ALU DESIGN DOCUMENT

NAME: Varshini S

EMP ID: 6121

CONTENTS

1. INTRODUCTION

An Arithmetic Logical Unit (ALU) is used in the Central Processing Unit (CPU) of the computer to perform arithmetic and logical operations on the given inputs, based on the instructions received. The performance of the entire processor and the system it regulates is directly impacted by the effectiveness and adaptability of an ALU.

In addition to bit shifting and comparisons, the ALU can execute a large number of operations, including addition, subtraction, multiplication, division, logical AND, OR, XOR, and NOT. In both general-purpose and embedded processors, these operations serve as the cornerstone for the execution of instructions. ALUs in contemporary processors are extremely well-optimized and can accommodate extra functions like floating-point arithmetic, saturation arithmetic, and bitwise manipulation.

The implemented ALU design takes two input operands which have parametrized width with additional input control signals like MODE, CMD (command), INP_VALID (input valid), CE (clock enable) in order to generate the result along with output flags such as ERR (error), OFLOW (overflow), GEL (comparison), COUT (carry out).

2. OBJECTIVE

The objective of this project is to design and verify the functionalities of an ALU which implements the following arithmetic and logical operations (The command names are given inside the parenthesis).

- Arithmetic: Addition, Subtraction, Multiplication, Increment and Decrement
- Logical: AND, OR, XOR, NOT
- Logical shift and rotate operations
- Comparison: Equal, less than, greater than

An important aspect of this design is its **verification**. To ensure the correctness and robustness of the ALU, a **self-checking testbench** will be developed. This testbench will automatically apply a series of test vectors to the ALU and validate the output against expected results. This approach eliminates manual testing and enhances the reliability of the verification process by covering a broad range of input conditions and edge cases.

3. ARCHITECTURE

3.1 Design architecture

The design architecture of the implemented ALU is given below:

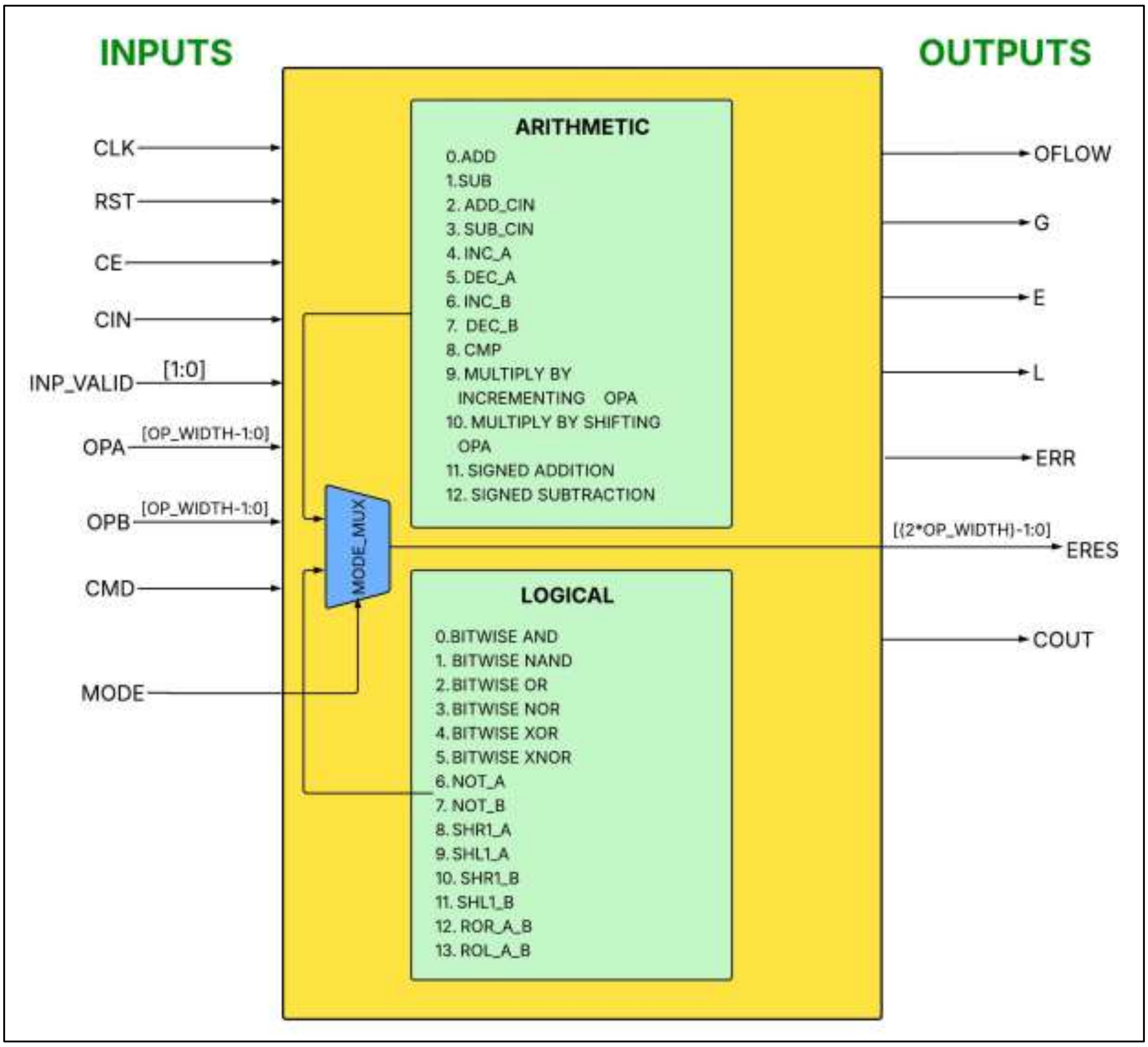


Figure 1: Design architecture of ALU

PIN LEVEL DESCRIPTION:

Serial No	Pin Name	Direction	No. of Bits	Function
1	OPA	INPUT	Parameterized (OP_WIDTH)	Parameterized operand 1
2	OPB	INPUT	Parameterized (OP_WIDTH)	Parameterized operand 2
3	CIN	INPUT	1	Active high carry-in input signal
4	CLK	INPUT	1	Clock signal to the design and it is positive edge-sensitive
5	RST	INPUT	1	Active high asynchronous reset to the design
6	CE	INPUT	1	Active high clock enable signal
7	MODE	INPUT	1	If MODE = 1, perform Arithmetic Operation; else perform Logical Operation
8	INP_VALID	INPUT	2	Indicates operand validity: 00 – No operand is valid 01 – Operand A is valid 10 – Operand B is valid 11- Both operands are valid
9	CMD	INPUT	4	Control signal which enables to choose the operation which needs to be performed.
10	RES	OUT	Parametrized (2*OP_WIDTH)	Total parameterized result of the instruction performed by the ALU.
11	OFLOW	OUT	1	This signal indicates an output overflow during Addition/Subtraction.
12	COUT	OUT	1	Carry-out signal during Addition/Subtraction.
13	G	OUT	1	The comparator output which indicates that the value of OPA is greater than the value of OPB.
14	L	OUT	1	The comparator output which indicates that the value of OPA is lesser than the value of OPB.
15	E	OUT	1	The comparator output of which indicates that the value of OPA is equal to the value of OPB.
16	ERR	OUT	1	When CMD is selected as 12 or 13 and mode is logical operation, if 4th, 5th, 6th, and 7th bit of OPB are 1, then ERR bit will be 1; else it is high impedance.

The below table describes the command value used for the arithmetic and logical operations:

COMMAND	ARITHMETIC MODE	LOGICAL MODE
0	Addition **(ADD)**	AND **(AND)**
1	Subtraction **(SUB)**	NAND (**NAND)**
2	Addition with Carry in **(ADD_CIN)**	OR **(OR)**
3	Subtraction with Carry in **(SUB_CIN)**	NOR **(NOR)**
4	Increment OPA **(INC_A)**	XOR **(XOR)**
5	Decrement OPA **(DEC_A)**	XNOR **(XNOR)**
6	Increment OPB **(INC_B)**	Complement OPA **(NOT_A)**
7	Decrement OPB **(DEC_B)**	Complement OPB **(NOT_B)**
8	Compare OPA and OPB (**CMP)**	Right shift OPA by 1 bit **(SHR1_A)**
9	Multiply by incrementing OPA by 1 (**MUL_BY_INC)**	Left shift OPA by 1 bit **(SHL1_A)**
10	Multiply by shifting OPA to left by 1 **(MUL_BY_SHIFT)**	Right shift OPB by 1 bit **(SHR1_B)**
11	Signed addition **(ADD_SIGNED)**	Left shift OPB by 1 bit **(SHL1_B)**
12	Signed subtraction **(SUB_SIGNED)**	Rotate OPA by OPB bits to the left **(ROL_A_B)**
13		Rotate OPA by OPB bits to the right **(ROR_A_B)**

3.2 Testbench architecture

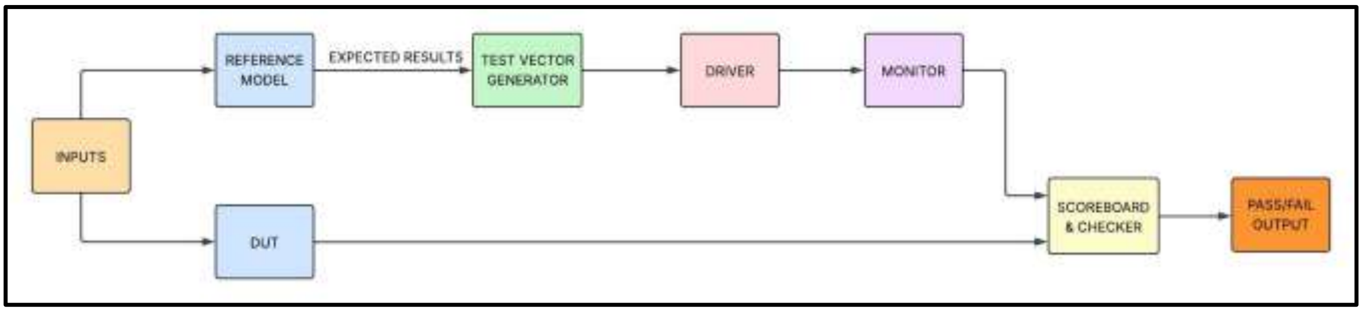


Figure 2: Testbench architecture

The following are the components of the verification testbench:

1. **Reference model:**

This is the ALU design model which generates the expected outputs with which the outputs of DUT (design under test) will be compared.

2. **Test vector generator:**

The inputs are given to the ALU in the form of a test vector and all the test vectors are loaded into the stimulus source. The structure of the test vector is given below:

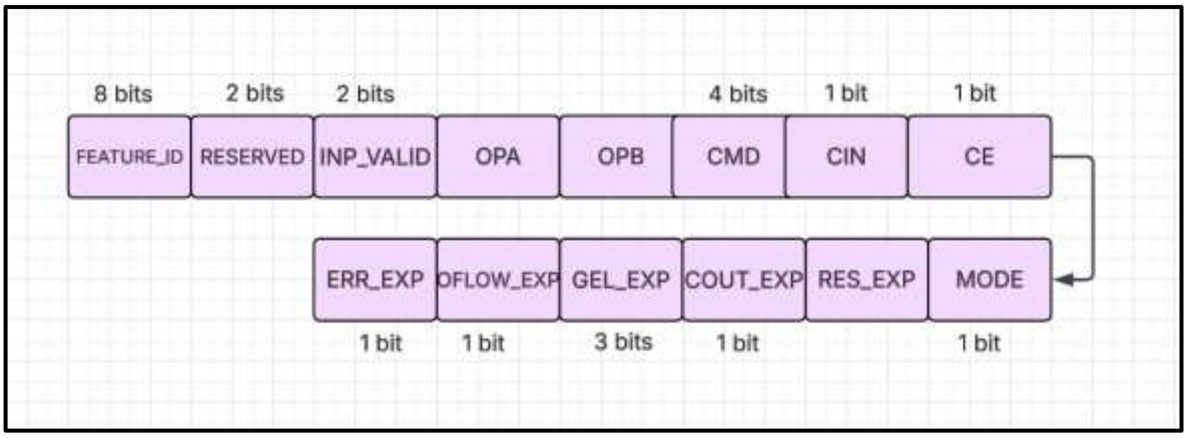


Figure 3: Test vector structure

The expected outputs generated from the reference model are included in the test vector which will be compared with the output generated from the DUT.

3. **Driver:**

Reads from the stimulus source and extracts fields from the text vector and drives the values obtained as input to the DUT as well as the reference model.

4. Monitor:

This module concatenates the test vector with the outputs obtained from the DUT to form a new vector called **response_packet.** This packet is sent to the scoreboard for the final comparison and verification.

5. Scoreboard and Checker:

This module compares the output generated from the DUT with the expected output obtained from the reference model. If both the outputs match, then the test case status is marked as PASS, else it is marked as FAIL.

4. WORKING

4.1 Working of Design:

- The design takes the inputs from the driver and perform the required operation based on the mode (MODE) and command (CMD) signals. The mode signal being low would select the arithmetic mode, else logical mode of operation will be selected.
- Further, the type of operation which needs to be performed is dictated by the command signal.
- The default value of the output registers and flags is set to low logic.
- The INP_VALID signal dictates when the inputs are valid and the operation is performed only if the operands are valid else the default value (low logic value) is assigned to the outputs.
- The outputs are generated with a delay of one clock cycle, after the inputs are given DUT. This delay logic is implemented by including addition buffers for the output registers in the design.
- The assignment of values from the buffers to the original output ports of the design is done in sequential blocks. While, all the operations performed by the ALU is carried out in combination block.
- The result register has a width of twice of that of input operands to accommodate multiplication operation.

4.2 Working of Verification testbench:

- A self-checking testbench is implemented in order to verify the outputs generated by the DUT.
- A reference model is implemented in order to verify the outputs generated by the testbench.
- The inputs to the DUT as well as the reference model are given in the form of test vectors.

- The fields from the test vector are extracted by the driver module which then drives the extracted values to the DUT and reference model.
- The expected outputs generated from the reference model are compared with the outputs of the DUT by the Scoreboard and checker module, and the pass/ fail of the testcase is decided from whether the expected outputs match the DUT outputs or not.

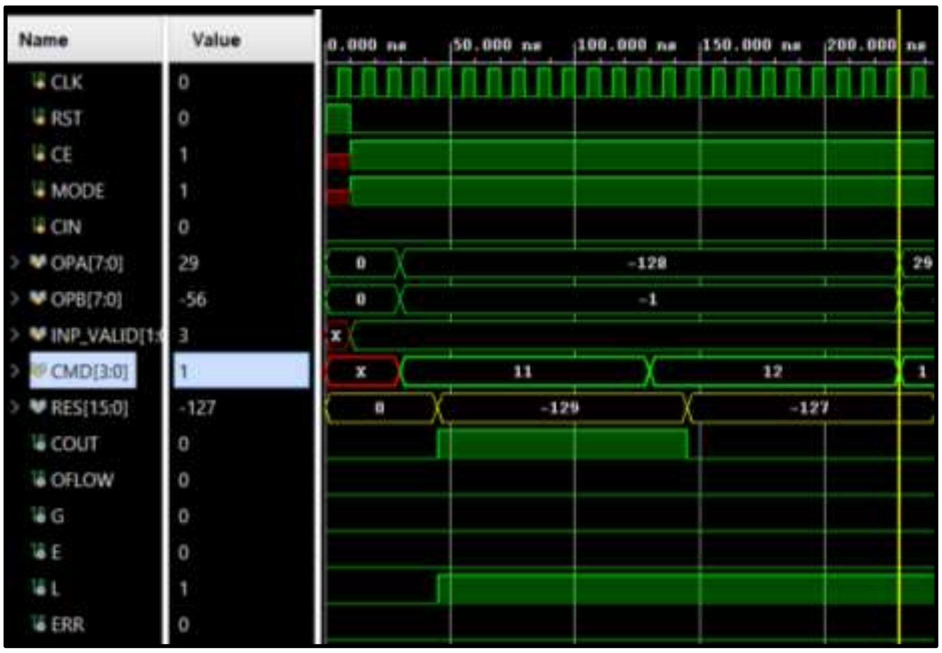
5. RESULTS

5.1 Design Results

5.1.1 Arithmetic operations

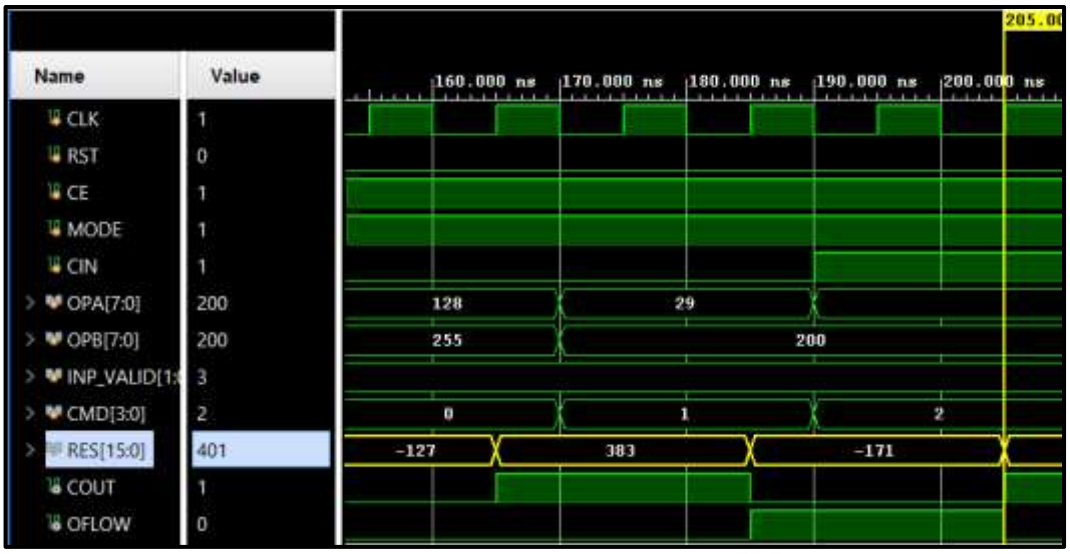
<u>Signed addition and subtraction</u>:

The waveform given below shows the signed addition and subtraction operation of the ALU. The operand values given are: OPA= -128, OPB= -1. For both operations, the operands are treated as signed numbers and the corresponding operation is performed. In addition to this, the G, E, L flags are also raised accordingly.

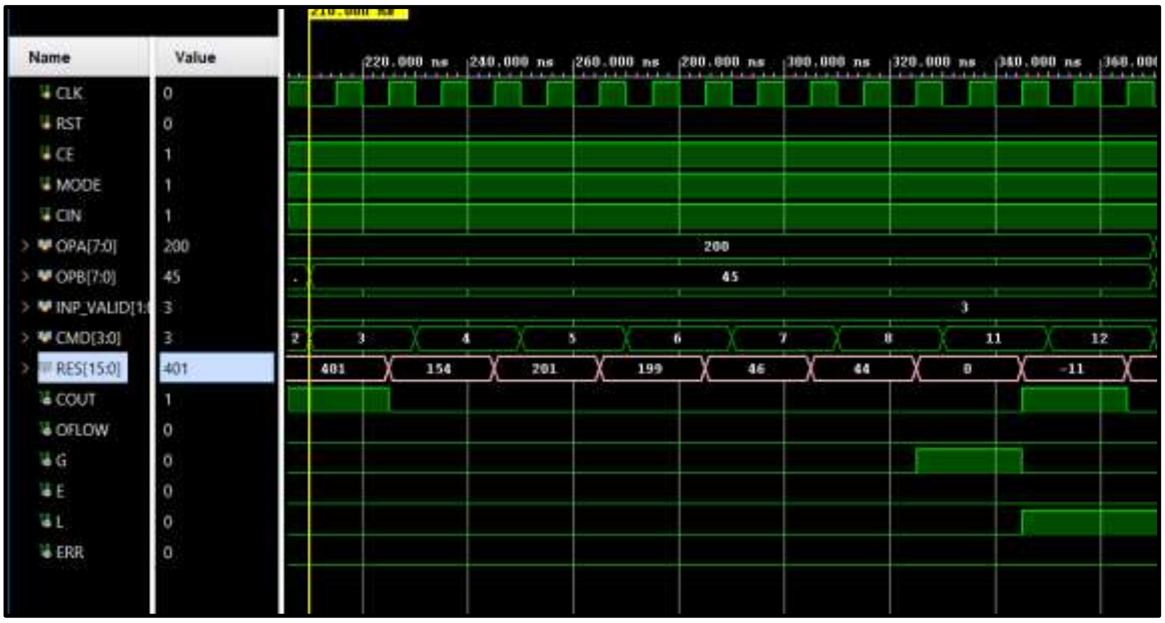


<u>Addition and Subtraction</u>:

Here, both the operands are treated as unsigned and the result is generated accordingly. The below waveform depicts the unsigned addition and subtraction:

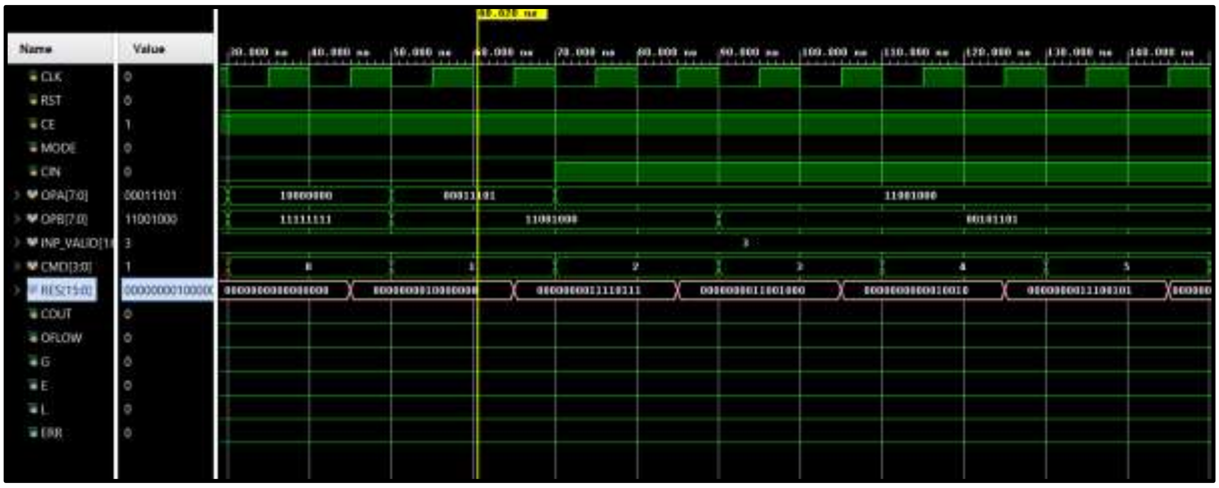


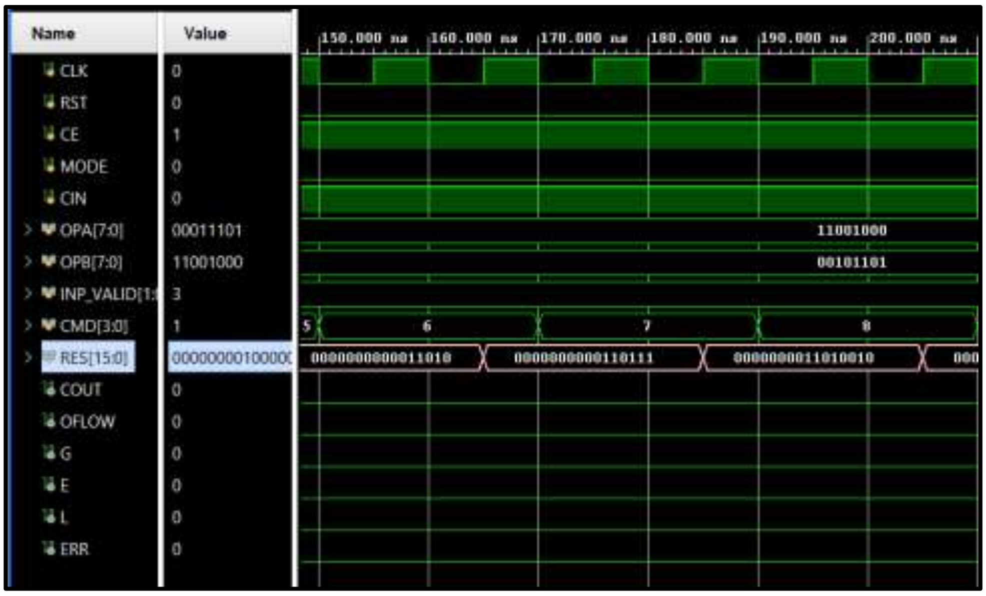
The following snippet shows the simulation results of the remaining arithmetic commands:

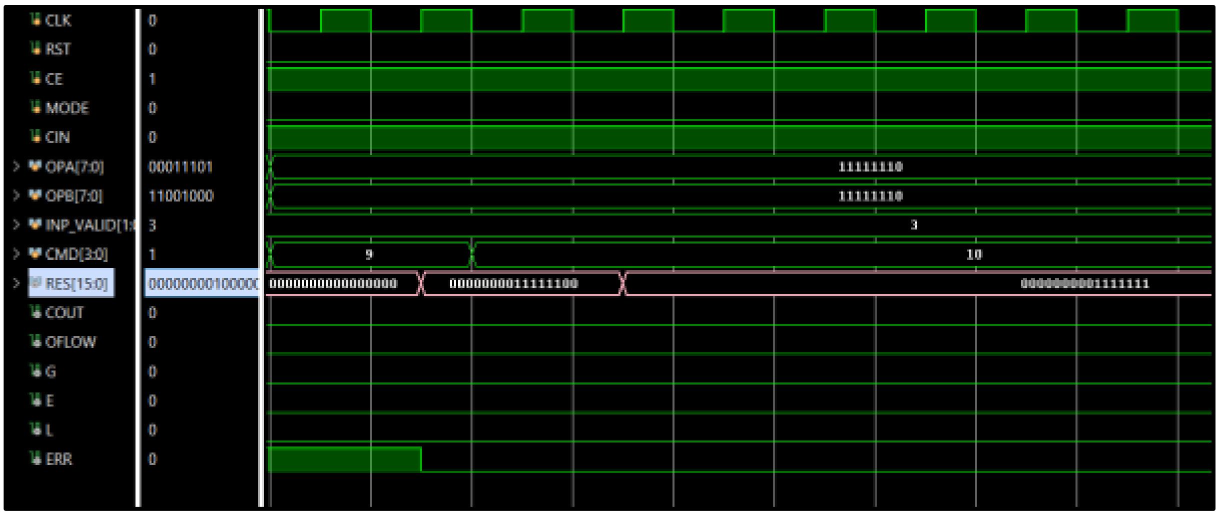


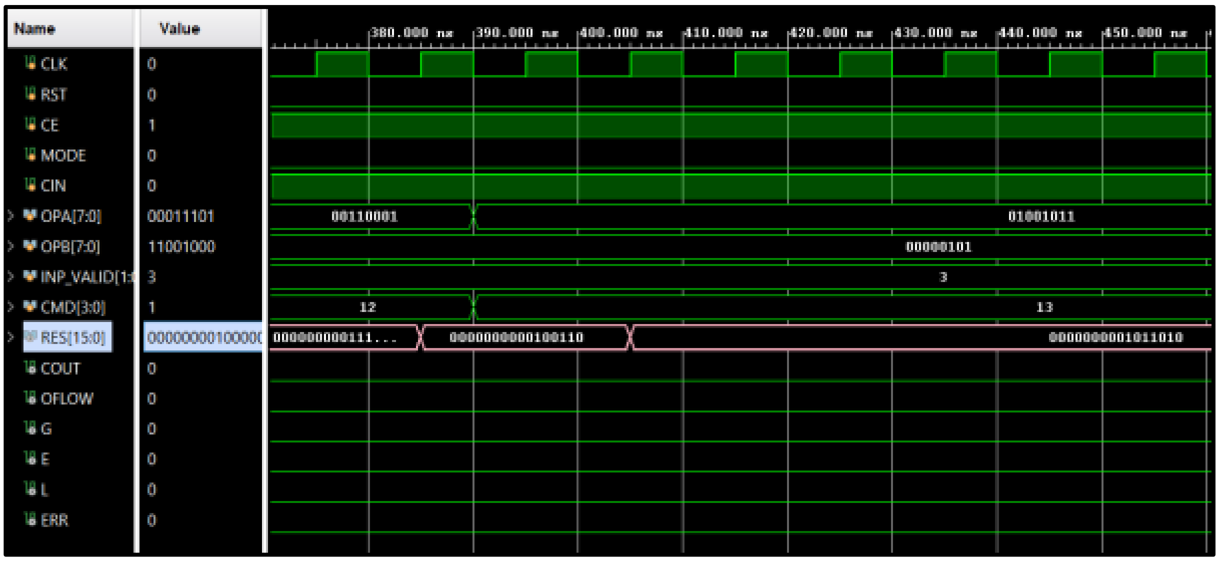
5.1.2 Logical operations

The outputs obtained from the logical operations are given below:


Name
Value
CLK 0
RST 0
CE 1
MODE 0
CIN 0
OPA[7:0] 00011101
OPB[7:0] 11001000
INP_VALID[1:0] 3
CMD[3:0] 1
RES[15:0] 000000000100000
COUT 0
OFLOW 0
G 0
E 0
L 0
ERR 0
10000000
00011101
11001000
11111111
11001000
00101101
3
0
1
2
3
4
5
0000000000000000
0000000010000000
0000000011110111
0000000011001000
0000000000010010
0000000011100101

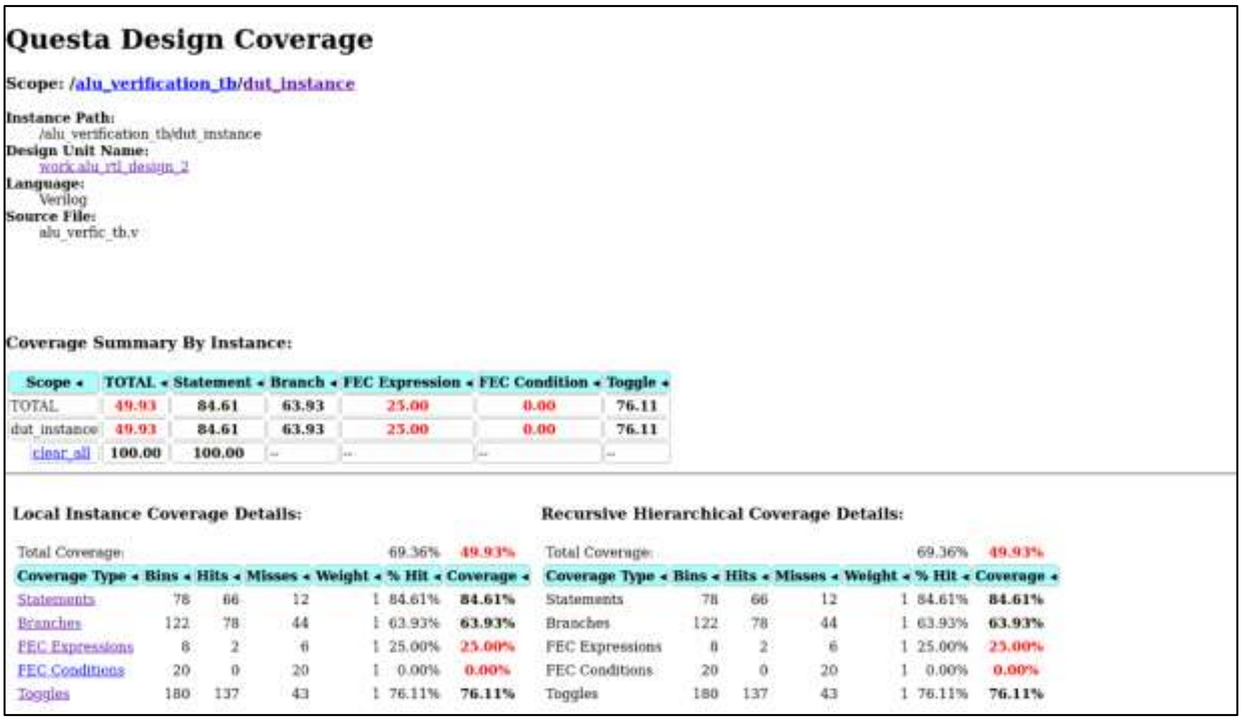

Name
Value
150.000 ns
160.000 ns
170.000 ns
180.000 ns
190.000 ns
200.000 ns
CLK 0
RST 0
CE 1
MODE 0
CIN 0
OPA[7:0] 00011101
OPB[7:0] 11001000
INP_VALID[1:0] 3
CMD[3:0] 1
RES[15:0] 000000000100000
COUT 0
OFLOW 0
G 0
E 0
L 0
ERR 0
11001000
00101101
5
6
7
8
0000000000011010
0000000000110111
0000000011010010
000


CLK 0
RST 0
CE 1
MODE 0
CIN 0
OPA[7:0] 00011101
OPB[7:0] 11001000
INP_VALID[1:0] 3
CMD[3:0] 1
RES[15:0] 000000000100000
COUT 0
OFLOW 0
G 0
E 0
L 0
ERR 0
11111110
11111110
3
9
10
0000000000000000
0000000011111100
0000000001111111



5.2 Verification Results:

The coverage report obtained from verification of ALU design is shown below. A total coverage of 49.93 % was obtained with statement, branch, FEC expression and toggle coverage as 84.61%, 63.93%, 25% and 76.11% respectively.



Questa Design Coverage

Scope: /alu_verification_tb/dut_instance

Instance Path:
/alu_verification_tb/dut_instance
Design Unit Name:
work.alu_rtl_design_2
Language:
Verilog
Source File:
alu_verfic_tb.v

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle
TOTAL	49.93	84.61	63.93	25.00	0.00	76.11
dut_instance	49.93	84.61	63.93	25.00	0.00	76.11
clear_all	100.00	100.00	--	--	--	--

Local Instance Coverage Details:

Total Coverage: 69.36% 49.93%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	78	66	12	1	84.61%	84.61%
Branches	122	78	44	1	63.93%	63.93%
FEC Expressions	8	2	6	1	25.00%	25.00%
FEC Conditions	20	0	20	1	0.00%	0.00%
Toggles	180	137	43	1	76.11%	76.11%

Recursive Hierarchical Coverage Details:

Total Coverage: 69.36% 49.93%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	78	66	12	1	84.61%	84.61%
Branches	122	78	44	1	63.93%	63.93%
FEC Expressions	8	2	6	1	25.00%	25.00%
FEC Conditions	20	0	20	1	0.00%	0.00%
Toggles	180	137	43	1	76.11%	76.11%

6. CONCLUSION

A functioning Arithmetic Logic Unit (ALU) that can execute 13 basic arithmetic and logical operations, such as addition, subtraction, bitwise logic, and comparison-based evaluations, was successfully built in this project. Using Verilog HDL, the design was created while following modular design principles, which improve reusability, scalability, and clarity.

The width of the input operands was parametrized which enhances the reusability of the design and output is generated after a delay of one clock cycle based on the control signals received. Additionally, the ALU design was verified using a self-checking testbench which made use of a reference model to generate the expected outputs.

To conclude, this ALU design shows expertise of both hardware design techniques and digital logic verification with HDL tools, and it offers a strong basis for more complex computing systems.

7. FUTURE WORK

In the future, the ALU design can be further improved by incorporating pipelining techniques to boost instruction throughput and overall performance, especially for high-speed computing environments. Another important enhancement would be the addition of signed number operation support, which would enable the ALU to handle both positive and negative operands, thereby increasing its applicability in a wider range of computational tasks. Moreover, exploring low-power design strategies, such as optimizing switching activity and reducing unnecessary logic transitions, would make the ALU more suitable for energy-efficient applications in embedded and mobile systems. These advancements would help transition the current ALU design from a functional prototype to a more robust and scalable component of a complete processor architecture.