The Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

VLSI Design flow for TSMC 28nm

RTL-to-GDSII Implementation with Verification

By:

Adan Mohsen & Leena Wattad & Lama Garrah & Ahmad Foqra

Mentor name: Zvi Webb

2024-2025

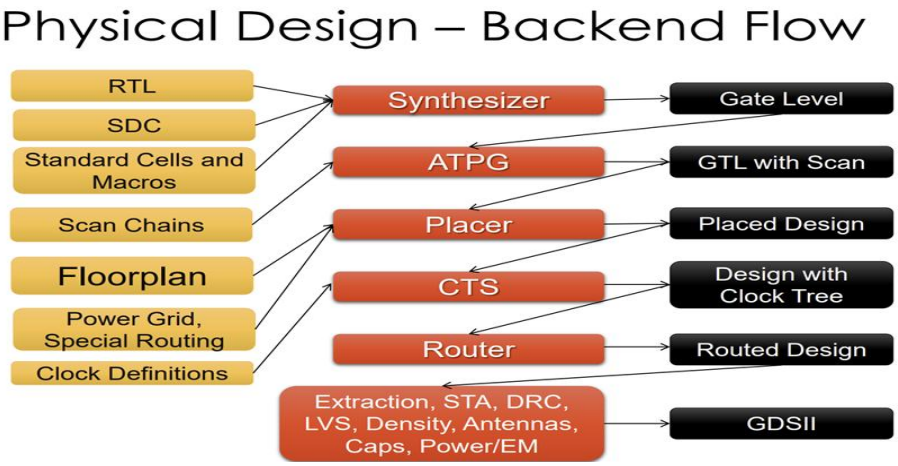
1. Introduction

The backend guide details the process of backend design carried out for the implementation of a chip design using TSMC 28nm technology.The manual describes the following steps:

1. Register Transfer Level (RTL):
   Describing the functionality of the system at a logical level using a high-level language.
2. RTL Verification:
   Verifying and simulating the correctness of the design using dedicated verification tools.
3. Memory Integration:
   Using compilation tools to generate and integrate memory blocks into the design (if required).
4. Synthesis:
   Converting the RTL description into a logical gate-level netlist.
5. Initial Floorplanning:
   Defining the preliminary placement of major components on the chip.
6. Component Placement:
   Physically arranging the components of the chip based on the logical design.
7. Clock Tree Synthesis (CTS):
   Designing a structure to uniformly distribute clock signals across the chip.
8. Routing:
   Establishing physical connections between components using metal layers.
9. Parameter Extraction:
   Extracting physical parameters of the design, such as resistance and capacitance of connections.
10. Static Timing Analysis (STA):
    Analyzing and optimizing the timing of signals in the design.
11. Input/Output Interface Addition (I/O):
    Integrating external interface components into the design.
12. Cell and Metal Filling:
    Adding filler cells and metal fills to enhance the design's stability.
13. Compliance Checks (LVS/DRC):
    Ensuring logical and physical compliance with the design specifications and technological standards.
14. GDSII File Generation:
    Producing the final GDSII file representing the design for manufacturing.

2. Backend Flow

The backend flow involves converting RTL code into a GDSII database, a critical and intricate step in chip manufacturing to ensure proper functionality. This manual presents instructions for the backend process using Cadence and Synopsys automated tools. Upcoming sections and a chart will summarize the key phases of the flow and the associated tools:



2.1. Synthesis with scan chain insertion

It involves translating RTL code into a gate-level netlist while embedding test structures for easier fault detection.

2.2. Gate Level Simulation

This is the process of verifying the logical correctness of a gate-level netlist against

the original RTL specifications.

2.3. FORM RC

It ensures that the post-synthesis gate-level netlist remains functionally identical to the

initial RTL code.

2.4. Performing the physical design (layout) with Fusion Compiler

It entails creating the chip's physical structure from the gate-level netlist.

2.5. Physical Verification and Signoff

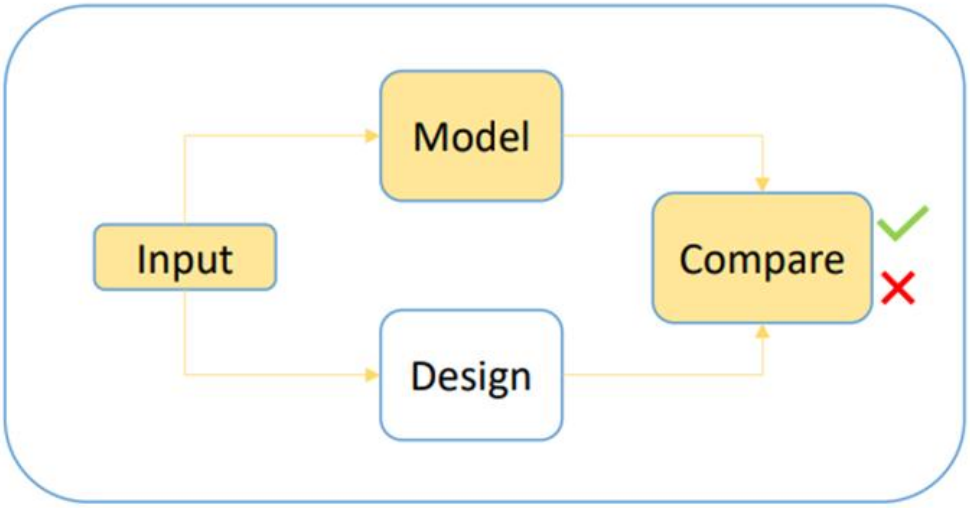
This is the final check to ensure the layout meets all design, fabrication, and regulatory standards before chip manufacturing.

3.Frontend Flow

RTL Verification:

1. Introduction to RTL Verification

RTL verification involves simulating the Register Transfer Level (RTL) description of a digital circuit. The goal is to ensure that the logic implementation matches the intended specification. Early detection of design errors is crucial for ensuring functional correctness before moving to later stages of development like synthesis or physical design.

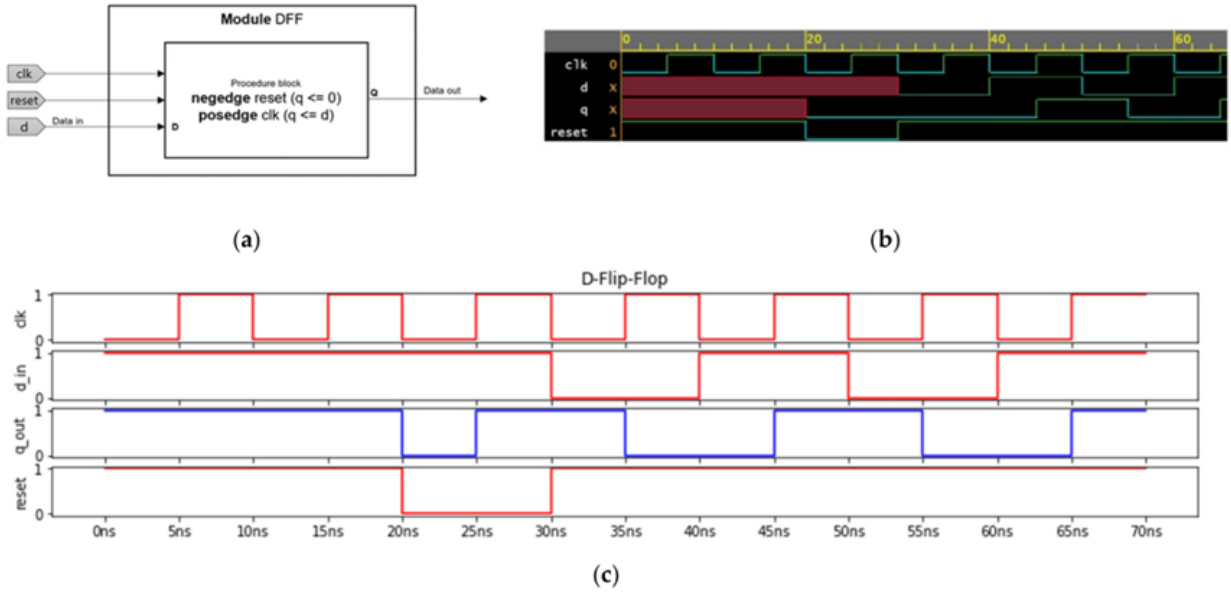


2. The Role of Simulation in RTL Verification

Simulation is one of the primary techniques for RTL verification, focusing on the following:

- **Driving inputs of the design: Providing input signals to the module.**
- **Checking the outputs of the design: Observing how the module responds to those inputs.**
- **Bug detection: Simulation allows the designer to observe potential functional bugs early in the design cycle.**

By simulating the logic, designers can ensure the RTL implementation behaves as expected and matches the design specification.

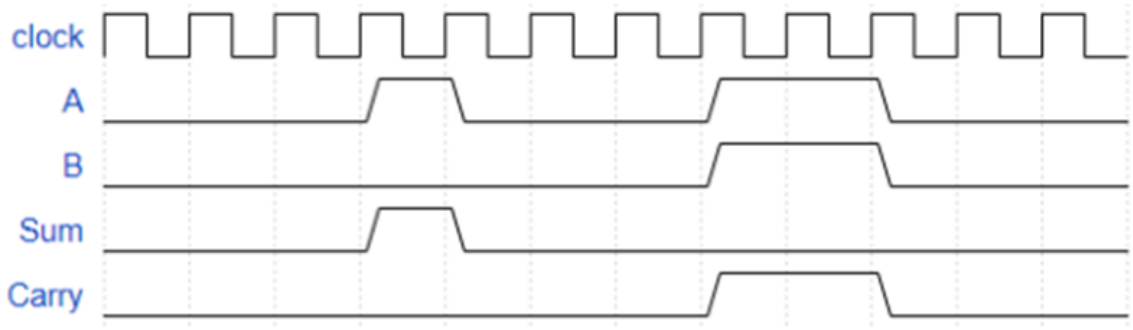


3. Various Verification Approaches

In RTL verification, there are multiple techniques to validate the design:

- **Formal verification: A mathematical approach to prove correctness.**
- **Simulation: A time-based approach where inputs are applied and outputs are checked.**
- **Emulation: A hybrid approach that combines hardware and software tools.**

While each has its strengths, we will focus primarily on simulation in this project.



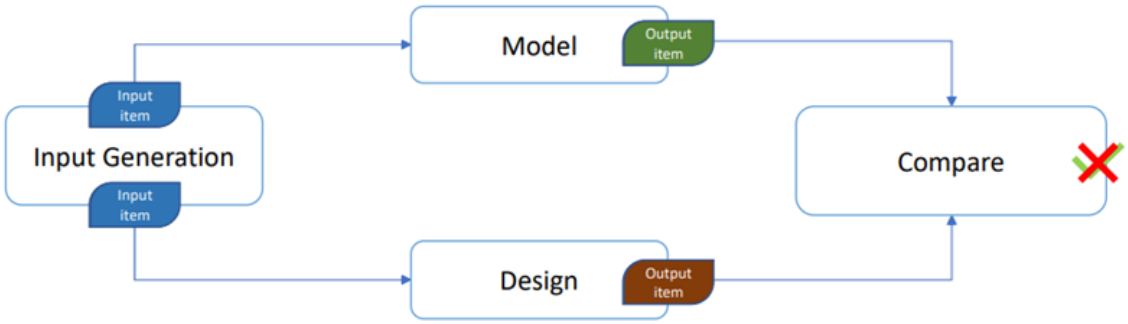
Try to know what we simulate in this image!!!

4. Core Concepts in RTL Verification

RTL verification revolves around two main concepts:

1. **Driving inputs: Feeding input values to the design under test .**
2. **Checking outputs: Validating the outputs to ensure the design behaves correctly.**

The ultimate goal of this process is to find bugs at the design stage to avoid costly mistakes later in the process.



5. Design Module Instantiation

For effective verification, a design module is instantiated within a testbench. The testbench is responsible for:

- **Driving the inputs.**
- **Observing and checking the outputs.**

The testbench must be well-structured to automate the verification process, applying various stimulus and checking the corresponding responses to identify bugs early.

6. Challenges in Scaling RTL Verification

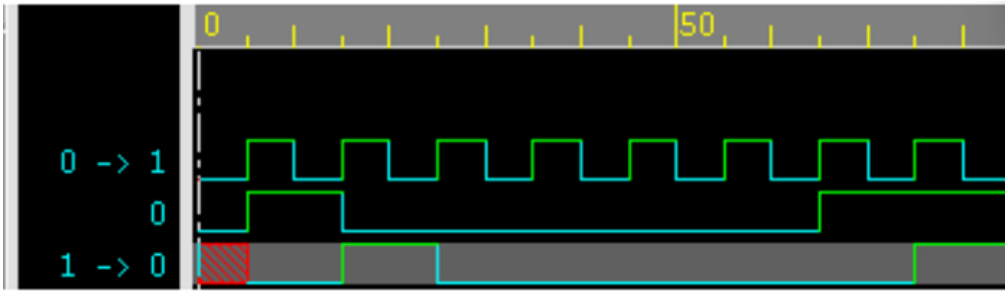
As designs grow in complexity, certain verification tasks can become overwhelming:

- **Waveform observation: Monitoring waveforms manually doesn't scale well for large designs.**

  Solution: Create a model of the design to automate checks and generate test cases systematically.

- **Manual input selection: Manually selecting inputs for large designs can be cumbersome.**

Solution: Automate input generation through constrained random stimulus to ensure wide coverage of test cases.



7. Automation in RTL Verification

Automating aspects of verification significantly enhances efficiency, scalability, and effectiveness. Here are some strategies:

- **Modeling the design: Using a program written in a Hardware Verification Language (HVL) like SystemVerilog to create automated testbenches.**
- **Automatic input generation: Using randomization to generate diverse input combinations, ensuring wide coverage without manual intervention.**

8. The Role of Verification Tools

Verification tools are essential for automating the RTL verification process. A typical setup includes:

- **HDL (Hardware Description Language): Used to describe the design.**
- **HVL (Hardware Verification Language): Used to write verification code, typically SystemVerilog.**
- **Simulator: The tool that runs the simulation, taking in both the design (HDL) and verification (HVL) code, and checks whether the design behaves as expected.**

******Popular simulators include ModelSim, VCS, and Xcelium.**

9. SystemVerilog for Verification

SystemVerilog is widely adopted in the industry for RTL verification, and it offers several key features:

- **Basic data types: For representing signals, registers, and other elements of the design.**
- **Flow control: Conditional and loop structures for controlling the flow of testbenches.**
- **Functions and tasks: Reusable code blocks for efficient testbench construction.**

- **Classes: Object-oriented features that help in structuring large verification environments.**

10. Testbench Architecture

A well-structured testbench is essential for effective simulation. The architecture typically consists of the following components:

- **DUT (Design Under Test): The RTL design being verified.**
- **Stimulus generator: Responsible for generating input sequences.**
- **Checker: Monitors and checks the outputs against expected results.**
- **Monitor: Observes and collects data during simulation for analysis.**

11. Writing a Testbench for a Priority Arbiter Module

Let's take a simple example of verifying a Priority Arbiter module using SystemVerilog.

- **Testbench Structure:**
  - **Instantiate the Priority Arbiter.**
  - **Define input signals and monitor outputs.**
  - **Apply a sequence of tests to check if the arbiter correctly prioritizes different requests.**
- **Randomization: To cover a wide range of scenarios, random inputs can be generated to test different combinations of requests and arbitration outcomes.**

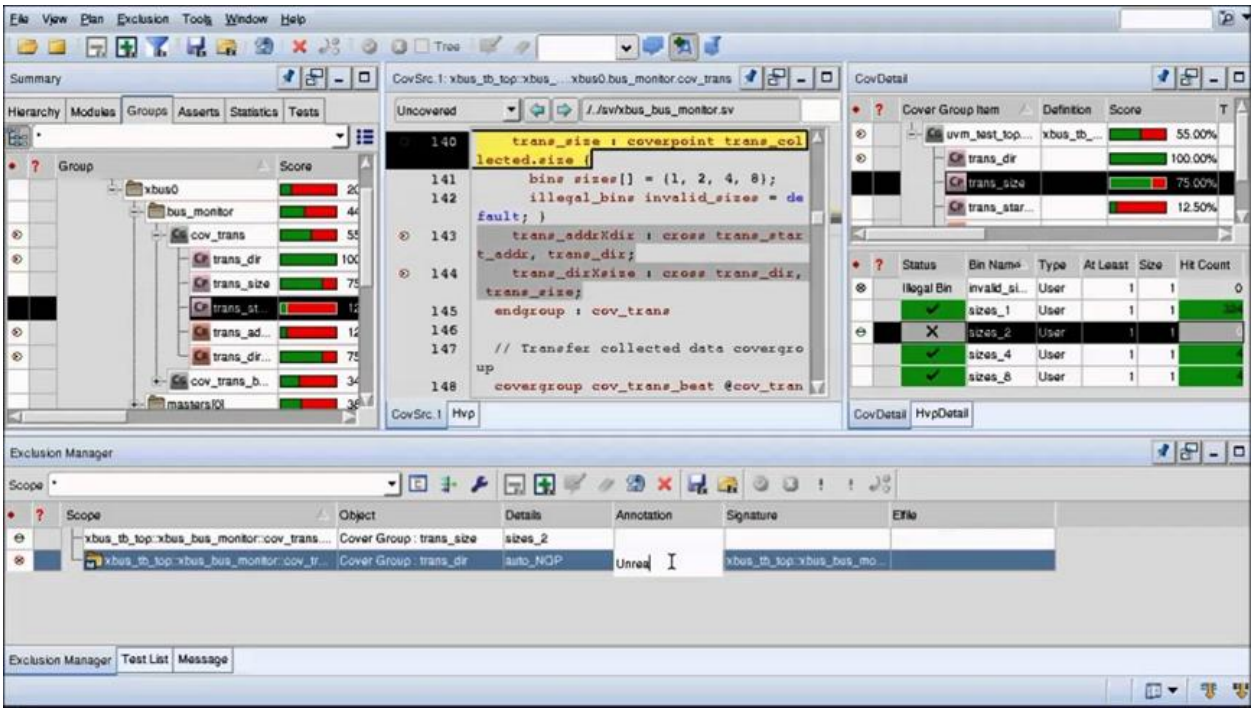
12. Constrained Random Verification

Constrained random verification involves generating random inputs while applying constraints to ensure that the inputs are within the bounds of valid scenarios. This approach helps explore a large design space, ensuring high coverage and uncovering corner-case bugs.

13. Coverage-Driven Verification

Coverage-driven verification is a key part of ensuring that all relevant scenarios are tested. Coverage metrics such as code coverage, functional coverage, and random coverage provide insights into which parts of the design have been exercised by the testbenches.

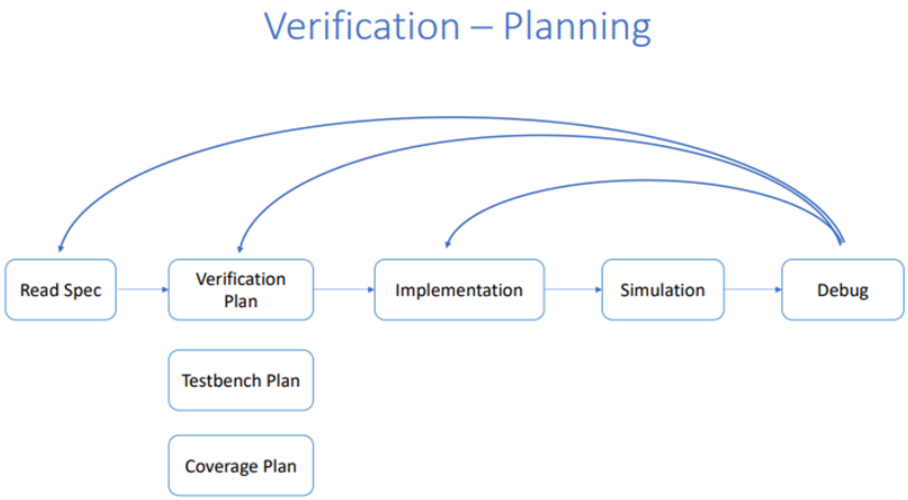
- **Code Coverage: Ensures that all lines of code in the design are exercised.**
- **Functional Coverage: Ensures that all functional scenarios are tested.**
- **Random Coverage: Ensures random input combinations cover all possible functional states.**



14. Integration and Verification Plan

Verification is a complex, iterative process that requires detailed planning. Key elements of a verification plan include:

- **Testbench architecture: The overall structure of the test environment.**
- **Test scenarios: Specific tests to be run.**
- **Coverage goals: What areas of the design need to be fully exercised.**
- **Automation goals: What aspects of the process can be automated for scalability.**



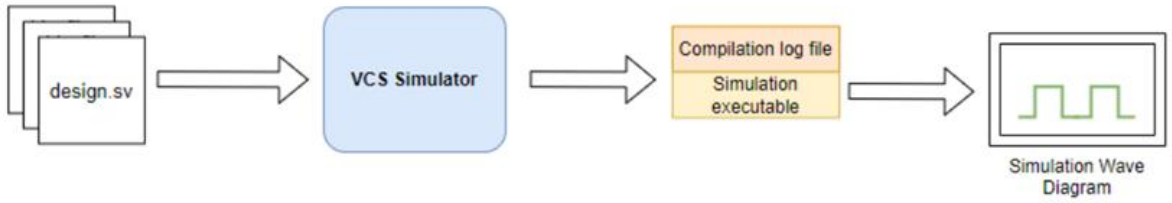
Verification Setup (using Linux machine)

In our project we will use Synopsys Euclide to simplifies RTL code writing, provides real-time bug detection, and optimizes code for design and verification flows in SystemVerilog.

Our goal is to demonstrate our simulation tools.

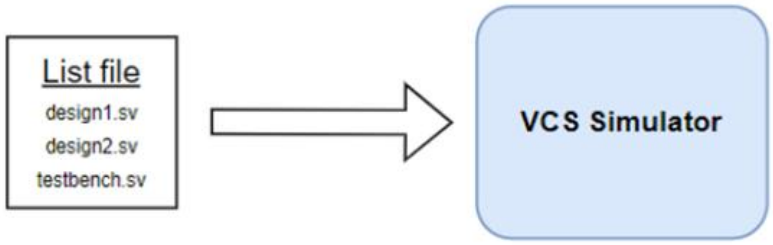
VLSI Coding Basics

As we have learned, verification can be done using simulation. We use a simulator software that takes in design/verification files (written in HDL/HVL) and compiles them. The simulator we'll use is called VCS.



The simulator can be run in different ways that will produce different results. The first result is the compilation log file, which will be printed to the screen and report any compilation errors.

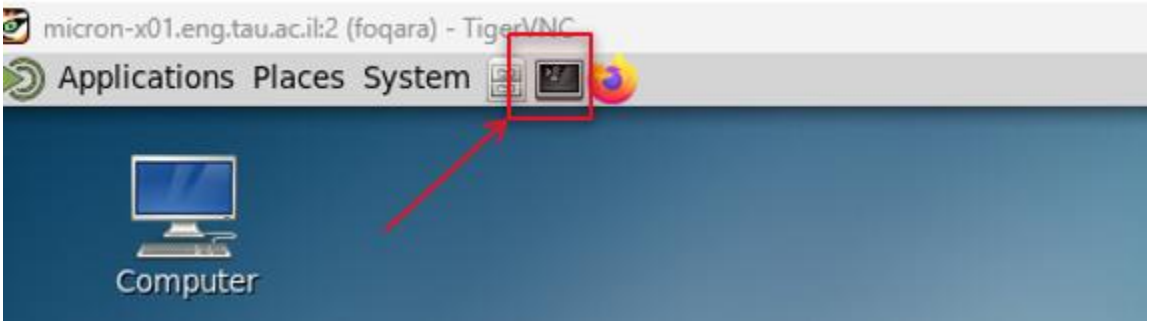
The second result is a simulation executable file, which can be run to execute the actual simulation (which will create the wave diagrams we've seen in recitation). However, if we just compile the design modules, there isn't any simulation going on. Recall that we need to instantiate the design module inside a testbench module, to 'give it life' by driving its inputs. So to produce the second result from the simulator, we need to make sure to compile the design files + the verification files. To make it easier, we usually have another file that lists all the files to compile, and this list file is given to the simulator.



Here, we assume that testbench.sv verification file instantiates the design modules described in design1.sv and design2.sv.

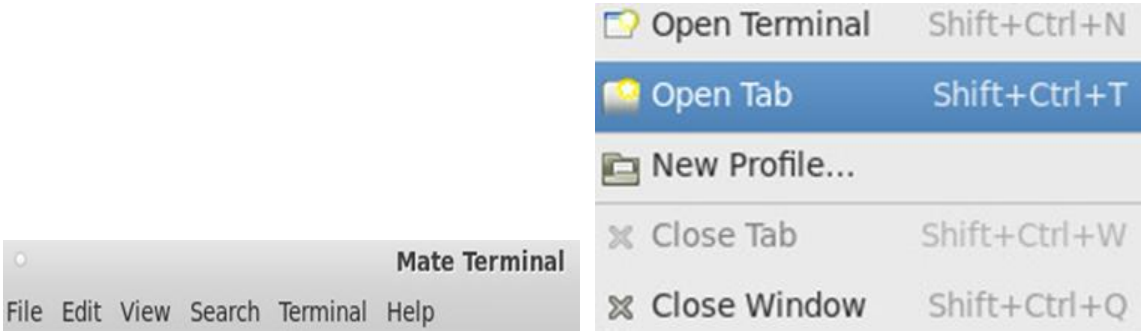
Setting up Our Work Environment

1. **After connecting to a Unix machine using PuTTY and TigerVNC, open a terminal by clicking the icon in the top-left corner of the screen.**



and if you have older version of the tiger you can open a terminal by following the menus, Applications à System Tools à MATE Terminal .

2. Select File → Open Tab, to create another tab of terminal .



3. **In the terminals console, type: advvlsi and press enter .**
   you will be moved to another directory and you will get the following message :

```
[foqara@micron-x01 ~]$ advvlsi
Project advvlsi setup is complete. Entering your workspace
[foqara@micron-x01 ws]$
```

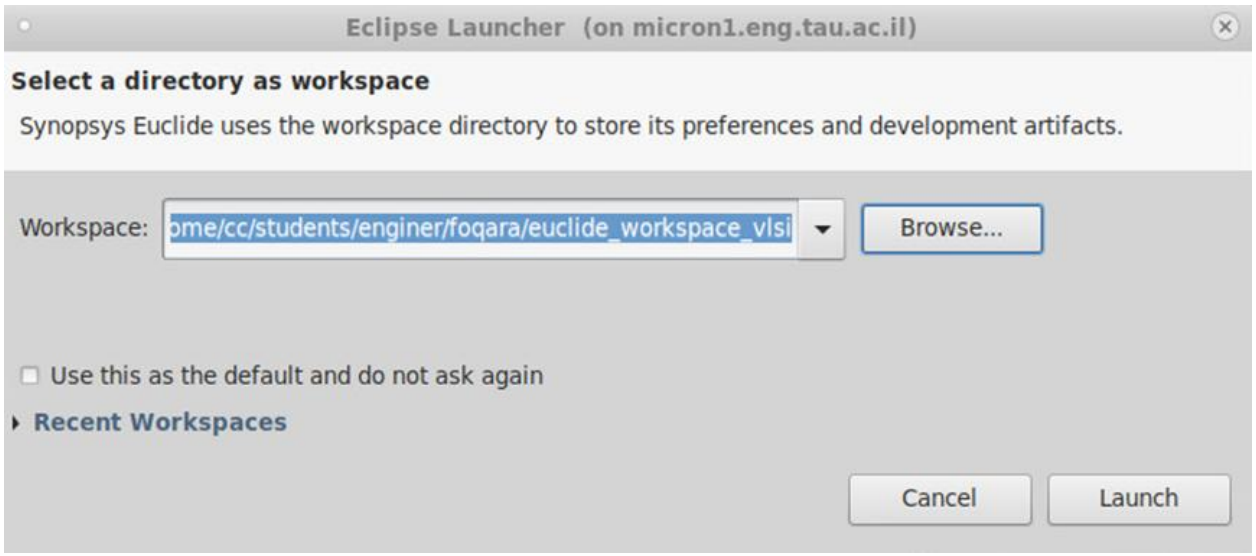
Note: Sometimes, the PUTTY connection will be lost, and when you re-connect, your terminal will be gone. In these cases, you need to repeat steps 1-3.

Make sure to download the file vlsi_setup.tar to your project directory.

This is a file archive (like zip). To unzip it, run this command: tar -xvf vlsi_setup.tar

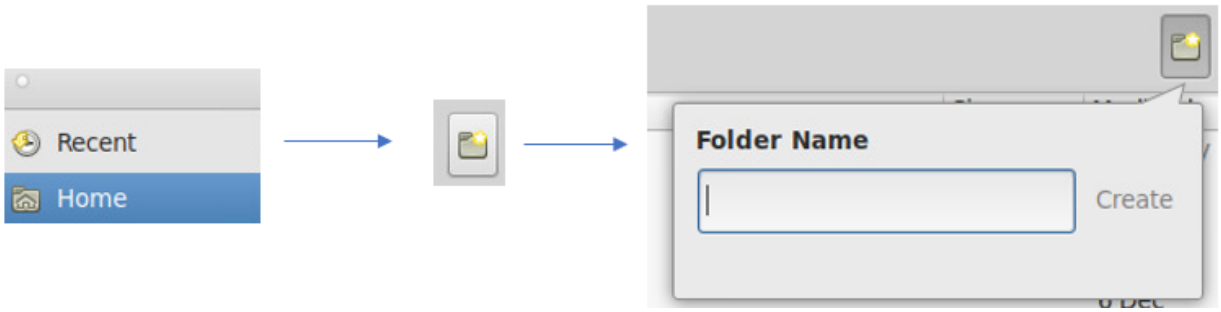
4. **In the terminal's console, type: euclide In order to launch the coding environment.**

5. **A launcher window will pop up:**



Select Browse, Home and then the Create Folder button, name it Euclide_worksapce_<the project name>.

Select Create and Ok.



Select Launch and wait for the main Euclide GUI to start.

Note: the following steps (6 – 9) need to be done only in the first time! The settings will remain fixed for the workspace that you defined in step 5, so even if PUTTY closes, you don't have to re-do them.

6. **We need to load some configurations into Euclide. To do that, go back to the second tab of the terminal.**

7. **In the second tab of the terminal, run advvlsi again.**

8. Before Run the command ./setup_euclide.csh, you should to see if the setup file is have the same path of files you work on in your project, to do that open the setup in your terminal :

```
foqara@micron-x01 RTL_Verification]$ ls
build.cud         create_project.csh  parameters.sv      Test.launch
clean.launch      Makefile            pvtech.lib         Verification_test
comp.launch       novas.conf          Run.launch         vlsi_setup.tar
coverage.launch   novas.rc            setup_euclide.csh  Waves.launch
foqara@micron-x01 RTL_Verification]$ vim setup_euclide.csh
```

press “a” to can you edit to the file

```
#!/bin/csh


set RED='\033[1;31m'
set NC='\033[0m'

if (`pwd` != "/project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification") then
   echo "You are in the wrong directory! Exiting"
   exit 1
endif

set euclide_workspace_base = "/data.cc/data/a/home/cc/students/enginer/foqara/Euclide worksapce example for project"
set euclide_launches_path = "$euclide_workspace_base/.metadata/.plugins/org.eclipse.debug.core/.launches"

mkdir $euclide_launches_path
if ($? != 0) then
  echo "${RED}Error${NC} Directory creation failed. The base path above probably doesn't exist - please check that you have followed step 5 in the exercise guide"
   exit 1
endif

cp ./*launch $euclide_launches_path
if ($? != 0) then
   echo "${RED}Error${NC} Copy operation failed. Please check that you have followed step 5 in the exercise guide"
   exit 1
endif

echo "Success : copied launch configurations into euclide workspace"
~
```

change the direction of the folder.

```
if (`pwd` != "/project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification") then
   echo "You are in the wrong directory! Exiting"
   exit 1
endif
```

Change the direction of the work space.

```
set euclide_workspace_base = "/data.cc/data/a/home/cc/students/enginer/foqara/Euclide worksapce example for project"
set euclide_launches_path = "$euclide_workspace_base/.metadata/.plugins/org.eclipse.debug.core/.launches"
```

and after finsh you will press “esc” and write “:wq” to save and exit from the file.

```
~
~
~
~
:wq
```

and now we will do the same thing to create_project.csh:

```
[foqara@micron-x01 RTL_Verification]$ vim setup_euclide.csh
[foqara@micron-x01 RTL_Verification]$ vim create_project.csh
```

Mate Terminal

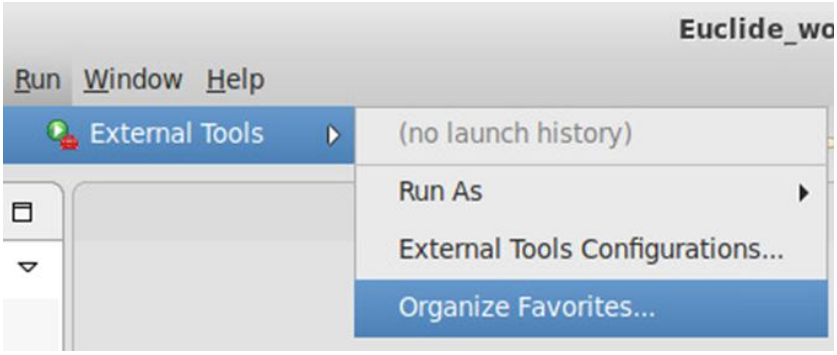
File Edit View Search Terminal Help

```
#!/bin/csh

set RED='\033[1;31m'
set NC='\033[0m'

if ($#argv != 1) then
  echo "Please use this script as follows: ./create_project.csh <single digit number>"
  exit 1
endif

if (`pwd` != "/project/advvlsi/users/$USER/ws") then
   echo "You are in the wrong directory! Exiting"
   exit 1
endif

set dirname = "ex_vlsi_$1";

if (! -f Makefile) then
   echo "You need to copy the Makefile into this directory"
   exit 1
endif

# Make a new directory inside X with a name from the command line
mkdir -p $dirname;
if ($? != 0) then
    echo "${RED}Error${NC} Directory creation failed"
    exit 1
endif

cp ./Makefile $dirname/
if ($? != 0) then
    echo "${RED}Error${NC} Copy operation failed. Are you in ws directory? make sure you are in ws directory where Makefile is accessible"
    exit 1
endif

echo "Success : created the following directory: $dirname, and copied the necessary Makefile into it!"
```

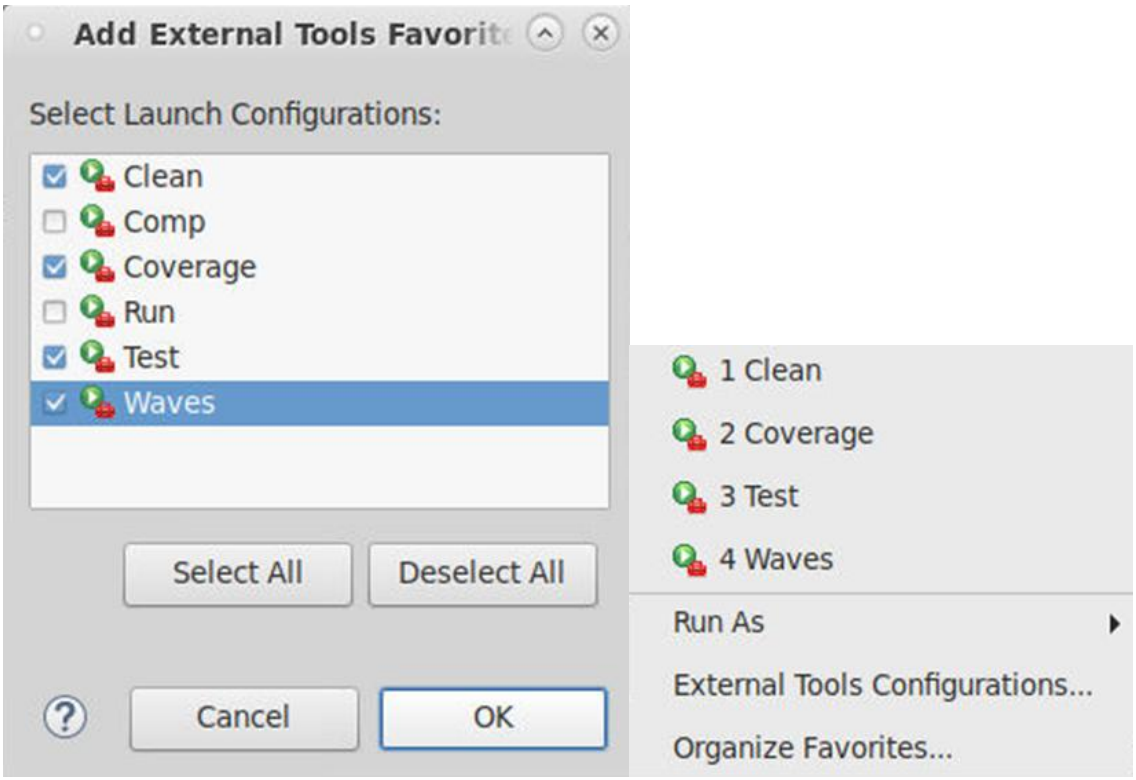
Run the command ./setup_euclide.csh.

9. **In the Euclide window, select File à Restart for the changes to take place.**

10. **We will use Euclide to launch simulations and not just to write code. To enable that, select Run à External Tools à Organize Favorites.**



11. Click Add, and select Clean, Test, Waves, Coverage Now, you will be able to launch any simulation operation from Euclide.



The operations are:

a. Clean: removing any temporary files or simulation files (not your code!) from the current directory.

b. Test: launch compilation and simulation test run .

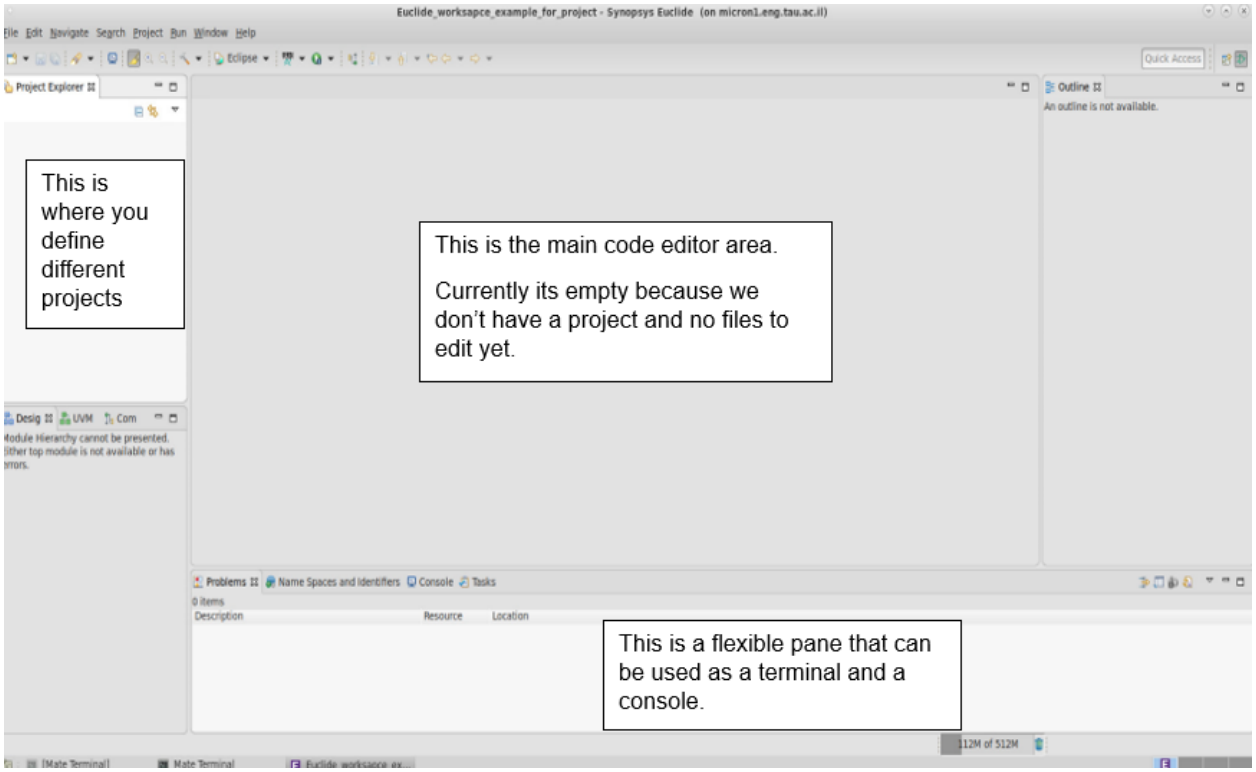
c. Waves: after simulation, launch waveform viewer to assist in debug.

d. Verification coverage attempts to answer the question: "How do you know you are finished verifying?", Verification coverage give us what events we want to see the design go through (input, output, states reached by the circuit...).

Setting up Project

1.

The main Euclide view looks like this.



In order to start writing design and/or verification code, we need to define a project. Each project can contain multiple files. One of the great advantages of Euclide is that all the files of the projects are compiled 'on-the-fly' while you edit them, so that errors and warnings are shown 'live'. We will now create a project and see this feature in action.

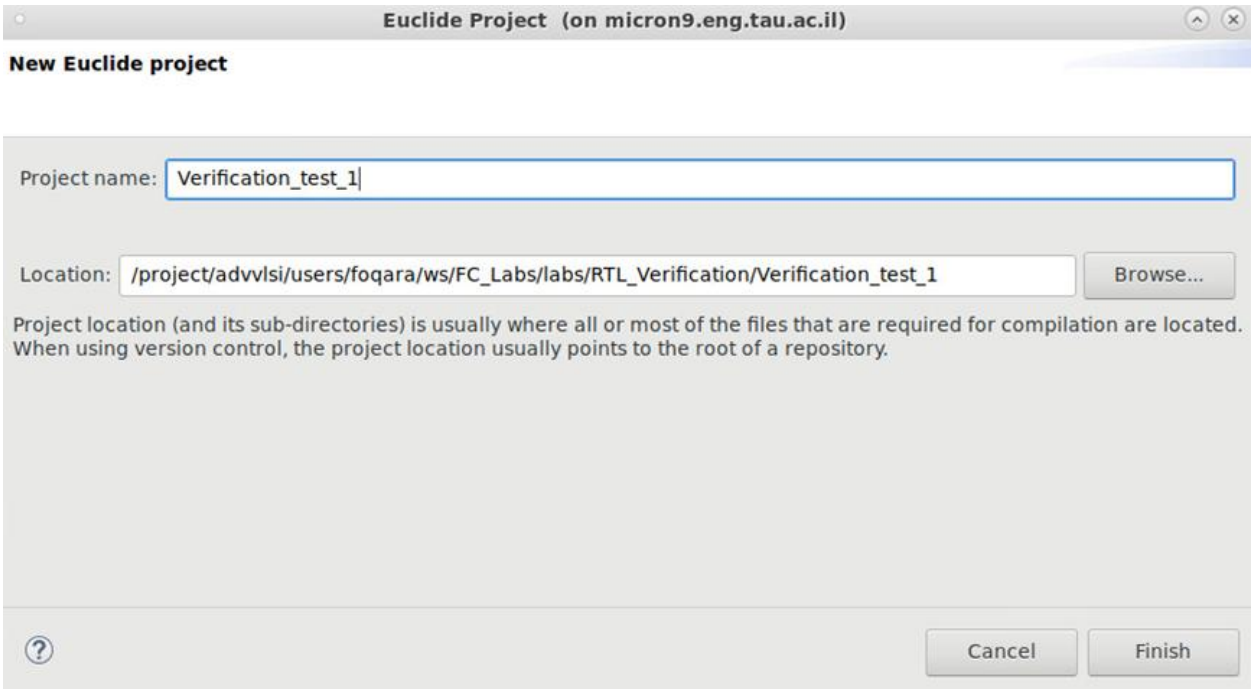
In MATE terminal, run the command: ./create_project.csh <x> Replace with the test number. For example, this first test can be created with: ./create_project.csh 1.

This step will create a new directory:/project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification/Verification_test_1 .

```
[foqara@micron-x01 RTL_Verification]$ ./create_project.csh 1
Success : created the following directory: Verification_test_1, and copied the necessary Makefile into it!
```

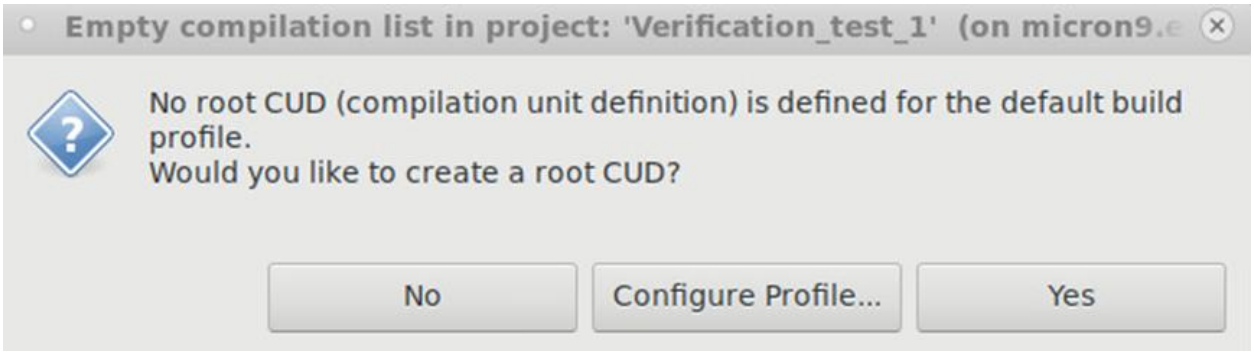
Return to Euclide and continue the steps below.

Select file→new→Euclide project and fill in the project name and location, as below:



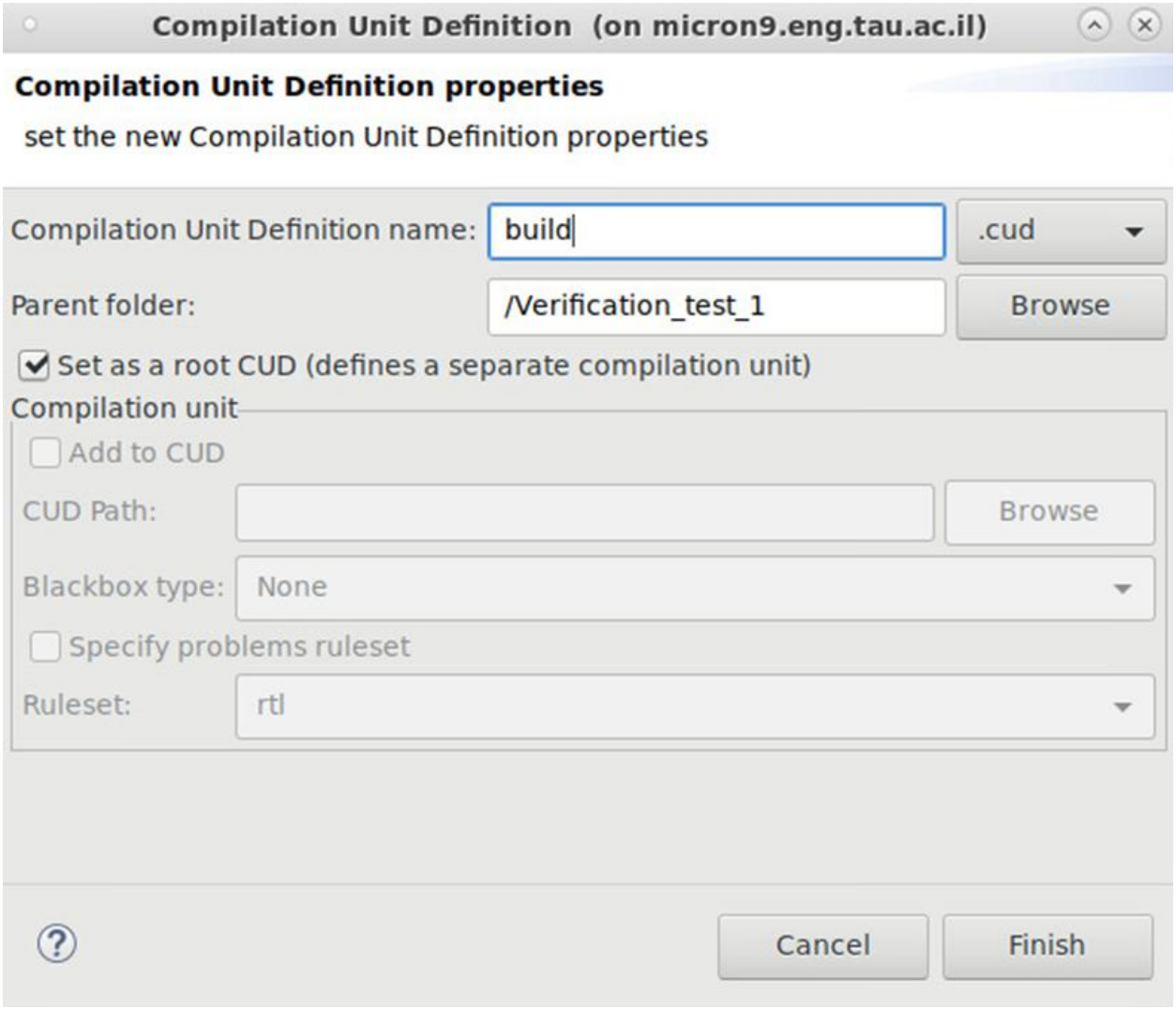
It's possible that the default location will be different, so change it if you need. The location should match the new directory that was created in step 2

The following prompt will pop up:

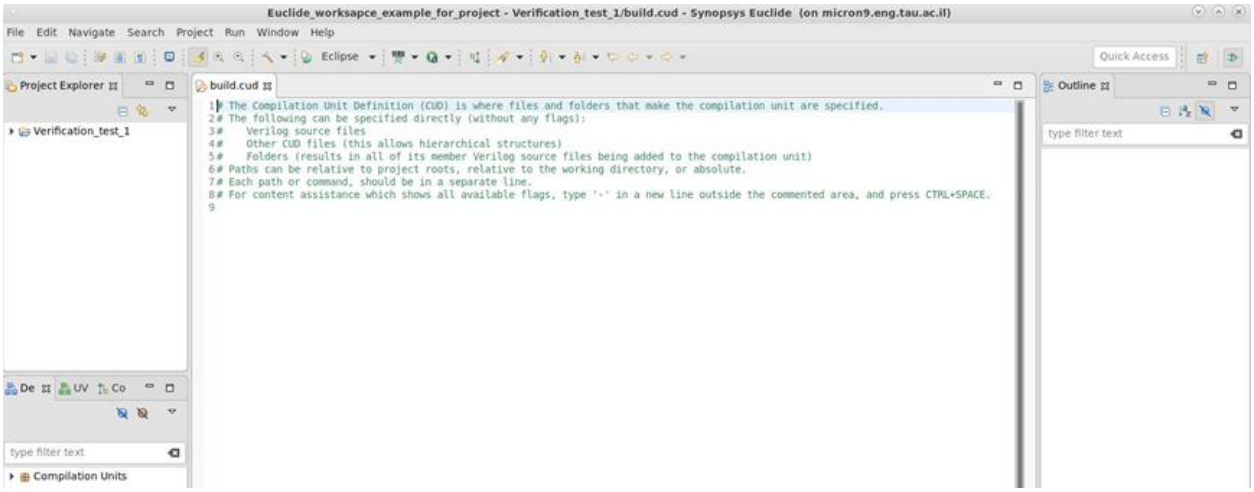


Euclide is asking us to create a CUD file. This is exactly the file listing the files we want to compile, which we described in the first section. Euclide will compile all the files listed there, 'on-the-fly', as you edit them. Select Yes.

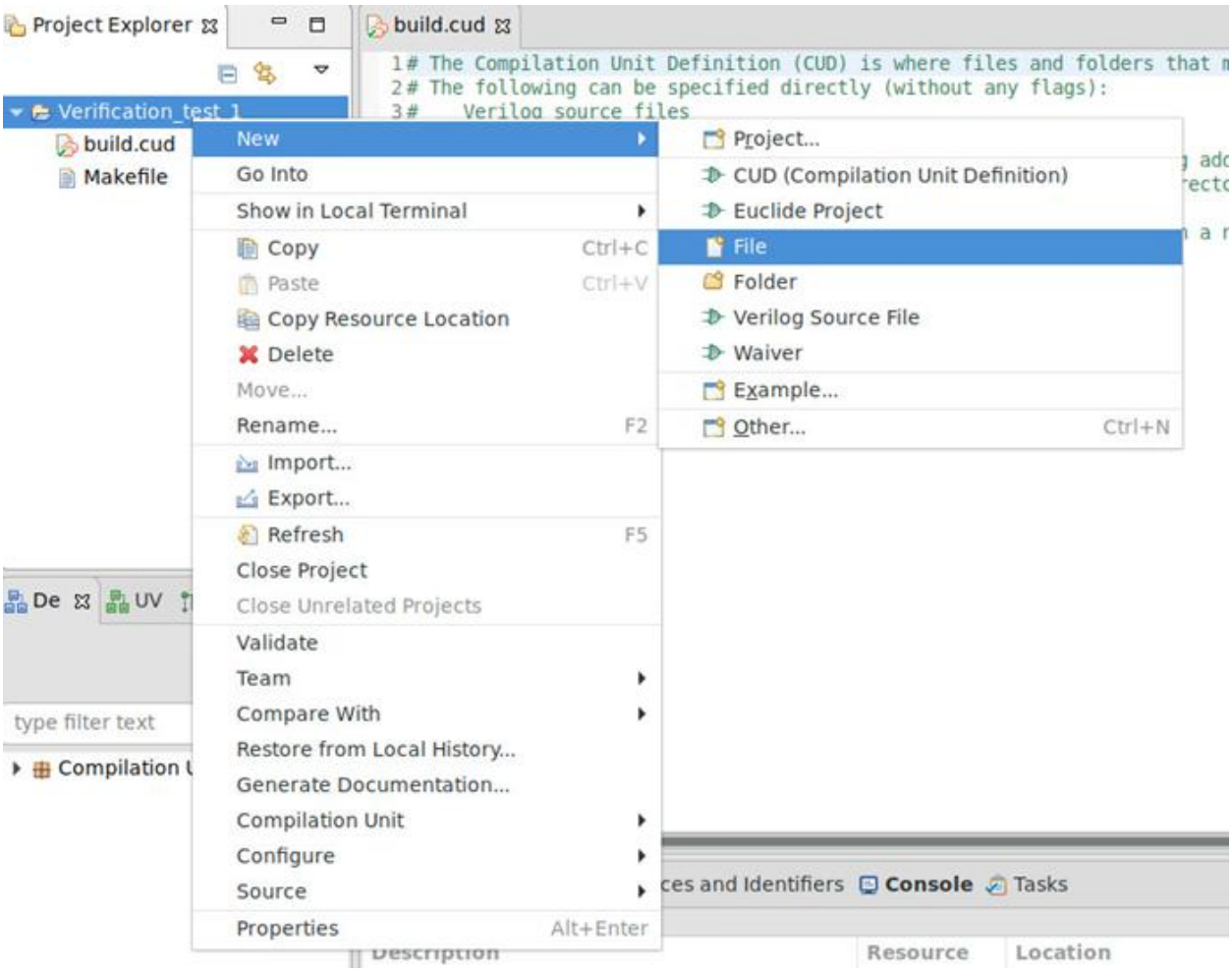
Name the file build.cud, make sure parent folder and as your exercise folder, and that Set as a root CUD is checked, as shown below, and select Finish

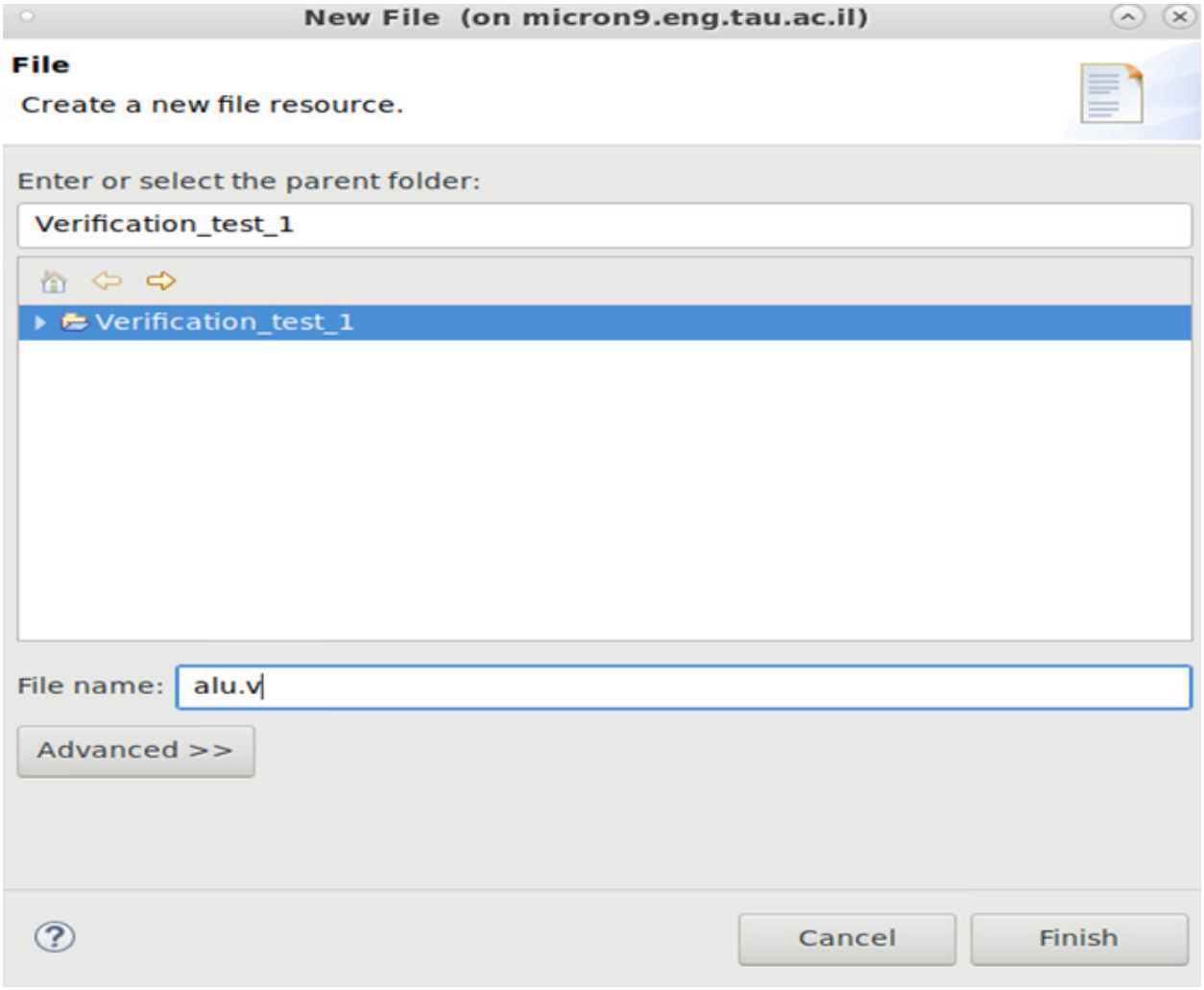


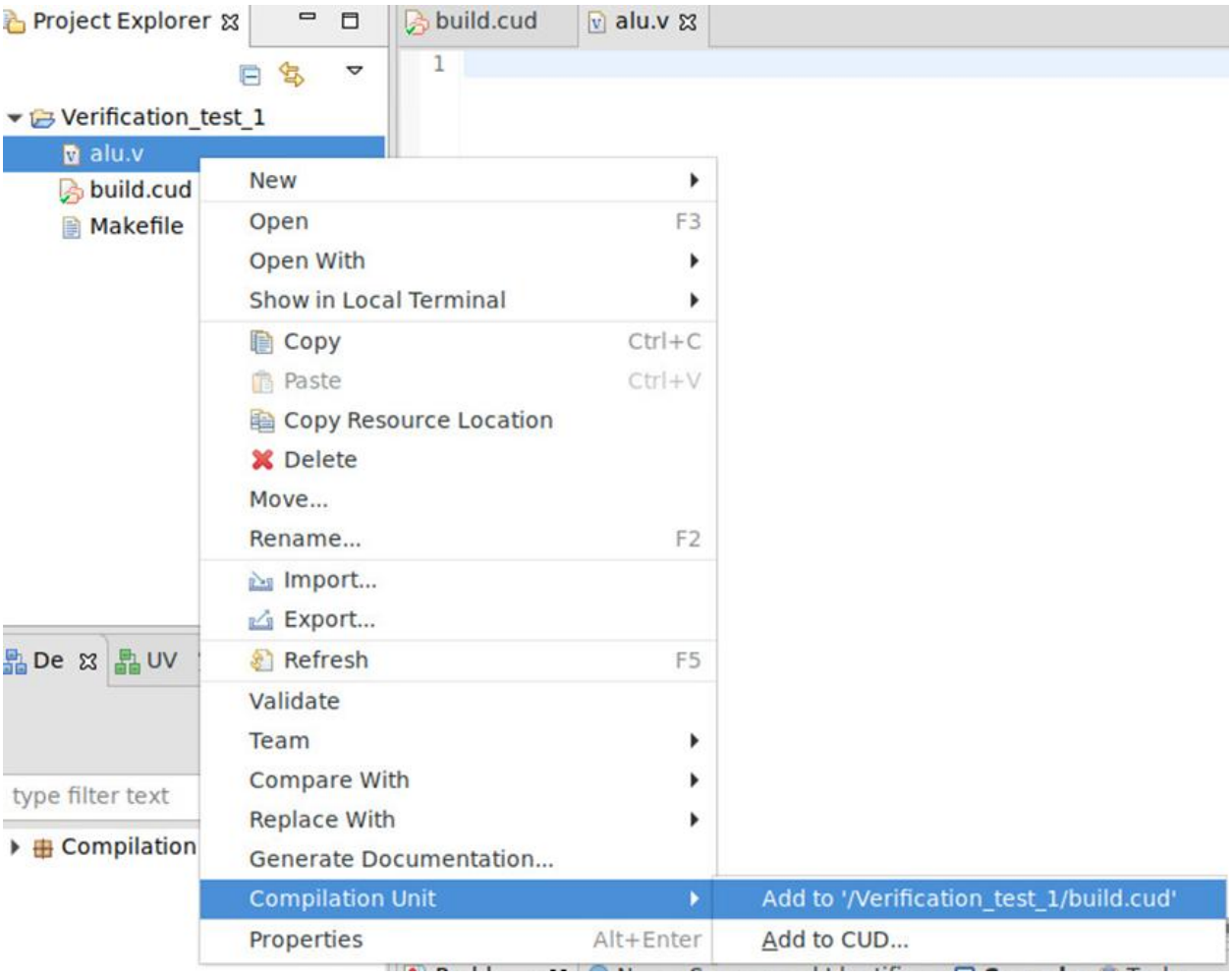
A new file will be created under your project, and will be opened in the main area of Euclide code editor

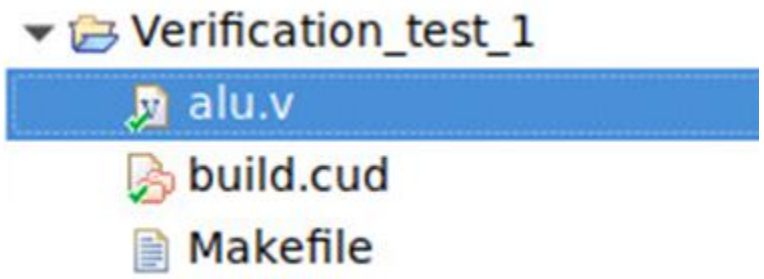


Create new file in your project by selecting Fileà New File. Make sure the file extension is .sv OR If a file is provided to you as a skeleton, copy it to the project directory (verifiction_test_x), and then under Project Explorer, right click and select Refresh In any case, under project explorer, right click the new file and select Compilation Unit

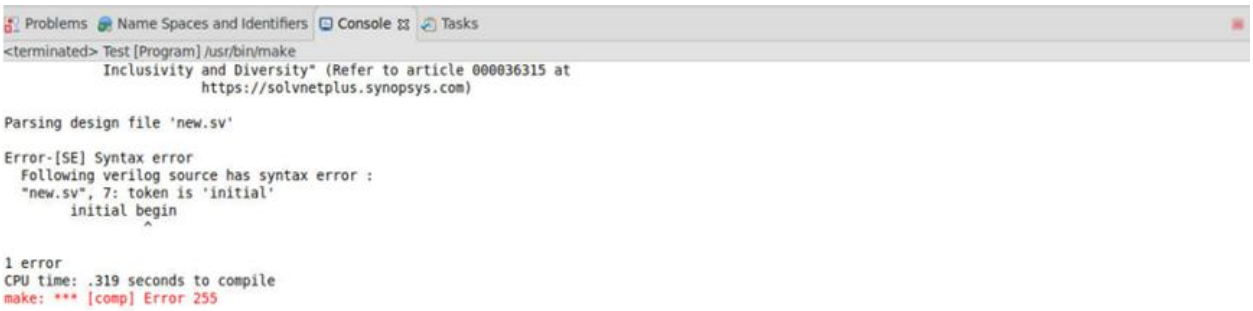



New File (on micron9.eng.tau.ac.il)
File
Create a new file resource.
Enter or select the parent folder:
Verification_test_1
Verification_test_1
File name:
alu.v
Advanced >>
Cancel
Finish



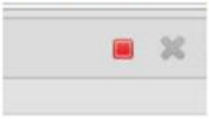


Then, select Run External Tools Test to launch compilation + simulation. In Euclide console you will see if there are any compilation or simulation errors:



Also, note the red stop button at the top right. If it is highlighted, the process is still ongoing.

- A common issue is a never-ending simulation. To prevent that, make sure your code contains a **$finish()** statement. If you still have a never-ending simulation (you will see that because the red stop button will always be highlighted), press that stop button.



You also have to manually remove any temp files in the terminal (files named **novas.***)

Eventually, a simulation that didn't fail should show this in the console:

```
V C S   S i m u l a t i o n   R e p o r t
Time: 100
CPU Time:      0.420 seconds;       Data structure size:   0.0Mb
Sat Mar  4 11:29:13 2023
```

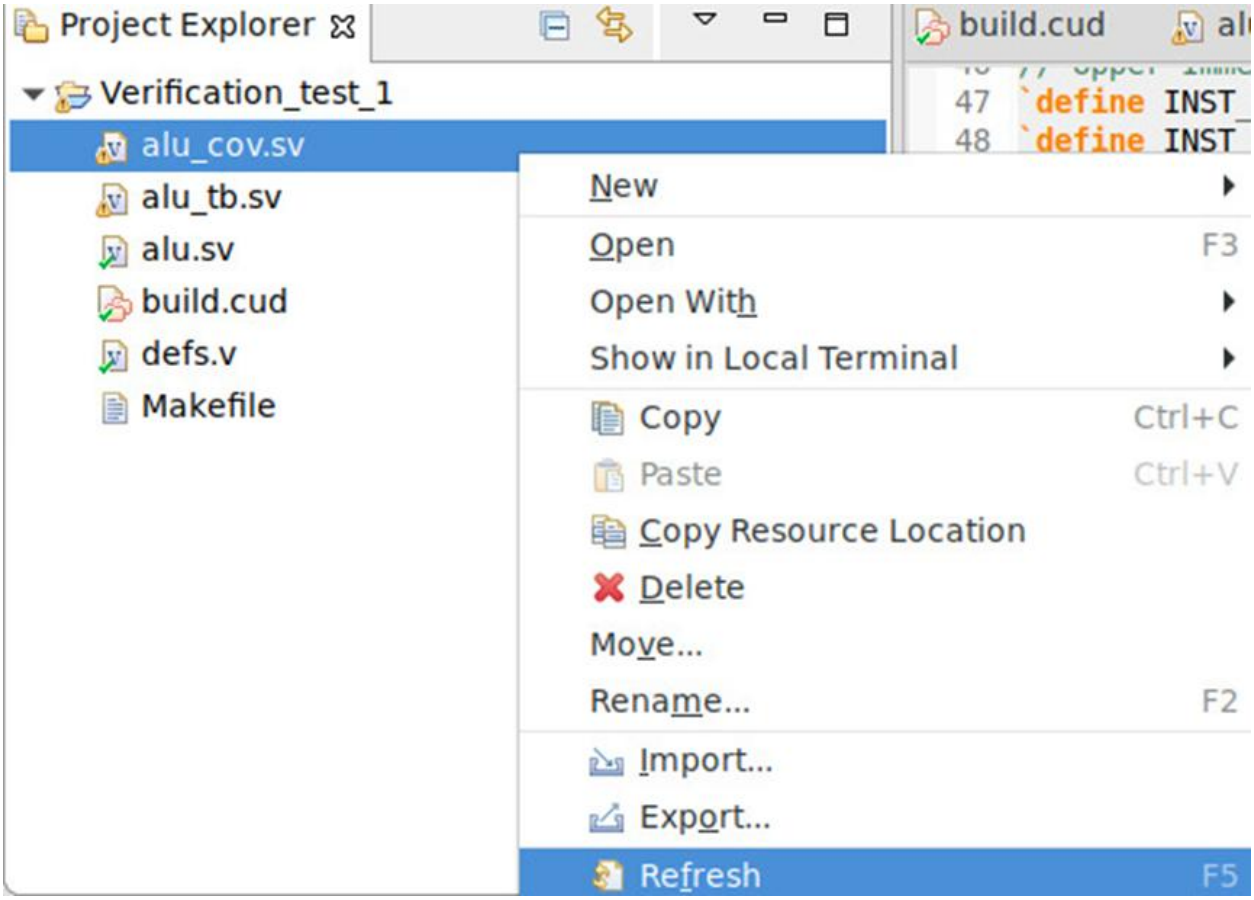
Now you are ready to use your Euclide

Example for verification

In this example we will do verification for ALU -32 bit, This ALU is a core component of a CPU, performing essential calculations and comparisons while providing status flags for control logic. It's designed to handle both signed and unsigned operations, making it versatile for RISC-V instructions.

****Make sure you have done the Euclide setup and it works properly**

You can upload your RTL file to test folder and do refresh in project explorer.



Or you can write your RTL file in Euclide like we explain in the setup.

Step 1: Understand the Design

Before starting verification, you need to thoroughly understand the ALU design:

1. **Review alu.sv: This is the RTL (Register Transfer Level) model of the ALU. It describes the behavior of the ALU in terms of its inputs, outputs, and internal logic.**

2. **Review des.v: This file contains the definitions of the ALU and its components. It includes details like the ALU's operations (e.g., ADD, SUB, AND, OR, XOR, etc.), bit-width (32-bit in this case), and any control signals.**

```
1  `timescale 1ns/1ps
2  //-----------------------------------------------
3  // Include Guards
4  //-----------------------------------------------
5  `ifndef RISCV_DEFS_V
6  `define RISCV_DEFS_V
7
8  //-----------------------------------------------
9  // ALU Operations
10 //-----------------------------------------------
11 `define ALU_NONE              4'b0000
12 `define ALU_SHIFTL            4'b0001
13 `define ALU_SHIFTR            4'b0010
14 `define ALU_SHIFTR_ARITH      4'b0011
15 `define ALU_ADD               4'b0100
16 `define ALU_SUB               4'b0110
17 `define ALU_AND               4'b0111
18 `define ALU_OR                4'b1000
19 `define ALU_XOR               4'b1001
20 `define ALU_LESS_THAN         4'b1010
21 `define ALU_LESS_THAN_SIGNED  4'b1011
22
23 //-----------------------------------------------
27 `define INST_ANDI              32'h7013
28 `define INST_ANDI_MASK         32'h707f
29 `define INST_ADDI              32'h13
30 `define INST_ADDI_MASK         32'h707f
31 `define INST_SLTI              32'h2013
32 `define INST_SLTI_MASK         32'h707f
33 `define INST_SLTIU             32'h3013
34 `define INST_SLTIU_MASK        32'h707f
35 `define INST_ORI               32'h6013
36 `define INST_ORI_MASK          32'h707f
37 `define INST_XORI              32'h4013
38 `define INST_XORI_MASK         32'h707f
39 `define INST_SLLI              32'h1013
40 `define INST_SLLI_MASK         32'hfc00707f
41 `define INST_SRLI              32'h5013
42 `define INST_SRLI_MASK         32'hfc00707f
43 `define INST_SRAI              32'h40005013
44 `define INST_SRAI_MASK         32'hfc00707f
45
46 // Upper Immediate Instructions
47 `define INST_LUI               32'h37
```

ALU-32 model

```
 5 `timescale 1ns/1ps
 6 module alu
 7 (
 8     // Inputs
 9     input  [3:0]  alu_op_i,
10     input  [31:0] alu_a_i,
11     input  [31:0] alu_b_i,
12
13     // Outputs
14     output [31:0] alu_p_o,
15     output        zero,
16     output        negative,
17     output        carry,
18     output        overflow
19 );
20
21 //-----------------------------------------------------------------
31 `ifndef RISCV_DEFS_V
207 // INCLUDE EXPANSION: End of expanded text
208
209 //-----------------------------------------------------------------
210 // Internal Signals
211 //-----------------------------------------------------------------
212 reg [31:0] result_r;
213
214 //-----------------------------------------------------------------
215 // ALU Logic
216 //-----------------------------------------------------------------
217 always @ (*)
242
243 // Status Flags
244 assign zero     = (result_r == 32'b0);
245 assign negative = result_r[31];
246 assign carry    = (alu_op_i == `ALU_ADD) ? (alu_a_i + alu_b_i < alu_a_i) :
247                   (alu_op_i == `ALU_SUB) ? (alu_a_i < alu_b_i) :
248                   1'b0;
249 assign overflow = (alu_op_i == `ALU_ADD) ? ((alu_a_i[31] == alu_b_i[31]) && (result_r[31] != alu_a_i[31])) :
250                   (alu_op_i == `ALU_SUB) ? ((alu_a_i[31] != alu_b_i[31]) && (result_r[31] != alu_a_i[31])) :
251                   1'b0;
252
253 // Output
254 assign alu_p_o = result_r;
255
256 endmodule
```

The ALU performs arithmetic, logical, and comparison operations on two 32-bit inputs (alu_a_i and alu_b_i) based on a 4-bit control signal (alu_op_i). It outputs the result (alu_p_o) and status flags (zero, negative, carry, overflow).

After you upload/write you RTL file do not forget to add it to “build.cud” .

And to use the defs.v in alu.sv or in any file you ca you use the instruction include :

`include "/project/advvlsi/users/<user>/ws/FC_Labs/labs/RTL_Verification/Verification_test_1/defs.v"

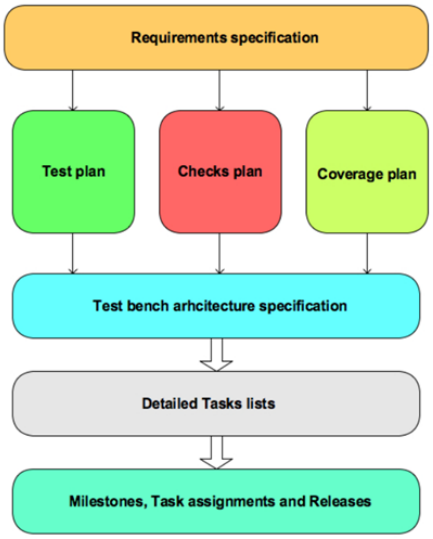
Step 2: Define the Verification Plan

The verification plan outlines what needs to be tested and how. It includes:

1. **Functional Verification: Ensure the ALU performs all operations correctly (e.g., ADD, SUB, AND, OR, XOR, etc.).**

2. **Corner Cases: Test boundary conditions, such as maximum and minimum 32-bit values, zero inputs, and overflow/underflow conditions.**

3. **Random Testing: Use constrained random testing to cover a wide range of input combinations.**



4. **Coverage Goals: Define what coverage metrics need to be achieved .**

Step 3: Develop the Testbench (alu_tb.sv)

The testbench is responsible for applying inputs to the ALU and checking the outputs. Here's how to structure it:

Instantiate the ALU: Instantiate the ALU module from alu.sv in the testbench.

```
// Inputs
logic [3:0]  alu_op_i;
logic [31:0] alu_a_i;
logic [31:0] alu_b_i;

// Outputs
logic [31:0] alu_p_o;

// Instantiate the ALU
   alu uut (
    .alu_op_i(alu_op_i),
    .alu_a_i(alu_a_i),
    .alu_b_i(alu_b_i),
    .alu_p_o(alu_p_o)
);
```

1. **Generate Test Vectors:**

   Create directed test cases for specific operations (e.g., ADD, SUB, etc.).

   Use constrained random testing to generate a wide range of inputs.

```
// Testbench variables
int pass_count = 0;
int fail_count = 0;
int num_random_tests = 100; // Number of random tests
logic [31:0] expected;
logic [31:0] rand_a = $urandom();
logic [31:0] rand_b = $urandom();
logic [3:0]  rand_op = $urandom_range(0, 15);

// Test cases
initial begin
    $display("Starting ALU Testbench...\n");

    // Arithmetic Operations
    test_alu_op(`ALU_ADD, 32'h00000005, 32'h00000003, 32'h00000008, "ADD");
    test_alu_op(`ALU_SUB, 32'h00000008, 32'h00000003, 32'h00000005, "SUB");

    // Logical Operations
    test_alu_op(`ALU_AND, 32'h0000FFFF, 32'hFFFF0000, 32'h00000000, "AND");
    test_alu_op(`ALU_OR,  32'h0000FFFF, 32'hFFFF0000, 32'hFFFFFFFF, "OR");
    test_alu_op(`ALU_XOR, 32'h0000FFFF, 32'hFFFF0000, 32'hFFFFFFFF, "XOR");

    // Shift Operations
    test_alu_op(`ALU_SHIFTL, 32'h00000001, 32'h00000004, 32'h00000010, "SLL");
    test_alu_op(`ALU_SHIFTR, 32'h80000000, 32'h00000001, 32'h40000000, "SRL");
    test_alu_op(`ALU_SHIFTR_ARITH, 32'h80000000, 32'h00000001, 32'hC0000000, "SRA");

    // Comparison Operations
    test_alu_op(`ALU_LESS_THAN, 32'h00000005, 32'h00000008, 32'h00000001, "SLT");
    test_alu_op(`ALU_LESS_THAN_SIGNED, 32'hFFFFFFFB, 32'h00000008, 32'h00000001, "SLT Signed");
    test_alu_op(`ALU_LESS_THAN, 32'hFFFFFFFB, 32'h00000008, 32'h00000000, "SLTU");

    // Edge Cases
    test_alu_op(`ALU_ADD, 32'hFFFFFFFF, 32'h00000001, 32'h00000000, "ADD Overflow");
    test_alu_op(`ALU_SUB, 32'h00000000, 32'h00000001, 32'hFFFFFFFF, "SUB Underflow");
    test_alu_op(`ALU_SHIFTL, 32'h00000001, 32'h00000020, 32'h00000000, "SLL Large Shift");
    test_alu_op(`ALU_SHIFTR, 32'h80000000, 32'h00000020, 32'h00000000, "SRL Large Shift");
    test_alu_op(`ALU_SHIFTR_ARITH, 32'h80000000, 32'h00000020, 32'hFFFFFFFF, "SRA Large Shift");
```

simple test

```
// Randomized Tests
$display("\nRunning Randomized Tests...");
repeat (num_random_tests) begin

     rand_a = $urandom();
     rand_b = $urandom();
     rand_op = $urandom_range(0, 15);

    // Skip invalid opcodes
    if (rand_op > `ALU_XOR) continue;

    // Calculate expected result

    case (rand_op)
        `ALU_ADD: expected = rand_a + rand_b;
        `ALU_SUB: expected = rand_a - rand_b;
        `ALU_AND: expected = rand_a & rand_b;
        `ALU_OR:  expected = rand_a | rand_b;
        `ALU_XOR: expected = rand_a ^ rand_b;
        `ALU_SHIFTL: expected = rand_a << rand_b[4:0];
        `ALU_SHIFTR: expected = rand_a >> rand_b[4:0];
        `ALU_SHIFTR_ARITH: expected = $signed(rand_a) >>> rand_b[4:0];
        `ALU_LESS_THAN: expected = (rand_a < rand_b) ? 32'h1 : 32'h0;
        `ALU_LESS_THAN_SIGNED: expected = ($signed(rand_a) < $signed(rand_b)) ? 32'h1 : 32'h0;
        default: expected = 32'h0;
    endcase
```

random test

2. Apply Inputs and Capture Outputs:

Apply the test vectors to the ALU inputs.

Capture the outputs and compare them with expected results.

```
            // Run test
            test_alu_op(rand_op, rand_a, rand_b, expected, "Random Test");
        end

        // Summary
        $display("\nTest Summary:");
        $display("Passed: %0d, Failed: %0d", pass_count, fail_count);
        if (fail_count == 0) $display("ALL TESTS PASSED!");
        else $display("SOME TESTS FAILED!");
        $finish;
    end

    // Task to test ALU operations
    task test_alu_op(
        input logic [3:0]  op,
        input logic [31:0] a,
        input logic [31:0] b,
        input logic [31:0] expected,
        input string op_name
    );
        begin
            alu_op_i = op;
            alu_a_i = a;
            alu_b_i = b;
            #10; // Wait for signals to propagate

            if (alu_p_o === expected) begin
                $display("[PASS] %s: A=0x%h, B=0x%h ? Result=0x%h (Expected=0x%h)",
                         op_name, a, b, alu_p_o, expected);
                pass_count++;
            end else begin
                $display("[FAIL] %s: A=0x%h, B=0x%h ? Result=0x%h (Expected=0x%h)",
                         op_name, a, b, alu_p_o, expected);
                fail_count++;
            end
        end
    endtask

endmodule
```

Summary of test

3. **Check Results:**

   Use assertions or conditional checks to verify that the outputs match the expected results.

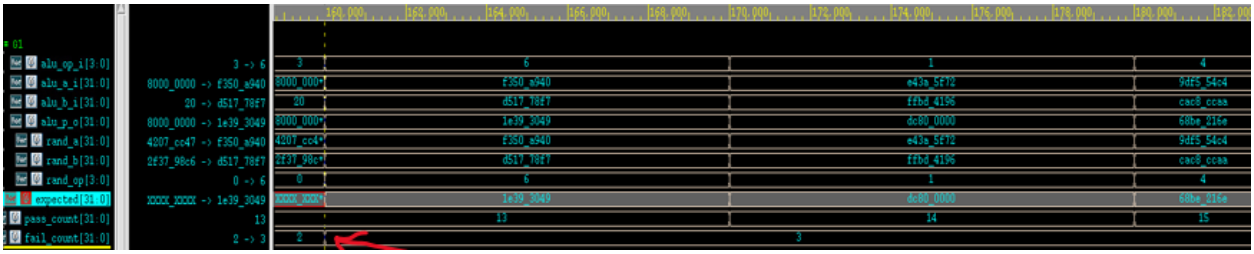
   Log any mismatches for debugging.

```
Running Randomized Tests...
[PASS] Random Test: A=0xf350a940, B=0xd51778f7 ? Result=0x1e393049 (Expected=0x1e393049)
[PASS] Random Test: A=0xe43a5f72, B=0xffbd4196 ? Result=0xdc800000 (Expected=0xdc800000)
[PASS] Random Test: A=0x9df554c4, B=0xcac8ccaa ? Result=0x68be216e (Expected=0x68be216e)
[PASS] Random Test: A=0x11ab26cf, B=0x38646d4f ? Result=0x4a0f941e (Expected=0x4a0f941e)
[PASS] Random Test: A=0x7337dd17, B=0x76264a88 ? Result=0x37dd1700 (Expected=0x37dd1700)
[PASS] Random Test: A=0x2da1e52c, B=0x94d34dce ? Result=0x00000000 (Expected=0x00000000)
[PASS] Random Test: A=0xbb91ab7a, B=0xb1487ba2 ? Result=0x00000000 (Expected=0x00000000)
[PASS] Random Test: A=0xf7fb42c9, B=0x7dc155dc ? Result=0x90000000 (Expected=0x90000000)
[PASS] Random Test: A=0x84285d3f, B=0x2890cc12 ? Result=0xacb92951 (Expected=0xacb92951)
[PASS] Random Test: A=0xa8be83df, B=0x53a24ec5 ? Result=0x00a202c5 (Expected=0x00a202c5)
[PASS] Random Test: A=0x92d2a694, B=0xa5568088 ? Result=0xd2a69400 (Expected=0xd2a69400)
[PASS] Random Test: A=0x867b8d31, B=0x26b88d29 ? Result=0x5fc30008 (Expected=0x5fc30008)
[PASS] Random Test: A=0xfc69c200, B=0xfe1f66dd ? Result=0x00000007 (Expected=0x00000007)
[PASS] Random Test: A=0xd3f72ea6, B=0x98bd5359 ? Result=0xfffffe9 (Expected=0xfffffe9)
[PASS] Random Test: A=0x52577345, B=0xb4a83bf2 ? Result=0x00001495 (Expected=0x00001495)
[PASS] Random Test: A=0x89552bc1, B=0xb079bb20 ? Result=0x80512b00 (Expected=0x80512b00)
[PASS] Random Test: A=0x4c054120, B=0xaf80c4e1 ? Result=0x0c004020 (Expected=0x0c004020)
[PASS] Random Test: A=0x2f835717, B=0x76391499 ? Result=0x59ba438e (Expected=0x59ba438e)
[PASS] Random Test: A=0x01325d8c, B=0x8fef25ec ? Result=0x714337a0 (Expected=0x714337a0)
[PASS] Random Test: A=0x39c7e4bb, B=0xa49501a5 ? Result=0x208500a1 (Expected=0x208500a1)
[PASS] Random Test: A=0x3d6ccdcb, B=0x5b732c32 ? Result=0x372c0000 (Expected=0x372c0000)
[PASS] Random Test: A=0x9f67d9ee, B=0x83fad457 ? Result=0xffffff3e (Expected=0xffffff3e)
[PASS] Random Test: A=0x83085fcd, B=0xdcdcf262 ? Result=0xdfdcffef (Expected=0xdfdcffef)
[PASS] Random Test: A=0x5d9d827b, B=0x0f1ec7e6 ? Result=0x67609ec0 (Expected=0x67609ec0)
[PASS] Random Test: A=0x33ae2b9c, B=0x0355a495 ? Result=0x03042094 (Expected=0x03042094)
[PASS] Random Test: A=0x59c1d8ca, B=0x6e113cec ? Result=0x00000000 (Expected=0x00000000)
[PASS] Random Test: A=0xdd822335, B=0x9c315b41 ? Result=0x6ec1119a (Expected=0x6ec1119a)
[PASS] Random Test: A=0x1715c44e, B=0xb8418c89 ? Result=0xbf55cccf (Expected=0xbf55cccf)
[PASS] Random Test: A=0x0bc95bd6, B=0x903f6fd5 ? Result=0x9bff7fd7 (Expected=0x9bff7fd7)
[PASS] Random Test: A=0xe9b6c90c, B=0xc5c81bde ? Result=0x2c7ed2d2 (Expected=0x2c7ed2d2)
[PASS] Random Test: A=0x81bbcd99, B=0x44ff2ff5 ? Result=0xfffffc0d (Expected=0xfffffc0d)
[PASS] Random Test: A=0x96afb9d4, B=0x61ad15b8 ? Result=0xf85ccf8c (Expected=0xf85ccf8c)
[PASS] Random Test: A=0x07af75b7, B=0x1a1ad3c3 ? Result=0x1db5a674 (Expected=0x1db5a674)
[PASS] Random Test: A=0x88ebfb7b, B=0x9b5a74a3 ? Result=0x475fdbd8 (Expected=0x475fdbd8)
[PASS] Random Test: A=0x4a3a6a2b, B=0x474fc8b4 ? Result=0x00000000 (Expected=0x00000000)
[PASS] Random Test: A=0x46ab8854, B=0x7d386222 ? Result=0x00000000 (Expected=0x00000000)
[PASS] Random Test: A=0x567708df, B=0x3eb0f117 ? Result=0x68c7f9c8 (Expected=0x68c7f9c8)
[PASS] Random Test: A=0x823d8ea9, B=0x4b6d537f ? Result=0xffffffff (Expected=0xffffffff)
[PASS] Random Test: A=0x2de75a1e, B=0xd1bd2812 ? Result=0xfdff7a1e (Expected=0xfdff7a1e)
[PASS] Random Test: A=0xe0585c71, B=0xa1625ee6 ? Result=0x3ef5fd8b (Expected=0x3ef5fd8b)
[PASS] Random Test: A=0xe619b489, B=0x6a7fbec6 ? Result=0x7b99f5c3 (Expected=0x7b99f5c3)
[PASS] Random Test: A=0xf7436520, B=0x7e3328a0 ? Result=0xf7436520 (Expected=0xf7436520)
```

With the simple test we can compare between the input and what it should to gave in the output.

```
Test Summary:
Passed: 76, Failed: 3
SOME TESTS FAILED!
$finish called from file "alu_tb.sv", line 108.
$finish at simulation time                790000
           V C S   S i m u l a t i o n   R e p o r t
Time: 790000 ps
CPU Time:      0.520 seconds;       Data structure size:   0.0Mb
Sat Feb  1 14:46:54 2025
#qrsh -V -cwd -b y -q sim_light ./simv +ntb_random_seed_automatic 2>&1 | tee log
```

and for the random test we can use wave called expected to compare direct :



Step 4: Define Coverage Points (alu_cov.sv)

Coverage ensures that all parts of the design are tested. Here's how to define coverage:

1. **Functional Coverage:**

   Cover all ALU operations (e.g., ADD, SUB, AND, OR, XOR, etc.).
   Cover input ranges (e.g., positive, negative, zero, and boundary values).
2. **Code Coverage:**

   Ensure all lines of code in alu.sv are executed during testing.
   Check for branches, conditions, and expressions.
3. **Assertion Coverage:**

   Use assertions to check specific behaviors (e.g., overflow detection).

```
covergroup alu_op_cg @(posedge alu_op_i);
    option.per_instance = 1;

    // Coverpoint for ALU operations
    coverpoint alu_op_i {
        bins add  = {`ALU_ADD};
        bins sub  = {`ALU_SUB};
        bins And  = {`ALU_AND};
        bins Or   = {`ALU_OR};
        bins Xor  = {`ALU_XOR};
        bins sll  = {`ALU_SHIFTL};
        bins srl  = {`ALU_SHIFTR};
        bins sra  = {`ALU_SHIFTR_ARITH};
        bins slt  = {`ALU_LESS_THAN};
        bins sltu = {`ALU_LESS_THAN_SIGNED};
    }

    // Cross coverage between ALU operations and operands
        cross alu_op_i, alu_a_i, alu_b_i {
        bins add_operands  = binsof(alu_op_i) intersect {`ALU_ADD};
        bins sub_operands  = binsof(alu_op_i) intersect {`ALU_SUB};
        bins and_operands  = binsof(alu_op_i) intersect {`ALU_AND};
        bins or_operands   = binsof(alu_op_i) intersect {`ALU_OR};
        bins xor_operands  = binsof(alu_op_i) intersect {`ALU_XOR};
        bins sll_operands  = binsof(alu_op_i) intersect {`ALU_SHIFTL};
        bins srl_operands  = binsof(alu_op_i) intersect {`ALU_SHIFTR};
        bins sra_operands  = binsof(alu_op_i) intersect {`ALU_SHIFTR_ARITH};
        bins slt_operands  = binsof(alu_op_i) intersect {`ALU_LESS_THAN};
        bins sltu_operands = binsof(alu_op_i) intersect {`ALU_LESS_THAN_SIGNED};
    }
endgroup
```

Step 5: Run Simulations

1. **Compile the Design and Testbench:**

   Use a simulator verdi to compile alu.sv, des.v, alu_cov.sv, and alu_tb.sv.
2. **Run the Simulation:**

   Execute the testbench and observe the outputs.
   Check for any mismatches or errors.
3. **Analyze Coverage:**

   Use the simulator's coverage tools to analyze functional, code, and assertion coverage.
   Identify any uncovered areas and add additional test cases if necessary.

Run this code in terminal :

Compile with Coverage Options:

Copy

```
vcs -sverilog -cm line+cond+fsm+tgl+branch -cm_dir ./coverage -full64 defs.v alu.sv alu_tb.sv alu_cov.sv
```

Run the Simulation:

Copy

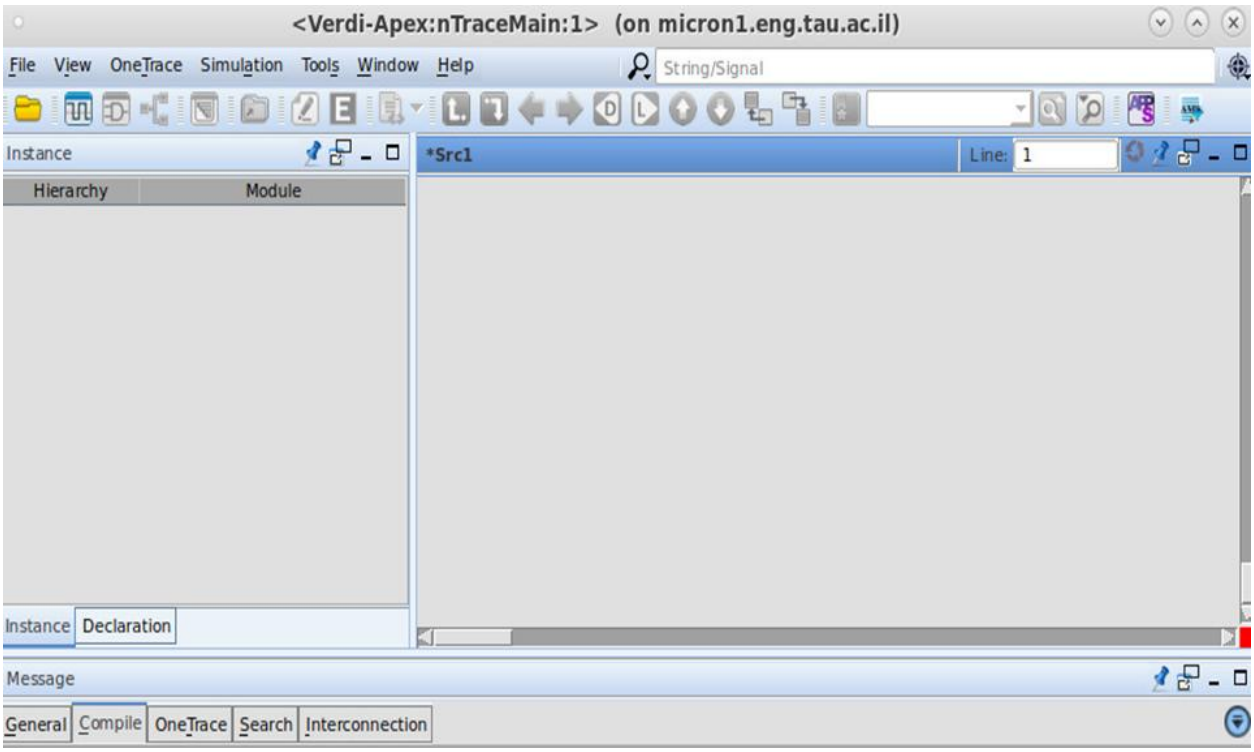
```
./simv -cm line+cond+fsm+tgl+branch
```

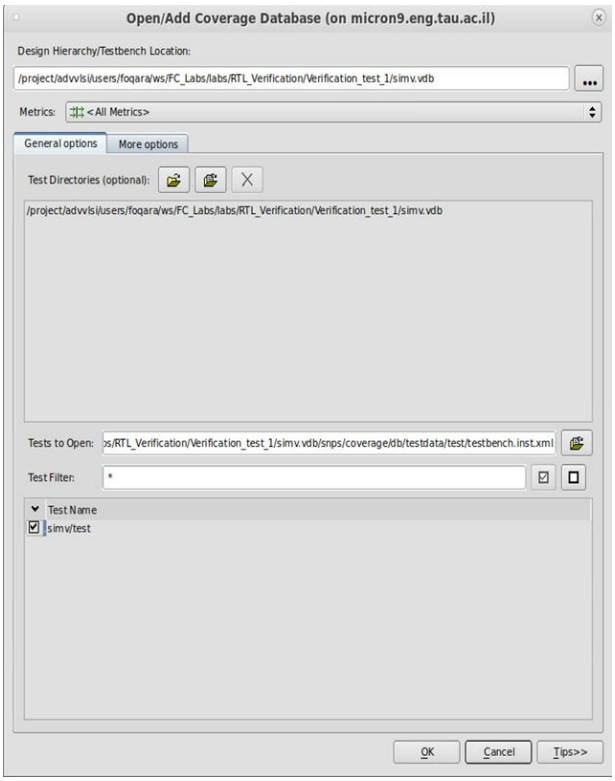
analyze Coverage in Verdi:

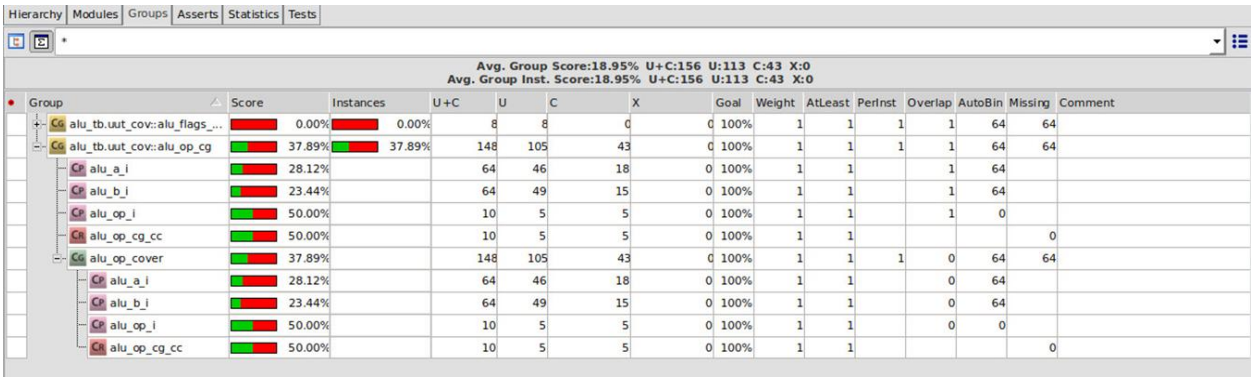
Copy

```
verdi -covdir ./coverage
```

```
Test Summary:
Passed: 76, Failed: 3
SOME TESTS FAILED!
$finish called from file "alu_tb.sv", line 108.
$finish at simulation time              790000

------------------------------------------------------------------------
VCS Coverage Metrics: during simulation line, cond, FSM, branch, tgl was monitored
------------------------------------------------------------------------

 Coverage status: End of All Coverages ...

          V C S   S i m u l a t i o n   R e p o r t
Time: 790000 ps
CPU Time:      0.520 seconds;       Data structure size:   0.0Mb
Sat Feb  1 15:24:24 2025
[foqara@micron-x01 Verification_test_1]$ verdi -covdir ./coverage
logDir = /project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification/Verification_test_1/verdiLog


                                Verdi (R)

              Version T-2022.06-SP1 for linux64 - Aug 28, 2022

                   Copyright (c) 1999 - 2022 Synopsys, Inc.
   This software and the associated documentation are proprietary to Synopsys,
 Inc. This software may only be used in accordance with the terms and conditions
 of a written license agreement with Synopsys, Inc. All other use, reproduction,
   or distribution of this software is strictly prohibited.  Licensed Products
     communicate with Synopsys servers for the purpose of providing software
    updates, detecting software piracy and verifying that customers are using
    Licensed Products in conformity with the applicable License Key for such
  Licensed Products. Synopsys will use information gathered in connection with
    this process to deliver software updates and pursue software pirates and
                                infringers.
```


<Verdi-Apex:nTraceMain:1> (on micron1.eng.tau.ac.il)
File View OneTrace Simulation Tools Window Help
String/Signal
Instance
Hierarchy
Module
*Src1
Line: 1
Instance
Declaration
Message
General
Compile
OneTrace
Search
Interconnection


Open/Add Coverage Database (on micron9.eng.tau.ac.il)
Design Hierarchy/Testbench Location:
/project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification/Verification_test_1/simv.vdb
Metrics:
< All Metrics>
General options
More options
Test Directories (optional):
/project/advvlsi/users/foqara/ws/FC_Labs/labs/RTL_Verification/Verification_test_1/simv.vdb
Tests to Open:
s/RTL_Verification/Verification_test_1/simv.vdb/snps/coverage/db/testdata/test/testbench.inst.xml
Test Filter:
*
Test Name
simv/test
OK
Cancel
Tips>>



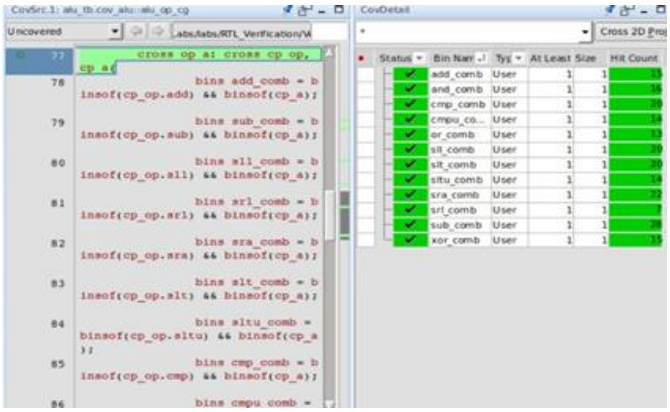
Group	Score	Instances	U+C	U	C	X	Goal	Weight	AtLeast	PerInst	Overlap	AutoBin	Missing	Comment
alu_tb.uut_cov::alu_flags_...	0.00%	0.00%	8	8	0	0	100%	1	1	1	1	64	64	
alu_tb.uut_cov::alu_op_cg	37.89%	37.89%	148	105	43	0	100%	1	1	1	1	64	64	
alu_a_i	28.12%		64	46	18	0	100%	1	1		1	64		
alu_b_i	23.44%		64	49	15	0	100%	1	1		1	64		
alu_op_i	50.00%		10	5	5	0	100%	1	1		1	0		
alu_op_cg_cc	50.00%		10	5	5	0	100%	1	1				0	
alu_op_cover	37.89%		148	105	43	0	100%	1	1	1	0	64	64	
alu_a_i	28.12%		64	46	18	0	100%	1	1		0	64		
alu_b_i	23.44%		64	49	15	0	100%	1	1		0	64		
alu_op_i	50.00%		10	5	5	0	100%	1	1		0	0		
alu_op_cg_cc	50.00%		10	5	5	0	100%	1	1				0	

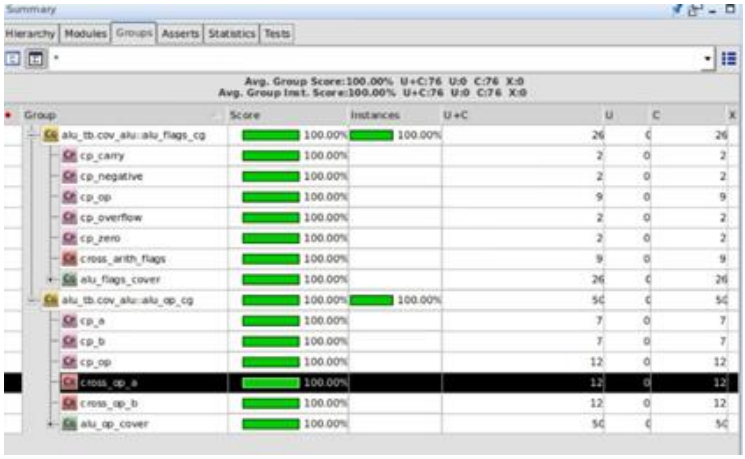
We can see that the coverage is too low, that mean the verification is not good because is not cover up to 95%, so to solve that we should to add more test .

So now we will up the random test to 1000, and constrained randomization to ensure a wider range of input values, and we get :

```
rand_a = $urandom_range(0, 32'hFFFFFFFF);
rand_b = $urandom_range(0, 32'hFFFFFFFF);
rand_op = $urandom_range(`ALU_ADD, `ALU_LESS_THAN_SIGNED);
```

Group	Score	Definition	U+C	U	C	X	Goal	Weight	AtLeast	PerInst	Overlap	AutoBin	Missing	Comment
Definitions	37.50%		156	18	138	0								
alu_tb	37.50%		156	18	138	0								
uut_cov	37.50%		156	18	138	0								
Instances	37.50%		156	18	138	0								
global_group_item	37.50%		156	18	138	0								
alu_flags_cover	0.00%	0.00%	8	8	0	0	100%	1	1	1	0	64	64	
alu_op_cover	75.00%	75.00%	148	10	138	0	100%	1	1	1	0	64	64	
alu_a_i	100.00%	100.00%	64	0	64	0	100%	1	1		0	64		
alu_b_i	100.00%	100.00%	64	0	64	0	100%	1	1		0	64		
alu_op_i	50.00%	50.00%	10	5	5	0	100%	1	1		0	0		
alu_op_cg_cc	50.00%	50.00%	10	5	5	0	100%	1	1				0	





continue to constrained randomization to get the coverage you want in your project .

Basic test + random test + constrains + define coverage points + cross coverage + flags + cross flags

=

~100% + good structure for integrations

Step 6: Debug and Iterate

1. **Debug Failures:**

   If any test cases fail, debug the issue by analyzing the waveforms and logs.
   Fix any bugs in the RTL or testbench.
2. **Iterate:**

   Re-run the simulation and coverage analysis until all coverage goals are met.

Step 7: Final Sign-Off

1. **Review Coverage Reports:**

   Ensure all functional, code, and assertion coverage goals are met.
2. **Document Results:**

   Document the verification process, test cases, coverage results, and any bugs found and fixed.
3. **Sign-Off:**

   Once all tests pass and coverage goals are achieved, sign off on the verification.

NOW you have the knowledge to do any verification simulation to any RTL file.

Introduction to RISC-V and Its Integration with SYNOPSYS Tools

For our project, we have chosen to leverage the RISC-V architecture to implement a full flow RTL-to-GDSII process targeting TSMC 28nm technology. This decision is rooted in RISC-V's distinct advantages, including its open-source nature, scalability, and compatibility with modern design workflows. Combined with the powerful tools provided by SYNOPSYS, RISC-V enables an efficient, cost-effective, and customizable approach to semiconductor design.

Overview of RISC-V

RISC-V is a groundbreaking open standard Instruction Set Architecture (ISA) designed for flexibility and modularity. Unlike proprietary ISAs, RISC-V is free and open-source, removing licensing restrictions and costs. Its broad configurability supports a range of applications, from low-power IoT devices to high-performance computing systems.

Key Features of RISC-V

1. Open-Source Customization:
   RISC-V allows developers to tailor the ISA to their specific requirements, enabling optimizations for energy efficiency, performance, or task-specific functionality.
   Example: Adding custom instructions for accelerating encryption or AI computations.

2. Modular and Scalable Architecture:
   The ISA supports various extensions, such as vector processing for AI and machine learning or floating-point operations for scientific workloads. This modularity ensures compatibility with simple microcontrollers and complex processors alike.

3. Robust Ecosystem:
   A rapidly growing ecosystem, supported by industry leaders and academic research, provides access to open-source tools, libraries, and frameworks such as GCC, LLVM, and QEMU. Commercial tools, like those from SYNOPSYS, enhance the development pipeline with advanced design and verification capabilities.

Advantages of RISC-V in Advanced Technology Nodes

- Cost-Effectiveness: Free from licensing fees, RISC-V reduces the financial burden of proprietary architectures, making it ideal for projects with tight budgets.
- Future-Proof: As an open standard, RISC-V’s roadmap is community-driven, ensuring long-term relevance and adaptability.

- Customizable Innovation: Developers can add proprietary extensions or tweak processor pipelines to optimize for specific workloads or industries.

Integrating RISC-V with SYNOPSYS Tools

SYNOPSYS, a leader in EDA solutions, provides a comprehensive suite of tools designed to streamline and enhance RISC-V-based development. Key benefits of this integration include:

1. End-to-End Design Flow:
   Tools like SYNOPSYS Design Compiler enable seamless translation from high-level descriptions to gate-level implementation. Coupled with Verification Suite tools, these ensure accuracy and robustness throughout the design process.

2. Custom Processor Development:
   Using SYNOPSYS Processor Designer, teams can create specialized RISC-V cores tailored to their unique requirements. This includes incorporating custom ISA extensions or optimizing for power, performance, and area (PPA).

3. Pre-Verified IP Integration:
   SYNOPSYS provides a rich library of RISC-V IP cores, pre-verified for integration. This saves significant development time and ensures compatibility with other design blocks.

4. Advanced Verification:
   Tools like Verdi and VCS simplify debugging, coverage analysis, and testbench generation. Combined with SYNOPSYS ZeBu for emulation, these tools accelerate the verification process and ensure design correctness under real-world conditions.

Why RISC-V and SYNOPSYS?

The combination of RISC-V's open-source flexibility with SYNOPSYS' cutting-edge tools offers unparalleled advantages for modern semiconductor design. It enables teams to:

- Develop domain-specific processors optimized for power, performance, and cost.
- Expedite time-to-market by leveraging SYNOPSYS' streamlined workflows and pre-verified IP.
- Future-proof designs by relying on an open standard and robust ecosystem support.

By integrating RISC-V into our SYNOPSYS-based RTL-to-GDSII flow, we are paving the way for a highly efficient, customizable, and cost-effective solution, tailored to meet the demands of TSMC 28nm technology.

To read more:

https://www.synopsys.com/glossary/what-is-risc-v.html

Technology File Syntax Overview (VLSI, TSMC 28LP)

In VLSI design, the Technology File is a critical component that provides process-specific parameters essential for physical design tools. For Synopsys tools, this file typically has a `.tf` extension and contains detailed information about the manufacturing process, guiding the design such as layer properties, routing rules and implementation phases.

Key Elements of the Synopsys Technology File:

1. **Technology Information:**

   Name and Date: Specifies the technology node (e.g., "65nm") and the file's creation or update date.

   Dielectric Constant: Indicates the dielectric constant value for the technology.

   Units and Precision: Defines units for time, length, voltage, capacitance, and their respective precision levels.

2. **Color Definitions:**

   Assigns RGB values to different layers for visualization purposes in layout tools.

3. **Tile Definitions:**

   Defines unit tiles used in site rows to guide the placement engine for cell placement.

   Specifies dimensions such as width and height for these tiles.

4. **Layer Details:**

   Provides comprehensive information about each layer, including:

   - **Layer Number and Name:** Identifiers for each layer.
   - **Mask Name:** Associated mask for the layer.
   - **Visual Attributes:** Color, pattern, and visibility settings.
   - **Physical Parameters:** Pitch, minimum width, default width, minimum length, and spacing rules.
   - **Routing Rules:** Guidelines for routing, including preferred directions and spacing constraints.

5. **Via Definitions:**

   Details for vias, including:

   - **Layer Associations:** Cut layer, lower layer, and upper layer.
   - **Dimensions:** Cut width, cut height, and enclosure dimensions.
   - **Resistance Values:** Minimum, nominal, and maximum resistance per unit.

6. **Design Rules:**

   Specifies rules for:

   - **Layer Spacing:** Minimum spacing requirements between features on the same layer.
   - **Via Spacing:** Rules governing the placement and spacing of vias.
   - **Antenna Effects:** Definitions and rules to mitigate antenna effects during manufacturing.

Syntax Structure:

The Technology File is organized into sections, each encapsulated within curly braces `{}`. Attributes within these sections are defined using the syntax: `attribute_name = attribute_value;`. Comments can be added using the `#` symbol.

For detailed reference, open the link: https://spdocs.synopsys.com/dow_retrieve/qsc-w/dg/icc2olh/W-2024.09/icc2olh/icctf/technology_file_syntax/technology_file_syntax.html

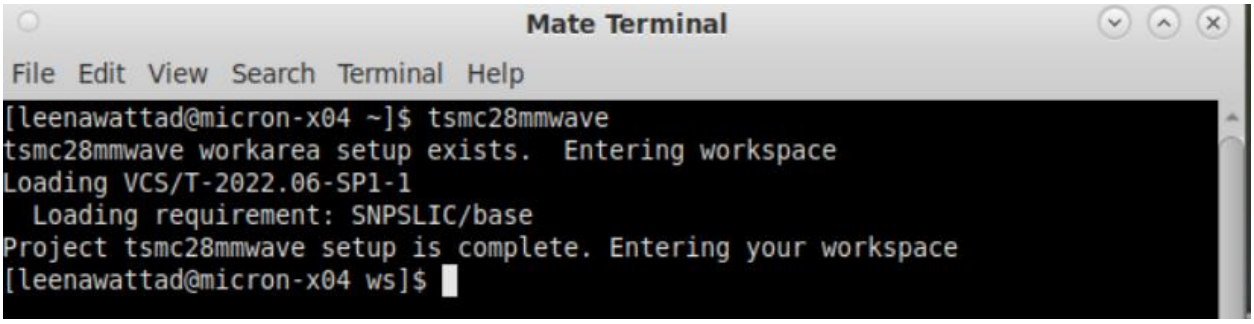
Environment Setup:

To successfully run our scripts, it's essential to organize the environment tree and working directories according to the guidelines provided in the chapters. This setup is required for accessing tools such as Fusion Compiler. Below are detailed and illustrative images showing how to open the working environment:

1. Initializing the Environment:

   - When you in the micron terminal write this command "tsmc28mmwave" which sets up the working environment for the project.

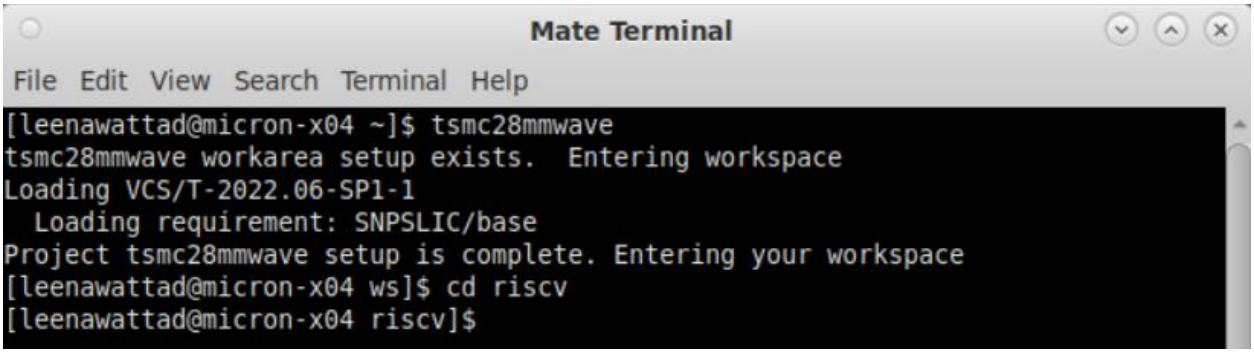
   - The terminal output confirms the environment is loaded successfully and the required tools (e.g., VCS and SNPSLIC) are initialized:



2. Navigating to the Project Directory:

   - After setting up the environment, by writing "cd file_name" we navigate into the specific project folder (e.g., `riscv`) that contains the design files, scripts, and configurations.

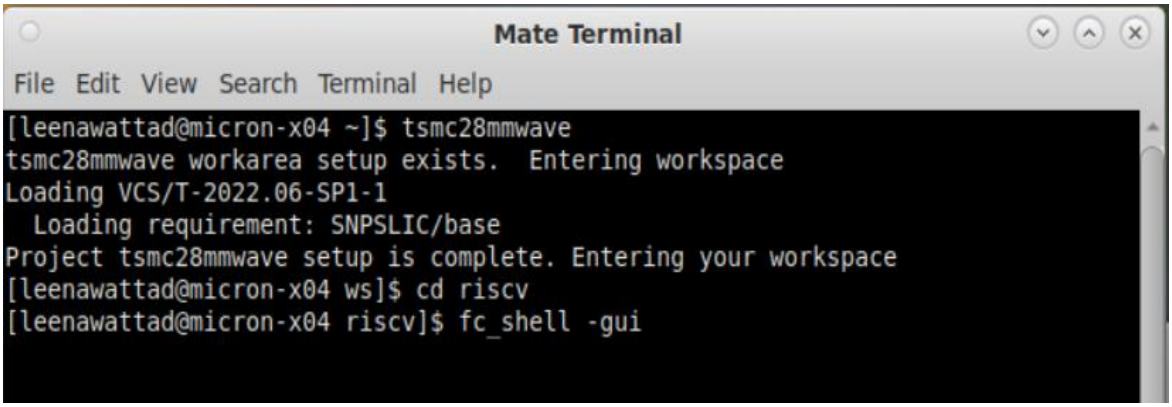
   - This directory structure ensures clean project organization.



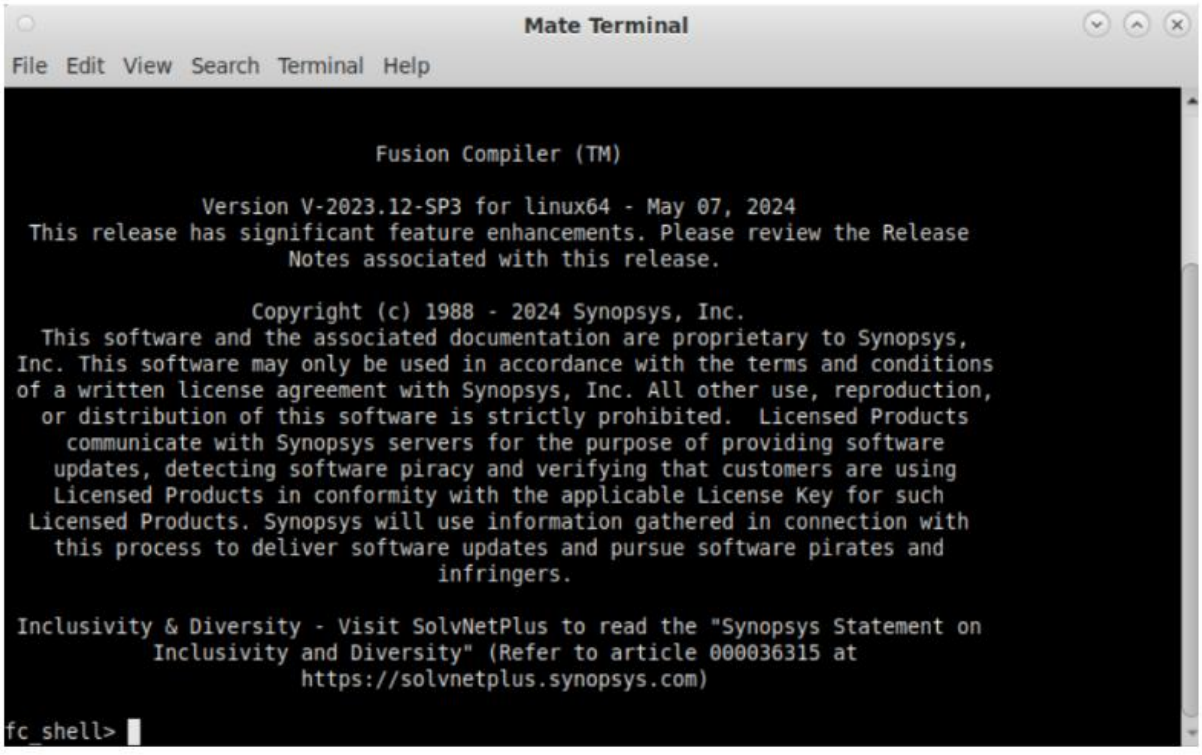
3. Launching Fusion Compiler in GUI Mode:

   - Then you write fc_shell -gui, this command starts Fusion Compiler in Graphical User Interface mode.

   - The GUI provides an intuitive way to write and run scripts, visualize the design, and manage project data.

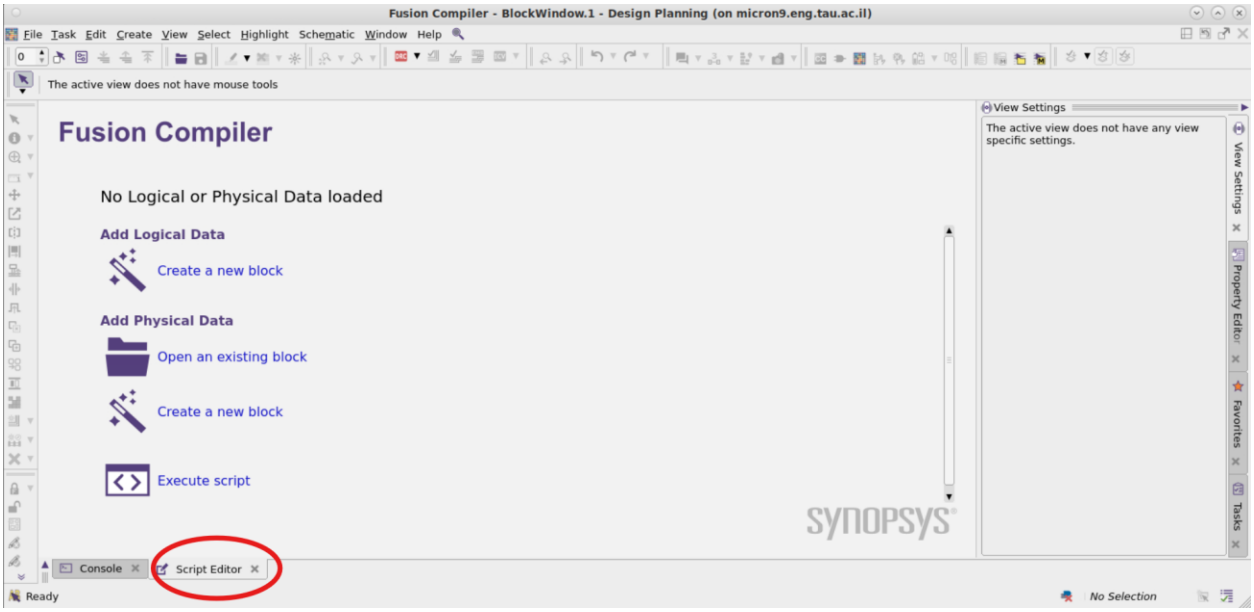


This is the standard welcome screen of Fusion Compiler:

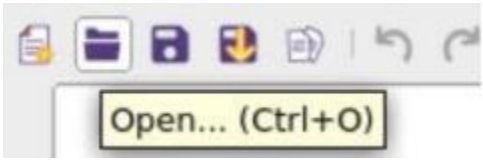


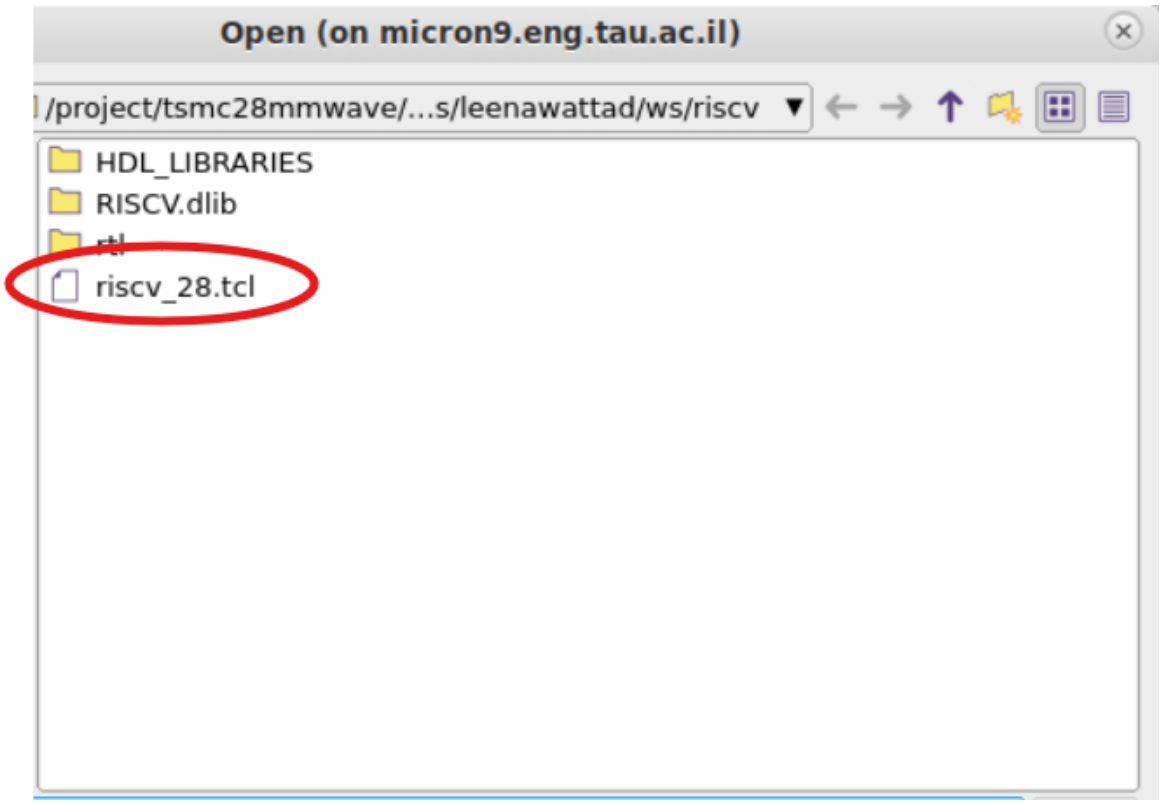
4. Fusion Compiler Startup Terminal Window:

   - While you are in Fusion Compiler program, look for the script editor (bottom panel near the console).



Open the file_name.tcl (the file that you would work on, here riscv_28.tcl) by using:





Press the script editor once again at the bottom panel. Enable the “Selected” button so you could mark commands in the file and run it using the “play” button one by one:

(If it’s in grey and unclickable, double click the first line of the code, to make it colored in blue and try again).

Building libraries:

Building libraries is a foundational step for enabling an effective RTL-to-GDS flow. Libraries serve as repositories of standard cells, technology data, and constraints, ensuring that synthesis, place-and-route, and timing analysis are performed accurately. This guide outlines the key components and processes required to create and configure libraries in Synopsys, based on industry best practices and academic principles.

Before building libraries, certain prerequisites must be met to ensure a seamless flow. Access to Process Design Kits (PDKs) provided by TSMC is essential, as they include SPICE models, layer definitions, and DRC/LVS rules for the 28nm node. Additionally, the Synopsys toolchain, including Fusion Compiler, Formality RC, and PrimeTime, must be installed and configured. Supporting resources, such as Tcl scripts for automation and configuration files tailored for specific tools, are also necessary to streamline the library creation process and maintain consistency across the design flow.

Opening a Library in Synopsys

Opening a library in Synopsys involves configuring the tools to recognize the library files and access them for the required tasks. Libraries are accessed using paths defined in setup files and loaded into tools such as Fusion Compiler, or PrimeTime. For large projects, automation scripts (in Tcl) are used to dynamically load and configure libraries, ensuring consistency and minimizing manual errors.

Storage and Management of Libraries

Libraries are centrally stored and managed to ensure consistency, ease of access, and version control across teams and tools.

1. **Library Storage**: Libraries are typically stored in shared network repositories or directories, organized by project or technology node (e.g., TSMC 28nm).

   The repository includes: Liberty files (.lib) for timing and power data, LEF files (.lef) for layout abstraction, GDSII files (.gds) for detailed layout representation, Technology files (.tf) defining process-specific rules, Milkyway databases for physical implementation.

2. **Version Control**: Libraries are managed using version control systems like Git or SVN to track changes and maintain consistency across design iterations.
3. **Accessing Libraries**: Synopsys tools access libraries via environment variables or explicit paths defined in setup files. For example:

“lappend search_path scripts design_data

. /tools/synopsys/fc/V-2023.12-SP3/dw/syn_ver /tools/synopsys/fc/V-2023.12-SP3/dw/sim_ver /tools/synopsys/fc/V-2023.12-SP3/libraries/syn scripts design_data"

This centralized organization ensures that all team members and tools reference the same version of the libraries, minimizing discrepancies.

Types of Libraries

1. Standard Cell Libraries

Standard cell libraries form the foundation of digital circuit design. They include a set of pre-designed logic cells that perform basic operations, such as logic gates and sequential elements. These cells are optimized for the target technology node (e.g., TSMC 28nm) and are used across all stages of the design process, from synthesis to physical layout.

- **Components**:

  Combinational Cells: Logic gates like AND, OR, NAND, NOR, and XOR.

  Sequential Cells: Flip-flops and latches used for data storage and synchronization.

  Specialized Cells: Buffers, multiplexers, and decoders.

- **Voltage Threshold Variants**:

  HVT (High Voltage Threshold): Designed with a higher threshold voltage to minimize leakage power. Slower switching speeds make them suitable for non-critical paths where timing constraints are relaxed. The are commonly used in power-saving modes or standby circuits.

  SVT (Standard Voltage Threshold): A balanced option offering moderate speed and power consumption. They are a default choice for general-purpose logic paths.

  LVT (Low Voltage Threshold):

  Designed for high-speed operation with a lower threshold voltage. Higher leakage power makes them ideal for timing-critical paths where performance is crucial.

Each cell in the library is characterized under multiple conditions, such as varying process, voltage, and temperature (PVT) corners, and is provided in Liberty (.lib) format for timing, power, and area data.

2. Register Libraries

Register libraries consist of sequential elements that store and manage state information within a digital design. These libraries are essential for implementing data storage, control logic, and clock synchronization.

- **Components**:

  Flip-Flops: Single-bit storage elements that capture data on a clock edge, Variants include D-flip-flops with enable and reset functionality.

  Latches: Level-sensitive storage elements used for asynchronous control logic.

  Clock-Gating Cells: Reduce dynamic power by selectively disabling the clock signal in idle sections of the design.

- **Features**: Optimized for low-power operation with variations in voltage thresholds (HVT, SVT, LVT). Integrated clock-tree design considerations for efficient timing distribution.

Register libraries are characterized for setup and hold times, power consumption, and switching speed, ensuring accurate performance modeling.

3. I/O Libraries

Input/Output (I/O) libraries manage the interaction between the chip and its external environment. They include cells designed to handle signal transmission, power delivery, and electrostatic protection.

- **Components**:

  Pad Cells: Handle data input and output signals, and support high-speed communication and low-power signaling standards.

  Power and Ground Pads: Distribute power and provide grounding for the chip and include cells for power-on-reset and voltage regulators.

  ESD Protection Cells: Protect the chip from electrostatic discharge damage during manufacturing and operation.

  Clock Pads: Designed for clock signals, supporting high-frequency operations.

- **Design Considerations**: Must meet signal integrity, noise, and voltage tolerance requirements. Furthermore, include impedance matching and shielding for high-speed signals.

These libraries are provided in LEF and GDSII formats for physical layout and Liberty format for timing analysis.

4. IP Libraries

Intellectual Property (IP) libraries consist of pre-designed functional blocks that can be integrated into a larger design. IP blocks are categorized based on their flexibility and intended use.

- **Types**:

  Hard IP: has a fixed physical layout, optimized for specific process nodes, and is silicon-proven for direct integration but lacks customization flexibility.

  Soft IP: is provided as synthesizable RTL code, offering high flexibility for customization but requires synthesis and additional verification.

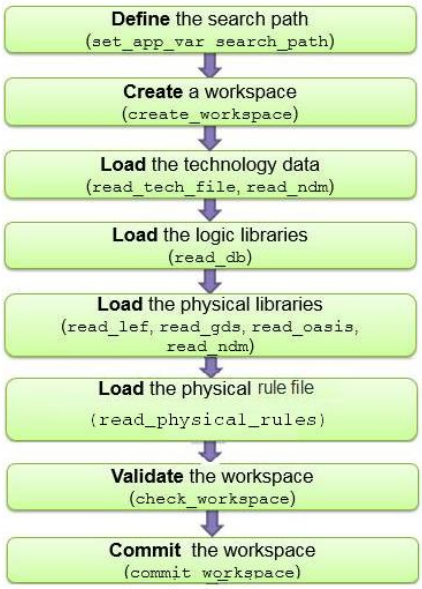
  Firm IP: is partially implemented with predefined physical constraints, balancing customization options and reduced design effort.

- **Features**: Verified and characterized by IP vendors or foundries, include timing, power, and layout information. Also, often come with integration guidelines and testbenches for simulation.

IP libraries save significant design time and effort by providing pre-verified components that integrate seamlessly with the custom design.

→ Together, these libraries form the backbone of any VLSI design flow, ensuring functionality, performance, and manufacturability.

Steps to build libraries:



To build a technology library, you follow the same flow, but omit steps four through eight, which involve the source files for the library cells.

After a cell library is built, it does not need to be rebuilt unless one of the library source files changes or a new version of the implementation tool requires an updated library.

The library manager uses a search path to look for files that are specified with a relative path or no path.

To specify the search path, set the search_path application variable to the list of directories, in order, in which to look for files. When the library manager looks for a file, it starts searching in the leftmost directory specified in the search_path variable and uses the first matching file it finds, for example:

```
lappend search_path scripts design_data
. /tools/synopsys/fc/V-2023.12-SP3/dw/syn_ver /tools/synopsys/fc/V-2023.12-SP3/dw/sim_ver /tools/synopsys/fc/V-2023.12-SP3/libraries/syn scripts design_data
```

To create a library workspace, use the create_workspace command. When you run this command to create a new cell library, you must specify the following items:

- The name of the workspace: For all flows except the edit flow, specify the workspace name without a file extension. For the edit flow, specify the path to an existing cell library.
- The library preparation flow you are using.

The create_workspace command fails if there are any cell libraries in memory. Before running the create_workspace command, use the close_lib command to close any open libraries.

An example:

```
#################################################################
create_lib  -technology $TECH_FILE  -ref_libs {/data/tsmc/28HPCPMMWAVE/synopsys/libs/tcbn28hpcplusbwp30p140.ndm\
 /data/tsmc/28HPCPMMWAVE/synopsys/libs/tcbn28hpcplusbwp30p140hvt.ndm /data/tsmc/28HPCPMMWAVE/synopsys/libs/tcbn28hpcplusbwp30p140lvt.ndm }  RISCV.dlib
```

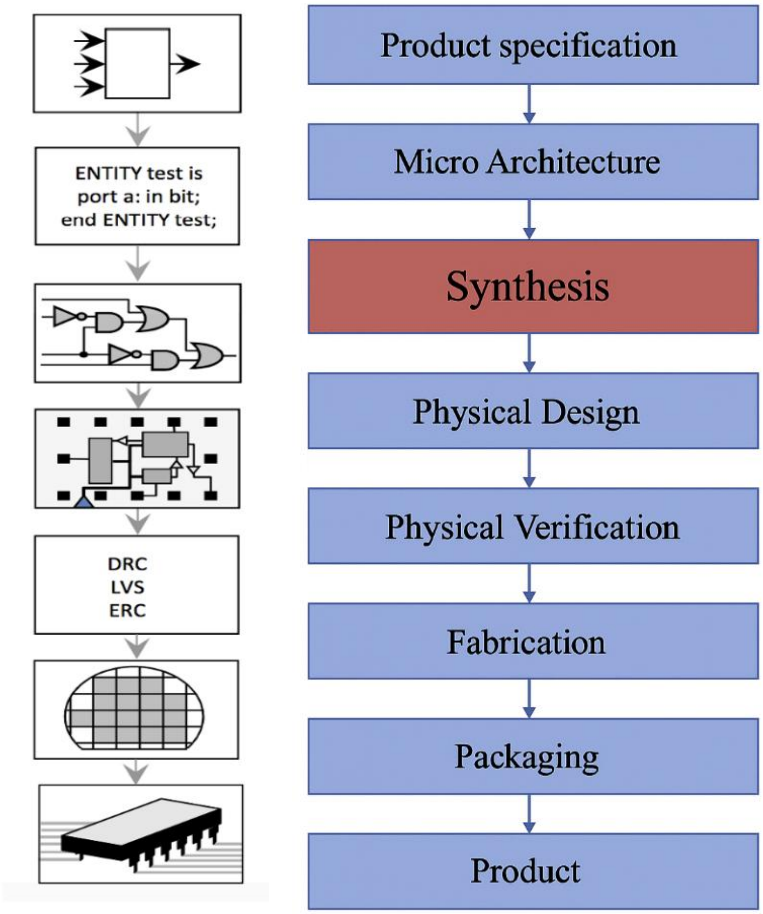
For more information on how to build libraries visit:

https://spdocs.synopsys.com/dow_retrieve/qsc-w/dg/fcolh/W-2024.09/fcolh/icc2lib/preparing_cell_libraries/building_cell_libraries.html

VLSI Design Flow

The figure below presents the standard VLSI design flow, from product specification to final chip production.

Synthesis is the key stage that converts the RTL design into a gate-level representation, preparing the design for physical implementation.



The next sections explain the goals, steps, and implementation of the synthesis process in this project.

RTL-to-Gate-Level Logic Synthesis

Goals

The primary goal of this synthesis flow is to convert the RTL-level description of the `riscv_core` design into an optimized, technology-specific gate-level netlist suitable for fabrication in the TSMC 28nm HPC+ process. This process ensures that the synthesized design meets all defined performance constraints across multiple operating scenarios (Fast and Slow corners), thereby guaranteeing robustness and reliability in real-world conditions. Specifically, the synthesis flow aims to achieve optimal trade-offs among critical parameters such as timing (setup and hold), area utilization, and power consumption, leveraging Synopsys Fusion Compiler's advanced optimization capabilities. Additionally, the defined multi-corner multi-mode (MCMM) approach ensures accurate modeling and verification of the design across realistic variations in process, voltage, and temperature (PVT), thus enhancing overall quality and manufacturability.

Flow Overview

The synthesis flow includes the following key stages:

- Technology and Physical Library Setup
  Defines the technology file and attaches the relevant standard cell libraries (SVT, HVT, LVT) for synthesis and physical implementation.

- RTL Compilation and Elaboration
  The Verilog source files are compiled and elaborated into a design hierarchy. The top module (`riscv_core`) is set for synthesis.

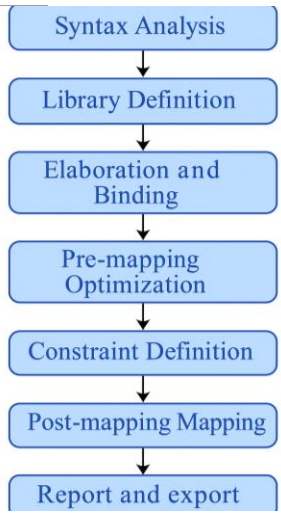
- Multi-Corner Multi-Mode (MCMM) Setup
  Defines different operating conditions (Fast/Slow corners) and functional modes to

ensure timing is met across all realistic scenarios.

- Constraint Loading
  Loads the timing constraints from the SDC file, including clock definitions, input/output delays, and transition/load conditions.

- Output Block Generation
  The synthesized and elaborated design is saved as a named block, enabling further analysis, optimization, and integration in downstream flow stages.

Main Steps of Logic Synthesis



- Syntax Analysis
  Checks RTL code for syntax correctness and consistency.

- Library Definition
  Utilizes standard cell libraries defined by several file formats:

  - Liberty (.lib): Contains timing, power, and noise characterization data.
  - Library Exchange Format (LEF): Provides abstract views for placement and routing.
  - GDSII: Defines full layouts required for mask generation.
  - NDM (New Data Model): Integrates both logic and physical design data used by Synopsys tools.

- Elaboration and Binding
  Converts the RTL into Boolean data structures and binds RTL modules to technology-independent generic gates.

- Pre-mapping Optimization
  Performs Boolean logic minimization to simplify the logic functions, reducing gate count and improving performance.
  Utilizes methods like Espresso heuristic minimization and Binary Decision Diagrams (BDDs).

- Constraint Definition
  Defines design constraints such as clock periods, input/output delays, capacitive loads, and environmental conditions in SDC format.

- Technology Mapping
  Maps the generic gate-level logic onto cells available in the selected technology library.
  Employs algorithms like recursive tree-covering to optimally select standard cells based on constraints.

- Post-mapping Optimization
  Applies timing optimization techniques such as resizing gates, buffering signals, cloning cells to distribute load, restructuring fan-in and fan-out trees, and retiming sequential elements.

- Report and Export
  Generates detailed reports on timing, area, power consumption, and other critical parameters.
  Exports the optimized gate-level netlist for further processing in the physical design stage.

Synopsys Fusion Compiler Integration

Fusion Compiler combines synthesis and place-and-route optimization in a unified RTL-to-GDSII platform. It supports:

- Unified Flow: Seamlessly transitions from logic synthesis to physical implementation.
- Design Constraints: Enables comprehensive constraint specification (SDC format).
- Physical Awareness: Integrates automatic floorplanning, placement-aware synthesis, clock tree synthesis (CTS), and routing.
- Advanced Optimization: Supports techniques like clock gating (global and local), multibit optimization, and congestion-aware placement.

By following structured synthesis methodologies and leveraging the comprehensive capabilities provided by Synopsys tools (as detailed in Synopsys SOLVNET documentation), designers can significantly improve the quality and efficiency of their VLSI designs.

Script Breakdown and Explanation

Below is a detailed explanation of the Fusion Compiler synthesis script used in this project. Each section corresponds to a major step in the RTL-to-gates transformation flow.

Environment Setup

```
lappend search_path scripts design_data
```

Adds the directories (scripts, design_data) to the search path used by Fusion Compiler.

```
set_host_options -max_cores 8
```

Limits the synthesis tool to use up to 8 CPU cores for parallel processing.

```
set TECH_FILE
"/data/tsmc/28HPCPMMWAVE/synopsys/tsmcn28_9lm6X1Z1URDL.tf"
```

Defines the technology file (.tf) for the TSMC 28HPC+ process, containing all layer and rule definitions.

Physical Library Settings

```
create_lib -technology $TECH_FILE -ref_libs { ... } RISCV1.dlib
```

Creates a new physical design library (RISCV1.dlib) linked to the technology file and standard cell libraries (.ndm).

```
open_lib RISCV1.dlib
```

Opens the design library for use.

```
report_ref_libs
```

Reports all reference libraries attached to the design library (sanity check).

Parasitic Technology Definitions

```
read_parasitic_tech -tlup ...rcbest... -name rcbest
```

Loads the "best-case" RC parasitic model (typically used for setup analysis).

```
read_parasitic_tech -tlup ...rcworst... -name rcworst
```

Loads the "worst-case" RC parasitic model (typically used for hold analysis).

```
save_lib
```

Saves the current state of the library after tech/RC setup.

```
set_svf ./guidFM/riscv_core1.svf
```

Defines an SVF file for Formality (formal equivalence checking between RTL and synthesized netlist).

RTL Compilation and Elaboration

```
analyze -format sverilog [glob rtl/*.v]
```

Analyzes all SystemVerilog files in the rtl/ directory and compiles them into the internal representation.

```
elaborate riscv_core
```

Elaborates the RTL hierarchy with riscv_core as the top-level module.

```
set_top_module riscv_core
```

Explicitly sets riscv_core as the synthesis top.

```
start_gui
```

Launches the graphical user interface (optional; useful for debugging/inspection).

```
save_block -as RISCV1/elaborate
```

Saves a checkpoint (snapshot) of the elaborated design block for later stages.

MCMM Setup

```
remove_corners -all
 remove_modes -all
 remove_scenarios -all
 remove_pin_blockages -all
```

Clears prior MCMM definitions and pin blockages to ensure a clean configuration.

```
create_corner Fast
 create_corner Typical
 create_corner Slow
```

Defines three PVT corners for analysis.

```
set_parasitics_parameters -early_spec rcbest -late_spec rcbest -
corners {Fast}
```

```
 set_parasitics_parameters -early_spec rcbest -late_spec rcbest
-corners {Typical}
 set_parasitics_parameters -early_spec rcworst -late_spec
rcworst -corners {Slow}
```

Assigns RC parasitic models to each corner (best for Fast/Typical, worst for Slow).

```
create_mode FUNC
 current_mode FUNC
```

Creates the functional mode and sets it active.

```
create_scenario -mode FUNC -corner Fast -name FUNC_Fast
 create_scenario -mode FUNC -corner Typical -name FUNC_Typical
 create_scenario -mode FUNC -corner Slow -name FUNC_Slow
```

Creates analysis scenarios that combine the functional mode with each corner.

Timing Constraints (SDC)

```
current_scenario FUNC_Fast
 source riscv.sdc
```

Activates FUNC_Fast and applies constraints from riscv.sdc.

```
current_scenario FUNC_Typical
 source riscv.sdc
```

Activates FUNC_Typical and applies the same constraints.

```
current_scenario FUNC_Slow
 source riscv.sdc
```

Activates FUNC_Slow and applies the same constraints.

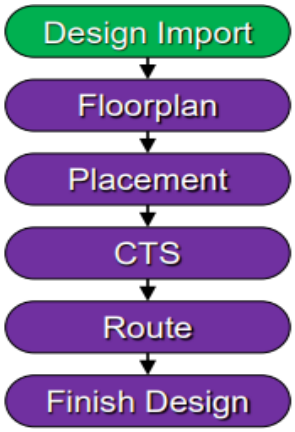
Output

After running this synthesis flow:

- A technology-mapped gate-level netlist is generated for the `riscv_core` module.
- The netlist is saved as a named block (`RISCV/elaborate`) for use in physical design.
- Timing constraints are verified across Fast and Slow corners using loaded SDC files.
- The design is fully prepared for downstream steps such as placement, clock tree synthesis (CTS), and routing.

Moving from Logical to Physical:

During synthesis, our world view was a bit idealistic. So,we didn't care about power supplies, physical connections/entities, and clock non-idealities. Therefore, in order to start physical implementation: We need to define "global nets" and how they connect to physical instances, provide technology rules and cell abstracts (.ndm files), provide physical cells, unnecessary for logical functionality: Tie cells, P/G Pads, DeCaps, Filler cells, etc, define hold constraints and all operating modes and conditions (MMMC), hold was "easy to meet" with an ideal clock, so we didn't really check it, and set up "low power" definitions, such as voltage domains, power gates, body taps, etc.



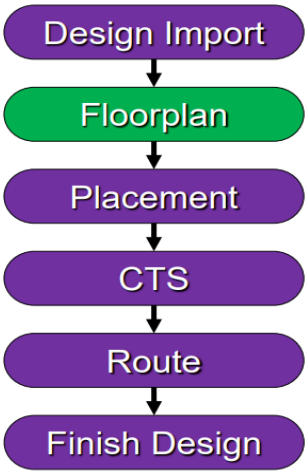
Floorplan:

What is Floorplanning?

Floorplanning is the process of defining the physical structure of a digital integrated circuit before the actual placement and routing of standard cells.

It's the first major step in the physical design phase of a chip. Floorplanning acts as the bridge between the logical netlist (produced during synthesis) and the physical layout of the chip (GDSII output).

Think of it like urban planning for a city. Just as you would plan where to place schools, roads, parks, and utilities before building a city, in chip design, floorplanning determines where macros, logic cells, I/Os, and power resources will go—*before* any actual cells are placed.



🔷 Why is Floorplanning Necessary?

Floorplanning is a critical step because it marks the transition from a purely logical view of the design to a physical implementation that must respect real-world constraints. While the RTL (Register Transfer Level) and synthesis stages focus on functionality and logic optimization, they are largely technology-independent—they don't care about actual distances, wire congestion, or power delivery.

In contrast, the physical design must address many technology-dependent concerns. These include the size and shape of the die, the physical location of each block, how power and ground will be distributed across the chip, and how the clock signal will be delivered to all sequential elements with minimal skew. It must also consider routing congestion, which happens when too many wires compete for space in a small area, and timing and signal integrity, ensuring signals arrive on time without degradation.

Because these factors can significantly affect the chip's performance, power efficiency, and manufacturability, floorplanning is essential. It lays the groundwork to ensure that the chip will not only function correctly but also be feasible to build using the chosen semiconductor process.

◇ Goals of Floorplanning:

1. **Defining the Core and Die Area:** The first goal of floorplanning is to define the die and core area-that is, how big the chip will be and how much space is used for placing logic. This step balances performance, cost, and available area for routing and power.
2. **Placing Macros and Hard IPs:** we need to place large components like RAMs or PLLs, called macros or hard IPs. These blocks must be positioned early to avoid blocking important routing paths and to reduce timing delays.
3. **Determining I/O and Pad Placement:** another goal is to plan where to put the I/O pads. These pads connect the chip to the outside world, and their location affects how easily signals can enter and leave the chip.
4. **Creating Power and Ground Structures:** we also define power and ground networks, like metal rings and stripes. These ensure the chip receives stable voltage and prevents issues like voltage drop or overheating due to high current.
5. **Defining Placement Blockages and Regions:** Floorplanning also sets up placement regions and blockages, which help guide tools to put logic in the right places while keeping sensitive areas clear-especially near large blocks or critical paths.
6. **Targeting Performance Metrics:** floorplanning aims to optimize timing, congestion, and power. A smart layout improves performance, prevents routing overload, and helps the chip use less energy.

🔷 Advantages and Disadvantages of Floorplanning:

Advantages of Floorplanning	Disadvantages of Floorplanning
Better Performance: Shorter wirelengths improve speed and timing.	Estimation-Dependent: Early decisions rely on non-final data.
Lower Power Consumption: Efficient layout reduces switching and leakage power.	Requires Experience: Needs manual skill and planning knowledge.
Easier Routing: Prevents congestion by reserving enough space	Area vs. Performance Tradeoff: Small chips may increase congestion.
Improved Timing Closure: Helps meet setup and hold constraints.	Macro Blocking: Poor placement causes routing detours.
Predictable Clock Tree: Simplifies CTS with well-placed flip-flops.	Hard to Automate: Often requires manual adjustments.
Less Rework: Reduces the need for later changes.	

🔷 How to Perform Floorplanning in Fusion Compiler:

In Fusion Compiler, floorplanning is typically initiated using the command:

“set_auto_floorplan_constraints -core_utilization 0.6 -side_ratio {1 1} -core_offset 2”

This command automatically generates a basic floorplan layout based on the design's logic size, timing constraints, and technology settings provided earlier in the flow. It sets the structure of the core area and reserves space for future routing and power planning.

Explanation of Parameters:

- -core_utilization 0.6:
  This sets the target utilization of the core area to 60%. It means that 60% of the core area will be used for standard cell placement, while 30% is left open for routing and optimization flexibility.

- -side_ratio {1 1}:
  This defines the aspect ratio of the core area. {1 1} means the core will be a square (equal width and height), which helps achieve balanced placement and routing.

- -core_offset 2:
  This sets a 2-micron offset (margin) between the edge of the core and the die boundary. It is used to reserve space for I/O pads, power rings, and routing tracks around the core.

Early Design Checks

Run this command:

```
set_app_options -name place.coarse.continue_on_missing_scandef -value true

compile_fusion -check_only
```

This allows the tool to proceed even without a scan definition file (SCANDEF).And, runs basic design sanity checks (library loading, constraints, connectivity) before placement.

Mapping and Logic Optimization

```
compile_fusion -to initial_map

compile_fusion -from logic_opto -to logic_opto
```

- **initial_map** – maps the logical netlist into physical standard cells.

- **logic_opto** – performs early logic optimization and produces a coarse placement seed.

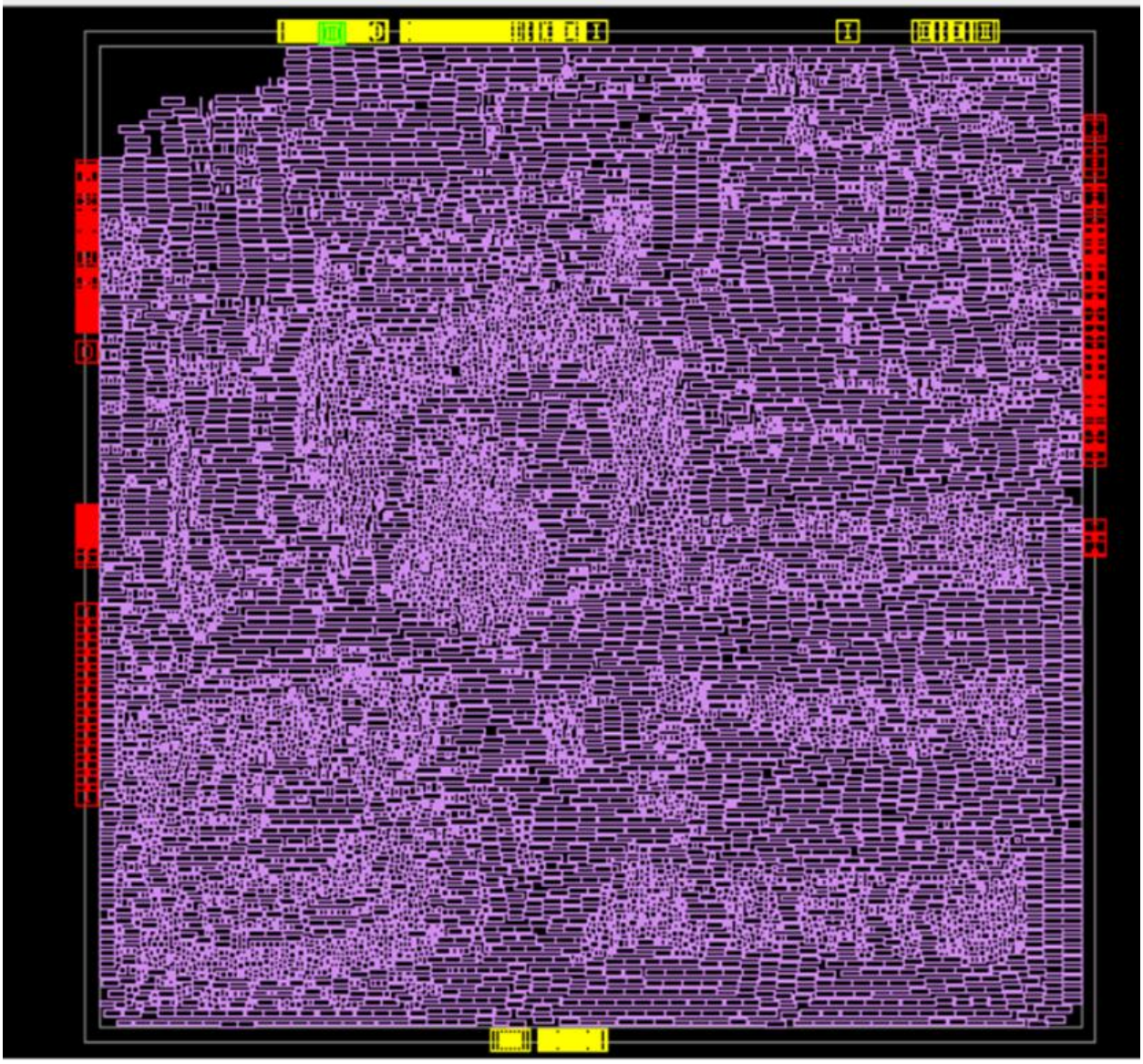
Pin Placement (I/O)

```
set_app_options -name compile.auto_floorplan.place_pins -value all

set ports [remove_from_collection [get_ports] {VDD VSS}]

report_block_pin_constraints -self

place_pins -use_existing_routing -self
```

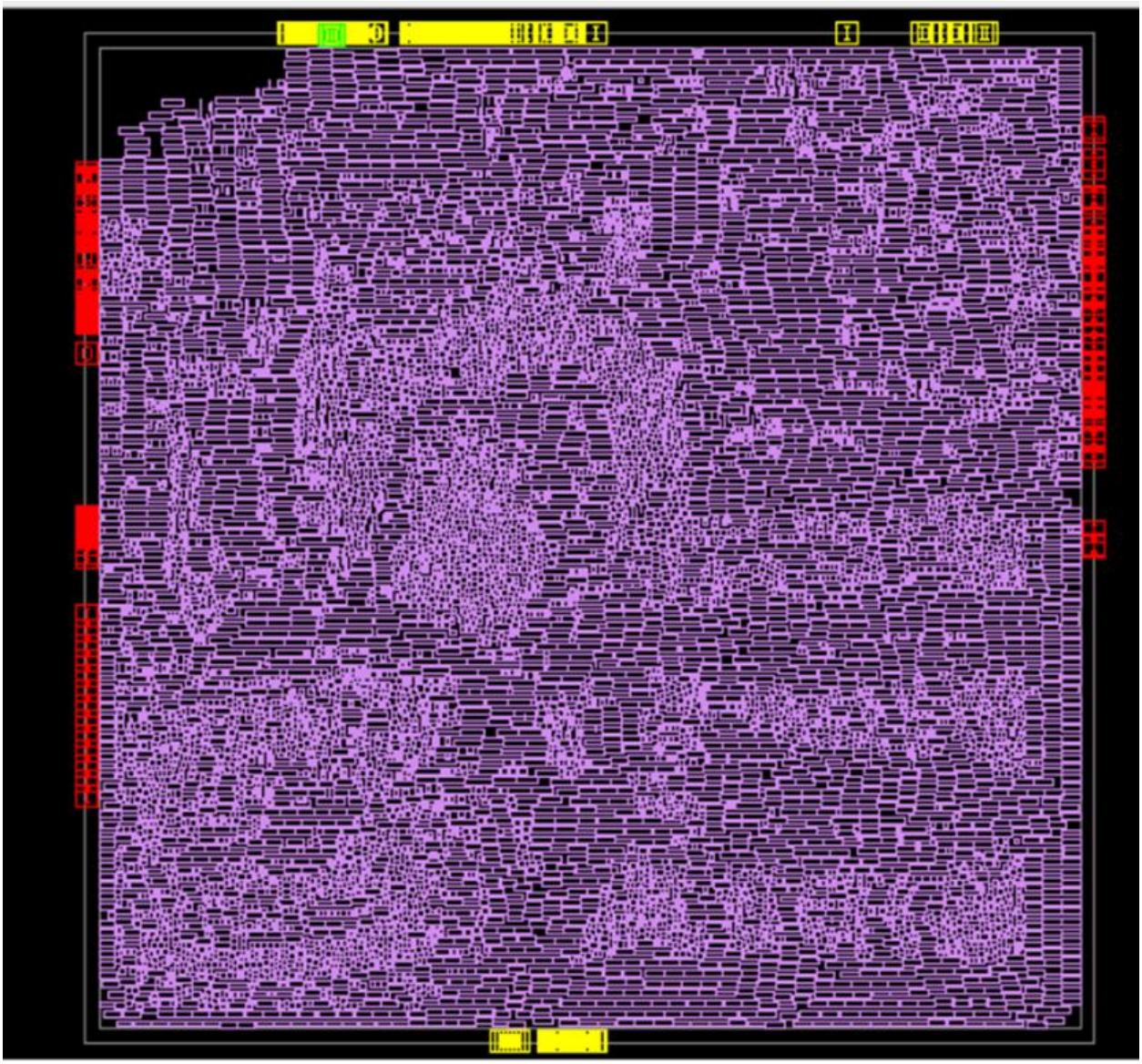
- Enables automatic placement of I/O pins around the block boundary.
- Excludes VDD/VSS from normal pin rules (they are global power/ground nets).
- Reports pin constraints and places pins, reusing existing routing if available.

```
compile_fusion -from initial_place -to initial_place

compile_fusion -from initial_drc -to initial_drc

compile_fusion -from initial_opto -to initial_opto
```

- **initial_place** – creates the first legal placement (no overlaps, aligned to rows).
- **initial_drc** – performs an early design-rule check on placement.



- **initial_opto** – improves timing and congestion after initial placement.



Final Placement and Optimization

compile_fusion -from final_place -to final_place

compile_fusion -from final_opto -to final_opto

- **final_place** – tightens the placement for better density and organization.
- **final_opto** – performs final optimization considering timing, congestion, and power.

Save

check_legality

save_block -as RISCV1/floorplan

- **check_legality** – verifies that placement is legal (no overlaps, correct spacing, valid rows).

- **save_block** – saves the design block in its floorplan state, including pin placement, legal cell placement, and early optimization.

What Happens During This Step?

When this command is executed, Fusion Compiler analyzes the size and structure of the netlist and automatically generates a core layout inside the die. The tool estimates the required area, shapes the core, and reserves margins for I/Os and power distribution. This floorplan is then used as the starting point for placement and clock tree synthesis (CTS).

It is possible to refine this auto-generated floorplan manually afterward by placing macros, defining regions, and adjusting routing blockages before proceeding to the next stage of the flow.

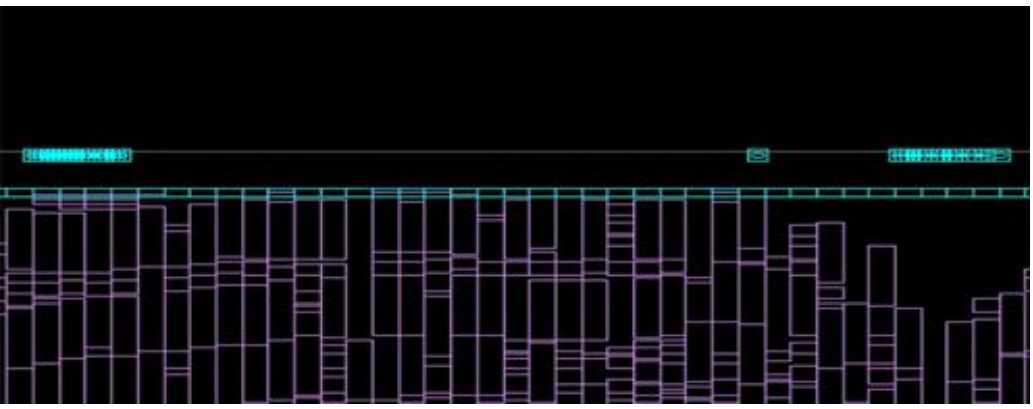
Once the basic floorplan has been generated using the set_auto_floorplan_constraints command, the following step in the flow is to compile the design up to the logic optimization stage:

Boundary Cells

What are Boundary Cells?

Boundary cells are physical-only cells that are placed at the edges of standard cell rows or at block boundaries. They do not implement any logical functionality, but they play a critical role in ensuring manufacturability and physical integrity of the design.

Unlike functional cells, boundary cells exist purely to handle physical effects at the layout edges. They cap the design rows, maintain the continuity of wells and power rails, and prevent manufacturing damage or Design Rule Check (DRC) violations at the boundaries.



Why are Boundary Cells Important?

1. **Protect Edge Transistors**
   Transistors at the edges of rows are vulnerable during fabrication. Boundary cells act as protective buffers.
2. **Ensure Continuity**
   They extend Nwell/Pwell and VDD/VSS rails all the way to the block edge, preventing broken connections.
3. **Avoid DRC Violations**
   By legally terminating standard cell rows, they eliminate spacing and enclosure violations at boundaries.

4. **Support Clean Flow**
   Their insertion simplifies the addition of tap cells, fillers, and routing steps later in the flow.

Summary: Boundary cells = safe row edges + continuous wells/rails + DRC clean.

When to Insert Boundary Cells?

Boundary cells are inserted early in the physical design flow, right after floorplanning and initial placement, and before inserting tap cells and filler cells.

Recommended sequence in the flow:

1. Floorplan & initial placement
2. **Boundary cell insertion**
3. Tap cell insertion
4. Power grid generation
5. Placement optimization
6. Clock Tree Synthesis (CTS)
7. Routing
8. Final signoff

Why at this stage?

- At this point, the row structure is already defined, so boundary cells can properly terminate the rows.
- They must be present before tap and filler cells to ensure well continuity and avoid DRC issues at the edges.
- Early insertion avoids costly placement rework later in the flow.

How to Insert Boundary Cells in Fusion Compiler

Boundary cells are physical-only cells placed at the edges of standard cell rows to protect edge transistors, ensure well/rail continuity, and avoid DRC violations. In Synopsys Fusion Compiler, they are added after floorplanning and initial placement.

Steps:

1. **Insert Boundary Cells**

Use the `create_boundary_cells` command to add left and right edge cells from the technology library.

```
create_boundary_cells \

  -left_boundary_cell
tcbn28hpcplusbwp30p140/BOUNDARY_LEFTBWP30P140 \

  -right_boundary_cell
tcbn28hpcplusbwp30p140/BOUNDARY_RIGHTBWP30P140 \

  -prefix BOUND
```

- Defines left/right termination cells.
- Adds a BOUND prefix for easy identification.

2. **Legalize Placement**

   After insertion, legalization aligns the cells properly and fixes overlaps.

```
legalize_placement -incremental

check_legality
```

3. **Verify Insertion**

   Check that the boundary cells were added and the layout is clean.

```
set bnd_cells [get_cells -hierarchical -filter "name =~
BOUND*"]

puts "Boundary count = [sizeof_collection $bnd_cells]"

signoff_check_drc
```

- Counts inserted cells.
- Runs legality and DRC checks.

Advantages and Disadvantages of Boundary Cells:

Advantages	Disadvantages
Protect edge transistors from fabrication damage	Slight increase in chip area
Ensure continuity of Nwell/Pwell and VDD/VSS rails	Placement complexity near macros or blockages
Prevent DRC violations at row boundaries	Require extra verification (legality & DRC checks)
Provide a clean foundation for tap/filler insertion, CTS, and routing	Dependent on correct library variants (SVT/HVT/LVT)
Improve manufacturability and alignment with foundry rules	—

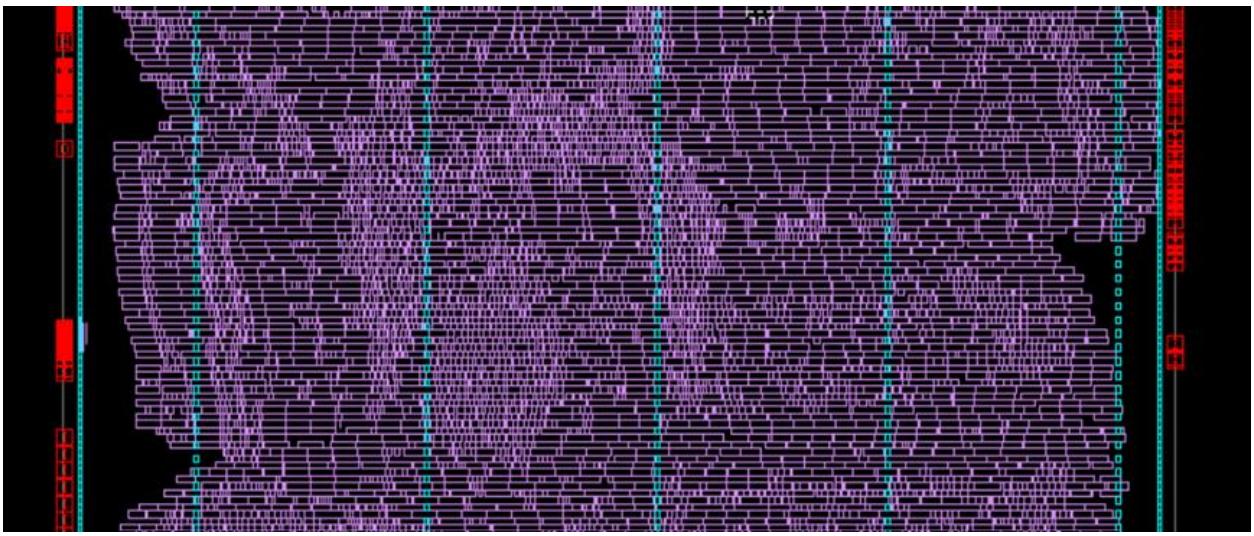
Common Issues & Fixes

- **Cell not found in library** → Verify library paths and ensure the correct `.ndm` files are loaded.
- **Overlap with macros** → Adjust halos or re-run incremental legalization.
- **Remaining edge DRCs** → Re-insert boundary cells with correct variants, or ensure row definitions are clean.

Interaction with Other Physical-Only Cells

- **Tap cells:** inserted after boundary cells, ensure Nwell/Pwell are tied to power rails across rows.
- **Filler cells:** fill internal gaps between standard cells, while boundary cells terminate row edges. Both are essential for continuity.

Tap Cells



What are Tap Cells?

Tap cells are *physical-only* cells that connect the substrate and wells of the chip to the power (VDD) and ground (VSS) networks. Their purpose is to prevent latch-up, reduce leakage currents, and ensure stable operation of transistors. Unlike logic cells, they have no functional role in computation, but they are required by design rules of the foundry.

Why are Tap Cells Necessary?

- ❖ Ensure reliable well and substrate biasing.
- ❖ Prevent latch-up effects caused by parasitic structures.

- ❖ Distribute well and substrate connections uniformly across the layout.
- ❖ Comply with foundry design rules specifying maximum allowed distance between taps.

When to Insert Tap Cells?

Tap cells are inserted after placement legalization and before building the power grid (PG). They are often inserted together with boundary cells, which close the placement rows at the edges of the core.

Goals of Tap Cell Insertion:

1. Ensure every well/substrate region is tied to the correct potential.
2. Maintain DRC compliance regarding maximum tap spacing (e.g., ≤ 60 µm in TSMC 28nm).
3. Preserve placement legality and avoid overlaps with fixed cells and macros.

How to Insert Tap Cells in Fusion Compiler:

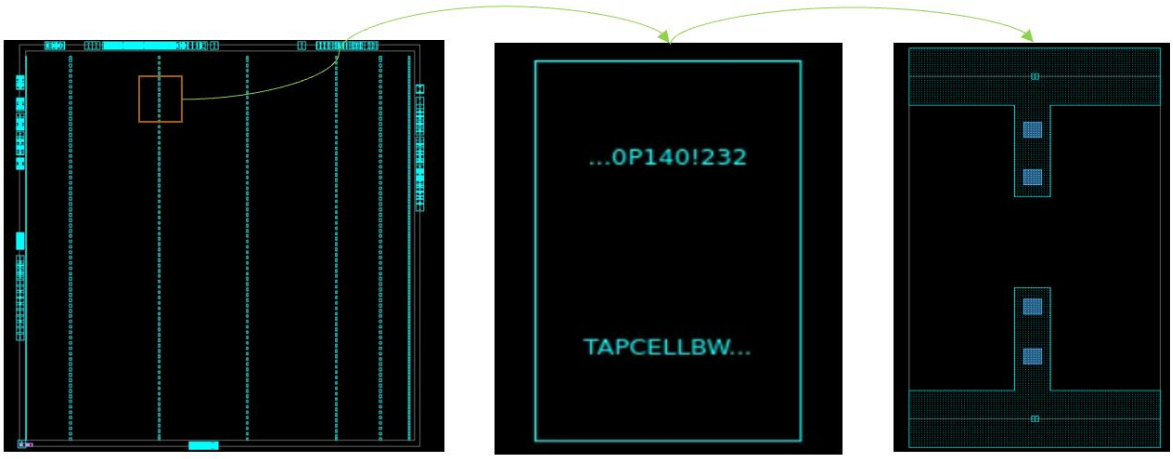
```
################################################################################
## top_placed_with_tap_Cells
################################################################################
# Insert TAP cells to ensure well and substrate ties
create_tap_cells \
  -lib_cell tcbn28hpcplusbwp30p140/TAPCELLBWP30P140 \
  -distance 60 \
  -pattern stagger \
  -skip_fixed_cells
```

#Checking legality of placement

legalize_placement -incremental

check_legality

save_block -as RISCV1/top_placed_with_tap_and_boundary



Advantages and Disadvantages of Tap Cells:

Advantages	Disadvantages
Prevent latch-up and leakage	Consume extra area (though small)
Ensure compliance with foundry rules	Must be carefully planned around macros
Improve power integrity	May increase runtime slightly

References for Further Reading:

- Synopsys Fusion Compiler – Product Page\

  https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler.html

- Fusion Compiler Command Reference (includes `create_tap_cells`) https://www.scribd.com/document/856392879/Fc-Commands

Formality Handshake in Fusion Compiler (SVF & Netlist Export)

Purpose

This section explains how to prepare the artifacts that Formality needs — the SVF guidance file and the gate-level netlist — directly from Fusion Compiler (FC). Doing this correctly prevents "global unmatched" issues and streamlines RTL↔Netlist equivalence.

Key idea: enable `set_svf` before RTL analyze/elaborate, synthesize/optimize, write the implementation netlist, then `set_svf -off`. These artifacts are then loaded in Formality GUI.

Where Each Command Runs

- Fusion Compiler (FC): library/tech setup, RC tech, RTL analyze/elaborate, MCMM, synthesis/opt, netlist export, SVF on/off.
- Formality (FM): Auto Setup, clock-gating options, load SVF (Stage 0), load RTL (Stage 1), load Netlist + DB (Stage 2), Matching (Stage 4), Verify (Stage 5), Debug (Stage 6).

Minimal FC Recipe for Formality

> Use this condensed block when you only need to generate clean inputs for Formality. Run inside fc_shell.

--- Workspace & Tech ---

```
lappend search_path scripts design_data
set_host_options -max_cores 8
set TECH_FILE "/data/tsmc/28HPCPMMWAVE/synopsys/tsmcn28_9lm6X1Z1URDL.tf"

create_lib -technology $TECH_FILE -ref_libs {
  /data/.../tcbn28hpcplusbwp30p140.ndm
  /data/.../tcbn28hpcplusbwp30p140hvt.ndm
  /data/.../tcbn28hpcplusbwp30p140lvt.ndm
} RISCV1.dlib
open_lib RISCV1.dlib
read_parasitic_tech -tlup /data/.../rcbest.tluplus  -name rcbest
read_parasitic_tech -tlup /data/.../rcworst.tluplus -name rcworst
Save_lib
file mkdir ./guidFM

# --- SVF MUST be ON before RTL analyze ---
set_svf ./guidFM/riscv_core1.svf   ;# <— critical for Formality

# --- Analyze / Elaborate RTL ---
analyze   -format sverilog [glob rtl/*.v]
elaborate riscv_core
set_top_module riscv_core

# --- MCMM (example) ---
remove_corners -all; remove_modes -all; remove_scenarios -all
create_corner Fast; create_corner Typical; create_corner Slow
set_parasitics_parameters -early_spec rcbest -late_spec rcbest -corners {Fast Typical}
set_parasitics_parameters -early_spec rcworst -late_spec rcworst -corners {Slow}
create_mode FUNC; current_mode FUNC
create_scenario -mode FUNC -corner Fast    -name FUNC_Fast
create_scenario -mode FUNC -corner Typical -name FUNC_Typical
create_scenario -mode FUNC -corner Slow    -name FUNC_Slow
current_scenario FUNC_Fast;    source riscv.sdc
current_scenario FUNC_Typical; source riscv.sdc
current_scenario FUNC_Slow;    source riscv.sdc

# --- Synthesis / Physical Opt (illustrative milestones) ---
compile_fusion -check_only
compile_fusion -to initial_map
compile_fusion -from logic_opto -to logic_opto
compile_fusion -from initial_place -to initial_place
```

```
compile_fusion -from final_place   -to final_place
compile_fusion -from final_opto    -to final_opto
check_legality

# --- Export Implementation Netlist for FM ---
file mkdir ./netlist
write_verilog -hierarchy all ./netlist/riscv_core_syn.v

# --- Turn SVF OFF after synthesis completes ---
set_svf -off
```

How It Connects to Formality (GUI Stages)

Formality is Synopsys's formal verification tool used to ensure functional equivalence between RTL and the synthesized gate-level netlist. This guarantees that synthesis optimizations did not alter the design intent. The flow is GUI-driven, supported by SVF (Synopsys Verification File) guidance, and performed after synthesis.

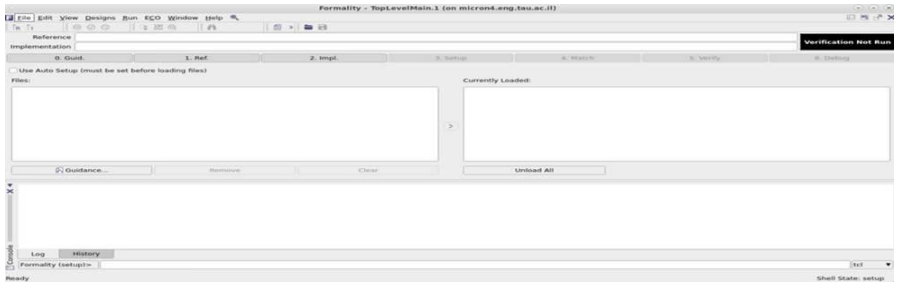
Required Inputs

File / Input	Generated In	Purpose
RTL sources (.v/.sv)	Design team	Golden reference design
Netlist (riscv_core_syn.v)	Fusion Compiler	Synthesized gate-level design
SVF (Synopsys Verification File)	Fusion Compiler	Guides equivalence checking
Library DB/NDM files	Foundry PDK	Technology libraries for gate-level comparison

Preparing the Environment

```
RISCV1.dlib/ RISCV.dlib/
[foqara@micron-x01 ws]$ cd riscv
```

1. Move to the working directory.
2. Launch Formality in GUI mode:
   fm_shell -gui



3. The GUI will be used for setup, file loading, matching, and verification.

```
Formality (setup)> set verification_clock_gate_reverse_gating true
true
```

Stage 0 – Setup

In the GUI, select Auto Setup. Configure clock gating analysis to handle clock transformations:
set verification_clock_gate_reverse_gating true
set verification_clock_gate_hold_mode true
set verification_clock_gate_edge_analysis true
Upload the SVF guidance file before RTL analysis.

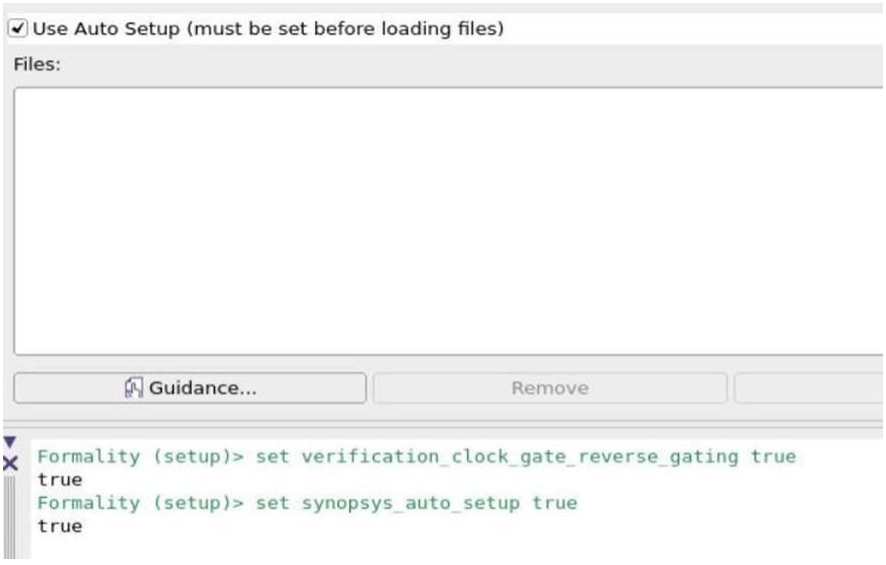


Figure 1: Formality Auto Setup window.

Now we need the Guidance file that tells the tool what cells we have and It helps Formality overcome potential mismatches and verification challenges by providing hints, constraints, or directives.

So, to build this file you can use:

set_svf ./{WORK DIR}/{name}.svf

for example we use :

set_svf ./guidFM/riscv_core1.svf

** you should to put this line before the analyze of the rtl

file mkdir ./netlist

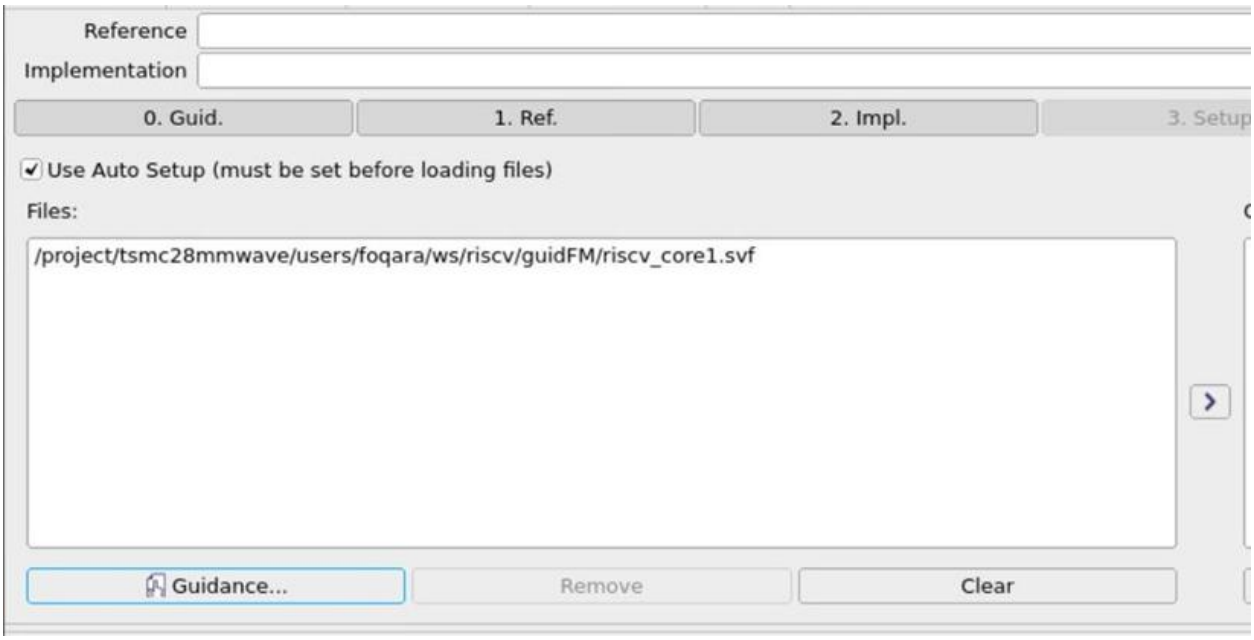
write_verilog -hierarchy all ./netlist/riscv_core_syn.v

and to save the svf file:

set_svf -off

```
# final_opto stage
compile_fusion -from final_opto -to final_opto

check_legality

#############################################################
## Export Netlist for Formality
#############################################################
file mkdir ./netlist
write_verilog -hierarchy all ./netlist/riscv_core_syn.v

#############################################################
## Save SVF for Formality
#############################################################
set_svf -off
#############################################################
```

So now we have the netlist file and the guidance. upload your guidance file to stage 0:



Stage 1 – Load RTL

Upload RTL source files and set the top module.

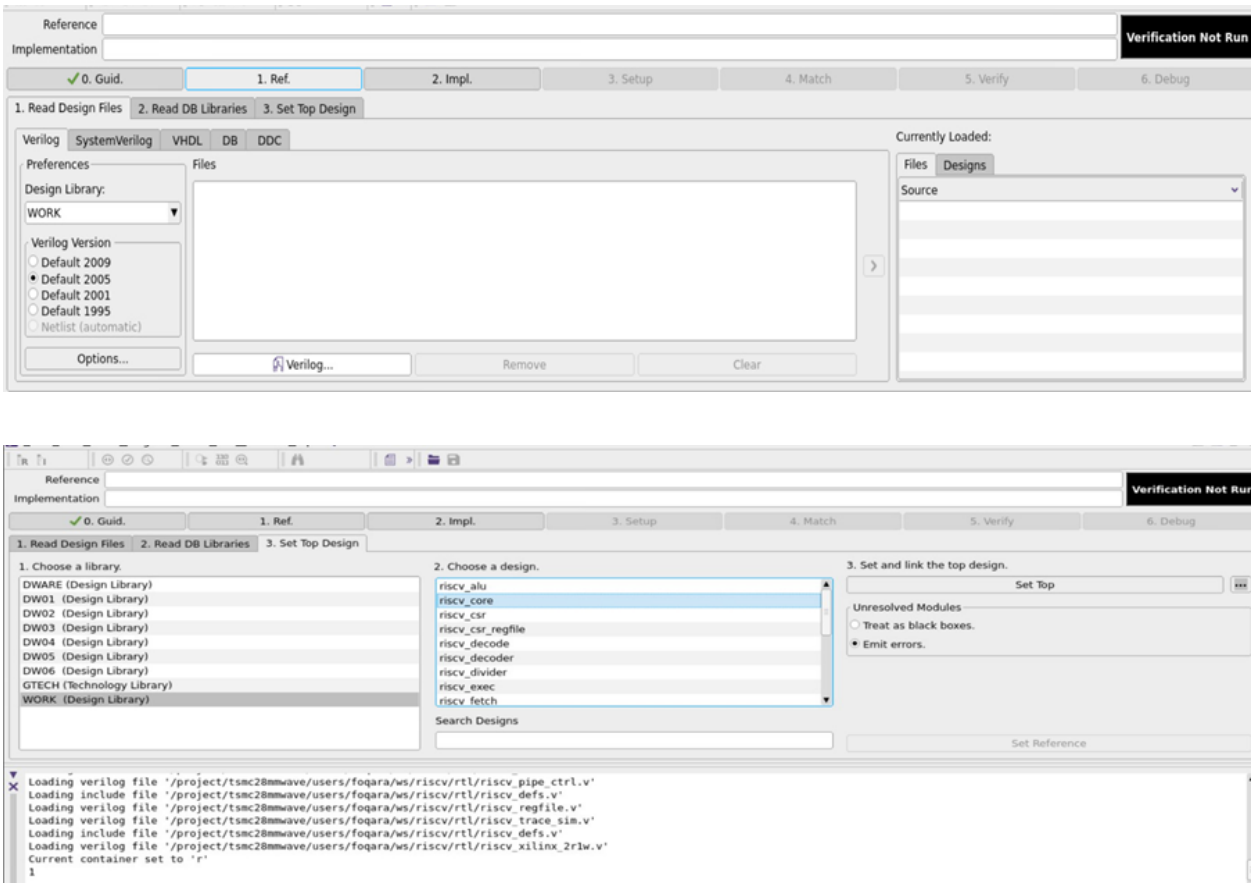
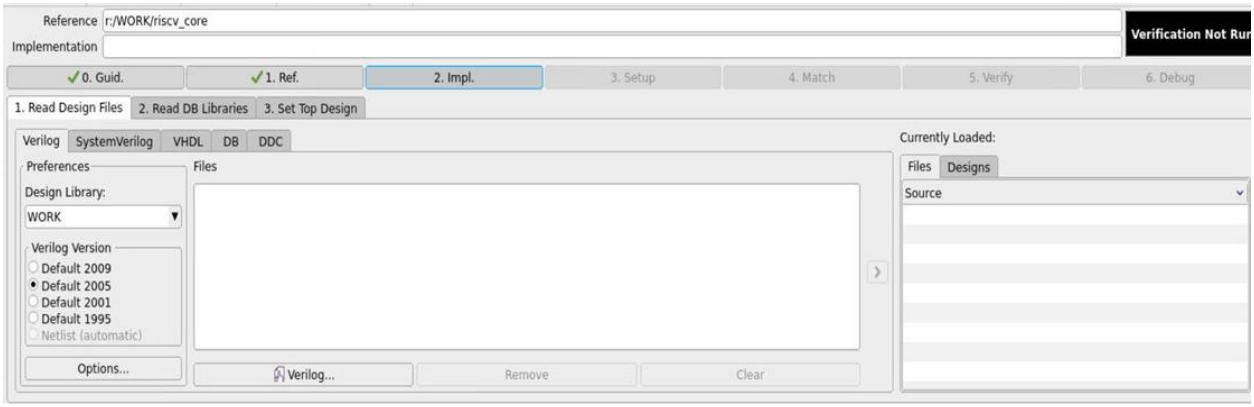


Figure 2: Loading RTL files and selecting the top module.

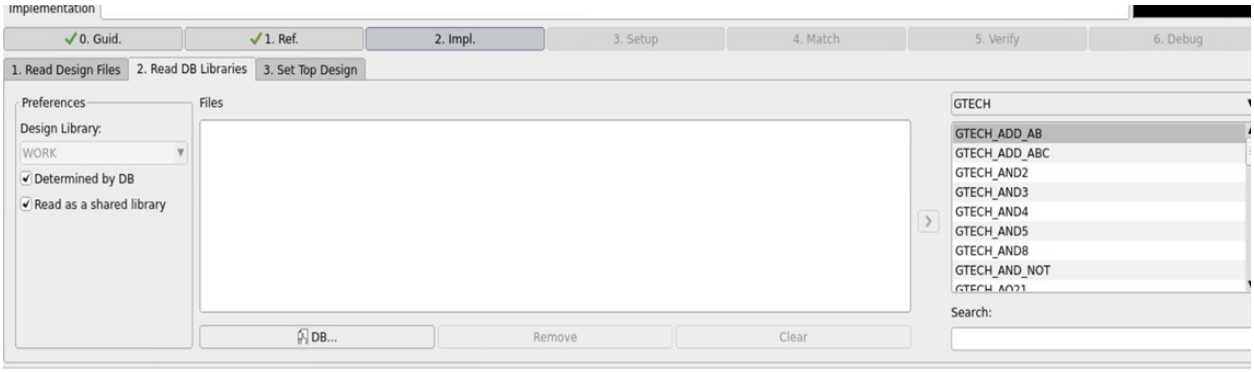
Stage 2 – Load Netlist and Libraries

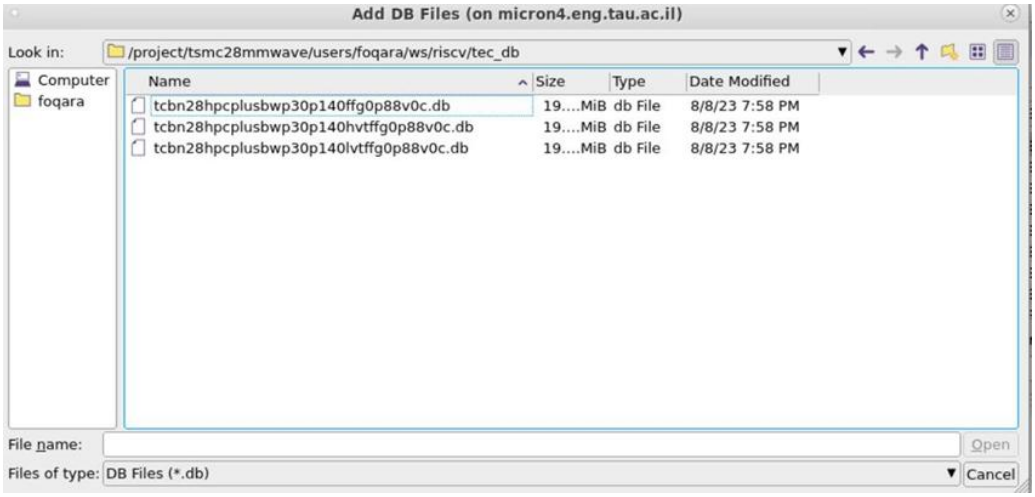
Upload the synthesized netlist and required DB libraries. Then set and link the top design.

Upload the netlist file.



Put the DB files:





Set the top design:

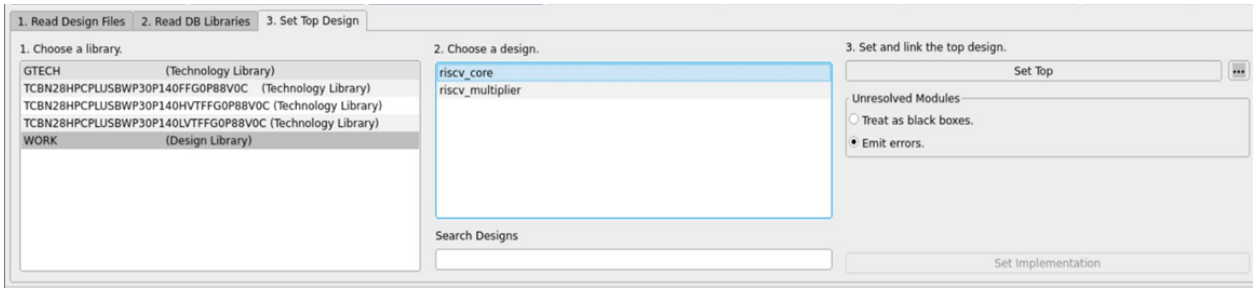


Figure 3: Loading DB libraries and setting top design.

Stage 3 – Constraints

Constraints can be specified if DFT or synthesis constraints exist. In most synthesis-only flows, no constraints are needed.

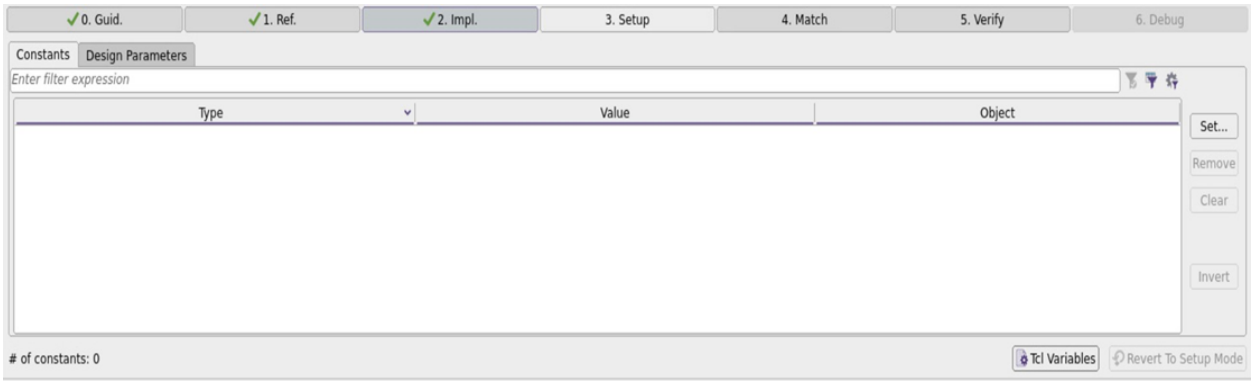
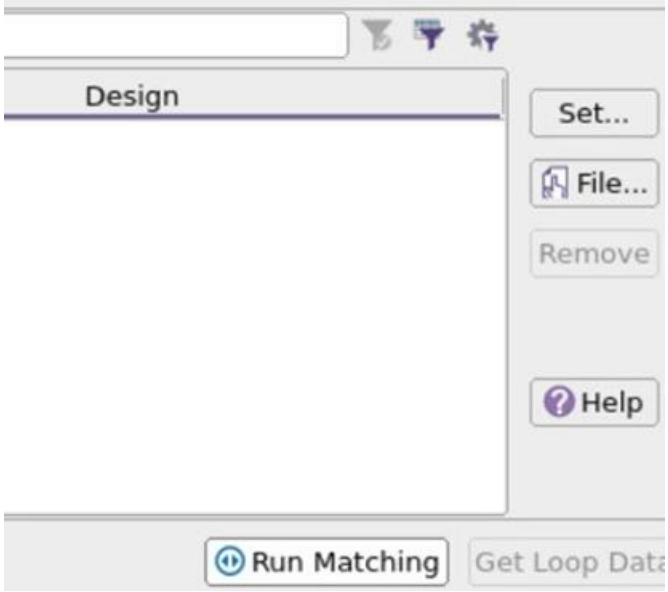
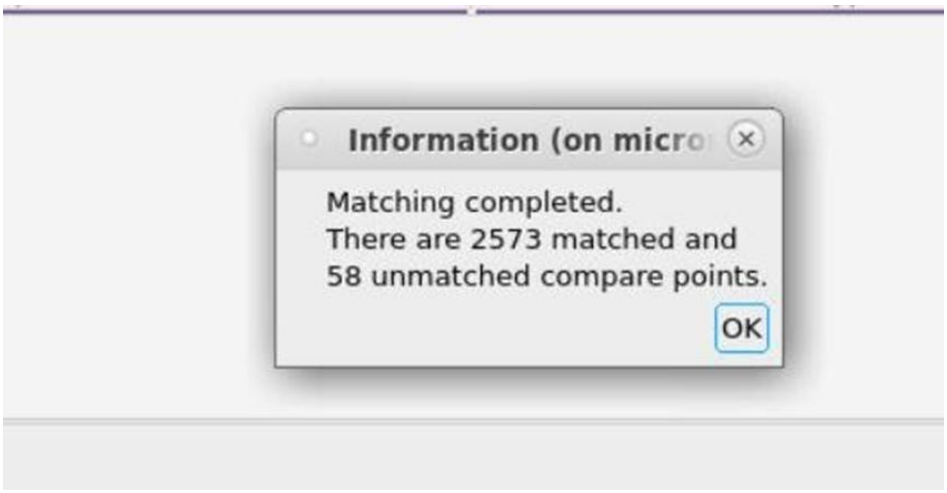


Figure 4: Constraints setup window.

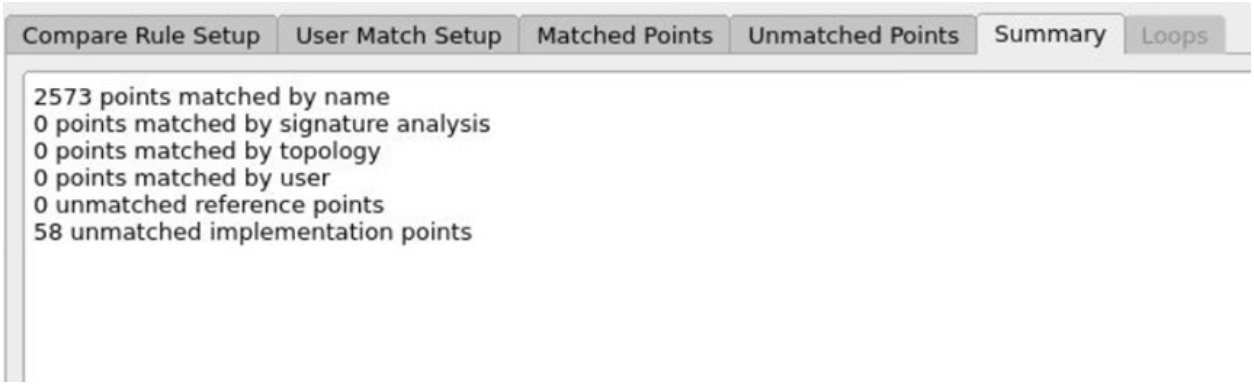
Stage 4 – Matching

Run Matching. Formality compares RTL and netlist hierarchies. Reports indicate matched and unmatched points.





In stage 4 you have summary of the matching and unmatched and unmatched points:



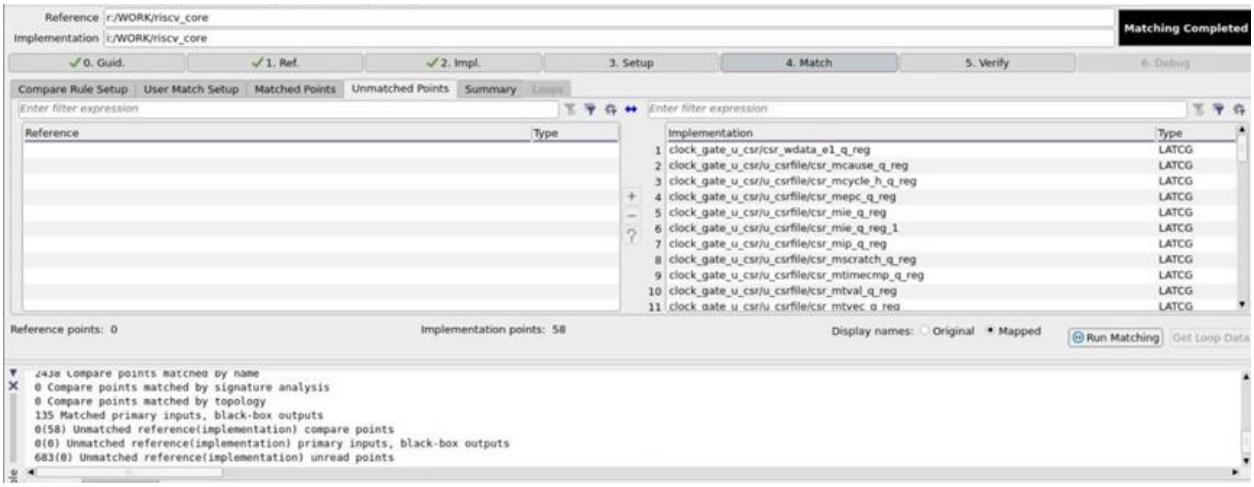
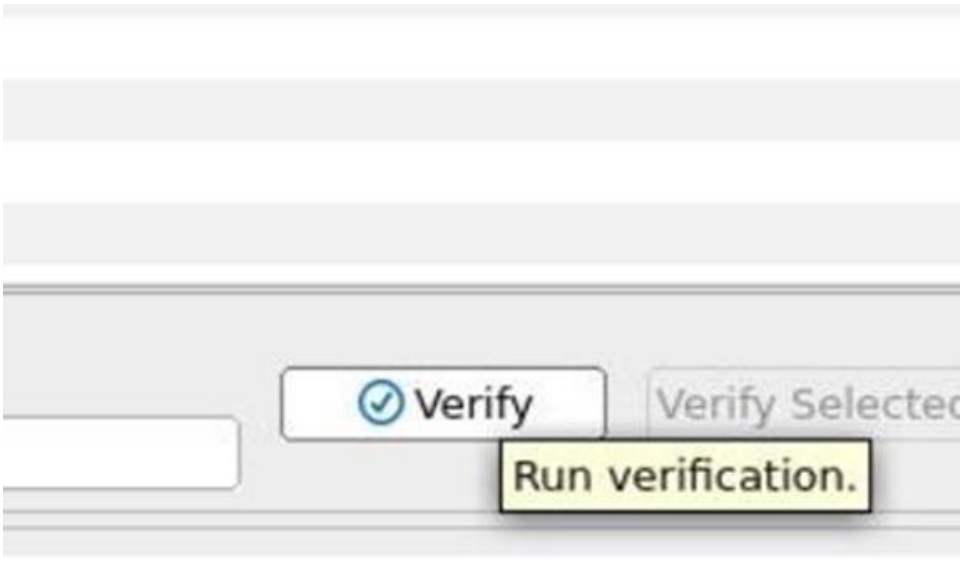


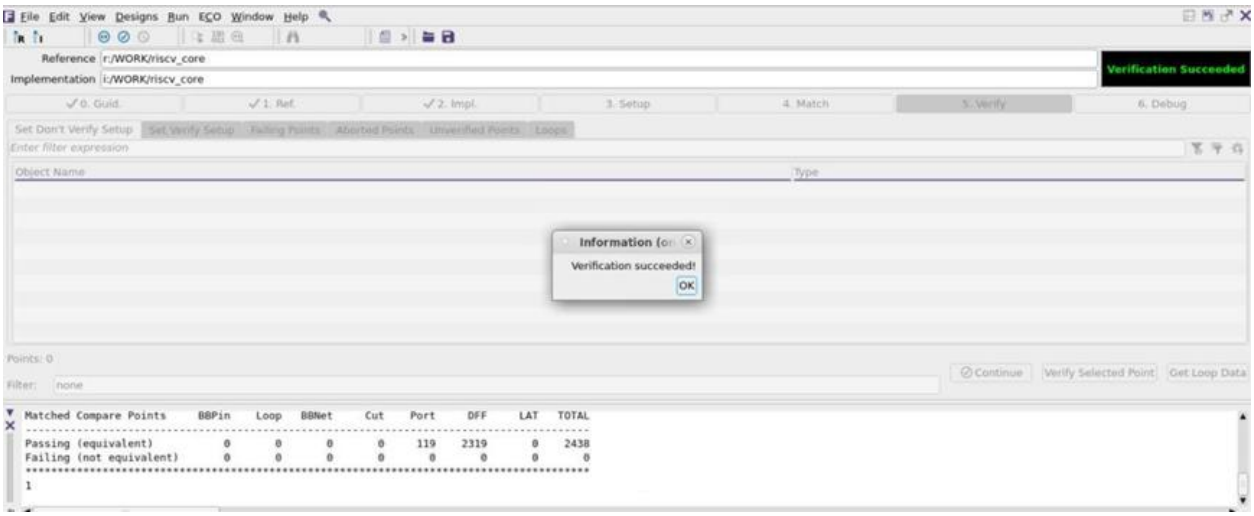
Figure 5: Matching summary report.

So, we see that the unmatched points have a clock gate, so we can move on because we set in stage 0 the line that will solve that in the verification.

Stage 5 – Verification

Click Verify. If successful, Formality outputs 'Verification Succeeded'.



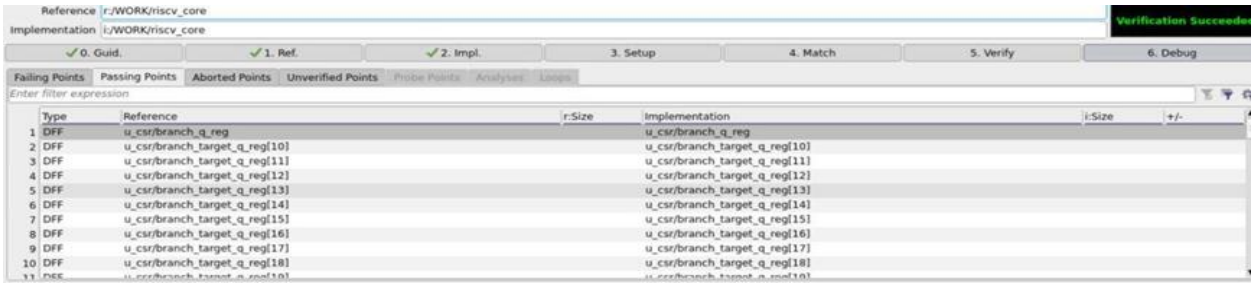


Now we have it!!!

Figure 6: Verification succeeded window.

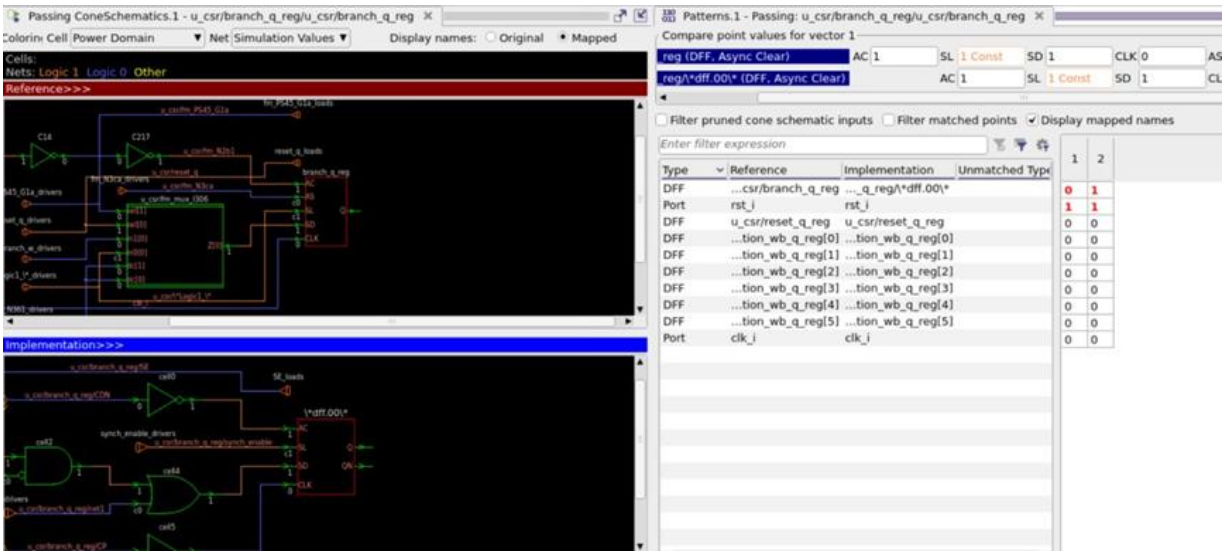
Stage 6 – Debug

Stage 6 is a debug that has many tools that can help you to know where the error is.



If mismatches remain, use Debug tools:

- Double-click the unmatched component to see the RTL vs. netlist schematic.
- Expand blocks to analyze logic differences.



- Global unmatched type indicates SVF was not built correctly.

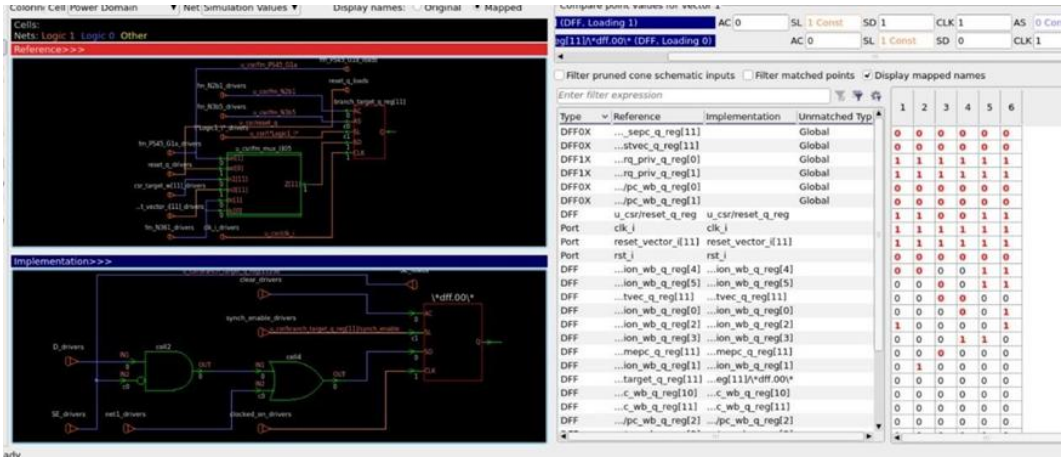


Figure 7: Debug schematic comparison.

This drive link has many files from synopses that explain how to run your formality and how to fix advanced problems.

https://drive.google.com/drive/folders/1S8u4vYmcDT1XTIuADV_w 59Q-vAEbbIfA?usp=sharing

---

Why `set_svf` Timing Matters

- Synthesis transforms (retiming, clock-gating insertion, duplication, resource sharing, etc.) are recorded in the SVF only if it's enabled from the very start of the FC flow (before RTL analyze).
- If SVF is enabled too late, Formality misses critical transformation context → global unmatched type during Stage 6.

Checklist (Copy/Paste for Runs)

- `set_svf <path>.svf` before `analyze/elaborate`.

- Use consistent RTL/tag, libraries (.ndm/.db), and SDC across FC and FM.
- Export hierarchical netlist for easier matching: `-hierarchy all`.
- After writing the netlist, `set_svf -off`.
- In FM Stage 0, load SVF and enable the three clock-gating checks.
- If clock-gate nodes show unmatched at Matching, proceed to Verify (settings should resolve).
- If global unmatched persists → regenerate SVF correctly and rerun.

Notes Aligned to Your Full FC Script

Your full TCL also covers: floorplan, tap/boundary insertion, PG mesh, place_opt, CTS (`clock_opt` stages and shielding), detailed routing, DRC/LVS/mfill signoff, and final GDS/export. Those are excellent for the physical chapters; for the formal checkpoint, the only required deliverables are:

- `./guidFM/riscv_core1.svf` (built with SVF on from the start)
- `./netlist/riscv_core_syn.v` (implementation)
  Everything else remains part of your mainstream P&R/signoff narrative.

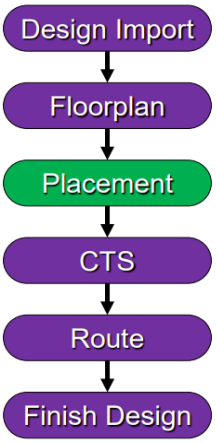
Common Pitfalls & Remedies

- Missing DB/NDM in FM: load the same tech libs used in FC.
- Clock-gating mismatches: ensure the three verification switches are set in FM Stage 0.
- Black boxes: confirm all macro/IP views (.db/.ndm) are present.
- Different tops / mismatched hierarchies: verify `set_top_module` in both FC and FM.

Placement:

◇ What is Placement?

Placement is the physical design process of assigning specific coordinates on the silicon die to each logic cell (also known as a *standard cell*) in a digital chip. After the design has been synthesized into gates and a floorplan has defined the general layout of the chip, placement determines where each gate will physically go within the designated core area. The goal is to position the cells in a way that respects design rules, minimizes wirelength, meets timing constraints, avoids routing congestion, and prepares the design for successful clock tree synthesis and routing. It is a critical step that directly affects the performance, power, and area (PPA) of the final chip.



◇ Why is Placement Important?

Placement is a critical step in physical design because it directly impacts the overall performance, power efficiency, routability, and area of the chip. A poor placement can lead to long and inefficient wires, which cause signal delays, increase power consumption, and make timing closure difficult. It can also create routing congestion, where there isn't enough physical space to connect all the cells properly, resulting in design rule violations or even a failure to complete routing. On the other hand, a well-executed placement places related cells close together, shortens critical paths, reduces power, and ensures enough space for clean, legal routing in later stages. Without proper placement, the chip may not function correctly or could be too slow, too large, or too power-hungry.

◇ Goals of Placement:

1. Placing all standard cells legally and without overlaps
   Cells must fit exactly into placement rows without overlapping, following the physical design rules.
2. Minimizing total wirelength between connected cells
   Shorter wires reduce delay, save space, and lower power consumption.
3. Improving timing by reducing delay on critical paths
   Critical cells are placed closer to improve signal arrival time and meet timing constraints.
4. Minimizing power by shortening nets with high switching activity
   Cells with frequent toggling are grouped to reduce dynamic power.
5. To enable successful routing by avoiding congestion.

🔷 Advantages and Disadvantages of Placement:

Advantages	Disadvantages
Improves timing by placing critical path cells closer together	Sensitive to floorplan quality – bad macro placement can block good layouts
Reduces wirelength, lowering delay and dynamic power	Congestion can arise if cells are packed too tightly
Enables legal routing by leaving enough space and alignment	Tools may miss optimization opportunities if placement constraints are loose
Supports power optimization by grouping high-toggle cells smartly	Placement takes time and may require multiple iterations
Helps achieve design closure in later steps like CTS and routing	Requires accurate constraints and technology data

◇ How to Run Placement in Fusion Compiler:

To run placement in Fusion Compiler, follow these steps in your script:

1. **Coarse Placement:** this step spreads cells throughout the available floorplan area based on logic connectivity. It ensures that cells are close to each other based on the netlist but does not yet guarantee legal placement (cells may overlap or be misaligned).

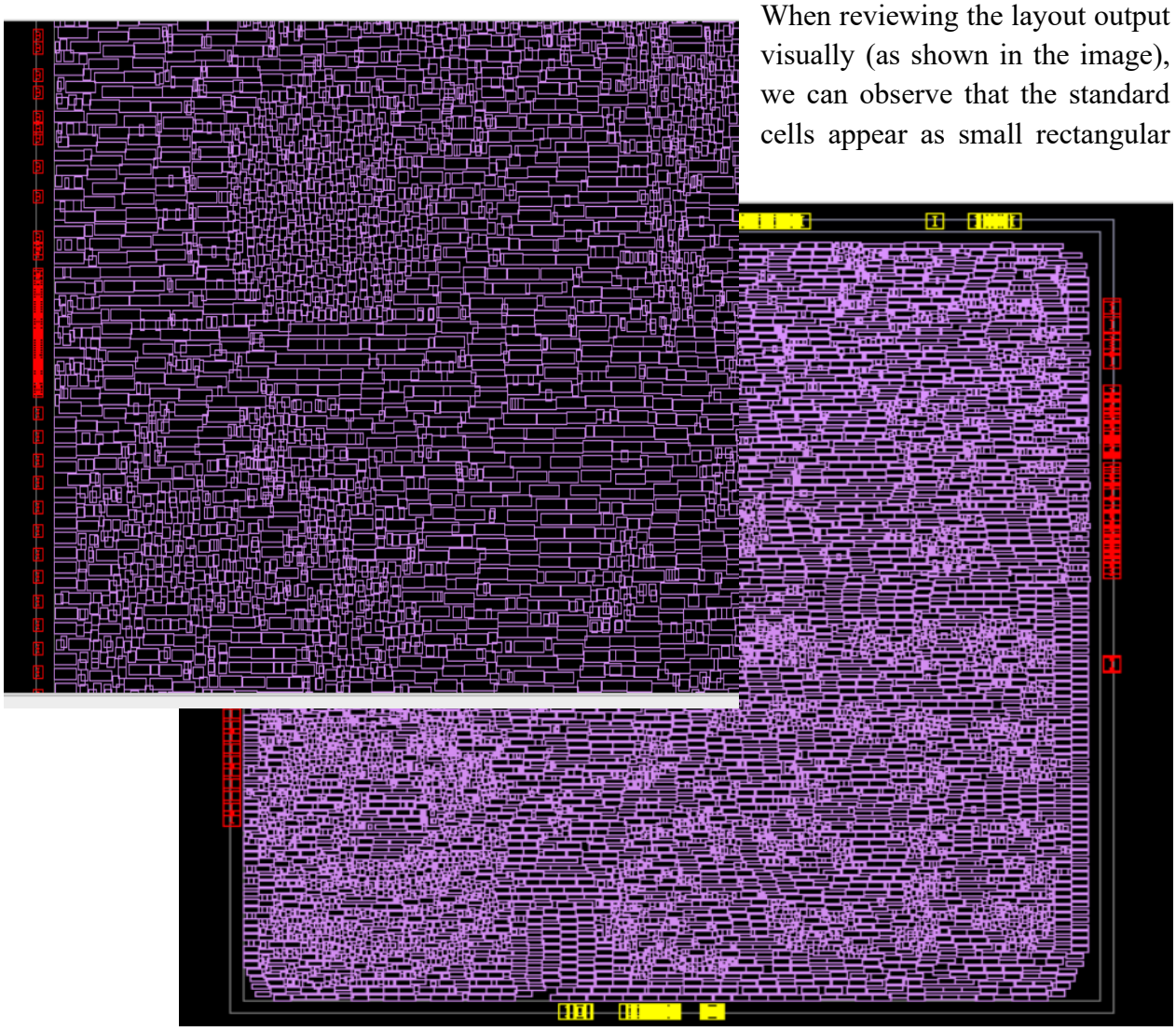
   Command:

   #create_placement

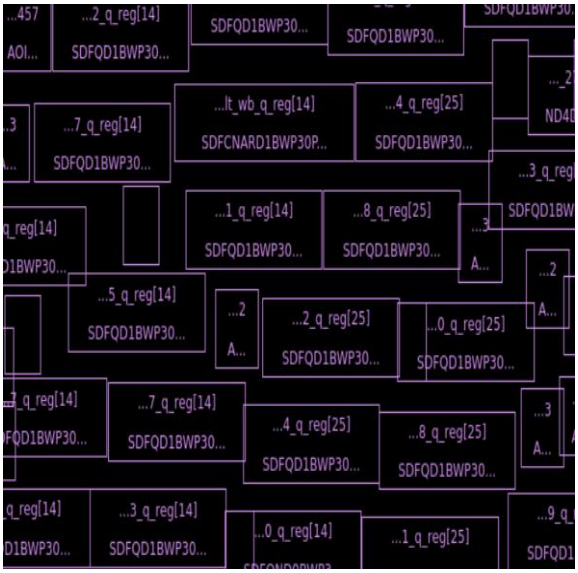
   compile_fusion -to logic_opto

   After executing the command "compile_fusion -to logic_opto", Fusion Compiler performs a set of automated actions which include logic optimization, initial cell placement, and preliminary legalization. This stage is essential for building a physical representation of the netlist so that further timing-driven steps can be applied later in the flow. The placement created at this stage is referred to as coarse placement. This means that the standard cells are distributed across the floorplan in a way that reflects

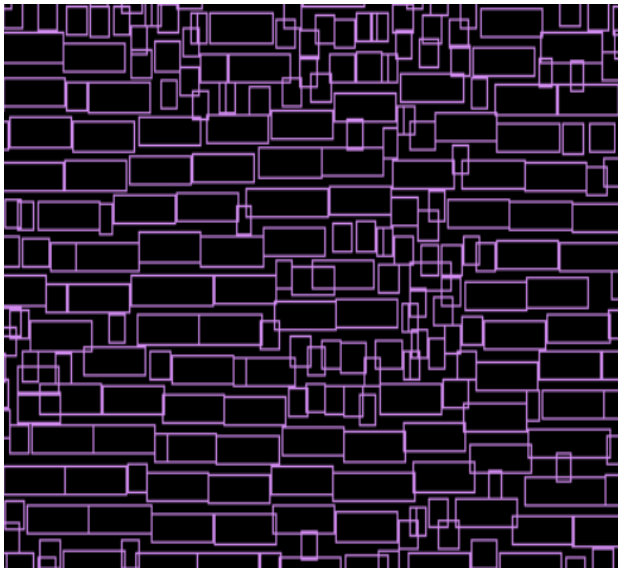
the logical connectivity between them, but without detailed optimization for timing, So, after running this command in fusion compiler you will see the following layout:



When reviewing the layout output visually (as shown in the image), we can observe that the standard cells appear as small rectangular blocks that fill the core area. These cells are aligned along rows and occupy most of the die surface. The placement may appear dense in some areas and sparse in others — this is expected at this stage, as the optimization engine hasn't yet adjusted for timing-critical paths or routing congestion. The purpose here is to create a first-pass layout that the tool can analyze and refine in subsequent steps.



If we zoom in the layout, we can see that the standard cells are placed legally along the rows, but their arrangement is still unoptimized. The cells appear scattered and loosely packed, with irregular spacing and no timing-driven alignment yet — which is typical after coarse placement at the logic_opto stage.



Reports:

1. Timing:

```
                                                  0.03      2.14 f
u_exec/u_alu/add_163/ctmi_31459/Z (XOR3UD1BWP30P140HVT)
                                                  0.05      2.19 f
ctmi_29859/Z (AO22D0BWP30P140HVT)                 0.03      2.22 f
ctmi_29836/ZN (AOI211D1BWP30P140HVT)              0.02      2.25 r
ctmi_29816/ZN (OAI211D1BWP30P140)                 0.03      2.28 f
u_exec/result_q_reg[31]/D (SDFCNARD2BWP30P140HVT)
                                                  0.00      2.28 f
data arrival time                                           2.28

clock clk_i (rise edge)                          10.00     10.00
clock network delay (ideal)                       0.00     10.00
u_exec/result_q_reg[31]/CP (SDFCNARD2BWP30P140HVT)
                                                  0.00     10.00 r
clock uncertainty                                -0.15      9.85
library setup time                               -0.03      9.82
data required time                                          9.82
-------------------------------------------------------------------------
data required time                                          9.82
data arrival time                                          -2.28
-------------------------------------------------------------------------
slack (MET)                                                 7.54


1
fc_shell>
```

2. Area:

```
fc_shell> report_area
****************************************
Report : area
Design : riscv_core
Version: V-2023.12-SP3
Date   : Tue Jan 28 14:51:31 2025
****************************************

Number of ports:                          448
Number of nets:                         10611
Number of cells:                         9690
Number of combinational cells:           7313
Number of sequential cells:              2377
Number of macros/black boxes:               0
Number of buf/inv:                        773
Number of references:                     137

Combinational area:                  5488.69
Buf/Inv area:                         239.02
Noncombinational area:               6364.76
Macro/Black Box area:                   0.00

Total cell area:                    11853.45
1
```

3. Power:

```
  Cell Internal Power    = 2.32e+05 nW ( 83.1%)
  Net Switching Power    = 4.72e+04 nW ( 16.9%)
Total Dynamic Power      = 2.79e+05 nW (100.0%)

Cell Leakage Power       = 1.02e+04 nW


  Attributes
  ----------
      u  -  User defined power group
      i  -  Includes clock pin internal power

Power Group         Internal Power        Switching Power          Leakage Power            Total Power    (   %  )    Attrs
------------------------------------------------------------------------------------------------------------------------------
io_pad                    0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
memory                    0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
black_box                 0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
clock_network             2.21e+05              3.01e+04              5.28e+02              2.51e+05    ( 87.0%)        i
register                  4.44e+03              2.16e+03              3.93e+03              1.05e+04    (  3.6%)
sequential                0.00e+00              0.00e+00              0.00e+00              0.00e+00    (  0.0%)
combinational             6.38e+03              1.49e+04              5.72e+03              2.70e+04    (  9.3%)
------------------------------------------------------------------------------------------------------------------------------
Total                     2.32e+05 nW           4.72e+04 nW           1.02e+04 nW           2.89e+05 nW
1
fc_shell>
```

You can find “how to issue reports” in the appendix.

2. **Legalization:** after the coarse placement, cells are moved slightly to ensure they are placed in **legal sites**: proper rows, correct spacing, and no overlaps.

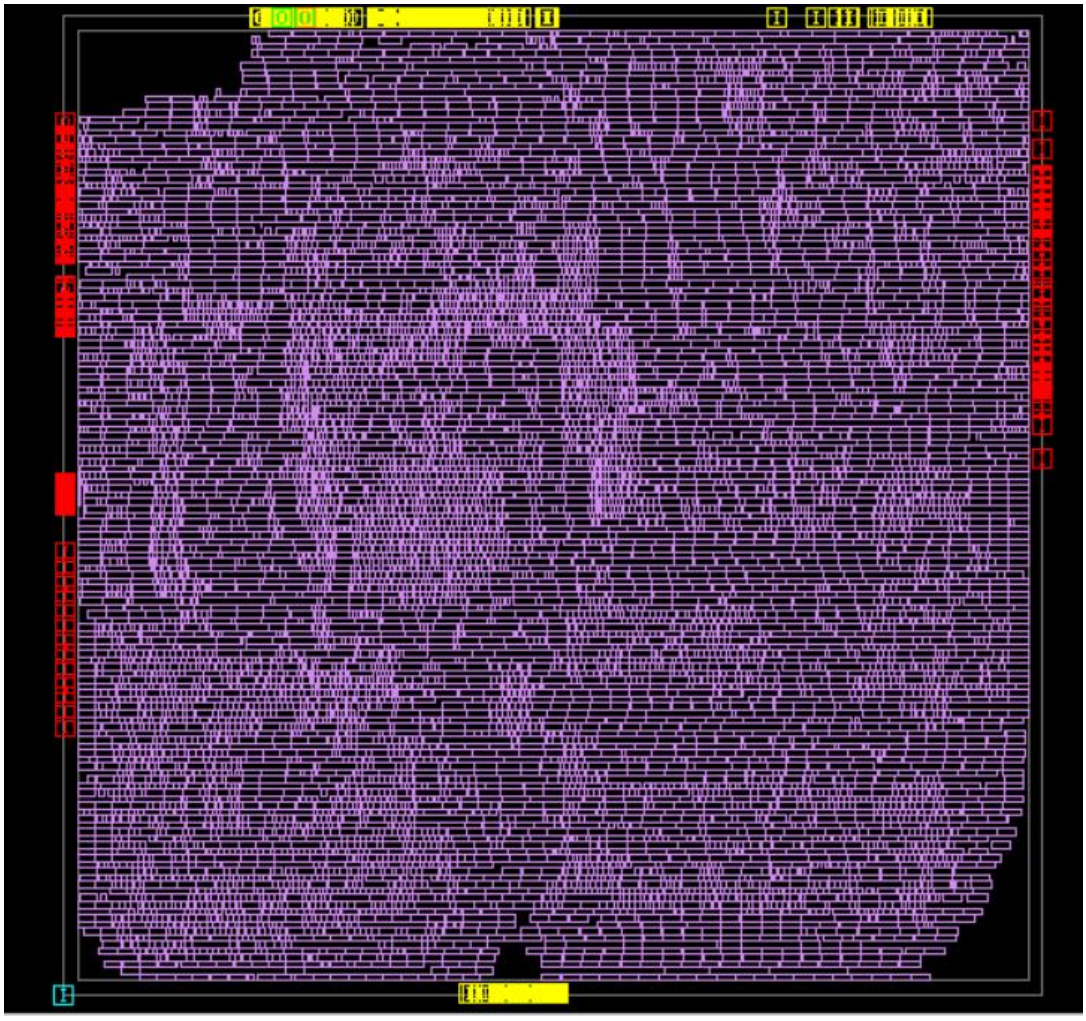
Command:

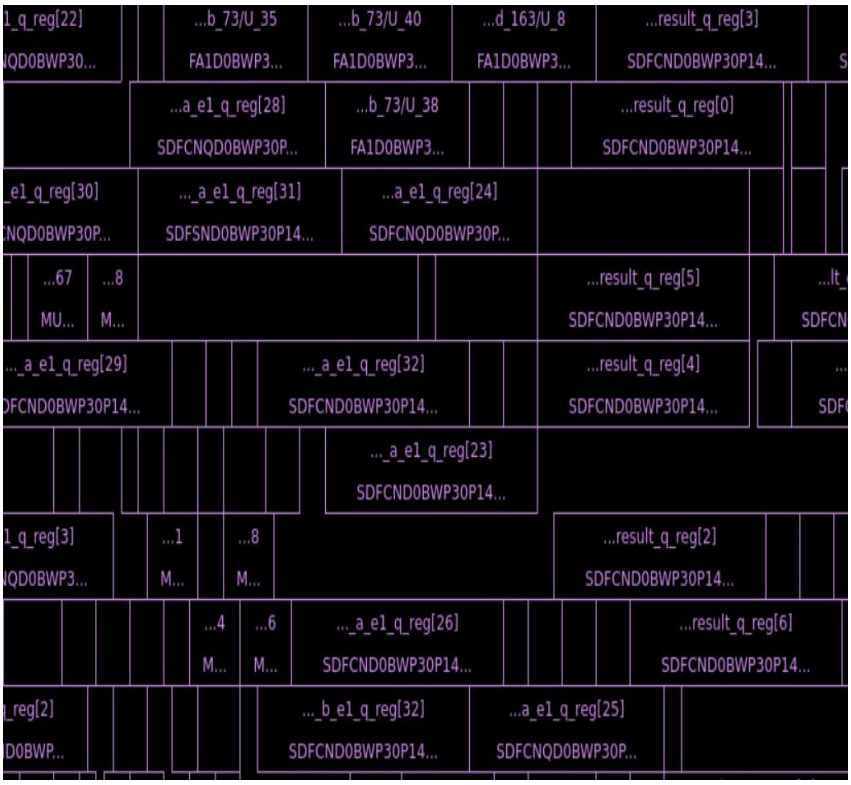
#legalize_placement

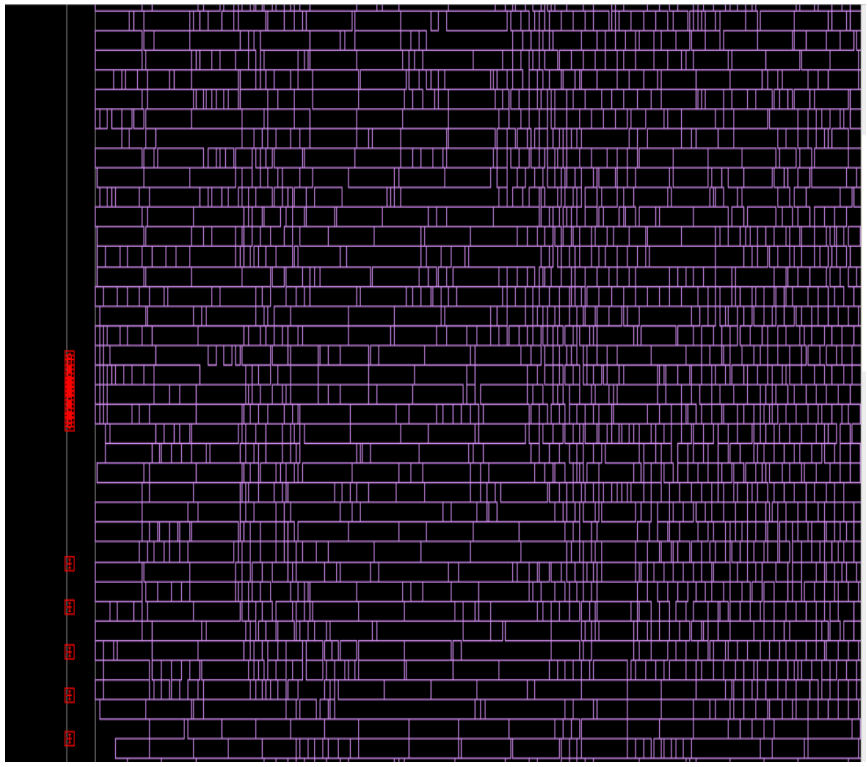
compile_fusion -to final_opto

```
#legalize_placement
compile_fusion -to final_opto
```

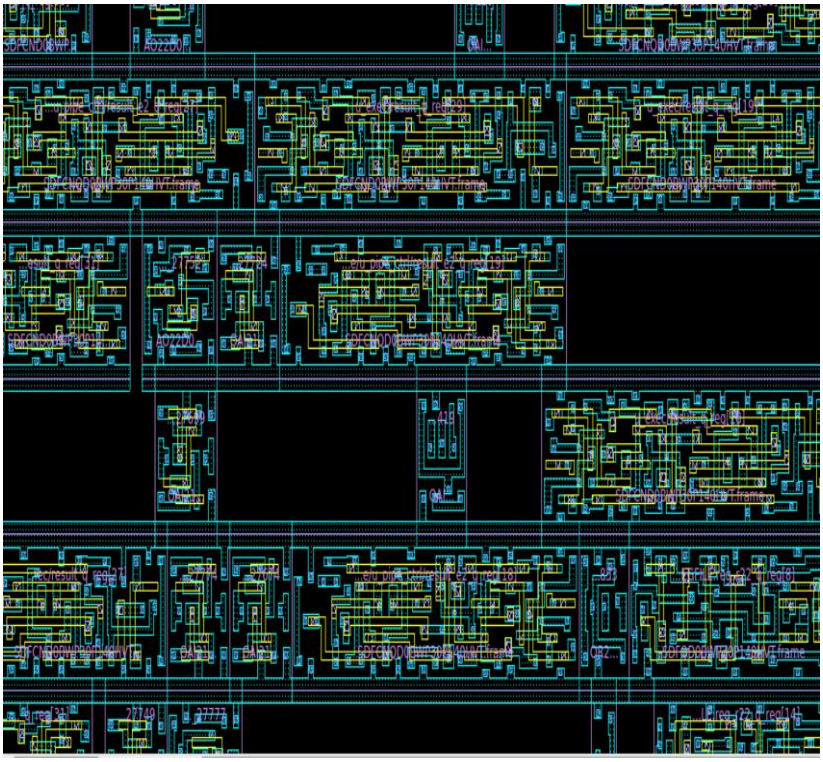
This stage performs a timing-driven optimization of the placement and completes the legalization of all cells in the design. During this step, Fusion Compiler refines the position of each cell based on real timing analysis, congestion information, and power estimates. The result will be more compact, organized, and performance-aware placement. Cells are moved slightly to ensure they are aligned with legal sites, meaning that they sit on proper rows, have no overlaps, and meet the spacing and orientation rules required by the technology library.



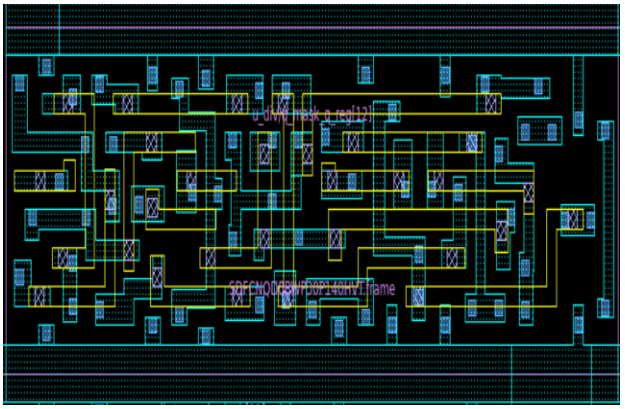




If we zoom into the layout after running compile_fusion -to final_opto, we can clearly observe that the standard cells are now well-organized and aligned. Each cell is placed neatly within a legal placement row, and there are no overlaps between adjacent cells. The gaps between cells are minimal and consistent, reflecting an efficient use of area. Additionally, we can see that cells with related functionality—such as registers or logic gates from the same module—tend to be grouped together. This improved structure not only ensures legality but also enhances the routing quality and timing of the design in later stages.



These layouts show clean, compact rows with well-aligned cells and visible metal routing tracks forming internal connections between transistors, reflecting a placement-ready netlist for CTS and routing stages.



Reports:

1. Timing:

```
                                                  0.04      3.65 r
ctmi_2706/ZN (MAOI222D0BWP30P140HVT)              0.04      3.69 f
ctmi_2705/ZN (MAOI222D0BWP30P140HVT)              0.05      3.74 r
ctmi_2704/ZN (MAOI222D0BWP30P140HVT)              0.04      3.78 f
ctmi_2709/ZN (XNR3UD0BWP30P140HVT)                0.04      3.82 r
ctmi_30320/ZN (AOI211D0BWP30P140HVT)              0.03      3.85 f
ctmi_30319/ZN (AOI32D0BWP30P140HVT)               0.03      3.88 r
ctmi_30318/ZN (INR4D0BWP30P140HVT)                0.04      3.93 r
ctmi_30315/ZN (OAI211D0BWP30P140HVT)              0.06      3.99 f
u_exec/result_q_reg[0]/D (SDFCND0BWP30P140HVT)    0.00      3.99 f
data arrival time                                           3.99

clock clk_i (rise edge)                          10.00     10.00
clock network delay (ideal)                       0.00     10.00
u_exec/result_q_reg[0]/CP (SDFCND0BWP30P140HVT)
                                                  0.00     10.00 r
clock uncertainty                                -0.15      9.85
library setup time                               -0.04      9.81
data required time                                          9.81
-------------------------------------------------------------------------
data required time                                          9.81
data arrival time                                          -3.99
-------------------------------------------------------------------------
slack (MET)                                                 5.82


1
fc_shell>
```

2. Area:

```
fc_shell> report_area
****************************************
Report : area
Design : riscv_core
Version: V-2023.12-SP3
Date   : Tue Jan 28 15:16:38 2025
****************************************

Number of ports:                          269
Number of nets:                         10850
Number of cells:                         9961
Number of combinational cells:           7581
Number of sequential cells:              2380
Number of macros/black boxes:               0
Number of buf/inv:                        688
Number of references:                      78

Combinational area:                   5295.28
Buf/Inv area:                          180.05
Noncombinational area:                6355.44
Macro/Black Box area:                    0.00

Total cell area:                     11650.72
1
fc_shell>
```

3. Power:

```
 Attributes
 ----------
     u  -  User defined power group
     i  -  Includes clock pin internal power

Power Group         Internal Power       Switching Power         Leakage Power           Total Power    (   %  )    Attrs
--------------------------------------------------------------------------------------------------------------------------
io_pad                  0.00e+00             0.00e+00              0.00e+00             0.00e+00    (  0.0%)
memory                  0.00e+00             0.00e+00              0.00e+00             0.00e+00    (  0.0%)
black_box               0.00e+00             0.00e+00              0.00e+00             0.00e+00    (  0.0%)
clock_network           1.90e+05             2.23e+04              5.56e+02             2.13e+05    ( 89.2%)        i
register                4.12e+03             1.06e+03              3.28e+03             8.46e+03    (  3.5%)
sequential              0.00e+00             0.00e+00              0.00e+00             0.00e+00    (  0.0%)
combinational           4.34e+03             1.10e+04              2.03e+03             1.73e+04    (  7.3%)
--------------------------------------------------------------------------------------------------------------------------
Total                   1.98e+05 nW          3.43e+04 nW           5.87e+03 nW          2.39e+05 nW
1
fc_shell>
```

🔷 Comparison: compile_fusion -to logic_opto vs compile_fusion -to final_opto:

Aspect	After logic_opto	After final_opto
Timing (Slack)	Slack = 7.54 ns. Indicates timing is met, but optimization is still rough.	Slack = 5.82ns. Still met, but optimized considering clock uncertainty and physical placement.
Data Arrival	Arrival time = -2.28 ns, likely due to lack of placement and path balancing.	Arrival time = -3.99 ns, indicates tighter placement and faster critical paths.
Total Cell Area	~11,853 µm² (reported via report_area)	~11,650µm² (reported via report_area)

Power (Total)	2.89×10^5 nW, with clock network 87%, indicating early stage high switching cost.	2.39×10^5 nW, lower total power after timing-aware placement and resizing.
Power Breakdown	Power dominated by clock network, registers, and combinational logic.	Better power distribution; lower leakage and switching due to optimization.
Placement State	Cells are coarsely placed, spacing is irregular, mainly for estimation.	Cells are legally placed and tightly packed, ready for CTS and routing.

➔ We can conclude that the placement process successfully transitioned from an early estimation stage to a fully optimized and legalized layout. Timing was met in both stages, but final placement improved path delays and reduced arrival time significantly. The total cell area remained relatively stable, with minor reductions due to gate sizing and buffer insertion.

One of the most significant improvements was observed in the power report, where total power consumption decreased by about 17%, mainly due to better placement, reduced switching, and improved balance in power distribution. In addition, the layout became more compact and legally structured, which is essential for the success of upcoming steps.

◇ Preparing for Reliable Power Delivery:

Once all standard cells are legally placed in the chip area during the placement stage, the next step is to ensure that each of these cells receives stable power and ground. This is done by creating a power grid, which must be built after placement to account for the actual cell locations, and before clock tree synthesis or routing, as it defines key physical structures that later stages must work around.

🔷 What is the Power Grid?

In an integrated circuit (IC), the power grid is the system of metal wires that delivers power (VDD) and ground (VSS) to every part of the chip.

You can think of it like the electricity grid in a city: it connects the power source to every building, making sure each one gets enough electricity to function.

On a chip, the power grid must deliver stable and consistent voltage to millions of tiny logic gates that need power to switch on and off very quickly. These gates are spread all across the chip, and they can't work properly if the power is weak or unstable.

🔷 Why Do We Need a Power Grid?

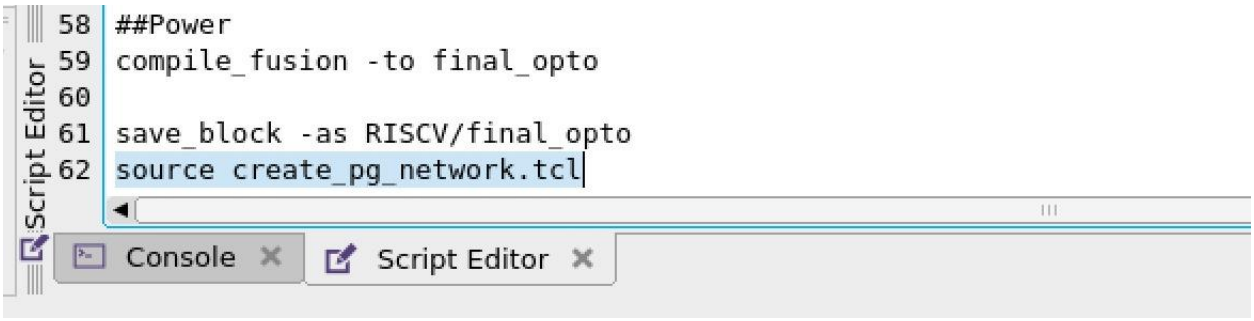
The main goal of the power grid is to ensure reliable power delivery. Without a strong grid:

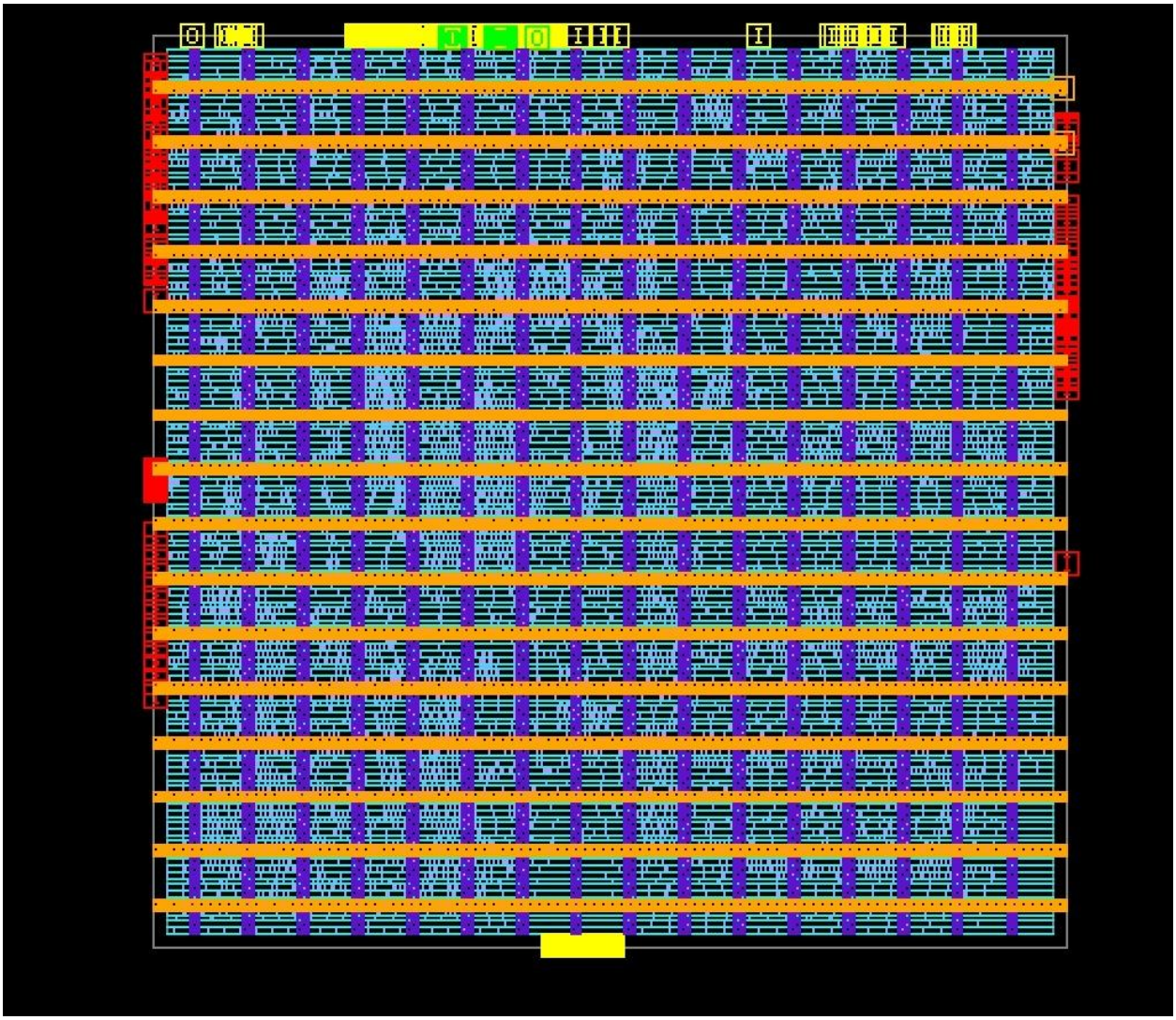
- Parts of the chip may not get enough voltage, leading to failures.
- High current in narrow wires may damage the chip due to electromigration (metal atoms get pushed and wear out the wire).
- Timing performance gets worse if voltage drops (known as IR drop).
- Later stages in the flow, like clock tree synthesis (CTS) and routing, rely on knowing where the power lines are, so they can avoid crossing them or shorting them.

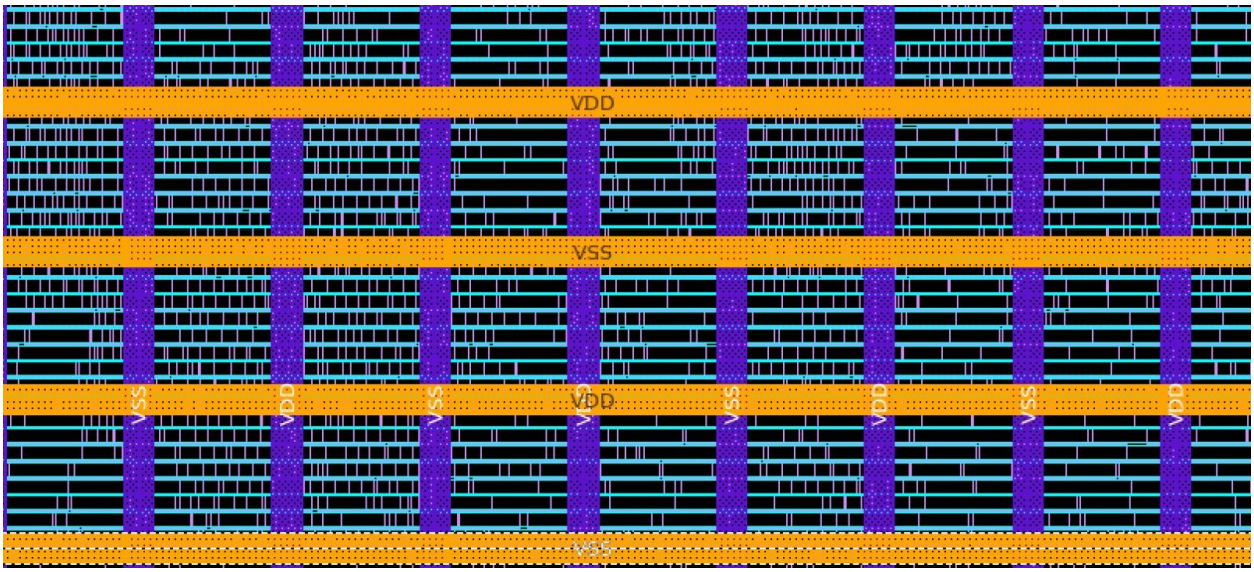
🔷 How Does It Help in the Next Design Stages?

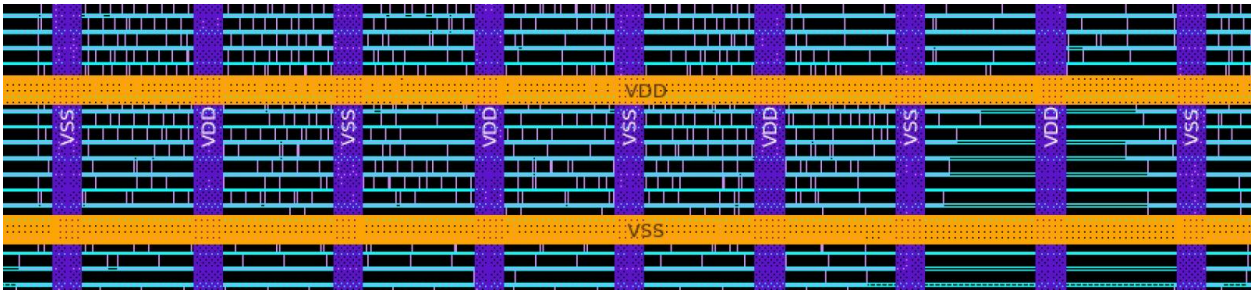
1. **Clock Tree Synthesis (CTS):** The clock signal must reach all flip-flops evenly. A well-designed power grid ensures that the flip-flops receive stable voltage, so the clock can work without issues.
2. **Routing:** When connecting all signals, the router needs to avoid crossing over power wires in illegal ways. The grid provides a clear structure that the router can plan around.
3. **Timing and Power Analysis:** In the final analysis, tools check how much power each part consumes, how fast signals travel, and whether any violations exist. Without a power grid, those results would be inaccurate or meaningless.
4. **IR Drop and EM Signoff:** The power grid is analyzed at the end of the flow to make sure no part of the chip suffers from voltage drop or wire damage due to current stress.

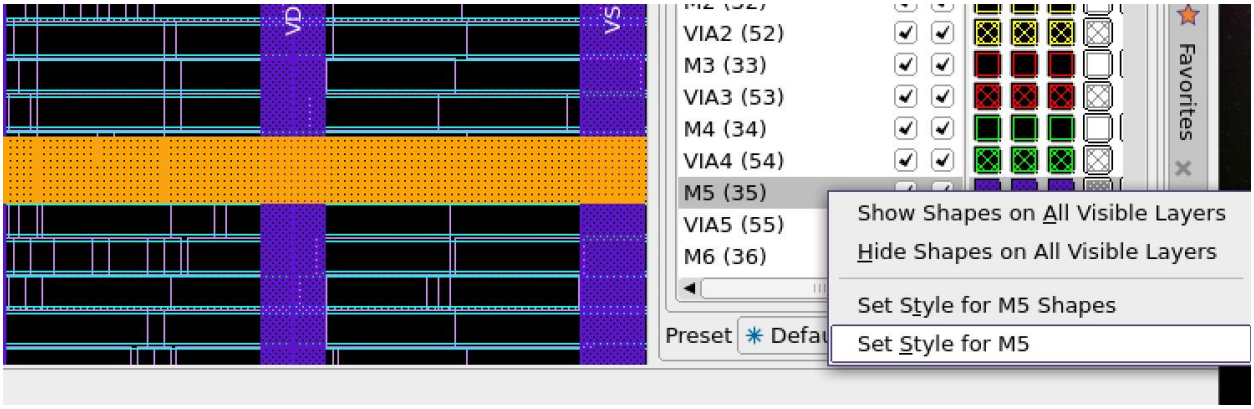
🔷 How to add Power Grid to the chip in Fusion Compiler?

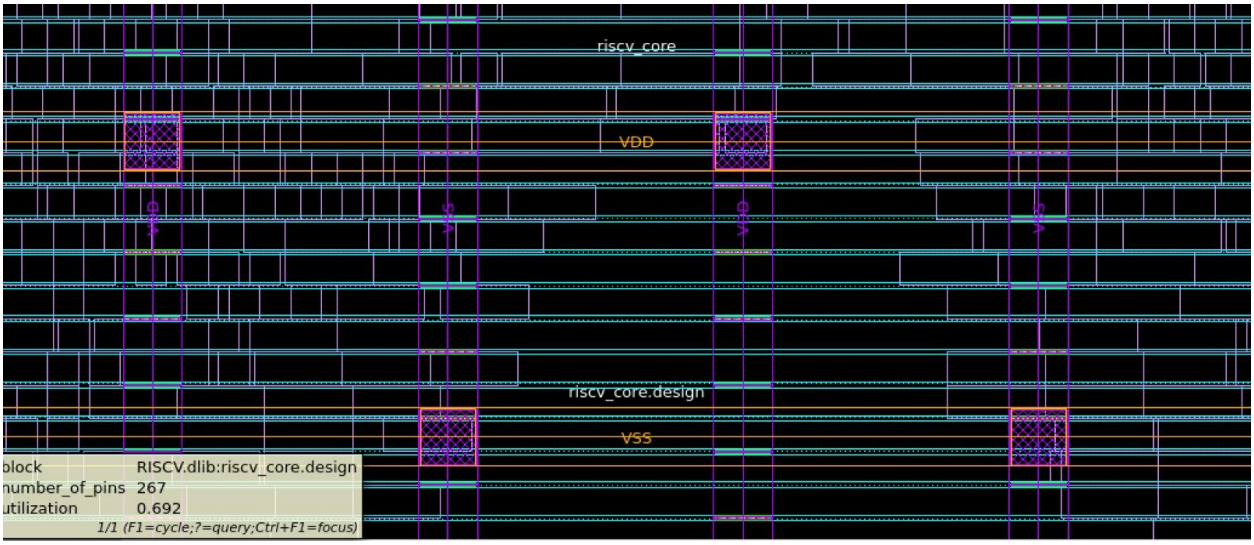


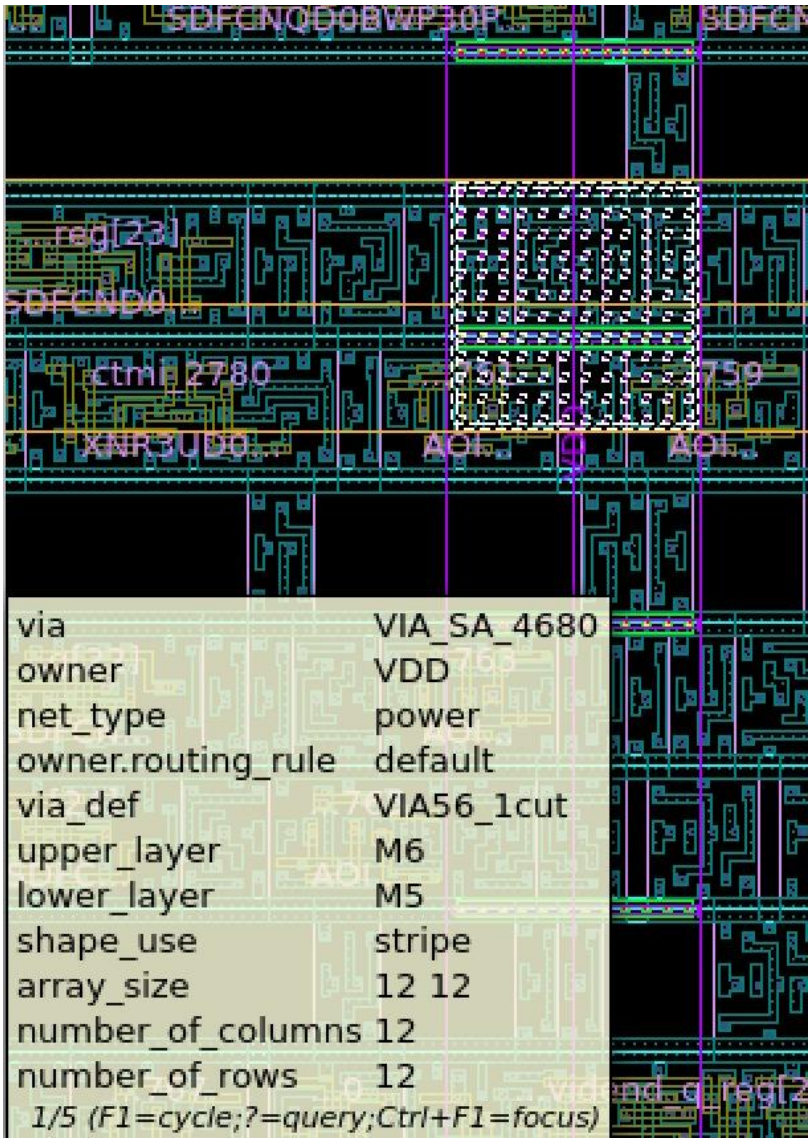


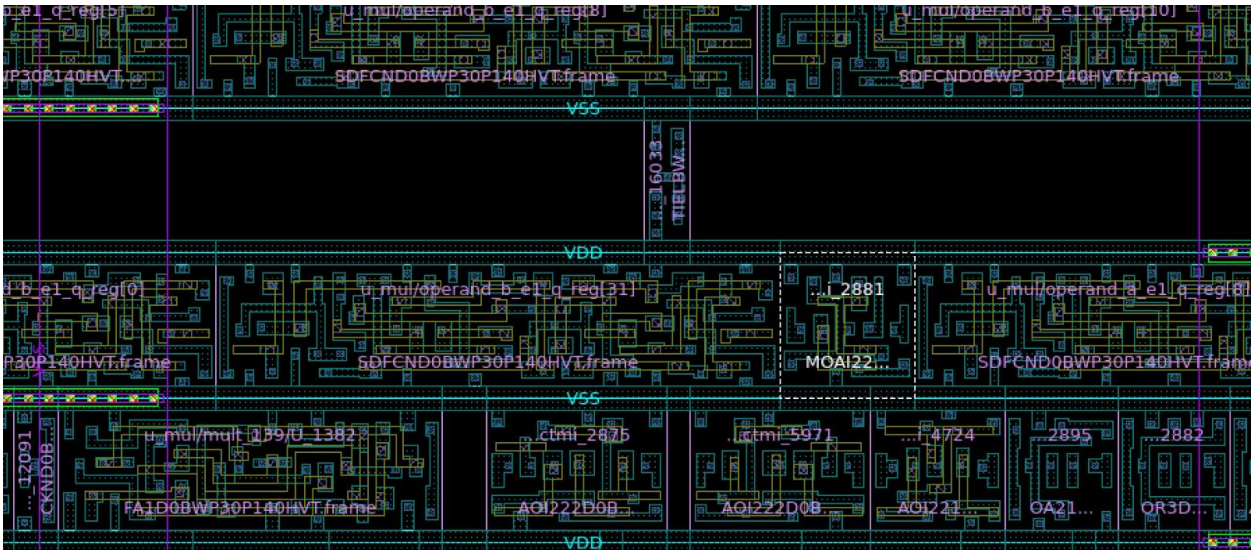

VDD
VSS
VDD
VSS
VDD
VSS
VDD
VSS
VDD
VSS

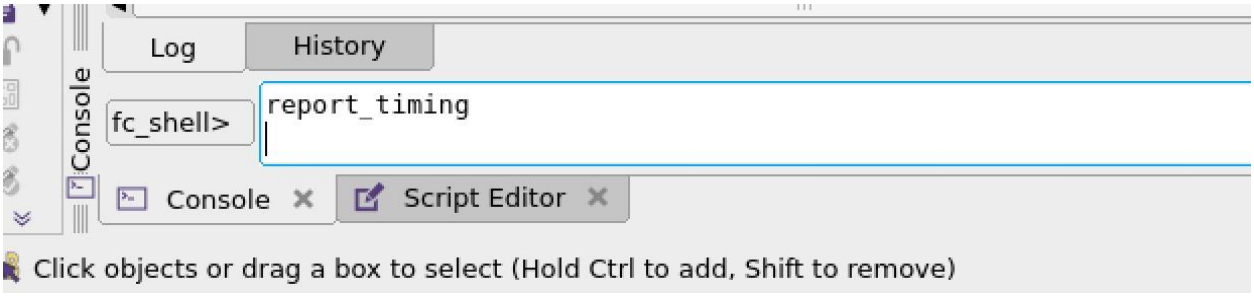

VDD
VSS
VDD
VSS
VDD
VSS
VDD
VSS
VDD
VSS
VSS


VDD
VSS
VIA2 (52)
M3 (33)
VIA3 (53)
M4 (34)
VIA4 (54)
M5 (35)
VIA5 (55)
M6 (36)
Favorites
Show Shapes on All Visible Layers
Hide Shapes on All Visible Layers
Set Style for M5 Shapes
Set Style for M5
Preset


riscv_core
VDD
VSS
VDD
VSS
VDD
VSS
riscv_core:design
VSS
block RISCV.dlib:riscv_core.design
number_of_pins 267
utilization 0.692
1/1 (F1=cycle;?=query;Ctrl+F1=focus)


SDFCNQD0BWP30P...
reg[23]
SDFCND0...
ctmi_2780
XNR3UD0...
AOI...
AOI...
via VIA_SA_4680
owner VDD
net_type power
owner.routing_rule default
via_def VIA56_1cut
upper_layer M6
lower_layer M5
shape_use stripe
array_size 12 12
number_of_columns 12
number_of_rows 12
1/5 (F1=cycle;?=query;Ctrl+F1=focus)

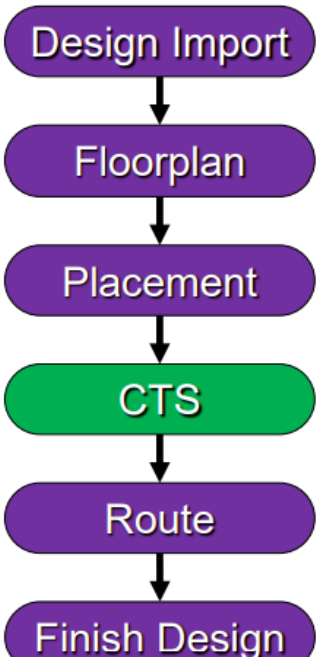

SDFCND0BWP30P140HVT.frame
VSS
VDD
u_mul/operand_b_e1_q_reg[31]
MOAI22...
u_mul/mult_139/U_1382
ctmi_2875
ctmi_5971
AOI222D0B
AOI222D0B
AOI221...
OA21...
OR3D...
FA1D0BWP30P140HVT.frame




```
****************************************
Report : timing
        -path_type full
        -delay_type max
        -max_paths 1
        -report_by design
Design : riscv_core
Version: V-2023.12-SP3
Date   : Sat Jan 18 13:46:31 2025
****************************************

  Startpoint: u_fetch/skid_valid_q_reg (rising edge-triggered flip-flop clocked by clk_i)
  Endpoint: u_exec/result_q_reg[0] (rising edge-triggered flip-flop clocked by clk_i)
  Mode: FUNC
  Corner: Fast
  Scenario: FUNC_Fast
  Path Group: clk_i
  Path Type: max
```

```
  -----------------------------------------------------------------------
  data required time                                              9.81
  data arrival time                                              -4.25
  -----------------------------------------------------------------------
  slack (MET)                                                     5.57
```

CTS (Clock Tree Synthesis) :



What is CTS?

Clock Tree Synthesis (CTS) is a critical step in the physical design flow of digital VLSI circuits, responsible for distributing the clock signal from a single source to all sequential elements (like flip-flops and registers) in the chip. Unlike regular signal routing, CTS must carefully balance skew, insertion delay, and power consumption to ensure reliable and synchronized operation. It builds a buffered tree structure to minimize timing issues such as jitter and skew, which can significantly affect setup and hold constraints. CTS aims to deliver a non-ideal but optimized clock network that meets timing, signal integrity, and design rule requirements.

Why is CTS Necessary?

Clock Tree Synthesis (CTS) is essential because modern digital circuits rely on a precisely timed clock signal to synchronize all sequential elements. Without CTS, we would assume an ideal clock, but in reality, physical constraints introduce skew, jitter, and insertion delays that can severely affect timing margins and lead to functional errors. A naïve approach—routing the clock like any other net—would result in poor timing, excessive power, and signal integrity problems. CTS creates a balanced and optimized clock distribution network, reducing skew, ensuring signal integrity, and meeting strict timing and power constraints. Therefore, CTS plays a critical role in enabling the chip to function reliably at high speeds.

Goals of CTS:

- Minimize clock skew – Reduce timing differences between clock arrival at various flip-flops.
- Meet target insertion delay – Ensure the clock reaches all sinks within a defined delay window.
- Satisfy design rule constraints (DRV) – Respect max fanout, transition time, and wire capacitance.
- Enable reliable setup and hold timing – Support correct data capture and prevent violations.
- Improve signal integrity – Avoid glitches and maintain stable clock transitions.
- Reduce power consumption – Minimize switching activity and buffer count.

- Optimize for area – Use efficient clock routing and buffering to reduce layout overhead.

Constraints of CTS:

- Maximum Transition – The slew rate (rise/fall time) of the clock signal must stay within limits to ensure proper flip-flop operation.
- Maximum Load Capacitance – The total capacitance on clock nets must be controlled to avoid delay and power issues.
  Maximum Fanout – Each clock buffer must not drive more loads than allowed to avoid signal degradation.
- Maximum Buffer Levels – Limit the depth of buffering stages to reduce insertion delay and complexity.
- Routing Constraints – Clock wires often require non-default routing rules (e.g., shielding, spacing).
- Skew and Jitter Tolerance – The design must tolerate or compensate for variations in clock arrival time and period.

Advantages and Disadvantages of CTS:

Advantages	Disadvantages
Ensures synchronized clock arrival and reduced skew	Increases power consumption due to high switching activity and buffer count
Reduces insertion delay, enabling high-speed operation	Occupies significant routing area, especially in dense designs
Improves signal integrity by controlling slew and reducing noise	Requires careful buffer planning and tuning for optimal performance
Supports timing closure by aligning setup/hold across the chip	Adds complexity to the design and increases tool runtime
Adheres to design constraints (fanout, cap, transition)	Trade-offs between power, area, and skew may be hard to balance

How to Run Clock Tree Synthesis (CTS) in Fusion Compiler

Running the CTS Script:

In our design flow, all steps of **Clock Tree Synthesis (CTS)** are collected inside one Tcl script. To execute the CTS stage, run the following command in Fusion Compiler:

```
source/project/tsmc28mmwave/users/leenawattad/ws/riscv/CTS.tcl
```

Note: Replace the path above with the actual path where you saved your own `CTS.tcl` file in your project directory.

What the CTS Script Contains

The `CTS.tcl` script fully automates the clock tree synthesis flow. It includes the creation of routing rules for clock nets, setting skew constraints, building and routing the tree, adding shielding, reconnecting power nets, verification, and saving the design state.

1. Create NDR (Non-Default Routing) Rule

```
create_routing_rule CLK_NDR \
  -default_reference_rule \
  -multiplier_width 2 \
  -spacings {M2 0.052 M3 0.052 M4 0.08 M5 0.08} \
  -shield \
  -shield_spacings {M2 0.026 M3 0.026 M4 0.04 M5 0.04} \
  -snap_to_track
```

- Defines a special routing rule (`CLK_NDR`) only for clock nets.
- Doubles the width of the wires to reduce resistance and delay.
- Sets custom spacings on M2–M5 to reduce coupling noise.
- Enables shielding with defined distances to improve signal integrity.
- Ensures wires are aligned to legal routing tracks.

2. Restrict Routing Layers

```
set_clock_routing_rules -rules CLK_NDR \
  -min_routing_layer M2 \
  -max_routing_layer M5
```

- Limits clock routing between metal layers M2 (minimum) and M5 (maximum).
- Balances performance (low RC) with routing congestion.

3. Set Target Skew

```
set_clock_tree_options -clocks [all_clocks] -target_skew
0.1
```

- Defines a maximum clock skew of 0.1 ns.
- Forces the tool to balance the clock tree and reduce timing variations across sinks.

4. Clock Optimization Flow

```
get_clocks
clock_opt -list_only
clock_opt -to build_clock
clock_opt -from build_clock -to route_clock
clock_opt -to final_opto
```

Lists all available clocks in the design.

- `clock_opt -list_only`: shows the different stages available.
- `build_clock`: builds the initial tree, inserts buffers, and balances fanouts.
- `route_clock`: performs clock routing according to the NDR rule.
- `final_opto`: optimizes and legalizes the clock tree for timing and power.

5. Remove Global Routes and Add Shielding

```
remove_routes -global_route

set clock_nets [get_nets -hierarchical -filter "net_type == clock"]

create_shields -nets ${clock_nets} -with_ground VSS
```

- Removes previous global routes so only the clock tree remains visible.
- Identifies all clock nets and adds ground (VSS) shielding wires for noise immunity.

6. Reconnect Power Nets and Verification

```
connect_pg_net -net VDD [get_pins -hierarchical */VDD]

connect_pg_net -net VSS [get_pins -hierarchical */VSS]

check_legality

report_congestion

report_utilization
```

- Reconnects all VDD/VSS nets across the design.
- Runs legality checks to confirm no violations exist.
- Reports routing congestion and utilization after CTS.

7. Save CTS Results

```
get_blocks -all

list_blocks

save_lib

save_block -as RISCV1/CTS
```

- Saves the updated library and design database.
- Stores the CTS result under the block name `RISCV1/CTS`.

This image shows the physical layout of the chip after the updated Clock Tree Synthesis (CTS) stage has been completed in Fusion Compiler. With the improved script, the tool applies a dedicated non-default routing (NDR) rule for clock nets, uses specific routing layers (M2–M5), enforces shielding with VSS, and targets a skew of 0.1 ns. These constraints ensure a robust and balanced clock tree across the entire design.

In this updated layout view:

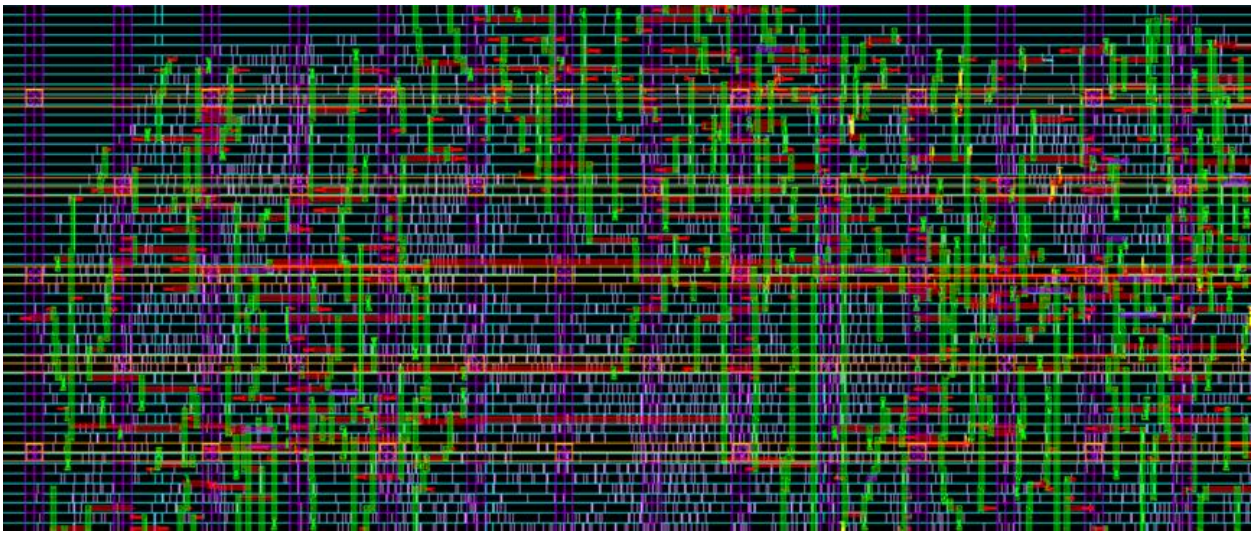
- **Colored metal tracks** represent routed signals across multiple layers (M1, M2, …, M6), with clock nets following the custom NDR rule.

- **Shielded regions** show where ground (VSS) shielding wires have been added to protect clock nets from noise and crosstalk.

- **Dense clock buffers and nets** can be observed in areas with high register activity, reflecting the optimized CTS distribution.

- The overall structure reflects both the initial floorplan and the effects of clock-aware placement, routing, and shielding applied in the updated flow.

This visualization highlights how the enhanced CTS process improves timing quality, signal integrity, and clock distribution reliability, while still meeting the design rule requirements.



In these zoomed-in layout views, we can observe the **fine-grained routing structures** generated during the Clock Tree Synthesis (CTS) stage after applying the updated script. These images focus on small regions of the chip, where clock buffers, registers, and routing tracks are clearly visible.

Key observations in the updated layout:

- **Clock buffers and gates** (e.g., `CLK`, `reg[]`) are explicitly visible, annotated with their cell names, and connected to the synthesized tree.
- **Colored wires**:

  Green and red tracks represent clock and data routing across metal layers.

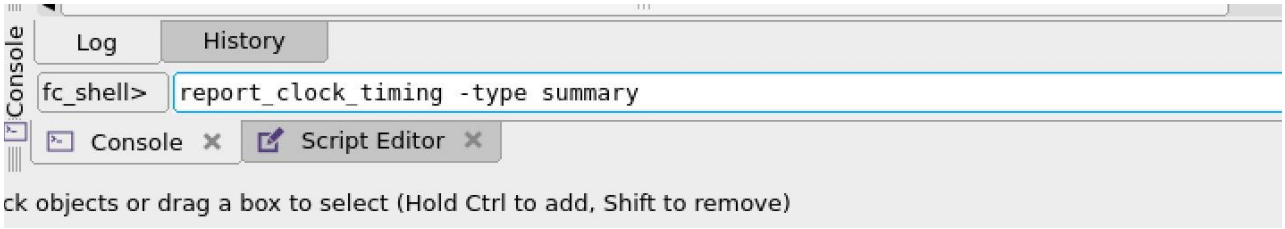
  Yellow paths highlight critical vertical and horizontal interconnects.

- **Vertical and horizontal lines** show the precise routing paths.
- **Square via markers** represent vertical connections between layers, enabling signal transitions from one metal layer to another.
- The CTS engine has intelligently placed buffers and inserted shielding wires to minimize skew, delay, and crosstalk.

This view highlights the complexity of clock tree optimization and the precision required to ensure consistent clock arrival across the entire design. The result is a tightly coordinated network that supports reliable high-speed operation.

Clock Timing Report After CTS:

After completing the Clock Tree Synthesis (CTS) stage, it is critical to analyze the quality and behavior of the clock network to ensure it meets timing constraints. In Fusion Compiler, this is done using the following command in the console:




```
report_clock_timing -type summary
```

This command generates a summary report that includes key timing characteristics of the clock, such as:

- Maximum and minimum setup/hold latencies
- Clock skew values across different flip-flops and clock paths
- Transition times (rise/fall) of the clock signal at critical points
- Launch and capture edge delays under different corners (e.g., slow/fast)

In our example shown:

```
u_exec/result_q_reg[0]/D (SDFCND0BWP30P140HVT)    0.00      4.33 r
data arrival time                                           4.33

clock clk_i (rise edge)                          10.00     10.00
clock network delay (propagated)                  0.14     10.14
u_exec/result_q_reg[0]/CP (SDFCND0BWP30P140HVT)
                                                  0.00     10.14 r
clock uncertainty                                -0.15      9.99
library setup time                               -0.02      9.97
data required time                                          9.97
-----------------------------------------------------------------
data required time                                          9.97
data arrival time                                          -4.33
-----------------------------------------------------------------
slack (MET)                                                 5.64
```

- The clock domain analyzed is `clk_i` at the rising edge.
- The report indicates a clock network delay (propagated) of 0.14 ns, which reflects the insertion delay from the root clock pin to the destination register.
- The endpoint register is `u_exec/result_q_reg[0]/CP`, where the clock arrives for capture.
- The clock uncertainty is –0.15 ns, representing additional margins for jitter and variations.
- The library setup time is –0.02 ns, which corresponds to the intrinsic setup requirement of the cell.
- The report shows a data required time of 9.97 ns compared to a data arrival time of –4.33 ns.
- The resulting slack (MET) is +5.64 ns, meaning that setup timing is met with a safe margin.

This analysis confirms that the CTS achieved its objectives:

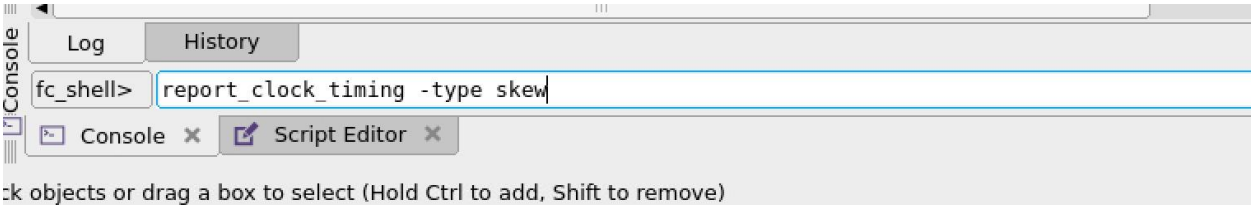
- Minimizing clock insertion delay (0.14 ns).
- Accounting for real-world uncertainties (–0.15 ns).

- Meeting setup timing constraints with significant positive slack (+5.64 ns).

Such reports are essential for validating that the clock tree is balanced, reliable, and capable of supporting the target operating frequency across all design corners.

Analyzing Clock Skew After CTS:

After building the clock tree using CTS, one of the most important timing characteristics to validate is clock skew — the difference in clock arrival times between various flip-flops or registers in the design. To extract this information in Fusion Compiler, the following command is used:




```
report_clock_timing -type skew
```

This command reports the worst-case clock skew between different clock pins in the design. The report helps verify that the CTS stage successfully minimized skew, which is essential for ensuring setup and hold timing are not violated.

```
-
fc_shell> report_clock_timing -type skew
****************************************
Report : clock timing
        -type skew
        -nworst 1
        -setup
Design : riscv_core
Version: V-2023.12-SP3
Date   : Tue Jan 28 12:41:05 2025
****************************************

  Mode: FUNC
  Clock: clk_i

  Clock Pin                                          Latency      Skew             Corner
-------------------------------------------------------------------------------------------------------
  u_csr/u_csrfile/csr_mcycle_h_q_reg[5]/CP             0.17              rp-+       Slow
  u_csr/rd_result_e1_q_reg[5]/CP                       0.05      0.12    rp-+       Slow

-------------------------------------------------------------------------------------------------------
1
fc_shell>
```

In our example:

- The tool compares two clock pins (`u_csrfile/.../reg[5]/CP`) that belong to the clock domain `clk_i`.
- It reports their latency (time for the clock to arrive) and the skew between them, which in this case is 0.12 ns.
- The analysis is done under the Slow corner, representing one of the worst-case PVT (process-voltage-temperature) scenarios.

Why this matters:
Minimizing skew is critical to avoid timing violations:

- Positive skew can help with setup but hurt hold time.
- Negative skew can cause setup violations.
  This report helps identify whether the clock tree needs further balancing or if buffers should be repositioned.

It is common to combine this report with:

- `report_clock_timing -type summary` for overall stats
- `report_timing` for path-based setup/hold analysis
- `report_clock_balance_points` in case of useful skew control

routing:

What is routing?

Routing is the stage in VLSI physical design where all logical connections (nets) are implemented as physical wires on the chip. It transforms the abstract netlist into real, manufacturable metal paths that connect placed cells.

Routing is divided into:

- Global Routing – Plans approximate wire paths and metal layers, avoiding congestion.
- Detailed Routing – Generates exact wire shapes, assigns tracks, adds vias, and fixes design rule violations.

It must meet timing, signal integrity, and manufacturing rules, making it one of the most critical and complex steps in the design flow.

Why is Routing Important?

Routing is essential because it directly impacts the functionality, performance, and manufacturability of the chip. Even if the logical design is correct, poor routing can lead to:

- Timing violations, caused by long or unbalanced wire paths
- Signal integrity issues, such as crosstalk and noise
- Physical design rule violations that prevent fabrication
- Increased power consumption due to unnecessary wire length and vias
- Reduced yield, from manufacturing defects like shorts or opens

A well-executed routing stage ensures that all nets are connected reliably, within timing and electrical constraints, while optimizing area and power.

Goals of Routing:

- Ensure 100% connectivity – All nets in the design must be physically connected between their source and target pins.
- Minimize wirelength – Shorter wires reduce delay, capacitance, and power consumption.

- Avoid routing congestion – Distribute routing resources efficiently to prevent blockages and bottlenecks.
- Meet timing constraints – Routes must support setup/hold timing and critical path performance.
- Maintain signal integrity – Prevent crosstalk, noise, and glitches by proper spacing and shielding.
- Comply with design rules (DRC) – All wires and vias must follow manufacturing constraints (e.g., spacing, width, metal layer rules).
- Minimize via usage – Reduces resistance, delay, and potential reliability issues.
- Optimize for power and area – Efficient routing reduces switching power and maximizes silicon utilization.

Constraints of Routing:

Routing must meet several important constraints to ensure functionality and manufacturability. These include a limited number of metal layers with fixed directions and capacities, and strict design rules such as minimum spacing, wire width, and via requirements. Routing must also respect timing constraints, avoiding delays caused by long wires or excessive vias.

In addition, it must maintain signal integrity, preventing crosstalk and noise between nearby wires. Other key constraints include routing congestion, antenna effects, and manufacturing rules like double patterning and redundant vias to ensure reliability.

Advantages and Disadvantages of Routing:

Advantages	Disadvantages
Ensures complete connectivity of all nets	Routing is computationally complex and time-consuming
Allows timing-driven optimization to meet setup/hold constraints	May lead to congestion if routing resources are insufficient

Supports signal integrity improvements (via spacing, shielding, buffering)	Can cause crosstalk and SI issues if not managed properly
Enables DFM enhancements like redundant vias and wire straightening	Subject to strict design rules (DRC, antenna, double patterning)
Provides visual feedback (congestion maps, DRC reports) for fixing problems	Routing errors may cause timing violations or make the chip non-manufacturable
Optimizes area and power via wirelength and via reduction	

How to Run Routing in Fusion Compiler:

After completing placement and CTS, the next major step is **Routing**. Routing connects all standard cells, macros, and IO pins using metal interconnect layers, while ensuring timing closure, signal integrity, and design rule compliance.

In our flow, the routing procedure is stored in a Tcl script. To execute it, the user simply runs:

```
source /project/tsmc28mmwave/users/leenawattad/ws/riscv/ROUTING.tcl
```

Note: Replace the path with the actual directory where you saved your `ROUTING.tcl` file.

What the Routing Script Contains

The `ROUTING.tcl` file defines all routing steps in Fusion Compiler. It is structured into the following stages:

1. Setup Application Options

```
set_app_options -name
route.global.force_rerun_after_global_route_opt -value true

set_app_options -name route.global.timing_driven -value
true

set_app_options -name route.track.timing_driven -value true

set_app_options -name route.detail.timing_driven -value
true
```

- Enables **timing-driven routing** at all levels (global, track, detail).
- Ensures routing is optimized not only for connectivity but also for delay and congestion.

2. TSMC 28nm Routing Options

```
set_app_options -name
route.common.connect_within_pins_by_layer_name -value {{M1
via_wire_standard_cell_pins} {M2 off} {M3 off} ... {M8
off}}

set_app_options -name route.common.net_max_layer_mode -
value allow_pin_connection

set_app_options -name route.common.global_max_layer_mode -
value allow_pin_connection

set_app_options -name route.common.net_min_layer_mode -
value soft

set_app_options -name route.common.global_min_layer_mode -
value allow_pin_connection

set_app_options -name
route.common.number_of_vias_under_net_min_layer -value 5
```

```
set_app_options -name
route.common.number_of_vias_over_net_max_layer -value 5

set_app_options -name
route.common.number_of_vias_over_global_max_layer -value 5

set_app_options -name route.common.rotate_default_vias -
value false

set_app_options -name route.common.route_top_boundary_mode
-value stay_half_min_space_inside

set_app_options -name route.common.shielding_nets -value {}

set_app_options -name route.common.threshold_noise_ratio -
value 0.20
```

- Defines **routing rules specific to TSMC 28nm technology**.
- Controls how vias are inserted, which layers are allowed for routing, and how nets interact with boundaries.
- Introduces noise thresholds and shielding options to improve **signal integrity**.

3. Routing Constraints

```
set_ignored_layers \

  -min_routing_layer M1 \

  -max_routing_layer M6
```

- Restricts routing to **M1 through M6**.
- Prevents unnecessary use of higher layers, which are often reserved for power/clock or special nets.

4. Routing Flow Execution

```
check_routability

route_global
```

```
route_track

route_detail -max_number_iterations 5
```

- **check_routability**: Verifies design readiness for routing.
- **route_global**: Assigns nets to regions and layers (coarse plan).
- **route_track**: Maps nets to routing tracks.
- **route_detail**: Finalizes wire shapes and performs design rule fixes (shorts, spacing).

5. Routing Optimization

```
route_opt

add_redundant_vias

route_eco
```

- **route_opt**: Optimizes critical paths to improve timing and reduce congestion.
- **add_redundant_vias**: Adds extra vias to improve reliability and reduce resistance.
- **route_eco**: Applies Engineering Change Orders (ECO) fixes late in the flow.

6. Verification and PG Connectivity

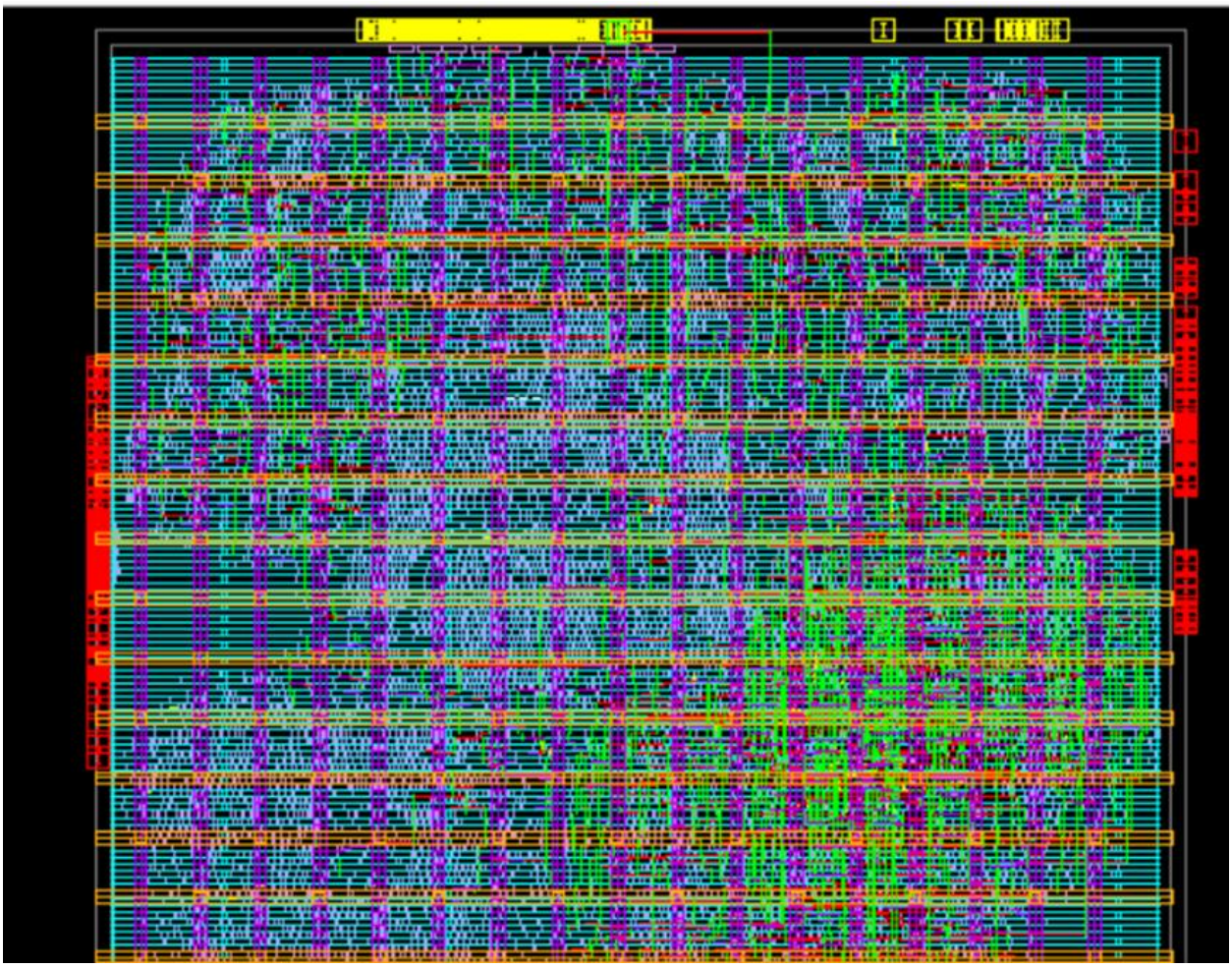
```
check_pg_drc

check_routes

check_lvs

connect_pg_net -net VDD [get_pins -hierarchical */VDD]

connect_pg_net -net VSS [get_pins -hierarchical */VSS]
```

- Validates **power and ground routing**.
- Runs **LVS checks** to confirm consistency between layout and netlist.
- Reconnects all VDD and VSS pins.

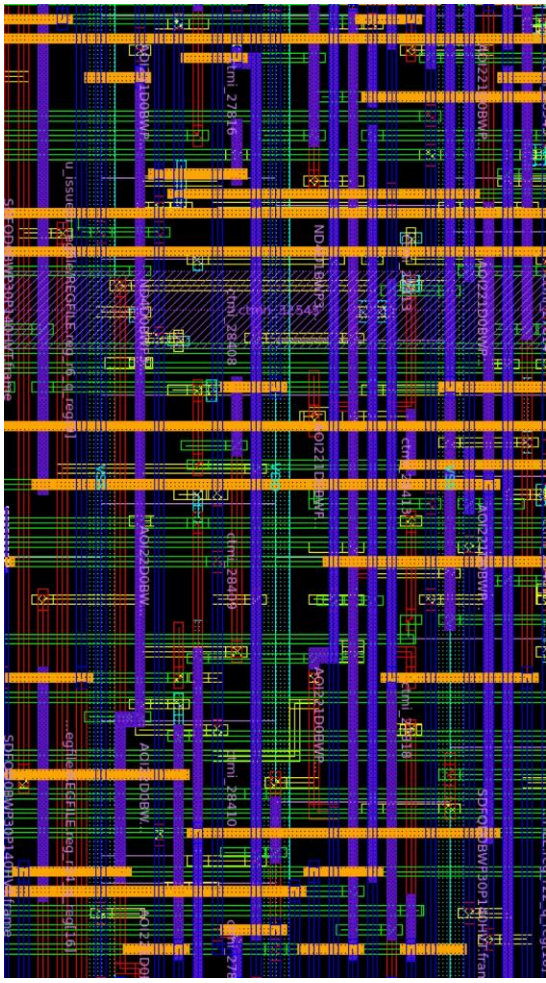
7. Final Analysis and Save

```
check_legality

report_congestion

report_utilization

save_block -as RISCV1/Routing

save_lib
```

- Performs final legality and congestion checks.
- Saves the fully routed design as **`RISCV1/Routing`**.



These images present the **final physical layout of the chip after the Routing stage** has been completed. At this stage, all signal nets, clock nets, and power connections are physically realized across multiple routing layers. Each color in the layout corresponds to a different metal layer (e.g., M1, M2, M3, ..., M6), making it possible to visualize the multi-layer interconnect structure.



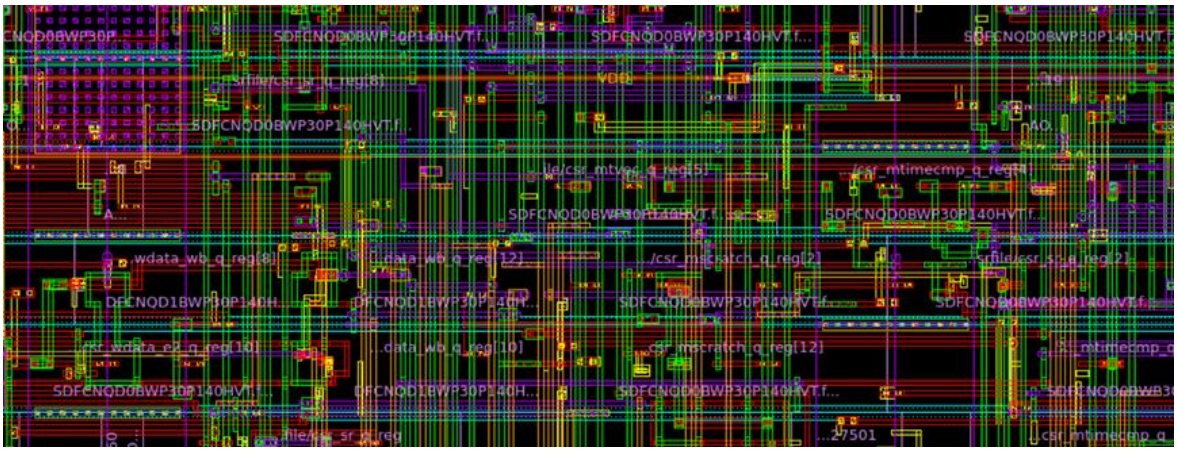
In this updated layout view:

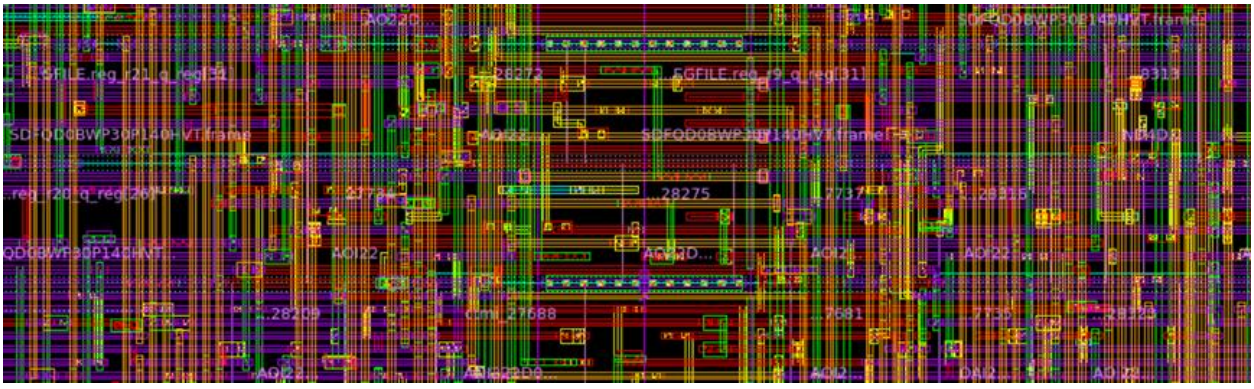
- The router has completed full connectivity for both clock and data paths, following the design rules and respecting layer constraints.
- Colored routing tracks (green, blue, red, yellow, purple) represent the actual wiring of nets across the chip, where denser regions indicate high logic activity or complex interconnections.
- The layout demonstrates track balancing: utilization of routing resources is spread across available metal layers to minimize congestion.
- Via structures are inserted where vertical connections between layers are required, ensuring robust 3D connectivity.
- Timing-driven routing ensures that critical nets are assigned shorter or less congested paths, improving overall performance.

The layout is now:

- **Fully routed and DRC-clean**, with design rule violations fixed during detail routing.
- **Optimized for timing and congestion**, ensuring high utilization without hotspots.
- **Ready for post-route checks** such as LVS (Layout vs. Schematic) and signoff verification.

This marks the transition from a partially placed and clocked design (post-CTS) to a **fully implementable, manufacturable chip layout**, where every net is physically wired and validated.





These zoomed-in images show the **fine-grained routing details** inside a small region of the chip after the Routing stage. Compared to the high-level full layout, this view emphasizes the density and precision of wiring between cells and macros.

In this updated layout view you can observe:

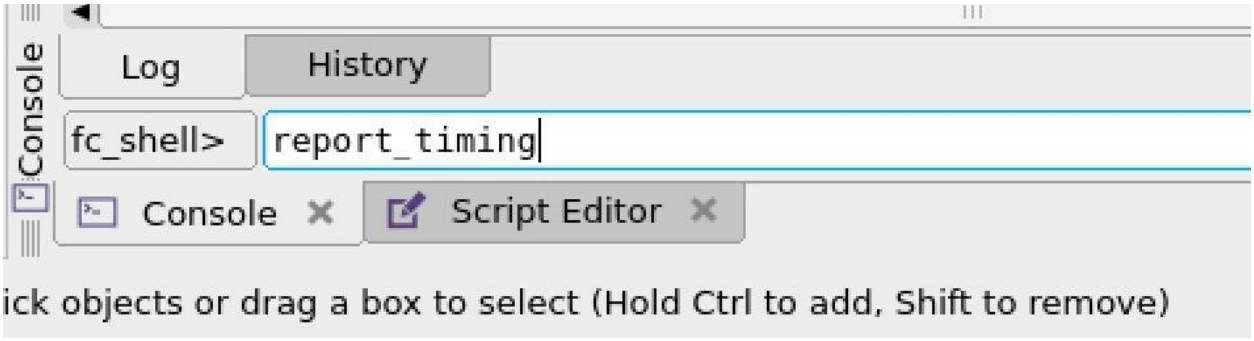
- Multi-colored metal wires (green, red, blue, yellow, purple) representing horizontal and vertical routing tracks across different layers (M1–M6).
- Standard cell pins and register labels (e.g., `reg[ ]`, `csr`, `data_wb`) clearly visible and connected through dense routing.
- Vias (tiny squares/dots) marking vertical connections between stacked metal layers.
- Vertical yellow and purple lines highlight stacked tracks and highly congested routing areas.
- Dense interconnect clusters show regions of high logic activity, requiring careful routing to minimize delay and congestion.

The router ensures that:

- Wires are assigned to legal tracks and follow design rules.
- Vias are strategically placed to link wires between layers without violating spacing.
- Timing closure and signal integrity are preserved even in highly dense regions.
- DRC-clean routing is maintained, avoiding shorts, spacing errors, and congestion overflow.

Route Verification with `check_routes:`

After completing the routing stage, it is important to re-check the design timing since routing adds wire lengths and parasitic effects that can slightly change delays. The following timing report excerpt shows the analysis for the clock domain clk_i:



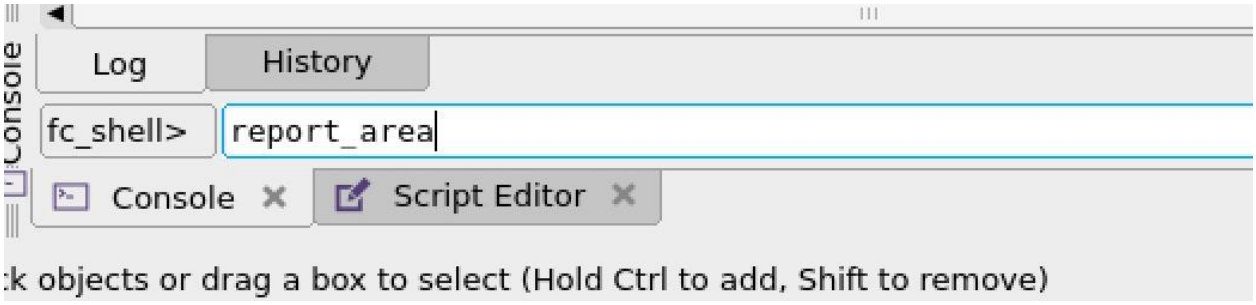

```
clock network delay (propagated)                 0.15      10.15
u_exec/result_q_reg[0]/CP (SDFCND0BWP30P140HVT)
                                                 0.00      10.15 r
clock uncertainty                               -0.15      10.00
library setup time                              -0.02       9.98
data required time                                          9.98
---------------------------------------------------------------------------
data required time                                          9.98
data arrival time                                          -5.09
---------------------------------------------------------------------------
slack (MET)                                                 4.88
```

In our example shown:

- The clock domain analyzed is **clk_i** at a rising edge.
- The clock network delay after propagation is **0.15 ns**.
- The report shows a **slack of +4.88 ns**, which means setup timing is still met.
- Compared to the CTS stage (where slack was +5.64 ns), the slack decreased slightly due to added routing delays and parasitics.

This confirms that even after routing, the design remains timing-clean, meeting all setup requirements and ensuring reliable operation for signoff.

After routing and timing checks, it is also important to evaluate the **area utilization** of the design. In Fusion Compiler, this is done using the following command:



This command generates a summary of the cell count, logic breakdown, and total area consumed by the synthesized and placed design.

```
fc_shell> report_area
****************************************
Report : area
Design : riscv_core
Version: V-2023.12-SP3
Date   : Sat Sep  6 00:49:01 2025
****************************************

Number of ports:                          269
Number of nets:                         10867
Number of cells:                         9967
Number of combinational cells:           7585
Number of sequential cells:              2382
Number of macros/black boxes:               0
Number of buf/inv:                        656
Number of references:                     103

Combinational area:                  5314.43
Buf/Inv area:                         178.04
Noncombinational area:               6373.08
Macro/Black Box area:                   0.00

Total cell area:                    11687.51
1
```

In our example shown:

The total number of standard cells is 9,967, out of which 7,585 are combinational (logic gates) and 2,382 are sequential (flip-flops and registers). Additionally, 656 buffers and inverters were inserted for optimization.

The total cell area is 11,687.51 μm^2, with about half dedicated to combinational logic (5314 μm^2) and the rest mainly to sequential elements (6373 μm^2). No macros or black-boxes are present, meaning the entire area comes from standard cells.

Filler Cells

What are Filler Cells?

In the context of digital VLSI design, filler cells are a special category of *non-functional standard cells* that are inserted into the empty physical spaces left between logic cells after placement and routing. Unlike logic gates (e.g., AND, OR, flip-flops), filler cells contain no transistors for computation. Their purpose is purely *physical*: they maintain continuity of the underlying silicon layers and ensure proper connectivity of the chip's power distribution.

In a standard-cell based design, millions of small logic cells are placed side by side in *rows* on the silicon die. Because the automated placement and routing tools rarely fill the rows

perfectly, gaps inevitably appear between cells. If these gaps are left unoccupied, several serious issues can occur:

1. Discontinuity of Nwell/Pwell Regions – Modern CMOS processes rely on carefully defined *well* structures (Nwell and Pwell) and *implant* layers to isolate and bias transistors. If a gap interrupts the well region, the foundry's design rules are violated, leading to potential latch-up risks and unmanufacturable layouts.

2. Breaks in VDD/VSS Rails – Each standard cell row includes horizontal rails (usually on Metal 1) that distribute the global power (VDD) and ground (VSS). When there is a gap between cells, these rails can be broken, leaving parts of the circuit without a stable supply. This results in floating cells, unpredictable behavior, or even chip failure.

3. Design Rule Check (DRC) Errors – Foundries enforce strict geometric rules for spacing, well enclosures, and implant continuity. Empty spaces violate these rules, causing DRC errors that must be fixed before tape-out.

4. Density and Planarity Concerns – Semiconductor manufacturing requires reasonably uniform pattern density across the chip. Gaps reduce local density, complicating chemical-mechanical polishing (CMP) and lowering yield.

Filler cells solve all of these problems by *filling the gaps* with small, predefined physical-only structures that "extend" the well, implant, and metal rails seamlessly across the row.

Why are Filler Cells Necessary?

The insertion of filler cells is not optional; it is an essential stage of the physical design flow. Their importance can be summarized as follows:

- **Electrical Continuity:** They connect and extend VDD/VSS rails so that every standard cell has a stable supply path.

- **Physical Continuity:** They ensure that wells and diffusion regions remain continuous across the entire row, preventing DRC violations.

- **Manufacturability:** They satisfy foundry rules for density, continuity, and planarity, thus improving silicon yield.

- **Reliability:** They reduce the risk of latch-up and prevent opens in power networks.

Without filler cells, the design would almost certainly fail DRC signoff and could not be fabricated

How to Insert Filler Cells in Fusion Compiler

In modern tools such as Synopsys Fusion Compiler (or ICC2), filler cells are inserted automatically through dedicated commands after routing and optimization are complete.

The general procedure is:

1. Remove old fillers (in case of ECOs or re-insertion).

2. Run the filler insertion command, specifying a list of available filler cells from the technology library (typically multiple sizes, e.g., FILL64, FILL32, … FILL2, to cover all gap widths).

3. Legalize the placement so all fillers are aligned properly in rows.

4. Run verification checks (routes, power connectivity, DRC).

You can run the following commands:

```
# Remove old fillers

set old_fillers [get_cells -hier -filter "ref_name =~ *FILL*"]

if {[sizeof_collection $old_fillers] > 0} {

  remove_cell $old_fillers

  legalize_placement -incremental

}

# Insert fillers
```

```
create_stdcell_fillers \
  -lib_cells {
    tcbn28hpcplusbwp30p140/FILL64BWP30P140
    tcbn28hpcplusbwp30p140/FILL32BWP30P140
    tcbn28hpcplusbwp30p140/FILL16BWP30P140
    tcbn28hpcplusbwp30p140/FILL8BWP30P140
    tcbn28hpcplusbwp30p140/FILL4BWP30P140
    tcbn28hpcplusbwp30p140/FILL2BWP30P140
  } \
  -prefix FILLER
# Verify placement
legalize_placement -incremental
check_routes
Check_pg_connectivity
```

The result that you will see after that:

xofiller!FILLER!FILL64BWP30P140!x24200y623000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y614000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y605000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y596000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y587000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y578000

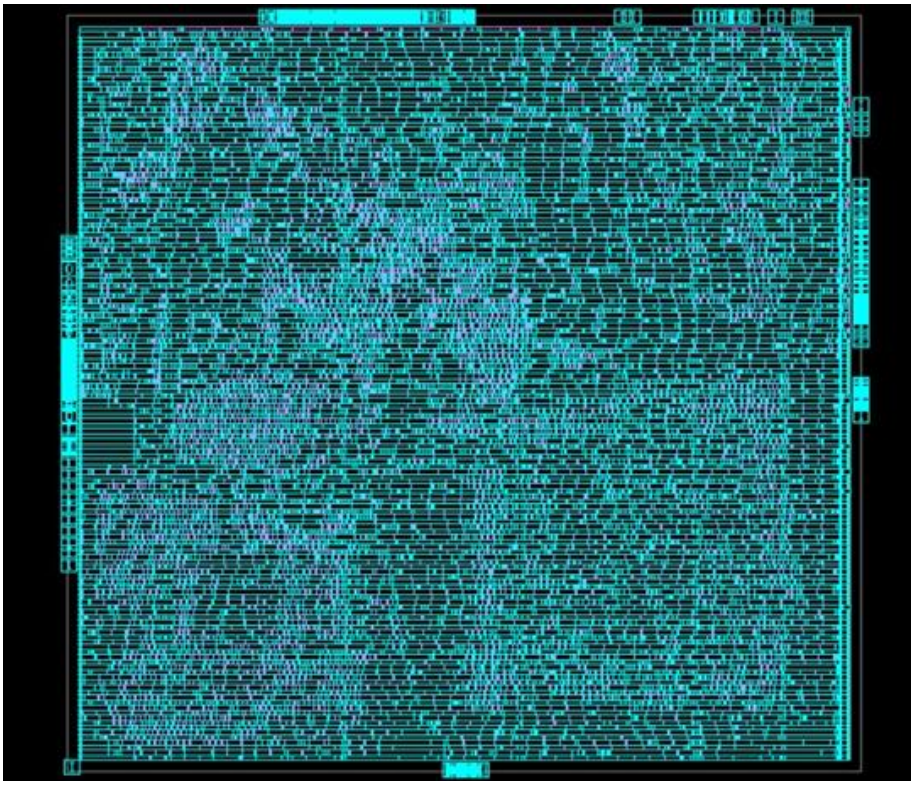
FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y569000

FILL64BWP30P140.frame

xofiller!FILLER!FILL64BWP30P140!x24200y560000

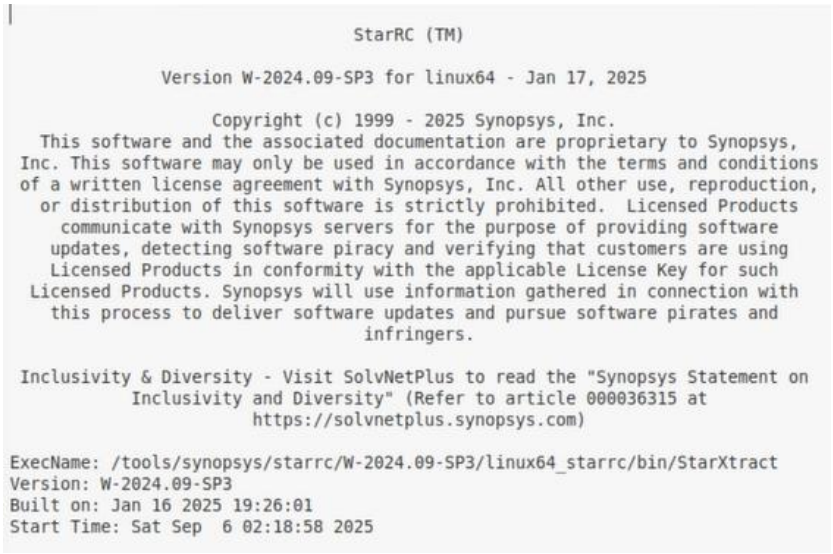
FILL64BWP30P140.frame



Advantages and Disadvantages of Filler Cells:

Advantages	Disadvantages
Maintain continuous VDD/VSS power rails.	Increase the number of cell instances (affects database size)
Prevent DRC violations caused by well or implant discontinuities	Slightly increase GDSII file size
Ensure continuity of Nwell/Pwell and diffusion regions	Require re-insertion after ECO changes or re-routing
Improve manufacturability and yield by satisfying density rules	Occupy whitespace that could otherwise host decap or spare cells
Simplify LVS comparison by avoiding unmatched gaps in layout	Add to tool runtime during insertion and legalization
Reduce risk of latch-up by ensuring uniform well ties across rows	Need to maintain correct library versions (different Vt fillers if multiple threshold voltages are used)
Support DFM (Design for Manufacturability) by homogenizing local layout patterns	Improper usage can cause illegal overlaps if keepouts are ignored

StarRC – Parasitic Extraction



```
                                StarRC (TM)

                Version W-2024.09-SP3 for linux64 - Jan 17, 2025

                     Copyright (c) 1999 - 2025 Synopsys, Inc.
   This software and the associated documentation are proprietary to Synopsys,
 Inc. This software may only be used in accordance with the terms and conditions
 of a written license agreement with Synopsys, Inc. All other use, reproduction,
   or distribution of this software is strictly prohibited.  Licensed Products
     communicate with Synopsys servers for the purpose of providing software
    updates, detecting software piracy and verifying that customers are using
    Licensed Products in conformity with the applicable License Key for such
  Licensed Products. Synopsys will use information gathered in connection with
    this process to deliver software updates and pursue software pirates and
                                infringers.

 Inclusivity & Diversity - Visit SolvNetPlus to read the "Synopsys Statement on
            Inclusivity and Diversity" (Refer to article 000036315 at
                          https://solvnetplus.synopsys.com)

ExecName: /tools/synopsys/starrc/W-2024.09-SP3/linux64_starrc/bin/StarXtract
Version: W-2024.09-SP3
Built on: Jan 16 2025 19:26:01
Start Time: Sat Sep  6 02:18:58 2025
```

Parasitic Extraction (PE) is a critical stage in the RTL-to-GDSII flow. After routing, wires and vias introduce non-ideal effects — resistance (R) and capacitance (C) — which directly impact timing, noise, and power.

StarRC is Synopsys' industry-standard tool for extracting these parasitics. It generates a SPEF (Standard Parasitic Exchange Format) file that is later imported into PrimeTime (STA) to enable signoff-quality timing analysis.

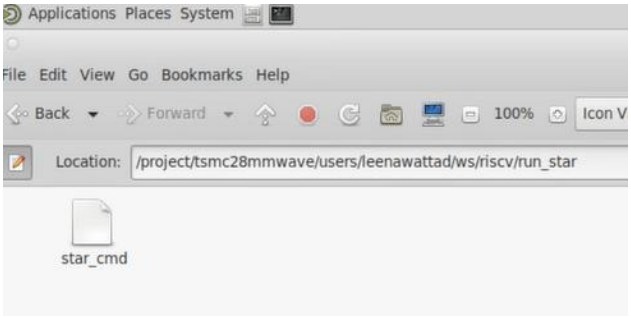
Goals of StarRC

- Perform accurate RC extraction of routed interconnects.
- Generate SPEF files for all or selected nets.
- Improve timing accuracy in PrimeTime signoff.
- Support Multi-Corner Multi-Mode (MCMM) analysis.
- Provide runtime, memory, and reduction reports for verification.

Required Files

1. NDM Database (.dlib) – contains the design database.
2. Reference Libraries (.ndm) – standard cell libraries (SVT, HVT, LVT).
3. Technology RC Models (TLU+/NXTGRD) – define R and C for each process layer.
4. Mapping File (.map) – maps design layers (DEF/GDS) to technology layers.
5. DEF/GDSII – final routed layout.
6. Verilog Netlist – logical connectivity.
7. SDC – design timing constraints.

Running StarRC



1. Star Command File (`star_cmd`)

```
* Specify block name for parasitic extraction

BLOCK: RISCV1/filler

* Provide the input NDM design database

NDM_DATABASE:  /project/tsmc28mmwave/users/leenawattad/ws/riscv/RISCV1.dlib

* Specify nxtgrd file which consists of capacitance models

TCAD_GRD_FILE:  /data/tsmc/28HPCPMMWAVE/synopsys/StarRC/1.9p1a/typical/crn28hpc+_1p09m+ut-
alrdl_5x1y1z1u_typical.nxtgrd

* Provide the mapping file in which design layers mapped to process layers

MAPPING_FILE:
/data/tsmc/28HPCPMMWAVE/synopsys/StarRC/1.9p1a/typical/Reference/MAP/star.map_icc_crn28hpc+_1p9m_
5x1y1z1u_ut-alrdl

* Reduction setting fro STA Analysis

REDUCTION: NO_EXTRA_LOOPS

* Use '*' to extract all signal nets in the design. Otherwise, provide the net names to be
extracted separated by a space. Wildcards '?' and '!' are accepted for net names

NETS: *

* Use 'RC' to perform resistance and capacitance extraction on the nets

EXTRACTION: RC
```

```
* Provide operating temperature in degree celsius at which extraction is performed

* OPERATING_TEMPERATURE: <temperature_in_celsius>

* Choose maximum of 2 cores for designs less than 100k nets, 4 to 6 cores for designs around
1Million nets and 8 to 16 cores for designs around 10Million nets

* NUM_CORES: 4

* Provide settings to distribute StarRC job on Gridwire or LSF. Use Command Reference manual for
reference

* STARRC_DP_STRING:

SKIP_CELLS: *

COUPLE_TO_GROUND: NO

COUPLING_ABS_THRESHOLD: 3e-15

COUPLING_REL_THRESHOLD: 0.03

REDUCTION_MAX_DELAY_ERROR: 1e-14

* Provide the name of a directory

* GPD: ./results/<gpd_dir>

NETLIST_FORMAT: SPEF

NETLIST_FILE: ./results/top_typical.spef

* Provide the name of a summary file to which runtime and memory usage is written

SUMMARY_FILE: ./results/top.star_sum

* Provide the working directory name to which StarRC internal information is written in binary

STAR_DIRECTORY: ./star
```

---

2. Terminal Command

```
cd /project/tsmc28mmwave/users/user_name/ws/riscv

StarXtract ./run_star/star_cmd | tee run_star/starrc.log
```

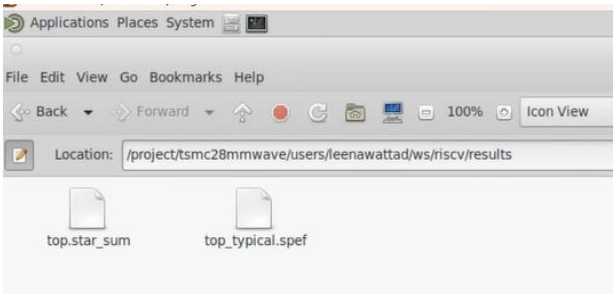
Flow Stages in StarRC

- Setup / Layers / Cells – Load libraries, mapping, and database.

- xTract – Extract parasitics (resistors, capacitors, nodes).
- Reduction – Reduce network size while keeping delay accuracy.
- Report Violations – Identify shorts, opens, mapping mismatches.
- Output – Write SPEF and summary reports.

Reports and Outputs

- SPEF File: `./results/top_typical.spef`
- Summary Report: `./results/top.star_sum`



Metric	Value
Resistors extracted	1,126,826 → reduced to **109,681** (~90.3%)
Capacitors extracted	863,902 → reduced to **137,057** (~84.1%)
Coupling capacitors	**18,078** (~13% of total caps)
Nodes	1,027,448 → reduced to **118,979** (~88.4%)
Runtime (overall)	**19 sec (`xTract` ≈15 sec)**
Peak memory	**1.73 GB**
SPEF output	**Generated successfully**

Advantages and Disadvantages

Advantages	Disadvantages
Signoff-level timing accuracy	**Higher runtime for big designs**
Supports Multi-Corner Multi-Mode	**Requires large memory footprint**
Seamless integration with PrimeTime	**Depends on high-quality mapping/tech files**
Provides detailed violation reports	**Debug effort required for shorts/layers**

Conclusion

StarRC is an essential signoff parasitic extraction step in the backend flow. By replacing idealized RC models with real extracted parasitics, it ensures that timing closure in PrimeTime accurately represents silicon behavior.

In our project, StarRC successfully reduced over 1 million parasitic elements into a manageable SPEF, revealed shorts and mapping mismatches, and provided the foundation for reliable STA signoff.

Static Timing Analysis with PrimeTime

After completing placement, CTS, and routing in Fusion Compiler, the design must undergo signoff-level timing analysis to ensure it meets all performance requirements

across process, voltage, and temperature (PVT) corners. This is done using Synopsys PrimeTime, an industry-standard tool for Static Timing Analysis (STA).

What is PrimeTime?

PrimeTime is Synopsys's golden signoff tool for Static Timing Analysis (STA). Unlike dynamic simulation, which requires input vectors to exercise logic paths, STA systematically checks all possible timing paths in the design without running test patterns. This makes it faster, more comprehensive, and mandatory before fabrication (tape-out).

The tool relies on several key inputs:

- **Gate-level netlist (Verilog):** The full connectivity of the placed-and-routed design.
- **Timing libraries (.lib):** Technology-specific cell models that describe propagation delays, setup/hold requirements, and power characteristics at different corners (e.g., fast, typical, slow).
- **Parasitic data (SPEF/RC extraction):** Information about wire resistance and capacitance extracted from the routed layout, which directly impacts signal delays.
- **Constraints (SDC):** User-defined timing goals, such as clock definitions, input/output delays, multicycle paths, and false paths.

With this data, PrimeTime performs:

- **Arrival time calculation:** Determines how long it takes for a signal to travel from its source (e.g., a clock edge or input port) to its destination (e.g., a flip-flop or output pin).
- **Required time calculation:** Determines when the signal must arrive to meet setup or hold requirements.
- **Slack analysis:** Compares arrival vs. required times to report whether paths meet or violate timing.

PrimeTime also incorporates advanced analysis features, including:

- **Clock skew and jitter modeling** to account for variations in clock distribution.
- **On-Chip Variation (OCV)** and derating to capture real-world manufacturing variations.

- **Multi-mode, multi-corner (MMMC) analysis** to ensure the design meets timing across all operating scenarios (e.g., different voltages, temperatures, and process corners).

In short, PrimeTime transforms the raw design data into a timing signoff certificate. If the design passes PrimeTime analysis, it is considered safe for tape-out, as this guarantees that signals will arrive correctly and reliably in silicon.

Why PrimeTime?

- Provides signoff accuracy final verification before tape-out.
- Analyzes multiple PVT corners and modes.
- Identifies critical timing paths and possible violations.
- Ensures that the design meets timing, power, and reliability goals beyond Fusion Compiler's internal checks.

How to Run PrimeTime in Fusion Compiler

To perform signoff-level Static Timing Analysis (STA) with **PrimeTime**, we use a dedicated script that sets up the required libraries, reads the design files, and generates timing reports. The script ensures consistent and repeatable analysis across different runs.

In the PrimeTime shell, you simply run:

```
pt_shell> source pt_run
```

This command loads and executes the script file `pt_run.tcl`.
Users must replace the file path with the correct location where their script is saved.

pt_run.tcl (Full Script):

```
lappend link_path " \

/data/tsmc/28HPCPMMWAVE/dig_libs/TSMCHOME/digital/Front
_End/timing_power_noise/NLDM/tcbn28hpcplusbwp30p140hvt_
180a/tcbn28hpcplusbwp30p140tt0p8v1v25c.db \
```

```
/data/tsmc/28HPCPMMWAVE/dig_libs/TSMCHOME/digital/Front
_End/timing_power_noise/NLDM/tcbn28hpcplusbwp30p140hvt_
180a/tcbn28hpcplusbwp30p140vttt0p8v1v25c.db \

/data/tsmc/28HPCPMMWAVE/dig_libs/TSMCHOME/digital/Front
_End/timing_power_noise/NLDM/tcbn28hpcplusbwp30p140hvt_
180a/tcbn28hpcplusbwp30p140lvttt0p8v1v25c.db \

/data/tsmc/28HPCPMMWAVE/dig_libs/TSMCHOME/digital/Front
_End/timing_power_noise/NLDM/tcbn28hpcplusbwp30p140hvt_
180a/tcbn28hpcplusbwp30p140hvttlv25c.db \

/data/tsmc/28HPCPMMWAVE/dig_libs/TSMCHOME/digital/Front
_End/timing_power_noise/NLDM/tcbn28hpcplusbwp30p140hvt_
180a/tcbn28hpcplusbwp30p140vtttlv25c.db "

read_verilog riscv_filer.v

link

read_parasitics -keep_capacitive_coupling
results/top_typical.spef

read_sdc riscv.sdc

update_timing

report_timing
```

Step-by-Step Explanation

1. **Library Setup (`lappend link_path ...`)**
    - Adds all timing library files (.db) from the TSMC 28HPC PDK into the link path.
    - These libraries correspond to different process corners (TT, VTT, LVTT, HVTT, etc.) at specific voltages and temperatures.

- PrimeTime uses them to model how delays change under different operating conditions.

2. **Read Netlist (`read_verilog riscv_filer.v`)**
   - Loads the post-route gate-level Verilog netlist of the design (`riscv_filer.v`).
   - This describes all the standard cell instances and their interconnections after physical design.

3. **Link Design (`link`)**
   - Resolves references between the netlist and the timing libraries.
   - Ensures every cell used in the netlist has a matching definition in the `.db` files.

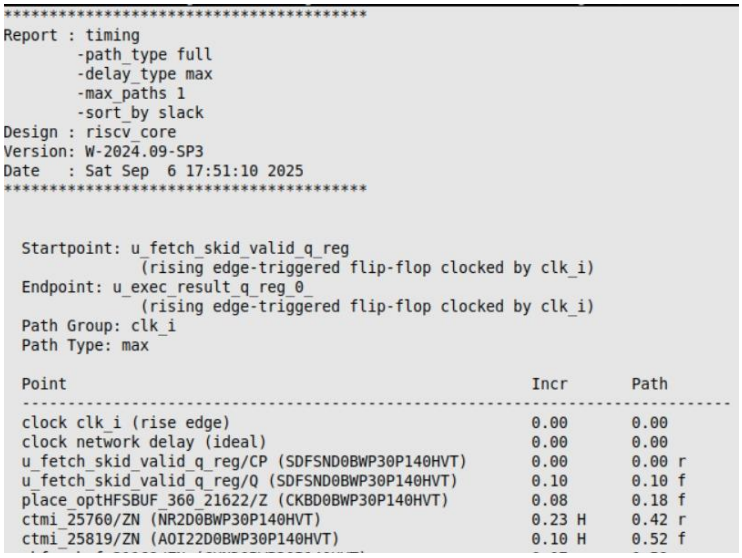
4. **Read Parasitics (`read_parasitics ... top_typical.spef`)**
   - Loads the SPEF file generated during extraction in Fusion Compiler.
   - This includes wire resistance and capacitance, plus capacitive coupling between nets (enabled here with `-keep_capacitive_coupling`).
   - These values directly affect timing accuracy.

5. **Read Constraints (`read_sdc riscv.sdc`)**
   - Reads the SDC constraint file, which defines clocks, input/output delays, false paths, multicycle paths, etc.
   - Without constraints, PrimeTime cannot evaluate setup/hold requirements.

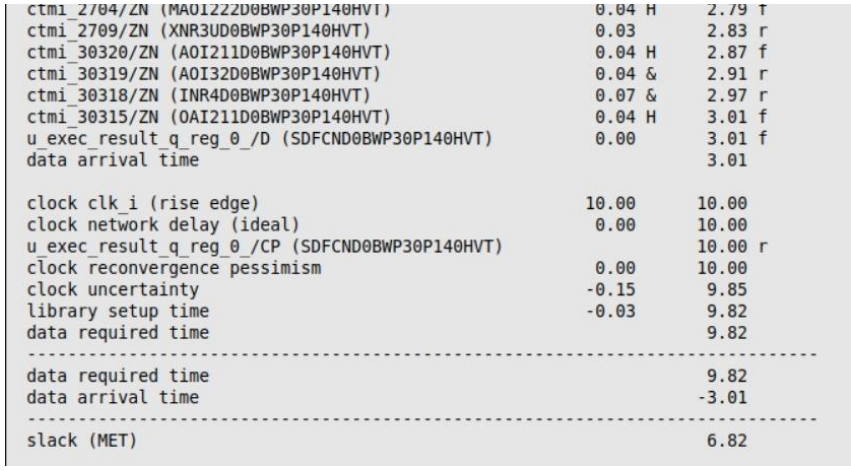
6. **Update Timing (`update_timing`)**
   - Recomputes delays using netlist + parasitics + libraries + constraints.
   - Ensures that any new data loaded is reflected in the timing database.

7. **Report Timing (`report_timing`)**
   - Generates the final timing report, listing critical paths, setup/hold slack, and violations.
   - This report is used to check if the design meets signoff timing.

PrimeTime Timing Path Report:



```
****************************************
Report : timing
         -path_type full
         -delay_type max
         -max_paths 1
         -sort_by slack
Design : riscv_core
Version: W-2024.09-SP3
Date   : Sat Sep  6 17:51:10 2025
****************************************


  Startpoint: u_fetch_skid_valid_q_reg
              (rising edge-triggered flip-flop clocked by clk_i)
  Endpoint: u_exec_result_q_reg_0_
              (rising edge-triggered flip-flop clocked by clk_i)
  Path Group: clk_i
  Path Type: max

  Point                                                   Incr       Path
  -----------------------------------------------------------------------
  clock clk_i (rise edge)                                 0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  u_fetch_skid_valid_q_reg/CP (SDFSND0BWP30P140HVT)       0.00       0.00 r
  u_fetch_skid_valid_q_reg/Q (SDFSND0BWP30P140HVT)        0.10       0.10 f
  place_optHFSBUF_360_21622/Z (CKBD0BWP30P140HVT)         0.08       0.18 f
  ctmi_25760/ZN (NR2D0BWP30P140HVT)                       0.23 H     0.42 r
  ctmi_25819/ZN (AOI22D0BWP30P140HVT)                     0.10 H     0.52 f
```



```
ctmi_2704/ZN (MAOI222D0BWP30P140HVT)                    0.04 H     2.79 f
ctmi_2709/ZN (XNR3UD0BWP30P140HVT)                      0.03       2.83 r
ctmi_30320/ZN (AOI211D0BWP30P140HVT)                    0.04 H     2.87 f
ctmi_30319/ZN (AOI32D0BWP30P140HVT)                     0.04 &     2.91 r
ctmi_30318/ZN (INR4D0BWP30P140HVT)                      0.07 &     2.97 r
ctmi_30315/ZN (OAI211D0BWP30P140HVT)                    0.04 H     3.01 f
u_exec_result_q_reg_0_/D (SDFCND0BWP30P140HVT)          0.00       3.01 f
data arrival time                                                  3.01

clock clk_i (rise edge)                                10.00      10.00
clock network delay (ideal)                             0.00      10.00
u_exec_result_q_reg_0_/CP (SDFCND0BWP30P140HVT)                   10.00 r
clock reconvergence pessimism                           0.00      10.00
clock uncertainty                                      -0.15       9.85
library setup time                                     -0.03       9.82
data required time                                                 9.82
-------------------------------------------------------------------------
data required time                                                 9.82
data arrival time                                                 -3.01
-------------------------------------------------------------------------
slack (MET)                                                        6.82
```

This timing report shows the **worst-case setup path** analyzed by PrimeTime after loading the design, constraints, and parasitics. The report is configured with:

- `-path_type full` → prints the entire path from launch to capture.
- `-delay_type max` → checks maximum delay (setup analysis).
- `-max_paths 1` → reports the single worst path.
- `-sort_by slack` → sorts by slack (worst case first).

Key observations from the report:

- **Startpoint**: `u_fetch_skid_valid_q_reg` (a flip-flop clocked by `clk_i`).
- **Endpoint**: `u_exec_result_q_reg_0` (another flip-flop also clocked by `clk_i`).
- This indicates the path is a register-to-register path within the same clock domain.

Clock definition:

- Launch clock edge = `clk_i` at **0 ns**.
- Capture clock edge = `clk_i` at **10 ns** (period of 10 ns).

Data Path Delay:

- The combinational logic between the two registers introduces a **data arrival time of 3.01 ns**.
- This delay is broken down gate by gate (e.g., buffers and logic cells like `ctmi_25760`, `ctmi_25819`).

Required Time:

- Setup check requires the data to arrive before **9.82 ns** (10 ns period minus setup/uncertainty).

Slack:

- Calculated as **Required Time (9.82) – Arrival Time (3.01) = 6.82 ns**.
- Since slack is positive, the path **meets timing comfortably**.

Conclusion:
This report confirms that the design's worst setup path in the `clk_i` domain is well within timing limits, with a **healthy margin of 6.82 ns**. The data arrives much earlier than required, ensuring reliable operation.

Explanation of QoR (Quality of Results) Report:

This report provides a high-level summary of the timing quality of the design after running PrimeTime. It checks the most critical paths in different path groups (clock domains) and shows if the design meets the required setup and hold constraints.

1. Path Group: `clock_gating_default` (max_delay/setup)

```
Timing Path Group '**clock_gating_default**' (max_delay/setup)
-----------------------------------------
Levels of Logic:                    57
Critical Path Length:             2.46
Critical Path Slack:              7.37
Total Negative Slack:             0.00
No. of Violating Paths:              0
-----------------------------------------
```

- Levels of Logic: 57
  → This means the critical path in this group passes through 57 logic levels.

- Critical Path Length: 2.46 ns
  → The longest data path delay in this group is 2.46 ns.
- Critical Path Slack: 7.37 ns (positive)
  → The setup constraint is satisfied with a wide margin.
- No. of Violating Paths: 0
  → No violations here, all setup paths are clean.

2. Path Group: `clk_i` (max_delay/setup)

```
Timing Path Group 'clk_i' (max_delay/setup)
-------------------------------------------
Levels of Logic:                         60
Critical Path Length:                  3.01
Critical Path Slack:                   6.82
Total Negative Slack:                  0.00
No. of Violating Paths:                   0
-------------------------------------------
```

- Levels of Logic: 60
  → Critical path includes 60 logic levels.
- Critical Path Length: 3.01 ns
  → The longest data path delay here is 3.01 ns.
- Critical Path Slack: 6.82 ns (positive)
  → Setup time is fully met, and there is no timing violation.
- No. of Violating Paths: 0
  → Again, this domain is completely clean.

3. Path Group: `clock_gating_default` (min_delay/hold)

```
Timing Path Group '**clock_gating_default**' (min_delay/hold)
-------------------------------------------
Levels of Logic:                         1
Critical Path Length:                 0.05
Critical Path Slack:                  0.06
Total Negative Slack:                 0.00
No. of Violating Paths:                  0
-------------------------------------------
```

- Levels of Logic: 1
  → This is a very short path, just one logic level.
- Critical Path Length: 0.05 ns
  → Minimum delay is extremely small, only 0.05 ns.
- Critical Path Slack: 0.06 ns (positive)
  → Hold requirement is satisfied without violations.
- No. of Violating Paths: 0
  → No hold violations exist in this group.

Overall Conclusion

- All path groups (`clk_i` and `clock_gating_default`) show positive slack values.
- There are no setup or hold violations in the design.
- The design comfortably meets timing requirements, both for setup (max delay) and hold (min delay) checks.
- This confirms that the clock gating logic and main clock domain (`clk_i`) are well-optimized and stable for operation.

4. Area Report

This section of the report provides information about the physical area utilization of the design:

```
Area
---------------------------------------------
Net Interconnect area:                   0.00
Total cell area:                     11687.72
Design Area:                         11687.72
---------------------------------------------
```

- Net Interconnect Area: 0.00
  → Indicates that no additional area was reported for interconnect wires. This is often because the report focuses only on cell area, not routing layers.

- Total Cell Area: 11687.72
  → This is the total silicon area consumed by all standard cells (combinational, sequential, and buffers) used in the design.
  → It reflects the logic density and complexity of the implemented circuit.
- Design Area: 11687.72
  → Same as the total cell area in this case, meaning the design area is entirely determined by the standard cells without any overhead from macros or extra blocks.

Conclusion

The report confirms that the **entire design fits into an area of 11687.72 units²**, fully accounted for by standard cells.
This shows the design is compact and efficiently mapped, with no wasted interconnect area being reported.

5. Cell & Pin Count Report

This section summarizes the number of pins and cells present in the design

```
Cell & Pin Count
--------------------------------------------
Pin Count:                            45377
Hierarchical Cell Count:                  0
Hierarchical Port Count:                  0
Leaf Cell Count:                       9983
--------------------------------------------
```

- Pin Count: 45,377
  → The total number of pins (inputs, outputs, and internal connections) across all cells.
  → A high pin count usually indicates a complex design with many interconnections.
- Hierarchical Cell Count: 0
  → No hierarchical (macro-level) cells were used. The design is fully flattened, meaning it consists only of standard cells without additional hierarchy blocks.

- Hierarchical Port Count: 0
  → Since there are no hierarchical blocks, there are no associated ports to connect them.

- Leaf Cell Count: 9,983
  → The total number of standard (leaf) cells used in the design. These include combinational gates, flip-flops, and buffers that implement the logic.

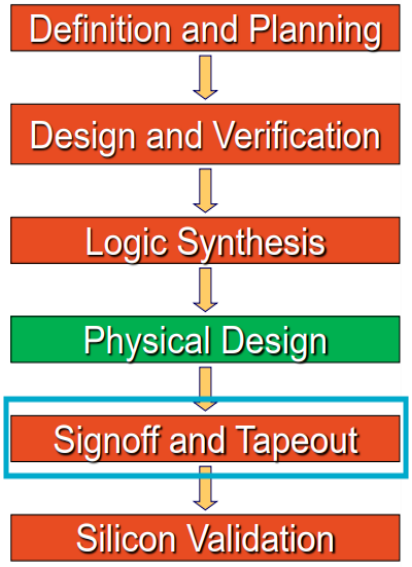
Conclusion

This report confirms that the design is composed entirely of standard leaf cells (9,983 in total), with no hierarchical blocks.
The 45,377 pins represent the connectivity complexity of the design, while the leaf cells reflect the actual logic and storage elements used to implement the circuit.

Signoff

What is Signoff?

Signoff is the final verification stage in the VLSI design flow. It ensures that the physical design of the chip is manufacturable, functionally correct, and meets all performance and reliability requirements before tapeout.
At this stage, the design is subjected to industry-standard checks such as Design Rule Check (DRC), Layout versus Schematic (LVS), timing verification, power integrity analysis, and metal density validation. Only after a chip is "signoff clean" can it be released to the foundry for fabrication in GDSII format.



Think of signoff as the quality control gate in chip manufacturing. Just as a product must pass rigorous inspections before leaving the factory, the IC design must undergo comprehensive checks to ensure it will work correctly in silicon.

Why is Signoff Necessary?

Signoff is critical because any undetected errors at this stage could result in a non-functional or unreliable chip, leading to wasted fabrication cost, schedule delays, and possible product failure in the field. Earlier stages (RTL, synthesis, floorplanning, routing) optimize and implement the design, but only signoff verifies it under realistic manufacturing and operating conditions.
It addresses technology-dependent concerns such as lithography limitations, parasitic effects, IR drop, electromigration, and CMP (Chemical Mechanical Planarization) uniformity. Without passing signoff, the design cannot be guaranteed to meet foundry requirements or customer specifications.

Goals of Signoff

- **Design Rule Compliance:** Ensure the layout follows all foundry PDK rules via DRC.
- **Functional Equivalence:** Verify layout matches the logical netlist through LVS.
- **Timing Closure:** Confirm all setup and hold constraints are met under extracted parasitics.
- **Power Integrity:** Validate IR drop, electromigration, and power noise margins.
- **Metal Density Compliance:** Insert dummy fills to achieve uniform density across layers.
- **Final Tapeout Preparation:** Deliver a clean, verified GDSII ready for mask generation.

Advantages and Disadvantages of Signoff

Advantages of Signoff

- **Manufacturability Guarantee:** Ensures the design can be fabricated without violations.
- **Improved Reliability:** Detects and mitigates risks like IR drop, EM, or timing failures.
- **Confidence for Tapeout:** Provides a final confirmation that all performance and physical requirements are satisfied.
- **Standardization:** Uses industry-proven signoff tools (e.g., Synopsys PrimeTime, IC Validator, StarRC).

Disadvantages of Signoff

- **High Complexity:** Requires multiple specialized tools and large compute resources.
- **Time-Consuming:** Full-chip checks with extracted parasitics can take many hours or days.
- **Late Discovery of Issues:** If major errors are found at signoff, fixing them may require significant rework.
- **Dependency on Foundry Rules:** Designs are constrained by ever-growing rule sets at advanced nodes (e.g., 28nm and below).

How to Perform Signoff

1. **Run DRC:** Use signoff-quality physical verification (e.g., Synopsys IC Validator) to ensure compliance with foundry rules.

2. **Perform LVS:** Compare extracted layout netlist against the logical schematic/netlist.

3. **Insert Metal Fill:** Add dummy metal shapes to meet density requirements; re-run density checks.

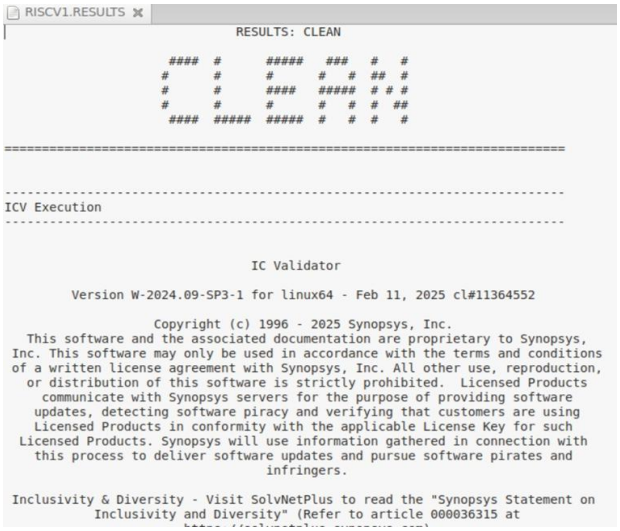
4. **Conduct STA with Parasitics:** Perform static timing analysis (Synopsys PrimeTime) using RC-extracted data from tools such as StarRC.

5. **Check Power Integrity:** Run IR-drop and electromigration analysis to validate power delivery.

6. **Consolidate Results:** Verify all violations are resolved and generate signoff reports.

7. **Export GDSII:** Create the final database, marking the transition from design to fabrication.

Signoff : Metal Fill Result:



The metal fill stage was executed successfully, with both the pre-fill and post-fill density checks reported as CLEAN. This outcome indicates that the inserted dummy metal shapes satisfied all TSMC 28nm foundry density requirements.

Specifically, the tool confirmed that:

- The minimum and maximum density thresholds were met across all targeted layers (M2–M6).
- No hotspots or density violations were detected after fill insertion.
- Planarity for the Chemical Mechanical Planarization (CMP) process is ensured, reducing the risk of dishing, erosion, or thickness variations.

- The added metal fills did not introduce design rule violations, preserving both manufacturability and design reliability.

In conclusion, the post-fill layout complies fully with foundry density specifications and is ready for subsequent signoff steps and final tapeout.