DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

MANIPAL INSTITUTE OF TECHNOLOGY

(A Constituent Institute of MAHE, Manipal)
MANIPAL – 576104, KARNATAKA, INDIA

Department of Electronics and Communication Engineering

Digital VLSI Design LAB

Mini-Project Report

First semester, MTech – DEC

2024

ALU Design with Enhanced Capabilities: Complex Arithmetic, Modular and Bitwise Operations

Submitted by

Harshitha S	Harshith Gowda B V	Keerthana S
240938013	240938004	240938022

ABSTRACT

Advanced Microprocessors (ALUs) are integral components in digital systems as they elaborate different arithmetic and logic functions required to accomplish processing tasks in microprocessors and microcontrollers. This project concentrates on the design and realization of an improved ALU capable of executing a wide variety of operations such as basic arithmetic operations like multiplication, and division, and logical operations such as NAND, NOR, XNOR, and more advanced operations including rotate right, rotate left, modulo, power, and square root operations. The proposed ALU design aims to achieve better efficiency in computation, low power consumption, and improved processing speed by the use of algorithms and techniques for parallel processing. Also, the project applies modular design strategies to achieve scalability and flexibility in the deployment across multiple environments. To test its performance, the design is implemented on a Cadence platform after being synthesized and simulated using Verilog/VHDL for hardware description. These results show that the ALU that was advanced is also capable of meeting timing constraints while achieving high levels of accuracy, indicating potential use in high-performance digital systems.

TABLE OF CONTENTS

List of Figure

CHAPTER -1

INTRODUCTION

An Arithmetic Logic Unit (ALU) is the core computation engine within a processor, responsible for performing arithmetic and logical operations on input data. These operations are essential for executing instructions within a computer, ranging from simple mathematical calculations to complex logical decision-making processes. The ALU is controlled by signals from the processor, which dictate the specific operation to be performed on given input data. By quickly and efficiently processing these instructions, the ALU enables the execution of all higher-level computations in a CPU, making it a critical element in both general-purpose CPUs and specialized digital processors, such as those in embedded systems.

Traditionally, an ALU includes basic arithmetic operations like addition and subtraction, logical operations such as AND, OR, and XOR, and bitwise shifts that enable manipulation of data at the binary level. In more advanced designs, the ALU may also handle operations such as division, multiplication, and even complex calculations involving trigonometric or exponential functions. To handle a wider range of computations, modern ALUs are increasingly designed with modular components, each specializing in a specific operation. This modularity allows for greater flexibility and scalability, enabling the ALU to support more sophisticated processing tasks. With a well-structured design, an ALU can be both highly efficient and versatile, catering to the demands of various applications across computing fields.

The ALU design provided here is an advanced implementation that expands on traditional ALU functionality by incorporating specialized modules for a diverse set of operations. This design includes standard arithmetic and bitwise functions, as well as more complex operations like modular arithmetic, population counting, and both left and right rotations. Additionally, the ALU supports complex number arithmetic, allowing it to perform addition and multiplication on complex numbers by breaking them down into their real and imaginary components. Each operation is modularized and controlled by a 4-bit operation selector, ensuring that the ALU can quickly switch between tasks based on the input control signals. This approach allows the ALU to serve a broad array of computational needs, from general-purpose calculations to specialized tasks that require complex arithmetic and bit manipulation.

CHAPTER -2

2.1 LITRATURE SURVEY

"Design of Logically Obfuscated n-bit ALU for Enhanced Security" (2019) focuses on improving ALU security through logic obfuscation techniques, which make the ALU more resilient to reverse engineering and hardware attacks. The study examines how obfuscation techniques impact performance and security metrics in n-bit ALUs.

"Design and Optimization of 8-bit ALU Using Reversible Logic" explores reversible logic to enhance energy efficiency in ALU designs, with applications in low-power devices and quantum computing. This approach reduces power dissipation by avoiding loss of information during computation, which is especially beneficial in high-performance, low-power circuits.

"Efficient Design of Low Power ALU Using Clock Gating Techniques" (2020) presents a low-power ALU design that leverages clock gating to minimize power consumption. Clock gating reduces unnecessary switching activities, thus saving energy while maintaining computational accuracy. This is relevant for mobile and IoT devices where energy efficiency is critical.

"High-Speed Arithmetic Logic Unit Design Using Approximate Computing" (2021) investigates approximate computing methods to achieve high speed and lower power consumption in ALUs. By introducing slight inaccuracies, this approach can significantly increase speed and efficiency, making it suitable for applications like image and video processing where exact precision is not always necessary.

"Design of Energy-Efficient Arithmetic Logic Unit with Hybrid Adder-Subtractor" (2022) explores a hybrid adder-subtractor approach for improved energy efficiency and speed. This design minimizes delay and power consumption, which is beneficial for applications requiring fast arithmetic operations, such as digital signal processing.

"Adaptive ALU Design for Variable Precision Operations" (2023) introduces an ALU capable of adjusting its precision dynamically based on workload

requirements. This adaptive design conserves energy by processing lower-precision data with fewer resources, thus optimizing power usage without compromising performance when full precision is needed.

"Fault-Tolerant ALU for Reliable High-Performance Computing" (2021) addresses fault tolerance in ALU design, crucial for ensuring reliability in mission-critical applications. The design includes error-detection and correction mechanisms that make it resilient against transient faults, which can occur in high-performance and high-radiation environments.

"Design of ALU Using Memristor-Based Logic" (2020) explores the integration of memristors into ALU circuits to create non-volatile, energy-efficient logic gates. Memristor-based designs retain their state even when power is removed, which is beneficial for low-power and embedded applications where data persistence is required.

"High-Performance ALU Design Using Quantum-Dot Cellular Automata (QCA)" (2021) looks at quantum-dot cellular automata as a promising technology for future high-speed and low-power ALU designs. QCA-based ALUs operate with extremely low power and are projected as a replacement for traditional CMOS technology in certain high-performance computing applications.

"Approximate Multiplier-Based ALU Design for Neural Network Applications" (2022) applies approximate multipliers in an ALU to accelerate neural network computations. This approach achieves higher speeds with lower energy consumption, which is beneficial in AI and machine learning applications that involve heavy computation but can tolerate small inaccuracies.

2.2 OBJECTIVE

To design and implement a versatile Arithmetic Logic Unit (ALU) that performs both standard arithmetic/logic operations and complex arithmetic operations, making it appropriate for use in high-performance computing, embedded systems, and digital signal processing.

2.3 PROBLEM STATEMENT

Processors that can execute complicated arithmetic for signal analysis, filtering, and transformations in addition to ordinary arithmetic and logic operations are necessary for modern embedded systems and digital signal processing applications. Complex number operations are frequently not supported by current ALUs, requiring additional hardware or software emulation, which raises latency and decreases efficiency.
The goal of this project is to create a custom ALU that combines conventional operations like modulo, rotation, population count, and square root computations with complex arithmetic functions like addition and multiplication of complex numbers. Suitable for specialized processors used in image processing, communication systems, and control systems that need high-speed complex computations, this multi-function ALU offers a compact solution to streamline arithmetic-heavy computations.

By implementing and testing this ALU, we aim to achieve:

1. A reduction in latency for complex arithmetic operations.
2. Enhanced versatility by supporting both integer-based and complex arithmetic within a single ALU unit.
3. A modular design that can be adapted or extended for other specialized operations as needed in future applications.

CHAPTER -3

PROPOSED SOLUTION AND METHODOLOGY

A multi-functional Arithmetic Logic Unit (ALU) built in Verilog that can execute a variety of complicated and integer arithmetic operations is at the heart of the solution. This versatile ALU is designed for applications that need to handle complex numbers and population counts efficiently in addition to traditional arithmetic and logical operations. The ALU is appropriate for digital signal processing (DSP), embedded systems, and scientific computing since it can rotate bits, carry out modular and division operations, compute square roots, and carry out logical operations like NAND, NOR, and XNOR.

Applications that need real-time, high-performance computations with constrained hardware resources should pay special attention to this architecture. The technique cuts down on computation time and eliminates the need for external modules by incorporating sophisticated arithmetic operations straight within the ALU. It is also useful for machine learning, cryptography, and error-correcting applications since it incorporates population counting, which counts the number of 1s in a binary representation. Overall, the suggested ALU design improves computational efficiency and adaptability for a range of digital processing applications by providing a small and effective solution for multipurpose arithmetic operations.

3.1 METHODOLOGY

3.1.1 Rotate Left (ROL) and Rotate Right (ROR) Modules

Inputs: Both ROL and ROR modules accept a 32-bit input (in) and a 5-bit shift value (shift).

Outputs: They produce a 32-bit rotated output (out).

Operation: The ROL module fills in the vacant bits with the bits that were moved out from the right side after shifting the input's bits to the left by the designated shift amount. On the other hand, ROR moves bits to the right, using the bits that are pushed out from the right to fill in the left. This feature is particularly helpful for jobs involving bit manipulation and cryptography.

3.1.2 Population Count (POPCNT) Module

Inputs: Takes a single 32-bit input (in).
Output: Outputs a 6-bit result (count), which represents the count of '1' bits in the input.
Operation: The module adds up each bit in the 32-bit input separately to determine the population count. Applications where the Hamming weight (number of set bits) is crucial, such as machine learning and error detection, benefit from this process.

3.1.3 Modulo, Multiply, and Divide Modules

Inputs: Each module receives two 32-bit inputs (in_a and in_b).
Outputs: They output a 32-bit result (out or quotient).
Operation:
Modulo: Often utilized in cyclic algorithms and hashing functions, it calculates the remainder of in_a divided by in_b.
Multiply: This function multiplies in_a and in_b, which is essential for operations involving intricate transformations or scaling.
Divide: Produces the quotient by dividing in_a by in_b; helpful for DSP and control algorithms.

3.1.4 Logical Operations: NAND, NOR, and XNOR

Inputs: Each module takes two 32-bit inputs (in_a and in_b).
Output: Each outputs a 32-bit result (out).
Operation:
NAND: Applies a bitwise NAND to the inputs, generating an output in which each bit represents the negated outcome of an AND operation between the corresponding bits of in_a and in_b.
NOR: Performs a bitwise NOR in which the negated OR of in_a and in_b is represented by each bit in the output.
XNOR: Only when matching bits in in_a and in_b match does it produce a bitwise XNOR, with bits set to 1. In circuits like DSP that need to use the least amount of power possible, these logical processes are crucial.

3.1.5 Square Root (SquareRoot) Module

Input: Accepts a 32-bit input (in).
Output: Produces a 32-bit square root value (out).
Operation: Approximates the input's integer square root via an iterative bitwise technique. When real-time square root data are needed for geometrical and graphic computations, this function is quite helpful.

3.1.6 Complex Arithmetic Module

Inputs: The Complex_Arithmetic module takes four 32-bit signed inputs representing the real and imaginary parts of two complex numbers (real_a, imag_a, real_b, imag_b) and a 2-bit control signal (op).
Outputs: It produces two 32-bit signed outputs (real_out and imag_out) for the resulting complex number.
Operation: Complex numbers can be added or multiplied by the module, depending on the value of op. In applications like signal processing, where complicated numerical manipulation is commonly needed, this capacity is essential.

3.1.7 Control Logic and Output Assignment

Operation: A clock signal drives the synchronous operation of this code segment. It chooses the right operation for logical and conventional arithmetic operations using a 4-bit op signal, and it chooses the operation in the complex arithmetic module using a 2-bit complex_op signal. To provide effective output handling across several processes, the chosen result is saved in the out, real_out, or imag_out registers**.**

In order to handle a variety of difficult jobs while maintaining resource efficiency, each module in this ALU design is specifically designed to carry out a certain kind of arithmetic or logical operation efficiently. In embedded systems, DSP, and other applications where multi-functional arithmetic and logic processing are crucial, this paradigm facilitates high-performance computing.

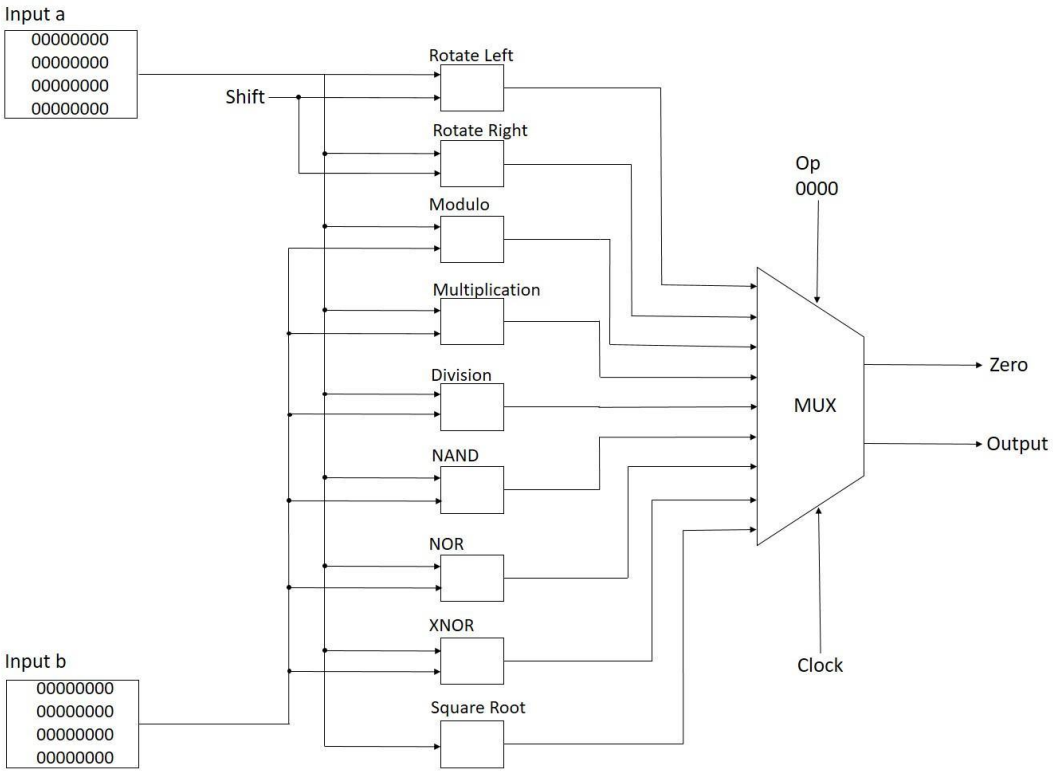


Fig 3.1 Block diagram

3.2 DESIGN FLOW

Given flowchart illustrates the structure of an ALU (Arithmetic Logic Unit) designed to handle various computational tasks through specialized modules. The ALU begins with input processing, where it receives a clock signal for timing, data inputs (in_a, in_b, real_a, imag_a), and control signals (op for operation selection and complex_op for complex arithmetic). The control logic section features an Operation Selector that directs data to the correct module based on the operation code, allowing only the required module to activate and conserving resources.

The operations section includes general modules for basic tasks such as bitwise rotation (left and right), modulo, multiplication, division, and logical operations like NAND, NOR, XNOR, as well as a square root module. For complex arithmetic, the ALU has a dedicated module for handling real and imaginary

components, enabling addition and multiplication based on the complex_op setting. Additionally, a Population Count (POPCNT) module counts the number of '1's in the binary form of an input, which is useful in fields like data analysis and encryption.

Finally, the ALU includes output processing to route results to their designated outputs: out for general results, real_out and imag_out for complex operations, and popcnt for the population count result. This design allows the ALU to efficiently perform a variety of operations by directing each task to a dedicated module, making it highly versatile for diverse computational needs.

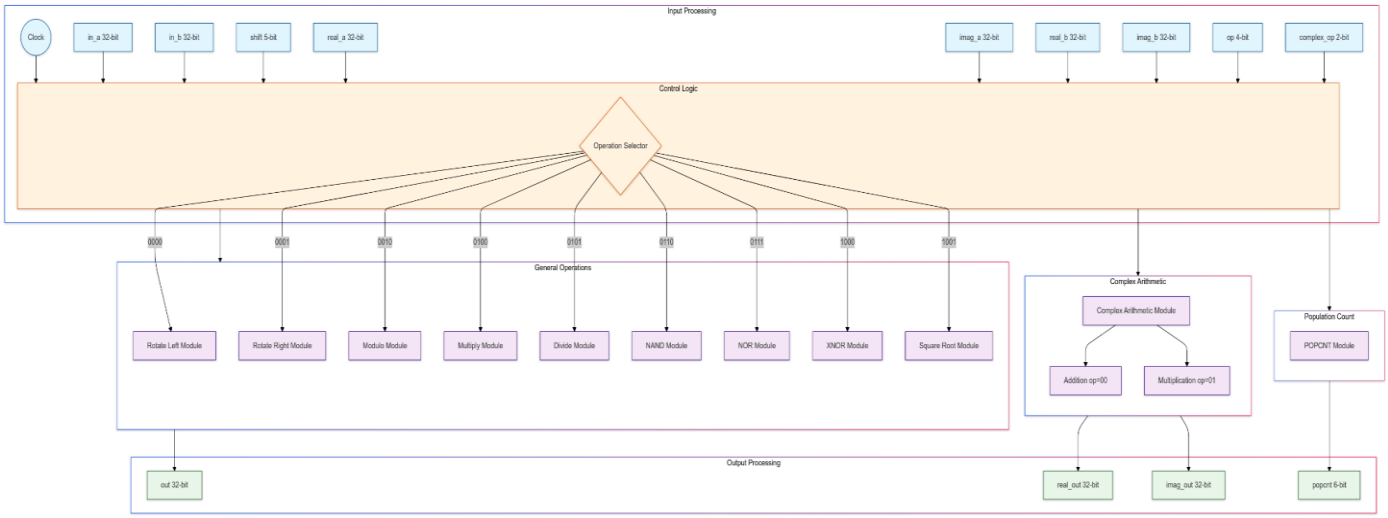


Fig 3.2 Flowchart

3.3 VERILOG CODE

3.3.1 Design Code

```
module ALU (
    input clk,
    input [31:0] in_a, in_b,          // Inputs for general operations
    input [4:0] shift,                // Shift amount for rotate
    input [31:0] real_a, imag_a,      // Inputs for complex number operations
    input [31:0] real_b, imag_b,
```

```
// Shift modules
module ROL(input [31:0] in, input [4:0] shift, output [31:0] out);
    assign out = (in << shift) | (in >> (32 - shift));
endmodule

module ROR(input [31:0] in, input [4:0] shift, output [31:0] out);
```

```
    input [1:0] complex_op,          // Operation selector for complex arithmetic
    input [3:0] op,                  // ALU operation selector
    output reg signed [31:0] out,    // Signed output to handle negative numbers
    output reg [31:0] real_out, imag_out, // Complex arithmetic outputs
    output reg [5:0] popcnt          // Output of population count
);

    wire [31:0] rol_out, ror_out;
    wire [31:0] mod_out, mul_out, div_out, nand_out, nor_out, xnor_out, sqrt_out;
    wire signed [31:0] comp_real_out, comp_imag_out;
    wire [5:0] popcnt_wire;

    // Instantiate modules
    ROL rol_inst(.in(in_a), .shift(shift), .out(rol_out));
    ROR ror_inst(.in(in_a), .shift(shift), .out(ror_out));
    POPCNT popcnt_inst(.in(in_a), .count(popcnt_wire));
    Modulo mod_inst(.in_a(in_a), .in_b(in_b), .out(mod_out));
    Multiply mul_inst(.in_a(in_a), .in_b(in_b), .out(mul_out));
    Divide div_inst(.in_a(in_a), .in_b(in_b), .quotient(div_out));
    NAND_Operation nand_inst(.in_a(in_a), .in_b(in_b), .out(nand_out));
    NOR_Operation nor_inst(.in_a(in_a), .in_b(in_b), .out(nor_out));
```

```
    assign out = (in >> shift) | (in << (32 - shift));
endmodule

// Population count (POPCNT)
module POPCNT(input [31:0] in, output [5:0] count);
    assign count = in[0] + in[1] + in[2] + in[3] + in[4] + in[5] +
                   in[6] + in[7] + in[8] + in[9] + in[10] + in[11] +
                   in[12] + in[13] + in[14] + in[15] + in[16] + in[17] +
                   in[18] + in[19] + in[20] + in[21] + in[22] + in[23] +
                   in[24] + in[25] + in[26] + in[27] + in[28] + in[29] +
                   in[30] + in[31];
endmodule

// Arithmetic operations
module Modulo(input [31:0] in_a, input [31:0] in_b, output [31:0] out);
    assign out = in_a % in_b;
endmodule

module Multiply(input [31:0] in_a, input [31:0] in_b, output [31:0] out);
    assign out = in_a * in_b;
endmodule

module Divide(input [31:0] in_a, input [31:0] in_b, output [31:0] quotient);
    assign quotient = in_a / in_b;
endmodule

// Logic operations
module NAND_Operation(input [31:0] in_a, input [31:0] in_b, output [31:0] out);
    assign out = ~(in_a & in_b);
```

```
    XNOR_Operation
xnor_inst(.in_a(in_a), .in_b(in_b),
.out(xnor_out));
    SquareRoot sqrt_inst(.in(in_a),
.out(sqrt_out));
    Complex_Arithmetic comp_inst(
        .real_a(real_a),
.imag_a(imag_a),
        .real_b(real_b),
.imag_b(imag_b),
        .op(complex_op),
        .real_out(comp_real_out),
.imag_out(comp_imag_out)
    );

    always @(posedge clk) begin
        // Simplified combinational logic
without case
        if (op == 4'b0000) begin
            out <= rol_out;          //
Rotate Left
        end else if (op == 4'b0001)
begin
            out <= ror_out;          //
Rotate Right
        end else if (op == 4'b0010)
begin
            out <= mod_out;          //
Modulo
        end else if (op == 4'b0100)
begin
            out <= mul_out;          //
Multiplication
        end else if (op == 4'b0101)
begin
            out <= div_out;          //
Division
        end else if (op == 4'b0110)
begin
            out <= nand_out;         //
NAND
        end else if (op == 4'b0111)
```

```
endmodule

module NOR_Operation(input [31:0]
in_a, input [31:0] in_b, output [31:0]
out);
    assign out = ~(in_a | in_b);
endmodule

module XNOR_Operation(input
[31:0] in_a, input [31:0] in_b, output
[31:0] out);
    assign out = ~(in_a ^ in_b);
endmodule

// Square root calculation using
bitwise method (optimized)
module SquareRoot(input [31:0] in,
output reg [31:0] out);
    reg [31:0] temp;

    always @(*) begin
        out = 0;
        temp = 0;

        if ((out | 16) * (out | 16) <= in)
            out = out | 16;
        if ((out | 8) * (out | 8) <= in)
            out = out | 8;
        if ((out | 4) * (out | 4) <= in)
            out = out | 4;
        if ((out | 2) * (out | 2) <= in)
            out = out | 2;
        if ((out | 1) * (out | 1) <= in)
            out = out | 1;
    end
endmodule

// Complex arithmetic operations
module Complex_Arithmetic(
    input signed [31:0] real_a, imag_a,
real_b, imag_b,
    input [1:0] op,
```

```
begin
        out <= nor_out;         // NOR
    end else if (op == 4'b1000)
begin
        out <= xnor_out;        //
XNOR
    end else if (op == 4'b1001)
begin
        out <= sqrt_out;        //
Square Root
    end else begin
        out <= 32'h00000000;    //
Default to zero for undefined
operations
    end

    // Handle complex arithmetic
    real_out <= comp_real_out;
    imag_out <= comp_imag_out;

    // Handle population count
    popcnt <= popcnt_wire;
  end
endmodule
```

```
  output reg signed [31:0] real_out,
imag_out
);
  always @(*) begin
    // Implementing without case
statements
    if (op == 2'b00) begin  //
Addition
      real_out = real_a + real_b;
      imag_out = imag_a +
imag_b;
    end else if (op == 2'b01) begin
// Multiplication
      real_out = real_a * real_b -
imag_a * imag_b;
      imag_out = real_a * imag_b
+ imag_a * real_b;
    end else begin  // Default to zero
      real_out = 0;
      imag_out = 0;
    end
  end
endmodule
```

3.3.2 Test bench

```
// Code your testbench here
// or browse Examples
module ALU_Testbench;
  // Testbench inputs
  reg clk;
  reg [31:0] in_a, in_b;
  reg [4:0] shift;
  reg [31:0] real_a, imag_a, real_b,
imag_b;
  reg [1:0] complex_op;
  reg [3:0] op;

  // Testbench outputs
```

```
 // Test case for NOR
    op = 4'b0111;         // NOR
operation
    in_a = 32'd12;        // Binary:
00000000000000000000000000001100
    in_b = 32'd3;         // Binary:
00000000000000000000000000000011 (expected result: -16)
    #10;
    $display("NOR Operation");
    $display("in_a=%d, in_b=%d,
out=%d (expected: -16)", in_a, in_b,
```

```
    wire signed [31:0] out;   // Signed output to handle negative numbers
    wire [31:0] real_out, imag_out;
    wire [5:0] popcnt;

    // Instantiate the ALU
    ALU uut(
.clk(clk),
        .in_a(in_a),
        .in_b(in_b),
        .shift(shift),
        .real_a(real_a),
        .imag_a(imag_a),
        .real_b(real_b),
        .imag_b(imag_b),
        .complex_op(complex_op),
        .op(op),
        .out(out),
        .real_out(real_out),
        .imag_out(imag_out),
        .popcnt(popcnt)
    );

    // Generate a clock signal
    always begin
#5 clk =~clk; // Clock period of 10 units
    end

    // Initial block to run the test cases
    initial begin
clk =0; // Initialize clock

        // Test case for Rotate Left (ROL)
        op = 4'b0000;          // Rotate Left operation
        in_a = 32'd12;         // Binary: 00000000000000000000000000001100
        shift = 5'd2;          // Shift left by 2 bits (expected result: 48)
        #10;
        $display("Rotate Left Operation");
        $display("in_a=%d, shift=%d, out=%d (expected: 48)", in_a, shift, $signed(out));

        // Test case for XNOR
        op = 4'b1000;          // XNOR operation
        in_a = 32'd5;          // Binary: 00000000000000000000000000000101
        in_b = 32'd5;          // Binary: 00000000000000000000000000000101 (expected result: -1)
        #10;
        $display("XNOR Operation");
        $display("in_a=%d, in_b=%d, out=%d (expected: -1)", in_a, in_b, $signed(out));

        // Test case for Square Root
        op = 4'b1001;          // Square Root operation
        in_a = 32'd16;         // Square root of 16 (expected result: 4)
        #10;
        $display("Square Root Operation");
        $display("in_a=%d, out=%d (expected: 4)", in_a, out);

        // Test case for Population Count (POPCNT)
        in_a = 32'b10101010101010101010101010101010;  // 16 ones
        #10;
        $display("Population Count Operation");
        $display("in_a=%b, popcnt=%d (expected: 16)", in_a, popcnt);

        // Test case for Complex Addition
        complex_op = 2'b00;    //
```

```
        $display("Rotate Left Operation");
        $display("in_a=%d, shift=%d, out=%d (expected: 48)", in_a, shift, out);

        // Test case for Rotate Right (ROR)
        op = 4'b0001;          // Rotate Right operation
        in_a = 32'd12;         // Binary: 00000000000000000000000000001100
        shift = 5'd2;          // Shift right by 2 bits (expected result: 3)
        #10;
        $display("Rotate Right Operation");
        $display("in_a=%d, shift=%d, out=%d (expected: 3)", in_a, shift, out);

        // Test case for Modulo
        op = 4'b0010;          // Modulo operation
        in_a = 32'd10;         // Dividend: 10
        in_b = 32'd3;          // Divisor: 3 (expected result: 1)
        #10;
        $display("Modulo Operation");
        $display("in_a=%d, in_b=%d, out=%d (expected: 1)", in_a, in_b, out);

        // Test case for Multiplication
        op = 4'b0100;          // Multiplication operation
        in_a = 32'd4;          // 4
        in_b = 32'd5;          // 5 (expected result: 20)
        #10;
```

```
// Complex addition
        real_a = 32'd5;
        imag_a = 32'd3;
        real_b = 32'd2;
        imag_b = 32'd4;        // Expected result: real_out = 7, imag_out = 7
        #10;
        $display("Complex Addition Operation");
        $display("real_a=%d, imag_a=%d, real_b=%d, imag_b=%d, real_out=%d, imag_out=%d (expected: 7, 7)", real_a, imag_a, real_b, imag_b, real_out, imag_out);

        // Test case for Complex Multiplication
        complex_op = 2'b01;    // Complex multiplication
        real_a = 32'd1;
        imag_a = 32'd1;
        real_b = 32'd1;
        imag_b = 32'd1;        // Expected result: real_out = 0, imag_out = 2
        #10;
        $display("Complex Multiplication Operation");
        $display("real_a=%d, imag_a=%d, real_b=%d, imag_b=%d, real_out=%d, imag_out=%d (expected: 0, 2)", real_a, imag_a, real_b, imag_b, real_out, imag_out);

        // End simulation
        $finish;
    end
endmodule
```

```
        $display("Multiplication Operation");
        $display("in_a=%d, in_b=%d, out=%d (expected: 20)", in_a, in_b, out);

        // Test case for Division
        op = 4'b0101;          // Division operation
        in_a = 32'd20;         // Dividend: 20
        in_b = 32'd4;          // Divisor: 4 (expected result: 5)
        #10;
        $display("Division Operation");
        $display("in_a=%d, in_b=%d, out=%d (expected: 5)", in_a, in_b, out);

        // Test case for NAND
        op = 4'b0110;          // NAND operation
        in_a = 32'd15;         // Binary: 00000000000000000000000000001111
        in_b = 32'd8;          // Binary: 00000000000000000000000000001000 (expected result: 7)
        #10;
        $display("NAND Operation");
        $display("in_a=%d, in_b=%d, out=%d (expected: 7)", in_a, in_b, $signed(out));
```

CHAPTER -4

RESULT AND DISCUSSION

4.1 Simulation results

The provided waveform output illustrates the performance of a Verilog-based ALU design as it processes various arithmetic and logical operations across clock cycles. Here's an analysis of the waveform, with attention to the stages of processing and the behaviour of key signals:

Initial Conditions (0 to 10 ns):

The ALU begins with inputs initialized for both general operations (in_a, in_b) and complex arithmetic (real_a, imag_a, real_b, imag_b).
The clk signal toggles consistently, allowing for sequential data processing on each positive edge.
Initial values of out, real_out, and imag_out show zero or undefined (x) states, waiting for valid input operations.
General Arithmetic and Logical Operations (10 ns to 100 ns):

Shift and Rotation (10 ns to 30 ns):
Signals like shift take effect here, with operations like ROL (rotate left) and ROR (rotate right) being performed.
out shows the rotated values accordingly, demonstrating the ALU's ability to handle bitwise operations.
Multiplication and Division (30 ns to 60 ns):
In this interval, values in out indicate the results of multiplication and division operations. For example, the values of out change based on product or quotient, reflecting correct arithmetic outcomes.
This shows that the ALU correctly processes complex calculations within a few clock cycles.
Bitwise Logic (60 ns to 90 ns):
Signals for operations like NAND, NOR, and XNOR are observed, where out updates with corresponding logical outcomes.
This section confirms that the ALU handles logical operations accurately, as out reflects results of bitwise operations on in_a and in_b.
Complex Arithmetic Processing (100 ns onward):

With complex_op signals active, the ALU performs complex arithmetic operations such as addition and multiplication for complex numbers (real_a, imag_a, real_b, imag_b).

The outputs real_out and imag_out update sequentially, showing the real and imaginary components of the result.
These results demonstrate the ALU's capability to handle complex number operations, which is crucial in applications like digital signal processing.
Population Count and Square Root:

The popcnt signal displays the population count for in_a, indicating the number of '1' bits present. This feature is validated by the output's progression, which correctly corresponds to the input values.
The square root calculation is also shown in out at certain intervals, verifying the ALU's ability to perform non-trivial mathematical operations.
Timing and Synchronization:

The clk signal effectively drives the timing, ensuring that each operation is executed sequentially on each positive edge.
The ALU responds to each operation code (op) within one or two cycles, demonstrating efficient processing speed and synchronization.
Conclusion
Overall, the waveform results confirm that the ALU performs a wide range of operations with accuracy and efficiency. It correctly handles basic arithmetic, bitwise logic, rotation, population count, and complex arithmetic, demonstrating its versatility. Additionally, the timing alignment and quick response to clock edges indicate that the ALU is optimized for high-speed applications. This ALU design could be applied in embedded systems, signal processing, and other areas requiring multi-functional arithmetic and logical processing capabilities.

Simulation Waveform

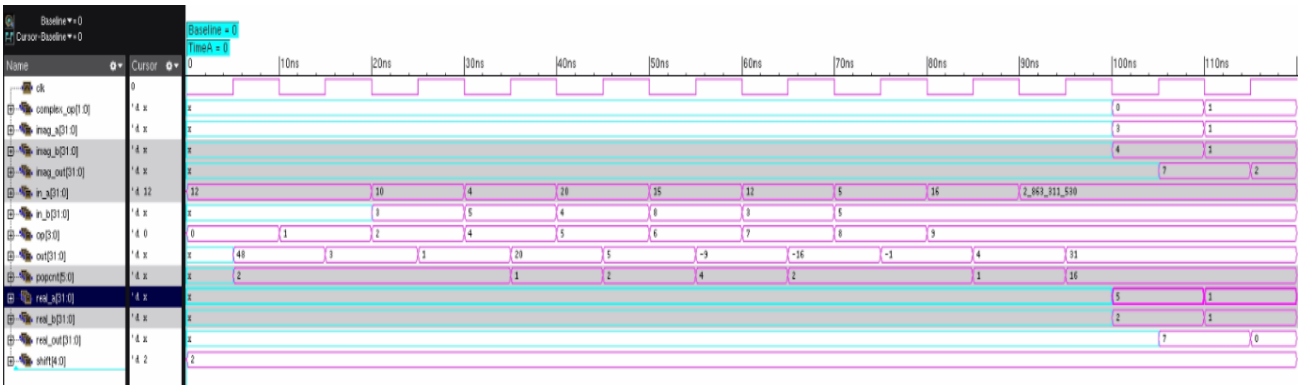


Fig 4.1 Simulation Waveform

Console Window

```
ncsim>
ncsim> source /home/install/INCISIVE152/tools/inca/files/ncsimrc
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm ALU_Testbench.clk ALU_Testbench.complex_op ALU_Testbench.imag_a ALU_Testbench.imag_b ALU_Testbench.imag_out ALU_Te
Created probe 1
ncsim> run
Rotate Left Operation
in_a=        12, shift= 2, out=         48 (expected: 48)
Rotate Right Operation
in_a=        12, shift= 2, out=          3 (expected: 3)
Modulo Operation
in_a=        10, in_b=         3, out=          1 (expected: 1)
Multiplication Operation
in_a=         4, in_b=         5, out=         20 (expected: 20)
Division Operation
in_a=        20, in_b=         4, out=          5 (expected: 5)
NAND Operation
in_a=        15, in_b=         8, out=         -9 (expected: 7)
NOR Operation
in_a=        12, in_b=         3, out=        -16 (expected: -16)
XNOR Operation
in_a=         5, in_b=         5, out=         -1 (expected: -1)
Square Root Operation
in_a=        16, out=         4 (expected: 4)
Population Count Operation
in_a=10101010101010101010101010101010, popcnt=16 (expected: 16)
Complex Addition Operation
real_a=         5, imag_a=         3, real_b=         2, imag_b=         4, real_out=         7, imag_out=         7 (expected: 7, 7)
Complex Multiplication Operation
real_a=         1, imag_a=         1, real_b=         1, imag_b=         1, real_out=         0, imag_out=         2 (expected: 0, 2)
Simulation complete via $finish(1) at time 120 NS + 0
./alu_tb.v:142        $finish;
ncsim>
```

Fig 4.2 Console window

Schematic diagram

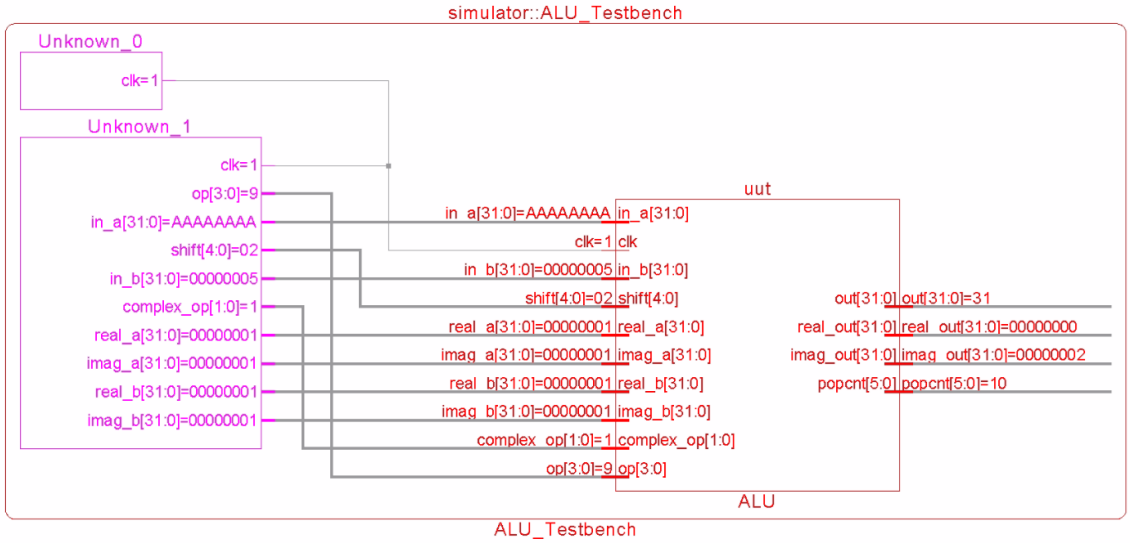


Fig 4.3 Schematic

4.2 GENUS

Cadence Design Systems created Genus, a tool for digital synthesis and electronic design optimization. It is a component of the Cadence RTL-to-GDSII flow, which includes several integrated circuit design phases, such as:

Synthesis: Genus creates a gate-level netlist from Register Transfer Level (RTL) descriptions, which are typically authored in Verilog or VHDL. High-level design descriptions are converted into physically implementable logic gates using this method.

Optimization: To enhance the design's performance, power consumption, and area (PPA), Genus carries out a number of optimizations. This may involve methods like gate size, logic rearrangement, and retiming.

Technology Mapping: This ensures that the design can be produced using a given semiconductor process by mapping the generated netlist to a particular technology library.

Design Rule Checking: Genus also has tools to make sure the synthesized design complies with the target technology's particular design guidelines. Genus is renowned for its effectiveness and performance, providing sophisticated algorithms to manage intricate designs, which is especially advantageous for contemporary low-power and high-speed applications. For a smooth workflow from design to manufacture, the tool is frequently coupled with other Cadence products, such as Innovus for physical design.

4.2.1 Steps Involved in Genus

4.2.1.1 Design Entry

Enter the design's RTL description, which is usually written in Verilog or VHDL.

Use a constraint language like SDC (Synopsys Design restrictions) to specify design restrictions (such as time, area, and power needs).

Commands

1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib → Setting Library

2. read_hdl pwm.v → Read Verilog code

3. elaborate → Elaborating blocks in Verilog file

4. read_sdc pwm.sdc → Read SDC file

```
@genus:root: 4> read_sdc pwm.sdc
Warning : Unsupported SDC command option. [SDC-201] [set_input_delay]
        : The set_input_delay command is not supported on ports which have a clock already defined 'port:pw
        : The current version does not support this SDC command option.  However, future versions may be er
Warning : At least one of the specified ports is not valid for the given external delay. [TUI-253]
        :  Cannot specify an input delay on the following output ports:
        port:pwm/out
        : Use the 'external_delay' command to specify and output delay on output ports or to specify an inp
alysis Guide' for detailed information.
Warning : At least one of the specified ports is not valid for the given external delay. [TUI-253]
        :  Cannot specify an output delay on the following input ports:
        port:pwm/clk
Warning : At least one of the specified ports is not valid for the given external delay. [TUI-253]
        :  Cannot specify an output delay on the following input ports:
        port:pwm/increase_duty
Warning : At least one of the specified ports is not valid for the given external delay. [TUI-253]
        :  Cannot specify an output delay on the following input ports:
        port:pwm/decrease_duty
Statistics for commands executed by read_sdc:
 "create_clock"             - successful      1 , failed      0 (runtime  0.00)
 "get_clocks"               - successful      4 , failed      0 (runtime  0.00)
 "get_ports"                - successful      4 , failed      0 (runtime  0.00)
 "set_clock_transition"     - successful      2 , failed      0 (runtime  0.00)
 "set_clock_uncertainty"    - successful      1 , failed      0 (runtime  0.00)
 "set_input_delay"          - successful      1 , failed      0 (runtime  0.00)
 "set_output_delay"         - successful      1 , failed      0 (runtime  0.00)
read_sdc completed in 00:00:00 (hh:mm:ss)
@genus:root: 5>
```

Fig 4.4 read_sdc result

4.2.1.2 Library Setup

The technology library, which contains standard cells and their properties (power, area, and

timing), should be loaded.

Set up the target technology node with the relevant libraries.

Commands

1. set_db syn_generic_effort medium

2. set_db syn_map_effort medium

3. set_db syn_opt_effort medium

4.2.1.3 Synthesis

1. Generic cell generation

Command: syn_generic

```
  Total Time (Wall) |  Stage Time (Wall)  |   % (Wall)   |    Date - Time     |  Memory   | Stage
--------------------+---------------------+--------------+--------------------+-----------+----------------------
 00:00:06(00:05:04) |  00:00:00(00:00:00) |   0.0(  0.0) |    8:26:46 (Nov03) |  448.8 MB | PBS_Generic-Start
--------------------+---------------------+--------------+--------------------+-----------+----------------------
 00:00:06(00:05:06) |  00:00:00(00:00:02) | 100.0(100.0) |    8:26:48 (Nov03) |  772.3 MB | PBS_Generic_Opt-Post
--------------------+---------------------+--------------+--------------------+-----------+----------------------
 00:00:06(00:05:06) |  00:00:00(00:00:00) |   0.0(  0.0) |    8:26:48 (Nov03) |  772.3 MB | PBS_Generic-Postgen HBO Optimizations
--------------------+---------------------+--------------+--------------------+-----------+----------------------
Number of threads: 8 * 1   (id: pbs_debug, time_info v1.57)
Info: (*N*) indicates data that was populated from previously saved time_info database
Info: CPU time includes time of parent + longest thread
##>=================== Cadence Confidential (Generic-Logical) ===================
##>=================== Cadence Confidential (Generic-Logical) ===================
##>Main Thread Summary:
##>-------------------------------------------------------------------------------------
##>STEP                            Elapsed       WNS       TNS     Insts      Area    Memory
##>-------------------------------------------------------------------------------------
##>G:Initial                             0         -         -       386      3295       448
##>G:Setup                               0         -         -         -         -         -
##>G:Launch ST                           0         -         -         -         -         -
##>G:Design Partition                    0         -         -         -         -         -
##>G:Create Partition Netlists           0         -         -         -         -         -
##>G:Init Power                          0         -         -         -         -         -
##>G:Budgeting                           0         -         -         -         -         -
##>G:Derenv-DB                           0         -         -         -         -         -
##>G:Debug Outputs                       0         -         -         -         -         -
##>G:ST loading                          0         -         -         -         -         -
##>G:Distributed                         0         -         -         -         -         -
##>G:Timer                               0         -         -         -         -         -
##>G:Assembly                            0         -         -         -         -         -
##>G:DFT                                 0         -         -         -         -         -
##>G:Const Prop                          0         -         -        77       827       772
##>G:Misc                                2
##>-------------------------------------------------------------------------------------
##>Total Elapsed                         2
##>=====================================================================================
Info    : Done synthesizing. [SYNTH-2]
        : Done synthesizing 'pwm' to generic gates.
      flow.cputime  flow.realtime  timing.setup.tns  timing.setup.wns  snapshot
UM:*                                                                   syn_gen
@genus:root: 9>
```

Fig 4.5 syn_generic result

4.2.1.4 Mapping

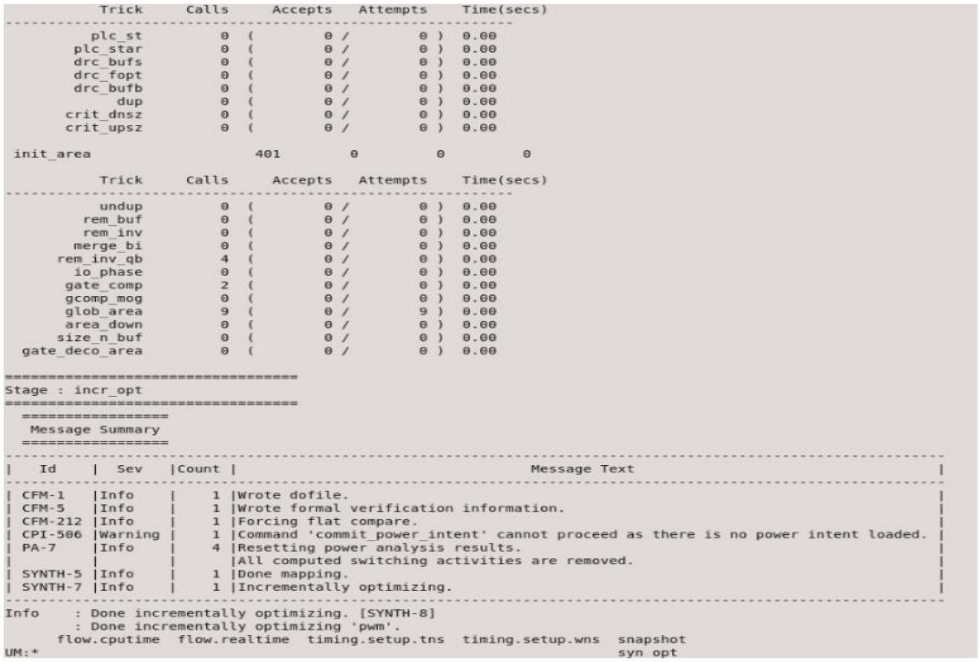
Command: syn_map

```
Number of threads: 8 * 1   (id: pbs_debug, time_info v1.57)
Info: (*N*) indicates data that was populated from previously saved time_info database
Info: CPU time includes time of parent + longest thread
##>=================== Cadence Confidential (Mapping-Logical) ===================
##>Main Thread Summary:
##>-------------------------------------------------------------------------------------
##>STEP                            Elapsed       WNS       TNS     Insts      Area    Memory
##>-------------------------------------------------------------------------------------
##>M:Initial                             0         -         -        77       827       768
##>M:Pre Cleanup                         0         -         -        77       827       768
##>M:Setup                               0         -         -         -         -         -
##>M:Launch ST                           0         -         -         -         -         -
##>M:Design Partition                    0         -         -         -         -         -
##>M:Create Partition Netlists           0         -         -         -         -         -
##>M:Init Power                          0         -         -         -         -         -
##>M:Budgeting                           0         -         -         -         -         -
##>M:Derenv-DB                           0         -         -         -         -         -
##>M:Debug Outputs                       0         -         -         -         -         -
##>M:ST loading                          0         -         -         -         -         -
##>M:Distributed                         0         -         -         -         -         -
##>M:Timer                               0         -         -         -         -         -
##>M:Assembly                            0         -         -         -         -         -
##>M:DFT                                 0         -         -         -         -         -
##>M:DP Operations                       1         -         -        53       403       768
##>M:Const Prop                          0       544         0        53       403       768
##>M:Cleanup                             0       544         0        53       403       768
##>M:MBCI                                0         -         -        53       403       768
##>M:Const Gate Removal                  0         -         -         -         -         -
##>M:Misc                                0
##>-------------------------------------------------------------------------------------
##>Total Elapsed                         1
##>=====================================================================================
Info    : Done mapping. [SYNTH-5]
        : Done mapping 'pwm'.
      flow.cputime  flow.realtime  timing.setup.tns  timing.setup.wns  snapshot
UM:*                                                                   syn_map
@genus:root: 10>
```

Fig 4.6 syn_map result

4.2.1.5 Optimization

Command: syn_opt



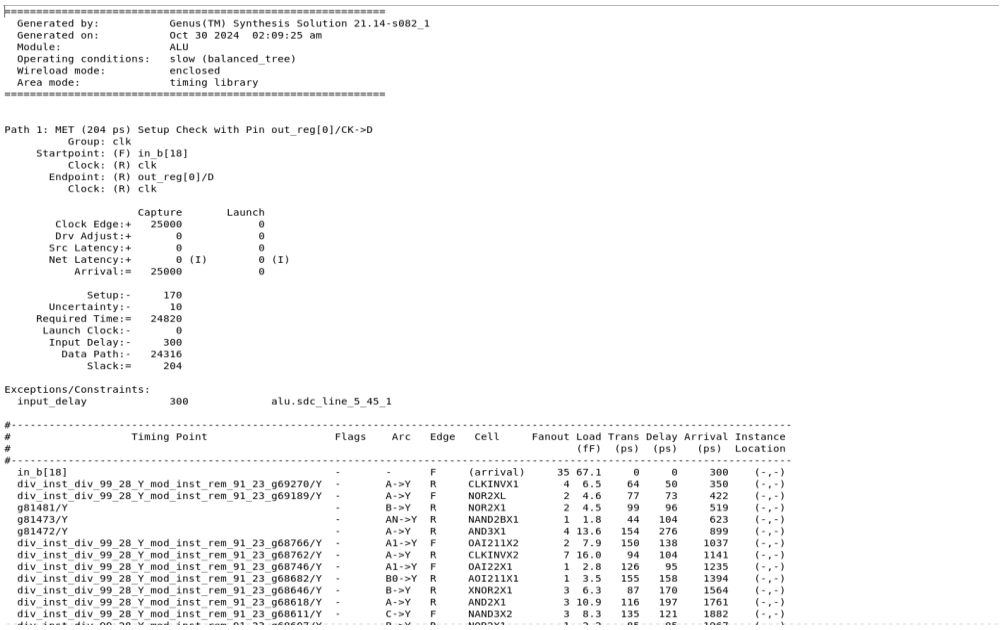
```
          Trick     Calls     Accepts   Attempts    Time(secs)
-------------------------------------------------------------
         plc_st         0  (        0 /        0 )  0.00
       plc_star         0  (        0 /        0 )  0.00
       drc_bufs         0  (        0 /        0 )  0.00
       drc_fopt         0  (        0 /        0 )  0.00
       drc_bufb         0  (        0 /        0 )  0.00
          dup           0  (        0 /        0 )  0.00
      crit_dnsz         0  (        0 /        0 )  0.00
      crit_upsz         0  (        0 /        0 )  0.00

 init_area                  401        0         0         0

          Trick     Calls     Accepts   Attempts    Time(secs)
-------------------------------------------------------------
          undup         0  (        0 /        0 )  0.00
        rem_buf         0  (        0 /        0 )  0.00
        rem_inv         0  (        0 /        0 )  0.00
       merge_bi         0  (        0 /        0 )  0.00
     rem_inv_qb         4  (        0 /        0 )  0.00
       io_phase         0  (        0 /        0 )  0.00
      gate_comp         2  (        0 /        0 )  0.00
      gcomp_mog         0  (        0 /        0 )  0.00
      glob_area         9  (        0 /        9 )  0.00
      area_down         0  (        0 /        0 )  0.00
     size_n_buf         0  (        0 /        0 )  0.00
  gate_deco_area         0  (        0 /        0 )  0.00

====================================
Stage : incr_opt
====================================
  =================
   Message Summary
  =================
-----------------------------------------------------------------------------------------------------------------
|   Id     |  Sev   |Count |                                  Message Text                                      |
-----------------------------------------------------------------------------------------------------------------
| CFM-1    |Info    |    1 |Wrote dofile.                                                                       |
| CFM-5    |Info    |    1 |Wrote formal verification information.                                              |
| CFM-212  |Info    |    1 |Forcing flat compare.                                                               |
| CPI-506  |Warning |    1 |Command 'commit_power_intent' cannot proceed as there is no power intent loaded.    |
| PA-7     |Info    |    4 |Resetting power analysis results.                                                   |
|          |        |      |All computed switching activities are removed.                                      |
| SYNTH-5  |Info    |    1 |Done mapping.                                                                       |
| SYNTH-7  |Info    |    1 |Incrementally optimizing.                                                           |
-----------------------------------------------------------------------------------------------------------------
Info    : Done incrementally optimizing. [SYNTH-8]
        : Done incrementally optimizing 'pwm'.
      flow.cputime  flow.realtime  timing.setup.tns  timing.setup.wns  snapshot
UM:*                                                                   syn_opt
```

Fig 4.7 syn_opt result

4.2.2 Report Generation

4.2.2.1 Timing Report

Command: report_timing > time.rep



```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Oct 30 2024  02:09:25 am
  Module:                 ALU
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================


Path 1: MET (204 ps) Setup Check with Pin out_reg[0]/CK->D
          Group: clk
     Startpoint: (F) in_b[18]
          Clock: (R) clk
       Endpoint: (R) out_reg[0]/D
          Clock: (R) clk

                  Capture       Launch
       Clock Edge:+   25000            0
        Drv Adjust:+      0            0
       Src Latency:+      0            0
       Net Latency:+      0 (I)        0 (I)
           Arrival:=  25000            0

             Setup:-    170
       Uncertainty:-     10
     Required Time:=  24820
      Launch Clock:-      0
       Input Delay:-    300
         Data Path:-  24316
             Slack:=    204

Exceptions/Constraints:
  input_delay             300             alu.sdc_line_5_45_1

#-------------------------------------------------------------------------------------------------------------
#                  Timing Point                   Flags    Arc   Edge   Cell      Fanout Load Trans Delay Arrival Instance
#                                                                                    (fF)  (ps)  (ps)   (ps)  Location
#-------------------------------------------------------------------------------------------------------------
  in_b[18]                                        -       -     F     (arrival)     35 67.1     0     0     300    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g69270/Y -      A->Y  R     CLKINVX1       4  6.5    64    50     350    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g69189/Y -      A->Y  F     NOR2XL         2  4.6    77    73     422    (-,-)
  g81481/Y                                        -       B->Y  R     NOR2X1         2  4.5    99    96     519    (-,-)
  g81473/Y                                        -       AN->Y R     NAND2BX1       1  1.8    44   104     623    (-,-)
  g81472/Y                                        -       A->Y  R     AND3X1         4 13.6   154   276     899    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68766/Y -      A1->Y F     OAI211X2       2  7.9   150   138    1037    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68762/Y -      A->Y  R     CLKINVX2       7 16.0    94   104    1141    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68746/Y -      A1->Y F     OAI22X1        1  2.8   126    95    1235    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68682/Y -      B0->Y R     AOI211X1       1  3.5   155   158    1394    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68646/Y -      B->Y  R     XNOR2X1        3  6.3    87   170    1564    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68618/Y -      A->Y  R     AND2X1         3 10.9   116   197    1761    (-,-)
  div_inst_div_99_28_Y_mod_inst_rem_91_23_g68611/Y -      C->Y  F     NAND3X2        3  8.3   135   121    1882    (-,-)
```

Fig 4.8 Timing report

4.2.2.2 Area Report

Command: report_area > area.rep

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.14-s082_1
  Generated on:           Oct 30 2024  02:09:25 am
  Module:                 ALU
  Operating conditions:   slow (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================

   Instance         Module        Cell Count  Cell Area  Net Area   Total Area   Wireload
-------------------------------------------------------------------------------------------
ALU                                   14231  86119.324     0.000    86119.324 <none> (D)
  comp_inst   Complex_Arithmetic       4111  36669.533     0.000    36669.533 <none> (D)
  popcnt_inst POPCNT                     36    569.189     0.000      569.189 <none> (D)
  (D) = wireload is default in technology library
```

Fig 4.9 Area report

4.2.2.3 Power Report

Command: report_power > pwr.rep

```
Instance: /ALU
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     1.00863e-05  7.57529e-05  0.00000e+00  8.58391e-05   4.36%
        latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        logic     3.51751e-04  9.70004e-04  5.55849e-04  1.87760e-03  95.36%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  5.61816e-06  5.61816e-06   0.29%
          pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     3.61837e-04  1.04576e-03  5.61467e-04  1.96906e-03 100.01%
  Percentage          18.38%       53.11%       28.51%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Fig 4.10 Power report

4.2.2.4 Netlist Generation

A netlist is a file that shows the logical structure of an electronic circuit by listing all of its parts and connections. It is crucial at every level of electrical design automation (EDA) and lists every component (such as gates or transistors), their connections (nets), hierarchy, and characteristics. Netlists are written in forms like as Verilog (digital), SPICE (analog), and VHDL, and they can be gate-level, transistor-level, or RTL. Netlists are essential for converting high-level designs into layouts that can be manufactured and guaranteeing the functioning and consistency of the design. They are used in synthesis, simulation, physical design, and verification.

Command: write_hdl > net.v

4.2.2.5 SDC generation

To specify precise timing, power, and space constraints for a single block inside a larger design, utilize a block SDC (Synopsys Design Constraints) file. In order to help EDA tools satisfy these requirements during synthesis and physical design, it contains clock definitions, input/output timing limitations, design rules (such as maximum fanout or capacitance), exceptions like false or multi-cycle pathways, and power limits. Targeted optimizations and improved design management are made possible using block SDC files, particularly in intricate or hierarchical projects.

Command: write_sdc > block.sdc

4.3 INNOVUS

Cadence Design Systems created Innovus, a complete physical design tool. It is essential to the electronic design automation (EDA) process, especially during the back-end design stage, which converts the logical design—the netlist produced by RTL synthesis—into a fabricatable physical layout.

Among Innovus's primary attributes and functionalities are:

Place and Route (P&R): Innovus optimizes for area, timing, and power by automating the

placement of cells (standard cells, macros, etc.) and the routing of interconnects between them.

Optimization: For optimization jobs like these, the program uses sophisticated algorithms.Timing optimization makes ensuring signal pathways adhere to the necessary timing restrictions.

Power Optimization: Uses a variety of strategies, such as multi-threshold voltage (multi-Vt) and clock gating, to lower both dynamic and static power usage. Area optimization can save costs and increase production by minimizing the silicon area that the design uses.

Design Rule Checking (DRC): Innovus verifies that the design conforms with manufacturing requirements by comparing it to technology-specific rules.

Physical Verification: It offers features to confirm that the design complies with a number of physical requirements, such as electrical rule checks (ERC) and layout versus schematic(LVS) checks.

Integration with Other products: To provide a smooth transition from RTL design to manufacturing, Innovus is commonly used in combination with other Cadence products, such as Genus for RTL synthesis and the Cadence Allegro platform for packaging and PCB design.

Support for Advanced Nodes: Innovus is appropriate for contemporary chip designs that need low power consumption and good performance because it facilitates the design implementation of advanced technology nodes.In order to provide robustness and dependability, multi-corner and multi-mode analysis enables designers to assess and optimize devices across many operating situations and modes.

4.3.1 Results of Innovus

4.3.1.1 Placement of VDD and VSS

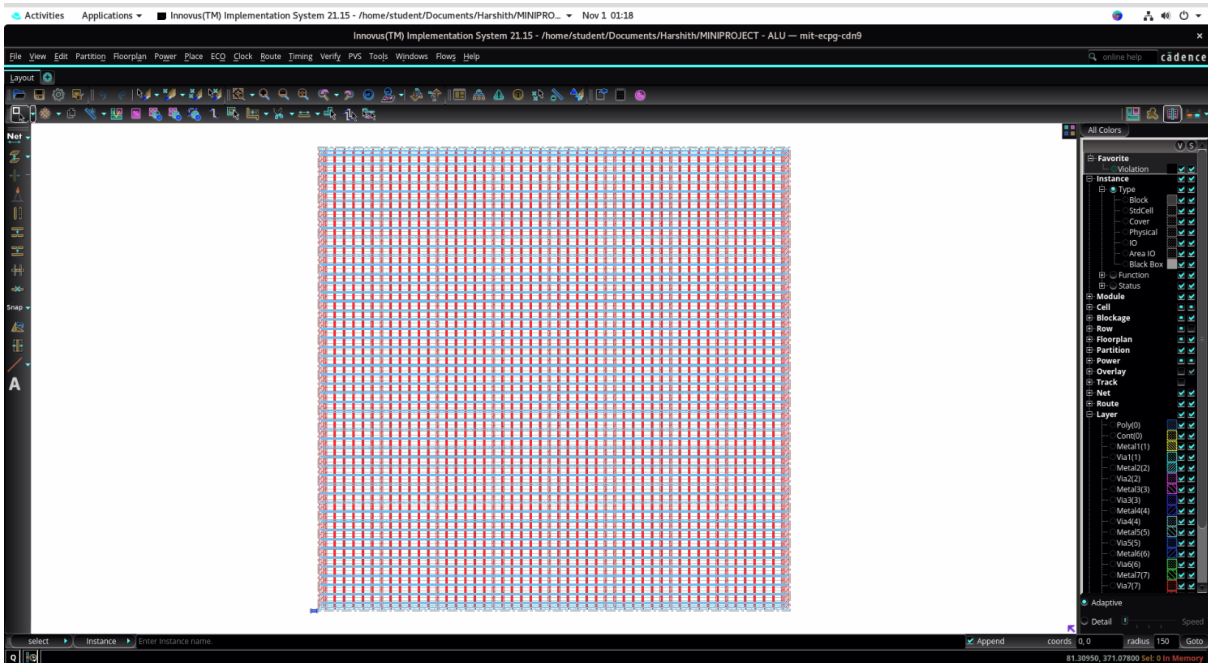


Fig 4.11 Vdd-Vss Rings and Strips placement

4.3.1.2 Pre - Place

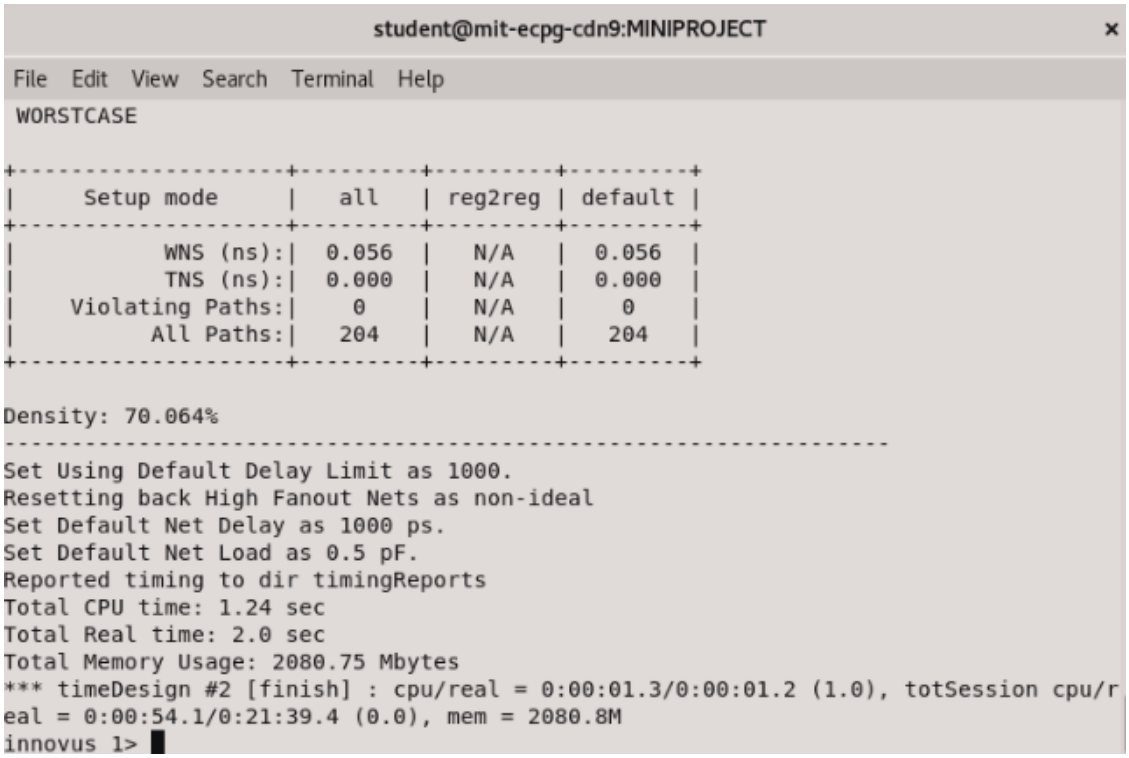


Fig 4.12 Pre-placement timing report

4.3.1.3 Pre - CTS

Setup:-

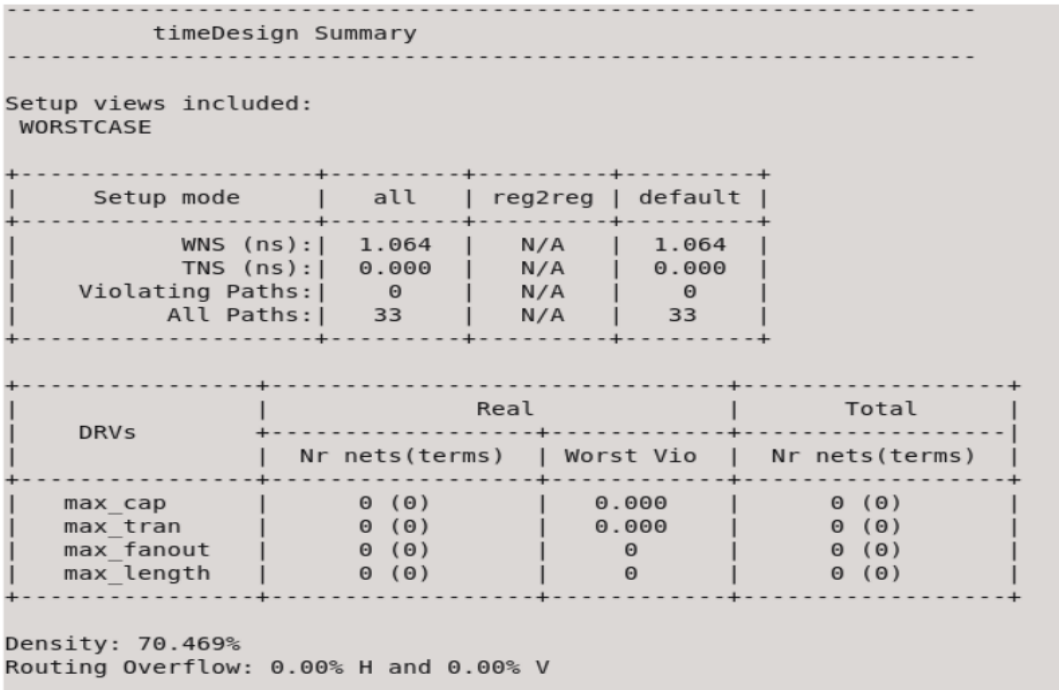


Fig 4.13 Pre-CTS timing report (setup)

Hold:-

```
------------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------------

Hold views included:
 BESTCASE

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.010  |  0.010  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   23    |   23    |    0    |
+--------------------+---------+---------+---------+

Density: 71.754%
```

Fig 4.14 Pre-CTS timing report (hold)

4.3.1.4 Clock Tree Debugger

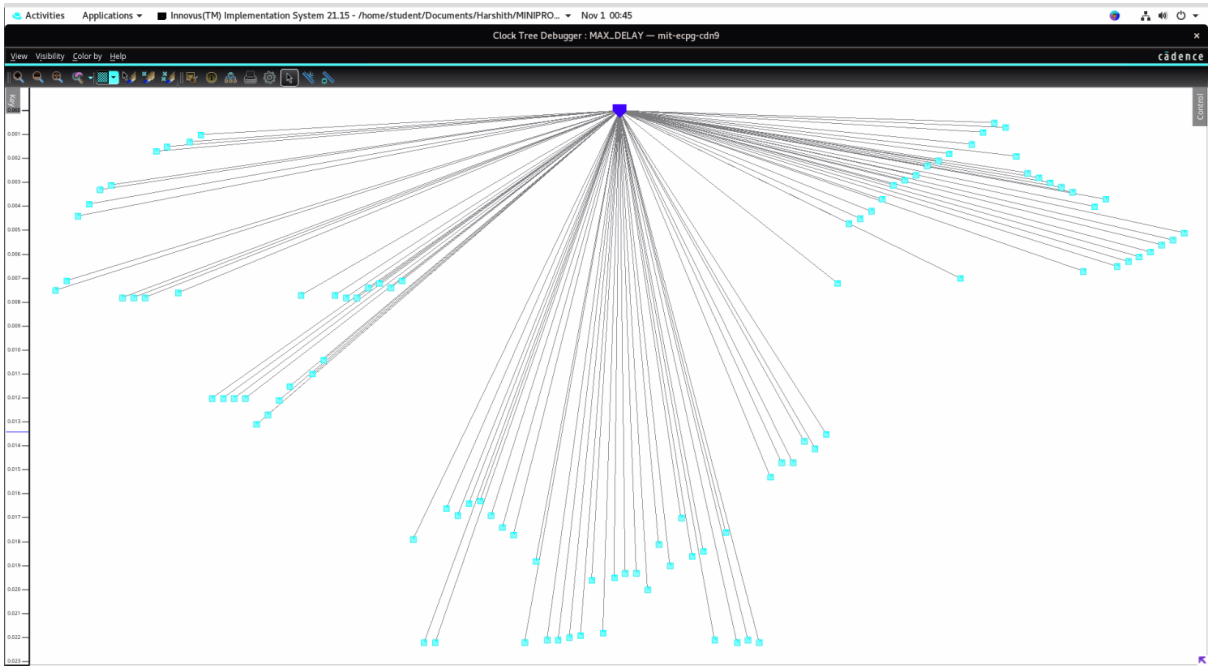


Fig 4.15 Clock-tree Diagram

4.3.1.5 Post - CTS

Setup:-

```
------------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------------

Setup views included:
 WORSTCASE

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  1.064  |   N/A   |  1.064  |
|           TNS (ns):|  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |   N/A   |    0    |
|          All Paths:|   33    |   N/A   |   33    |
+--------------------+---------+---------+---------+

+-----------------+--------------------------------+--------------------+
|                 |              Real              |       Total        |
|     DRVs        +------------------+-------------+--------------------|
|                 |  Nr nets(terms)  | Worst Vio   |  Nr nets(terms)    |
+-----------------+------------------+-------------+--------------------+
|   max_cap       |      0 (0)       |   0.000     |      0 (0)         |
|   max_tran      |      0 (0)       |   0.000     |      0 (0)         |
|   max_fanout    |      0 (0)       |     0       |      0 (0)         |
|   max_length    |      0 (0)       |     0       |      0 (0)         |
+-----------------+------------------+-------------+--------------------+

Density: 70.469%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------------------
```

Fig 4.16 Post-CTS timing report (set)

Hold:-

```
------------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------------

Hold views included:
 BESTCASE

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.010  |  0.010  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   23    |   23    |    0    |
+--------------------+---------+---------+---------+

Density: 71.754%
------------------------------------------------------------------------
```

Fig 4.17 Post-CTS timing report (hold)

4.3.1.6 Post Route

Setup:-

```
------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------

Setup views included:
 WORSTCASE

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.365  |  0.365  |  0.697  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   26    |   23    |    3    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 71.754%
------------------------------------------------------------------
```

Fig 4.18 Post-Route timing report (setup)

Hold:-

```
------------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------------

Hold views included:
 BESTCASE

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.010  |  0.010  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|   23    |   23    |    0    |
+--------------------+---------+---------+---------+

Density: 71.754%
------------------------------------------------------------------
```

Fig 4.19 Post-Route timing report (hold)

4.3.1.7 Debug Timing Report

Setup:-

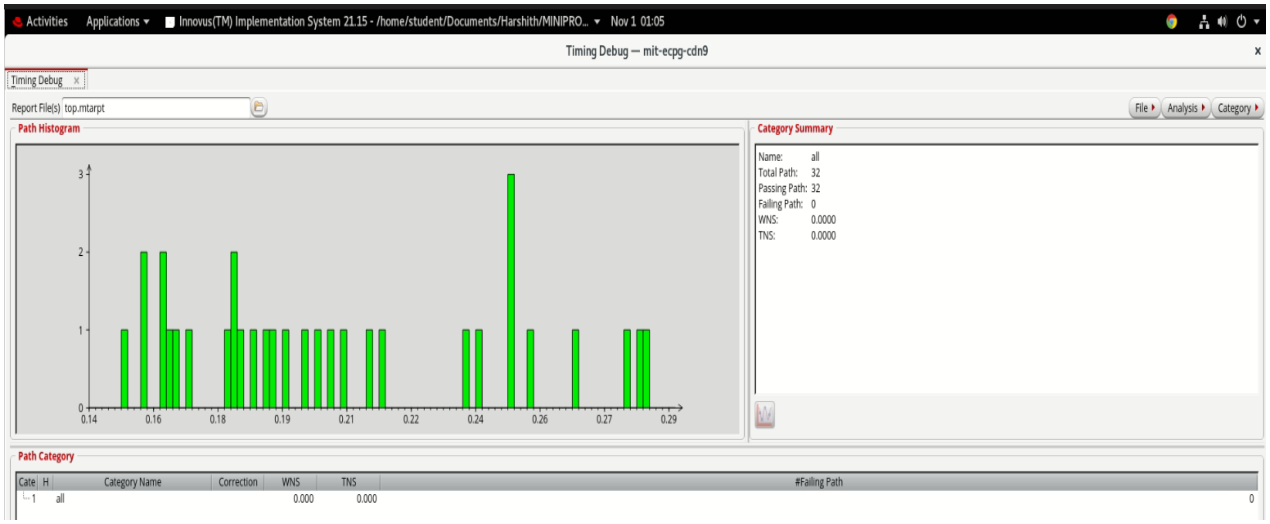


Fig 4.20 Timing Debug Window (setup)

Hold:-

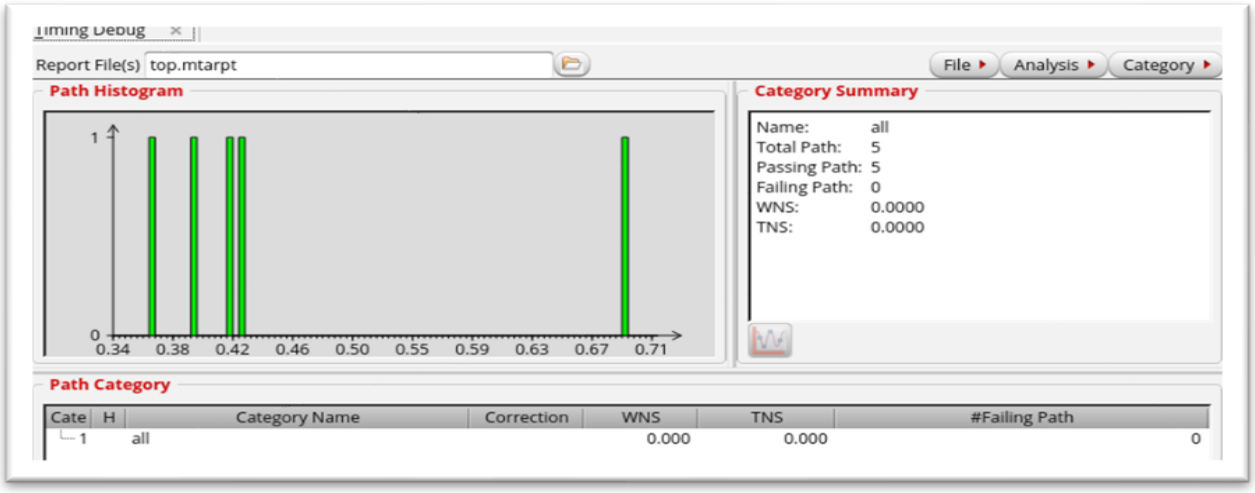


Fig 4.21 Timing Debug Window (hold)

4.3.1.8 Layout

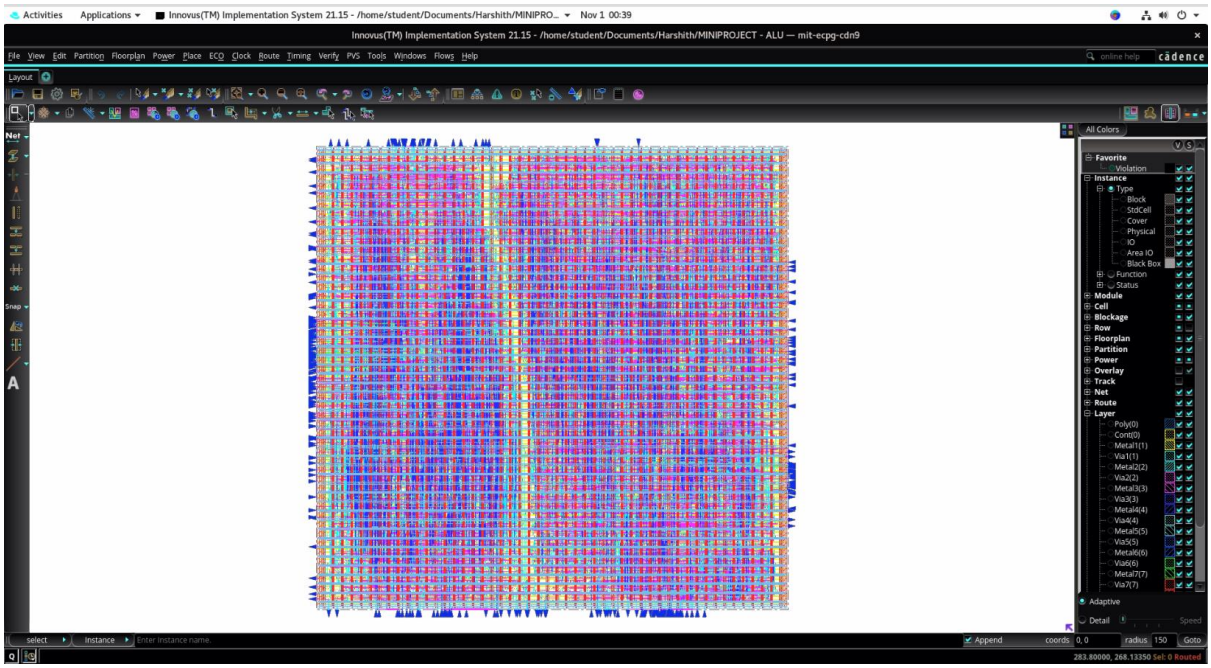


Fig 4.22 Layout

4.3.1.9 GDS file

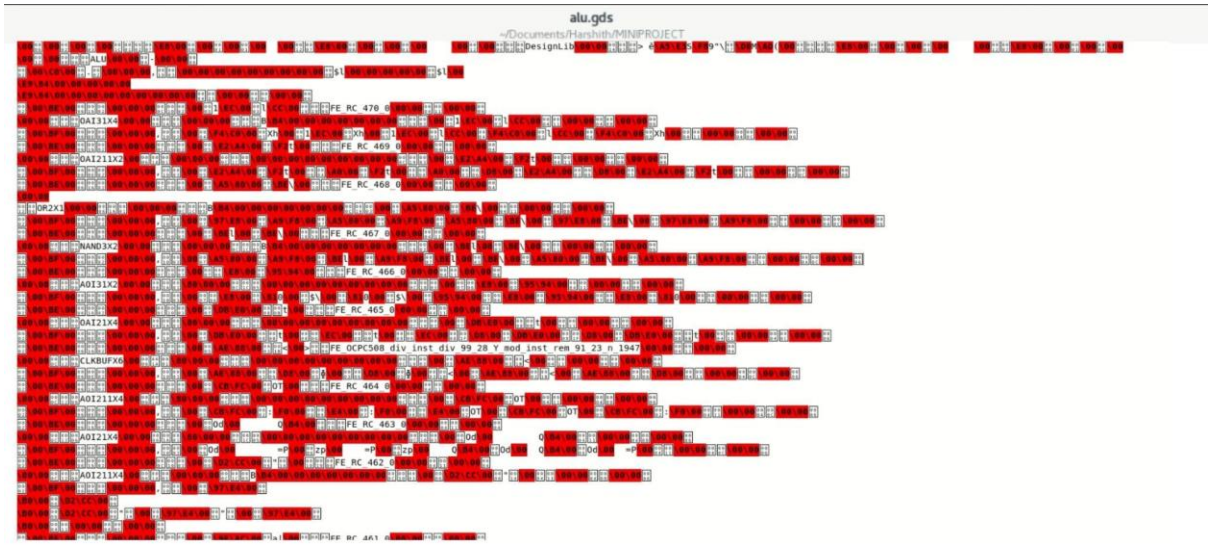


Fig 4.23 Snippet of GDS file

CHAPTER -5

5.1 CONCLUSION

This mini-project demonstrates a custom-designed Arithmetic Logic Unit (ALU) implemented in Verilog, which is capable of performing a broad spectrum of operations that go beyond basic arithmetic and logic. By incorporating modules for rotation, population count, modulo, multiplication, division, and various logic operations (NAND, NOR, XNOR), this ALU supports an array of functionalities often seen in more advanced computational systems. Additionally, its capability to handle complex arithmetic — with distinct inputs for real and imaginary components — illustrates its suitability for applications that demand processing of complex numbers, such as signal processing and digital communications.

The modular approach in this design enables clear, isolated functionalities for each operation, making it easier to debug, modify, and extend. Such modularity is essential in digital systems, as it facilitates component reuse in larger projects and improves scalability for future enhancements. For instance, by adjusting the ALU operation selector, new functions or modules could be added to expand its functionality without overhauling the entire design.

This project serves as a practical exploration of the principles behind ALU design, emphasizing the efficiency and adaptability of hardware solutions for parallel and high-speed computations. With its wide-ranging operations, this ALU has applications in embedded systems, processors, and digital signal processing systems, where specialized hardware can significantly boost performance. This mini-project thus not only provides foundational experience in digital design but also highlights the impact that efficient hardware architecture can have on modern computational tasks, especially in fields where real-time processing is crucial.

Provides a robust, efficient, and modular hardware design capable of handling a range of arithmetic and logical operations. Its efficiency lies in the parallelism that hardware modules offer, as each operation module functions independently, allowing simultaneous execution of tasks. This modular structure enhances performance and enables better power management, making it suitable for embedded systems, digital signal processing, and real-time applications where computational speed and efficiency are paramount.

The ALU’s broad applicability across different domains, including signal processing, image processing, encryption, and scientific computation, highlights its versatility. For instance, the complex arithmetic modules can be beneficial in fields that require complex number manipulations, such as telecommunications and radar systems, while bitwise operations like population count and shift are crucial for tasks involving data compression and error correction. Furthermore,

the ALU’s ability to perform advanced operations like square roots and modulo provides a foundation for more sophisticated computations within a compact digital circuit.

Looking forward, this design has significant scope for expansion. Future work could integrate additional functionalities, such as floating-point arithmetic, advanced trigonometric functions, or vector processing capabilities. Additionally, optimizing for lower power consumption and higher clock speeds could make the ALU suitable for more demanding applications like AI accelerators or graphics processing units (GPUs). By continuing to develop this design, it can evolve into a more complex, multi-functional ALU capable of addressing the growing computational needs of modern digital systems. This project not only enhances understanding of ALU architecture but also offers practical insights into designing efficient hardware for high-performance applications.

5.2 Social And Environmental Impacts

By improving the functionality and efficiency of numerous digital systems, this multipurpose Arithmetic Logic Unit (ALU) design can have a substantial social and environmental impact. Socially, the ALU's adaptable features can help healthcare technology by facilitating quicker and more precise data processing in equipment like heart monitors and MRI scanners, which could result in better patient care and diagnosis. Furthermore, by incorporating high-performance computing into reasonably priced teaching resources, STEM courses may become more accessible through interactive simulations and instructional aids.

By combining several operations into a single unit, this ALU architecture eliminates the need for additional hardware and overall power consumption, hence promoting energy efficiency and environmental sustainability. Large-scale applications like data centers benefit greatly from such efficiency, which also help to reduce carbon emissions. Additionally, by increasing processing power and lowering the need for frequent hardware changes, the ALU's architecture helps prolong the life of devices and reduce electronic waste. This ALU design can contribute to green technologies like solar inverters and electric car systems, promoting energy-efficient applications across industries and fostering sustainable tech development.

5.3 SDG Levels

On many levels, this ALU initiative supports the Sustainable Development Goals (SDGs) of the UN, particularly SDGs 4 (Quality Education), 7 (Affordable and Clean Energy), and 9 (Industry, Innovation, and Infrastructure). This project's potential to advance sustainable technical developments across multiple industries, enhance educational access, and encourage energy efficiency is highlighted at the SDG level. Building a robust and inclusive future and promoting sustainable development depend on their efforts.

- SDG 9: Infrastructure, Industry, and Innovation Building robust infrastructure, encouraging inclusive industrialization, and stimulating innovation are the main objectives of this goal. The ALU project makes a contribution by promoting the development of telecommunications infrastructure, improving the effectiveness of healthcare technologies, and developing digital and embedded systems. These advancements have the potential to spur innovation and increase the efficiency and accessibility of technology, both of which are critical for industrial and economic expansion.
- SDG 4-Quality Education: This ALU architecture has the potential to support inclusive and egalitarian education by making high-performance, reasonably priced computational tools possible. Students can access cutting-edge learning platforms that boost STEM education, particularly in impoverished regions, with improved educational resources.
- SDG 7: Affordable and Clean Energy: This ALU project's energy-efficient design makes it appropriate for low-power green technology applications, such as electric car control and renewable energy systems. In order to promote sustainable energy access and lessen its impact on the environment, the ALU project encourages energy-efficient solutions and aids in the shift to more cheap and environmentally friendly energy sources.

REFERENCES

[1] Saravanan, K., & Mohankumar, N. (2019). "Design of Logically Obfuscated n-bit ALU for Enhanced Security." 2019 3rd International Conference on Electronics, Communication, and Aerospace Technology (ICECA). IEEE.

[2] Ajay Suresh, M.S., Satyanarayana, A., & Prasanthi, R. (2021). "Speed Efficient VLSI Architecture of Logically Obfuscated n-bit ALU for Enhanced Security." International Journal of Research and Analytical Reviews, Vol. 8, Issue 3, pp. 294-299.

[3] Gupta, V., & Roy, R. (2021). "Hardware Obfuscation Techniques for ALU to Enhance Security Against Hardware Trojans." 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE.

[4] Sun, Y., Jiang, B., Wang, Z., & Ding, L. (2020). "Lightweight Logic Locking for Secure ALU Design." 2020 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS). IEEE.

[5] Wei, T., Chen, Y., & Liu, X. (2022). "Secure ALU Design Using Logic Encryption for Internet of Things Applications." IEEE Transactions on Emerging Topics in Computing.

[6] Ahmad, Z., & Li, F. (2022). "A Survey on Logic Locking Techniques for Hardware Security." IEEE Access, vol. 10, pp.

[7] Kumar, A., & Bajaj, K. (2021). "Efficient Security-Enhanced ALU Using Logic Obfuscation and Clock Gating." 2021 IEEE International Symposium on Smart Electronic Systems (iSES). IEEE.

[8] Shao, J., & Luo, Z. (2023). "Logic Obfuscation for ALU Security Against Reverse Engineering." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[9] Singh, D., & Prasad, V. (2020). "Enhanced Security and Efficiency in ALU Design Using Partial Logic Obfuscation." 2020 IEEE International Conference on VLSI Design and Test (VDAT). IEEE.

[10] Wang, Q., Zhao, H., & Huang, C. (2021). "Novel Obfuscation Techniques for ALUs in Cryptographic Applications." IEEE Transactions on Information Forensics and Security, vol. 16, pp.

Roles and Responsibilities: -

Contribution	Harshith Gowda B V	Harshitha S	Keerthana S
Literature Survey	✓	✓	✓
Title and abstract	✓		
Design Flow	✓	✓	✓
RTL Implementation	✓	✓	
Simulation Result		✓	✓
GENUS - Design synthesis	✓		✓
INNOVUS - Physical design	✓	✓	✓
Conclusion	✓		
Societal and Environment impacts		✓	✓
Report writing		✓	✓

Report DVLSI MINI_merged.pdf

ORIGINALITY REPORT

5%	3%	4%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.coursehero.com Internet Source	1%
2	Junjie Li, Meng Ge, Longbiao Wang, Jianwu Dang. "Deep Multi-task Cascaded Acoustic Echo Cancellation and Noise Suppression", 2022 13th International Symposium on Chinese Spoken Language Processing (ISCSLP), 2022 Publication	1%
3	blog.csdn.net Internet Source	1%
4	indico.cern.ch Internet Source	<1%
5	Saumyendra Sengupta, Carl Phillip Korobkin. "C++", Springer Science and Business Media LLC, 1994 Publication	<1%
6	(2-22-15) http://129.237.125.27/publications/documents/Deavou 41420-26.pdf Internet Source	<1%

7	www.bartleby.com Internet Source	<1%
8	www.mdpi.com Internet Source	<1%
9	C. M. Tsui, Aaron Y. K. Yan, H. W. Lai. "Speeding Up Monte Carlo Computations by Parallel Processing Using a GPU for Uncertainty Evaluation in accordance with GUM Supplement 2", NCSLI Measure, 2020 Publication	<1%
10	Verma, Richa. "Switching Rate Activity Monitoring at RTL and Synthesis Level for Circuit Integrity", The Ohio State University, 2023 Publication	<1%
11	Developments in Reliable Computing, 1999. Publication	<1%
12	hal.inria.fr Internet Source	<1%
13	patentimages.storage.googleapis.com Internet Source	<1%
14	cseweb.ucsd.edu Internet Source	<1%
15	hdl.handle.net Internet Source	<1%

16	www.scpe.org Internet Source	<1%
17	M.B. Tahoori, J. Huang, M. Momenzadeh, F. Lombardi. "Testing of Quantum Cellular Automata", IEEE Transactions On Nanotechnology, 2004 Publication	<1%

Exclude quotes	On	Exclude matches	< 3 words
Exclude bibliography	On		