ALU

BHUVANESH

27bhuvanesh5@gmail.com

INTRODUCTION

Arithmetic and logical units are an integral part of any SOC that performs Arithmetic and logical operations. The ALU designed in this project supports variety of functions including arithmetic operations such as addition, subtraction, increment, decrement, and multiplication, as well as logical operations such as AND, OR, XOR, NOT, NAND, NOR, and XNOR. In addition, it supports shift and rotate operations. The design also has comparator functions and error checking for invalid command conditions.

OBJECTIVE

The objective of the project is to:

- Design a synthesizable ALU using Verilog.
- Support parameterized command, input and output widths as required.
- Perform both signed and unsigned addition and subtraction, with appropriate generation of carry-out and overflow flags.
- Execute the specified command for the given inputs with a one-cycle delay, except for the multiplication operation, which produces the result at third clock cycle.
- Ensure the error flag is raised under defined invalid conditions.

ARCHITECTURE

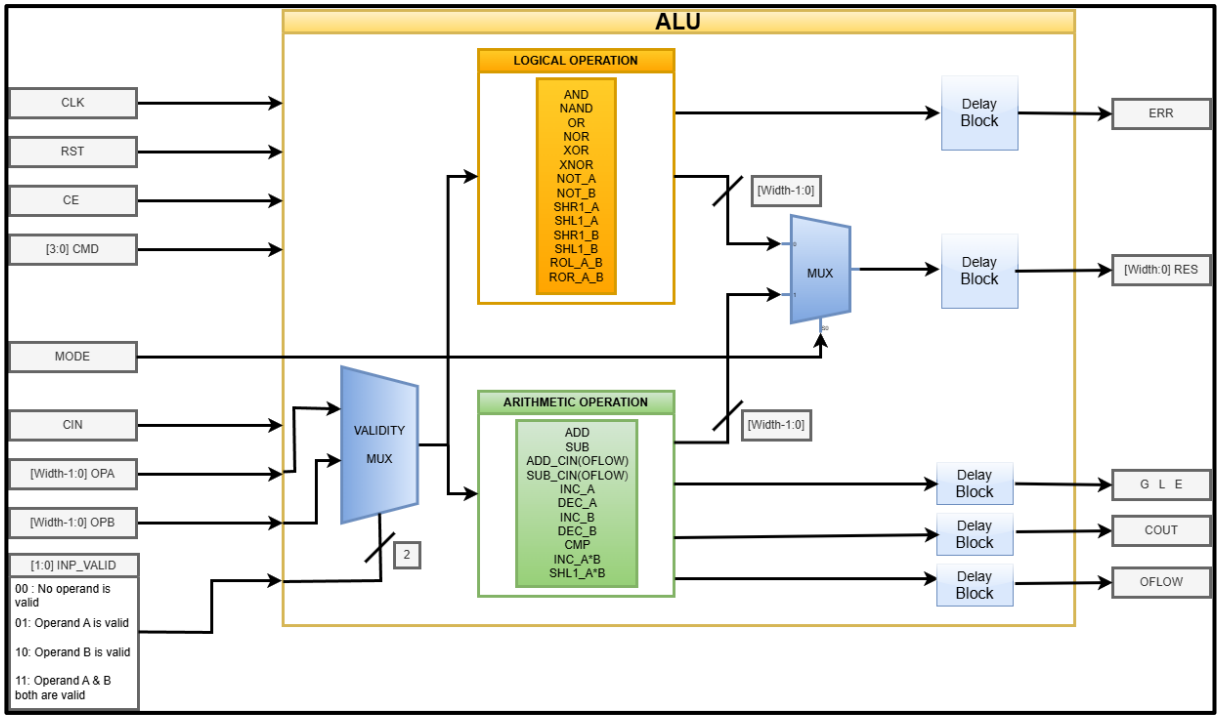

Figure 1. ALU Design Architecture

Below is the table which describes about the pins in the ALU architecture shown in Figure 1.

	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 which describes about the blocks used in the ALU architecture shown in Figure 1.

Blocks inside ALU architecture	Description
Validity MUX	This block is a 4:1 MUX which ensures that the operands are passed to the operation blocks based on the INP_VALID signal. It controls which of the input operands are used for computation
Operation Selection MUX	This block is a 2:1 MUX, based on the MODE signal, either the Logical Operation or Arithmetic Operation block result will be obtained
Logical Operation Block	This block performs 13 logical operations like: AND, OR, NAND, NOR, XOR, XNOR, NOT_A, NOT_B, SHR1_A, SHL1_A, SHR1_B, SHL1_B, ROL_A_B, ROR_A_B
Arithmetic Operation Block	This block performs 12 arithmetic operations like: ADD, SUB, ADD_CIN, SUB_CIN, INC_A, DEC_A, INC_B, DEC_B, CMP, INC_A*B, SHL1_A*B, Signed-ADD, Signed-SUB
Delay Block	This block delays the specific output as per specification

Below is the packet description for the ALU shown in Figure 2.

PACKET DESCRIPTION FOR OPERAND WIDTH = 8

PACKET HEADER	FEATURE ID	RST	INP_VALID	OPA	OPB	CMD	CIN	CE	MODE	RES_EXPECTED	COUT	GLE	OFLOW	ERR	TOTAL
BITS	8	1	2	8	8	4	1	1	1	16	1	3	1	1	56
PACKET(WIDTH)															
WIDTH	[55:48]	[47]	[46:45]	[44:37]	[36:29]	[28:25]	[24]	[23]	[22]	[21:6]	[5]	[4:2]	[1]	[0]	

PACKET_WIDTH	RES	COUT	GLE	OFLOW	ERR	FEATURE ID	RST	INP_VALID	OPA	OPB	CMD	CIN	CE	MODE	RES_EXPECTED	COUT	GLE	OFLOW	ERR	
BITS	16	1	3	1	1	8	1	2	8	8	4	1	1	1	16	1	3	1	1	78
PACKET(WIDTH)																				
WIDTH	[77:62]	[61]	[60:58]	[57]	[56]	[55:48]	[47]	[46:45]	[44:37]	[36:29]	[28:25]	[24]	[23]	[22]	[21:6]	[5]	[4:2]	[1]	[0]	

Figure 2. Packet Description

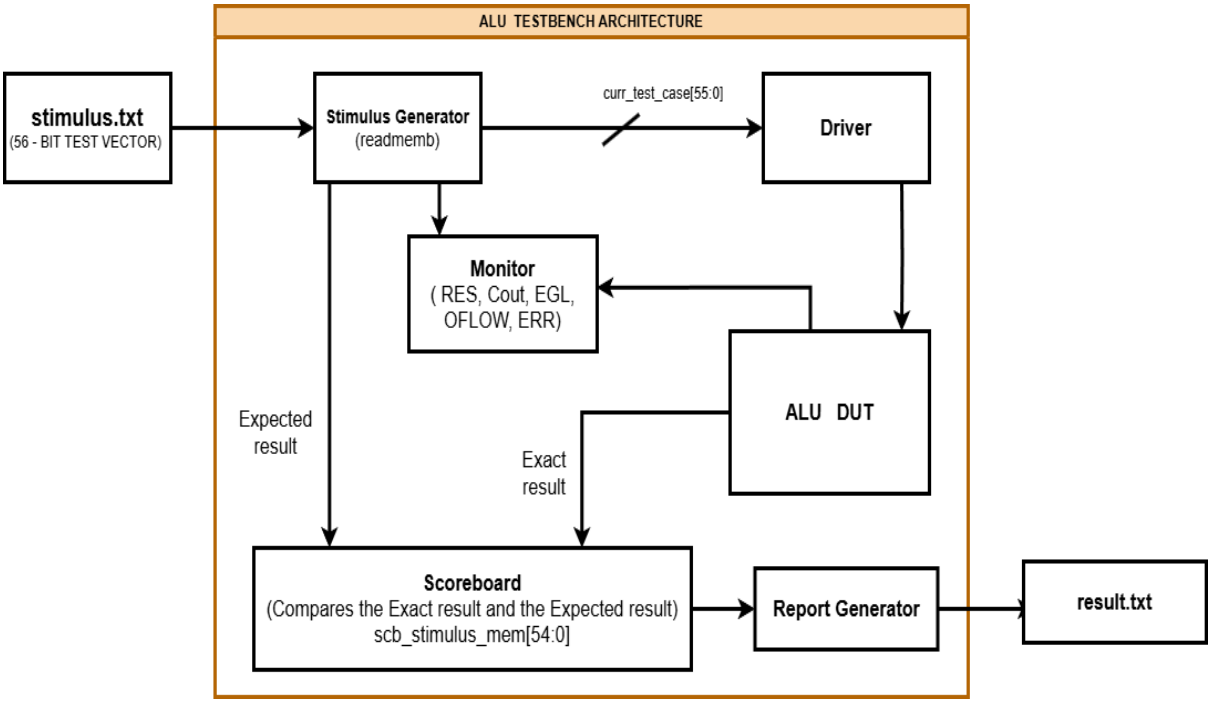

Figure 3. ALU Testbench Architecture

The ALU testbench architecture consist of as follows:

1. **stimulus.txt (56-BIT TEST VECTOR)**:
   This is an external file containing all test cases (test vectors) for the ALU. Each line represents a 56-bit input vector encoding all necessary signals for a test. Allows to easily add, remove, or modify test cases without changing the testbench code. Ensures that the same tests can be rerun for regression testing.
2. **Stimulus Generator (readmemb)** :
   Reads the stimulus.txt file using the $readmemb system task.
   Loads all test vectors into an internal memory array.
   Supplies the current test vector (curr_test_case[55:0]) to the driver.
   Converts file-based test cases into signals that can be used in simulation.
   Handles multiple test cases in a structured way.
3. **ALU DUT**
   The actual ALU hardware design being tested.
   Receives inputs from the driver and produces outputs (RES, COUT, EGL, OFLOW, ERR).
   Ensures the ALU behaves as expected for all input combinations.
4. **Driver**
   Extracts individual signals (inputs) from the current test vector.
   Drives these signals into the ALU DUT (Design Under Test) at the correct clock cycles.
   Ensures that the DUT receives the correct inputs for each test case.
5. **Monitor**
   Observes and captures outputs from the ALU DUT for each test case.
   Reads the DUT output and stores it into response packet. Combines results into exact data for comparison.
   Does not affect the DUT, only records its behaviour
6. **Scoreboard**
   Compares the actual outputs captured by the monitor with the expected outputs (from the test vector).
   Records pass/fail status for each test case in scb_stimulus_mem[54:0].
   Identifies mismatches between expected and actual results.
7. **Report Generator**
   Summarizes the results of all test cases.
   Writes a report (results.txt) indicating which test cases passed or failed.

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 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

The designed ALU operates based on the following priority scheme:

S.No.	Pin	Priority	Description
1	CLK	1	Clock toggles every cycle (CLK = ~CLK). Operations happen on the positive edge of CLK when enabled
2	RST	2	Reset signal. When RST == 1, the output register is set to zero immediately, overriding any operation
3	CLKEN	3	Clock Enable. When CLKEN == 1, the ALU performs the specified operation at the posedge of CLK. If CLKEN == 0, no operation occurs even if inputs are valid
4	INP_VALID	4	Input Validity and Operation Type: • INP_VALID == 2'b00 → Clear output register (no operation) • INP_VALID == 2'b01 or 2'b10 → Perform single-operand operation (e.g., NOT, INC) • INP_VALID == 2'b11 → Perform dual-operand operations; output is delayed by one clock cycle (except multiplication)
5	MODE	5	Operation Category Selection: • MODE == 1 → Perform Arithmetic operations (e.g., ADD, SUB, INC) • MODE == 0 → Perform Logical operations (e.g., AND, OR, NOT)
6	CMD	----	Command signal, selects the particular operation implied for the given operand

The default value given out by the ALU is as follows:

Output pins	ERR	RES [2*WIDTH – 1: 0]	G	L	E	COUT	OFLOW
Default output value	0	0	0	0	0	0	0

The designed ALU supports various arithmetic operations when the signal MODE is asserted which includes Addition, Subtraction, Addition with Carry-In, Subtraction with Borrow, Increment Operand A, Decrement Operand A, Increment Operand B, Decrement Operand B, Compare, Multiplication, Signed Addition and Signed Subtraction.

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	-
11	ADD_SIGNED	Signed Add, sets all relevant flags	RES = OPA + OPB with signed consideration	COUT, OFLOW, G, L, E
12	SUB_SIGNED	Signed Subtract, sets all relevant flags	RES = OPA - OPB with signed consideration	COUT, OFLOW, G, L, E

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

RESULTS

Below is the simulation output from the ALU when mode is set to arithmetic operation.

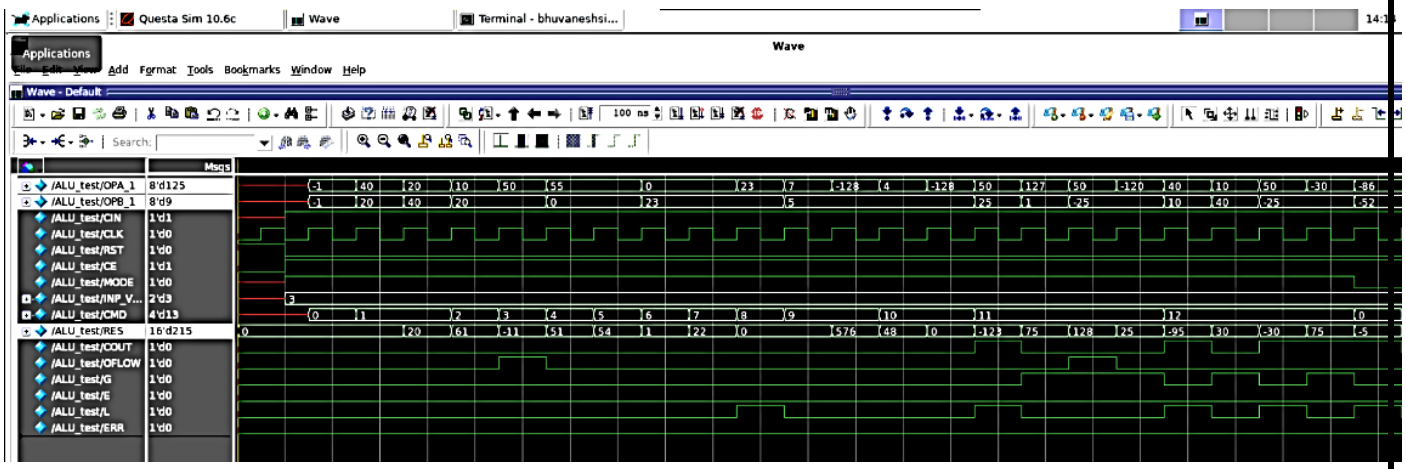

Figure 4. Simulation output of ALU for Arithmetic operation

Below is the simulation output from the ALU when mode is set to logical operation.

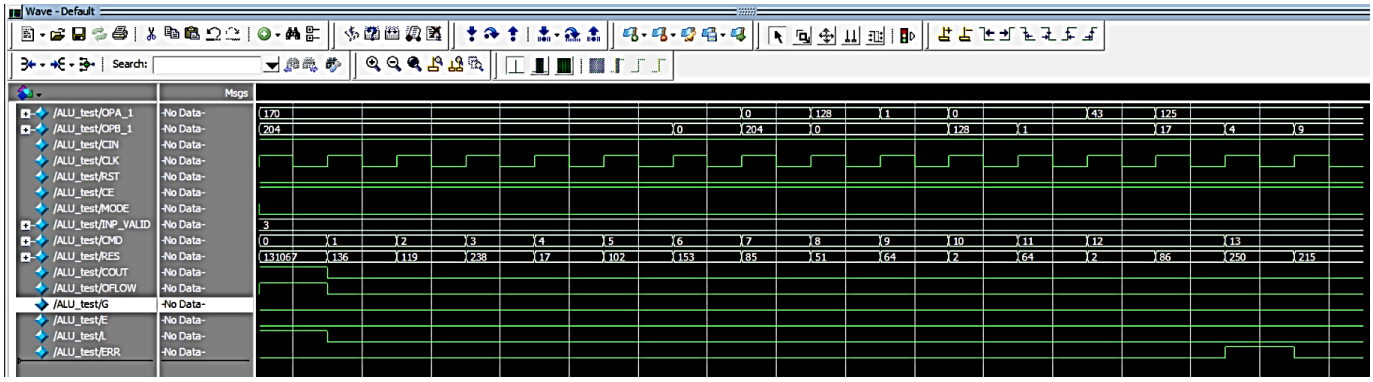

Figure 5. Simulation output of ALU for Logical operation

Below is the code coverage of the ALU shown in Figure 6.

Local Instance Coverage Details:

Total Coverage:					88.04%	**93.72%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	148	148	0	1	100.00%	**100.00%**
Branches	104	101	3	1	97.11%	**97.11%**
FEC Expressions	5	5	0	1	100.00%	**100.00%**
Toggles	270	210	60	1	77.77%	**77.77%**

Figure 6. Code coverage of ALU

Conclusion

The implementation of the ALU and its verification through a structured testbench ensures that the design meets functional requirements. By handling both arithmetic and logical operations effectively, integrating status flag management, and employing a systematic verification approach, this ALU guarantees computational accuracy, reliability, and seamless integration within digital systems. The use of a stimulus-driven testbench, golden reference models, and comparison techniques with a scoreboard mechanism ensures robust validation and detection of potential errors, improving overall design efficiency.

Future Improvement

- High-Speed Optimization: Improve architecture with pipeline execution to boost efficiency and reduce latency.
- Expanded Functionality: Add floating-point, complex number, and cryptographic operations for advanced computing.
- Automated Verification: Use AI-driven testbench generation and debugging tools to enhance accuracy and reduce development time.

By integrating these improvements, the ALU can evolve into a more powerful, efficient, and adaptive computing unit, catering to advanced digital applications while maintaining high accuracy and performance.