Table of Contents

1. Abstract

This project presents the **design and implementation of a 4-bit Arithmetic Logic Unit (ALU)** through the complete **RTL to GDS (Register Transfer Level to GDSII) design flow**. The ALU, a fundamental component in digital systems, is capable of performing various **arithmetic and logical operations**.

The project begins with the **RTL design** of the ALU using **Verilog**, followed by **synthesis and optimization** using the **Yosys tool** to generate a more efficient hardware description. The design is then subjected to **simulation and verification** through tools such as **Icarus Verilog** and **GTKWave** to ensure functionality and correctness.

In the next phase, **static timing analysis** and **logic-level synthesis** are performed using **OpenSTA** and **Qflow**, followed by the **physical design** stage using **OpenROAD** for place-and-route. The final output is the **GDSII file**, which represents the layout of the ALU in its physical form, ready for fabrication.

This comprehensive flow emphasizes key concepts in **VLSI design**, offering a hands-on approach to developing a **robust and optimized digital system**.

2. Introduction

The **Arithmetic Logic Unit (ALU)** is a fundamental component in digital systems, tasked with executing various **arithmetic** and **logical operations** such as **addition**, **subtraction**, **multiplication**, **division**, as well as **AND**, **OR**, and **NOT** operations. In this project, a **4-bit ALU** has been designed to perform these operations efficiently.

The design process follows a comprehensive **RTL to GDS (Register Transfer Level to GDSII) design flow**. The first step involves writing the **Verilog code** for the ALU, describing its functionality at the **RTL level**. Next, a **testbench** is developed in Verilog to verify the correctness of the ALU's operations through **simulation** using **Icarus Verilog (iverilog)**. The simulation results are then analyzed using **GTKWave** to confirm that the ALU behaves as expected.

Once the functional verification is complete, the design moves to **logic synthesis**, where the Verilog code is synthesized using **Yosys** to generate an optimized gate-level description of the ALU. **Static Timing Analysis (STA)** is then performed using **OpenSTA** to ensure that the design meets the required timing constraints.

The final stage of the design process involves **Physical Design (PD)**, where the **OpenROAD** tool is used for **place-and-route**, generating the physical layout of the ALU. The output of this stage is the **GDSII file**, which represents the ALU's physical layout, ready for fabrication.

The entire process is executed using the **SkyWater 130nm HD cell library**, allowing for an efficient and modern implementation of the ALU. While this project focuses on the design of a 4-bit ALU, the methodology can be extended to support larger bit-widths, such as 8-bit or 16-bit ALUs, through modular adjustments.

3. Objectives

Design and Implement a 4-bit ALU:
To design a 4-bit Arithmetic Logic Unit (ALU) capable of performing basic arithmetic operations (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT). The ALU should be optimized for functionality, reliability, and correctness.

Develop RTL Code Using Verilog:
To write the **Register Transfer Level (RTL)** code for the ALU using **Verilog** to describe the functional operations of the ALU at a hardware description level.

Verify Design Using Testbench and Simulation:
To develop a **testbench** in Verilog for functional verification and perform simulations using **Icarus Verilog (iverilog)** and **GTKWave** to validate that the ALU operates as expected under different test conditions.

Optimize Design Through Logic Synthesis:
To synthesize the Verilog design into a gate-level representation using **Yosys**, ensuring the design is optimized for area, timing, and power consumption.

Perform Static Timing Analysis (STA):
To conduct **Static Timing Analysis (STA)** using **OpenSTA** to ensure that the design meets the required timing constraints and operates within the desired clock speed.

Complete Physical Design Using Place-and-Route:
To use the **OpenROAD** tool to perform **place-and-route** operations, generating the final **GDSII file** that represents the physical layout of the ALU, suitable for fabrication.

Utilize SkyWater 130nm HD Cell Library:
To implement the ALU design using the **SkyWater 130nm HD cell library**, ensuring compatibility with modern fabrication processes and maximizing performance within the constraints of the chosen technology.

Provide Scalability and Future Extensions:
To create a design methodology that can be extended to support larger ALUs (e.g., 8-bit or 16-bit), promoting scalability and the possibility of future enhancements in performance and functionality.

4. System Design and Methodology

The design of the 4-bit ALU follows the comprehensive **RTL to GDS** flow, ensuring functionality, performance, and manufacturability. The methodology includes the following stages:

Specification and Design:
The ALU's functionality is defined to support various **arithmetic** (addition, subtraction, multiplication, division) and **logical** (AND, OR, NOT) operations. The design is structured to be modular, allowing future extension to an 8-bit or 16-bit ALU.

RTL Design in Verilog:
The ALU is described using **Verilog** at the **Register Transfer Level (RTL)**. The Verilog code incorporates multiplexers and conditional logic to implement the desired operations based on control signals, ensuring scalability and readability.

Testbench Development and Simulation:
A **testbench** is created in Verilog to simulate the ALU's functionality. The testbench is applied with various test cases, including edge cases, to ensure correctness. **Icarus Verilog (iverilog)** is used for simulation, and **GTKWave** is used for waveform verification.

Logic Synthesis:
After successful simulation, the **Verilog code** is synthesized into a gate-level netlist using **Yosys**. The synthesis process optimizes the design for area, power, and timing, ensuring that the ALU meets the required performance targets.

Static Timing Analysis (STA):
Static Timing Analysis (STA) is performed using **OpenSTA** to ensure the design meets the timing constraints. The STA checks for any timing violations (setup or hold time) and ensures the design is operational within the specified clock cycle.

Physical Design (PD) and Place-and-Route:
The synthesized netlist is then passed to the **Physical Design (PD)** stage, where the design is physically implemented and optimized. This phase is carried out using the **OpenROAD** tool and involves several sub-steps:

Floorplanning:
The floorplanning phase involves defining the overall layout of the chip, including the placement of the ALU and the necessary surrounding logic. This step optimizes the use of available silicon area while ensuring efficient power distribution.

Power Distribution Network (PDN):
In this step, the **Power Distribution Network (PDN)** is designed to ensure that the ALU receives stable and sufficient power throughout the chip. The PDN is critical for minimizing power-related issues like voltage fluctuations.

Placement:
The **placement** phase uses **OpenROAD** to place the logic cells within the defined floorplan. The algorithm ensures that the cells are positioned optimally to minimize wirelength and reduce timing delays.

Clock Tree Synthesis (CTS):
Clock Tree Synthesis (CTS) is performed to distribute the clock signal to all parts of the ALU. The goal is to minimize clock skew, ensuring that all components receive the clock signal at the same time, ensuring synchronous operation.

Routing:
After placement and CTS, the **routing** phase connects the cells with metal layers to create the electrical paths. **OpenROAD** ensures that routing is efficient and that signal delays are minimized.

Signoff:
The final phase of PD is **signoff**, which ensures that the design is ready for manufacturing. This involves performing **Design Rule Checks (DRC)**, **Layout Versus Schematic (LVS)** checks, and verifying **Electromigration (EM)** issues. The result is a **GDSII file**, representing the final layout of the ALU.

Utilization of SkyWater 130nm HD Cell Library:
The ALU design is implemented using the **SkyWater 130nm HD cell library**, optimized for the 130nm process node, ensuring compatibility with modern semiconductor fabrication techniques.

Scalability for Future Enhancements:
The design is modular, allowing easy scalability to higher bit-widths (e.g., 8-bit or 16-bit ALU) with minimal adjustments. This also opens up the possibility of incorporating additional advanced operations like floating-point arithmetic in future extensions.

5. Implementation

The implementation of the **4-bit ALU** follows a structured approach, transitioning from **RTL design** to **GDSII layout**, ensuring a seamless **front-end to back-end VLSI design flow**.

5.1 RTL Design and Testbench

The **Register-Transfer Level (RTL) design** of the **4-bit ALU** is implemented in **Verilog**, supporting **addition, subtraction, multiplication, division, AND, OR, and NOT** operations. The ALU is designed to operate synchronously with a clock signal, ensuring stable operation in real hardware.

ALU Design in Verilog

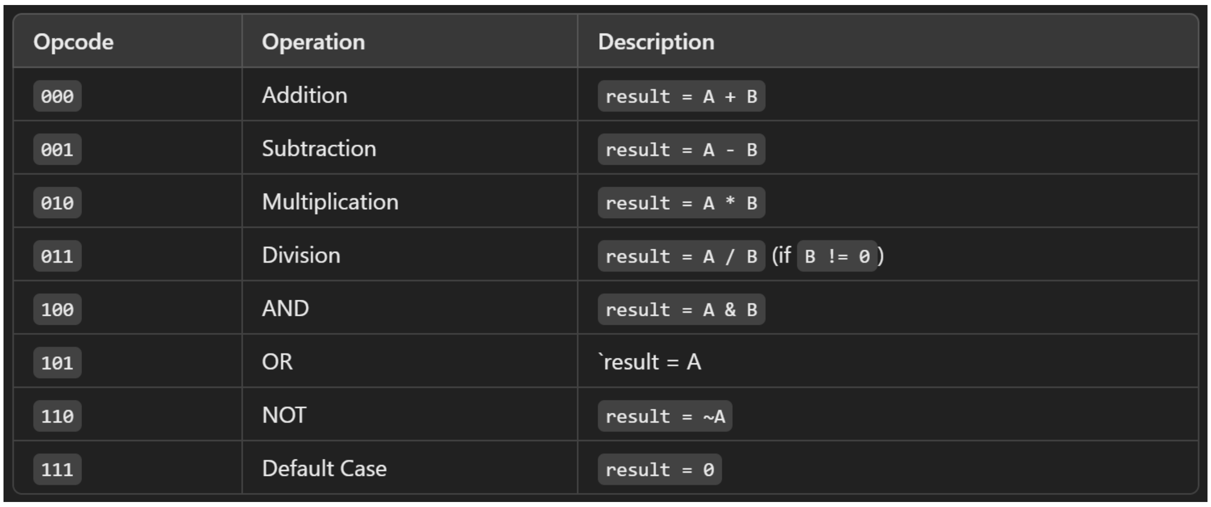
The ALU module takes two **4-bit inputs (A and B)** and a **3-bit opcode**, which determines the arithmetic or logical operation to be performed. The results are stored in a **4-bit output** (result), and an additional **carry-out** signal is used for operations requiring overflow detection.

Code Structure & Explanation

```
module ALU (
input [3:0] A, B,       // 4-bit input operands
input [2:0] opcode,     // 3-bit operation selector
input clk,              // Clock signal
output reg [3:0] result,// 4-bit result output
output reg carry_out    // Carry out for overflow detection
);
```

- ✓ The **inputs** A and B are **4-bit operands**.
- ✓ The **opcode** is a **3-bit selector**, allowing up to **8 operations**.
- ✓ The **clock signal (clk)** ensures synchronous operation.
- ✓ The **result** is stored in a **4-bit register** to retain values between clock cycles.
- ✓ The **carry_out** flag is used for **addition and subtraction overflow detection**.

The ALU supports the following operations, mapped to specific opcodes:



Opcode	Operation	Description
`000`	Addition	`result = A + B`
`001`	Subtraction	`result = A - B`
`010`	Multiplication	`result = A * B`
`011`	Division	`result = A / B` (if `B != 0`)
`100`	AND	`result = A & B`
`101`	OR	`result = A
`110`	NOT	`result = ~A`
`111`	Default Case	`result = 0`

```
  always @(posedge clk) begin
    carry_out = 0; // Reset carry_out
    case (opcode)
      3'b000: {carry_out, result} = A + B;        // Addition with carry detection
      3'b001: {carry_out, result} = A - B;        // Subtraction with borrow detection
      3'b010: result = A * B;                     // Multiplication
      3'b011: result = (B != 0) ? A / B : 4'b0000;  // Division (handling divide by zero)
      3'b100: result = A & B;                     // Logical AND
      3'b101: result = A | B;                     // Logical OR
      3'b110: result = ~A;                        // Logical NOT (ignores B)
      default: result = 4'b0000;                  // Default case
    endcase
  end
endmodule
```

- ✓ The **ALU executes the operation** based on the opcode inside an **always block triggered by the clock** (posedge clk).
- ✓ **Addition and subtraction** use **carry-out detection** to flag overflow conditions.
- ✓ **Multiplication and division** are performed directly, with division including a **zero-check** to prevent errors.
- ✓ **Logical operations** (AND, OR, NOT) modify the input bits as expected.
- ✓ The **default case** ensures that the ALU outputs 0 for any undefined opcode.

Testbench for ALU Verification

A **testbench** is created to verify ALU functionality by providing **stimuli (input test cases)** and monitoring the **output responses**. The testbench is designed to simulate all ALU operations sequentially and check if the results are correct.

Code Structure & Explanation

```
module ALU_TB;

  reg [3:0] A, B;        // 4-bit test inputs
  reg [2:0] opcode;      // 3-bit opcode
  reg clk;               // Clock signal
  wire [3:0] result;     // Output from ALU
  wire carry_out;        // Carry out flag

  // Instantiate the ALU module
  ALU uut (
    .A(A),
    .B(B),
    .opcode(opcode),
    .clk(clk),
    .result(result),
    .carry_out(carry_out)
  );
```

- ✓ **Registers (reg)** store **test inputs** and **control signals**.
- ✓ **Wires (wire)** receive output from the ALU.
- ✓ The **ALU module (uut) is instantiated**, linking the testbench to the design under test.

```
  // Clock generation: toggles every 5 time units
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end
```

- ✓ The **clock (clk) toggles every 5-time units**, creating a **10-time unit clock period**.
- ✓ This ensures **synchronous execution** of ALU operations.

```
  initial begin
    $dumpfile("ALU.vcd"); // File for waveform dump
    $dumpvars(0, ALU_TB); // Store testbench signals
    $display("Testing Clocked 4-bit ALU...");
    $display("Opcode | A (Dec) | B (Dec) | Result (Dec) | Carry Out");
```

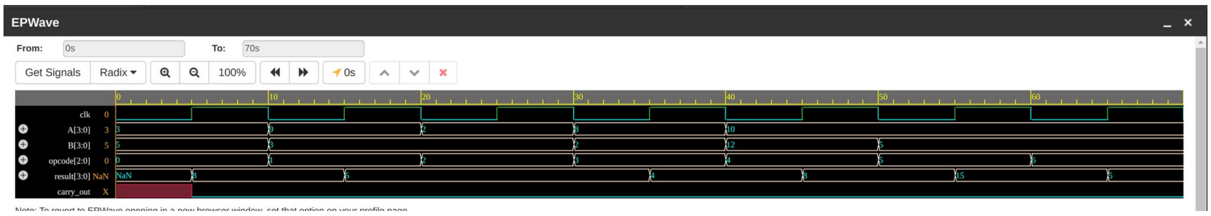
- ✓ **Waveform generation (ALU.vcd)** allows debugging in **GTKWave**.
- ✓ **Formatted console output** provides a readable summary of the tests.

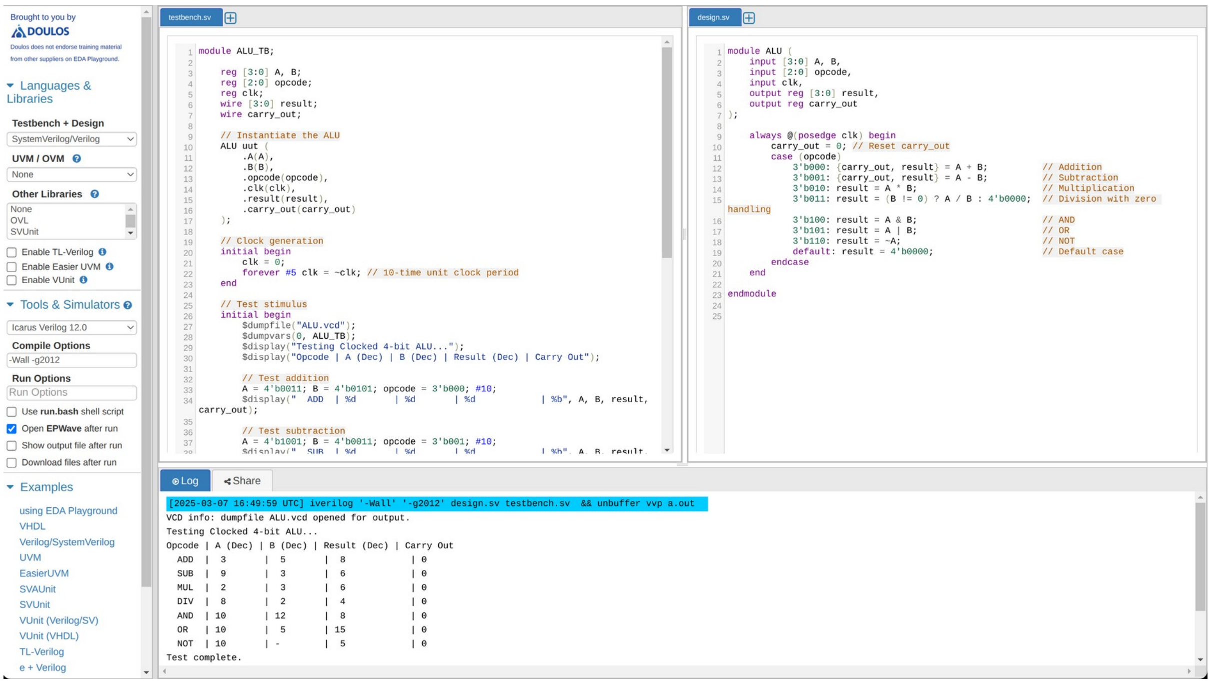
```
    // Test addition
    A = 4'b0011; B = 4'b0101; opcode = 3'b000; #10;
    $display("  ADD  | %d      | %d      | %d           | %b", A, B, result, carry_out);
```

```
        // Test subtraction
        A = 4'b1001; B = 4'b0011; opcode = 3'b001; #10;
        $display("  SUB  | %d       | %d       | %d           | %b", A, B, result, carry_out);

        // Test multiplication
        A = 4'b0010; B = 4'b0011; opcode = 3'b010; #10;
        $display("  MUL  | %d       | %d       | %d           | %b", A, B, result, carry_out);

        // Test division
        A = 4'b1000; B = 4'b0010; opcode = 3'b011; #10;
        $display("  DIV  | %d       | %d       | %d           | %b", A, B, result, carry_out);

        // Test AND
        A = 4'b1010; B = 4'b1100; opcode = 3'b100; #10;
        $display("  AND  | %d       | %d       | %d           | %b", A, B, result, carry_out);

        // Test OR
        A = 4'b1010; B = 4'b0101; opcode = 3'b101; #10;
        $display("  OR   | %d       | %d       | %d           | %b", A, B, result, carry_out);

        // Test NOT
        A = 4'b1010; opcode = 3'b110; #10;
        $display("  NOT  | %d       | -        | %d           | %b", A, result, carry_out);

        $display("Test complete.");
        $finish;
    end
endmodule
```

- ✓ Each ALU operation is tested **sequentially** with a **10-time unit delay** (#10).
- ✓ The **results are displayed in decimal format** for easy debugging.

Simulation and Verification of RTL design with Testbench:





Testbench + Design: SystemVerilog/Verilog
UVM / OVM: None
Other Libraries: None, OVL, SVUnit
☐ Enable TL-Verilog
☐ Enable Easier UVM
☐ Enable VUnit
Icarus Verilog 12.0
Compile Options: -Wall -g2012
Run Options: Run Options
☐ Use **run.bash** shell script
☑ Open **EPWave** after run
☐ Show output file after run
☐ Download files after run

```
module ALU_TB;

    reg [3:0] A, B;
    reg [2:0] opcode;
    reg clk;
    wire [3:0] result;
    wire carry_out;

    // Instantiate the ALU
    ALU uut (
        .A(A),
        .B(B),
        .opcode(opcode),
        .clk(clk),
        .result(result),
        .carry_out(carry_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10-time unit clock period
    end

    // Test stimulus
    initial begin
        $dumpfile("ALU.vcd");
        $dumpvars(0, ALU_TB);
        $display("Testing Clocked 4-bit ALU...");
        $display("Opcode | A (Dec) | B (Dec) | Result (Dec) | Carry Out");

        // Test addition
        A = 4'b0011; B = 4'b0101; opcode = 3'b000; #10;
        $display("  ADD  | %d       | %d       | %d           | %b", A, B, result,
carry_out);

        // Test subtraction
        A = 4'b1001; B = 4'b0011; opcode = 3'b001; #10;
        $display("  SUB  | %d       | %d       | %d           | %b", A, B, result,
```

```
module ALU (
    input [3:0] A, B,
    input [2:0] opcode,
    input clk,
    output reg [3:0] result,
    output reg carry_out
);

    always @(posedge clk) begin
        carry_out = 0; // Reset carry_out
        case (opcode)
            3'b000: {carry_out, result} = A + B;          // Addition
            3'b001: {carry_out, result} = A - B;          // Subtraction
            3'b010: result = A * B;                       // Multiplication
            3'b011: result = (B != 0) ? A / B : 4'b0000;  // Division with zero
handling
            3'b100: result = A & B;                       // AND
            3'b101: result = A | B;                       // OR
            3'b110: result = ~A;                          // NOT
            default: result = 4'b0000;                    // Default case
        endcase
    end

endmodule
```

```
[2025-03-07 16:49:59 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile ALU.vcd opened for output.
Testing Clocked 4-bit ALU...
Opcode | A (Dec) | B (Dec) | Result (Dec) | Carry Out
  ADD  |  3       |  5       |  8           | 0
  SUB  |  9       |  3       |  6           | 0
  MUL  |  2       |  3       |  6           | 0
  DIV  |  8       |  2       |  4           | 0
  AND  | 10       | 12       |  8           | 0
  OR   | 10       |  5       | 15           | 0
  NOT  | 10       | -        |  5           | 0
Test complete.
```

Conclusion

The **RTL design and testbench verification confirm the correct functionality** of the 4-bit ALU. The **waveform output in** further validates the results. This step forms the **foundation for logic synthesis**, ensuring that the ALU behaves as expected before moving to **hardware implementation**.

5.2 Logic Synthesis

Logic synthesis is a crucial step in the RTL-to-GDS flow, where the high-level Verilog description of the ALU is converted into a gate-level netlist using a standard cell library. In this project, Yosys was used for logic synthesis, targeting the **SkyWater 130nm HD cell library**.

Input Files for Logic Synthesis

The synthesis process requires several input files to define the design, constraints, and technology parameters:

- **ALU.v** → Verilog RTL description of the 4-bit ALU.
- **sky130_fd_sc_hd__tt_025C_1v80.lib** → Standard cell library file containing the technology-specific gates used for synthesis.
- **Synthesis Constraints File** → Defines timing constraints such as clock period, input arrival times, and output transition times.

Logic Synthesis Flow

The synthesis process was executed using the following Yosys script:

```
# Step 1: Read the RTL design file (ALU Verilog file)
read_verilog ALU.v

# Step 2: Check and set the top module for the design
hierarchy -check -top ALU

# Step 3: Read the library file for the technology
read_liberty -lib sky130_fd_sc_hd__tt_025C_1v80.lib

# Step 4: Run synthesis
synth

# Step 5: Map flip-flops to standard cells using the specified library
dfflibmap -liberty sky130_fd_sc_hd__tt_025C_1v80.lib

# Step 6: Perform technology mapping using the ABC command
abc -liberty sky130_fd_sc_hd__tt_025C_1v80.lib

# Step 7: Clean up the design by removing unnecessary logic (optional)
opt_clean

# Step 8: Write out the synthesized gate-level netlist (Verilog format)
write_verilog ALU_synth.v
```

This script follows a structured flow:

1. **Reading RTL Design** → Loads the Verilog description of the ALU.
2. **Hierarchy Check** → Ensures the ALU module is correctly defined as the top module.
3. **Library Loading** → Imports the SkyWater 130nm standard cell library.
4. **Synthesis Process** → Converts RTL code into a gate-level representation.
5. **Technology Mapping** → Maps flip-flops and logic gates to standard cells.
6. **Optimization** → Removes redundant logic and optimizes the design.
7. **Output Generation** → Saves the synthesized netlist for further analysis.

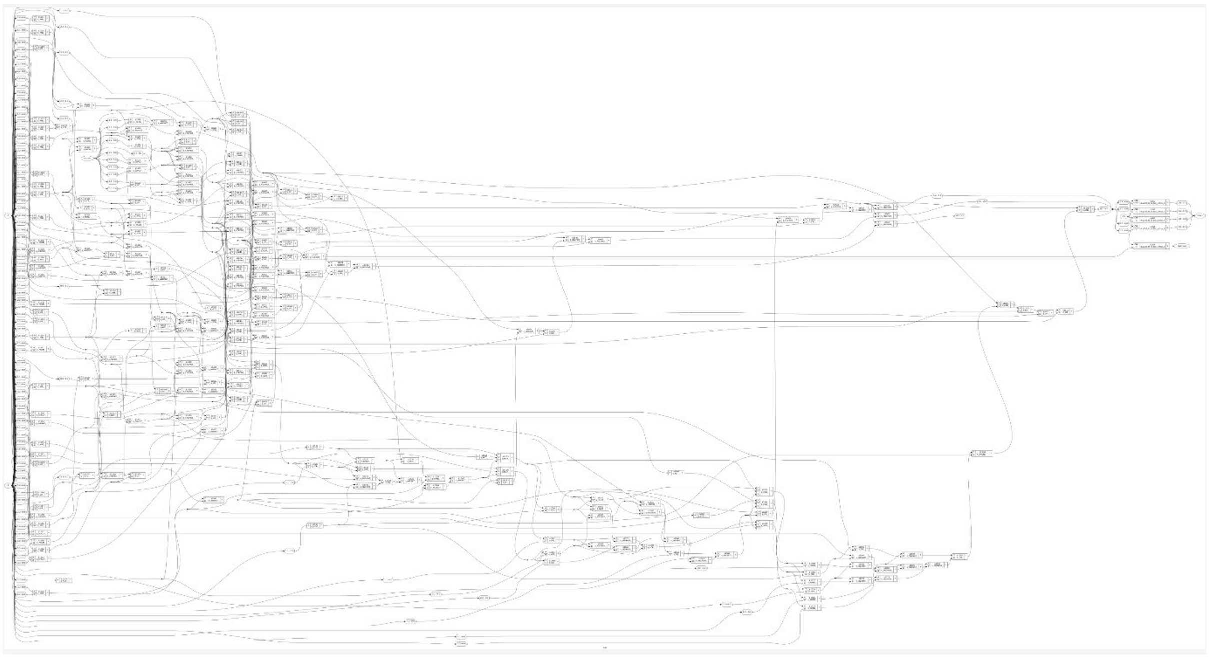
Output Files Generated

After running synthesis, the following output files were obtained:

- **BLIF File** → Berkeley Logic Interchange Format representation of the synthesized design.
- **Dot View** → A graphical visualization of the synthesized logic circuit.
- **Synthesized Netlist (ALU_synth.v)** → The gate-level Verilog representation of the ALU.

Graphical Representation of Synthesized ALU

*Below is the **Dot View** representation of the synthesized ALU circuit:*



This visualization helps in understanding the logical connections and complexity of the synthesized design.

Conclusion

The logic synthesis process successfully converted the high-level RTL design of the 4-bit ALU into a gate-level netlist using the SkyWater 130nm technology library. The synthesized netlist was optimized and mapped to standard cells, resulting in a gate-level representation that is ready for further verification and analysis. This process laid the foundation for the subsequent steps of Static Timing Analysis and Physical Design.

5.3 Static Timing Analysis (STA)

Static Timing Analysis (STA) is the process of verifying the timing of a digital design by analyzing its timing paths. It checks if the design meets the required timing constraints, such as setup and hold times for flip-flops, by calculating delays and comparing them to the constraints defined in the design. STA helps to ensure the design operates correctly at the intended clock frequency.

Input Files for STA

For Static Timing Analysis, the following input files are used to define the timing constraints, technology library, and synthesized design:

- **ALU_synth.v** → Gate-level Verilog netlist generated from the synthesis step, representing the logic of the 4-bit ALU.
- **sky130_fd_sc_hd__tt_025C_1v80.lib** → Standard cell library file containing timing models for the technology used in the design.
- **constraints.sdc** → SDC (Synopsys Design Constraints) file that defines timing constraints such as clock period, input arrival times, and output timing requirements.

STA Flow

The STA process was executed using the following script:

```
# Initialize STA environment
sta

# Step 1: Read Liberty file for cell library information
read_liberty sky130_fd_sc_hd__tt_025C_1v80.lib

# Step 2: Read the synthesized Verilog netlist
read_verilog ALU_synth.v

# Step 3: Link the design
link ALU

# Step 4: Source the SDC file for timing constraints
source constraints.sdc

# Step 5: Perform timing checks
report_checks -path_delay max -format full
report_checks -path_delay min -format full

# Step 6: Power Check
report_power
```

This script follows a structured flow:

1. **Environment Initialization** → Sets up the STA tool environment.
2. **Library and Netlist Loading** → Loads the Liberty file for cell information and the synthesized Verilog netlist.
3. **Design Linking** → Links the design so that it can be analyzed.

4. **Constraints Application** → Sources the SDC file, applying the timing constraints to the design.
5. **Timing Checks** → Performs maximum and minimum path delay checks to verify if the design meets the timing requirements.
6. **Power Report** → Provides an estimate of the design's power consumption.

Outputs from STA

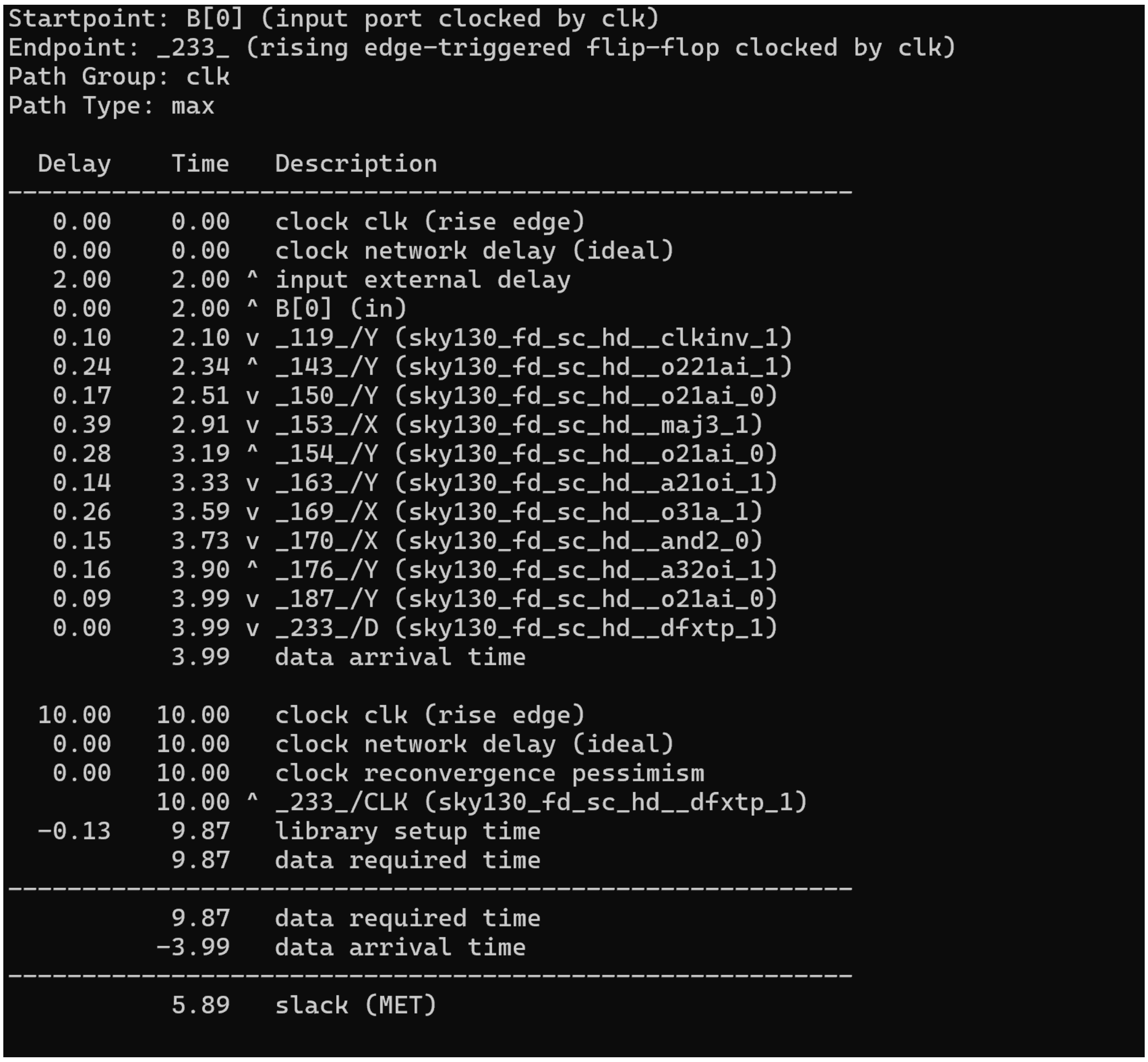
After running the STA, the following output files were obtained:

- **Timing Check Reports** → These reports show the maximum and minimum path delays and check whether the design meets the specified timing constraints.
- **Power Report** → This report provides an estimation of the power consumption of the design.

Graphical Representation of STA Results

Below are the screenshots of the STA results from the following commands:

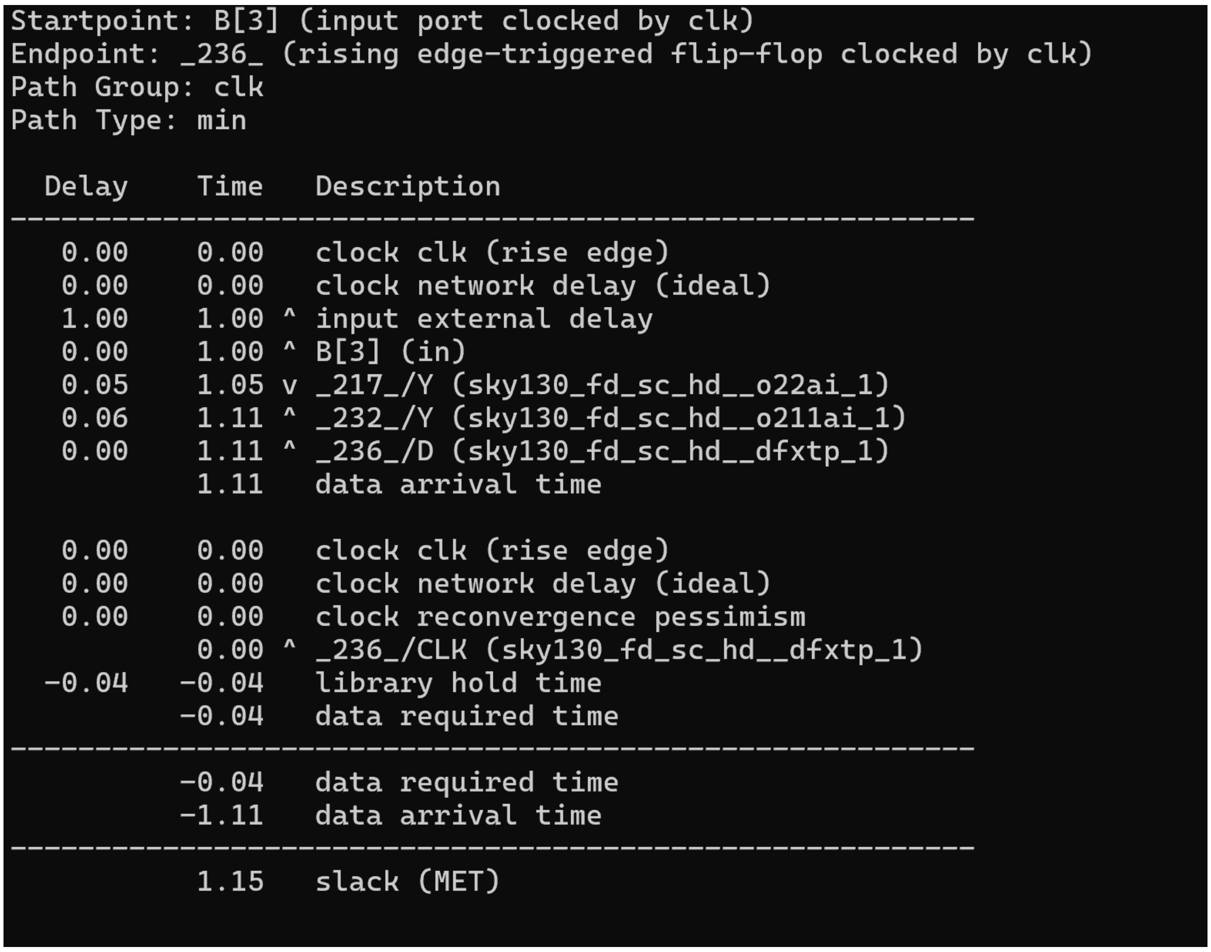
- report_checks -path_delay max -format full



```
Startpoint: B[0] (input port clocked by clk)
Endpoint: _233_ (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

  Delay    Time   Description
---------------------------------------------------------
   0.00    0.00   clock clk (rise edge)
   0.00    0.00   clock network delay (ideal)
   2.00    2.00 ^ input external delay
   0.00    2.00 ^ B[0] (in)
   0.10    2.10 v _119_/Y (sky130_fd_sc_hd__clkinv_1)
   0.24    2.34 ^ _143_/Y (sky130_fd_sc_hd__o221ai_1)
   0.17    2.51 v _150_/Y (sky130_fd_sc_hd__o21ai_0)
   0.39    2.91 v _153_/X (sky130_fd_sc_hd__maj3_1)
   0.28    3.19 ^ _154_/Y (sky130_fd_sc_hd__o21ai_0)
   0.14    3.33 v _163_/Y (sky130_fd_sc_hd__a21oi_1)
   0.26    3.59 v _169_/X (sky130_fd_sc_hd__o31a_1)
   0.15    3.73 v _170_/X (sky130_fd_sc_hd__and2_0)
   0.16    3.90 ^ _176_/Y (sky130_fd_sc_hd__a32oi_1)
   0.09    3.99 v _187_/Y (sky130_fd_sc_hd__o21ai_0)
   0.00    3.99 v _233_/D (sky130_fd_sc_hd__dfxtp_1)
           3.99   data arrival time

  10.00   10.00   clock clk (rise edge)
   0.00   10.00   clock network delay (ideal)
   0.00   10.00   clock reconvergence pessimism
          10.00 ^ _233_/CLK (sky130_fd_sc_hd__dfxtp_1)
  -0.13    9.87   library setup time
           9.87   data required time
---------------------------------------------------------
           9.87   data required time
          -3.99   data arrival time
---------------------------------------------------------
           5.89   slack (MET)
```

Timing Path Analysis: The timing path starts from input port **B[0]**, clocked by **clk**, and ends at the flip-flop ***233***, also clocked by **clk**. The analysis shows the total path delay of **3.99ns** (data arrival time) and a required time of **9.87ns**. The resulting **slack** is **5.89ns**, indicating that the design satisfies the timing constraints with positive slack.

- report_checks -path_delay min -format full



```
Startpoint: B[3] (input port clocked by clk)
Endpoint: _236_ (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

  Delay    Time   Description
---------------------------------------------------------
   0.00    0.00   clock clk (rise edge)
   0.00    0.00   clock network delay (ideal)
   1.00    1.00 ^ input external delay
   0.00    1.00 ^ B[3] (in)
   0.05    1.05 v _217_/Y (sky130_fd_sc_hd__o22ai_1)
   0.06    1.11 ^ _232_/Y (sky130_fd_sc_hd__o211ai_1)
   0.00    1.11 ^ _236_/D (sky130_fd_sc_hd__dfxtp_1)
           1.11   data arrival time

   0.00    0.00   clock clk (rise edge)
   0.00    0.00   clock network delay (ideal)
   0.00    0.00   clock reconvergence pessimism
           0.00 ^ _236_/CLK (sky130_fd_sc_hd__dfxtp_1)
  -0.04   -0.04   library hold time
          -0.04   data required time
---------------------------------------------------------
          -0.04   data required time
          -1.11   data arrival time
---------------------------------------------------------
           1.15   slack (MET)
```

Timing Path Analysis: The timing path starts from input port **B[3]**, clocked by **clk**, and ends at the flip-flop ***236***, also clocked by **clk**. The analysis shows the total path delay of **1.11ns** (data arrival time) and a required time of **-0.04ns**. The resulting **slack** is **1.15ns**, indicating that the design meets the timing requirements with positive slack.

Conclusion

The Static Timing Analysis was performed on the synthesized netlist to ensure the design meets the required timing constraints. The timing paths were analyzed for both maximum and minimum delays, revealing positive slack, which indicates that the design is timing-viable and will function correctly within the specified clock period. This ensures that the ALU design is ready for the next stage of physical design and implementation.

5.4 Physical Design

Physical design is the process of translating the synthesized design into a layout representation that can be fabricated on silicon. This step ensures that the design meets physical constraints like area, power, and signal integrity while adhering to manufacturing rules.

It involves several key processes:

- Floorplan
- Power Distribution Network
- Placement
- Clock Tree Synthesis
- Routing

5.4.1 Floorplan

Floorplanning is the first step in the physical design process, where the **die area, core area, and placement of standard cells, macros, and I/O pins** are determined. This step is crucial for optimizing **performance, area, power, and routability**. A well-structured floorplan ensures that the design meets timing constraints while minimizing congestion and power issues.

The key objectives of floorplanning include:

- Defining the **die and core area**
- Placing **I/O pins and macros** efficiently
- Ensuring **power distribution** through **tapcell insertion**
- Preparing the design for **placement and routing**

Input Files Used in Floorplanning

Before running the floorplanning script, several input files are required to define constraints, technology parameters, and design-specific configurations:

Technology & Standard Cell Library Files

- sky130_fd_sc_hd__tt_025C_1v80.lib → Standard cell **timing library**
- sky130_fd_sc_hd_merged.lef → **Technology LEF** defining metal layers, DRC rules, and cell placements
- sky130_fd_sc_hd.tlef → **Technology LEF file** with track definitions
- sky130hd.tracks → Track definitions for standard cell placement and routing
- sky130hd.vars → Stores **technology-specific variables** (e.g., metal layers, cell densities)
- flow_helpers.tcl → Assists in managing the **physical design flow** and constraints

Design-Specific Files

- ALU_synth.v → Synthesized **Verilog netlist** from logic synthesis
- constraints.sdc → Defines **clock constraints** and timing requirements
- ALU_PD.tcl → Script to manage the placement process.

Helper Scripts

- helpers.tcl → Utility script containing **reusable functions** for physical design tasks
- flow_helpers.tcl → Assists in managing the **physical design flow** and constraints

Execution of Floorplanning Flow (ALU_PD.tcl)

The ALU_PD.tcl script is the primary driver for executing the floorplanning process. It sources helper scripts, defines design parameters, and calls the Flow_Floorplan.tcl script to perform the actual floorplan operations. Below is a step-by-step breakdown of its execution.

Breakdown of ALU_PD.tcl Script

The ALU_PD.tcl script automates the floorplanning process and integrates all required input files. Below is a breakdown of its key sections:

```
source "helpers.tcl"
source "flow_helpers.tcl"
source "sky130hd.vars"
```

- ✓ These commands load **helper scripts** containing pre-defined functions and technology variables.

```
set synth_verilog "ALU_synth.v"
set design "ALU"
set top_module "ALU"
set sdc_file "constraints.sdc"
```

- ✓ Defines the **synthesized Verilog file**, design name, and **timing constraints file**.

```
set die_area {0 0 65 65}
set core_area {2.3 2.72 59.96 59.89}
```

Specifies the physical dimensions of the die and core:

- ✓ ***Die Area****: 65x65 units (~4202.9 µm²)*
- ✓ ***Core Area****: 59.96x59.89 units (~3315.1 µm²)*

Sets the **placement density** (92%) to control **cell packing** and avoid congestion.

```
source -echo "Flow_Floorplan.tcl"
```

- ✓ Calling of Flow_Floorplan.tcl script

Content of (Flow_Floorplan.tcl)

The ***Flow_Floorplan.tcl*** *script is responsible for generating the Floorplan. Below is the script:*

```
# Assumes flow_helpers.tcl has been read.
read_libraries
read_verilog $synth_verilog
link_design $top_module
read_sdc $sdc_file

set_thread_count [exec getconf _NPROCESSORS_ONLN]
# Temporarily disable sta's threading due to random failures
sta::set_thread_count 1

utl::metric "IFP::ord_version" [ord::openroad_git_describe]
# Note that sta::network_instance_count is not valid after tapcells are added.
utl::metric "IFP::instance_count" [sta::network_instance_count]

initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area

source $tracks_file

# Remove buffers inserted by synthesis
remove_buffers

################################################################
# IO Placement (random)
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

################################################################
# Macro Placement
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}

################################################################
# Tapcell insertion
eval tapcell $tapcell_args
```

Breakdown of Floorplan-Specific Commands

```
initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area
```

- ✓ Defines the floorplan by specifying:
  - **Die area** → Total chip area
  - **Core area** → Region where standard cells and macros are placed
- ✓ Ensures sufficient spacing for routing and power distribution.

```
source $tracks_file
```

- ✓ Loads **track definitions** for metal layers.
- ✓ Ensures proper routing grid alignment for metal connections.

```
remove_buffers
```

- ✓ Deletes redundant buffers inserted during **logic synthesis** to optimize the design for physical implementation.

```
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer
```

- ✓ Places **I/O pins randomly** on designated horizontal and vertical metal layers.
- ✓ Ensures proper pin accessibility for routing.

```
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}
```

If macros are present in the design, they are placed with:

- ✓ **Halo spacing** (gap around macros for routing)
- ✓ **Channel spacing** (extra space for interconnect routing.

```
eval tapcell $tapcell_args
write_def alu_post_tapcell.def
```

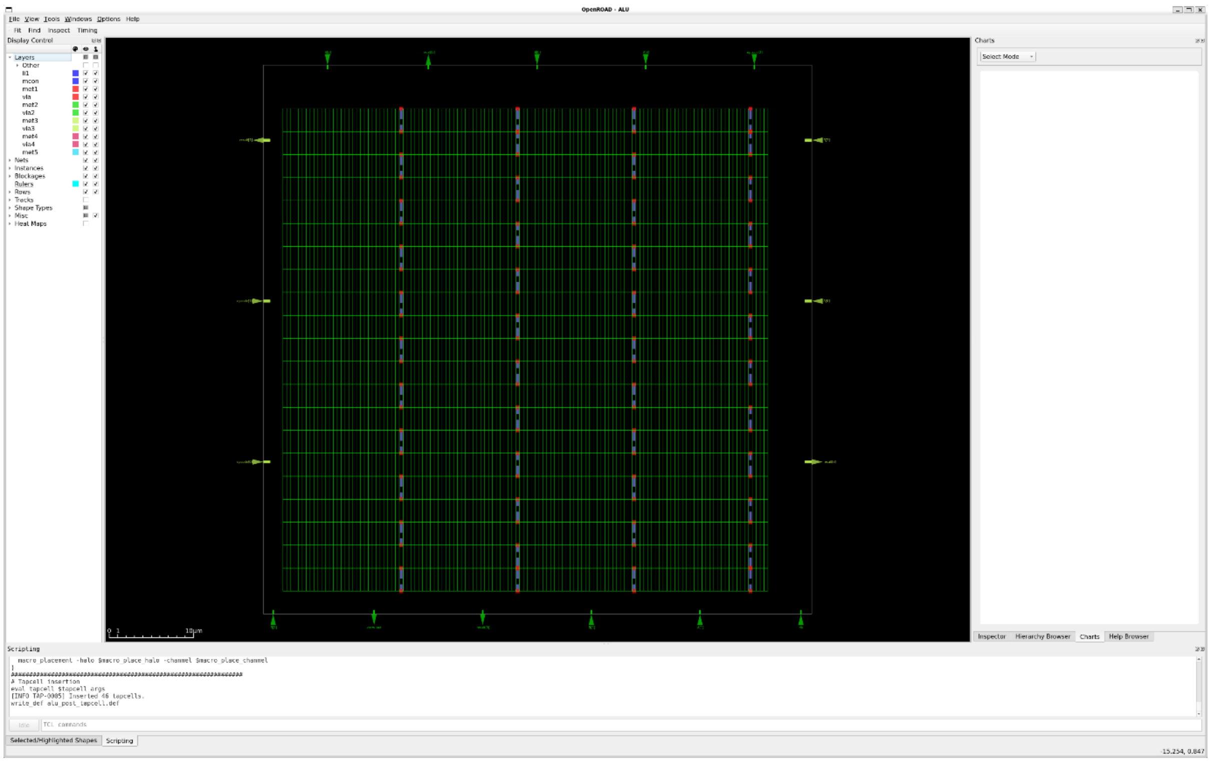
- ✓ Inserts **tapcells** to ensure power and ground connectivity across the core area.
- ✓ Generates a **DEF (Design Exchange Format) file** (alu_post_tapcell.def) for the next step in the flow.

Output Files Generated

After executing the floorplanning flow, the following output files were obtained:

- ❖ **alu_post_tapcell.def** – Contains the floorplan data after tapcell insertion.
- ❖ **Floorplan.log** – The log file capturing execution details, warnings, and errors.

Floorplan Result


OpenROAD - ALU
File View Tools Windows Options Help
Display Control
Charts
Select Mode
Scripting
Inspector Hierarchy Browser Charts Help Browser
Selected/Highlighted Shapes Scripting

```
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}
################################################################
# Tapcell insertion
eval tapcell $tapcell_args
[INFO TAP-0005] Inserted 46 tapcells.
write_def alu_post_tapcell.def
```

Conclusion

The floorplan execution was successfully completed with the following results:

- **Die area** was set to **4202.9 µm²** (coordinates: {0 0 65 65}).
- **Core area** was set to **3315.1 µm²** (coordinates: {2.3 2.72 59.96 59.89}).
- **13 layers** and **25 vias** were created from the LEF file (sky130_fd_sc_hd.tlef).
- The merged LEF file (sky130_fd_sc_hd_merged.lef) defined **441 library cells**.
- **21 rows** of **125 site unithd** were added to the design as part of the floorplan initialization.
- **Random IO pin placement** was carried out, ensuring a minimum distance of **2 tracks** between IO pins.
- No macro blocks were found, so no macro placement was performed.
- **46 tapcells** were inserted to handle power distribution and improve reliability.
- The final floorplan was written to the output file alu_post_tapcell.def.

This step of the process successfully established the groundwork for placement and routing, ensuring the proper organization of layers, vias, cells, and areas for the subsequent stages in the physical design flow.

5.4.2 Power Distribution Network (PDN)

The Power Distribution Network (PDN) is a crucial step in the physical design flow, responsible for ensuring a stable power supply to all standard cells, macros, and other components in the design. A well-designed PDN prevents issues like voltage drops (IR drop), electromigration, and power noise, ensuring the overall reliability of the circuit.

The key objectives of PDN include:

- Providing a robust and uniform power grid across the chip.
- Minimizing IR drop and electromigration risks.
- Ensuring power delivery to all standard cells and macros.
- Optimizing metal layer usage for power routing.

Input Files Used in PDN Generation

Before executing the PDN flow, several input files are required to define the power routing constraints and metal layer configurations.

Technology & Standard Cell Library Files

- **sky130_fd_sc_hd__tt_025C_1v80.lib** → Standard cell timing library.
- **sky130_fd_sc_hd_merged.lef** → Defines metal layers, DRC rules, and cell placements.
- **sky130_fd_sc_hd.tlef** → Defines track definitions for power routing.
- **sky130hd.tracks** → Specifies the track definitions for standard cell placement and routing.
- **sky130hd.vars** → Stores technology-specific variables (e.g., metal layers, densities).
- **Sky130hd.pdn.tcl** → Power grid configuration script defining power rails, straps, and connections.

Design-Specific Files

- **ALU_synth.v** → Synthesized Verilog netlist from logic synthesis.
- **constraints.sdc** → Defines clock constraints and timing requirements.

Helper Scripts

- **helpers.tcl** → Contains reusable functions for physical design tasks.
- **flow_helpers.tcl** → Assists in managing the physical design flow and constraints.

Execution of PDN Flow (ALU_PD.tcl)

The ALU_PD.tcl script automates the process of power distribution network (PDN) generation by calling the necessary flow scripts. Below is the script:

```
source "helpers.tcl"
source "flow_helpers.tcl"
source "sky130hd.vars"

# Design-specific variables
set synth_verilog "ALU_synth.v"
set design "ALU"
set top_module "ALU"  ;# Update this with your top module name
set sdc_file "constraints.sdc"

# Die and Core Area (Proportionally Adjusted for Density Control)
set die_area {0 0 65 65}       ;# Die area adjusted to ~4202.9 µm²
set core_area {2.3 2.72 59.96 59.89}  ;# Core area adjusted to ~3315.1 µm²

# Load and run the flow
source -echo "Flow_PDN.tcl"
```

Content of (Flow_PDN.tcl)

*The **Flow_PDN.tcl** script is responsible for generating the power distribution network after floorplanning. Below is the script:*

```
# Assumes flow_helpers.tcl has been read.
read_libraries
read_verilog $synth_verilog
link_design $top_module
read_sdc $sdc_file

set_thread_count [exec getconf _NPROCESSORS_ONLN]
# Temporarily disable sta's threading due to random failures
sta::set_thread_count 1

utl::metric "IFP::ord_version" [ord::openroad_git_describe]
# Note that sta::network_instance_count is not valid after tapcells are added.
utl::metric "IFP::instance_count" [sta::network_instance_count]

initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area

source $tracks_file

# Remove buffers inserted by synthesis
```

```
remove_buffers

####################################################################
# IO Placement (random)
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

####################################################################
# Macro Placement
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}

####################################################################
# Tapcell insertion
eval tapcell $tapcell_args

####################################################################
# Power distribution network insertion
source $sky130hd.pdn.tcl
pdngen
write_def alu_post_pdn.def
```

Breakdown of PDN-Specific Commands

```
source $pdn_cfg
pdngen
write_def alu_post_pdn.def
```

- ✓ **source $pdn_cfg** → Reads the power grid configuration file (sky130hd.pdn.tcl). This file defines the metal layers, strap widths, spacing, and connections for power (VDD) and ground (VSS).
- ✓ **pdngen** → Generates the power distribution network using the specified configuration. It ensures that power is evenly distributed across the core area.
- ✓ **write_def alu_post_pdn.def** → Writes the updated design with the power grid into a DEF file for further processing.

Output Files Generated

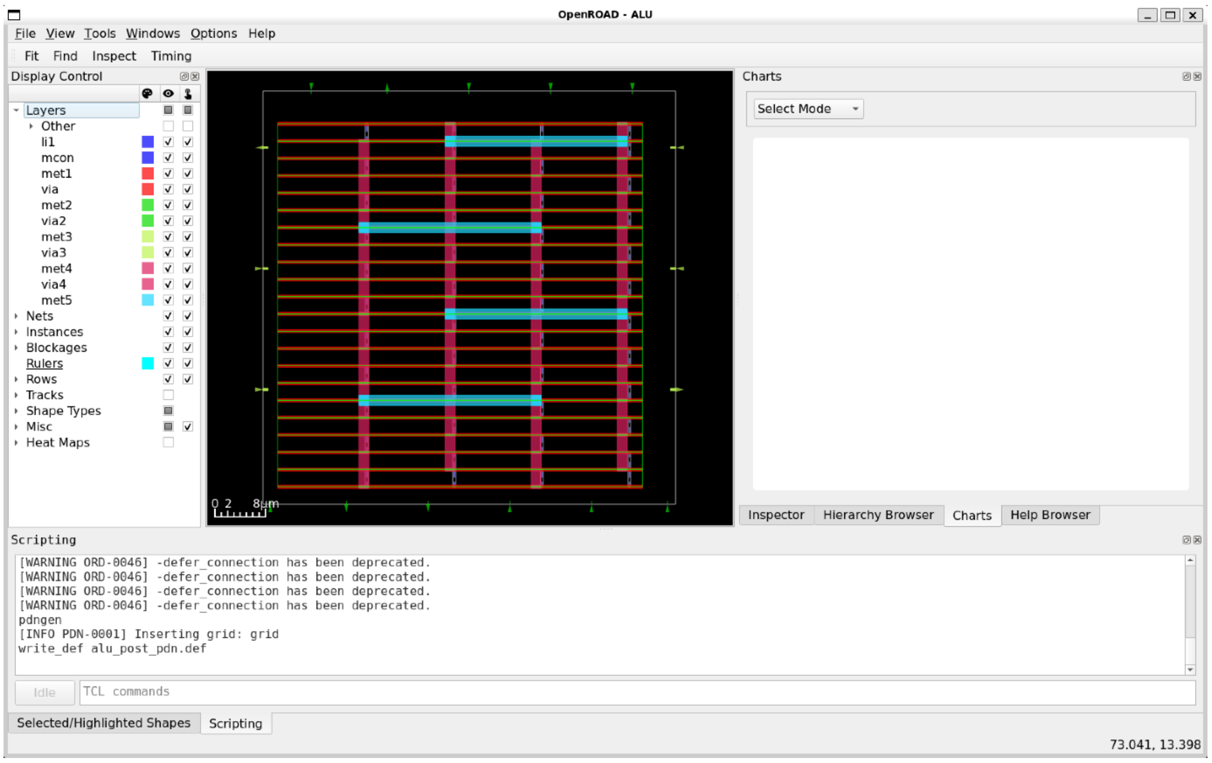
After executing the PDN flow, the following output files are obtained:

- ❖ **alu_post_pdn.def** → Contains the updated design with the power distribution network.
- ❖ **PDN.log** → Log file capturing execution details, warnings, and errors.

PDN Results

1. Combined Power and Ground Network

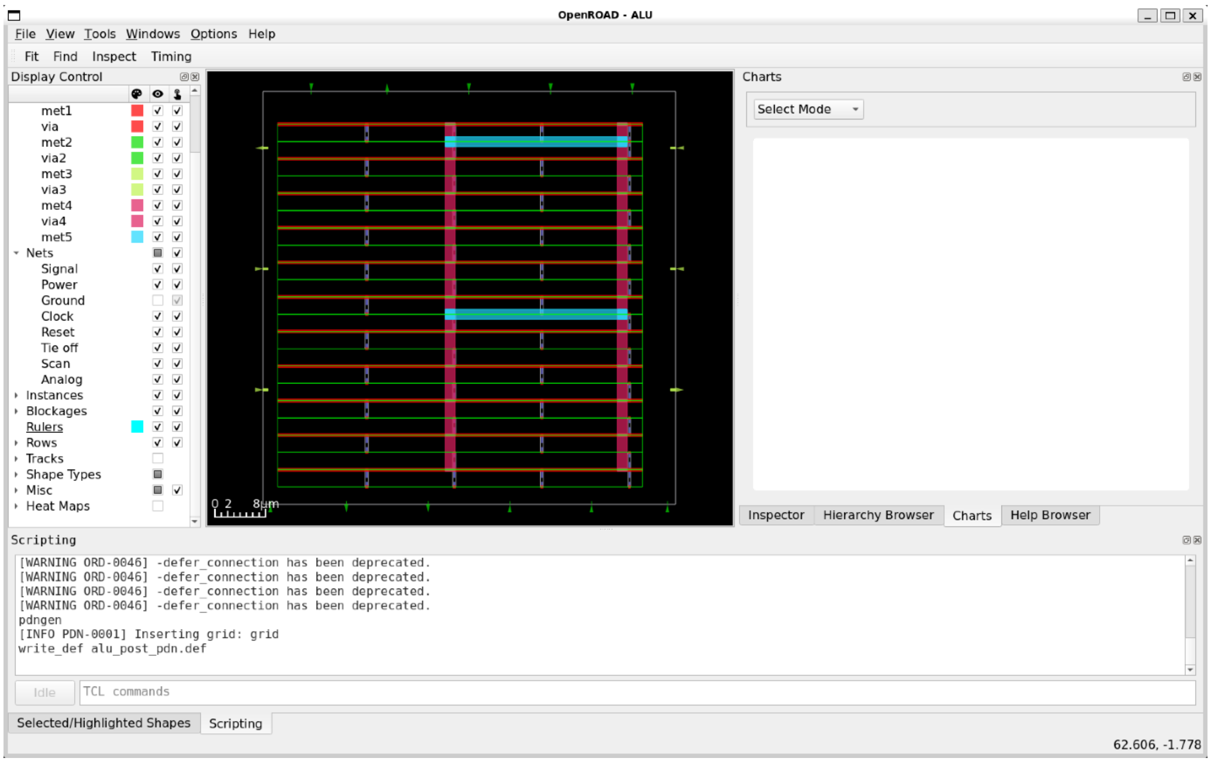
The image below represents the power and ground networks together, showing both the power (VDD) and ground (VSS) rails and their connections. This visualization allows us to observe how both networks are routed throughout the design.



```
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
pdngen
[INFO PDN-0001] Inserting grid: grid
write_def alu_post_pdn.def
```

2. Power Network Only

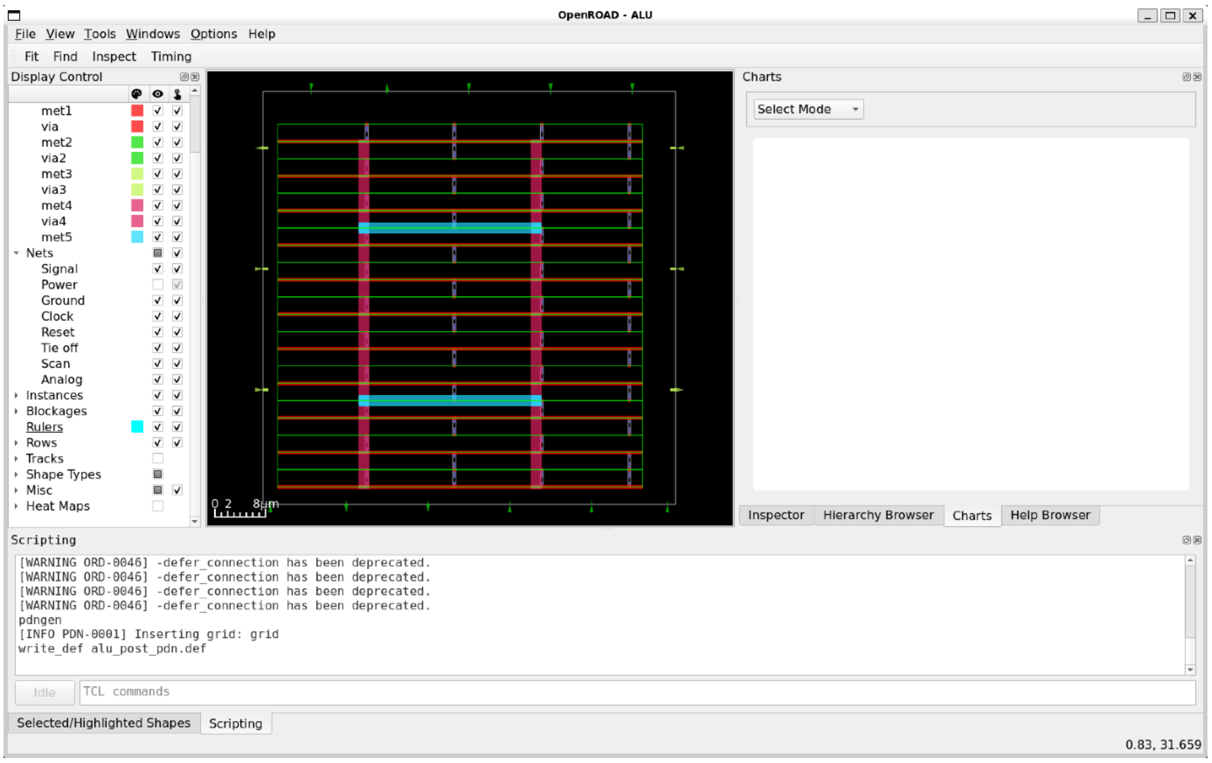
The next image focuses solely on the power network (VDD), highlighting the power rails and their distribution. It provides an overview of how the power grid is structured to ensure stable power delivery to the components.



```
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
pdngen
[INFO PDN-0001] Inserting grid: grid
write_def alu_post_pdn.def
```

3. Ground Network Only

The final image represents the ground network (VSS). It illustrates the ground connections and their layout across the design, ensuring proper ground plane continuity for reliable operation.



```
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
[WARNING ORD-0046] -defer_connection has been deprecated.
pdngen
[INFO PDN-0001] Inserting grid: grid
write_def alu_post_pdn.def
```

Conclusion

The Physical Design (PD) flow for the ALU project was successfully executed, with a focus on integrating the power distribution network (PDN) to ensure proper power and ground connectivity. The PDN insertion was crucial in optimizing the design for reliability and performance. The floorplan and PDN steps were completed, laying the foundation for the next stages of detailed placement and routing, ensuring the design is ready for further optimization and verification.

5.4.3 Placement

Placement is a critical step in the physical design flow that involves determining the precise location of all the cells (standard cells, macros, etc.) on the chip. The goal is to ensure the cells are placed in an optimal manner, balancing factors like timing, routability, power delivery, and overall chip density.

The key objectives of the placement step include:

- Ensuring optimal placement of cells to meet timing and power requirements.
- Maximizing routing efficiency and minimizing congestion.
- Maintaining chip area and ensuring proper utilization of the available space.
- Minimizing wire lengths for efficient signal propagation.

Input Files Used in Placement

Before executing the placement flow, several input files are required to define placement constraints, design libraries, and the floorplan configuration.

Technology & Standard Cell Library Files:

- sky130_fd_sc_hd__tt_025C_1v80.lib → Standard cell timing library.
- sky130_fd_sc_hd_merged.lef → Defines metal layers, DRC rules, and cell placements.
- sky130_fd_sc_hd.tlef → Defines track definitions for power routing.
- sky130hd.tracks → Specifies the track definitions for standard cell placement and routing.
- sky130hd.vars → Stores technology-specific variables (e.g., metal layers, densities).
- sky130hd.pdn.tcl → Defines power distribution network configuration.
- sky130hd.rc → Resistance and capacitance file for wire modeling.

Design-Specific Files:

- ALU_synth.v → Synthesized Verilog netlist from logic synthesis.
- constraints.sdc → Defines clock constraints and timing requirements.

Helper Scripts:

- helpers.tcl → Contains reusable functions for physical design tasks.
- flow_helpers.tcl → Assists in managing the physical design flow and constraints.

Execution of Placement Flow (ALU_PD.tcl)

The ALU_PD.tcl script automates the process of placement by calling the necessary flow scripts. Below is the script:

```
source "helpers.tcl"
source "flow_helpers.tcl"
source "sky130hd.vars"

# Design-specific variables
set synth_verilog "ALU_synth.v"
set design "ALU"
set top_module "ALU"
set sdc_file "constraints.sdc"

# Die and Core Area (Proportionally Adjusted for Density Control)
set die_area {0 0 65 65}       ;# Die area adjusted to ~4202.9 µm²
set core_area {2.3 2.72 59.96 59.89}  ;# Core area adjusted to ~3315.1 µm²

# Load and run the flow
source -echo "Flow_Placement.tcl"
```

Content of (Flow_Placement.tcl)

*The **Flow_Placement.tcl** script is responsible for Placement operaton. Below is the script:*

```
# Assumes flow_helpers.tcl has been read.
read_libraries
read_verilog $synth_verilog
link_design $top_module
read_sdc $sdc_file

set_thread_count [exec getconf _NPROCESSORS_ONLN]
# Temporarily disable sta's threading due to random failures
sta::set_thread_count 1

utl::metric "IFP::ord_version" [ord::openroad_git_describe]
# Note that sta::network_instance_count is not valid after tapcells are added.
utl::metric "IFP::instance_count" [sta::network_instance_count]

initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area

source $tracks_file

# Remove buffers inserted by synthesis
```

```
remove_buffers

################################################################
# IO Placement (random)
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

################################################################
# Macro Placement
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}

################################################################
# Tapcell insertion
eval tapcell $tapcell_args

################################################################
# Power distribution network insertion
source $pdn_cfg
pdngen

################################################################
# Global placement
foreach layer_adjustment $global_routing_layer_adjustments {
  lassign $layer_adjustment layer adjustment
  set_global_routing_layer_adjustment $layer $adjustment
}
set_routing_layers -signal $global_routing_layers \
  -clock $global_routing_clock_layers
set_macro_extension 2

global_placement -routability_driven -density $global_place_density \
  -pad_left $global_place_pad -pad_right $global_place_pad

# IO Placement
place_pins -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

# Checkpoint
set global_place_db [make_result_file ${design}_${platform}_global_place.db]
write_db $global_place_db

################################################################
# Repair max slew/cap/fanout violations and normalize slews
source $layer_rc_file
set_wire_rc -signal -layer $wire_rc_layer
set_wire_rc -clock  -layer $wire_rc_layer_clk
set_dont_use $dont_use

estimate_parasitics -placement
```

```
repair_design -slew_margin $slew_margin -cap_margin $cap_margin

repair_tie_fanout -separation $tie_separation $tielo_port
repair_tie_fanout -separation $tie_separation $tiehi_port

set_placement_padding -global -left $detail_place_pad -right $detail_place_pad
detailed_placement

# Post resize timing report (ideal clocks)
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
# Check slew repair
report_check_types -max_slew -max_capacitance -max_fanout -violators

utl::metric "RSZ::repair_design_buffer_count" [rsz::repair_design_buffer_count]
utl::metric "RSZ::max_slew_slack" [expr [sta::max_slew_check_slack_limit] * 100]
utl::metric "RSZ::max_fanout_slack" [expr [sta::max_fanout_check_slack_limit] * 100]
utl::metric "RSZ::max_capacitance_slack" [expr [sta::max_capacitance_check_slack_limit] * 100]

write_verilog post_detailed_placement.v
write_def post_detailed_placement.def
```

Breakdown of Placement-Specific Commands

```
foreach layer_adjustment $global_routing_layer_adjustments {
  lassign $layer_adjustment layer adjustment
  set_global_routing_layer_adjustment $layer $adjustment
}
```

- ✓ Iterates through the global routing layer adjustments and applies them to the routing layers.

```
set_routing_layers -signal $global_routing_layers \
  -clock $global_routing_clock_layers
```

- ✓ Defines the signal and clock routing layers.

```
set_macro_extension 2
```

- ✓ Specifies macro extension for placement.

```
global_placement -routability_driven -density $global_place_density \
  -pad_left $global_place_pad -pad_right $global_place_pad
```

- ✓ Performs the global placement driven by routability and density.

```
place_pins -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer
```

- ✓ Places the IO pins on the specified horizontal and vertical layers.

```
set global_place_db [make_result_file ${design}_${platform}_global_place.db]
```

write_db $global_place_db

- ✓ Saves the global placement result in a database file.

source $layer_rc_file
set_wire_rc -signal -layer $wire_rc_layer
set_wire_rc -clock -layer $wire_rc_layer_clk

- ✓ Sources the RC layer file and sets the signal and clock RC layers.

set_dont_use $dont_use

- ✓ Specifies the layers to avoid using in the design.

estimate_parasitics -placement

- ✓ Estimates parasitic capacitance and resistance based on placement.

repair_design -slew_margin $slew_margin -cap_margin $cap_margin

- ✓ Repairs violations related to slew rate, capacitance, and fanout.

repair_tie_fanout -separation $tie_separation $tielo_port
repair_tie_fanout -separation $tie_separation $tiehi_port

- ✓ Repairs tie fanout violations for specified ports.

set_placement_padding -global -left $detail_place_pad -right $detail_place_pad

- ✓ Sets padding for the placement on the left and right sides.

detailed_placement

- ✓ Executes the detailed placement step.

report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3

- ✓ Generates reports for the worst-case slack, total negative slack (TNS), and other timing metrics.

report_check_types -max_slew -max_capacitance -max_fanout -violators

- ✓ Reports any violations of slew, capacitance, or fanout limits.

utl::metric "RSZ::repair_design_buffer_count" [rsz::repair_design_buffer_count]
*utl::metric "RSZ::max_slew_slack" [expr [sta::max_slew_check_slack_limit] * 100]*
*utl::metric "RSZ::max_fanout_slack" [expr [sta::max_fanout_check_slack_limit] * 100]*
*utl::metric "RSZ::max_capacitance_slack" [expr [sta::max_capacitance_check_slack_limit] * 100]*

- ✓ Records metrics for repair counts and slack limits for slew, fanout, and capacitance.

write_verilog post_detailed_placement.v
write_def post_detailed_placement.def

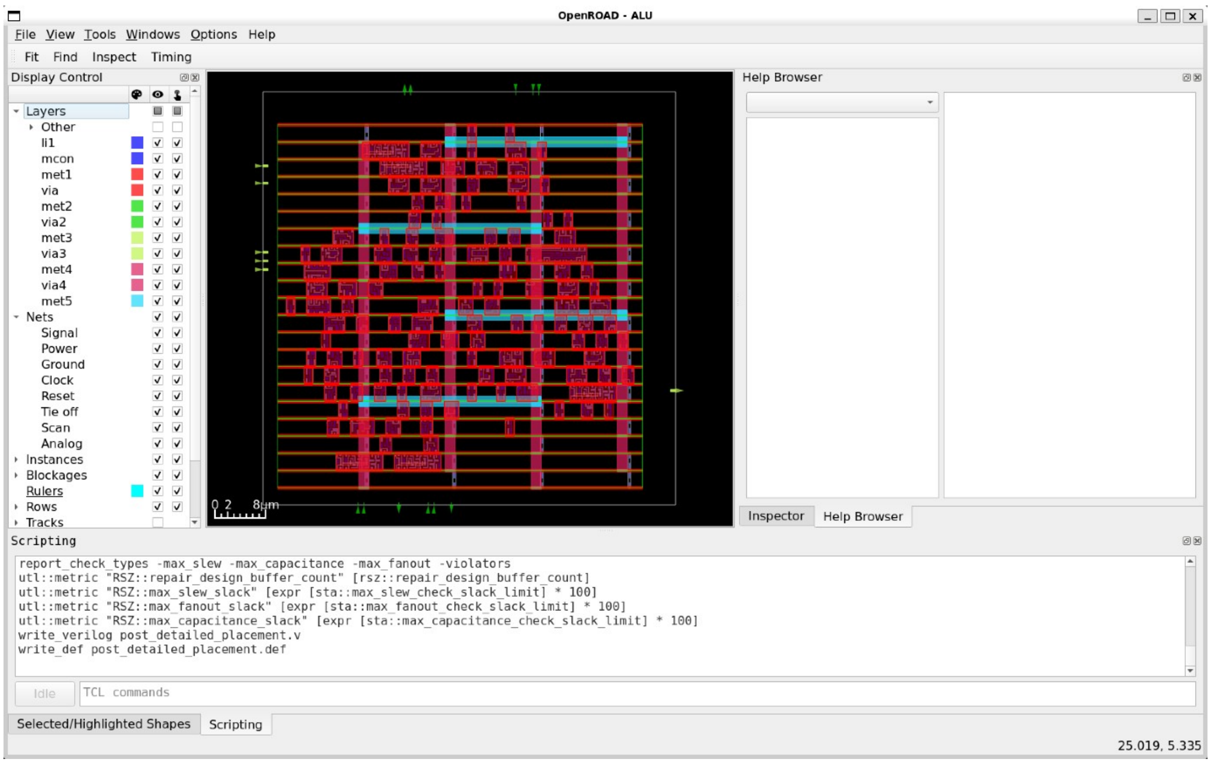
✓ Exports the design to Verilog and DEF files after detailed placement.

Output Files Generated

After executing the placement flow, the following output files are generated:

- **post_detailed_placement.def** → Contains the detailed placement information of the design.
- **post_detailed_placement.v** → Verilog netlist after placement.
- **ALU_DetailedPlacement.log** → Log file capturing the placement process, warnings, and errors.
- **ALU_sky130hd_global_place-tcl.db** → Global placement database.

Placement Results



```
report_check_types -max_slew -max_capacitance -max_fanout -violators
utl::metric "RSZ::repair_design_buffer_count" [rsz::repair_design_buffer_count]
utl::metric "RSZ::max_slew_slack" [expr [sta::max_slew_check_slack_limit] * 100]
utl::metric "RSZ::max_fanout_slack" [expr [sta::max_fanout_check_slack_limit] * 100]
utl::metric "RSZ::max_capacitance_slack" [expr [sta::max_capacitance_check_slack_limit] * 100]
write_verilog post_detailed_placement.v
write_def post_detailed_placement.def
```

Key Points from results:

1. **Core and Die Information:**

- Core Bounding Box: (2.300, 2.720) to (59.800, 59.840) µm.
- Die Bounding Box: (0.000, 0.000) to (65.000, 65.000) µm.

- Core Area: 3284.400 µm².
- Placement Utilization: 63.009%.

2. **Instance and Net Statistics:**
   - Total Instances: 169.
   - Placeable Instances: 123.
   - Fixed Instances: 46.
   - Nets: 135, Pins: 475.
3. **Placement and HPWL Optimization:**
   - Initial HPWL: 1,817,700 (Iter 1).
   - Final HPWL after multiple iterations: 2,054,528.
   - Nesterov optimization was performed with iterations improving congestion overflow.
4. **Routing Congestion and Density:**
   - Target Density: Initially 0.920, adjusted to 0.941.
   - Routing Congestion (FinalRC): 1.0497.
   - White Space Area: 3226.845 µm².
   - Filler Cell Area: 935.497 µm².
5. **Routability and Overflow Improvements:**
   - Routing overflow reduced from 0.799 to 0.318.
   - Number of overflow tiles reduced from 2.
   - Iterative refinement performed to reduce congestion and improve routability.

Conclusion

- The **global placement process** was successfully executed with **optimized placement density** (from 0.920 to 0.941).
- The **HPWL (Half-Perimeter Wire Length) was minimized**, ensuring better wire routing efficiency.
- **Routing congestion and overflow improved significantly** with iterative adjustments.
- **Placement utilization is at 63%,** indicating a well-balanced distribution of instances.
- **Further improvements in congestion reduction and density adjustments may be needed** for better final routing results.

5.4.4 Clock Tree Synthesis (CTS)

Clock Tree Synthesis (CTS) is a crucial step in the physical design flow, responsible for distributing the clock signal efficiently across the design while minimizing clock skew and optimizing timing. The goal of CTS is to ensure that all sequential elements (flip-flops and registers) receive the clock signal with minimal delay variation.

Key Objectives of CTS:

- Minimize clock skew and ensure synchronized signal propagation.
- Balance clock latency across the design.
- Reduce power consumption by optimizing clock buffers and inverters.
- Improve overall timing by optimizing the clock distribution network.

Input Files Used in CTS

Before executing the CTS flow, several input files are required, defining timing constraints, design specifications, and technology parameters.

Technology & Standard Cell Library Files:

- sky130_fd_sc_hd__tt_025C_1v80.lib → Standard cell timing library.
- sky130_fd_sc_hd_merged.lef → Defines metal layers, DRC rules, and cell placements.
- sky130_fd_sc_hd.tlef → Defines track definitions for power routing.
- sky130hd.tracks → Specifies the track definitions for standard cell placement and routing.
- sky130hd.vars → Stores technology-specific variables (e.g., metal layers, densities).
- sky130hd.pdn.tcl → Defines power distribution network configuration.
- sky130hd.rc → Resistance and capacitance file for wire modeling.

Design-Specific Files:

- ALU_synth.v → Synthesized Verilog netlist from logic synthesis.
- constraints.sdc → Defines clock constraints and timing requirements.

Helper Scripts:

- helpers.tcl → Contains reusable functions for physical design tasks.
- flow_helpers.tcl → Assists in managing the physical design flow and constraints.

Execution of CTS Flow (CTS_PD.tcl)

*The **CTS_PD.tcl** script automates the process of clock tree synthesis by invoking necessary flow scripts. Below is the script:*

```
source "helpers.tcl"
source "flow_helpers.tcl"
source "sky130hd.vars"

# Design-specific variables
set synth_verilog "ALU_synth.v"
set design "ALU"
set top_module "ALU"  ;# Update this with your top module name
set sdc_file "constraints.sdc"

# Die and Core Area (Proportionally Adjusted for Density Control)
set die_area {0 0 65 65}       ;# Die area adjusted to ~4202.9 µm²
set core_area {2.3 2.72 59.96 59.89}  ;# Core area adjusted to ~3315.1 µm²

# Load and run the flow
source -echo "Flow_CTS.tcl"
```

Content of Flow_CTS.tcl

The Flow_CTS.tcl script executes the clock tree synthesis operations. Below is the script:

```
# Assumes flow_helpers.tcl has been read.
read_libraries
read_verilog $synth_verilog
link_design $top_module
read_sdc $sdc_file

set_thread_count [exec getconf _NPROCESSORS_ONLN]
# Temporarily disable sta's threading due to random failures
sta::set_thread_count 1

utl::metric "IFP::ord_version" [ord::openroad_git_describe]
# Note that sta::network_instance_count is not valid after tapcells are added.
utl::metric "IFP::instance_count" [sta::network_instance_count]

initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area

source $tracks_file
```

```
# remove buffers inserted by synthesis
remove_buffers

################################################################
# IO Placement (random)
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

################################################################
# Macro Placement
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}

################################################################
# Tapcell insertion
eval tapcell $tapcell_args

################################################################
# Power distribution network insertion
source $pdn_cfg
pdngen

################################################################
# Global placement

foreach layer_adjustment $global_routing_layer_adjustments {
  lassign $layer_adjustment layer adjustment
  set_global_routing_layer_adjustment $layer $adjustment
}
set_routing_layers -signal $global_routing_layers \
  -clock $global_routing_clock_layers
set_macro_extension 2

global_placement -routability_driven -density $global_place_density \
  -pad_left $global_place_pad -pad_right $global_place_pad

# IO Placement
place_pins -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

# checkpoint
set global_place_db [make_result_file ${design}_${platform}_global_place.db]
write_db $global_place_db

################################################################
# Repair max slew/cap/fanout violations and normalize slews
source $layer_rc_file
set_wire_rc -signal -layer $wire_rc_layer
set_wire_rc -clock  -layer $wire_rc_layer_clk
set_dont_use $dont_use
```

```
estimate_parasitics -placement

repair_design -slew_margin $slew_margin -cap_margin $cap_margin

repair_tie_fanout -separation $tie_separation $tielo_port
repair_tie_fanout -separation $tie_separation $tiehi_port

set_placement_padding -global -left $detail_place_pad -right $detail_place_pad
detailed_placement

# post resize timing report (ideal clocks)
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
# Check slew repair
report_check_types -max_slew -max_capacitance -max_fanout -violators

utl::metric "RSZ::repair_design_buffer_count" [rsz::repair_design_buffer_count]
utl::metric "RSZ::max_slew_slack" [expr [sta::max_slew_check_slack_limit] * 100]
utl::metric "RSZ::max_fanout_slack" [expr [sta::max_fanout_check_slack_limit] * 100]
utl::metric "RSZ::max_capacitance_slack" [expr [sta::max_capacitance_check_slack_limit] * 100]

################################################################
# Clock Tree Synthesis

# Clone clock tree inverters next to register loads
# so cts does not try to buffer the inverted clocks.
repair_clock_inverters

clock_tree_synthesis -root_buf $cts_buffer -buf_list $cts_buffer \
  -sink_clustering_enable \
  -sink_clustering_max_diameter $cts_cluster_diameter

# CTS leaves a long wire from the pad to the clock tree root.
repair_clock_nets

# place clock buffers
detailed_placement

# checkpoint
set cts_db [make_result_file ${design}_${platform}_cts.db]
write_db $cts_db

################################################################
# Setup/hold timing repair

set_propagated_clock [all_clocks]

# Global routing is fast enough for the flow regressions.
# It is NOT FAST ENOUGH FOR PRODUCTION USE.
set repair_timing_use_grt_parasitics 0
```

```
if { $repair_timing_use_grt_parasitics } {
  # Global route for parasitics - no guide file requied
  global_route -congestion_iterations 100
  estimate_parasitics -global_routing
} else {
  estimate_parasitics -placement
}

repair_timing -skip_gate_cloning

# Post timing repair.
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
report_check_types -max_slew -max_capacitance -max_fanout -violators -digits 3

utl::metric "RSZ::worst_slack_min" [sta::worst_slack -min]
utl::metric "RSZ::worst_slack_max" [sta::worst_slack -max]
utl::metric "RSZ::tns_max" [sta::total_negative_slack -max]
utl::metric "RSZ::hold_buffer_count" [rsz::hold_buffer_count]

################################################################
# Detailed Placement

detailed_placement

# Capture utilization before fillers make it 100%
utl::metric "DPL::utilization" [format %.1f [expr [rsz::utilization] * 100]]
utl::metric "DPL::design_area" [sta::format_area [rsz::design_area] 0]

# checkpoint
set dpl_db [make_result_file ${design}_${platform}_dpl.db]
write_db $dpl_db

set verilog_file [make_result_file ${design}_${platform}.v]
write_verilog $verilog_file
```

Breakdown of CTS-Specific Commands

```
# Clone clock tree inverters next to register loads
# so CTS does not try to buffer the inverted clocks.
repair_clock_inverters
```

- ✓ Ensures **clock inverters** are placed **close to register loads**, preventing **CTS from inserting unnecessary buffers**.
- ✓ Helps **maintain clock signal integrity** and avoid extra delays.

```
clock_tree_synthesis -root_buf $cts_buffer -buf_list $cts_buffer \
  -sink_clustering_enable \
  -sink_clustering_max_diameter $cts_cluster_diameter
```

- ✓ **Builds the clock tree** by inserting **clock buffers** and ensuring a balanced clock network.
- ✓ -root_buf $cts_buffer → Specifies the **root buffer** used at the **clock tree source**.
- ✓ -buf_list $cts_buffer → List of **buffer cells** available for CTS.
- ✓ -sink_clustering_enable → **Groups clock sinks** (flip-flops) to create **balanced clusters**.
- ✓ -sink_clustering_max_diameter $cts_cluster_diameter → **Limits the maximum diameter** for clock sink clusters, preventing excessive clock skew.

```
# CTS leaves a long wire from the pad to the clock tree root.
repair_clock_nets
```

- ✓ After CTS, **long wires** might still exist between the **clock source (pad)** and **clock buffers**.
- ✓ This command **inserts buffers** to **shorten long wires**, improving **clock signal propagation**.

```
# Place clock buffers
detailed_placement
```

- ✓ Ensures **clock buffers** inserted during CTS are placed in **optimal locations**.
- ✓ Avoids **congestion** and maintains proper **cell alignment**.

```
# Save CTS checkpoint database for later stages
set cts_db [make_result_file ${design}_${platform}_cts.db]
write_db $cts_db
```

- ✓ Saves the **CTS results** into a database file (.db format) for **debugging and analysis**.
- ✓ This allows rollback if further modifications are needed.

```
set_propagated_clock [all_clocks]
```

- ✓ **Marks the clock as "propagated"**, meaning that **delays due to buffers** are now considered in **timing analysis**.

```
set repair_timing_use_grt_parasitics 0
if { $repair_timing_use_grt_parasitics } {
  # Global route for parasitics - no guide file required
  global_route -congestion_iterations 100
  estimate_parasitics -global_routing
} else {
  estimate_parasitics -placement
}
```

- ✓ Parasitics (wire delays) must be estimated **before** timing analysis.
- ✓ If repair_timing_use_grt_parasitics = 1, global routing is performed **before estimating parasitics**.
- ✓ Otherwise, parasitic values from **placement** are used for analysis.

```
repair_timing -skip_gate_cloning
```

- ✓ Tries to **fix setup and hold violations** by adjusting **buffer placement and wire delays**.
- ✓ -skip_gate_cloning → Prevents **unnecessary cloning** of gates.

```
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
```

report_check_types -max_slew -max_capacitance -max_fanout -violators -digits 3

Reports critical timing metrics after CTS:

- ✓ **Worst Slack (min/max)** → Measures the worst-case timing violation.
- ✓ **Total Negative Slack (TNS)** → Sum of all violated setup paths.
- ✓ **Slew, Capacitance, Fanout Violations** → Ensures signal integrity.

detailed_placement

- ✓ Ensures **all newly inserted buffers and cells** are placed correctly to avoid **overlaps** and **routing issues**.

```
utl::metric "DPL::utilization" [format %.1f [expr [rsz::utilization] * 100]]
utl::metric "DPL::design_area" [sta::format_area [rsz::design_area] 0]
```

- ✓ Measures **cell utilization (%)** before adding **filler cells** (which artificially make utilization 100%).

```
# Save Detailed Placement Checkpoint
set dpl_db [make_result_file ${design}_${platform}_dpl.db]
write_db $dpl_db
```

- ✓ Saves the **detailed placement database** after CTS for debugging.

```
# Generate Post-CTS Verilog Netlist
set verilog_file [make_result_file ${design}_${platform}.v]
write_verilog $verilog_file
```

- ✓ Exports the **new Verilog netlist** after CTS, which includes **clock buffers and optimized placement**.

Output Files After CTS

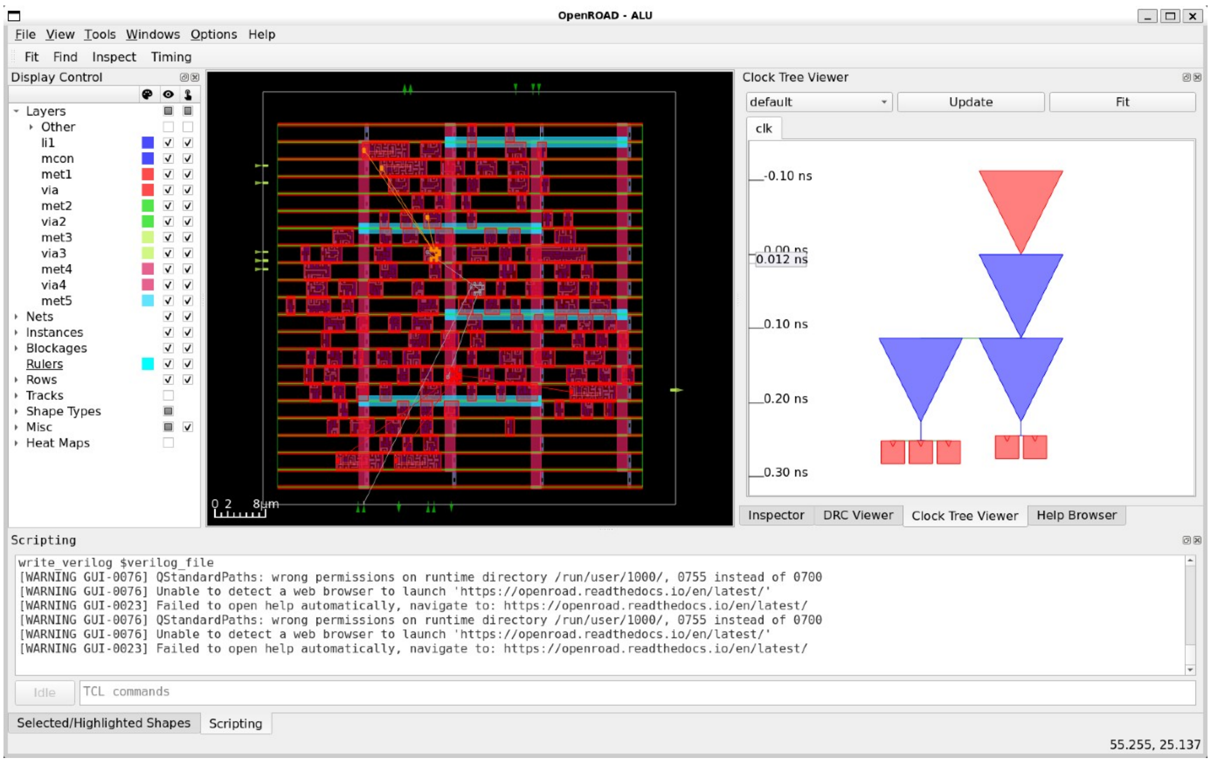
- ❖ **ALU_sky130hd-tcl.v** → Updated Verilog netlist after CTS.

- ❖ **ALU_sky130hd_cts-tcl.db** → CTS checkpoint database.

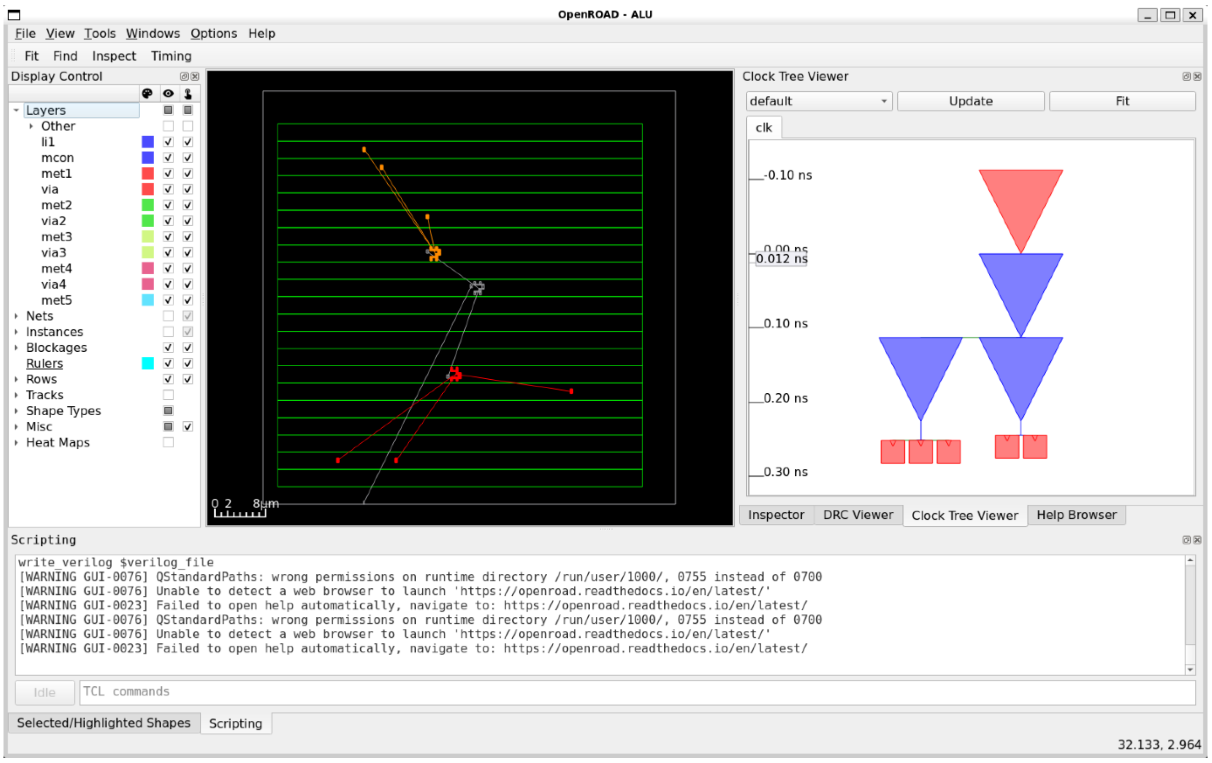
- ❖ **ALU_sky130hd_dpl-tcl.db** → Database file capturing detailed placement after CTS.

- ❖ **ALU_sky130hd_global_place-tcl.db** → Database file for global placement before CTS.

CTS Results



```
write_verilog $verilog_file
[WARNING GUI-0076] QStandardPaths: wrong permissions on runtime directory /run/user/1000/, 0755 instead of 0700
[WARNING GUI-0076] Unable to detect a web browser to launch 'https://openroad.readthedocs.io/en/latest/'
[WARNING GUI-0023] Failed to open help automatically, navigate to: https://openroad.readthedocs.io/en/latest/
[WARNING GUI-0076] QStandardPaths: wrong permissions on runtime directory /run/user/1000/, 0755 instead of 0700
[WARNING GUI-0076] Unable to detect a web browser to launch 'https://openroad.readthedocs.io/en/latest/'
[WARNING GUI-0023] Failed to open help automatically, navigate to: https://openroad.readthedocs.io/en/latest/
```



```
write_verilog $verilog_file
[WARNING GUI-0076] QStandardPaths: wrong permissions on runtime directory /run/user/1000/, 0755 instead of 0700
[WARNING GUI-0076] Unable to detect a web browser to launch 'https://openroad.readthedocs.io/en/latest/'
[WARNING GUI-0023] Failed to open help automatically, navigate to: https://openroad.readthedocs.io/en/latest/
[WARNING GUI-0076] QStandardPaths: wrong permissions on runtime directory /run/user/1000/, 0755 instead of 0700
[WARNING GUI-0076] Unable to detect a web browser to launch 'https://openroad.readthedocs.io/en/latest/'
[WARNING GUI-0023] Failed to open help automatically, navigate to: https://openroad.readthedocs.io/en/latest/
```

Key points from the results:

1. **Clock Tree Synthesis (CTS) Setup**
    - Root buffer: sky130_fd_sc_hd__clkbuf_4
    - Sink buffer: sky130_fd_sc_hd__clkbuf_4
    - 1 clock net (clk) found with 5 sinks
    - H-Tree topology generated for clk with max cluster diameter of 100.0 µm
    - 3 clock buffers created
    - 2 minimum and 2 maximum buffers in the clock path
    - Fanout distribution: 2:1, 3:1
    - Path depth: 2
2. **Clock Net Information**
    - Sinks: 6
    - No leaf buffers
    - Average sink wire length: 82.94 µm
    - 0 placement blockages and no placed hard macros treated as blockages
3. **Wire Segment Unit and Sink Regions**

- Wire segment unit: 13 µm
- Sink region (normalized):
  - Width: 2.7059
  - Height: 3.6000

4. **Timing Repair**
   - **Setup and Hold Timing**:
     - No setup or hold violations found
     - Estimated parasitics using placement-based calculations
     - Worst slack:
       - Min: 0.966
       - Max: 6.018
     - Total Negative Slack (TNS): 0.000
5. **Placement**
   - **Placement Analysis**:
     - Total displacement: 64.6 µm
     - Average displacement: 0.4 µm
     - Max displacement: 9.1 µm
     - HPWL (original): 2655.6 µm
     - HPWL (legalized): 2715.2 µm
     - 2% delta in HPWL after legalization
   - **Detailed Placement**:
     - Total displacement: 0.0 µm
     - Average displacement: 0.0 µm
     - Max displacement: 0.0 µm
     - HPWL remained the same after legalization

Conclusion

- The Clock Tree Synthesis (CTS) process was successfully executed with a balanced clock tree structure, ensuring efficient clock distribution.

- ➢ The H-Tree topology efficiently clustered sinks, with a maximum cluster diameter of 100.0 µm, ensuring minimal clock skew.
- ➢ Clock buffering was optimized, using sky130_fd_sc_hd__clkbuf_4 buffers for both root and sink, providing stable clock signals.
- ➢ Timing analysis showed no setup or hold violations, ensuring robust clock performance with minimal negative slack.
- ➢ The process resulted in a well-optimized clock tree, with a minimal increase in HPWL (2%) after legalization.

This concludes the CTS process with a reliable and efficient clock distribution network, ready for further stages in the design flow.

5.4.5 Routing

Routing is the final step in the physical design flow, responsible for creating the actual metal interconnections between the standard cells while adhering to design constraints and design rule checks (DRC). It ensures that all signal, power, and clock nets are properly connected while minimizing congestion and improving performance.

Key Objectives of Routing:

- Establish optimal wire connections between components while avoiding congestion.
- Ensure adherence to process technology design rules (DRC compliance).
- Minimize wire delay, crosstalk, and signal integrity issues.
- Optimize power and ground routing to ensure robust power delivery.
- Balance trade-offs between timing, power, and area constraints.

Input Files Used in Routing

Before executing the routing flow, several input files are required that define design constraints, routing resources, and process technology parameters.

Technology & Standard Cell Library Files:

- ❖ **sky130_fd_sc_hd__tt_025C_1v80.lib** → Standard cell timing library.
- ❖ **sky130_fd_sc_hd_merged.lef** → Layout exchange format file defining routing layers and DRC rules.

- **sky130_fd_sc_hd.tlef** → Technology LEF file specifying track definitions for routing.
- **sky130hd.tracks** → Specifies track definitions used for signal routing.
- **sky130hd.vars** → Stores technology-specific variables such as metal layer stack and routing parameters.
- **sky130hd.pdn.tcl** → Defines the power distribution network.
- **sky130hd.rc** → Resistance and capacitance file for wire modeling and signal integrity analysis.
- **sky130hd.rcx_rules** → Contains extraction rules for resistance and capacitance estimation.

Design-Specific Files:

- ALU_synth.v → Synthesized Verilog netlist from logic synthesis.
- constraints.sdc → Defines clock constraints and timing requirements.

Helper Scripts:

- helpers.tcl → Contains reusable functions for physical design tasks.
- flow_helpers.tcl → Assists in managing the physical design flow and constraints.

Execution of Routing Flow (Routing_PD.tcl)

The Routing_PD.tcl script automates the routing process using the required input files. Below is the script:

```
source "helpers.tcl"
source "flow_helpers.tcl"
source "sky130hd.vars"

# Design-specific variables
set synth_verilog "ALU_synth.v"
set design "ALU"
set top_module "ALU"  ;# Update this with your top module name
set sdc_file "constraints.sdc"

# Die and Core Area (Proportionally Adjusted for Density Control)
set die_area {0 0 65 65}       ;# Die area adjusted to ~4202.9 µm²
set core_area {2.3 2.72 59.96 59.89}  ;# Core area adjusted to ~3315.1 µm²

# Load and run the flow
source -echo "Flow_Routing.tcl"
```

Content of Flow_Routing.tcl

```
# Assumes flow_helpers.tcl has been read.
read_libraries
read_verilog $synth_verilog
link_design $top_module
read_sdc $sdc_file

set_thread_count [exec getconf _NPROCESSORS_ONLN]
# Temporarily disable sta's threading due to random failures
sta::set_thread_count 1

utl::metric "IFP::ord_version" [ord::openroad_git_describe]
# Note that sta::network_instance_count is not valid after tapcells are added.
utl::metric "IFP::instance_count" [sta::network_instance_count]

initialize_floorplan -site $site \
  -die_area $die_area \
  -core_area $core_area

source $tracks_file

# remove buffers inserted by synthesis
remove_buffers

################################################################
# IO Placement (random)
place_pins -random -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

################################################################
# Macro Placement
if { [have_macros] } {
  global_placement -density $global_place_density
  macro_placement -halo $macro_place_halo -channel $macro_place_channel
}

################################################################
# Tapcell insertion
eval tapcell $tapcell_args

################################################################
# Power distribution network insertion
source $pdn_cfg
pdngen

################################################################
# Global placement

foreach layer_adjustment $global_routing_layer_adjustments {
  lassign $layer_adjustment layer adjustment
  set_global_routing_layer_adjustment $layer $adjustment
}
```

```
set_routing_layers -signal $global_routing_layers \
  -clock $global_routing_clock_layers
set_macro_extension 2

global_placement -routability_driven -density $global_place_density \
  -pad_left $global_place_pad -pad_right $global_place_pad

# IO Placement
place_pins -hor_layers $io_placer_hor_layer -ver_layers $io_placer_ver_layer

# checkpoint
set global_place_db [make_result_file ${design}_${platform}_global_place.db]
write_db $global_place_db

################################################################
# Repair max slew/cap/fanout violations and normalize slews
source $layer_rc_file
set_wire_rc -signal -layer $wire_rc_layer
set_wire_rc -clock  -layer $wire_rc_layer_clk
set_dont_use $dont_use

estimate_parasitics -placement

repair_design -slew_margin $slew_margin -cap_margin $cap_margin

repair_tie_fanout -separation $tie_separation $tielo_port
repair_tie_fanout -separation $tie_separation $tiehi_port

set_placement_padding -global -left $detail_place_pad -right $detail_place_pad
detailed_placement

# post resize timing report (ideal clocks)
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
# Check slew repair
report_check_types -max_slew -max_capacitance -max_fanout -violators

utl::metric "RSZ::repair_design_buffer_count" [rsz::repair_design_buffer_count]
utl::metric "RSZ::max_slew_slack" [expr [sta::max_slew_check_slack_limit] * 100]
utl::metric "RSZ::max_fanout_slack" [expr [sta::max_fanout_check_slack_limit] * 100]
utl::metric "RSZ::max_capacitance_slack" [expr [sta::max_capacitance_check_slack_limit] * 100]

################################################################
# Clock Tree Synthesis

# Clone clock tree inverters next to register loads
# so cts does not try to buffer the inverted clocks.
repair_clock_inverters

clock_tree_synthesis -root_buf $cts_buffer -buf_list $cts_buffer \
  -sink_clustering_enable \
  -sink_clustering_max_diameter $cts_cluster_diameter

# CTS leaves a long wire from the pad to the clock tree root.
repair_clock_nets

# place clock buffers
detailed_placement

# checkpoint
set cts_db [make_result_file ${design}_${platform}_cts.db]
write_db $cts_db

################################################################
# Setup/hold timing repair

set_propagated_clock [all_clocks]

# Global routing is fast enough for the flow regressions.
# It is NOT FAST ENOUGH FOR PRODUCTION USE.
set repair_timing_use_grt_parasitics 0
if { $repair_timing_use_grt_parasitics } {
  # Global route for parasitics - no guide file requied
  global_route -congestion_iterations 100
  estimate_parasitics -global_routing
} else {
  estimate_parasitics -placement
}

repair_timing -skip_gate_cloning

# Post timing repair.
report_worst_slack -min -digits 3
report_worst_slack -max -digits 3
report_tns -digits 3
report_check_types -max_slew -max_capacitance -max_fanout -violators -digits 3

utl::metric "RSZ::worst_slack_min" [sta::worst_slack -min]
utl::metric "RSZ::worst_slack_max" [sta::worst_slack -max]
utl::metric "RSZ::tns_max" [sta::total_negative_slack -max]
utl::metric "RSZ::hold_buffer_count" [rsz::hold_buffer_count]

################################################################
# Detailed Placement

detailed_placement

# Capture utilization before fillers make it 100%
utl::metric "DPL::utilization" [format %.1f [expr [rsz::utilization] * 100]]
utl::metric "DPL::design_area" [sta::format_area [rsz::design_area] 0]
```

```
# checkpoint
set dpl_db [make_result_file ${design}_${platform}_dpl.db]
write_db $dpl_db

set verilog_file [make_result_file ${design}_${platform}.v]
write_verilog $verilog_file

################################################################
# Global routing

pin_access -bottom_routing_layer $min_routing_layer \
        -top_routing_layer $max_routing_layer

set route_guide [make_result_file ${design}_${platform}.route_guide]
global_route -guide_file $route_guide \
  -congestion_iterations 100 -verbose

set verilog_file [make_result_file ${design}_${platform}.v]
write_verilog -remove_cells $filler_cells $verilog_file

################################################################
# Repair antennas post-GRT

utl::set_metrics_stage "grt__{}"
repair_antennas -iterations 5

check_antennas
utl::clear_metrics_stage
utl::metric "GRT::ANT::errors" [ant::antenna_violation_count]

################################################################
# Detailed routing

# Run pin access again after inserting diodes and moving cells
pin_access -bottom_routing_layer $min_routing_layer \
        -top_routing_layer $max_routing_layer

detailed_route -output_drc [make_result_file "${design}_${platform}_route_drc.rpt"] \
            -output_maze [make_result_file "${design}_${platform}_maze.log"] \
            -no_pin_access \
            -save_guide_updates \
            -bottom_routing_layer $min_routing_layer \
            -top_routing_layer $max_routing_layer \
            -verbose 0

write_guides [make_result_file "${design}_${platform}_output_guide.mod"]
set drv_count [detailed_route_num_drvs]
utl::metric "DRT::drv" $drv_count

set routed_db [make_result_file ${design}_${platform}_route.db]
write_db $routed_db
```

```
set routed_def [make_result_file ${design}_${platform}_route.def]
write_def $routed_def

################################################################
# Repair antennas post-DRT

set repair_antennas_iters 0
utl::set_metrics_stage "drt__repair_antennas__pre_repair__{}"
while {[check_antennas] && $repair_antennas_iters < 5} {
  utl::set_metrics_stage "drt__repair_antennas__iter_${repair_antennas_iters}__{}"

  repair_antennas

  detailed_route -output_drc [make_result_file "${design}_${platform}_ant_fix_drc.rpt"] \
                -output_maze [make_result_file "${design}_${platform}_ant_fix_maze.log"] \
                -save_guide_updates \
                -bottom_routing_layer $min_routing_layer \
                -top_routing_layer $max_routing_layer \
                -verbose 0

  incr repair_antennas_iters
}

utl::set_metrics_stage "drt__{}"
check_antennas

utl::clear_metrics_stage
utl::metric "DRT::ANT::errors" [ant::antenna_violation_count]

if {![design_is_routed]} {
  error "Design has unrouted nets."
}

set repair_antennas_db [make_result_file ${design}_${platform}_repaired_route.odb]
write_db $repair_antennas_db

################################################################
# Filler placement

filler_placement $filler_cells
check_placement -verbose

# checkpoint
set fill_db [make_result_file ${design}_${platform}_fill.db]
write_db $fill_db

################################################################
# Extraction

if { $rcx_rules_file != "" } {
  define_process_corner -ext_model_index 0 X
  extract_parasitics -ext_model_file $rcx_rules_file

  set spef_file [make_result_file ${design}_${platform}.spef]
  write_spef $spef_file

  read_spef $spef_file
} else {
  # Use global routing based parasitics inlieu of rc extraction
  estimate_parasitics -global_routing
}
```

Breakdown of Routing-Specific Commands:

```
pin_access -bottom_routing_layer $min_routing_layer \
        -top_routing_layer $max_routing_layer
```

- ✓ Sets the **routing layer constraints**, ensuring that pins are accessible between the minimum and maximum routing layers.

```
set route_guide [make_result_file ${design}_${platform}.route_guide]
global_route -guide_file $route_guide \
  -congestion_iterations 100 -verbose
```

- ✓ Runs **global routing**, generating a **routing guide** (.route_guide file).
- ✓ Performs **100 congestion iterations** to minimize routing congestion.
- ✓ Uses the **verbose** option to provide detailed routing output.

```
set verilog_file [make_result_file ${design}_${platform}.v]
write_verilog -remove_cells $filler_cells $verilog_file
```

- ✓ Generates a **post-global-routing Verilog netlist**.
- ✓ Removes **filler cells**, which are temporary cells used during placement.

```
utl::set_metrics_stage "grt__{}"
repair_antennas -iterations 5
```

- ✓ Runs antenna repair for 5 iterations to insert diodes and modify routing.

```
check_antennas
utl::clear_metrics_stage
utl::metric "GRT::ANT::errors" [ant::antenna_violation_count]
```

- ✓ Checks for **remaining antenna violations** and logs the number of violations.

```
pin_access -bottom_routing_layer $min_routing_layer \
        -top_routing_layer $max_routing_layer
```

- ✓ Runs **pin access analysis again** after diode insertion to ensure all pins are accessible.

```
detailed_route -output_drc [make_result_file "${design}_${platform}_route_drc.rpt"] \
        -output_maze [make_result_file "${design}_${platform}_maze.log"] \
        -no_pin_access \
        -save_guide_updates \
        -bottom_routing_layer $min_routing_layer \
        -top_routing_layer $max_routing_layer \
        -verbose 0
```

- ✓ **Generates a DRC report** (_route_drc.rpt) to check for violations.
- ✓ Saves **maze-based routing logs** (_maze.log) for debugging.
- ✓ **Disables pin access updates** (-no_pin_access) for efficiency.
- ✓ Saves routing **guide updates** to refine the routing process.

```
write_guides [make_result_file "${design}_${platform}_output_guide.mod"]
```

- ✓ Saves the **modified routing guide** for further refinements.

```
set drv_count [detailed_route_num_drvs]
utl::metric "DRT::drv" $drv_count
```

- ✓ Stores the number of **Design Rule Violations (DRV)** detected after detailed routing.

```
set routed_db [make_result_file ${design}_${platform}_route.db]
write_db $routed_db
```

- ✓ Saves the **routed design database**.

```
set routed_def [make_result_file ${design}_${platform}_route.def]
write_def $routed_def
```

- ✓ Exports the **final routed DEF file**.

```
set repair_antennas_iters 0
utl::set_metrics_stage "drt__repair_antennas__pre_repair__{}"
while {[check_antennas] && $repair_antennas_iters < 5} {
  utl::set_metrics_stage "drt__repair_antennas__iter_${repair_antennas_iters}__{}"

  repair_antennas
```

- ✓ If there are **remaining antenna violations**, it iterates up to **5 times** to repair them.

```
  detailed_route -output_drc [make_result_file "${design}_${platform}_ant_fix_drc.rpt"] \
          -output_maze [make_result_file "${design}_${platform}_ant_fix_maze.log"] \
          -save_guide_updates \
          -bottom_routing_layer $min_routing_layer \
          -top_routing_layer $max_routing_layer \
          -verbose 0
  incr repair_antennas_iters
}
```

- ✓ After each **antenna repair**, the script **reruns detailed routing** to ensure fixes are properly connected.

```
utl::set_metrics_stage "drt__{}"
check_antennas

utl::clear_metrics_stage
utl::metric "DRT::ANT::errors" [ant::antenna_violation_count]
```

- ✓ Final **antenna check** is performed to ensure all violations are resolved.

```
if {![design_is_routed]} {
  error "Design has unrouted nets."
}
```

- ✓ **Error handling**: If there are any **unrouted nets**, the script stops execution.

```
set repair_antennas_db [make_result_file ${design}_${platform}_repaired_route.odb]
write_db $repair_antennas_db
```

- ✓ Saves the **final routed design database after antenna repair**.

```
filler_placement $filler_cells
check_placement -verbose
```

- ✓ **Fills gaps** between standard cells with **filler cells**.
- ✓ Runs a **placement check** to ensure correctness.

```
set fill_db [make_result_file ${design}_${platform}_fill.db]
write_db $fill_db
```

- ✓ Saves the **final placement database after filler cell insertion**.

```
if { $rcx_rules_file != "" } {
  define_process_corner -ext_model_index 0 X
  extract_parasitics -ext_model_file $rcx_rules_file
```

- ✓ If **rcx_rules_file** (e.g., sky130hd.rcx_rules) is available, it is used for **parasitic extraction**.
- ✓ Defines a **process corner** to model process variations.

```
  set spef_file [make_result_file ${design}_${platform}.spef]
  write_spef $spef_file

  read_spef $spef_file
```

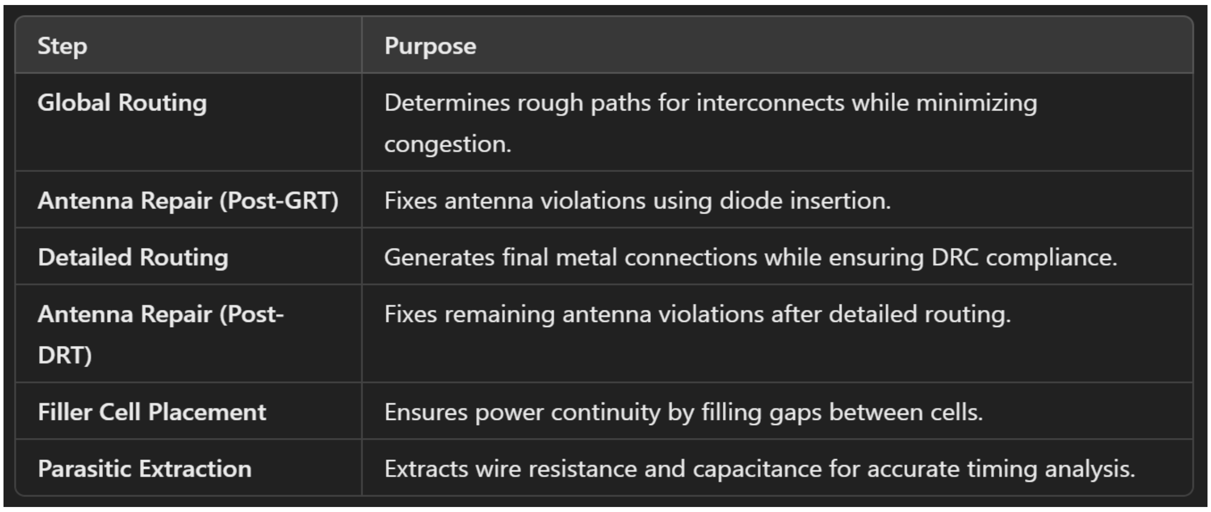
- ✓ Extracts and saves the **Standard Parasitic Exchange Format (SPEF) file**.
- ✓ The SPEF file contains **R, C values for wires** and is used in **Static Timing Analysis (STA)**.

```
} else {
  # Use global routing based parasitics inlieu of rc extraction
```

```
 estimate_parasitics -global_routing
}
```

- ✓ If no **rcx_rules_file** is provided, it **estimates parasitics based on global routing** (less accurate than full extraction).

Summary of Key Steps



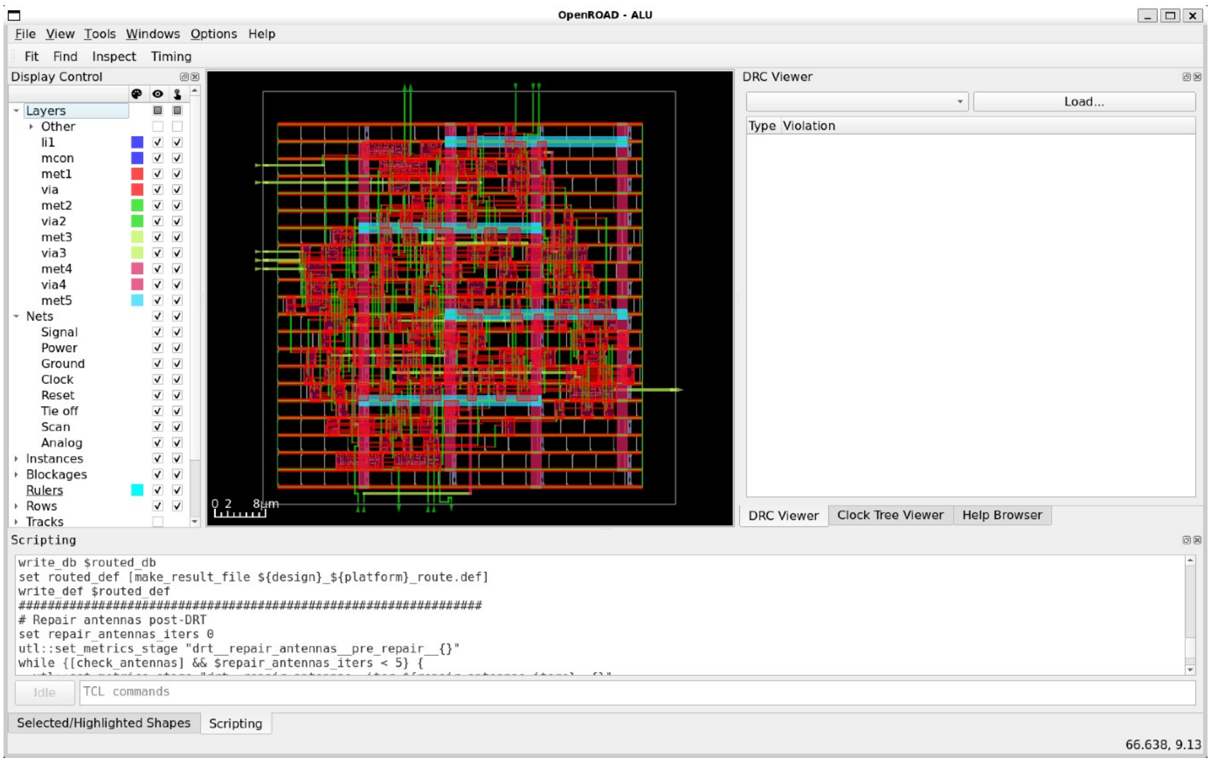
Step	Purpose
Global Routing	Determines rough paths for interconnects while minimizing congestion.
Antenna Repair (Post-GRT)	Fixes antenna violations using diode insertion.
Detailed Routing	Generates final metal connections while ensuring DRC compliance.
Antenna Repair (Post-DRT)	Fixes remaining antenna violations after detailed routing.
Filler Cell Placement	Ensures power continuity by filling gaps between cells.
Parasitic Extraction	Extracts wire resistance and capacitance for accurate timing analysis.

Output Files After Routing

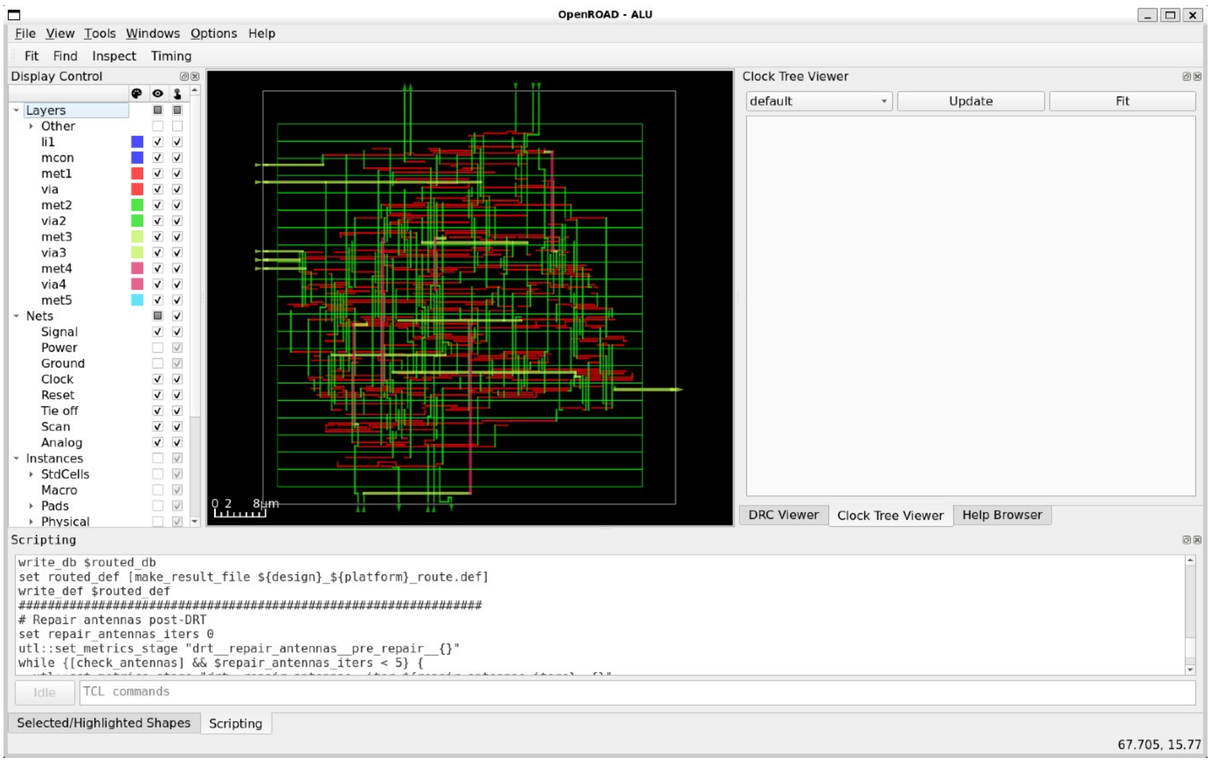
- ❖ **post_routing.def** → DEF file capturing final routing.
- ❖ **post_routing.v** → Updated Verilog netlist after routing.
- ❖ **ALU_Routing.log** → Log file for debugging.
- ❖ **ALU_sky130hd_routing.db** → Routing checkpoint database.
- ❖ **ALU_sky130hd-tcl.route_guide** → Routing guide information used during global routing.
- ❖ **ALU_sky130hd_route-tcl.db** → Global routing data, including routing paths.
- ❖ **ALU_sky130hd-tcl.spef** → Parasitic extraction file in SPEF format.
- ❖ **ALU_sky130hd_output_guide-tcl.mod** → Updated routing guide after global routing.
- ❖ **ALU_sky130hd_route-tcl.def** → DEF file containing final routing and placement information.
- ❖ **ALU_sky130hd_fill-tcl.db** → Filler cells placement data.
- ❖ **ALU_sky130hd_repaired_route-tcl.odb** → ODB++ file with final routed design and repairs.

❖ **ALU_sky130hd_route_drc-tcl.rpt** → DRC report identifying any design rule violations after routing.

Routing Results



```
write_db $routed_db
set routed_def [make_result_file ${design}_${platform}_route.def]
write_def $routed_def
################################################################
# Repair antennas post-DRT
set repair_antennas_iters 0
utl::set_metrics_stage "drt__repair_antennas__pre_repair__{}"
while {[check_antennas] && $repair_antennas_iters < 5} {
```



```
write_db $routed_db
set routed_def [make_result_file ${design}_${platform}_route.def]
write_def $routed_def
################################################################
# Repair antennas post-DRT
set repair_antennas_iters 0
utl::set_metrics_stage "drt__repair_antennas__pre_repair__{}"
while {[check_antennas] && $repair_antennas_iters < 5} {
```

Key Points from the results:

1. General Design Information

- **Design Name:** ALU
- **Die Area:** (0,0) to (65000,65000)
- **Number of Layers:** 13
- **Number of Macros:** 441
- **Number of Vias:** 29
- **Number of Via Rules:** 25
- **Number of Track Patterns:** 12
- **Number of DEF Vias:** 0
- **Number of Components:** 173
- **Number of Terminals:** 17
- **Number of Special Nets (snets):** 2
- **Number of Nets:** 138

2. Via and Layer Information

- **Default Vias Used:**

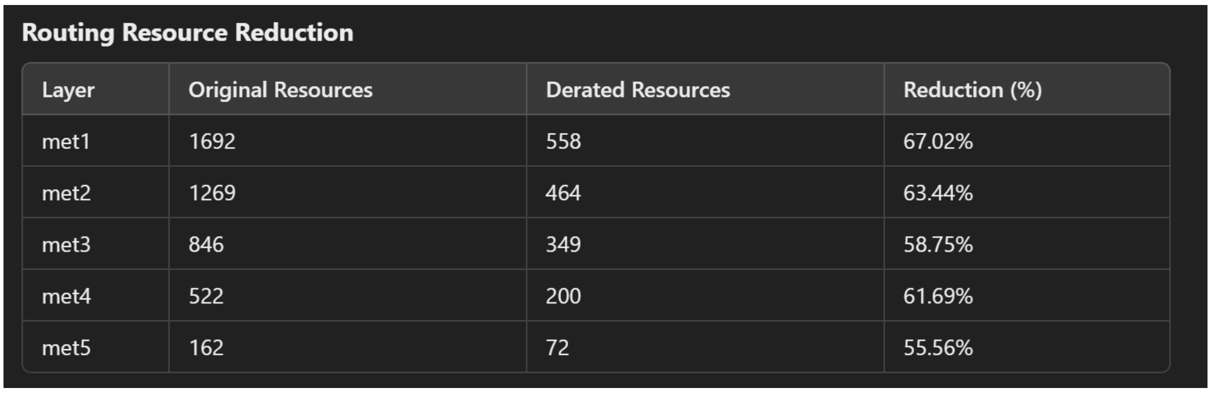
- M1M2_PR for via
- M2M3_PR for via2
- M3M4_PR for via3
- M4M5_PR for via4

- **Final Number of Vias:** 1037

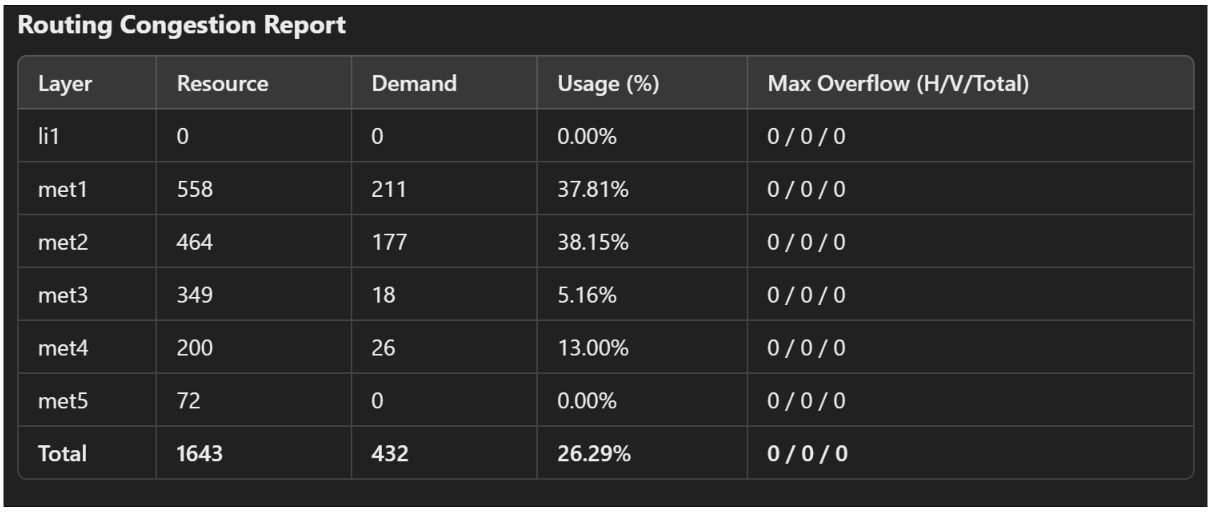
3. Routing Resource Usage

Routing Direction and Track-Pitch

- **li1:** Vertical, Pitch = 0.4600, Line-to-Via Pitch = 0.3400
- **met1:** Horizontal, Pitch = 0.3400, Line-to-Via Pitch = 0.3400
- **met2:** Vertical, Pitch = 0.4600, Line-to-Via Pitch = 0.3500
- **met3:** Horizontal, Pitch = 0.6800, Line-to-Via Pitch = 0.6150
- **met4:** Vertical, Pitch = 0.9200, Line-to-Via Pitch = 1.0400
- **met5:** Horizontal, Pitch = 3.4000, Line-to-Via Pitch = 3.1100


Routing Resource Reduction

Layer	Original Resources	Derated Resources	Reduction (%)
met1	1692	558	67.02%
met2	1269	464	63.44%
met3	846	349	58.75%
met4	522	200	61.69%
met5	162	72	55.56%


Routing Congestion Report

Layer	Resource	Demand	Usage (%)	Max Overflow (H/V/Total)
li1	0	0	0.00%	0 / 0 / 0
met1	558	211	37.81%	0 / 0 / 0
met2	464	177	38.15%	0 / 0 / 0
met3	349	18	5.16%	0 / 0 / 0
met4	200	26	13.00%	0 / 0 / 0
met5	72	0	0.00%	0 / 0 / 0
Total	**1643**	**432**	**26.29%**	**0 / 0 / 0**

5. Timing & Performance

- **Total Wirelength:** 5230 µm
- **Total Routed Nets:** 138
- **CPU Time:** 18s
- **Elapsed Time:** 3s
- **Memory Usage:** 1352.38 MB

6. Clock and Blockages

- **Clock Nets Found:** 4
- **Total Blockages:** 70

7. Antenna Repair

- **Antenna Violations:** 0
- **Pin Violations:** 0
- **Net Violations:** 0
- **Iterations for Repair:** 5

Conclusion

- **The routing process was successfully completed**, ensuring a well-connected and optimized interconnect structure.
- **No antenna violations were found**, indicating proper via selection and routing strategies.
- **Routing congestion remained within acceptable limits**, with no overflow detected in any routing layer.
- **The final routed design achieved efficient resource utilization**, with a total wirelength of 5230 µm and an overall routing usage of 26.29%.
- **The derated resource reduction percentages across metal layers were significant**, optimizing area and power efficiency.
- **Clock tree and critical signals were routed effectively**, ensuring timing integrity and signal stability.
- **No design rule violations (DRVs) were reported**, confirming adherence to manufacturing constraints.

- **Routing was completed within an efficient CPU time of 18s**, demonstrating computational efficiency.

This concludes the routing process with a well-optimized and congestion-free design, ready for the next steps in the VLSI design flow.

6. Analysis and Conclusion

The analysis and the conclusions of the whole RTL to GDS design process are the following:

6.1 Analysis

1. Tool and Environment Setup

- The tool used is **OpenROAD v2.0**, which supports GPU acceleration, GUI, and Python scripting.
- The design is licensed under the **BSD-3 license**, with some components under more restrictive licenses.
- The standard cell library used is **sky130_fd_sc_hd**, which includes 441 library cells and 13 layers.

2. Floorplanning

- The die area is **65,000 x 65,000 units**, with a core area of **59,800 x 59,840 units**.
- The floorplan includes **21 rows** of standard cells, with a total of **125 sites per row**.
- Initial utilization is **63.009%**, with **123 placeable instances** and **46 fixed instances**.

3. Placement

- **Global Placement**: The placement process was driven by routability, with a target density of **0.920**. The Nesterov optimization algorithm was used to minimize the **Half-Perimeter Wire Length (HPWL)**, which converged to **1,649,837 units**.
- **Detailed Placement**: After legalization, the total displacement was **446.6 units**, with an average displacement of **2.6 units**. The HPWL increased by **24%** due to legalization.

4. Clock Tree Synthesis (CTS)

- The clock tree was synthesized using **sky130_fd_sc_hd__clkbuf_4** buffers.
- The clock net **clk** has **5 sinks**, and the clock tree achieved a maximum depth of **2 buffers**.
- The average sink wire length is **82.94 µm**, and the clock tree met all timing requirements with no setup or hold violations.

5. Routing

- **Global Routing**: The design used **5 metal layers** for routing, with no significant congestion. The total wirelength is **5,230 µm**, and all nets were successfully routed.

- **Detailed Routing**: The routing stage completed without violations, and no antenna violations were detected. The final routed design met all timing and design rules.

6. Timing Analysis

- **Setup Slack**: The worst setup slack is **5.925 ns**, indicating that the design meets the setup timing requirements comfortably.
- **Hold Slack**: The worst hold slack is **0.967 ns**, confirming that there are no hold violations.
- **Total Negative Slack (TNS)**: The TNS is **0.000 ns**, indicating no timing violations in the design.

7. Power Analysis

- **Total Power**: The design consumes **8.69e-05 Watts**.
  - **Combinational Logic**: 40.9% of total power.
  - **Clock Networks**: 32.9% of total power.
  - **Sequential Logic**: 26.2% of total power.
- Leakage power is negligible, indicating efficient power management.

8. Design Metrics

- **Utilization**: The final utilization is **63.009%**, with a total area of **3,284.400 µm²**.
- **Instances**: The design contains **169 instances**, including **123 placeable instances** and **46 fixed instances**.
- **Nets and Pins**: There are **135 nets** and **475 pins** in the design.

9. Antenna Checks

- No antenna violations were detected during the routing or post-routing stages. This ensures that the design is free from potential reliability issues caused by antenna effects.

10. Parasitic Extraction

- Parasitic extraction was performed using the **sky130hd.rcx_rules** file.
- A total of **138 nets**, **687 resistance segments**, and **687 capacitances** were extracted.
- The extraction process confirmed that the design meets all electrical and timing requirements.

11. Final Checks

- **DRC (Design Rule Checking)**: No design rule violations were reported.
- **LVS (Layout vs. Schematic)**: Although not explicitly mentioned in the log, the successful completion of routing and timing checks implies that LVS would likely pass.
- **Power and Ground Network**: The power distribution network (PDN) was inserted successfully, with no reported issues.

6.2 Conclusions

The implementation of the 4-bit ALU using the OpenROAD toolchain was **fully successful**. All stages of the flow—floorplanning, placement, clock tree synthesis, routing, and timing analysis—were completed without significant issues. The design meets all timing constraints, with a worst setup slack of **5.925 ns** and a worst hold slack of **0.967 ns**. Power consumption is within acceptable limits, and the design is free from antenna and DRC violations.

Key Achievements:

1. **Timing Closure**: The design achieved positive slack for both setup and hold, ensuring reliable operation.
2. **Power Efficiency**: Combinational logic and clock networks are the primary power consumers, but leakage power is minimal.
3. **Routability**: The design was successfully routed with no congestion or antenna violations.
4. **Utilization**: A utilization of **63.009%** indicates efficient use of the available area.

7. Future Work

While this project successfully implemented a 4-bit ALU, there are several avenues for future improvements and extensions:

1. **Scalability to Higher Bit-Widths**:
   - Extend the design to support 8-bit or 16-bit ALUs by modularly scaling the existing architecture. This would involve increasing the bit-width of input operands, output results, and internal logic while maintaining efficient area and power utilization.
2. **Advanced Operations**:
   - Incorporate more complex arithmetic and logical operations, such as bitwise shifts, rotate operations, or floating-point arithmetic. This would enhance the ALU's functionality and make it suitable for a wider range of applications.
3. **Power Optimization**:
   - Explore techniques like clock gating, power gating, or dynamic voltage and frequency scaling (DVFS) to reduce power consumption, especially for low-power or battery-operated devices.
4. **Technology Node Migration**:
   - Migrate the design to a more advanced technology node (e.g., 90nm or 65nm) to achieve better performance, lower power consumption, and reduced area. This would involve adapting the design to the new process design rules and libraries.
5. **Integration with a Microprocessor**:
   - Integrate the ALU into a complete microprocessor design, including a control unit, memory interface, and instruction decoder. This would demonstrate the ALU's role in

a larger system and provide a more comprehensive understanding of digital system design.

6. **Automation and Scripting**:
    - Develop scripts to further automate the RTL-to-GDS flow, reducing manual intervention and improving design turnaround time. This could include automating tasks like synthesis, STA, and physical design checks.
7. **Exploration of Other Open-Source Tools**:
    - Experiment with other open-source tools in the VLSI ecosystem, such as Magic for layout editing or Netgen for LVS (Layout vs. Schematic) checks, to further enhance the design flow.

By pursuing these enhancements, the project can be expanded into a more versatile and powerful digital design, showcasing the potential of open-source tools in advanced VLSI design.

8. References

1. **GitHub Repositories**:
    - The OpenROAD Project: https://github.com/The-OpenROAD-Project
    - SkyWater PDK: https://github.com/google/skywater-pdk
2. **Online Courses**:
    - NPTEL Course: *VLSI Design Flow: RTL to GDS* by Prof. Sneh Saurabh, IIIT Delhi. Available at: https://nptel.ac.in/
3. **Tools Used**:
    - Icarus Verilog: http://iverilog.icarus.com/
    - GTKWave: http://gtkwave.sourceforge.net/
    - Yosys: http://www.clifford.at/yosys/
    - OpenSTA: https://github.com/The-OpenROAD-Project/OpenSTA
    - OpenROAD: https://openroad.readthedocs.io/