MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

LAB PROJECT REPORT

For

ECE2244: VLSI Design Lab

8-bit Arithmetic & Logic Unit (ALU)

Submitted by

KRISHANU DEY

Reg. No.: 230959026

PARSHVA SHUBHAN PARAKH

Reg. No.: 230959028

Batch: A1

DEPT. OF ELECTRONICS AND COMMUNICATION (ECE)

MANIPAL INSTITUTE OF TECHNOLOGY, MANIPAL-576104

April, 2025

Index

OBJECTIVE

The objective of this project is to design and implement an 8-bit Arithmetic Logic Unit (ALU) using Verilog HDL. The ALU can perform arithmetic, logical, shift, and comparison operations. The design is verified through a testbench that simulates different test cases.

CHAPTER 1: INTRODUCTION

An Arithmetic Logic Unit (ALU) is a fundamental component of a processor that performs arithmetic and logic operations. It is a combinational circuit that processes input data based on a given opcode and produces an output. The designed ALU in this project supports a variety of operations such as addition, subtraction, multiplication, bitwise logic, shift operations, and comparisons. The ALU outputs a 16-bit result along with carry and borrow flags where applicable.

CHAPTER 2: PROJECT THEORY EXPLANATION

2.1 Overview of ALU Operations

- Arithmetic Operations: Addition, subtraction, multiplication, and modulo division are basic arithmetic operations. Addition and subtraction are performed using standard binary arithmetic, while multiplication is a straightforward binary multiplication. Modulo division is implemented by checking if the divisor is zero to avoid division by zero errors.
- Bitwise Shift Operations: Logical left shift, logical right shift, and arithmetic right shift are implemented. Logical shifts involve moving bits and filling the vacant positions with zeros, while arithmetic right shift fills the vacant positions with the sign bit.
- Logical Operations: NOT, OR, AND, XOR, NOR, NAND, and XNOR operations are supported. These operations are performed bitwise between the two operands.
- Comparison Operations: Greater than and equal to comparisons are implemented. These operations return a binary value indicating whether the condition is true or false.

2.2 Carry and Borrow Flags

- Carry Flag (cout): Set when there is an overflow from the most significant bit during addition.
- Borrow Flag (bout): Set when the result of subtraction is negative (i.e., borrow is required).

Opcode (4-bit)	Operation performed
0000	Addition: a + b
0001	Subtraction: a - b
0010	Multiplication: a * b
0011	Modulo Division (gives remainder): a % b
0100	Logical Shift Left on operand 1: a << 1
0101	Logical Shift Right on operand 1: a >> 1
0110	Arithmetic Shift Right on operand 1: a >>> 1
0111	NOT of operand 1: ~a
1000	OR: a | b
1001	AND: a & b
1010	XOR: a ^ b
1011	NOR: ~(a | b)
1100	NAND: ~(a & b)
1101	XNOR: ~(a ^ b)
1110	Greater than or Less than: a > b ? 1 : 0
1111	Equal: a == b ? 1 : 0

Table 1. Table of operations

It has been made sure that the 16-bit output is fully utilised only for the Multiplication operation. All other operations have zeroes padded to the first 8 significant bits. The cout and bout are taken as separate outputs with the sum and difference being represented only with the last 8 significant bits.

CHAPTER 3: VERILOG CODE AND TESTBENCH

The Verilog code for the ALU consists of:

1. **ALU Module:** Implements the arithmetic, logical, shift, and comparison operations.

```
module alu_8bit(
input [3:0] opcode, //4-bit opcode input to select operation
input [7:0] operand_a, //8-bit operand 1 input
input [7:0] operand_b, //8-bit operand 2 input
output reg [15:0] alu_out, //16-bit ALU output(to handle
multiplication)
output reg cout, //Carry flag if there's overflow from bit 7
output reg bout //Borrow flag
);
reg [8:0] temp_result;
always @(*) begin
cout=1'b0; //Default value
bout=1'b0; //Default value
case (opcode)
// Arithmetic operations
4'b0000: begin // Addition
temp_result = operand_a + operand_b;
alu_out = {8'b0, temp_result[7:0]}; //Store lower 8 bits
cout = temp_result[8]; //Carry out from bit 8
end
4'b0001: begin // Subtraction
temp_result = operand_a - operand_b; //Store lower 8 bits
alu_out = {8'b0, temp_result[7:0]}; //Borrow if MSB is 0 after
subtraction
bout = ~temp_result[8];
end
4'b0010: alu_out = operand_a * operand_b; // Multiplication
4'b0011: alu_out = (operand_b != 0) ? operand_a % operand_b : 16'd0;
// Modulo division

// Bitwise shift operations
4'b0100: alu_out = {8'b0, operand_a << 1}; // Logical Left Shift
4'b0101: alu_out = {8'b0, operand_a >> 1}; // Logical Right Shift
4'b0110: alu_out = {operand_a[7], operand_a[7:1]}; // Arithmetic
Shift Right

// Logical operations
4'b0111: alu_out = {8'b0, ~operand_a[7:0]}; // NOT
4'b1000: alu_out = {8'b0, operand_a | operand_b}; // OR
4'b1001: alu_out = {8'b0, operand_a & operand_b}; // AND
4'b1010: alu_out = {8'b0, operand_a ^ operand_b}; // XOR
4'b1011: alu_out = {8'b0, ~(operand_a | operand_b)}; // NOR
4'b1100: alu_out = {8'b0, ~(operand_a & operand_b)}; // NAND
4'b1101: alu_out = {8'b0, ~(operand_a ^ operand_b)}; // XNOR

// Comparison operations
4'b1110: alu_out = (operand_a > operand_b) ? 16'd1 : 16'd0; //
Greater than or less than
4'b1111: alu_out = (operand_a == operand_b) ? 16'd1 : 16'd0; //
Equal
default: alu_out=16'd0; //don't care
endcase
end
endmodule
```

2. **Testbench Module:** Stimulates the ALU with different opcodes and random operands to verify functionality.

```
module alu_8bit_tb();
reg [3:0] opcode;
reg [7:0] operand_a;
reg [7:0] operand_b;
wire [15:0] alu_out;
wire cout;
wire bout;

alu_8bit uut (.opcode(opcode), .operand_a(operand_a),
.operand_b(operand_b), .alu_out(alu_out), .cout(cout), .bout(bout));
//instantiating the source code
integer i; //declaring variable i to iterate over a loop to generate
opcode

initial begin
$monitor ("Time = %0t, Opcode =  %0b, Operand A = %0b, Operand B =
%0b, ALU output = %0b, Carry out = %0b, Borrow out=
%0b",$time,opcode,operand_a,operand_b,alu_out,cout,bout); //Prints
simulation results
operand_a=8'b00000000; operand_b=8'b00000000;
for (i=0;i<=15;i=i+1) //Iterating from 4'b0000 to 4'b1111
begin
opcode=i;
operand_a=$random; //Generating a random 8-bit number
operand_b=$random; //Generating a random 8-bit number
#10; //10 time units delay
end
#50 $stop; //Stop simulation
end
endmodule
```

The testbench applies a range of inputs and captures the output responses to ensure correct operation. The simulation results are monitored using $monitor statements.

CHAPTER 4: RESULTS

The results of the project include:

1. **Waveform Analysis:** The simulation waveforms show the correct output for each opcode and confirm the ALU functionality.

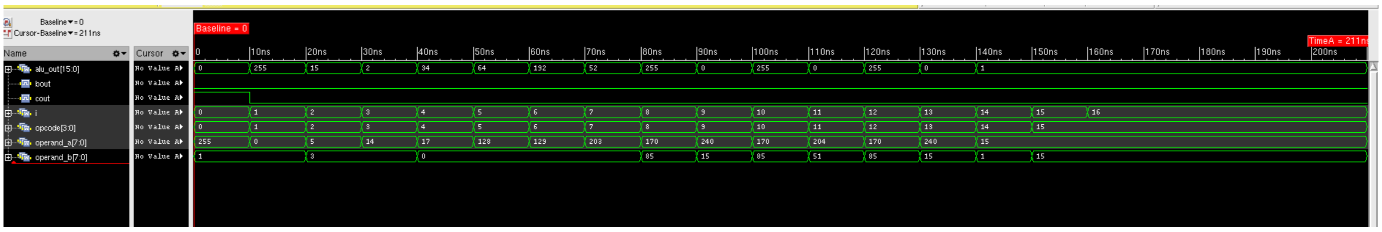
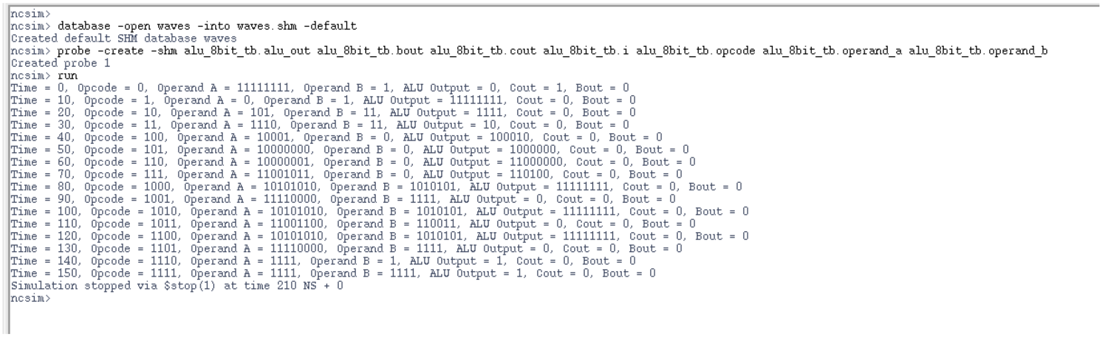


Fig 1. SimVision Waveform Results



```
ncsim>
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm alu_8bit_tb.alu_out alu_8bit_tb.bout alu_8bit_tb.cout alu_8bit_tb.i alu_8bit_tb.opcode alu_8bit_tb.operand_a alu_8bit_tb.operand_b
Created probe 1
ncsim> run
Time = 0, Opcode = 0, Operand A = 11111111, Operand B = 1, ALU Output = 0, Cout = 1, Bout = 0
Time = 10, Opcode = 1, Operand A = 0, Operand B = 1, ALU Output = 11111111, Cout = 0, Bout = 0
Time = 20, Opcode = 10, Operand A = 101, Operand B = 11, ALU Output = 1111, Cout = 0, Bout = 0
Time = 30, Opcode = 11, Operand A = 1110, Operand B = 11, ALU Output = 10, Cout = 0, Bout = 0
Time = 40, Opcode = 100, Operand A = 10001, Operand B = 0, ALU Output = 100010, Cout = 0, Bout = 0
Time = 50, Opcode = 101, Operand A = 10000000, Operand B = 0, ALU Output = 1000000, Cout = 0, Bout = 0
Time = 60, Opcode = 110, Operand A = 10000001, Operand B = 0, ALU Output = 11000000, Cout = 0, Bout = 0
Time = 70, Opcode = 111, Operand A = 11001011, Operand B = 0, ALU Output = 110100, Cout = 0, Bout = 0
Time = 80, Opcode = 1000, Operand A = 10101010, Operand B = 1010101, ALU Output = 11111111, Cout = 0, Bout = 0
Time = 90, Opcode = 1001, Operand A = 11110000, Operand B = 1111, ALU Output = 0, Cout = 0, Bout = 0
Time = 100, Opcode = 1010, Operand A = 10101010, Operand B = 1010101, ALU Output = 11111111, Cout = 0, Bout = 0
Time = 110, Opcode = 1011, Operand A = 11001100, Operand B = 110011, ALU Output = 0, Cout = 0, Bout = 0
Time = 120, Opcode = 1100, Operand A = 10101010, Operand B = 1010101, ALU Output = 11111111, Cout = 0, Bout = 0
Time = 130, Opcode = 1101, Operand A = 11110000, Operand B = 1111, ALU Output = 0, Cout = 0, Bout = 0
Time = 140, Opcode = 1110, Operand A = 1111, Operand B = 1, ALU Output = 1, Cout = 0, Bout = 0
Time = 150, Opcode = 1111, Operand A = 1111, Operand B = 1111, ALU Output = 1, Cout = 0, Bout = 0
Simulation stopped via $stop(1) at time 210 NS + 0
ncsim>
```

Fig 2. SimVision Console Results

2. **Synthesized Circuit:** The ALU is synthesized into a logic gate-level representation using the Genus synthesis tool.

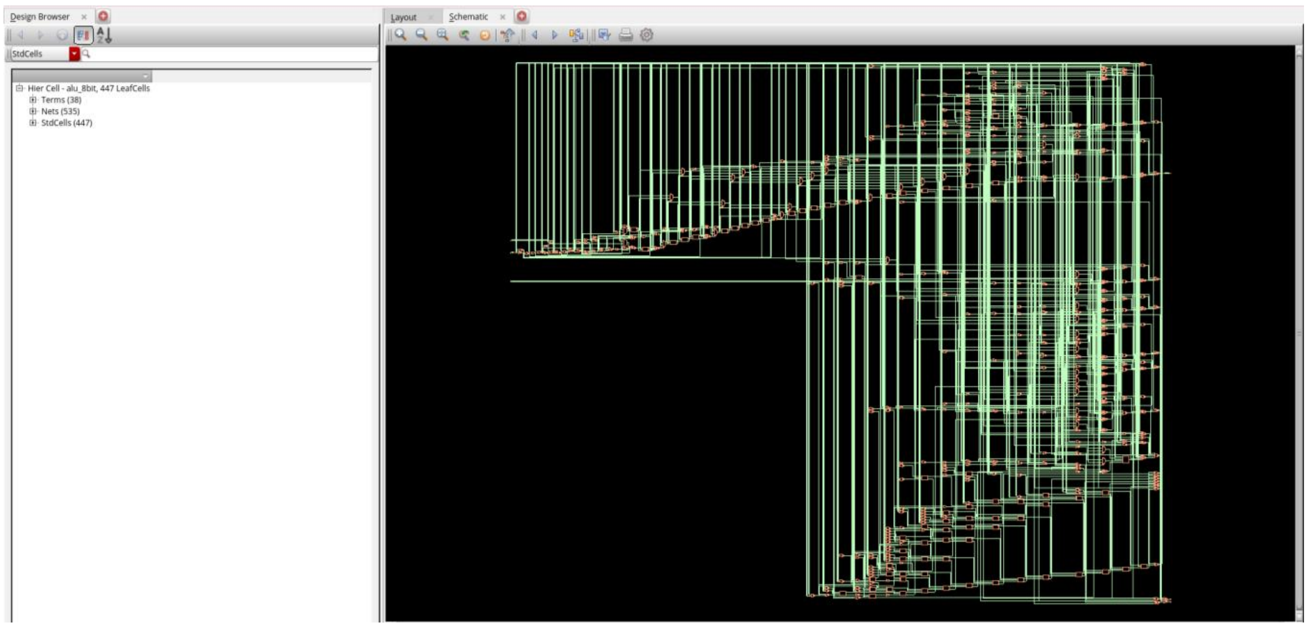


Fig 3. Synthesised schematic on Genus

3. **Layout Design:** The physical design layout is generated for ASIC implementation using the Innovus tool.

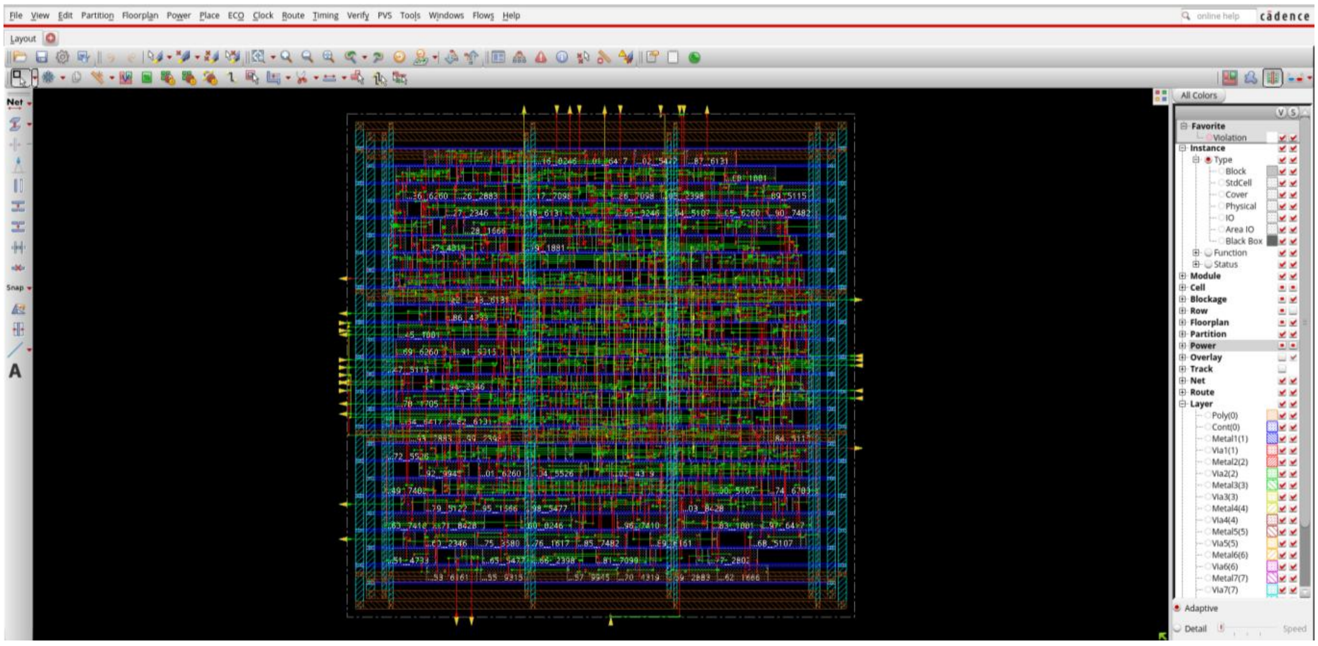


Fig 4. Layout generated on Innovus

CONCLUSION

This project successfully demonstrates the design and verification of an 8-bit ALU using Verilog. The ALU supports multiple operations and produces expected results for all test cases. The testbench effectively verifies functionality using random test vectors. This ALU design can be extended for more complex operations or integrated into larger processor architectures for further development.

REFERENCES

1. 8 Bit Arithmetic Logic Unit (ALU) Implementation in Verilog
2. Verilog code for Arithmetic Logic Unit (ALU)
3. Verilog Code for 8-Bit ALU
4. Arithmetic and Logical Unit (ALU) Verilog Code
5. GitHub repositories: https://github.com/SravanChittupalli/8-bit-ALU-in-verilog