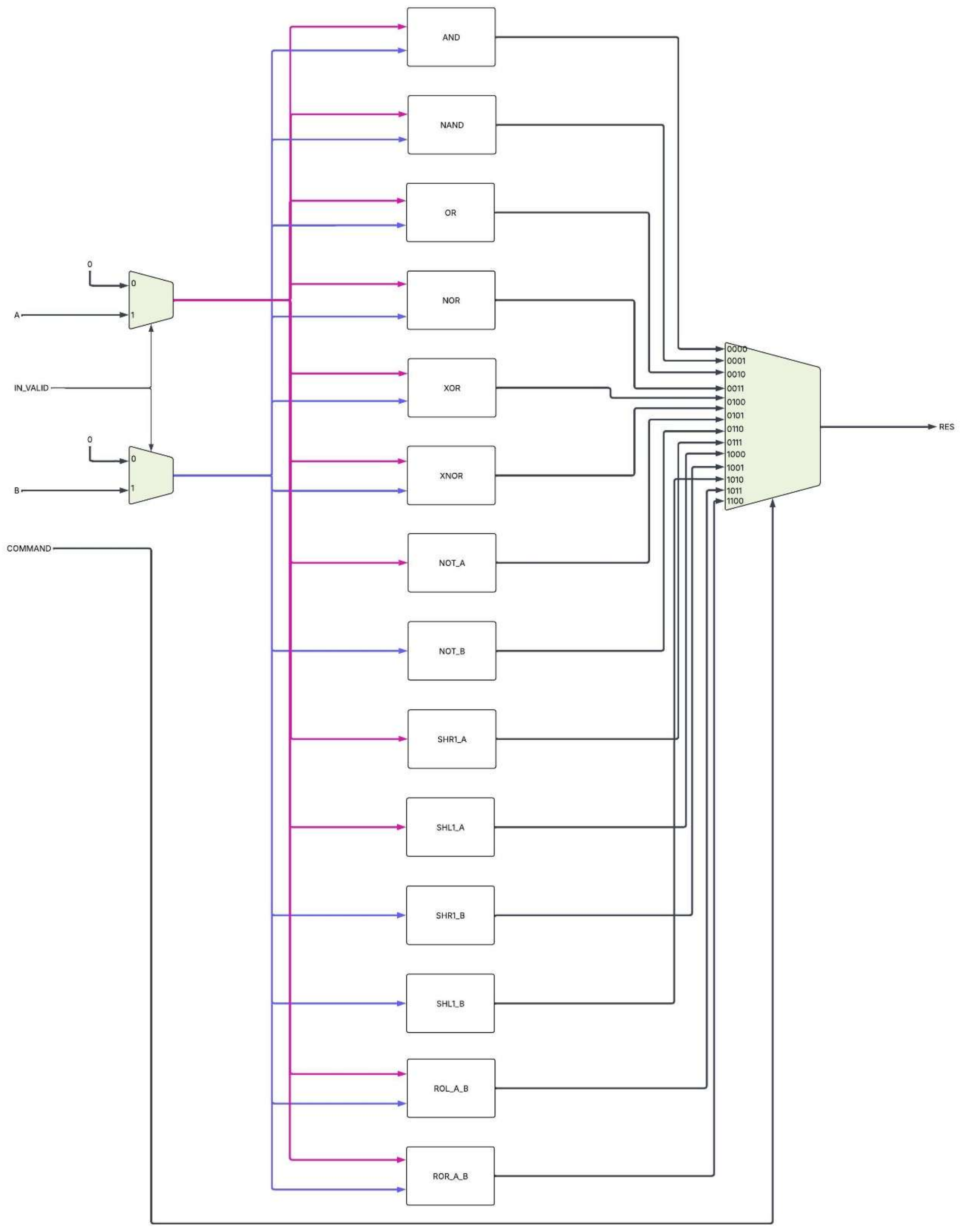
Arithmetic operations perform mathematical calculations like addition, subtraction, shifting, and rotation on binary inputs to manipulate numerical values.



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

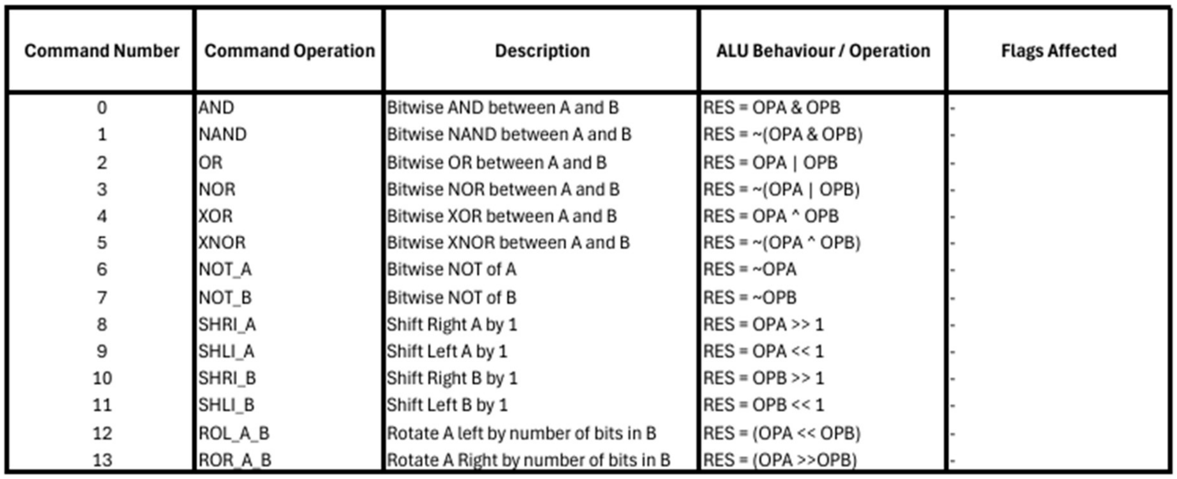
This table presents the arithmetic command operations implemented in an ALU, associating each Command Number with a specific arithmetic task such as addition, subtraction, increment, decrement, and comparisons. It explains the behavior of each operation using expressions like RES = OPA + OPB or RES = 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 Operation



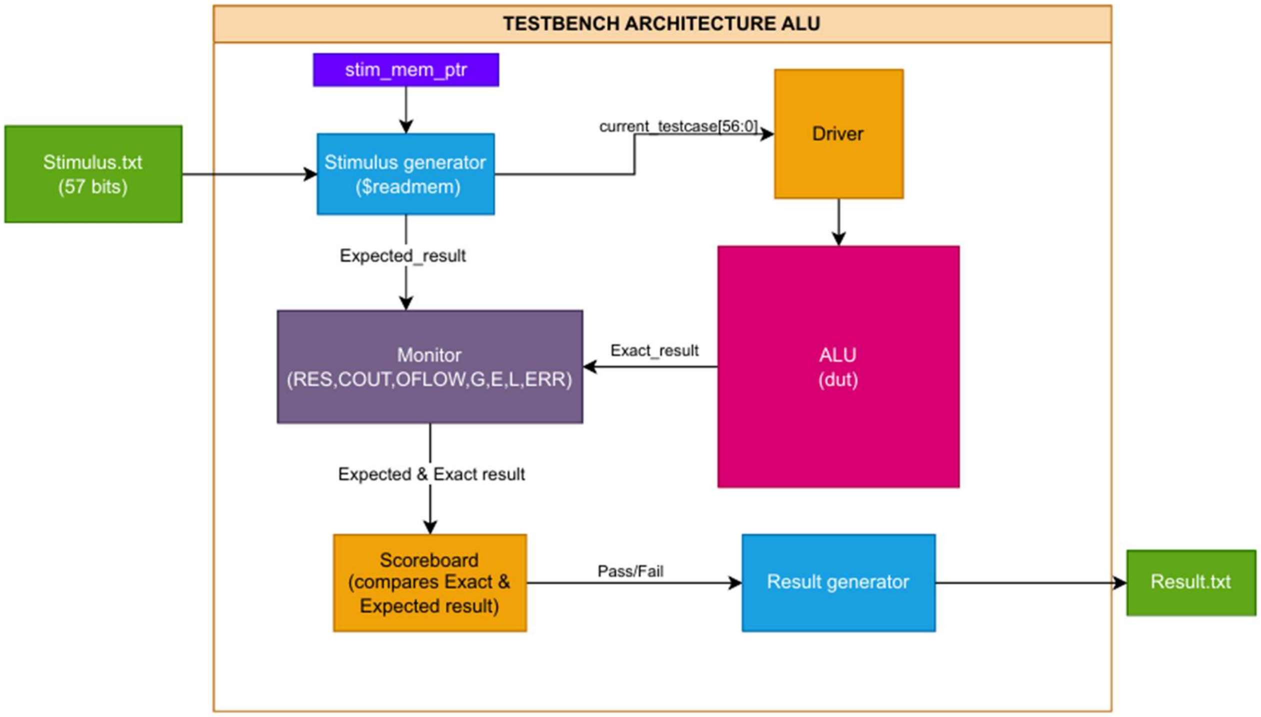
Logical Operations:

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)	-

This table 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.



This diagram illustrates the verification flow for an Arithmetic Logic Unit (ALU) using a structured testbench. Here's a breakdown:

→ Stimulus Packet (57 bits)

- Generated by stim_mem_ptr (via $readmem), it contains:
  - Input operands and control signals for the ALU.
  - Expected results for comparison.

→ Monitor

- Captures the ALU's actual outputs (result, carry-out, overflow, and error flags).

→ Scoreboard

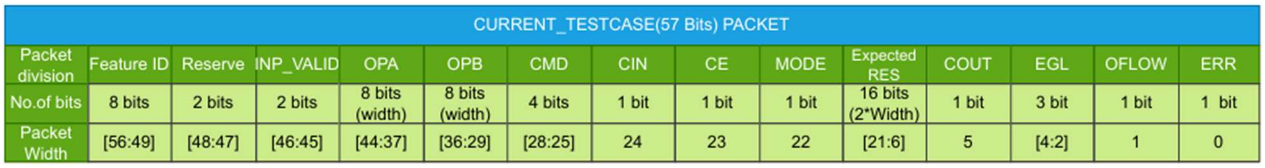
- Compares the actual results (from the monitor) with the expected results (from the stimulus).
- Flags Pass/Fail for each test case.

→ Driver

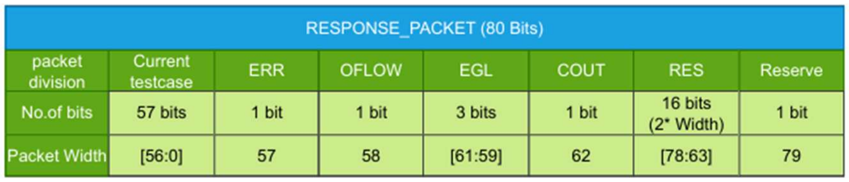
- Feeds the stimulus inputs to the ALU (DUT).

→ Result Logging

- Test results (Pass/Fail) are saved in Result.txt for analysis.



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

The diagram shows how the stimulus packet is structured and how data flows during simulation. The verification process uses a testbench with three key components: the DUT (Design Under Test), the driver, the monitor, and the scoreboard.

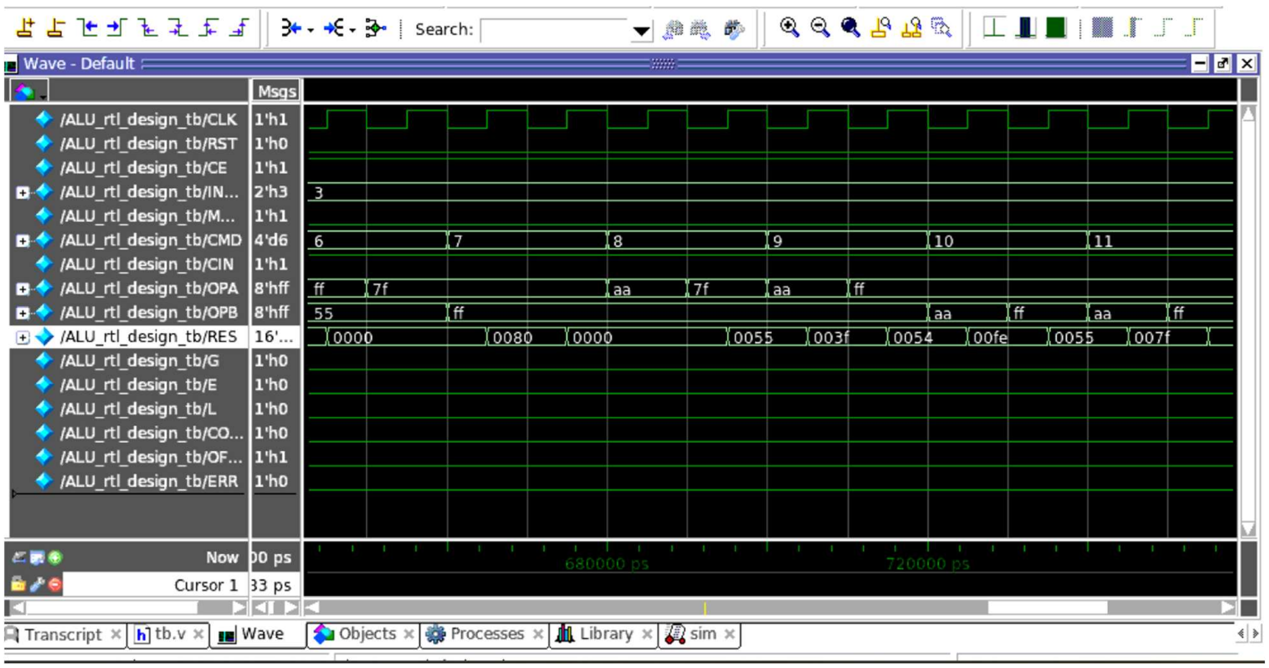
The driver generates input signals for the DUT (in this case, the ALU module) based on the stimulus packet. This packet contains not just the operands and control signals but also the expected results and a unique test ID. The ID helps track different test cases during simulation.

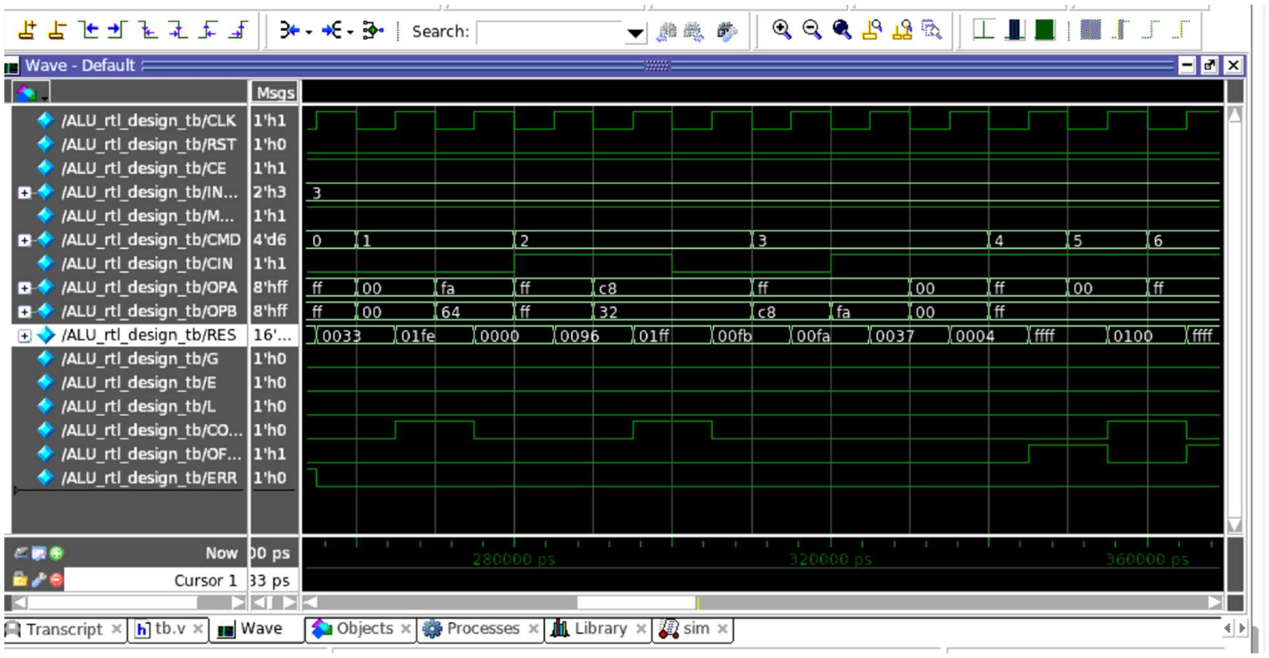
The monitor passively observes the actual inputs sent to the DUT and the outputs it produces. It records all signal activity without interfering with the DUT's operation.

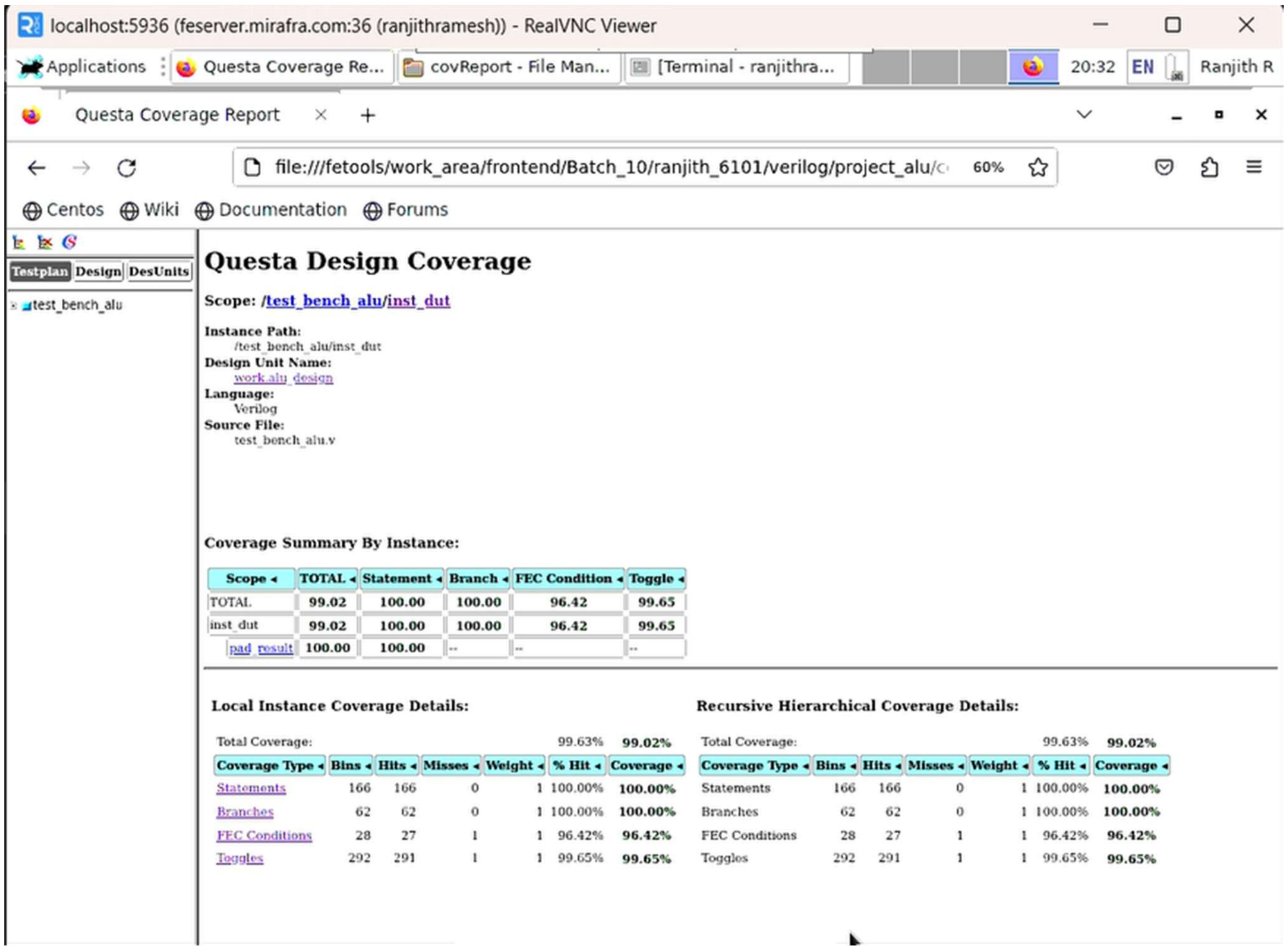
Meanwhile, the scoreboard checks for correctness by comparing the expected results (from the stimulus packet) with the actual outputs (captured by the monitor). If there's a mismatch, the test case is flagged as a failure; otherwise, it passes. Detailed results, including test IDs and pass/fail status are logged in a text file for later review, making debugging easier.

This structured approach ensures thorough verification while keeping track of every test scenario.

Result:


Wave - Default
Msgs
/ALU_rtl_design_tb/CLK 1'h1
/ALU_rtl_design_tb/RST 1'h0
/ALU_rtl_design_tb/CE 1'h1
/ALU_rtl_design_tb/IN... 2'h3
/ALU_rtl_design_tb/M... 1'h1
/ALU_rtl_design_tb/CMD 4'd6
/ALU_rtl_design_tb/CIN 1'h1
/ALU_rtl_design_tb/OPA 8'hff
/ALU_rtl_design_tb/OPB 8'hff
/ALU_rtl_design_tb/RES 16'...
/ALU_rtl_design_tb/G 1'h0
/ALU_rtl_design_tb/E 1'h0
/ALU_rtl_design_tb/L 1'h0
/ALU_rtl_design_tb/CO... 1'h0
/ALU_rtl_design_tb/OF... 1'h1
/ALU_rtl_design_tb/ERR 1'h0
3
6 7 8 9 10 11
ff 7f aa 7f aa ff
55 ff aa ff aa ff
0000 0080 0000 0055 003f 0054 00fe 0055 007f
Now 00 ps
Cursor 1 33 ps
680000 ps
720000 ps
Transcript
tb.v
Wave
Objects
Processes
Library
sim


Wave - Default
Msgs
/ALU_rtl_design_tb/CLK 1'h1
/ALU_rtl_design_tb/RST 1'h0
/ALU_rtl_design_tb/CE 1'h1
/ALU_rtl_design_tb/IN... 2'h3
/ALU_rtl_design_tb/M... 1'h1
/ALU_rtl_design_tb/CMD 4'd6
/ALU_rtl_design_tb/CIN 1'h1
/ALU_rtl_design_tb/OPA 8'hff
/ALU_rtl_design_tb/OPB 8'hff
/ALU_rtl_design_tb/RES 16'...
/ALU_rtl_design_tb/G 1'h0
/ALU_rtl_design_tb/E 1'h0
/ALU_rtl_design_tb/L 1'h0
/ALU_rtl_design_tb/CO... 1'h0
/ALU_rtl_design_tb/OF... 1'h1
/ALU_rtl_design_tb/ERR 1'h0
3
0 1 2 3 4 5 6
ff 00 fa ff c8 ff 00 ff 00 ff
ff 00 64 ff 32 c8 fa 00 ff
0033 01fe 0000 0096 01ff 00fb 00fa 0037 0004 ffff 0100 ffff
Now 00 ps
Cursor 1 33 ps
280000 ps
320000 ps
360000 ps
Transcript
tb.v
Wave
Objects
Processes
Library
sim



localhost:5936 (feserver.mirafra.com:36 (ranjithramesh)) - RealVNC Viewer

Questa Coverage Report

file:///fetools/work_area/frontend/Batch_10/ranjith_6101/verilog/project_alu/c

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:
test_bench_alu.v

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle
TOTAL	99.02	100.00	100.00	96.42	99.65
inst_dut	99.02	100.00	100.00	96.42	99.65
pad_result	100.00	100.00	--	--	--

Local Instance Coverage Details:

Total Coverage: 99.63% **99.02%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	166	166	0	1	100.00%	**100.00%**
Branches	62	62	0	1	100.00%	**100.00%**
FEC Conditions	28	27	1	1	96.42%	**96.42%**
Toggles	292	291	1	1	99.65%	**99.65%**

Recursive Hierarchical Coverage Details:

Total Coverage: 99.63% **99.02%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	166	166	0	1	100.00%	**100.00%**
Branches	62	62	0	1	100.00%	**100.00%**
FEC Conditions	28	27	1	1	96.42%	**96.42%**
Toggles	292	291	1	1	99.65%	**99.65%**

Conclusion

The ALU was successfully designed and implemented using Verilog, with thorough testing to ensure correct functionality. It supports a wide range of operations—from basic arithmetic and logic to more advanced shifts, rotations, and comparisons—while also handling flag setting and error detection. The modular and well-structured design makes it easy to integrate into larger systems, such as CPUs or control units.

To verify its accuracy, we used a testbench that automatically checks outputs against expected results using a scoreboard method. This approach helped catch potential errors early, improving the overall reliability of the design. Simulation results confirm that the ALU performs as intended, meeting all specified requirements.

Future Improvements

To further enhance the ALU, we could implement pipelining—dividing operations into stages (fetch, decode, execute, write-back) to boost throughput and efficiency. This would make it better suited for high-performance CPUs and complex computing tasks.

Other potential upgrades include:

- Expanding functionality (floating-point support, complex number arithmetic, cryptographic operations).
- Optimizing performance through advanced techniques like parallel execution or low-latency designs.
- Automating verification using AI-driven testbench generation to improve testing efficiency and catch bugs faster.

With these improvements, the ALU could become even more powerful, versatile, and efficient, making it ideal for cutting-edge digital systems while maintaining precision and speed.