Table of Contents

**1. Abstract** ..........................................................................

**2. Introduction** ....................................................................

**3. Objectives** .......................................................................

**4. System Design and Methodology** .....................................

5. **Implementation** ..............................................................

5.1. RTL Design and Testbench...............................................  
 5.2. Logic Synthesis .............................................................   
 5.3. Static Timing Analysis (STA) ...........................................  
 5.4. Physical Design .............................................................

5.4.1. Floorplan ...................................................................  
 5.4.2. PDN ...........................................................................  
 5.4.3. Placement ..................................................................  
 5.4.4. Clock Tree Synthesis (CTS) ...........................................  
 5.4.5. Routing ......................................................................

**6. Analysis and Conclusion** .........................................................

6.1. Analysis ......................................................................  
 6.2. Conclusions .................................................................

**7. Future Work** ...............................................................................

**8. Reference** ...................................................................................

# 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:  
  


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.

Conclusion

The **RTL design and testbench verification confirm the correct functionality** of the 4-bit ALU. The **waveform output in GTKWave** 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.

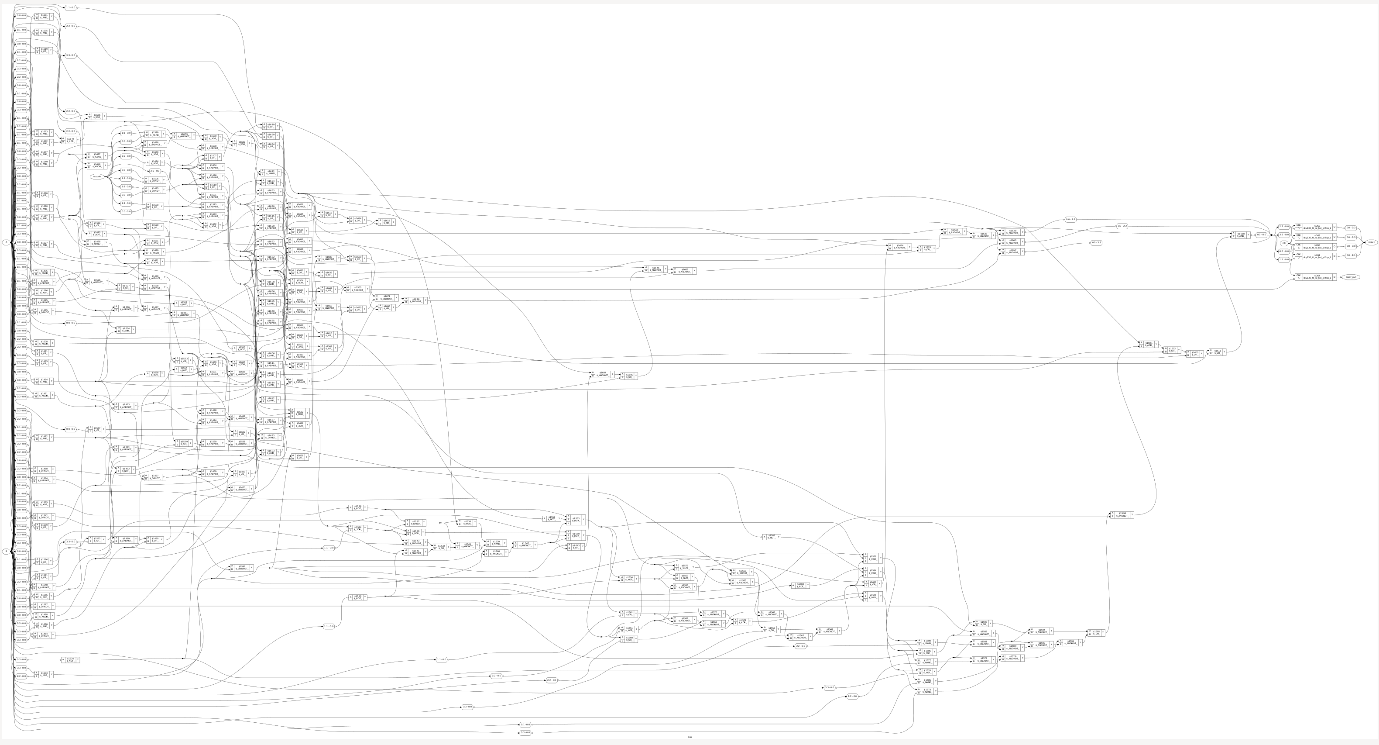
Output Files Generated

*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

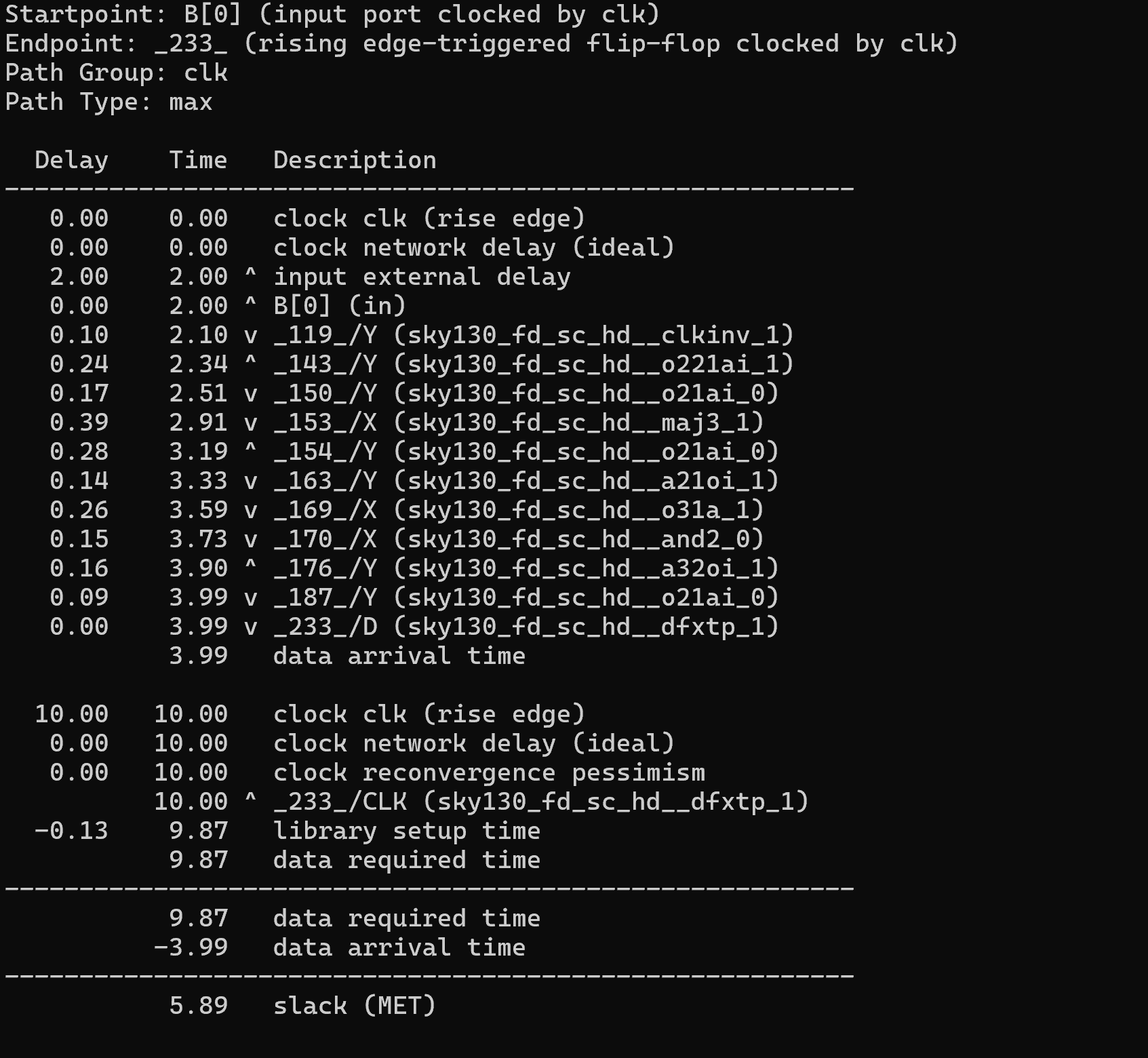
After running the STA, the following output files were obtained:

* **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

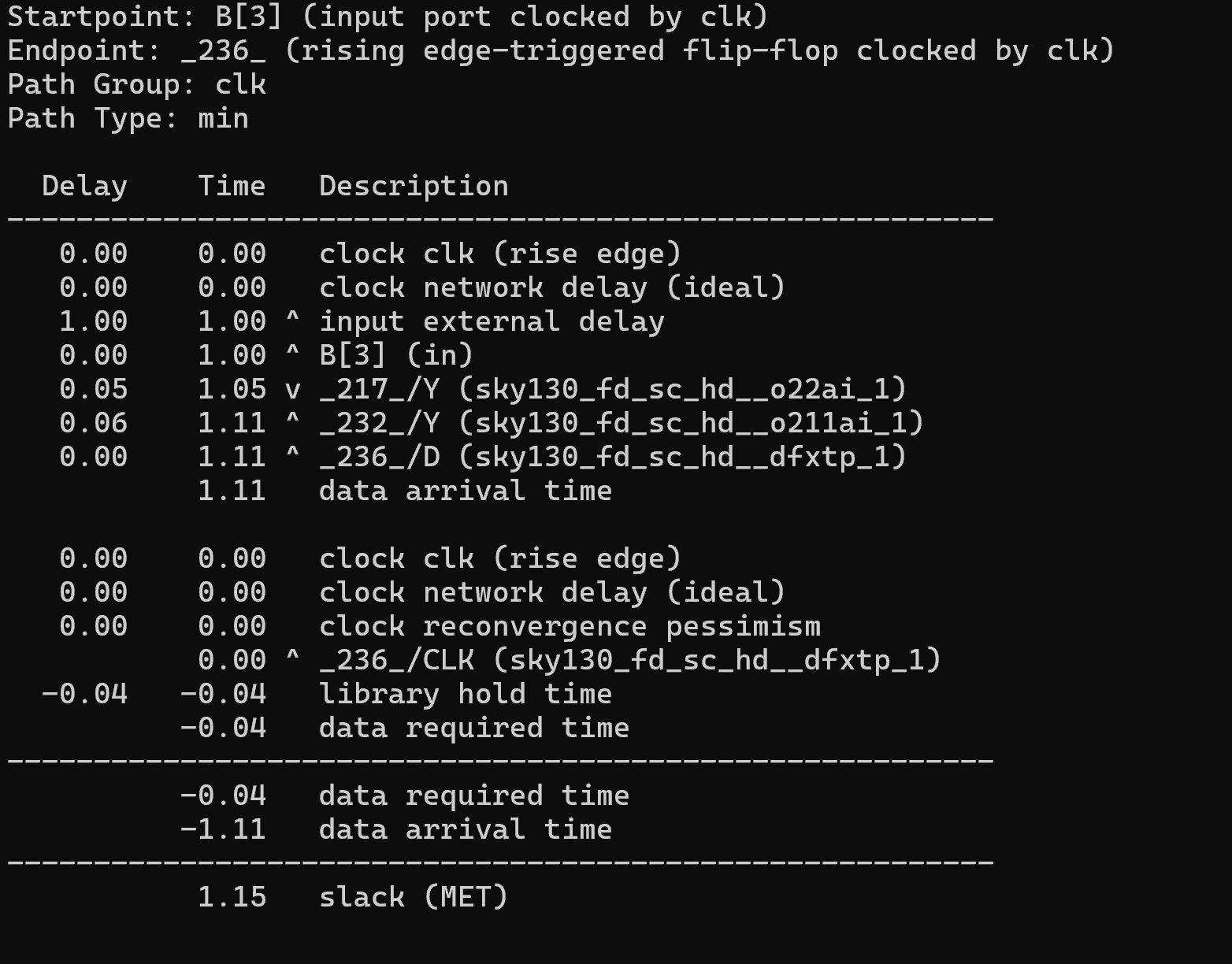
Below are the screenshots of the STA results from the following commands:

* report\_checks -path\_delay max -format full



**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



**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

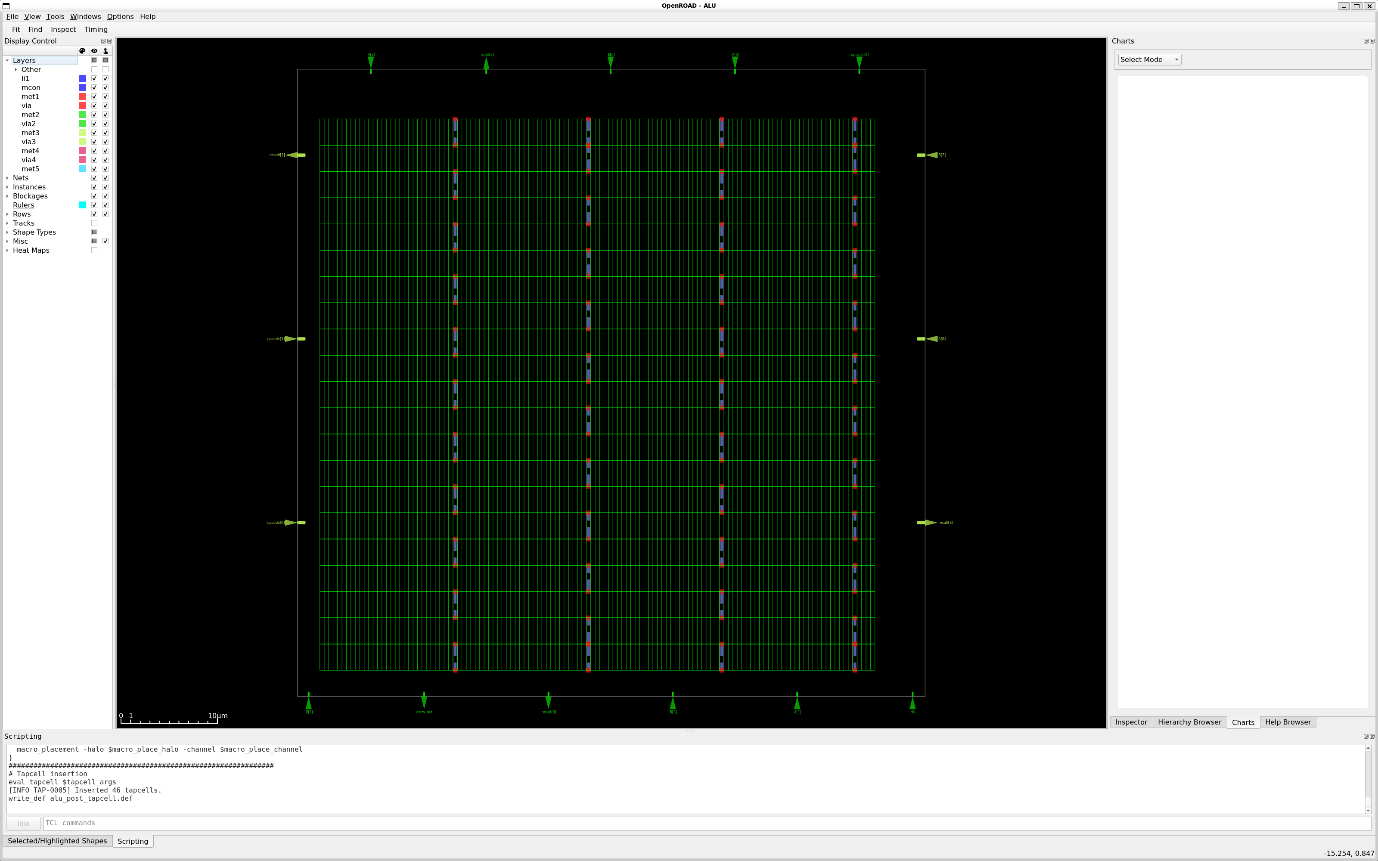
* Inserts **tapcells** to ensure power and ground connectivity across the core area.
* Generates a **DEF (Design Exchange Format) file** (alu\_post\_tapcell.def) for the next step in the flow.

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



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

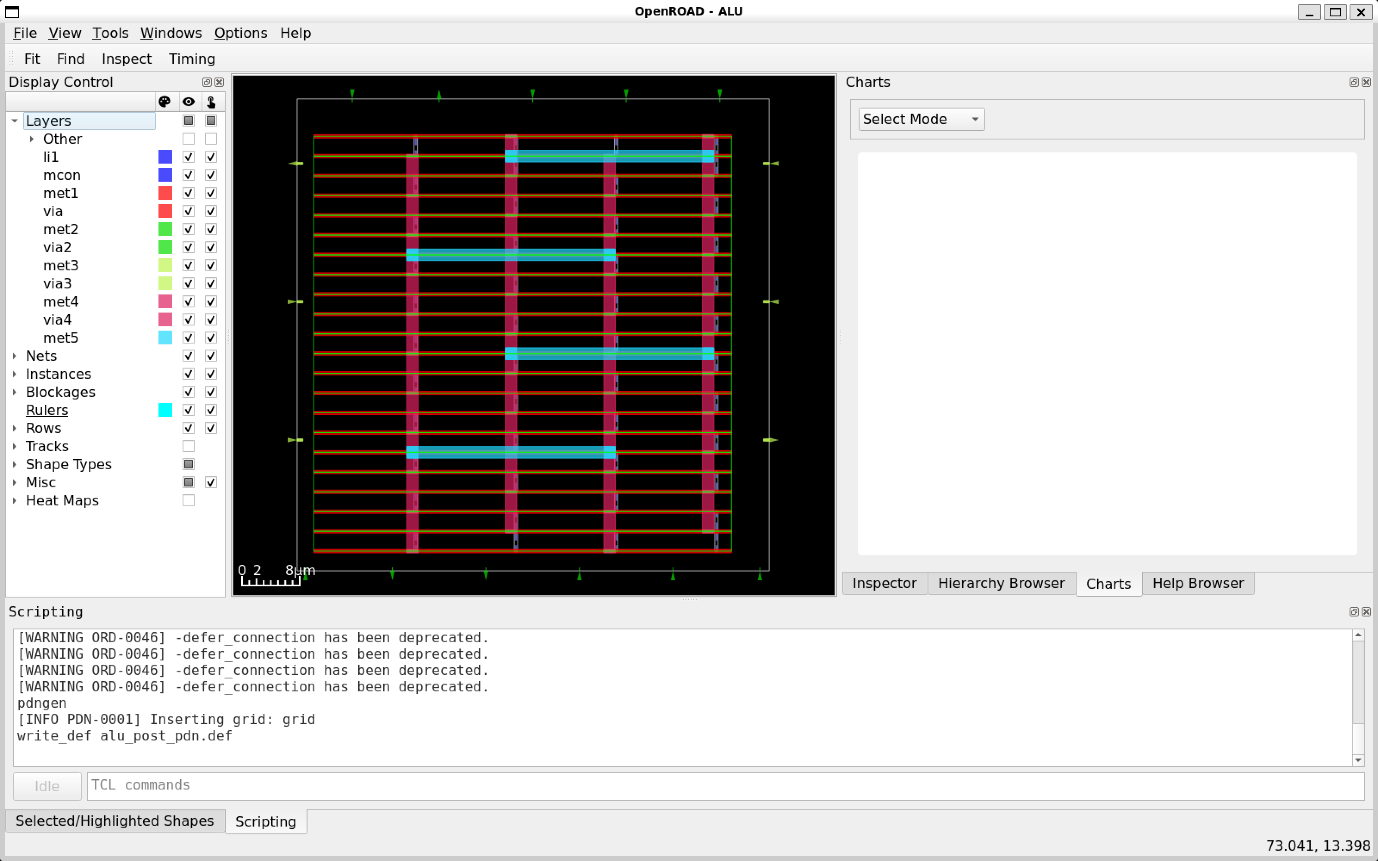
*After executing the PDN flow, the following output files are obtained:*

* **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

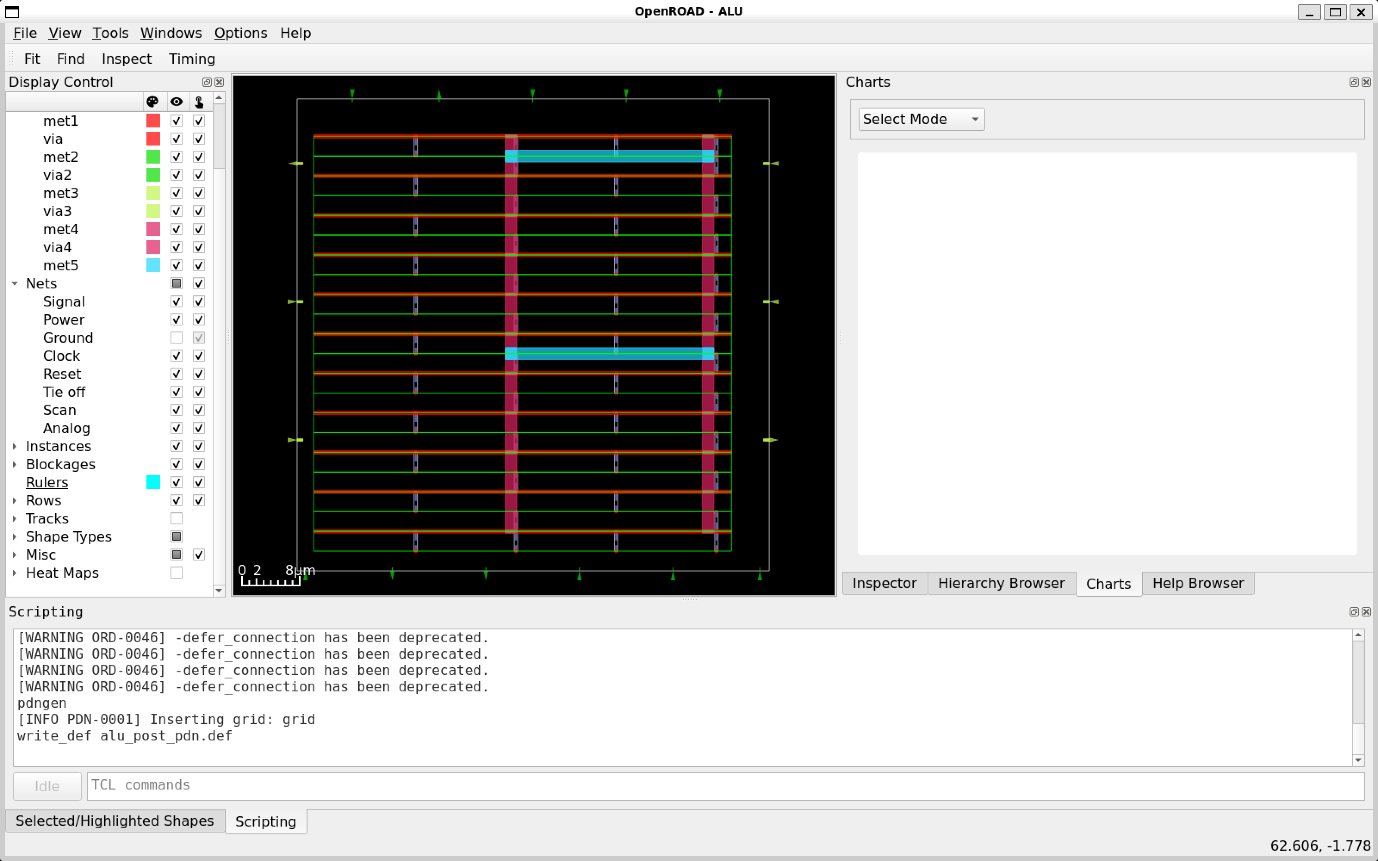
**1. Combined Power and Ground Network**

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.



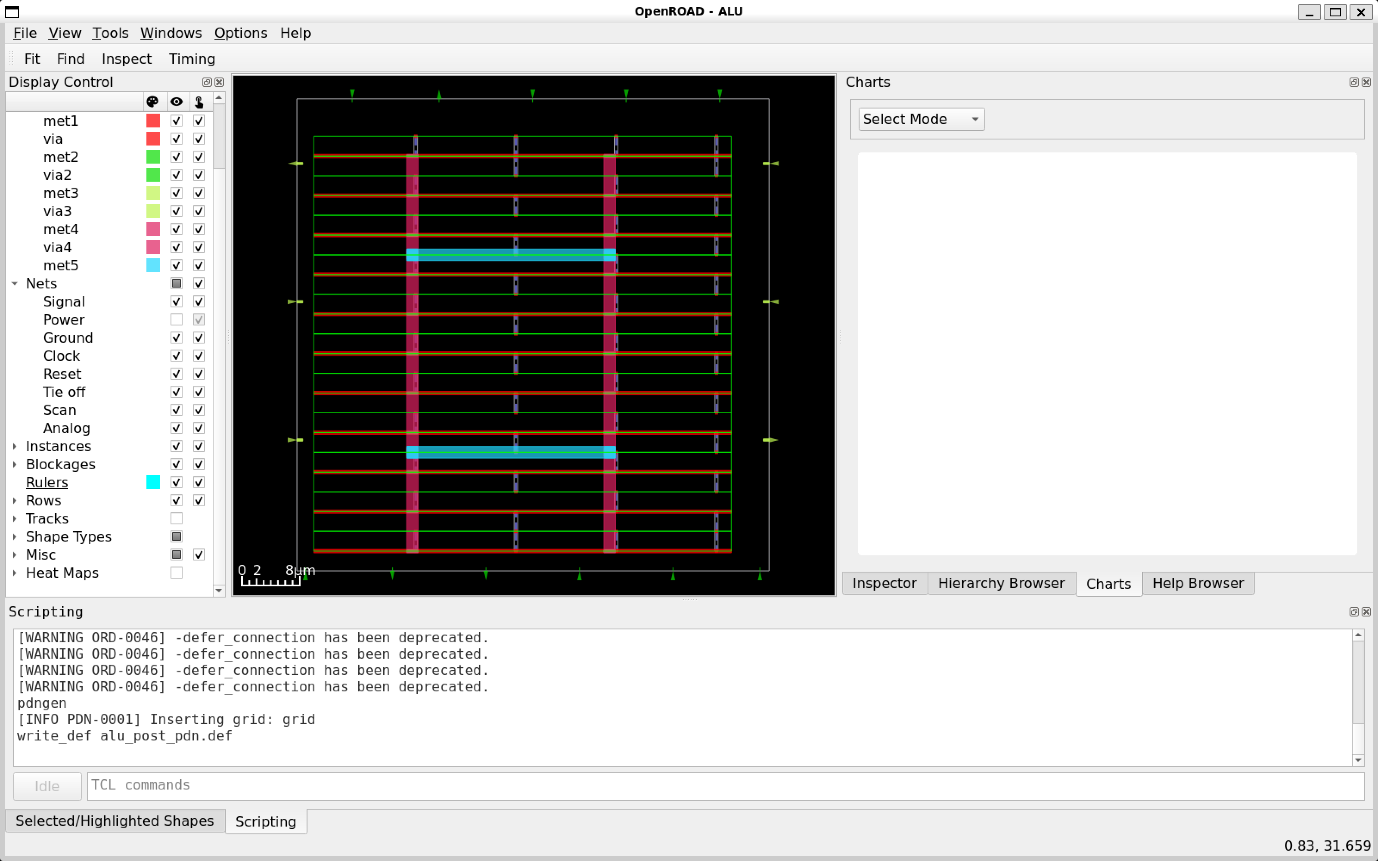
**2. Power Network Only**

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.



**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.



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

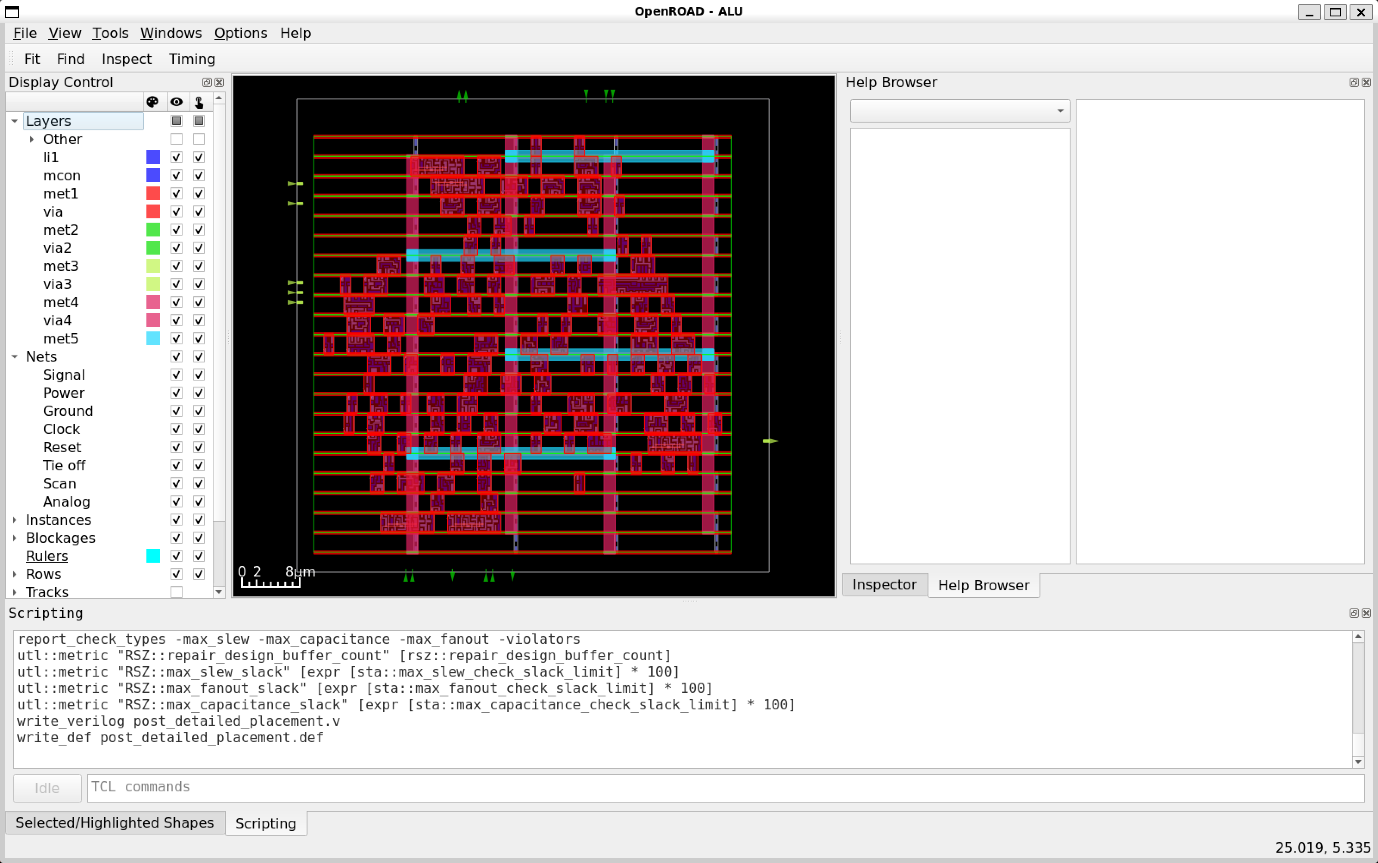
* Exports the design to Verilog and DEF files after detailed placement.

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



**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]

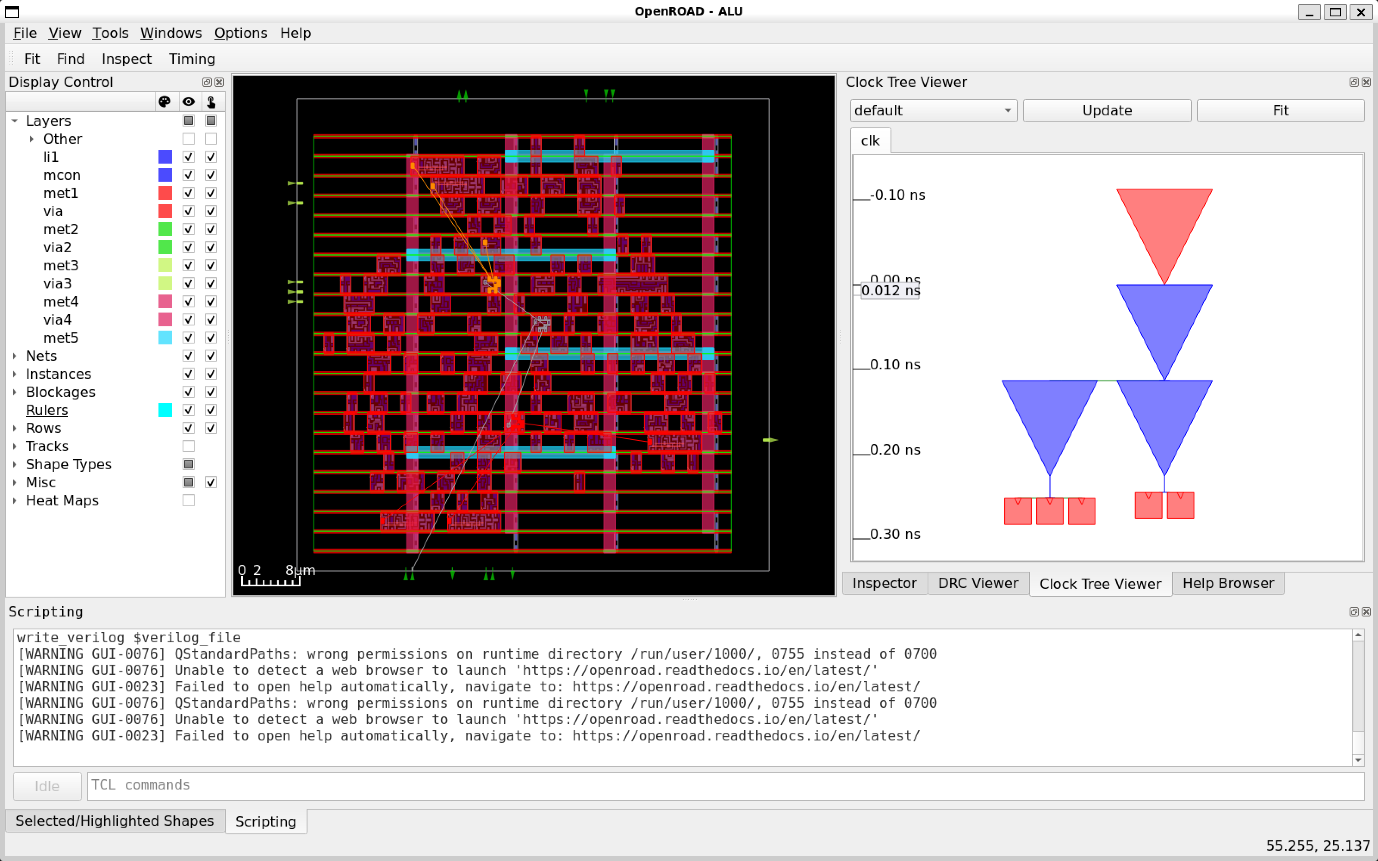
write\_verilog $verilog\_file

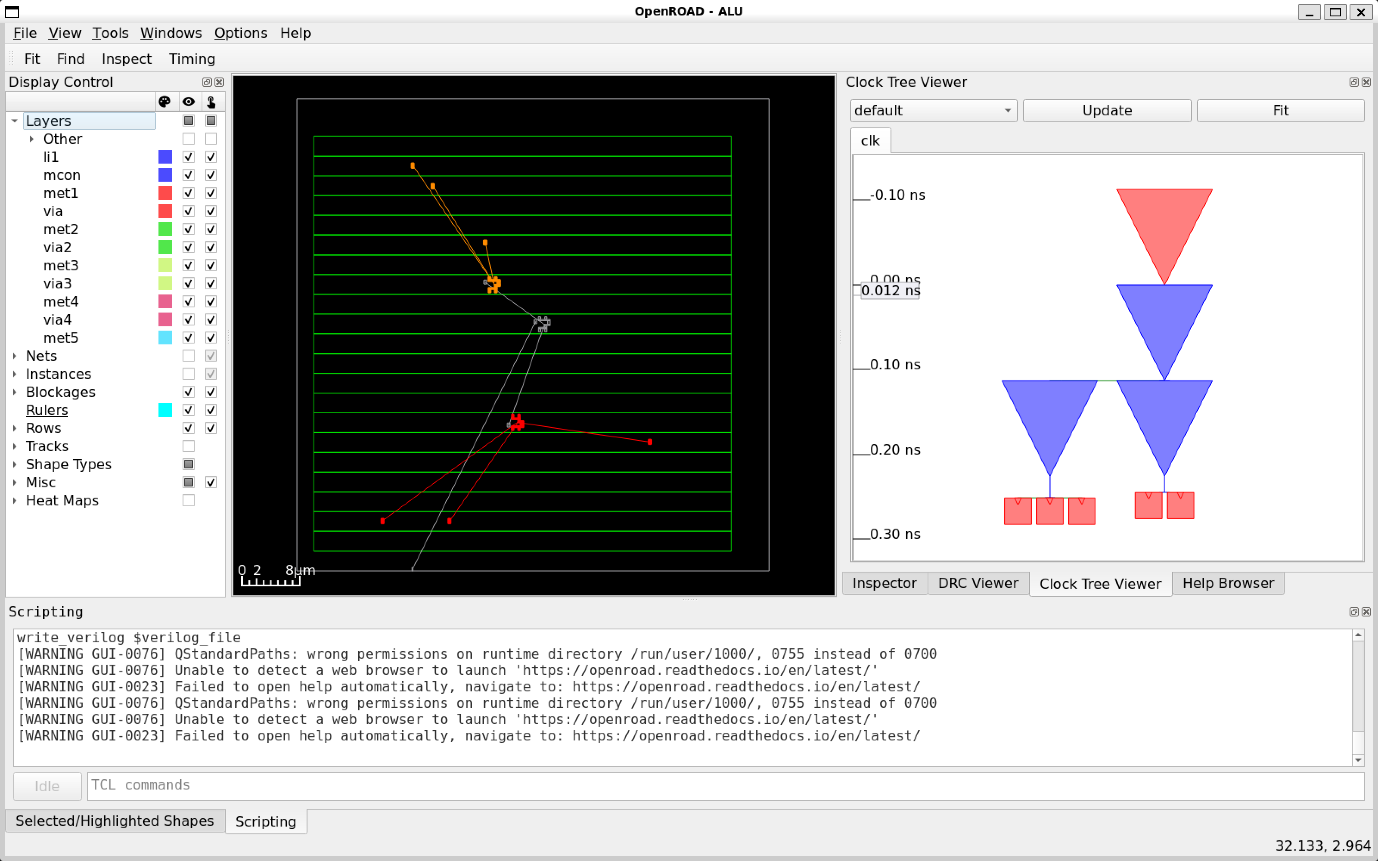
* Exports the **new Verilog netlist** after CTS, which includes **clock buffers and optimized placement**.

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





**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 {

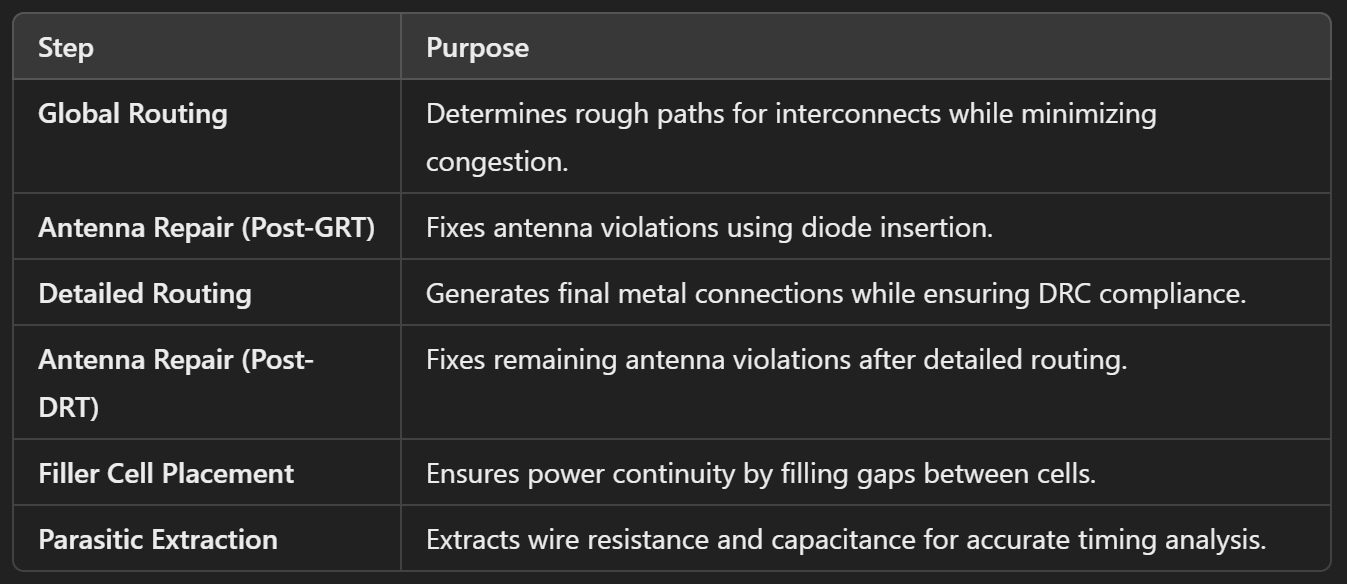
# Use global routing based parasitics inlieu of rc extraction

estimate\_parasitics -global\_routing

}

* If no **rcx\_rules\_file** is provided, it **estimates parasitics based on global routing** (less accurate than full extraction).

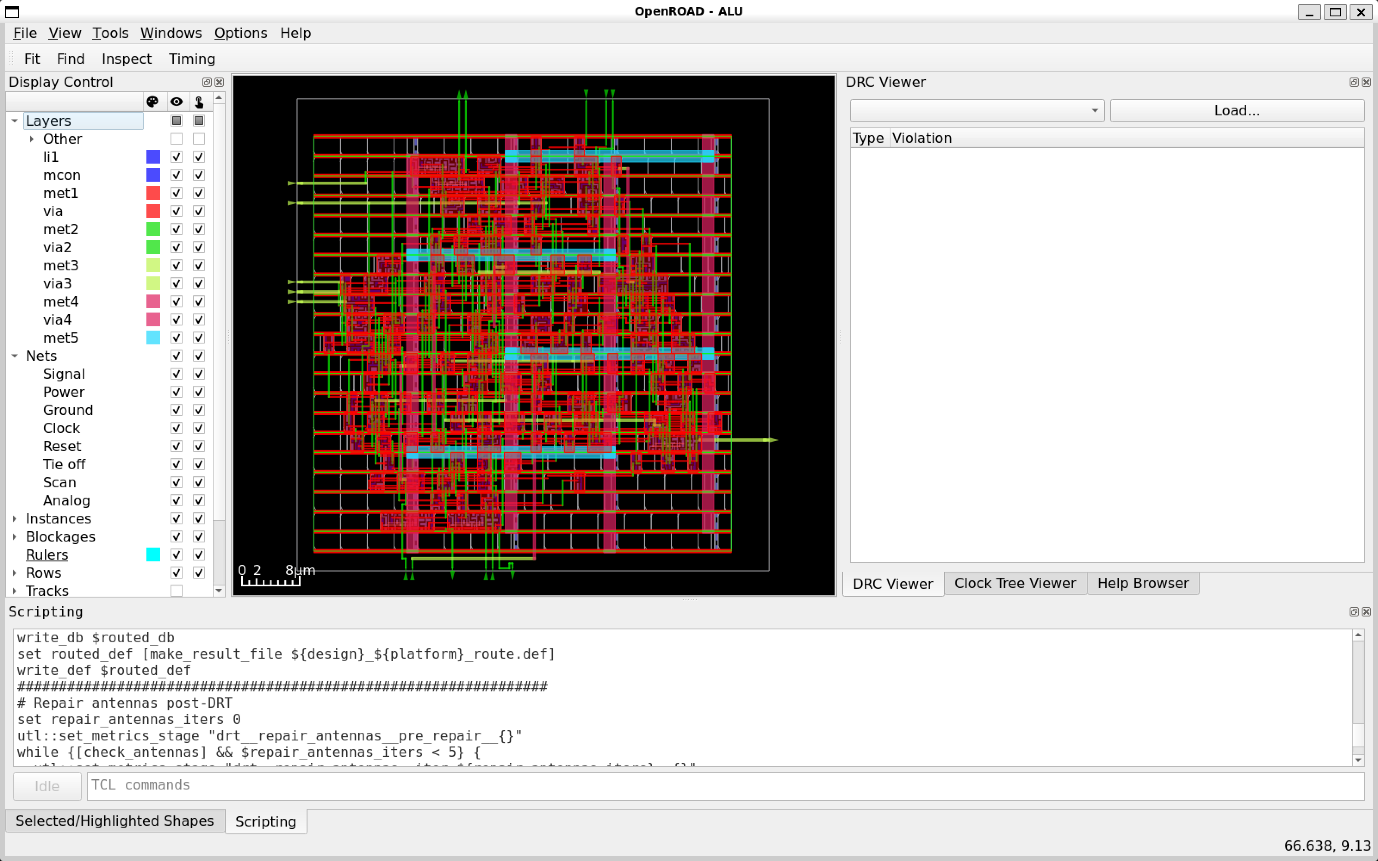
**Summary of Key Steps**

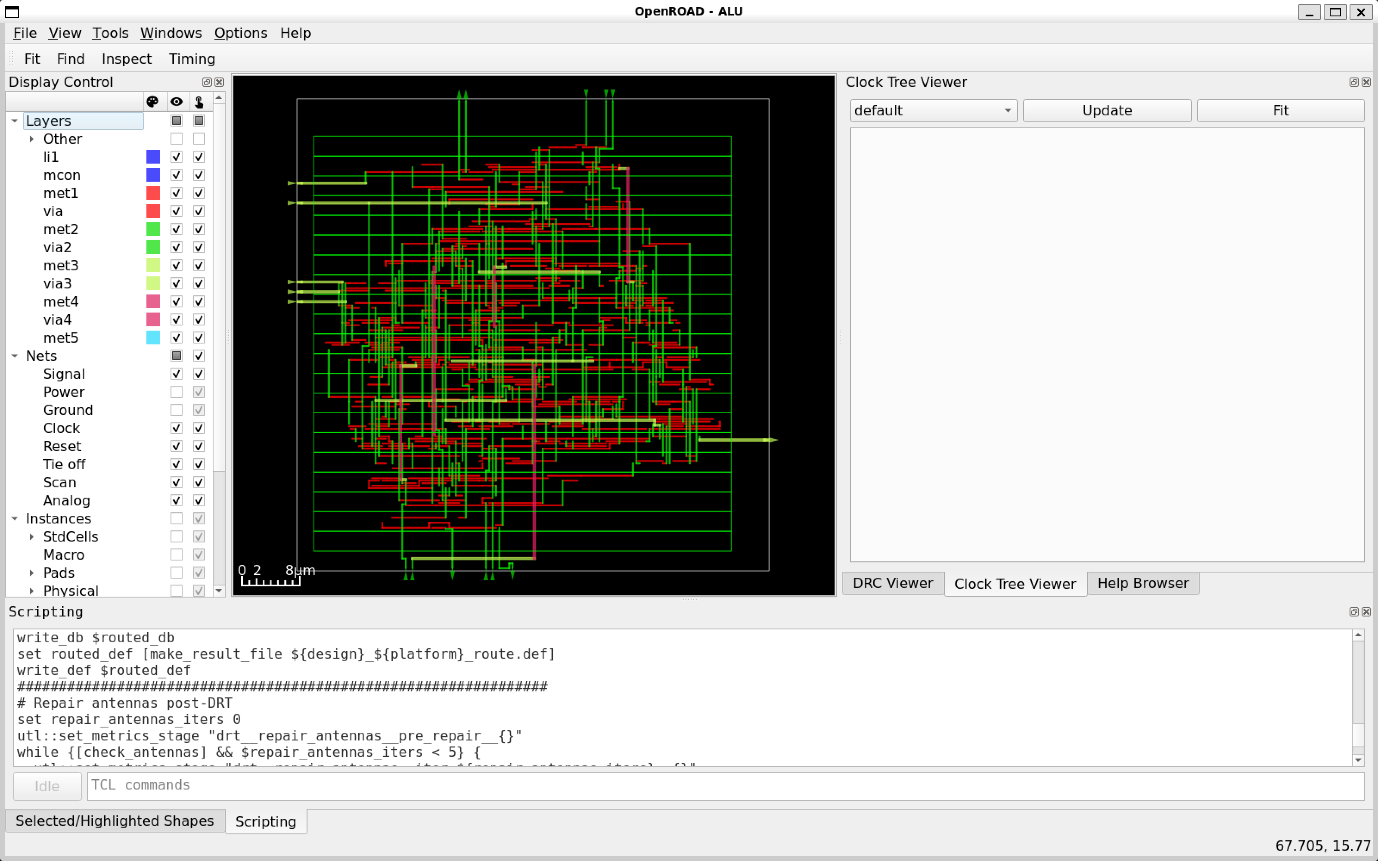


Output Files After Routing

* **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





**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

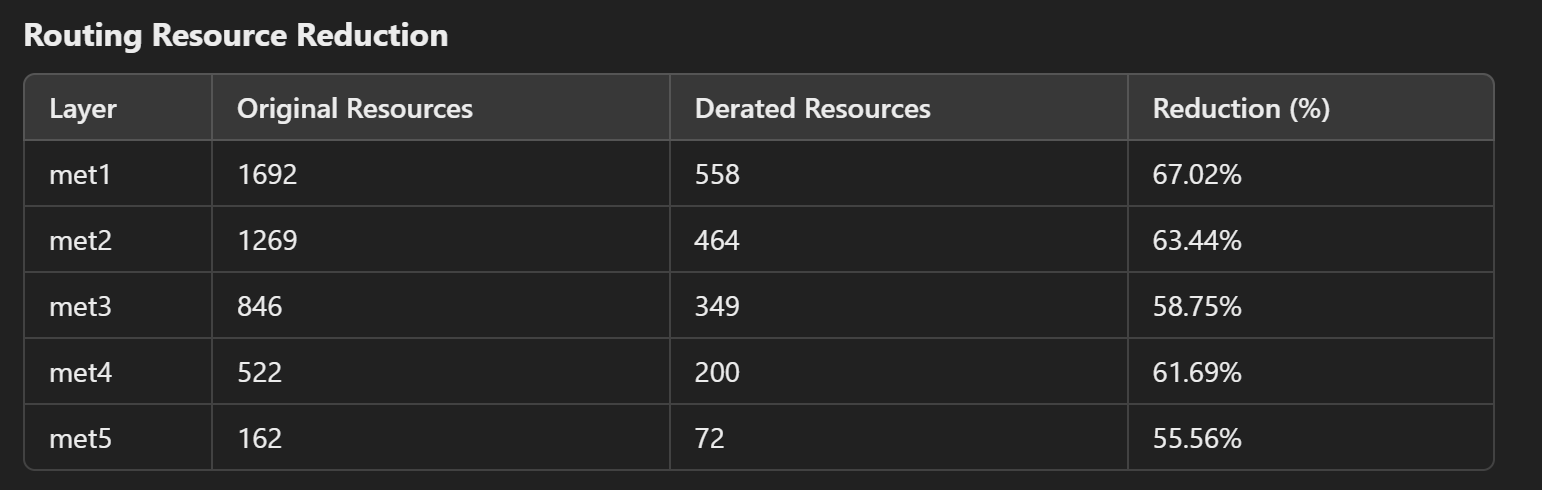
**2. Via and Layer Information**

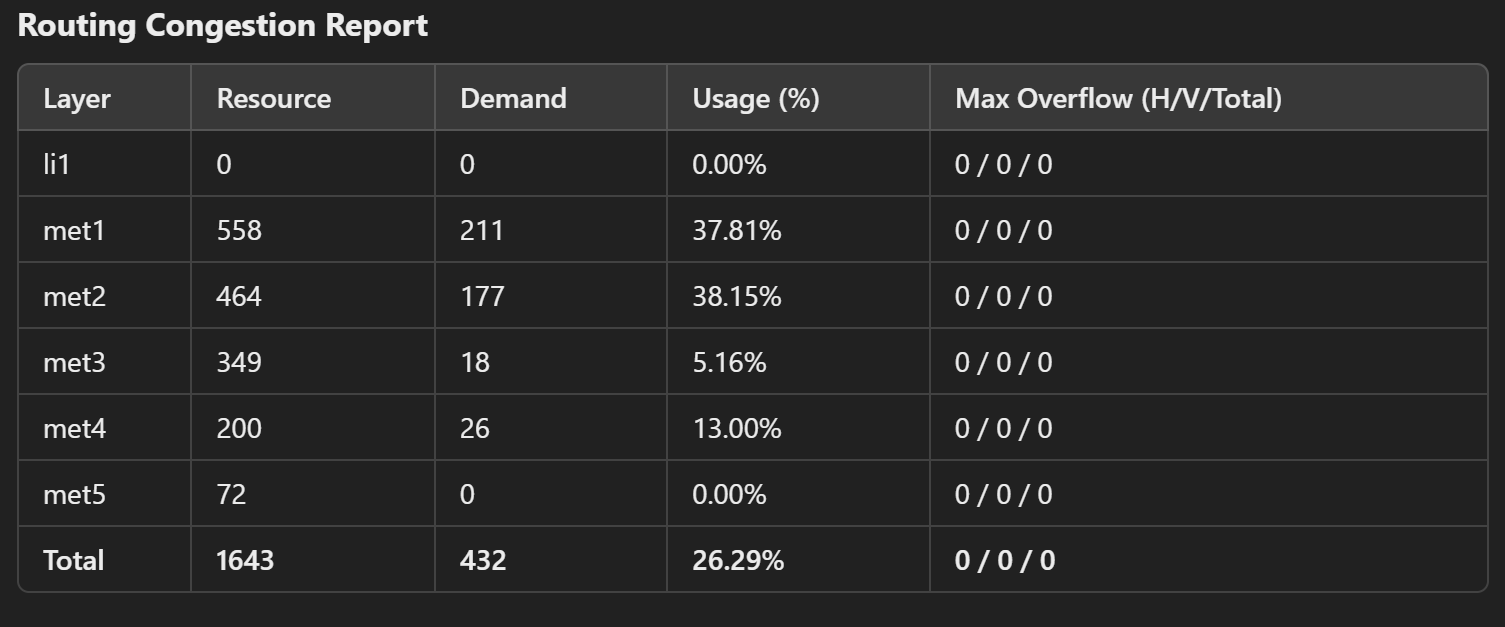
* **Default Vias Used:**
  + M1M2\_PR for via
  + M2M3\_PR for via2
  + M3M4\_PR for via3
  + M4M5\_PR for via4
* **Final Number of Vias:** 1037

**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

****



**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/>