ALU – VERIFICATION

BHUVANESH

27bhuvanesh5@gmail.com

INDEX

CONTENT

CHAPTER 1 PROJECT OVERVIEW AND SPECIFICATIONS
1.1 Project Overview
1.2 Verification Objectives
1.3 DUT Interfaces
CHAPTER 2 TESTBENCH ARCHITECTURE AND METHODOLOGY
2.1 Testbench Architecture
2.2 Component Details and Flowchart
CHAPTER 3 VERIFICATION RESULTS AND ANALYSIS
3.1 Error in the DUT
3.2 Coverage Report
CONCLUSION

FUTURE SCOPE

CHAPTER 1: PROJECT OVERVIEW AND SPECIFICATION

1.1 Project Overview

This project focuses on verifying a parameterized Arithmetic Logic Unit (ALU) that performs arithmetic operations (addition, subtraction, increment, decrement, multiplication), logical operations (AND, OR, XOR, NOT, NAND, NOR, XNOR), as well as shift, rotate, comparator, and error-checking functions. Verification is carried out using the Universal Verification Methodology (UVM) with constrained-random and directed tests, assertions, and functional coverage to ensure correctness and robustness of the design.

1.2 Verification Objective

The objective of the project is to:

- Verify functional correctness of all ALU operations — including arithmetic, logical, comparison, and shift/rotate — as determined by CMD and MODE.
- Ensure input protocol compliance by checking that INP_VALID reflects operand availability (2'b01, 2'b10, or 2'b11) and operations proceed only when valid.
- Verify that when only one operand is valid (INP_VALID = 2'b01 or 2'b10), the ALU waits up to 16 clock cycles for the second operand, and asserts ERR if it doesn't arrive.
- Confirm timing behaviour, ensuring results are available after 1 or 2 clock cycles, depending on the operation.
- Apply constrained-random testing and functional coverage to explore all valid/invalid input combinations, including edge cases like overflow, underflow, and invalid rotate commands.

1.3 DUT Interfaces

Block diagram of parameterized ALU is shown in Figure 1.

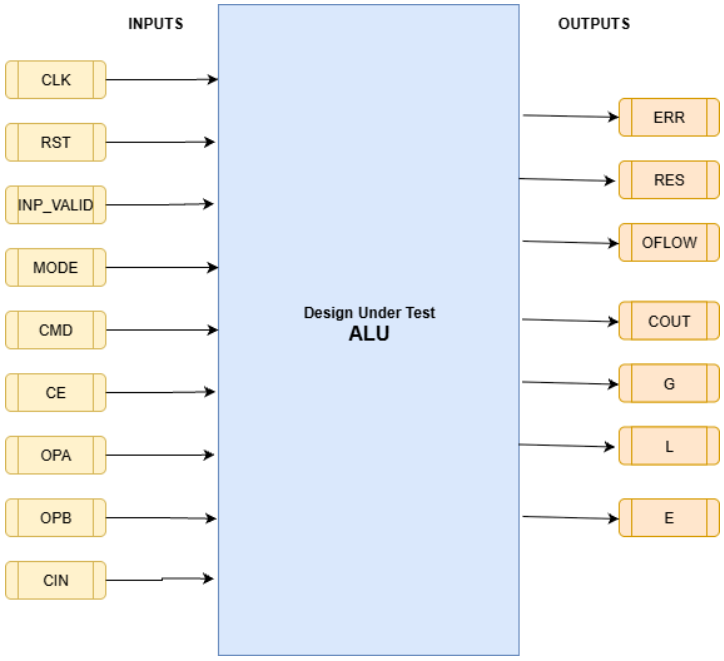


Figure 1. ALU under test

Below is the table which describes about the pins in the ALU which is design under test.

	PIN NAME	PIN DESCRIPTION
INPUTS	CLK (Clock)	Synchronous clock
	RST (Reset)	Asynchronous reset
	CE (Clock Enable)	Enables the ALU to perform operations on the clock edge
	CDM	A 4-bit command input that specifies the operation to be performed
	MODE	Determines the type of operation i.e. if MODE = 1 then Arithmetic operation else Logical operation
	CIN (Carry In)	Used for addition/subtraction with carry/borrow
	OPA / OPB [Width-1:0]	Parameterized inputs for operand A and operand B
	INP_VALID	It is a 2-bit input which indicates validity of input operands i.e. 00: No operand is valid 01: Operand A is valid 10: Operand B is valid 11: Both operands are valid
OUTPUTS	ERR	Error signal for invalid operations
	RES [Width-1:0]	Result from the logical or arithmetic operation
	G, L, E	Comparator outputs indicating relationship between operands i.e. greater-than, less-than and equal-to
	COUT	Carry out for arithmetic operation
	OFLOW	Overflow flag for arithmetic operation

Below is the table for the ALU showing specific arithmetic operation cases for given inputs, where the resulting outputs must be carefully considered as follows:

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 comparison	G, L, E
9	INC_A_B_MUL	Increment A and B, then multiply	RES = (OPA + 1) * (OPB + 1)	-
10	SHL_A_MUL_B	Left shift A by 1, then multiply with B	RES = (OPA << 1) * OPB	-

The designed ALU supports various logical operations when the signal MODE is deasserted which includes AND, OR, NAND, NOR, XOR, XNOR, NOT_A, NOT_B, SHR1_A, SHL1_A, SHR1_B, SHL1_B, ROL_A_B (Rotate Left A by bits specified in B), ROR_A_B (Rotate Right A by bits specified in B)

Below is the table for the ALU showing specific logical operation cases for given inputs, where the resulting outputs must be carefully considered as follows:

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	SHR1_A	Shift Right A by 1	RES = OPA >> 1	-
9	SHL1_A	Shift Left A by 1	RES = OPA << 1	-
10	SHR1_B	Shift Right B by 1	RES = OPB >> 1	-
11	SHL1_B	Shift Left B by 1	RES = OPB << 1	-
12	ROL_A_B	Rotate Left A by bits specified in B	Rotate A left by the lowest bits of B needed to cover the operand width; set ERR if any higher bits of B, are nonzero.	ERR
13	ROR_A_B	Rotate Right A by bits specified in B	Rotate A right by the lowest bits of B needed to cover the operand width; set ERR if any higher bits of B, are nonzero.	ERR

CHAPTER 2: TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 TESTBENCH ARCHITECTURE

GENERAL TESTBENCH ARCHITECTURE

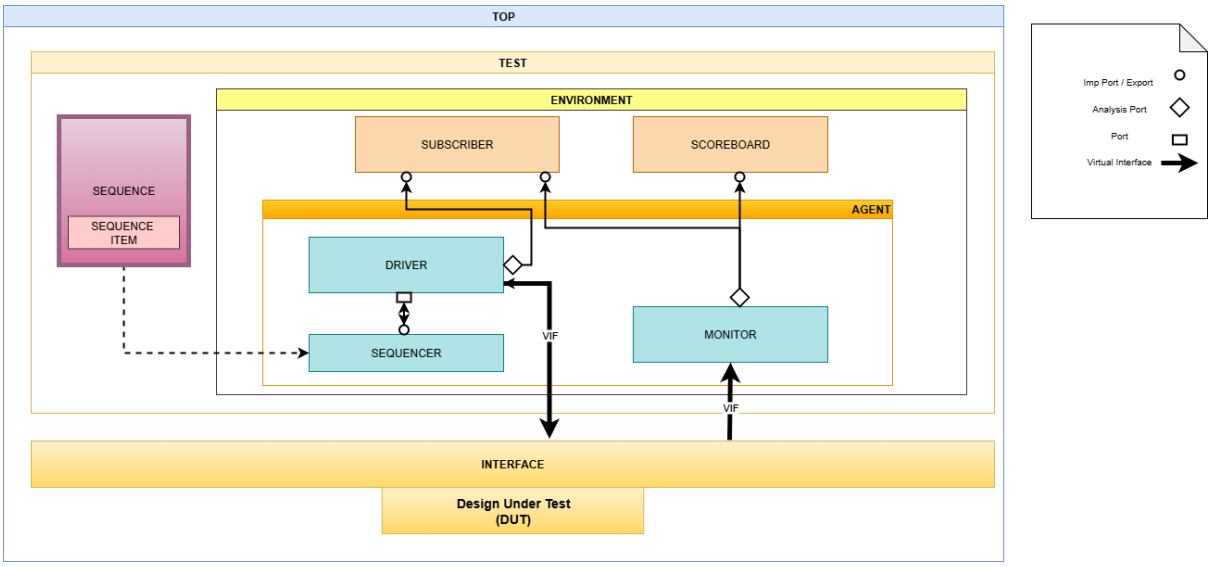


Figure 1. General UVM Testbench Architecture

ALU TESTBENCH ARCHITECTURE

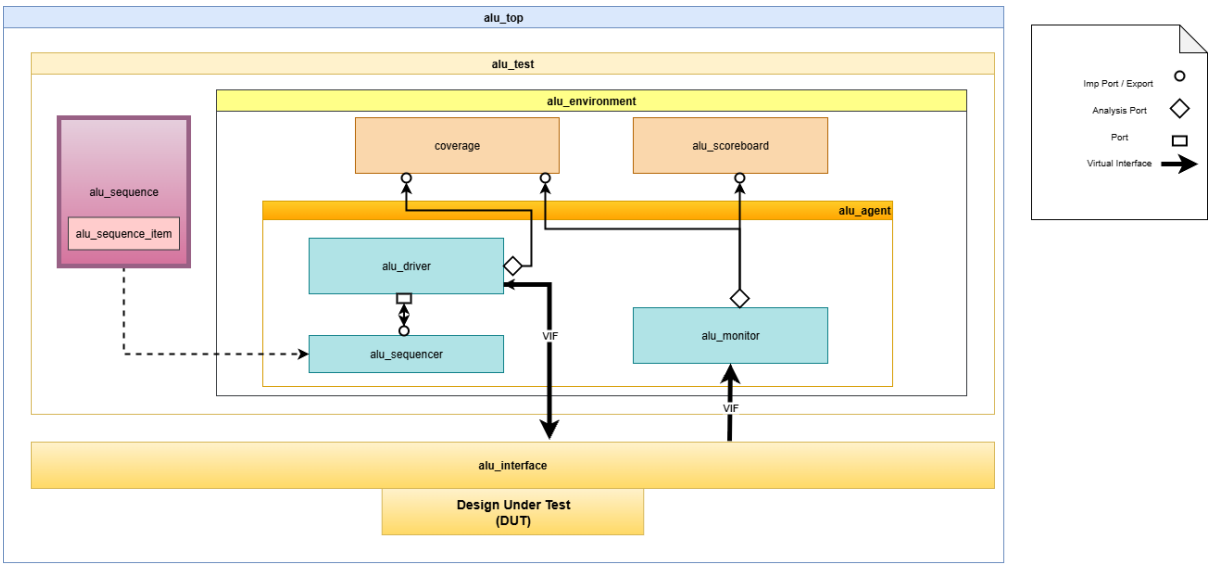


Figure 2. UVM Based ALU Testbench Architecture

2.2 COMPONENT DETAILS AND FLOWCHART

The verification environment of the ALU is designed using UVM components interconnected through well-defined ports and interfaces to enable communication and data flow, ensuring robust verification of the Design Under Test (DUT).

1. **alu_sequence to alu_sequencer**
   - Connected via a dashed line representing a stimulus flow.
   - alu_sequence generates transaction items (alu_sequence_item) that are fed into the alu_sequencer. These transaction items form the input stimulus for the testbench, exercising various ALU operations.
2. **alu_sequencer to alu_driver**
   - Connected via an implementation port (white diamond symbol).
   - The alu_sequencer acts as a master sequencer, issuing transaction items to the alu_driver through a sequencer-driver port. The alu_driver translates these high-level transactions into pin-level signals for the DUT.
3. **alu_driver to DUT (alu_interface)**
   - Connected through a virtual interface (VIF) (thick black arrow).
   - The alu_driver drives signals onto the alu_interface, which connects directly to the DUT inputs. This virtual interface abstracts the low-level signal connection, providing a clean and reusable method to interact with the DUT signals.
4. **alu_monitor to DUT (alu_interface)**
   - Connected also through a virtual interface (VIF) (thick black arrow).
   - The alu_monitor observes DUT outputs via the same alu_interface, capturing DUT responses for analysis without affecting the DUT signals.
5. **alu_monitor to alu_scoreboard**
   - Connected via an analysis port (diamond symbol).
   - The alu_monitor sends observed output data to the alu_scoreboard through this analysis port for result comparison and functional checking.
6. **alu_driver and alu_monitor to coverage and alu_scoreboard**
   - Connected via analysis ports (diamond symbols) to both coverage and alu_scoreboard.
   - Collected data is forwarded to the coverage module to track functional coverage metrics and to the scoreboard for assertion and correctness checking.

CHAPTER 3: VERIFICATION RESULT AND ANALYSIS

3.1 ERROR IN THE DUT

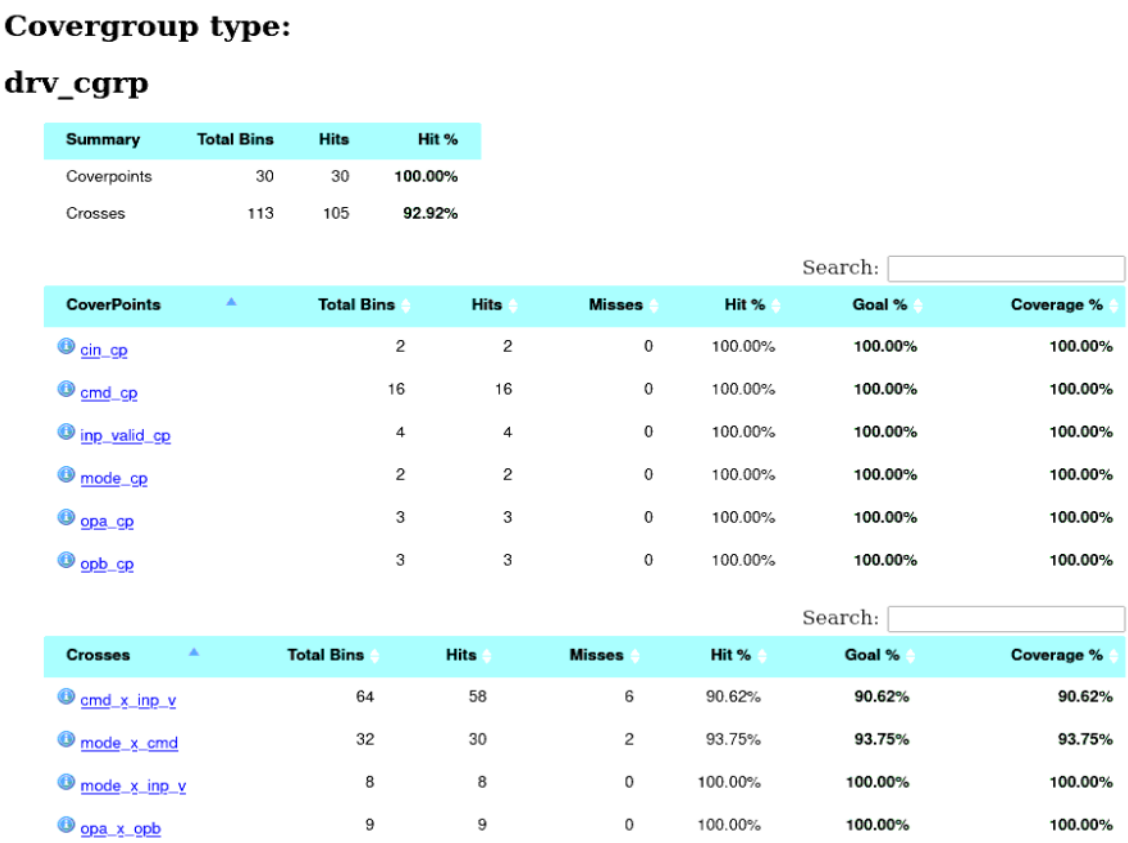
- The increment operation on operand A (OPA) is incorrect according to the specification
- The operation of increment and decrement on operand B (OPB) is incorrect according to the specification
- CMD 13 performs a left rotation instead of the required right rotation, leading to incorrect output
- When OPB[7:4] ≠ 0 in CMD 13, the ERR flag is not set, failing to indicate an invalid input condition
- RES is declared with a hardcoded width [8:0] instead of using parameter DW, breaking scalability
- The design does not verify INP_VALID = 2'b11 before executing dual-operand commands, allowing operations with missing operands
- For single-operand operations, the design accepts both operands instead of ignoring the unused one
- There is no priority handling when one operand arrives late (beyond 16 cycles) the later operand should overwrite the earlier one but does not
- The ALU does not flag an error when only one operand is present for operations that require both
- The command (CMD) is not validated against the required number of operands before execution, leading to unexpected behaviour
- Connection width does not match the declared port width for RES, causing simulation or synthesis issues

3.2 COVERAGE REPORT

Below is the coverage report obtained on verifying the ALU design.

Input Coverage:

Figure 3. shows the input coverage for the ALU



Covergroup type:

drv_cgrp

Summary	Total Bins	Hits	Hit %
Coverpoints	30	30	100.00%
Crosses	113	105	92.92%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
cin_cp	2	2	0	100.00%	100.00%	100.00%
cmd_cp	16	16	0	100.00%	100.00%	100.00%
inp_valid_cp	4	4	0	100.00%	100.00%	100.00%
mode_cp	2	2	0	100.00%	100.00%	100.00%
opa_cp	3	3	0	100.00%	100.00%	100.00%
opb_cp	3	3	0	100.00%	100.00%	100.00%

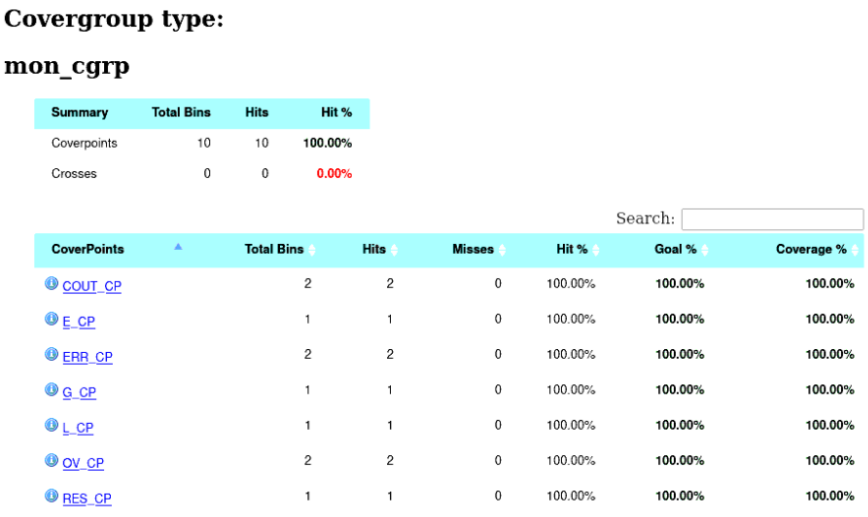
Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
cmd_x_inp_v	64	58	6	90.62%	90.62%	90.62%
mode_x_cmd	32	30	2	93.75%	93.75%	93.75%
mode_x_inp_v	8	8	0	100.00%	100.00%	100.00%
opa_x_opb	9	9	0	100.00%	100.00%	100.00%

Figure 3. Input Coverage for ALU

Output Coverage:

Figure 4. shows the output coverage for the ALU



Covergroup type:

mon_cgrp

Summary	Total Bins	Hits	Hit %
Coverpoints	10	10	100.00%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
COUT_CP	2	2	0	100.00%	100.00%	100.00%
E_CP	1	1	0	100.00%	100.00%	100.00%
ERR_CP	2	2	0	100.00%	100.00%	100.00%
G_CP	1	1	0	100.00%	100.00%	100.00%
L_CP	1	1	0	100.00%	100.00%	100.00%
OV_CP	2	2	0	100.00%	100.00%	100.00%
RES_CP	1	1	0	100.00%	100.00%	100.00%

Figure 4. Output Coverage for ALU

Assertion Coverage:

Figure 5. shows the assertion coverage for the ALU

Assertions Coverage Summary:

Search:

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
/alu_pkg/alu_sequence/body/#ublk#138110839#18/immed__20	0	50	-	-	-	-	-	Covered
/top/intrf/assert__CE_ASSERT	0	2	3487	3485	0	0	2	Covered
/top/intrf/assert__ROTATE_OP_CHECK	136	108	3487	3239	3	1	2	Failed
/top/intrf/assert__TIMEOUT_16Clk	388	0	3487	3096	3	0	17	Failed
/top/intrf/assert__VALID_INPUTS_CHECK	0	3484	3487	0	3	0	1	Covered
/top/intrf/assert__VERIFY_RESET	0	3	3487	3484	0	0	2	Covered
/work.alu_intf/assert__CE_ASSERT	0	2	3487	3485	0	0	2	Covered
/work.alu_intf/assert__ROTATE_OP_CHECK	136	108	3487	3239	3	1	2	Failed
/work.alu_intf/assert__TIMEOUT_16Clk	388	0	3487	3096	3	0	17	Failed
/work.alu_intf/assert__VALID_INPUTS_CHECK	0	3484	3487	0	3	0	1	Covered
/work.alu_intf/assert__VERIFY_RESET	0	3	3487	3484	0	0	2	Covered
/work.alu_pkg/alu_sequence/body/#ublk#138110839#18/immed__20	0	50	-	-	-	-	-	Covered

Figure 5. Assertion Coverage Summary for ALU

Overall Coverage:

Figure 6. shows the overall coverage for the ALU

Coverage Summary by Type:

Total Coverage:					48.32%	70.46%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	153	145	8	1	94.77%	99.21%
Statements	1636	765	871	1	46.76%	46.76%
Branches	1277	424	853	1	33.20%	33.20%
FEC Conditions	44	37	7	1	84.09%	84.09%
Toggles	294	273	21	1	92.85%	92.85%
Assertions	6	4	2	1	66.66%	66.66%

Figure 6. Overall Coverage for ALU

Output waveform:

Figure 7. shows the output waveform from the interface in the ALU testbench architecture

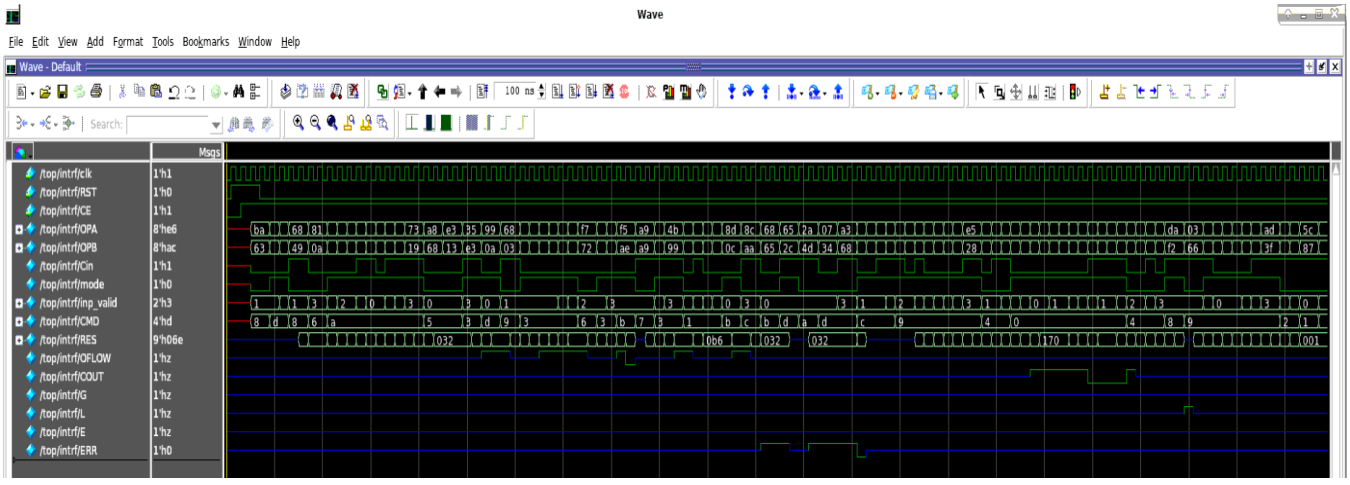


Figure 7. Output waveform from the interface in the ALU testbench architecture

CONCLUSION

The UVM based testbench for the ALU design helped identify whether the design functioned correctly. The testbench was organized into separate parts—generator, driver, monitor, reference model, and scoreboard—making it easier to test and debug each section of the design.

Several issues were found during simulation, such as incorrect handling of specific commands, missing error detection, and lack of parameterization, which reduced design flexibility. These problems were clearly visible due to the structured approach of the testbench.

FUTURE SCOPE

- The design is not fully parameterized; while DW (data width) and CW (command width) are declared as parameters, internal hardcoded values like bit widths (e.g., [8:0] for RES) should instead use parameter expressions to ensure scalability.
- The current implementation of CMD 13 performs a left-rotation instead of the required right-rotation; this must be corrected to match the expected behaviour.
- In CMD 13, when OPB [7:4] ≠ 0, the error flag (ERR) is incorrectly set to 0; it should be set to 1 to indicate an invalid rotation input, as per specification.
- The ALU does not enforce that both operands must be present (INP_VALID = 2'b11) for dual-operand commands like ADD, SUB, and MUL; operand validity checking should be enforced before processing.
- Single-operand operations such as INC_A or DEC_A do not restrict the design from accepting both operands; logic should be added to accept only the necessary operand(s).
- There is no priority logic to handle timing violations when one operand is delayed by more than 16 cycles; the later operand should overwrite the earlier one as per the requirement.
- Error handling is missing when invalid operand sequences are received (e.g., only one operand present when two are required); the design should flag an error in such cases.
- The design does not validate if the provided command (CMD) matches the expected number of operands before execution; this validation logic should be incorporated.