RTL to GDSII Flow of a 64-bit ALU Using Cadence EDA Tools

An Internship program report submitted in partial fulfilment of the requirements for the Award of Degree of

BACHELOR OF ENGINEERING
IN
ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

CHANDRAKIRAN G (312422106032)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
St. JOSEPH'S INSTITUTE OF TECHNOLOGY
(An Autonomous Institution)
OMR, CHENNAI – 600 119

2022 – 2026

June-2025

BONAFIDE CERTIFICATE

Certified that the Internship Program report titled “RTL to GDSII Flow of a 64-bit ALU Using Cadence EDA Tools” is the bona fide work of “CHANDRAKIRAN G” (312422106032),who completed the Internship program under my supervision.

SIGNATURE
Dr Noor Mahammad Sk.,PhD.,
Associate Professor

Indian Institute of Information Technology,
Design and Manufacturing (IIITDM) Kancheepuram
Melakottaiyur Village,
Chennai - 600 127

SIGNATURE
M Dhayalakumar, SMDP-C2SD
Project Staff

Indian Institute of Information Technology,
Design and Manufacturing (IIITDM) Kancheepuram
Melakottaiyur Village,
Chennai - 600 127

ACKNOWLEDGEMENT

At the outset, I would like to express my sincere gratitude to our beloved ***Chairman Dr. B. BABU MANOHARAN, M.A., M.B.A., Ph.D.***, for his constant guidance and support.

I would like to express my sincere thanks to our respected ***Executive Director, Mrs. S. JESSIE PRIYA, M. Com.***, and our ***Managing Director, Mr. B. SASHISEKAR, M.Sc*** for their kind encouragement and blessings.

I express my sincere gratitude and wholehearted thanks to our ***Principal Dr. S. ARIVAZHAGAN, M. E., Ph.D.***, for his encouragement to make this Internship a successful one.

I wish to express our truthful thanks and gratitude to our ***Head of the Department Dr Noor Mahammad Sk.,PhD., Associate Professor Computer Science and Engineering*** **at IIIT-DM Kancheepuram** for his constant support and motivation throughout my internship.

I have no words to express my heartfelt thanks to **Mr.M Dhayalakumar, SMDP-C2SD Project staff at IIIT-DM Kancheepuram** for her guidance, support and encouragement in completing the Internship and Project within the stipulated time

(Chandrakiran G)

CONTENTS

ABSTRACT

This project presents the design and implementation of a 64-bit Arithmetic Logic Unit (ALU) using the complete digital ASIC design flow, spanning from Register Transfer Level (RTL) to GDSII, utilizing **Cadence's industry-standard tools—Incisive/Xcelium**, **Genus**, and **Innovus**. The ALU supports a broad range of arithmetic and logic operations such as addition, subtraction, logical AND/OR/XOR, and shift operations, making it suitable for integration in high-performance computing systems.

The RTL was developed in **Verilog HDL**, following a modular and hierarchical structure to ensure scalability and reusability. Functional verification was performed using **Cadence Incisive/Xcelium**, with comprehensive testbenches that validated all operational modes, edge cases, and signed/unsigned scenarios.

The verified RTL was synthesized using **Cadence Genus**, targeting optimization for area, power, and timing. The synthesized netlist was imported into **Cadence Innovus** for physical implementation. Floorplanning, standard cell placement, **power grid design**, and **Clock Tree Synthesis (CTS)** were executed to meet physical and timing constraints. Signal routing was completed with attention to congestion and parasitic effects.

Post-routing, **Static Timing Analysis (STA)** and **Design Rule Checks (DRC)** were conducted to ensure timing closure and layout correctness. Power analysis validated dynamic and leakage power targets. The final **GDSII file** was generated for tape-out, marking the end of the ASIC design flow. This project demonstrates a complete front-to-back VLSI implementation of a 64-bit ALU using Cadence's digital design ecosystem.

1. INTRODUCTION

What is VLSI?

VLSI, or Very Large-Scale Integration, is a cutting-edge technology in the field of electronics that involves the design and fabrication of integrated circuits (ICs) by integrating millions or even billions of transistors onto a single silicon chip. These chips serve as the "brains" of electronic devices, enabling a wide range of functionalities from complex calculations to high-speed data processing. **The Revolution of VLSI Technology**

VLSI technology has transformed the way we create electronic circuits. Traditionally, electronic devices were built using bulky circuits composed of individual components like resistors, capacitors, and discrete transistors. VLSI replaces these cumbersome designs with highly compact and efficient chips, often no larger than a coin. This miniaturization has led to the development of powerful, portable, and energy-efficient devices that are integral to modern life.

Why VLSI is Important

Miniaturization of Devices:

Impact: VLSI allows for the creation of incredibly small and compact electronic devices. This miniaturization is what makes modern gadgets like smartphones, laptops, and smartwatches possible.

Example: A smartphone today has more computing power than the room-sized computers of the past, all thanks to VLSI technology.

Improved Performance:

Impact: By integrating more transistors onto a single chip, VLSI enhances the performance of electronic devices. This means faster processing speeds, better data handling, and overall improved functionality.

Example: High-performance gaming consoles and graphics cards rely on VLSI to deliver seamless and immersive gaming experiences.

Reduced Power Consumption:

Impact: VLSI chips are designed to be energy-efficient, reducing the power consumption of electronic devices. This makes devices not only faster but also more environmentally friendly.

Example: Energy-efficient processors in laptops and smartphones extend battery life, allowing users to stay connected for longer periods.

Cost Efficiency:

Impact: The mass production of VLSI chips reduces manufacturing costs significantly. This cost efficiency makes advanced electronics more affordable for consumers.

Example: The affordability of smartphones and other consumer electronics is largely due to the costeffective production of VLSI chips.

Key Steps in the VLSI Design Process Specification:

Purpose: Engineers define the functional requirements of the chip. This includes deciding what the chip should do, such as powering a smartphone, processing video, or managing data in a server.

Output: A detailed specification document outlining the chip's intended capabilities and performance metrics.

Design:

Purpose: A blueprint of the chip is created using specialized software tools. This design phase includes both the behavioural and structural aspects of the chip.

Process: Engineers use hardware description languages (HDLs) like Verilog or VHDL to describe the chip's functionality. This high-level design is then synthesized into a gate-level netlist.

Output: A comprehensive design that includes the layout and interconnections of all components on the chip.

Fabrication:

Purpose: The chip is physically built in a fabrication facility, also known as a foundry. This process involves layering and connecting millions of tiny components to create a functional integrated circuit.

Process: The design is translated into a series of photolithographic masks, which are used to pattern the layers of the chip. Advanced manufacturing techniques ensure precision and accuracy.

Output: A fully fabricated chip ready for testing and integration into electronic devices.

Testing:

Purpose: The final chip is rigorously tested to ensure it meets the specified performance and reliability standards. This includes functional testing, electrical testing, and environmental testing.

Process: Automated test equipment (ATE) is used to simulate various operating conditions and verify the chip's functionality.

Output: A certified chip that is ready for mass production and integration into end-user devices. Applications of VLSI

VLSI chips are used in a wide range of devices and industries, such as

1. Consumer Electronics:

- Smartphones (processors, memory, and sensors)
- Laptops and PCs (CPUs, GPUs, and RAM)
- Smart TVs and home assistants

2. Automotive Industry:

- Advanced driver-assistance systems (ADAS)
- Engine control units (ECUs) for efficient vehicle performance
- Electric vehicles (EVs) for battery management

3. Medical Devices:

- Wearable health monitors (like fitness trackers)
- Imaging systems (like MRI machines)
- Portable diagnostic tools

4. Aerospace and Defence:

- Satellite communication systems
- Radar and navigation systems
- High-speed data processing for military applications

5. Communication Networks:

- 5G and wireless communication infrastructure
- Routers and modems for internet connectivity

Conclusion

VLSI is at the heart of modern technology, powering the devices and systems we use daily. Its ability to integrate countless functions into a small chip makes it indispensable for innovation. From healthcare to entertainment, VLSI enables industries to thrive and evolve, pushing the boundaries of what's possible.

2. RTL2GDSII FLOW

2.1 Introduction

The entire Chip life cycle can be characterized by a few key stages that are briefly discussed below. The focus of this report will be the IC design stage

2.1.1 IC Design:

IC design is where the journey begins. It's the process of conceptualizing, defining, and creating the blueprint for an integrated circuit. It can be broadly divided into:

- **System-Level Design**: Translating user requirements and device functionality into system specifications.
- **RTL Design (Digital)**: Using hardware description languages like Verilog or VHDL to describe the logical behaviour.
- **Analog/Custom Design**: Creating precise transistor-level schematics for analog and mixedsignal components.
- **Verification and Validation**: Ensuring correctness through simulation, synthesis, and formal verification.
- **Physical Design**: Transforming logical designs into layouts for fabrication, including placement, routing, and timing closure.

This step combines creativity and engineering precision to design ICs that are smaller, faster, and energy efficient.

2.1.2 IC Manufacturing:

Once the design is finalized, the IC undergoes fabrication in highly specialized semiconductor foundries. This process includes:

- **Wafer Fabrication**: Creating wafers with multiple chips using photolithography, doping, etching, and deposition techniques.
- **Material Science**: Using advanced materials like silicon, gallium arsenide, or even emerging technologies like graphene.
- **Moore's Law in Action**: Shrinking transistor sizes for each generation to pack billions of transistors onto a single chip.
- **Packaging**: Encasing the fabricated chip in protective materials, ensuring electrical connections and heat dissipation.

This is a highly capital-intensive stage, requiring precision and expertise to achieve high yield and quality.

2.1.3 Post Tape-out IC Testing and Validation:

After tape-out (finalizing the layout for fabrication), rigorous testing and validation are performed to ensure the IC functions as intended:

- **Wafer Testing**: Probing individual chips on the wafer to test functionality before cutting them.
- **Package-Level Testing**: Checking electrical, thermal, and mechanical properties after packaging.
- **Burn-In Testing**: Running the chip at elevated temperatures and voltages to screen for early-life failures.
- **Field Tests and Debugging**: Evaluating performance in real-world conditions and fine-tuning as needed.

This stage ensures the IC is reliable and meets the required performance standards before it enters the market.

2.1.4 Integration into Devices:

Once validated, these ICs are integrated into smart devices such as smartphones, IoT devices, and autonomous vehicles. The entire process—from design to post-tape-out testing—is a marvel of modern engineering, pushing the boundaries of what's possible in technology.

The IC Design is the focus of the work we are concerned with here. To begin, we first understand the classification of IC design flows.

1. RTL to GDSII Flow
2. Custom / AMS Design flow

2.2 RTL2GDSII Design Flow:

Of the two available design flows, the RTL2GDSII is the most preferred for reasons mentioned below. While RTL2GDSII is the most preferred, it is by no means a sole option. In fact, any design of an IC can be achieved by any design flow, owning to their own caveats.

Benefits of RTL2GDSII over other methods:

Automation: The RTL-to-GDSII flow is highly automated, leveraging advanced tools for synthesis, placement, routing, timing analysis, and more. This reduces design time and effort, making it ideal for large-scale digital designs.

Scalability: Modern digital chips, like microprocessors and GPUs, contain billions of transistors. Managing such complexity at the transistor level would be impractical. RTL-to-GDSII enables designers to handle this complexity efficiently.

Time **Efficiency**: Since the process is automated, the turnaround time from specification to fabrication is much shorter than the manual, detail-oriented custom IC flow.

Reuse of IPs: The RTL-to-GDSII flow allows for easy reuse of pre-designed intellectual property (IP) blocks, further accelerating the design process.

Cost-Effectiveness: For digital ICs, the automated nature of RTL-to-GDSII reduces the need for manual intervention, which can lower costs in terms of labour and time.

Focus on Functionality: Designers can focus more on defining the functionality using RTL (Register Transfer Level) without getting into the nitty-gritty of transistor-level design.

The RTL2GDSII Design flow itself is further divided into many sub-categories and each one of the cans be as vast as an individual domain.

At high-level, it is divided into two distinct categories,

- Front End Design
- Back End Design

2.2.1 Front-End Design

Front-End Design involves defining the functionality and behaviour of the integrated circuit. It is focused on the logical and architectural aspects of the design. Here's a deeper look at its key components:

1. **System-Level Design:**
    - High-level architecture and specification of the IC based on its intended application. ○ Defines how different components (e.g., processors, memory, communication interfaces) interact with each other.
2. **RTL Design (Register Transfer Level):**
    - The functionality of the chip is described using hardware description languages (HDLs) such as Verilog or VHDL. ○ The RTL code defines how data flows between registers and how operations are performed.
3. **Functional Verification:**
    - Ensures that the RTL design works correctly according to the system-level specifications. ○ Techniques include simulation, formal verification, and emulation to identify bugs early in the process.
4. **Logic Synthesis:**
    - The RTL code is converted into a gate-level netlist using logic synthesis tools.
    - Maps the high-level design to basic logic gates while optimizing for power, performance, and area (PPA).
5. **Static Timing Analysis (STA):**
    - Verifies that the design meets timing requirements (setup and hold times).
    - Ensures the design operates reliably at the target clock frequency.

6. **Design for Testability (DFT):**
    - Adds test structures (e.g., scan chains, Built-In Self-Test) to the design to facilitate testing during manufacturing.
    - Ensures defects in the chip can be identified efficiently post-manufacturing.

2.2.2 Back-End Design

Back-End Design focuses on the physical implementation of the IC. It involves translating the logical design (netlist) from the Front-End stage into a physical layout that can be fabricated. Here's a detailed breakdown of this crucial stage:

1. **Floor planning:**

- This is the initial step where the chip's overall layout and organization are determined.
- The locations of major components (macros, standard cells, I/O pads, etc.) are defined based on connectivity and performance.
- Objectives include minimizing wire length, optimizing area, and ensuring proper power and clock distribution.

2. **Placement:**

- The standard cells (logic gates, flip-flops, etc.) are placed within the defined floorplan.
- Placement tools optimize for area, timing, and power consumption.
- Considerations like congestion and design rule compliance are considered.

3. **Clock Tree Synthesis (CTS):**

- A robust clock distribution network (clock tree) is designed to minimize clock skew and ensure that all parts of the chip are synchronized.
- This step is critical for timing closure, ensuring the IC operates correctly at the desired clock frequency.

4. **Routing:**

- After placement, the connections between components (nets) are physically routed using metal layers.
- Tools ensure routing adheres to design rules (e.g., spacing and layer constraints).
- Routing is optimized for signal integrity, power efficiency, and minimal delays.

5. **Power Planning:**

- Proper power distribution is ensured to minimize voltage drops and prevent hot spots.
- Power grids are designed to supply adequate power to all components of the chip.

6. **Design Rule Check (DRC):**

- A verification step to ensure the design adheres to the manufacturing process's design rules.
- Any violations are corrected before moving forward.

7. **Equivalence Checking:**

- Ensures that the physical layout matches the original netlist generated during synthesis.
- This is a critical step to confirm that no logical discrepancies exist in the design.

8. **Parasitic Extraction:**

- Parasitics (resistance, capacitance, and inductance) introduced by the physical layout are extracted.
- Accurate parasitic information is essential for post-layout simulations.

9. **Timing Signoff:**

- Using Static Timing Analysis (STA), the design is analysed to ensure it meets all timing requirements, including setup and hold times.
- Iterative adjustments may be made to meet these constraints.

10. **Power and Signal Integrity Analysis:**

- Verifies that the power distribution network can handle the chip's power demands without excessive noise or voltage drops.
- Signal integrity checks ensure there are no issues like crosstalk or electromagnetic interference.

11. **Final GDSII Generation:**

- The verified physical layout is converted into GDSII format, the standard file format used for IC fabrication.
- This marks the "tape-out" phase, where the design is finalized and sent to the foundry for manufacturing.

This complete semi-custom IC design flow is achieved using the following Cadence EDA tools:

- **Cadence Incisive® Enterprise Simulator**
  Used for RTL simulation and verification. It provides functional verification of the design through testbenches and waveform analysis.

- **Cadence Xcelium™ Parallel Logic Simulator**
  A next-generation simulator used for faster and more efficient RTL simulation. Supports advanced features like parallel execution and low-power simulation.

- **Cadence Genus™ Synthesis Solution**
  Performs logic synthesis from RTL to gate-level netlist. Optimizes the design for area, power, and timing constraints.

- **Cadence Innovus™ Implementation System**
  Used for complete physical design including floorplanning, placement, clock tree synthesis (CTS), routing, and GDSII generation. Ensures DRC/LVS clean layout and timing closure.

3. PROJECT IMPLEMENTATION AND DISCUSSION

This section describes the detailed step-by-step implementation of the 64-bit ALU design, from RTL to GDSII, using Cadence's digital IC design tools: **Incisive**, **Xcelium**, **Genus**, and **Innovus**.

3.1 Front End Design

Steps such as specification, design, verification of HDls and synthesis of RTL is categorized here

3.1.1 Specification

The design flow of all IC design methodologies always starts with the specifications, where the functionality and operating conditions among other parameters relevant to the product's use are formally specified.

Project Name:

64-bit ALU

Description:

The **64-bit Arithmetic Logic Unit (ALU)** is a critical combinational logic block designed to perform a wide range of arithmetic and logical operations on two 64-bit input operands. It serves as a fundamental component of CPUs, DSPs, and embedded processors, executing the core data-processing functions required by software instructions.

Key Features:

- **Operands:** Two 64-bit input buses (A[63:0], B[63:0])
- **Operations Supported:**
  - Arithmetic: **Addition**, **Subtraction**
  - Logical: **AND**, **OR**, **XOR**, **NOT**
  - Shift: **Logical Left Shift**, **Logical Right Shift**
  - Comparison: **Set on Less Than (SLT)**
- **Control Signals:** A multi-bit **opcode/control signal** selects the operation to be performed.
- **Output:**
  - **Result[63:0]**: 64-bit output result

- **Zero Flag**: Indicates if the result is zero
- **Overflow Flag**: Indicates signed overflow in arithmetic operations
- **Carry Out**: Shows carry generation in addition/subtraction

Design Approach:

- Built hierarchically using **1-bit ALU slices** in a **ripple-carry or carry-lookahead structure**.
- Modular and scalable design enables easy extension to higher bit-widths.
- Designed in **Verilog HDL**, verified through extensive testbenches simulating all operations and edge cases.
- Optimized for **speed, area, and power** using synthesis and physical design tools.

Functionality:

The **64-bit Arithmetic Logic Unit (ALU)** is a fundamental digital component designed to perform high-speed arithmetic and logical operations on two 64-bit binary operands. It forms a core part of modern processors, performing essential data processing tasks in microprocessors, digital signal processors (DSPs), and embedded systems.

Design Overview:

- The ALU is implemented using **modular Verilog HDL**, allowing scalability, clarity, and reusability.
- Internally, the design consists of **1-bit ALU slices** connected to form a 64-bit wide operation unit.
- The structure supports integration with processor datapaths and is optimized through synthesis and placement tools.The ALU accepts two 64-bit input operands A[63:0] and B[63:0], and a control signal Opcode[3:0] that selects one of several supported operations:

Opcode	Operation	Function
0000	Addition	Result = A + B
0001	Subtraction	Result = A - B
0010	Bitwise AND	Result = A & B
0011	Bitwise OR	`Result = A
0100	Bitwise XOR	Result = A ^ B
0101	Logical Left Shift	Result = A << 1
0110	Logical Right Shift	Result = A >> 1
0111	Set Less Than (SLT)	Result = (A < B) ? 1 : 0
1000	Bitwise NOT (on A)	Result = ~A

The ALU generates the following **status flags** based on the result:

- **Zero Flag**: High if the result is 0
- **CarryOut**: Indicates carry generation in arithmetic operations
- **Overflow**: Indicates signed arithmetic overflow
- **Negative**: High if the result is negative (MSB = 1)

Applications:

- Integral part of CPUs and custom processor designs
- Digital signal processing blocks
- Control units in embedded and IoT systems
- High-speed mathematical computation units

3.1.2 RTL Design

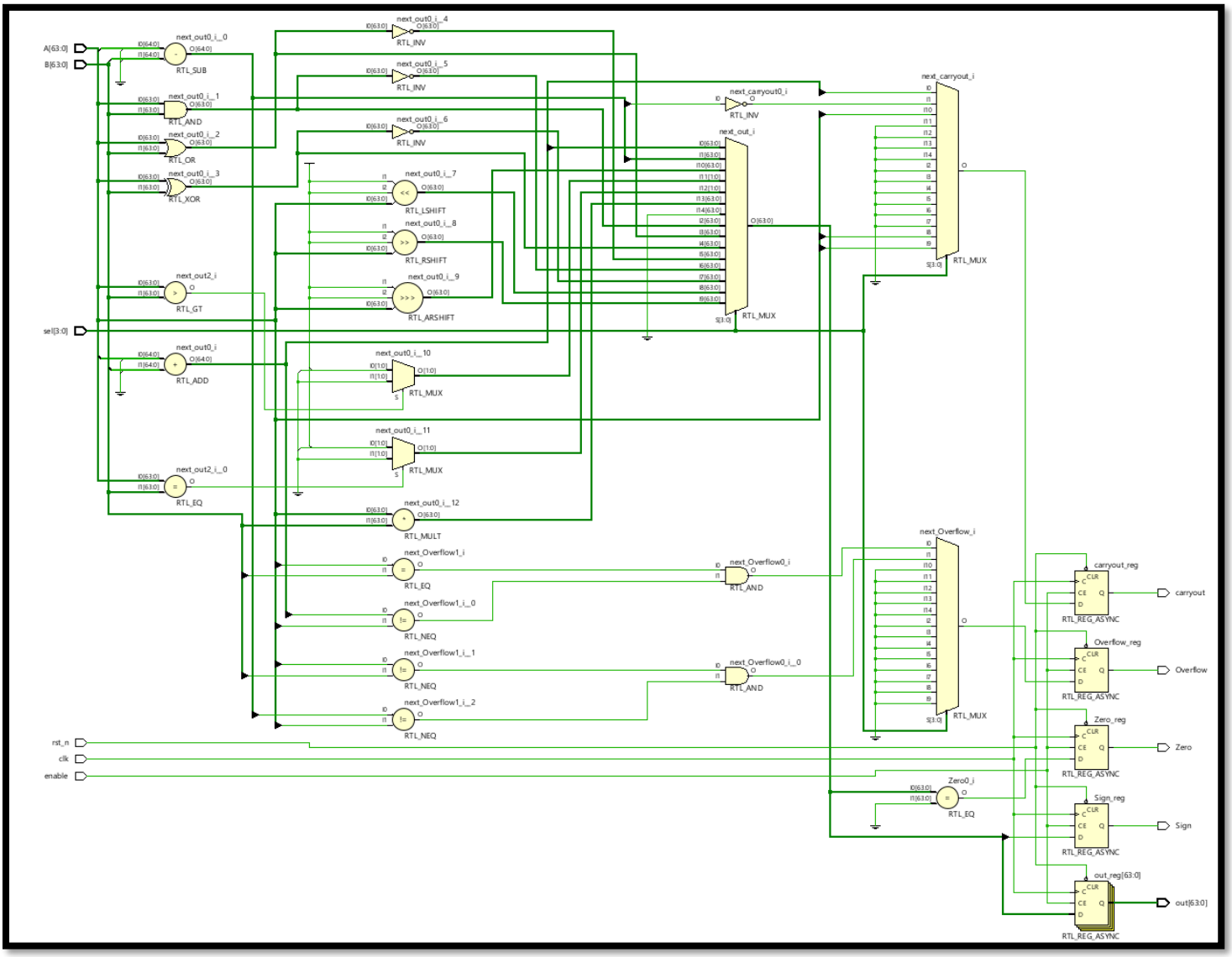


Figure 3.1: RTL Schematic for 64-bit ALU-Xilinx Vivado

Figure 3.2: Verilog Code for 64-bit ALU

3.1.3 RTL Verification

Front-End RTL Verification refers to the process of ensuring that a digital design, expressed at the Register Transfer Level (RTL) using hardware description languages (HDLs) such as Verilog or VHDL, meets its intended functional specifications. This verification is a critical step in the pre-silicon design lifecycle, as it identifies and rectifies potential design flaws before the design is sent for physical implementation.

64-bit ALU testbench:

The **testbench for the 64-bit Arithmetic Logic Unit (ALU)** is developed in **Verilog** to functionally verify the correct behavior of the ALU under different operations and input conditions. It serves as a controlled environment to apply stimulus to the ALU inputs and observe the output responses.

Purpose:

The primary objective of the testbench is to ensure that the ALU performs all intended operations correctly including arithmetic, logical, shift, and comparison functions — and correctly asserts status flags such as **zero**, **carryout**, **overflow**, and **negative**.

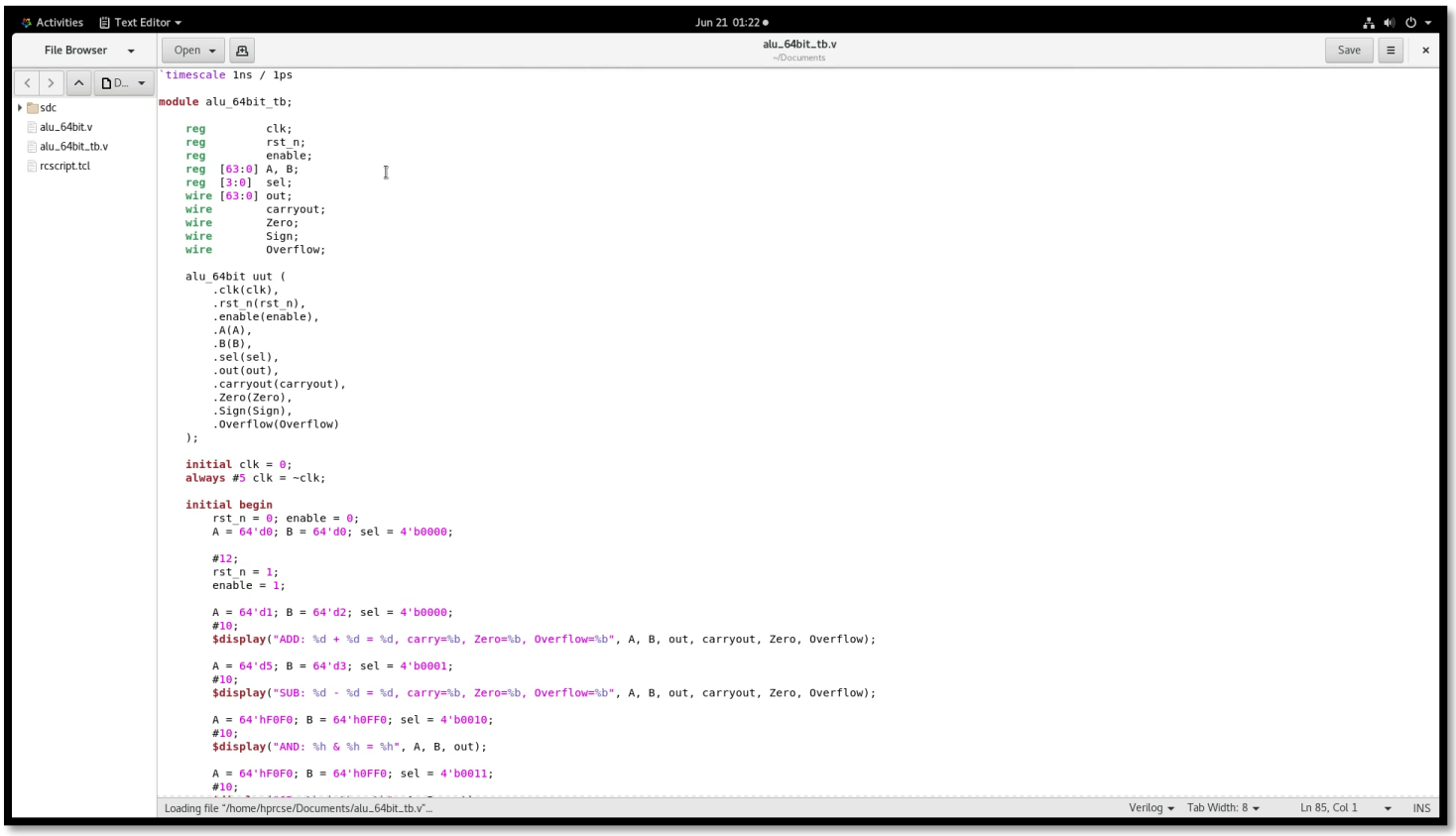


Figure 3.3: 64-bit ALU - Part 1 / 2

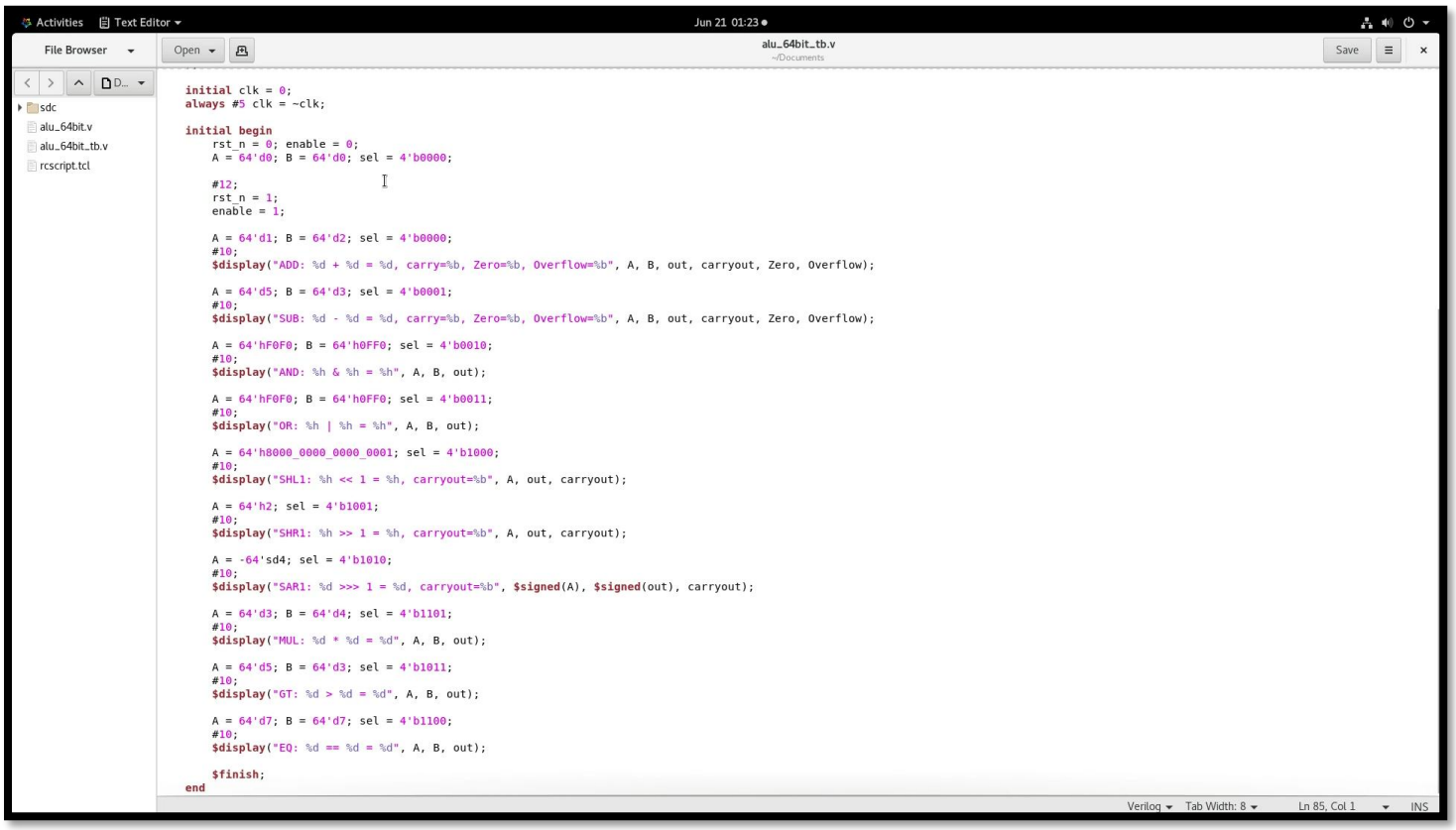


Figure 3.3: 64-bit ALU testbench – Part 2 / 2

Cadence Semi-Custom Design Flow Using Incisive, Xcelium, Genus, and Innovus

File Structure Example:

project/

├── rtl/

| ├── alu_64bit.v

| └── alu_64bit_tb.v

├── synth/

| ├── constraints.sdc

| └── tech_lib.db

├── pd/

| ├── synthesized_netlist.v

| ├── floorplan.tcl

| └── techlef.lef

| └── io.sdc

1. RTL Simulation Using Incisive and Xcelium

RTL simulation of the alu_64bit design can be performed using Cadence's Incisive or Xcelium simulator. For GUI-based simulation, open a terminal and launch NC Launch using the command nclaunch &. In the NC Launch GUI, add your source files (alu_64bit.v and alu_64bit_tb.v), set the top-level module as alu_64bit_tb, enable FSDB waveform generation, and run the simulation. You can view the generated waveforms using SimVision.

2. RTL Synthesis Using Genus

genus

In Genus TCL Shell:

- read_hdl rtl/alu_64bit.v
- read_sdc synth/constraints.sdc
- set_db / .library tech_lib.db
- elaborate
- syn_generic
- syn_map

- syn_opt
- write_hdl > synthesized_netlist.v
- write_sdc > post_synth.sdc
- report_timing > timing.rpt
- report_area > area.rpt

3. Physical Design Using Innovus

innovus

In Innovus TCL Shell:

- read_lef pd/techlef.lef
- read_def synthesized_netlist.v
- read_sdc post_synth.sdc
- floorPlan -core
- placeDesign
- clockDesign
- routeDesign
- optDesign
- verifyGeometry
- verifyConnectivity
- write_gds final.gds

Tool Summary

Step	Tool	Launch Command
RTL GUI Simulation	**Incisive**	nclaunch &
RTL CLI Simulation	**Incisive**	irun -alu_64bit -access +rwc alu_64bit_tb.v
Fast Simulation	**Xcelium**	xrun -alu_64bit.v alu_64bit_tb.v -access +rwc -gui &
RTL Synthesis	**Genus**	genus
Physical Design	**Innovus**	innovus

1. Cadence Incisive

Cadence Incisive is a comprehensive digital simulation and functional verification platform used in the VLSI design industry. It provides simulation tools for verifying Register Transfer Level (RTL) designs written in Verilog, VHDL, or SystemVerilog. It supports both GUI and command-line workflows, making it ideal for both beginners and professionals.

Key Features:

- **RTL Simulation**: Fast and accurate simulation of Verilog/VHDL designs
- **Integrated GUI**: Includes **NC Launch** for flow setup and **SimVision** for waveform viewing and debugging
- **Mixed-Language Support**: Supports Verilog, VHDL, SystemVerilog in one environment
- **Waveform Dumping**: Supports **FSDB** waveform format for efficient signal tracing
- **Assertions and Coverage**: Supports assertion-based verification and code/functional coverage metrics
- **Debug Tools**: Provides **source-level debugging**, breakpoints, and signal tracing with SimVision
- **Automation Friendly**: Supports batch-mode simulation using irun command

Tool Components:

Component	Purpose
nclaunch	GUI-based project setup and simulation launch
irun	Command-line simulation tool
simvision	Graphical waveform viewer and signal debugger
novas.fsdb	Output waveform file (Fast Signal DB format)

Typical Use in Design Flow:

1. **Design Entry**: RTL code in Verilog/VHDL
2. **Testbench Development**: Write testbench to apply stimulus
3. **Simulation**: Run using irun or nclaunch
4. **Debugging**: Analyze waveforms with SimVision
5. **Verification Sign-off**: Validate correctness before synthesis

Why Use Incisive in VLSI Projects?

- Industry-standard tool for **RTL verification**
- Easy **integration with Verdi**, Genus, and Innovus
- Scalable for **large SoC designs** with mixed-language environments

Incisive Full Simulation Flow – Step-by-Step Guide

Step 1: Open Terminal & Setup Environment

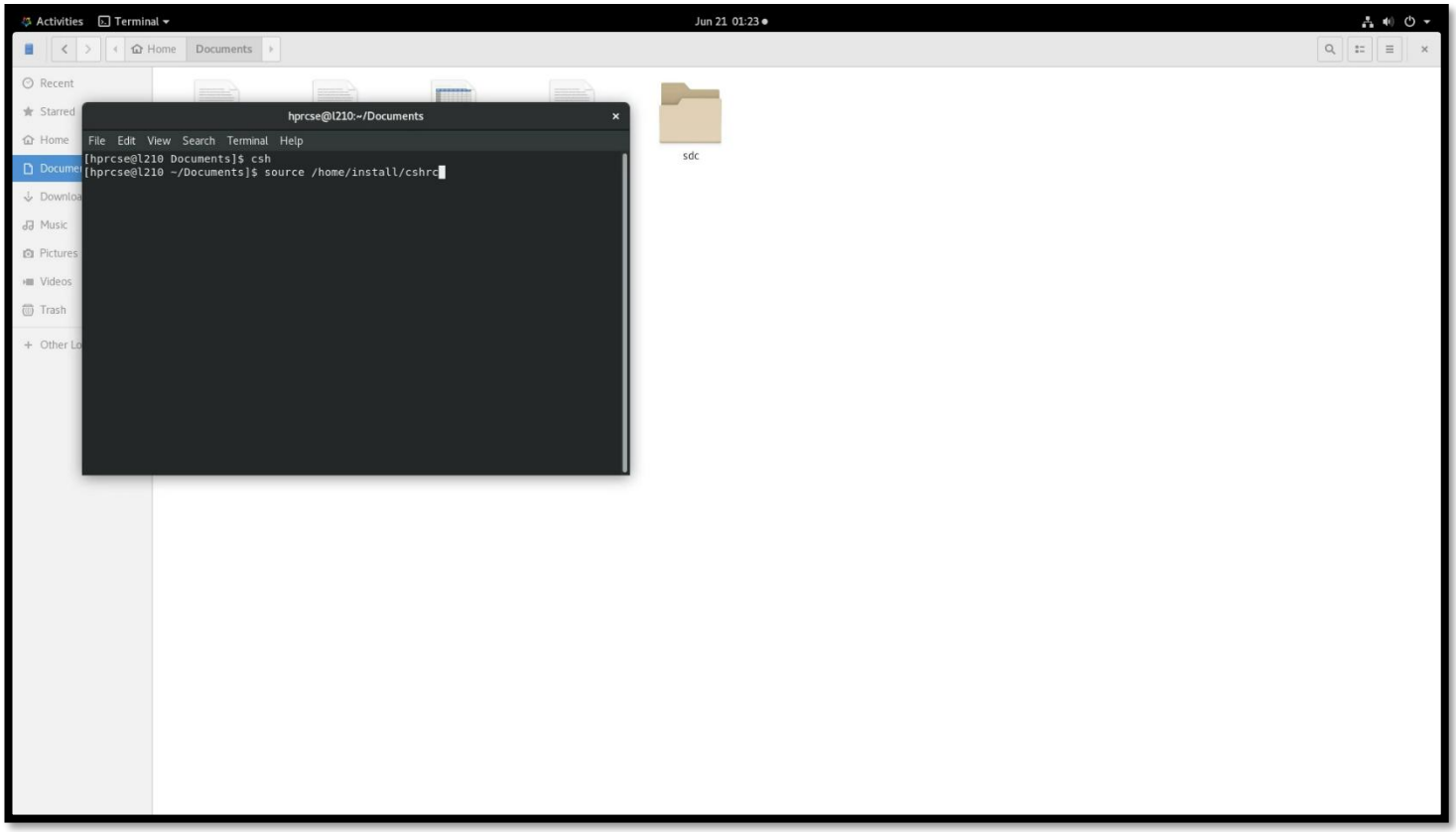


Figure 1: Cadence Installation Directory

1.Open Terminal and Switch to C Shell

Open a terminal and switch to the C Shell environment by typing:

=>csh

Then, set up the Cadence environment by sourcing the cahrc file:

csh

=>source /home/install/cahrc

Reference: Figure 1 – Sourcing cshrc in C Shell to Load Cadence Tools

Step 2: Launch NCLaunch (Graphical UI)

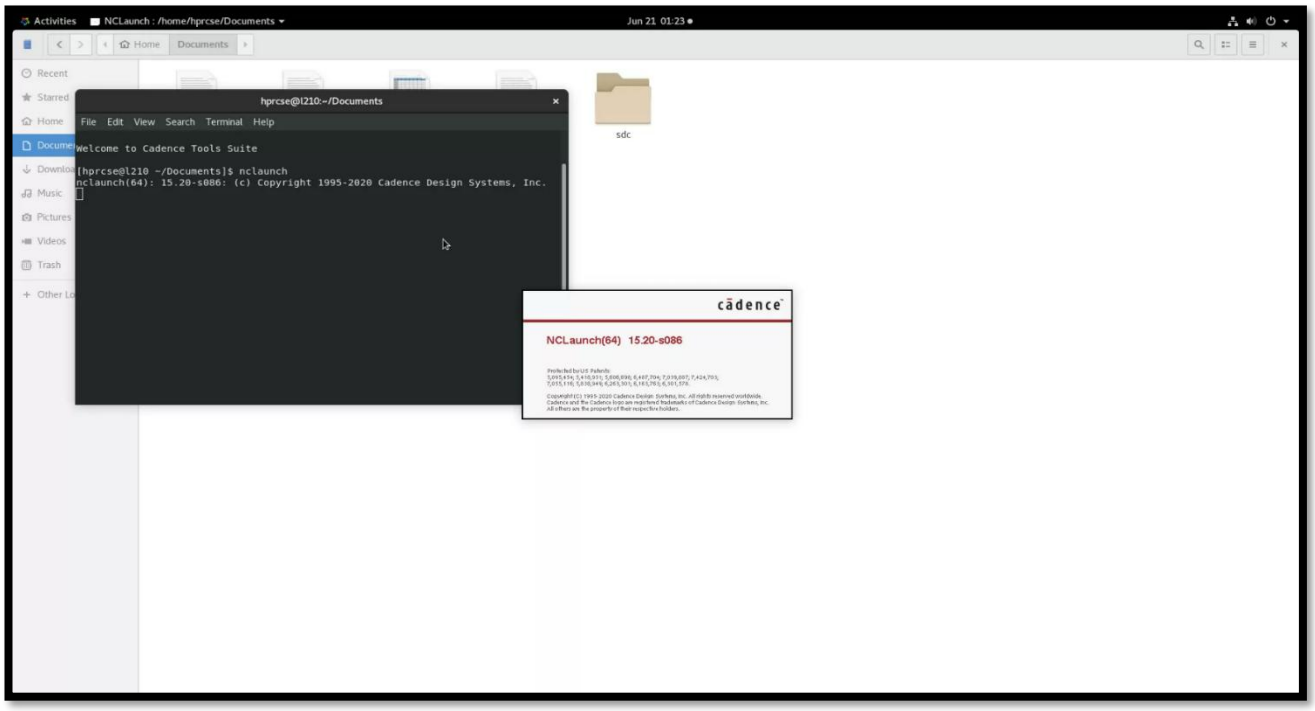


Figure 2: NCLaunch Tool Interface

2.Launch NCLaunch Tool (for Simulation Setup)

To open the graphical simulation setup interface, launch NCLaunch using:

⇒ nclaunch &

This opens the NCLaunch GUI where design files and testbenches can be configured.

Reference: Figure 2 – NCLaunch Tool Interface

Step 3: Add RTL and Testbench Files

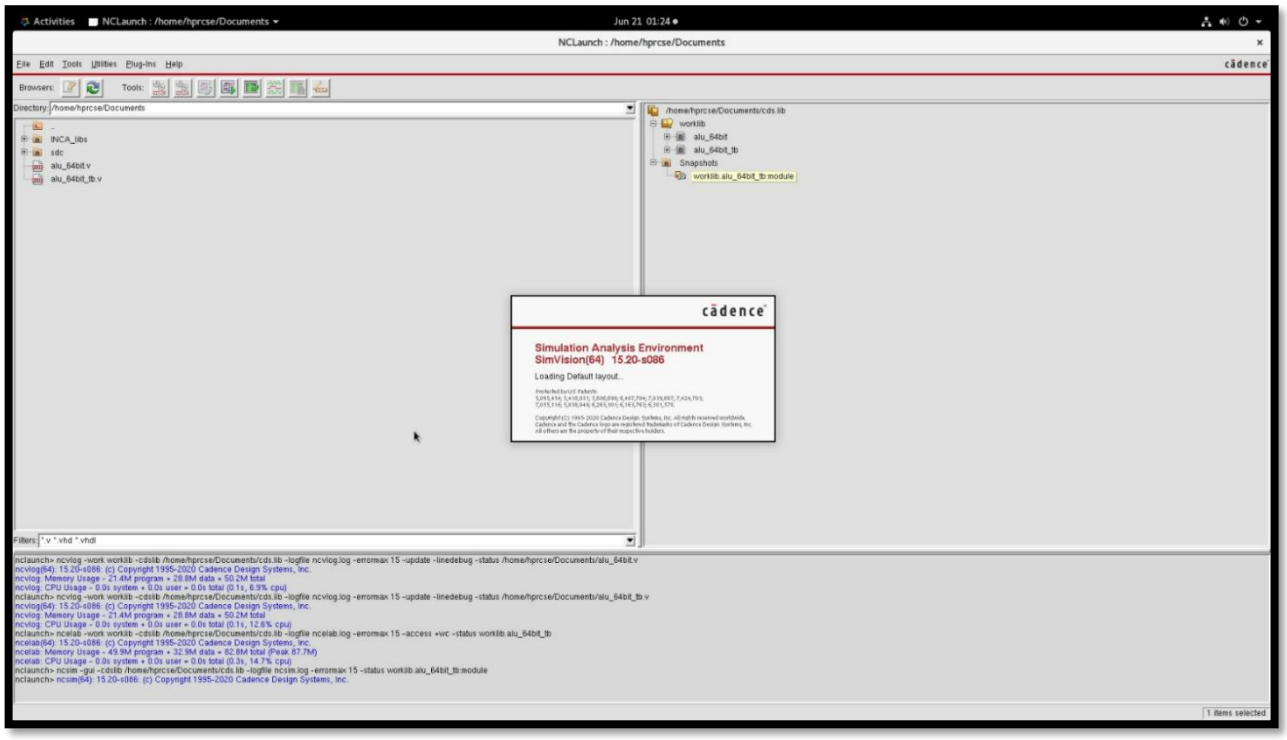


Figure 3: Adding Design and Testbench Files in NCLaunch

3.Add Design and Testbench Files

Inside the NCLaunch GUI, click **Design → Add Files** to include your Verilog/SystemVerilog source files and testbench files into the project.

Reference: Figure 3 – Adding Design and Testbench Files in NCLaunch

Step 4: Run Simulation

- Click on **Run Simulation** inside NC Launch.
- **SimVision** window will open automatically if selected.

Wait for the simulation to complete or press stop when waveforms are generated.

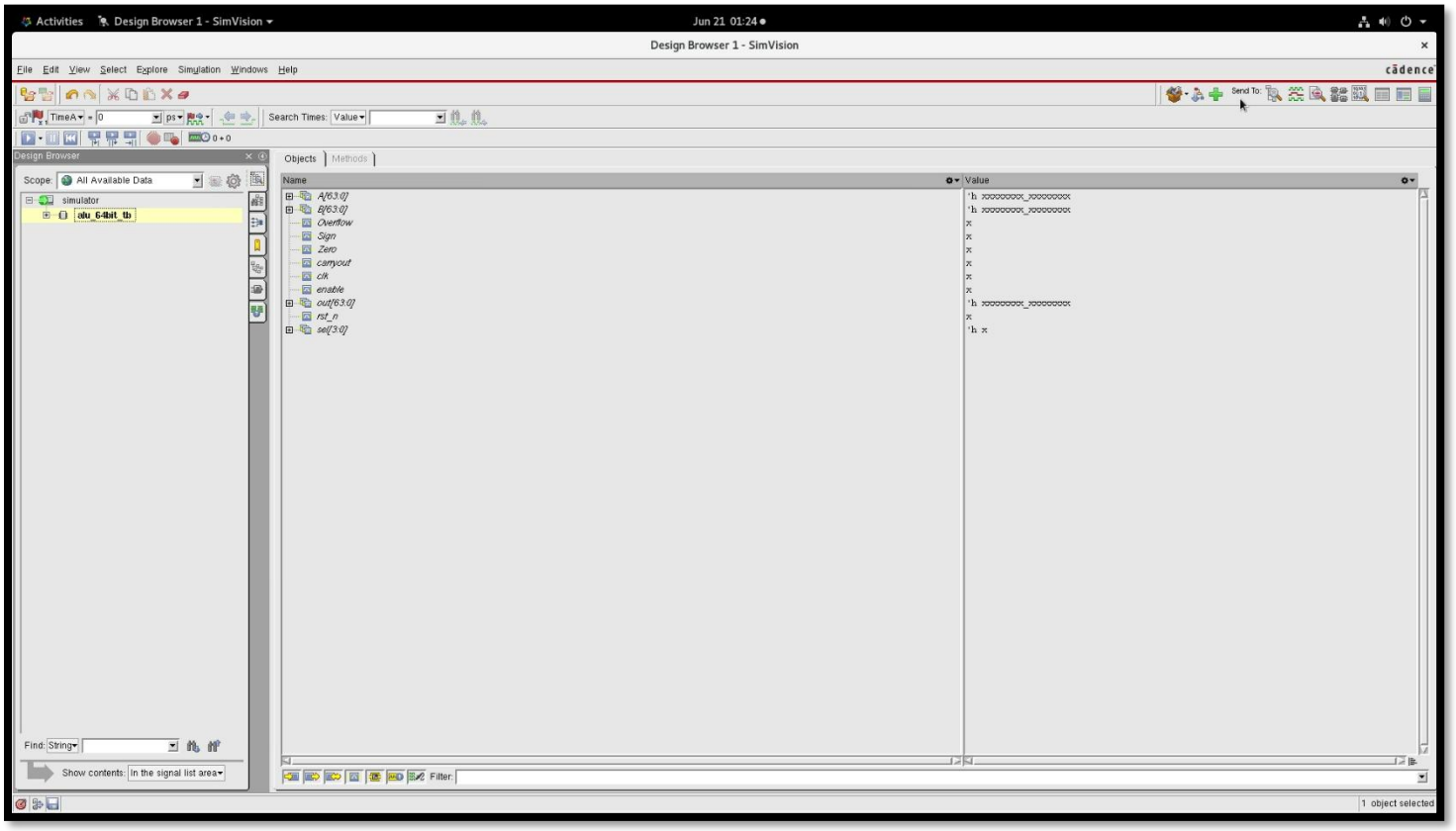


Figure 4: Running Simulation in NCLaunch

4.Run the Simulation

Click on the **Run Simulation** button or use **Simulation → Run** to start simulation. This will compile and simulate the design.

Reference: Figure 4 – Running Simulation in NCLaunch Use the **Hierarchy Browser** to navigate to signals.

Step 5: View Waveforms in SimVision:

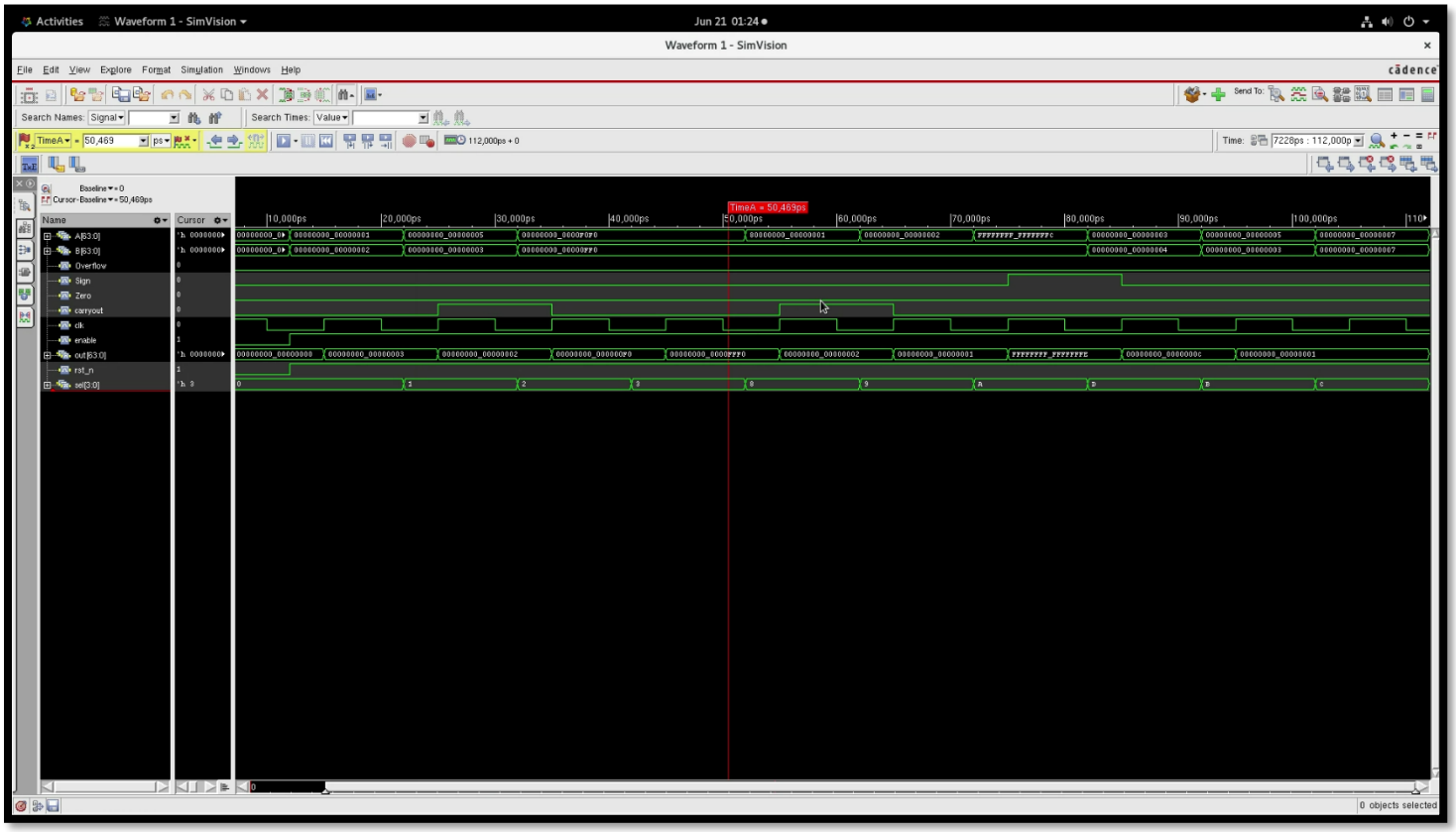


Figure 5: View Waveforms in SimVision

5. View Simulation Waveform in SimVision

After simulation completes, SimVision will open if FSDB is enabled. You can analyze signal transitions, zoom in, and debug waveform behavior.

Reference: Figure 5 – Viewing Simulation Waveform in SimVision

Step 6: FSDB Dump Code in Testbench (Important)

Ensure you have this code inside your **testbench**:

```
initial begin
  $fsdbDumpfile("novas.fsdb");
  $fsdbDumpvars(0, alu_64bit_tb);  // use your testbench name
End
```

Step 7: Command-Line Alternative (Optional)

You can skip GUI and use **Incisive command-line flow**:

```
irun -64bit -access +rwc alu_64bit_tb.v
```

- -access +rwc → Required for waveform dumping/debug access.

- If FSDB is enabled, it will generate novas.fsdb.

To view waveforms:

simvision novas.fsdb

Step 8 : Open the Incisive Simulation Console (Optional but Useful)

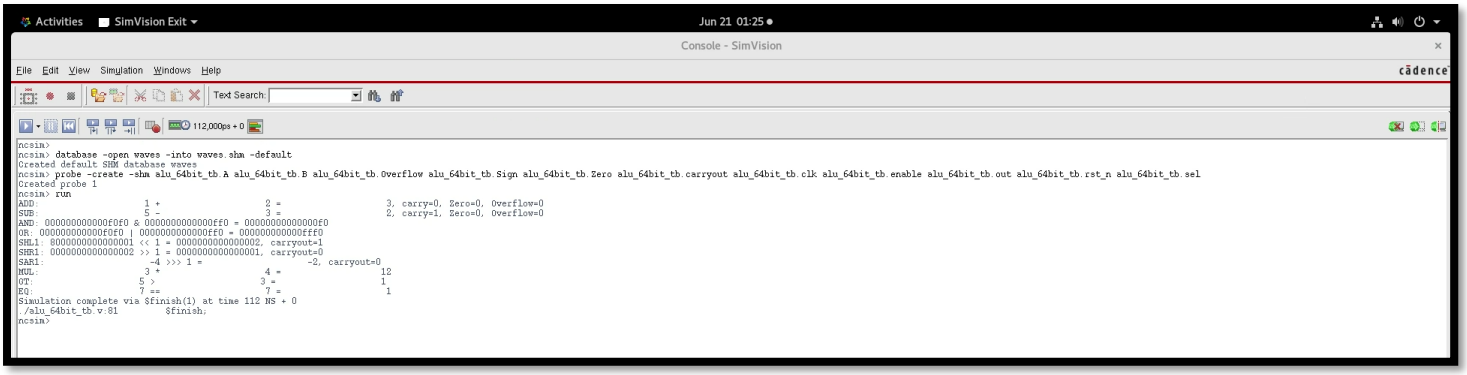


Figure 6: Testbench simulation Result

8. This figure shows the waveform output in SimVision after a successful testbench simulation, highlighting key signal transitions.

Summary

Task	**Command/Action**
Switch to csh shell	csh
Source Cadence env	source /home/install/cshrc
Launch GUI	nclaunch &
Run Sim	Run inside NC Launch
Dump Waveform (code)	$fsdbDumpfile("novas.fsdb");
Open Waveform Viewer	simvision novas.fsdb

1. Cadence Xcelium

Cadence Xcelium is a next-generation digital logic simulation and verification platform developed to replace and extend the capabilities of Cadence Incisive. It offers high-performance simulation, mixed-language support, and advanced features for scalable and accelerated verification of RTL designs. Xcelium is widely used in ASIC and SoC verification environments for its speed, flexibility, and compatibility with modern design flows.

Key Features:

- High-Performance RTL Simulation: Accelerated simulation using multithreading, parallelism, and compiler optimizations.
- Unified Simulation Kernel: Efficient for large mixed-language designs (Verilog, VHDL, SystemVerilog, SystemC).
- Waveform Dumping Support: Supports FSDB and SHM formats for waveform viewing in SimVision or Verdi.
- Integrated UVM & Formal Support: Native support for UVM, SystemVerilog Assertions (SVA), and formal verification.
- Power-Aware Simulation: Integrates with UPF/CPF for dynamic power intent verification.
- Parallel Run Options: Leverages multicore CPUs to reduce simulation runtime.
- Command-Line Friendly: Batch-mode support using xrun with flexible options and automation.

Tool Components:

Component	Purpose
xrun	Primary command-line tool for simulation and elaboration
simvision	Graphical waveform viewer and debug tool
xcelium.d	Directory storing elaboration and simulation data
xrun.fsdb	Output waveform file (Fast Signal Database format)

Typical Use in Design Flow:

1. Design Entry: RTL coding in Verilog/VHDL/SystemVerilog
2. Testbench Creation: Develop testbench using behavioral code or UVM
3. Compilation & Simulation: Execute using xrun
4. Waveform Dumping: Output FSDB file for signal-level debugging

5. Waveform Viewing: Open with SimVision

Why Use Xcelium in VLSI Projects?

- Faster Simulation Speeds than traditional Incisive tools
- Supports modern standards (SystemVerilog, UVM, SVA, VHDL-2008)
- Deep integration with SimVision, JasperGold, Genus, and Innovus
- Used in automated regression and CI flows
- Supports advanced power-aware, low-power, and concurrent assertions

Xcelium Full Simulation Flow – Step-by-Step Guide

Step 1: Open Terminal and Source Environment

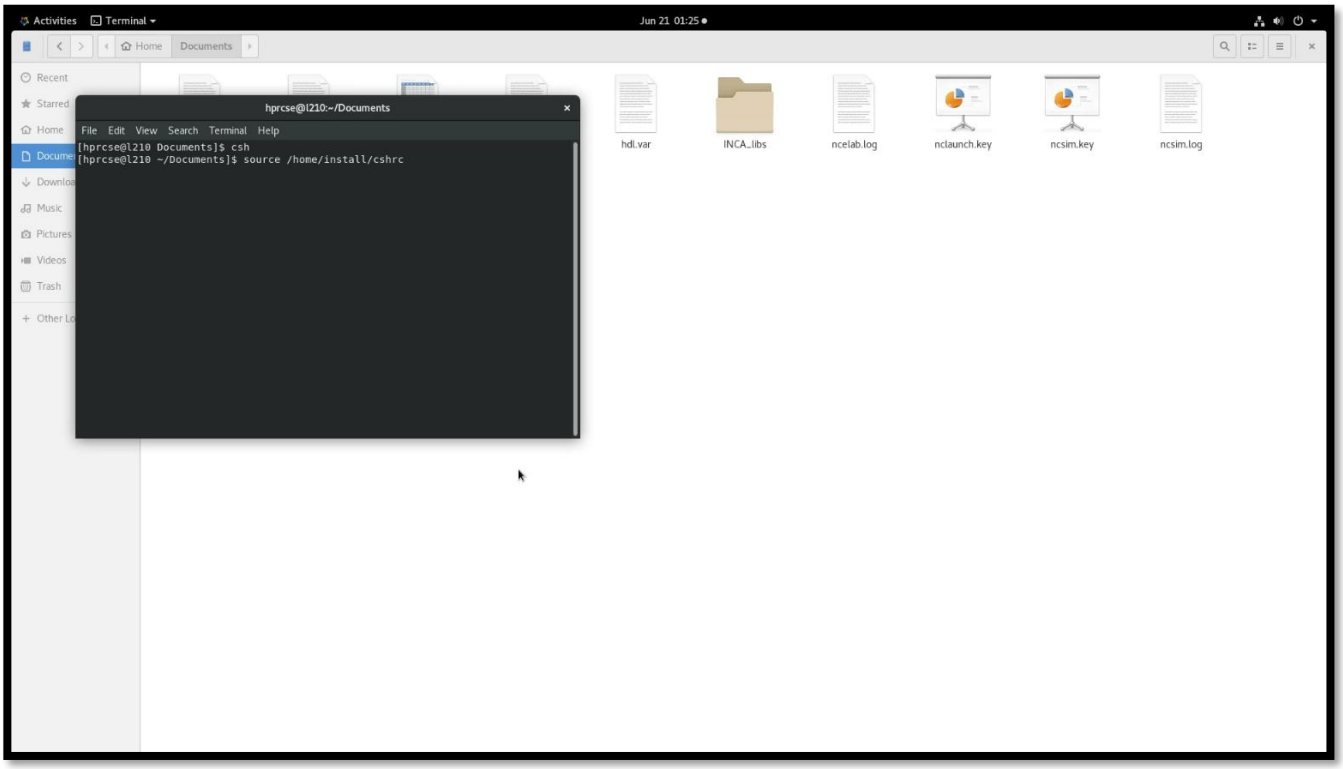


Figure 6: Cadence Xcelium Tool Installed under /home/tools/cadence/ Directory

1.Open Terminal and Switch to Shell

Open a terminal and switch to the C Shell environment by typing:

⇒ csh

Then, set up the Cadence environment by sourcing the cahrc file:

⇒ source /home/install/cahrc

Reference: Figure 6 – Sourcing cshrc in Shell to Load Cadence Tools

◇ This ensures that all environment variables for Cadence tools (including xrun, simvision) are set correctly.

Step 2: Prepare Your Design Files

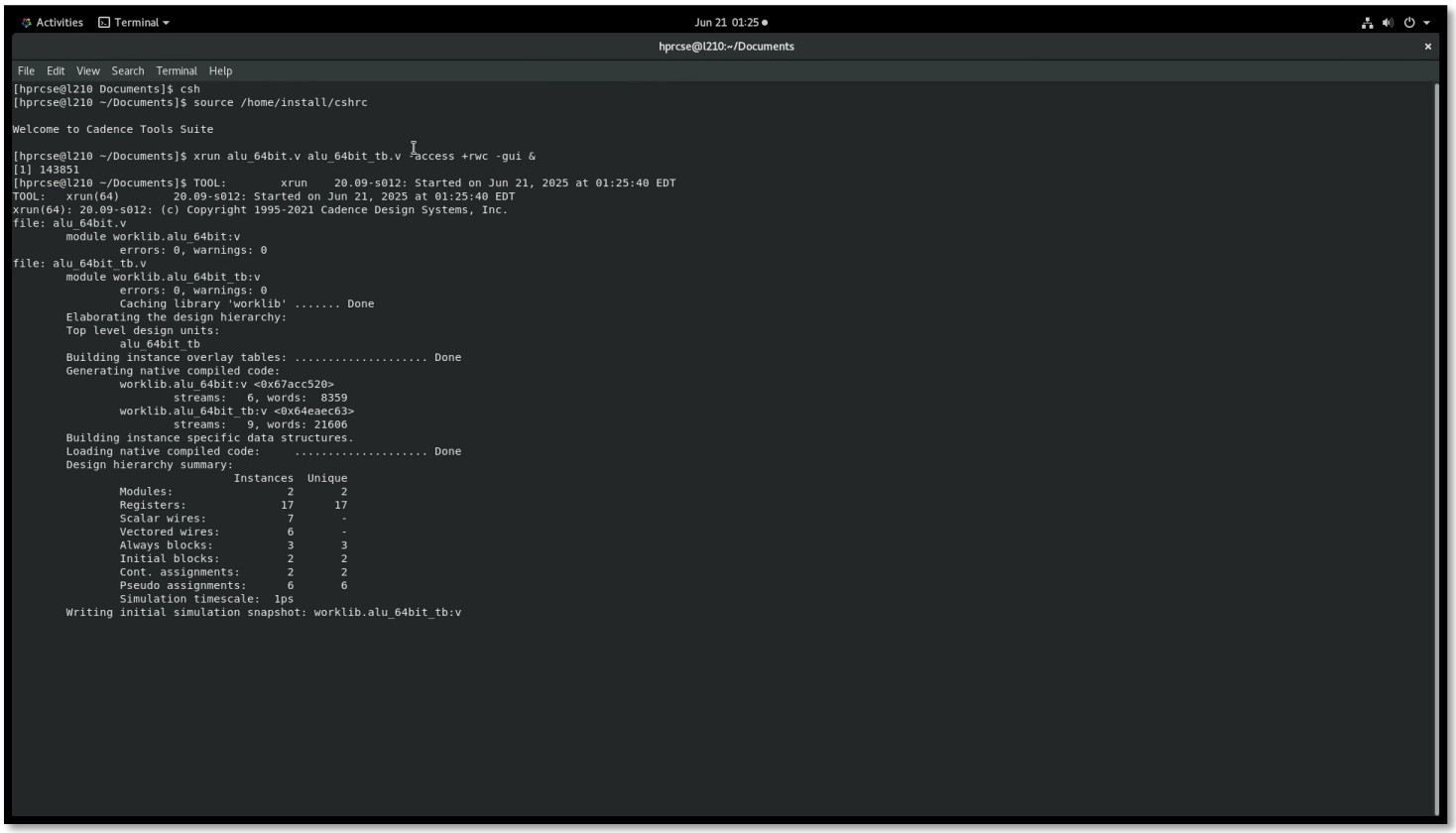


Figure 7: xrun Command in Xcelium Environment

Make sure the following files are ready:

rtl/

├── alu_64bit.v // RTL design

├── alu_64bit_tb.v // Testbench

2.Run Simulation Using xrun Command

Instead of using NCLaunch, you can simulate directly from the terminal using Xcelium with the xrun command. This method is faster and script-friendly.

⇒ xrun alu_64bit.v alu_64bit_tb.v -access +rwc -gui &

Reference: Figure 7 – xrun Command in Xcelium Environment

Step 3: Run Simulation Using Xcelium

Navigate to the project directory and use the following command:

xrun -64bit -access +rwc -linedebug -timescale 1ns/1ps alu_64bit.v alu_64bit_tb.v

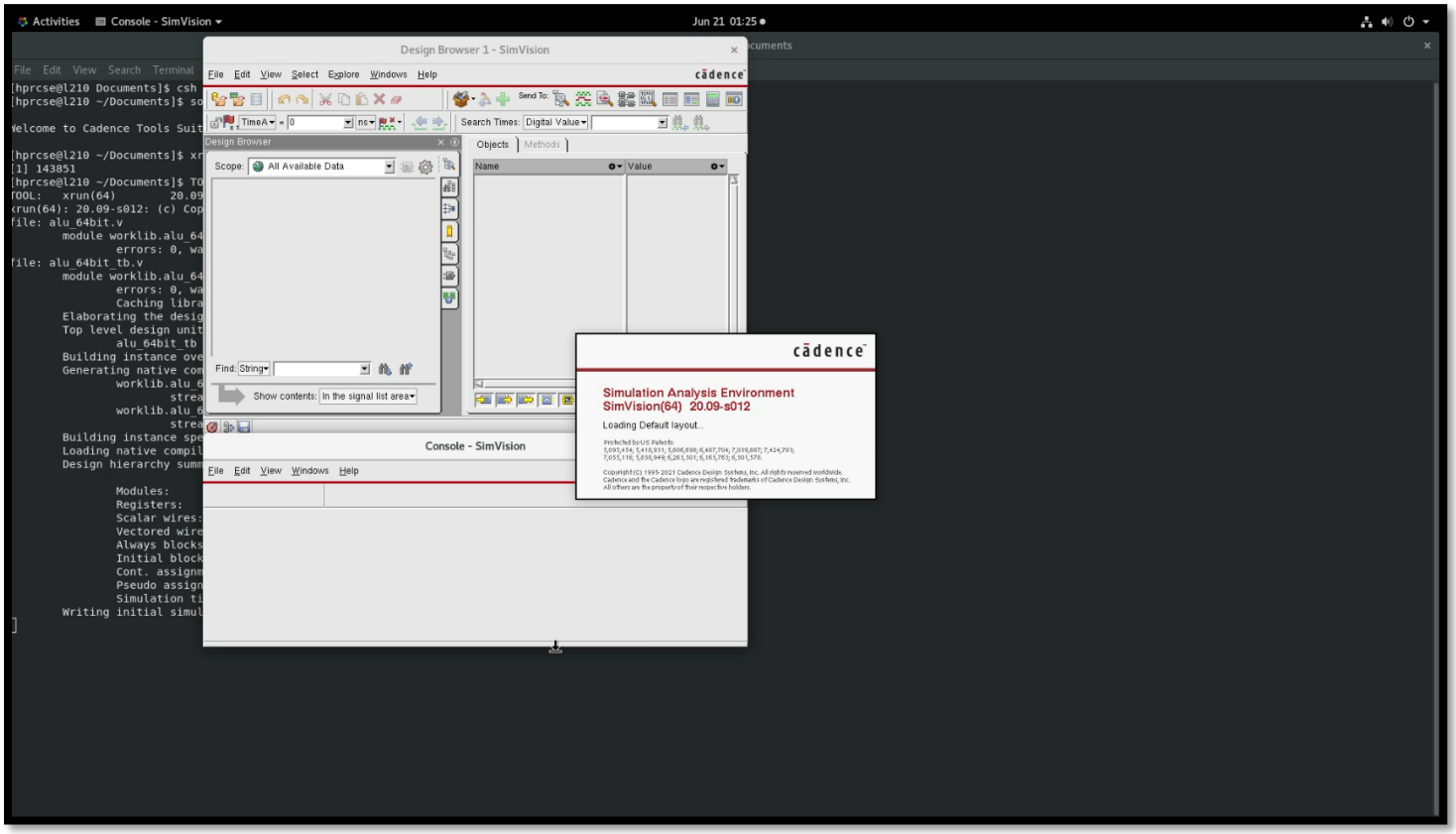


Figure 8: Running Simulation with xrun Command in Xcelium Environment

3.Run Simulation with xrun Command

Use the xrun command to compile and simulate your design in one step through the terminal. This method is efficient and commonly used in automated flows.

Example command:

⇒ xrun alu_64bit.v alu_64bit_tb.v -access +rwc -gui &

This opens SimVision after simulation, enabling waveform analysis.

Reference: Figure 8 – Running Simulation with xrun Command in Xcelium Environment

Command Breakdown:

- -64bit : Use 64-bit mode
- -access +rwc : Enable Read, Write, and Create access for all signals
- -linedebug : Enable debugging support for SimVision
- -timescale 1ns/1ps : Set timescale for simulation
- File list: RTL and testbench files

After running, it produces:

- Simulation console output
- Waveform file: xrun.fsdb
- Simulation database: xcelium.d/

Step 4: Launch SimVision for Waveform Analysis

After simulation, open waveforms with:

simvision xrun.fsdb &

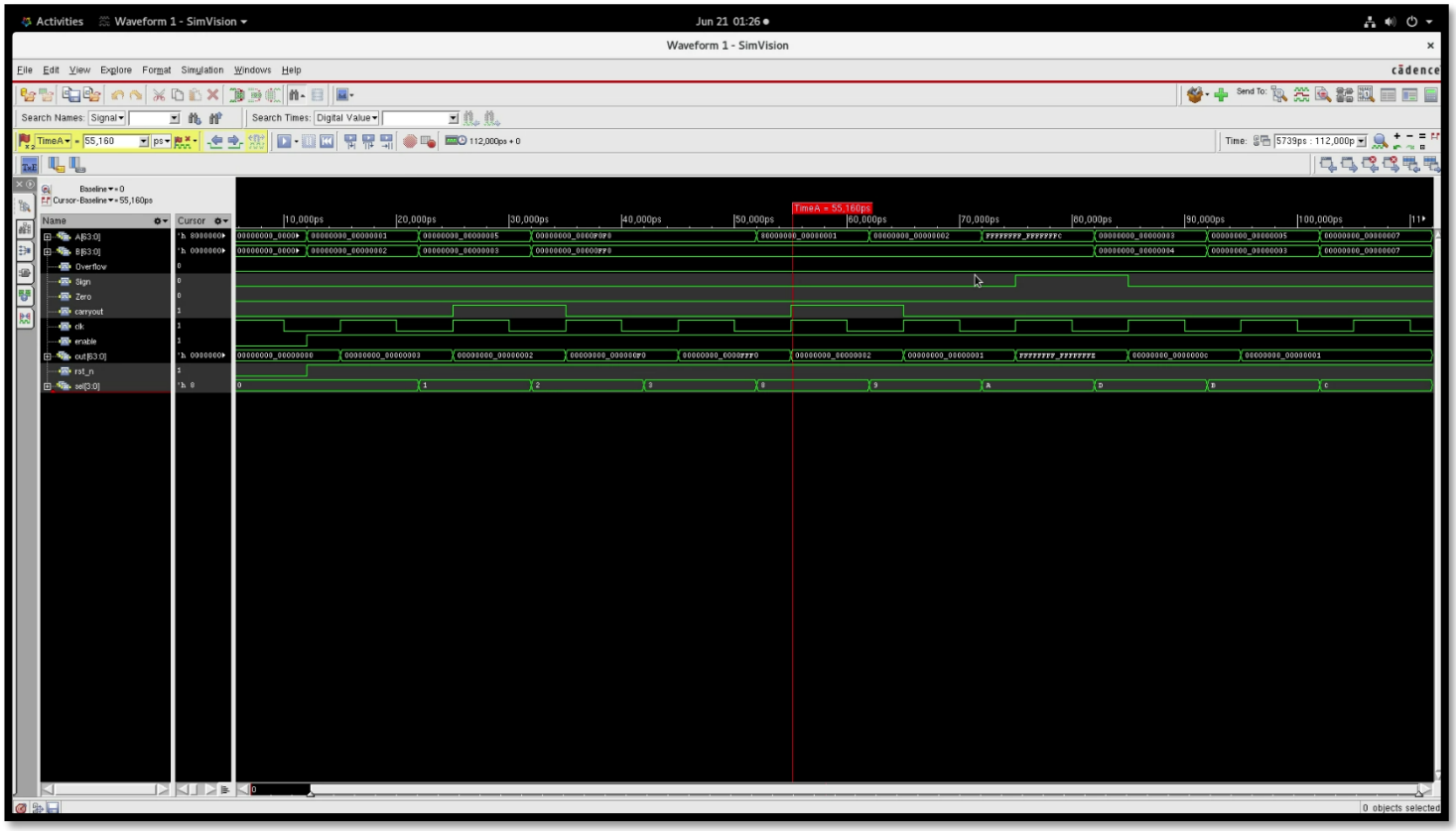


Figure 9: Viewing Simulation Waveform using SimVision after xrun Execution

4.View Simulation Waveform using SimVision

After executing the simulation using the xrun command, SimVision automatically launches if FSDB waveform dumping is enabled. In SimVision, you can view signal transitions, set cursors, measure delays, and debug timing or logic errors in your design. This step is essential for functional verification.

Reference: Figure 9 – Viewing Simulation Waveform using SimVision after xrun Execution

Step 5: Optional – Simulation GUI Launch (If Enabled)

If you want a **GUI-based simulation launch**, use:

xrun -gui -64bit -access +rwc alu_64bit.v alu_64bit_tb.v

This launches **SimVision** automatically after elaboration.

Summary of Workflow

Task	Command or Action
Setup Environment	csh → source /home/install/cshrc
Compile & Simulate	xrun -alu_64bit.v alu_64bit_tb.v -access +rwc -gui&
View Waveforms	simvision xrun.fsdb &
Optional GUI Simulation	xrun -gui -64bit -access +rwc alu_64bit.v alu_64bit_tb.v
Output Waveform File	xrun.fsdb

Advantages of Xcelium Simulation Flow

- **Fast simulation** with multicore support
- **SystemVerilog, UVM, and SVA** support
- **SimVision GUI** for waveform viewing and debugging

3.Cadence Genus – RTL Synthesis Tool

Cadence Genus Synthesis Solution is a high-performance, register-transfer level (RTL) logic synthesis tool used in the VLSI design flow. It converts RTL code written in **Verilog, VHDL, or SystemVerilog** into an optimized **gate-level netlist**, ready for physical implementation in tools like **Innovus**.

It provides advanced optimization for **area**, **timing**, **power**, and **design-for-test (DFT)** while supporting hierarchical and flat synthesis modes. Genus is used extensively in **ASIC and SoC design** projects.

Key Features:

- **High-Performance RTL Synthesis**: Fast and scalable logic synthesis engine for large SoCs
- **Multi-Mode, Multi-Corner Analysis (MMMC)**: Supports timing optimization across multiple operating modes

- **Power Optimization**: Integrates **clock gating**, **multi-Vt**, and **power-aware synthesis** techniques
- **DFT Integration**: Supports scan insertion and ATPG readiness
- **Support for Complex Architectures**: Efficient for datapath-intensive, control logic, and memory-rich designs
- **Hierarchical and Incremental Synthesis**: Helps improve run-time and memory usage
- **Constraint-Driven Flow**: Uses industry-standard **SDC (Synopsys Design Constraints)**

Genus in the Digital IC Design Flow

Step	Description
1.	Read RTL code (Verilog/VHDL)
2.	Apply SDC constraints (timing, clock, etc.)
3.	Elaborate and analyze design
4.	Perform logic optimization
5.	Generate technology-mapped netlist
6.	Export timing, area, and power reports

Typical Input Files:

- RTL files: .v, .sv, .vhdl
- Constraint files: constraints.sdc
- Technology library: .db, .lib
- Clock and reset definitions

Typical Output Files:

- Synthesized Netlist: synthesized_netlist.v
- Updated Constraints: post_synth.sdc
- Timing Report: timing.rpt
- Area Report: area.rpt
- Power Report: power.rpt

Launching Genus (TCL Mode)

Genus

Common Genus Commands (TCL)

- ❖ read_hdl alu_64bit.v
- ❖ read_sdc constraints.sdc
- ❖ io.sdc
- ❖ set_db / .library my_lib.db
- ❖ elaborate
- ❖ syn_generic
- ❖ syn_map
- ❖ syn_opt
- ❖ write_hdl > synthesized_netlist.v
- ❖ report_timing > timing.rpt
- ❖ report_area > area.rpt
- ❖ write_sdc > post_synth.sdc

Why Use Genus in VLSI Design Projects?

- Industry-proven tool for **synthesis in 16nm to 5nm process nodes**
- High-quality netlists suitable for timing-closure in **physical design**

Cadence Genus – RTL Synthesis Flow (Step-by-Step)

This guide shows how to synthesize an RTL design (e.g., a 64-bit ALU) into a gate-level netlist using **Cadence Genus**.

Step 1: Open Terminal and Setup Cadence Environment

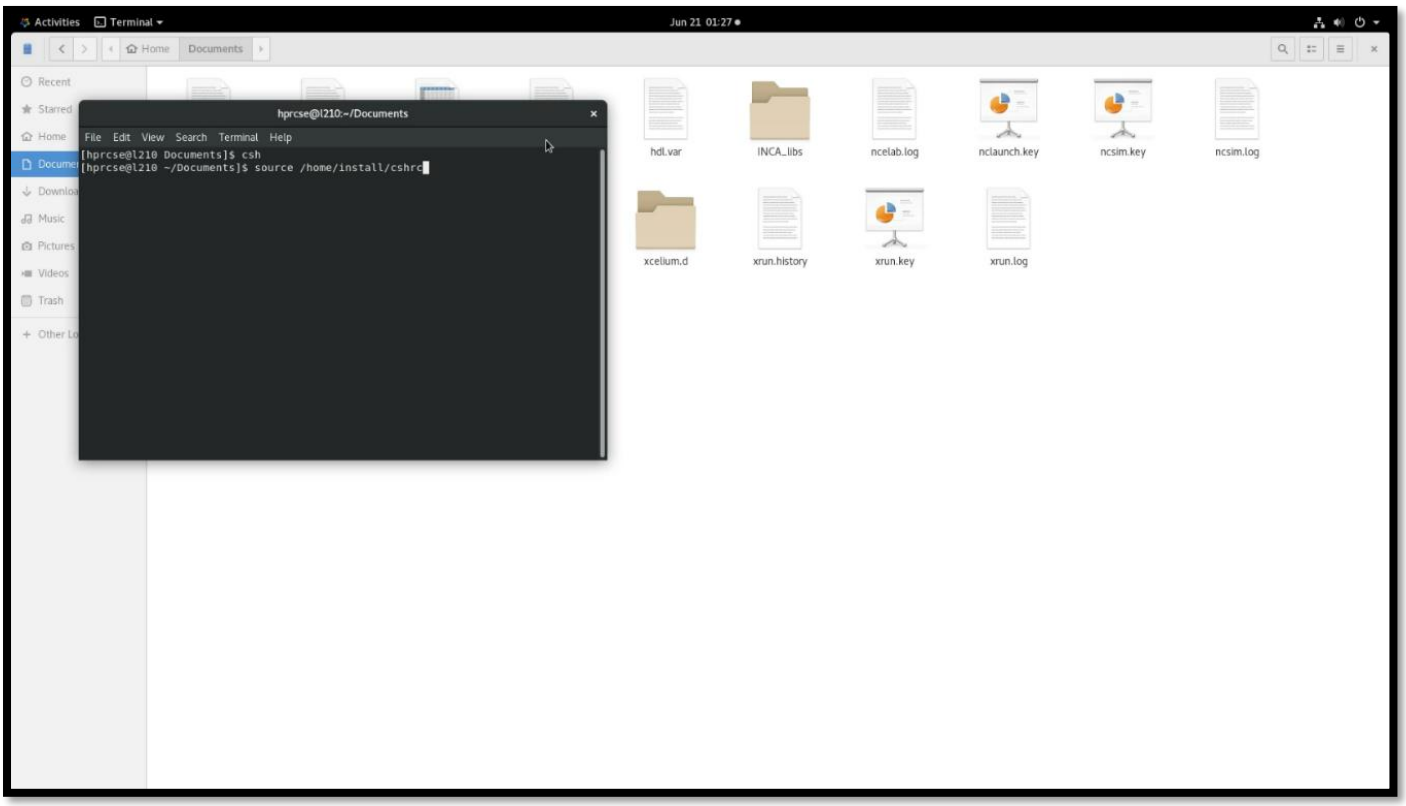


Figure 10: Cadence Genus Tool Installed Under /home/install/cadence/ Directory

1.Verify Genus Tool Installation Directory

Open a terminal and switch to the C Shell environment by typing:

⇒ csh

Then, set up the Cadence environment by sourcing the cahrc file:

⇒ source /home/install/cahrc

Reference: Figure 10– Sourcing cshrc in Shell to Load Cadence Tools

This loads Genus into your environment.

Step 2: Launch Cadence Genus

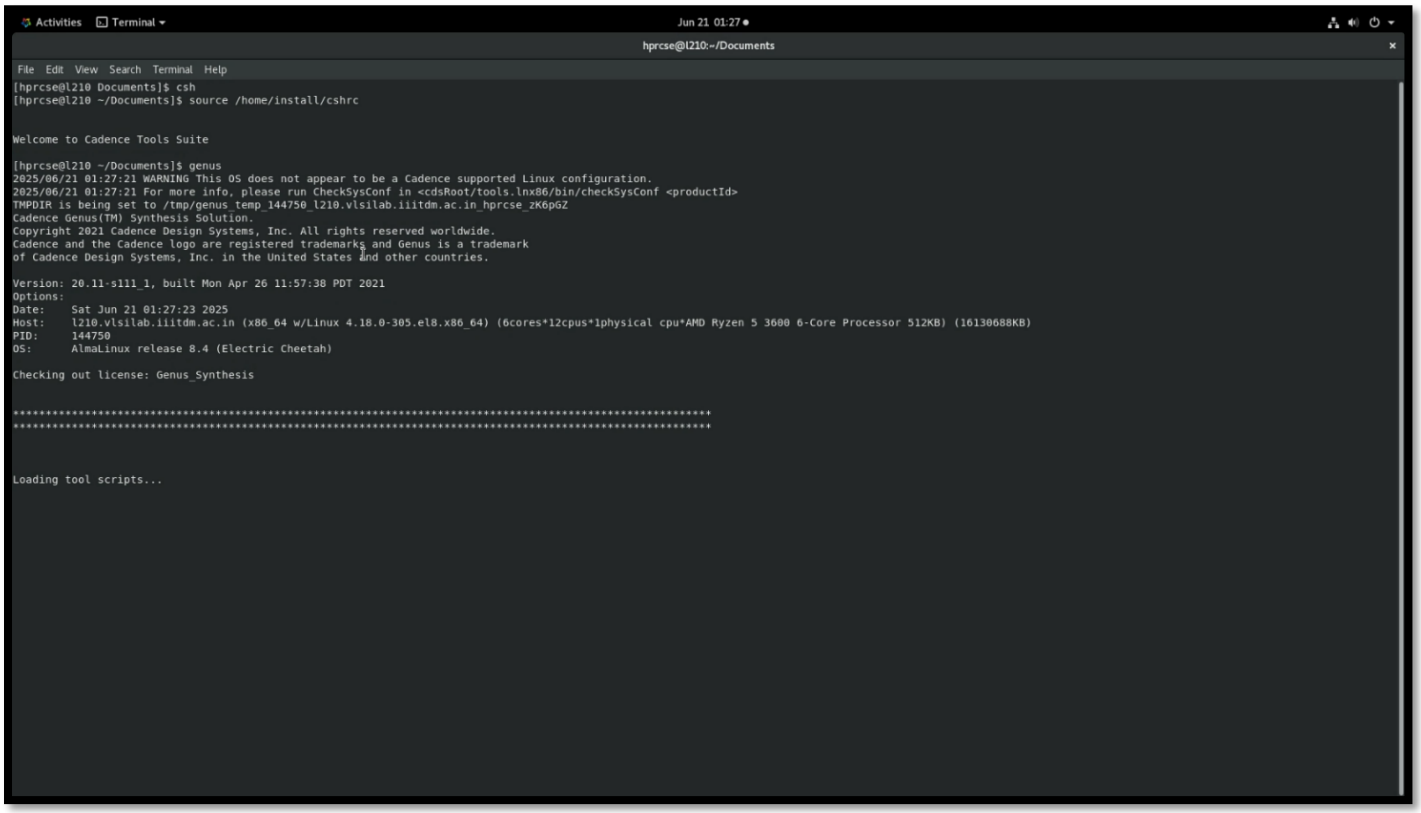


Figure 11: Launching Cadence Genus Using genus & Command in C Shell Terminal

2.Launch Cadence Genus Tool

After verifying the installation and sourcing the environment setup (cahrc), launch the Cadence Genus synthesis tool by typing the following command in the C Shell terminal:

⇒ genus &

This opens the Genus GUI where you can perform RTL synthesis, constraint loading, and netlist generation.

Reference: Figure 11 – Launching Cadence Genus Using genus & Command in C Shell Terminal

Step 3: Read RTL Design

read_hdl rtl/alu_64bit.v

You can read multiple files or use wildcards if needed:

read_hdl rtl/.v

Step 4: Read Constraints (SDC File)

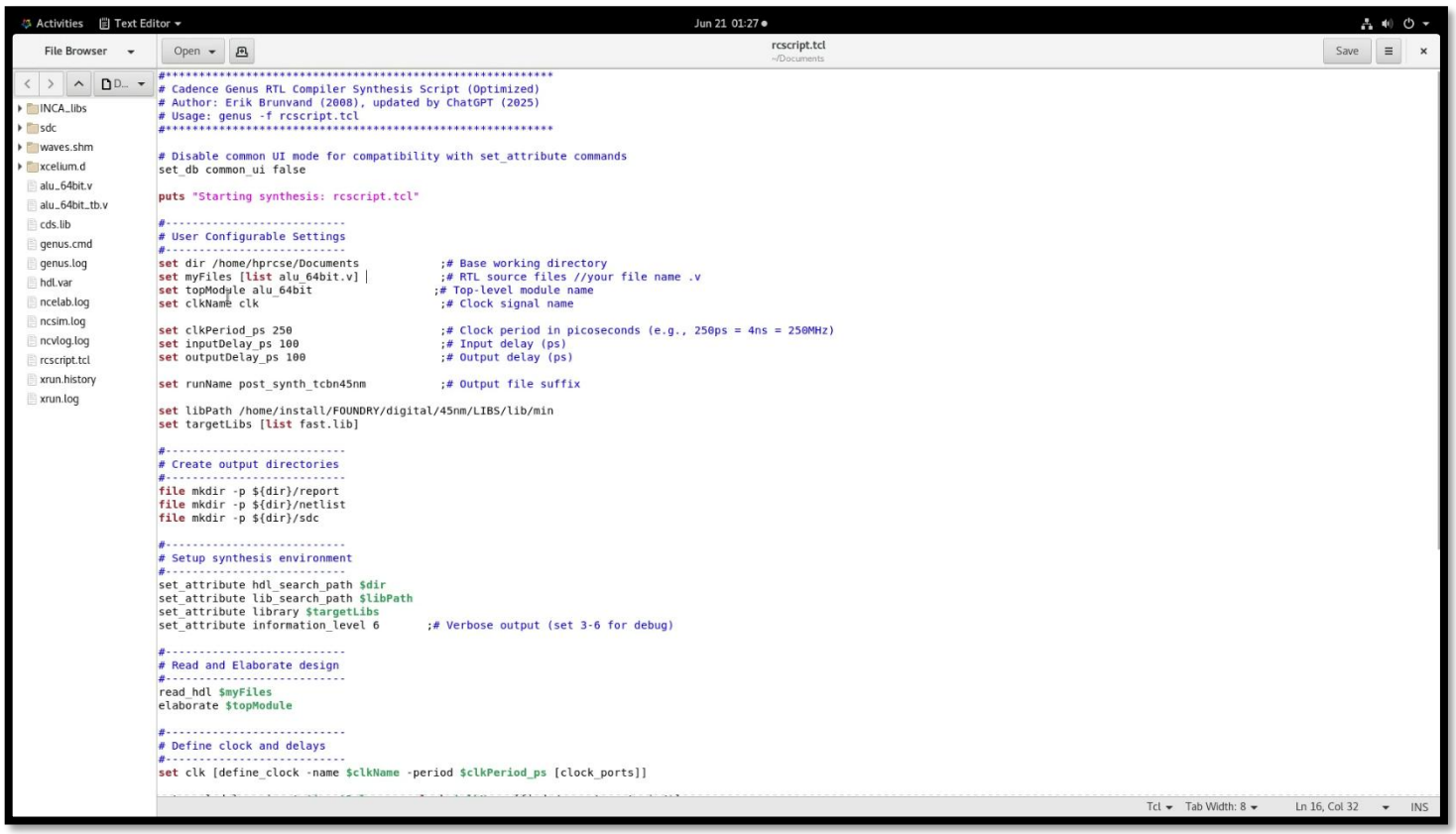


Figure 12: Viewing Constraints.sdc file

4. **Read Constraints (SDC File)**

After launching Genus, the next step is to read in your timing constraints using a Synopsys Design Constraints (SDC) file. This file defines clocks, input/output delays, and other timing exceptions. In the Genus terminal or TCL console, use the following command:

⇒ read_sdc path/to/constraints.sdc

This ensures that the synthesis process adheres to your design's timing requirements.

Reference: Figure 12 – Reading Timing Constraints Using read_sdc Command in Cadence Genus

read_sdc constraints/constraints.sdc

The SDC file defines:

- Clock definitions
- Input/output delays
- Load and timing exceptions

Step 5: Read Technology Library

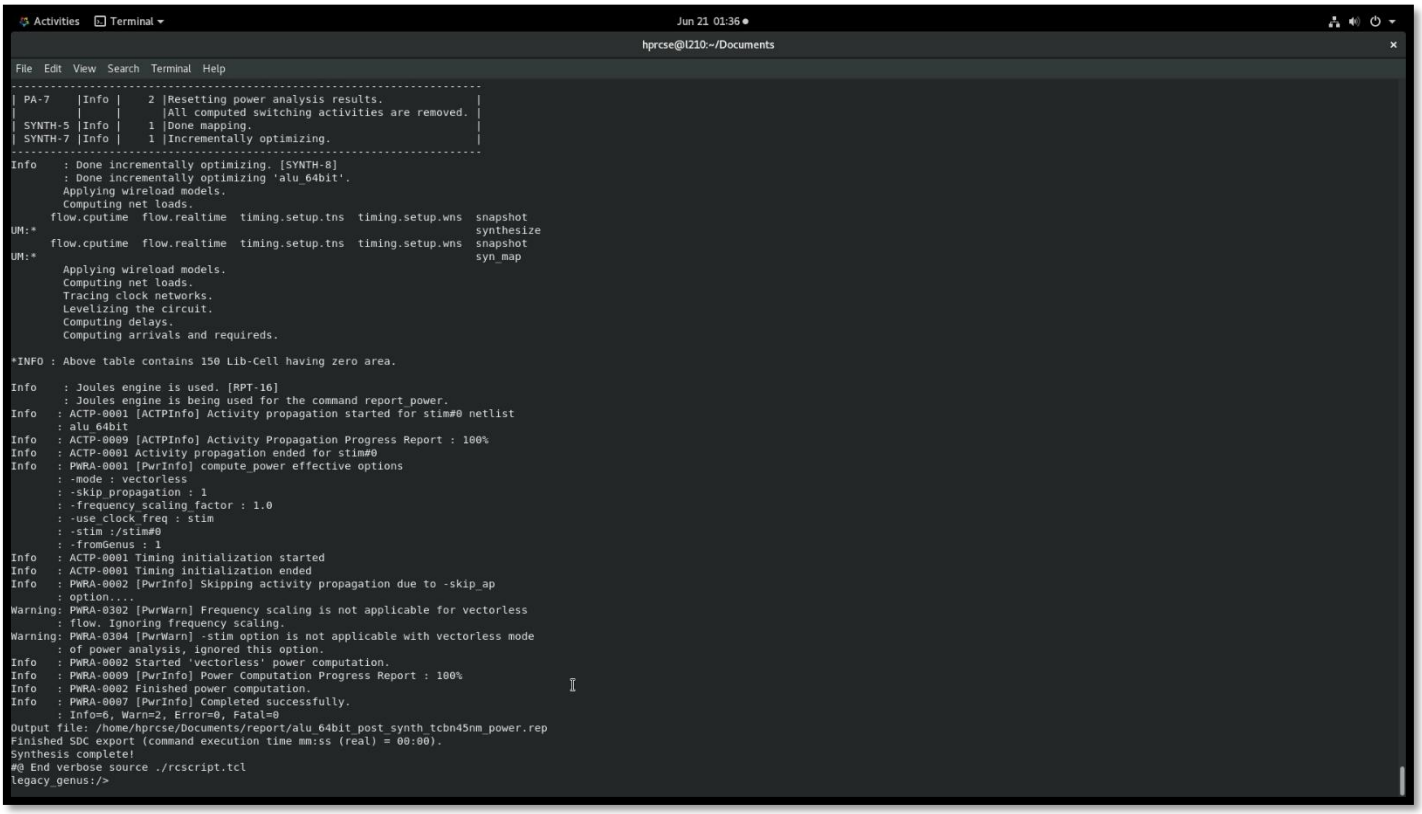


Figure 13: Source rcscript.tcl file

5.Source rcscript.tcl File

The rcscript.tcl file contains a predefined sequence of Genus commands to automate synthesis tasks like reading RTL, setting constraints, elaboration, and reporting. To execute the script, use the following command in the Genus TCL terminal:

⇒ source rcscript.tcl

This command automates the flow without requiring manual command-by-command input.

Reference: Figure 13 – Source rcscript.tcl File in Cadence Genus TCL Console

Use .db or .lib depending on availability:

set_db library lib/std_cell.dbOptional: Set link and target libraries:

set_db / .library [list lib/std_cell.db]

Step 6: Run Synthesis Stages

syn_generic ;# Technology-independent optimization

syn_map ;# Mapping to standard cells

syn_opt ;# Final optimization (timing, area, power)

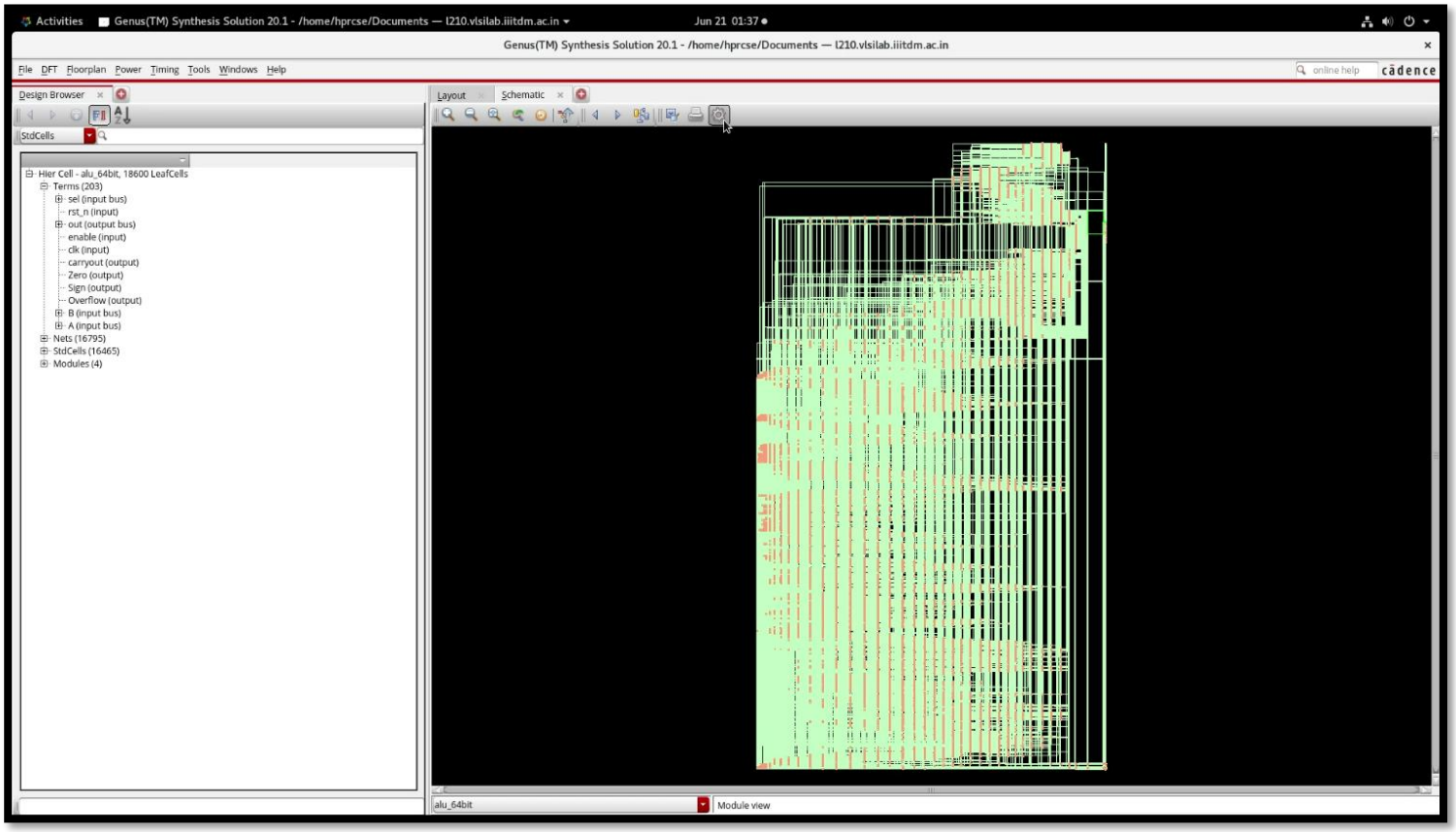


Figure 14: Viewing Synthesis for 64bit_alu

6.View Synthesis Results for 64bit_alu

Once the synthesis process is complete (either manually or through the rcscript.tcl), you can view the synthesized netlist and reports for your design module — in this case, 64bit_alu. Genus provides synthesis logs, timing reports, and resource utilization. Use commands like report_timing, report_area, or open the synthesized schematic view in the GUI.

Reference: Figure 14 – Viewing Synthesis Results for 64bit_alu in Cadence Genus

Step 7: Write Output Files

report_power > synth/power.rpt

```
Instance: /alu_64bit
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total     Row%
  ------------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     8.21061e-08  7.33105e-03  1.16368e-04  7.44750e-03   2.98%
        latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        logic     9.20074e-06  1.50975e-01  9.09670e-02  2.41952e-01  96.89%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  3.26177e-04  3.26177e-04   0.13%
          pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------------
    Subtotal     9.28285e-06  1.58307e-01  9.14096e-02  2.49725e-01 100.00%
  Percentage           0.00%       63.39%       36.60%      100.00% 100.00%
  ------------------------------------------------------------------------------
```

Figure 15: power Report for 64bit_alu

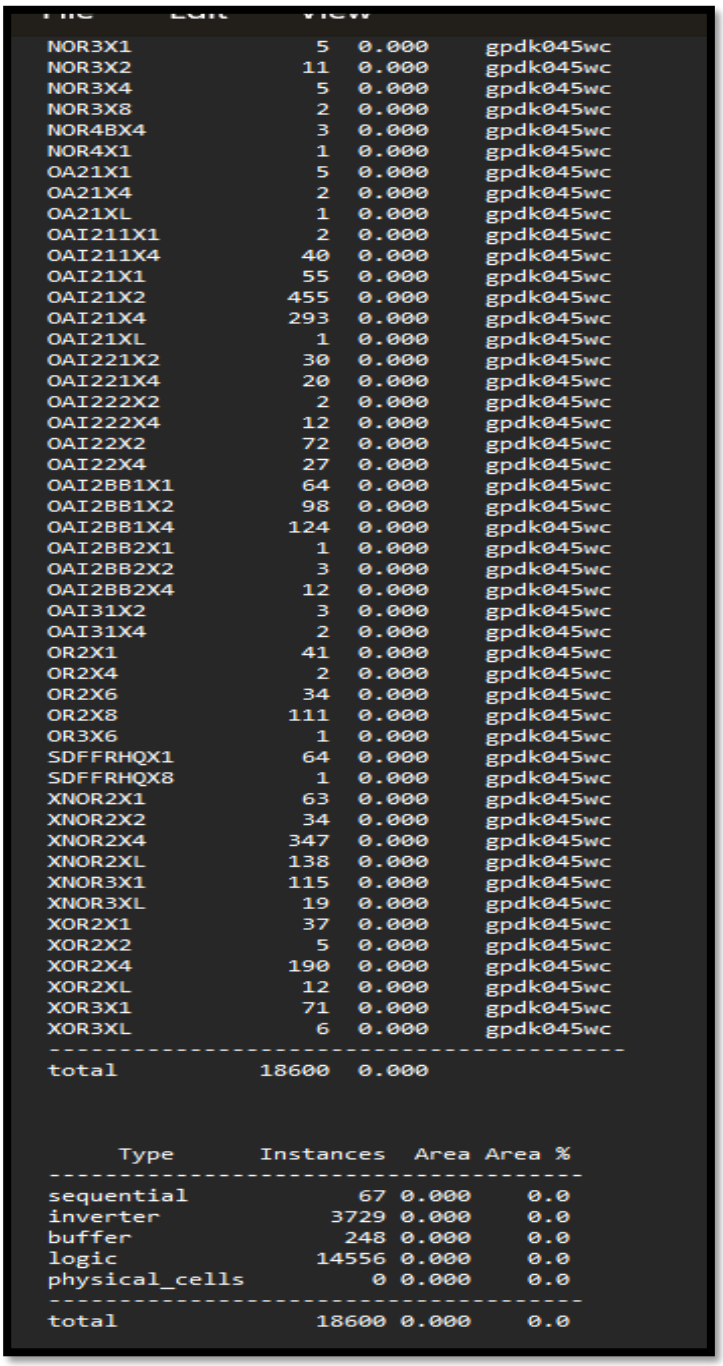
7.1 Generate Power Report for 64bit_alu

After synthesis, you can analyze the estimated power consumption of the design using Genus's built-in power reporting command. This helps in understanding dynamic, leakage, and total power at the RTL or gate level. Use the following command:

⇒ report_power > power_report.rpt

This will generate a power summary based on switching activity, load, and standard cell data.

Reference: Figure 15 – Power Report for 64bit_alu Generated in Cadence Genus



NOR3X1	5	0.000	gpdk045wc
NOR3X2	11	0.000	gpdk045wc
NOR3X4	5	0.000	gpdk045wc
NOR3X8	2	0.000	gpdk045wc
NOR4BX4	3	0.000	gpdk045wc
NOR4X1	1	0.000	gpdk045wc
OA21X1	5	0.000	gpdk045wc
OA21X4	2	0.000	gpdk045wc
OA21XL	1	0.000	gpdk045wc
OAI211X1	2	0.000	gpdk045wc
OAI211X4	40	0.000	gpdk045wc
OAI21X1	55	0.000	gpdk045wc
OAI21X2	455	0.000	gpdk045wc
OAI21X4	293	0.000	gpdk045wc
OAI21XL	1	0.000	gpdk045wc
OAI221X2	30	0.000	gpdk045wc
OAI221X4	20	0.000	gpdk045wc
OAI222X2	2	0.000	gpdk045wc
OAI222X4	12	0.000	gpdk045wc
OAI22X2	72	0.000	gpdk045wc
OAI22X4	27	0.000	gpdk045wc
OAI2BB1X1	64	0.000	gpdk045wc
OAI2BB1X2	98	0.000	gpdk045wc
OAI2BB1X4	124	0.000	gpdk045wc
OAI2BB2X1	1	0.000	gpdk045wc
OAI2BB2X2	3	0.000	gpdk045wc
OAI2BB2X4	12	0.000	gpdk045wc
OAI31X2	3	0.000	gpdk045wc
OAI31X4	2	0.000	gpdk045wc
OR2X1	41	0.000	gpdk045wc
OR2X4	2	0.000	gpdk045wc
OR2X6	34	0.000	gpdk045wc
OR2X8	111	0.000	gpdk045wc
OR3X6	1	0.000	gpdk045wc
SDFFRHQX1	64	0.000	gpdk045wc
SDFFRHQX8	1	0.000	gpdk045wc
XNOR2X1	63	0.000	gpdk045wc
XNOR2X2	34	0.000	gpdk045wc
XNOR2X4	347	0.000	gpdk045wc
XNOR2XL	138	0.000	gpdk045wc
XNOR3X1	115	0.000	gpdk045wc
XNOR3XL	19	0.000	gpdk045wc
XOR2X1	37	0.000	gpdk045wc
XOR2X2	5	0.000	gpdk045wc
XOR2X4	190	0.000	gpdk045wc
XOR2XL	12	0.000	gpdk045wc
XOR3X1	71	0.000	gpdk045wc
XOR3XL	6	0.000	gpdk045wc
total	18600	0.000	

Type	Instances	Area	Area %
sequential	67	0.000	0.0
inverter	3729	0.000	0.0
buffer	248	0.000	0.0
logic	14556	0.000	0.0
physical_cells	0	0.000	0.0
total	18600	0.000	0.0

Figure 16: Area Report for 64bit_alu

7.2 Generate Area Report for 64bit_alu

To analyze how much silicon area your synthesized design occupies, you can generate an area report in Cadence Genus. This report breaks down cell usage, total area, and instance counts. Use the following command:

⇒ report_area > area_report.rpt

This provides valuable insight into logic utilization and helps with early design optimization.

Reference: Figure 16 – Area Report for 64bit_alu Generated in Cadence Genus

```
DPOPT_INTERNAL_INSTi/csa_tree_eq_80_31_groupi_in_0[35]
final_adder_mux_next_out_32_15/A[35]
  g6062/A                                                    +0     642
  g6062/Y                    NOR2X6          2   5.1   16   +15     657 F
  fopt6610/A                                                 +0     657
  fopt6610/Y                 INVX4           2  10.1   22   +15     672 R
  g5804/A                                                    +0     672
  g5804/Y                    NAND2X8         3   7.8   24   +18     690 F
  g5580_1/A0                                                 +0     690
  g5580_1/Y                  OAI21X4         1   3.4   28   +22     712 R
  g5516/B                                                    +0     712
  g5516/Y                    NAND2X4         1   3.4   22   +18     730 F
  g5495/B                                                    +0     730
  g5495/Y                    NAND2X4         2   4.1   18   +13     742 R
  fopt8/A                                                    +0     742
  fopt8/Y                    INVX4           1   3.5   11   +10     752 F
  g5456/A                                                    +0     752
  g5456/Y                    NOR2X4          1   5.0   28   +18     770 R
  g5453/B                                                    +0     770
  g5453/Y                    NOR2X6          4  12.7   26   +18     787 F
  fopt6488/A                                                 +0     787
  fopt6488/Y                 INVX4           4   6.8   18   +14     802 R
  g5421/C                                                    +0     802
  g5421/Y                    NAND3X2         2   1.8   36   +22     824 F
  g5420/A                                                    +0     824
  g5420/Y                    INVX1           1   1.0   14   +15     839 R
  g5407/A1                                                   +0     839
  g5407/Y                    AOI21X1         1   1.0   28   +21     860 F
  g5209/C                                                    +0     860
  g5209/Y                    NAND3X1         1   0.9   22   +15     875 R
final_adder_mux_next_out_32_15/Z[63]
out_reg[63]/SI               SDFFRHQX1                      +0     875
out_reg[63]/CK               setup                     0    +51     925 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock clk)                  capture                              250 R
-------------------------------------------------------------------------
Timing slack : +75ps
Start-point  : B[0]
End-point    : out_reg[63]/SI
```

Figure 17: Timing Report for 64bit_alu

7.3 Generate Timing Report for 64bit_alu

After synthesis, perform a timing analysis to verify that your design meets the defined constraints (from the SDC file). This report includes data paths, slack, setup/hold violations, and critical path details. Use the following command in Genus:

⇒ report_timing > timing_report.rpt

This step is essential for ensuring the design functions correctly at the target frequency.

Reference: Figure 17 – Timing Report for 64bit_alu Generated in Cadence Genus

Step 8: Exit Genus

Exit

Quick Summary of Genus Commands

Task	Command
Launch Genus	genus
Read RTL	read_hdl rtl/alu_64bit.v
Read Constraints	read_sdc constraints.sdc
Read Library	set_db / .library lib/std_cell.db
Elaborate	elaborate
Synthesize	syn_generic, syn_map, syn_opt
Write Netlist	write_hdl > synthesized_netlist.v
Write Reports	report_timing, report_area, report_power
Exit	exit

Cadence Innovus™ – Physical Design Tool

Cadence Innovus Implementation System is a state-of-the-art **physical design (backend)** tool used for advanced digital IC implementation. It transforms a synthesized netlist into a manufacturable layout (GDSII), while meeting all design constraints for performance, area, power, and signal integrity

Key Features

Feature	Description
Placement & Floorplanning	Places standard cells, macros, I/O pads, and defines power grid architecture.
Clock Tree Synthesis (CTS)	Balances clock signal delays and minimizes skew across the design.
Routing	Performs signal routing considering DRC, congestion, and timing.
Static Timing Analysis (STA)	Checks setup/hold timing, slack, and path delays.
Power Optimization	Supports dynamic and leakage power reduction techniques.
Multi-Corner Multi-Mode (MCMM)	Verifies design across multiple PVT corners and functional modes.
Advanced Node Support	Optimized for FinFET and <10nm technology nodes.

Typical Flow Using Innovus

1. **Import Synthesized Netlist**
    - From Genus, along with SDC and constraints.

2. **Floorplanning**
    - Define core area, place macros, create power straps.
3. **Placement**
    - Automatically places standard cells considering congestion and timing.
4. **Clock Tree Synthesis (CTS)**
    - Builds a balanced and low-skew clock tree.
5. **Routing**
    - Connects all nets with metal layers while avoiding DRCs.
6. **Optimization**
    - Fixes timing violations, reduces power, and optimizes congestion.
7. **Signoff Checks**
    - Timing, IR drop, antenna, and DRC/LVS clean.
8. **GDSII Generation**
    - Final output sent to foundry for fabrication.

Why Use Innovus in VLSI?

- **High performance & capacity** for large SoCs
- **Tight integration** with Genus (synthesis) and Tempus (signoff STA)
- **Industry standard** used by top semiconductor companies
- **Supports ECOs**, advanced clocking, hierarchical designs

Core Tools Within Innovus

Component	Purpose
encounter	Command-line interface (CLI)
GUI Window	Visual control of placement/routing
report_timing	View slack and critical paths
report_power	Analyze dynamic and leakage power
verify_drc	Design rule checks (DRC)

📁 **Input Files Required**

- **Netlist** (.v from Genus)
- **SDC constraints**
- **Technology LEF** (layer info)
- **Library files** (.lib, .lef, .tf)
- **DEF** (optional for placement info)

Step 1: Launch Innovus

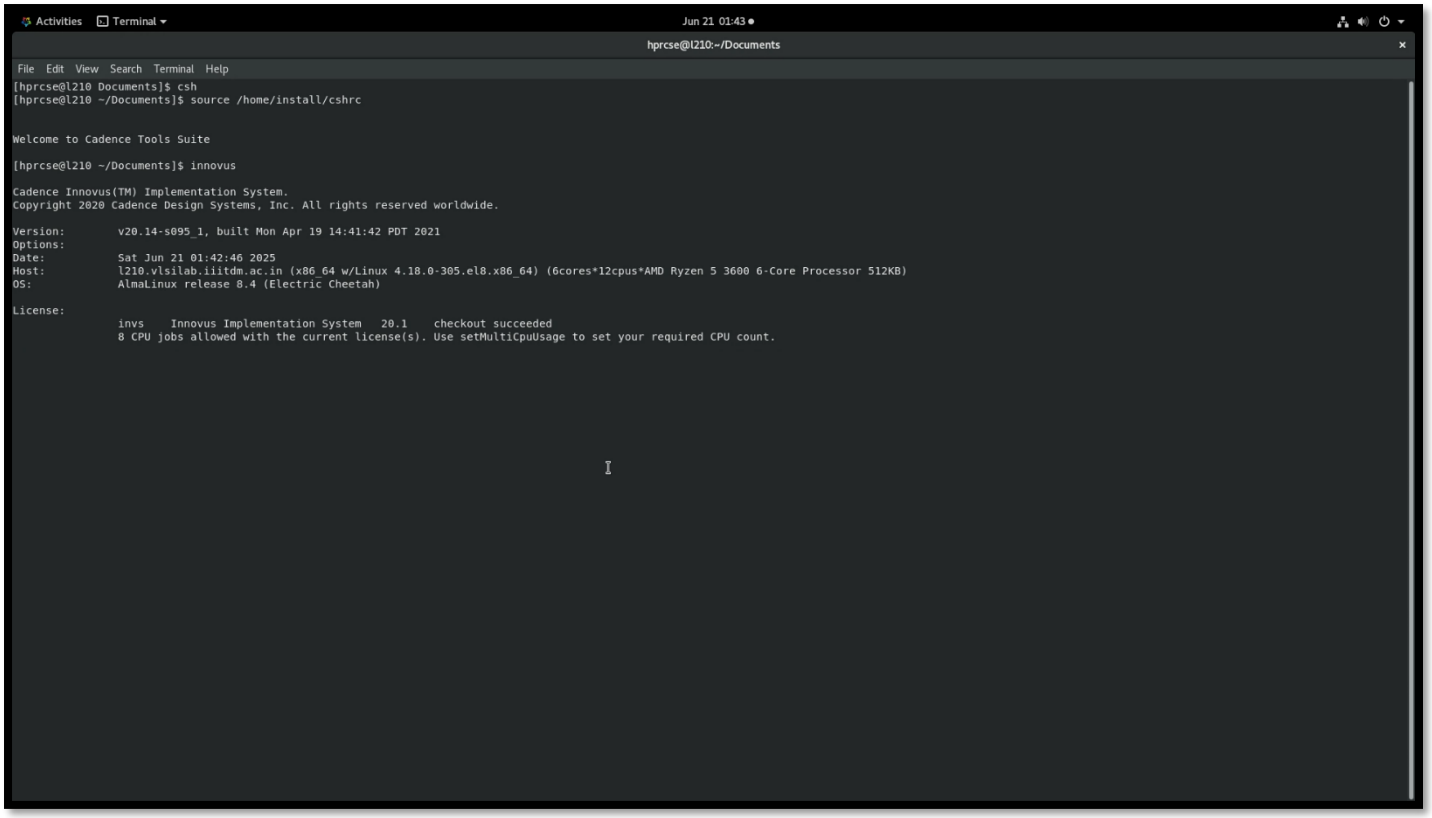


Figure 18: Launching Cadence Innovus Using innovus & Command in Terminal

1.Launch Cadence Innovus Tool

Once synthesis is complete in Genus and the netlist is ready, proceed to physical design by launching Cadence Innovus. This can be done directly from the terminal using:

⇒ innovus &

This command opens the Innovus GUI, where floorplanning, placement, CTS, routing, and signoff steps are performed.

Reference: Figure 18 – Launching Cadence Innovus Using innovus & Command in Terminal

Step 2: Setup Project Environment

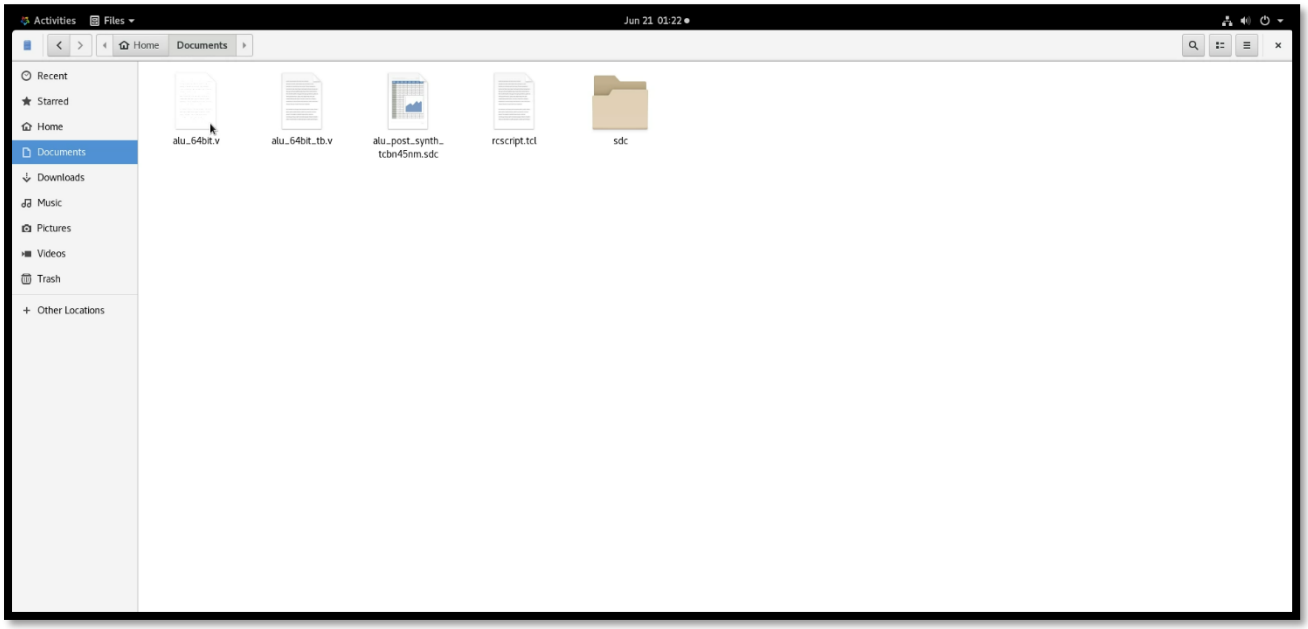


Figure 19: Essential input file for Physical Design

2.**Essential Input Files for Physical Design**

Before starting physical design in Innovus, provide the following key input files:

⇒ Synthesized Netlist (.v)

⇒ Timing Constraints (.sdc)

⇒ Technology File (.tf)

⇒ Library Exchange Format Files (.lef)

⇒ Floorplan/IO Constraints (optional)

These files are loaded during the init_design phase to initialize the physical design environment.

Reference: Figure 19 – Essential Input Files for Physical Design

Step 3: Sample encounter.tcl Script

```
============ Setup ============
set_design_name alu_64bit
set init_design_netlist alu_64bit.v
set init_design_sdc alu_64bit.sdc
set init_lef_file {stdcells.lef io.lef}
set init_lib_file {lib_typ.lib}
set init_top_cell alu_64bit
set init_mmmc_file mmmc.view
============ Design Flow ============
init_design
floorPlan -site core -r 1.0 0.6 10 10 10 10
addRing -type core -width 5 -spacing 2 -layer top
addStripe -set_to_set_distance 20 -width 2 -layer M2 -number_of_sets 2

placeDesign
clockDesign -idealClock
clockDesign -cts
routeDesign
optDesign -postRoute
verifyGeometry
verifyConnectivity
verify_drc
```

verify_lvs

report_timing > final_timing.rpt

report_power > final_power.rpt

saveDesign alu64_final.enc

streamOut alu64bit.gds2 -mapFile streamOut.map -libName my_lib

exit

Step 4: Create MMMC File (mmmc.view)

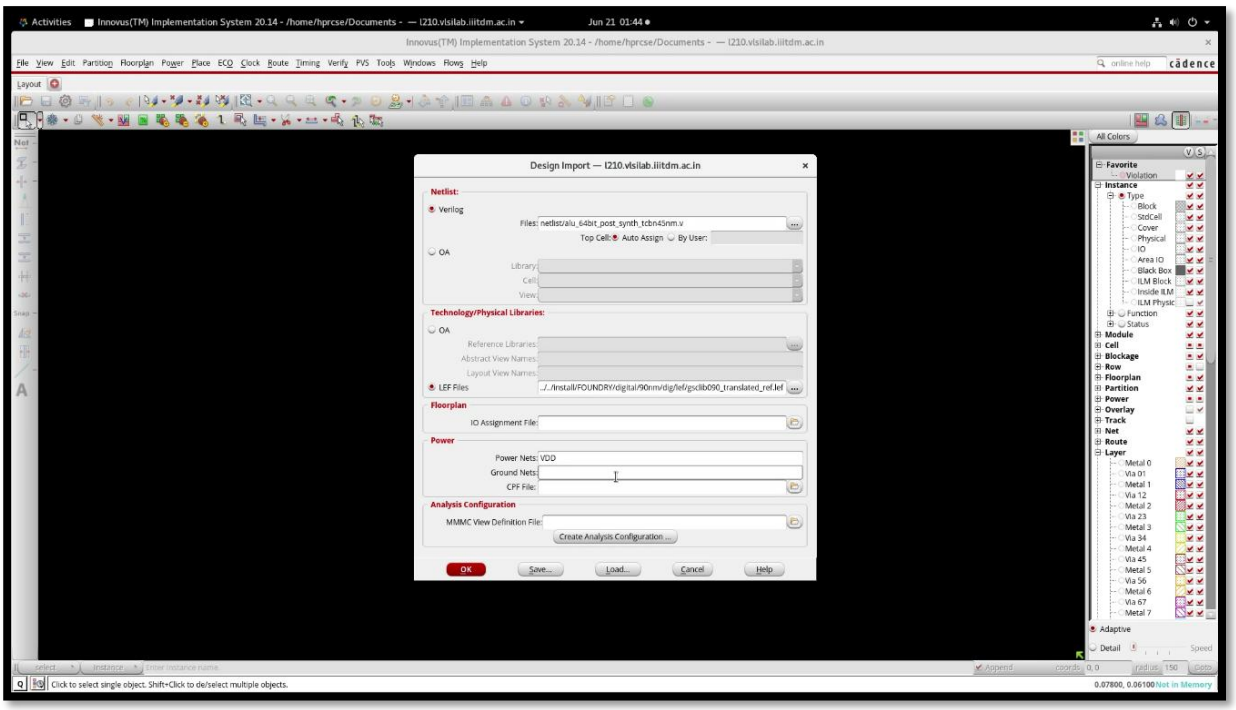


Figure 20: MMMC window

4. Set Up MMMC (Multi-Mode Multi-Corner) Constraints

MMMC setup is used to define multiple operating modes (e.g., functional, test) and corners (e.g., slow, fast) for accurate timing analysis across all scenarios. This is done using the MMMC GUI or TCL commands like create_corner, create_mode, and create_scenario.

Reference: Figure 20 – MMMC Window in Cadence Innovus

create_library_set -name typical -timing lib_typ.lib

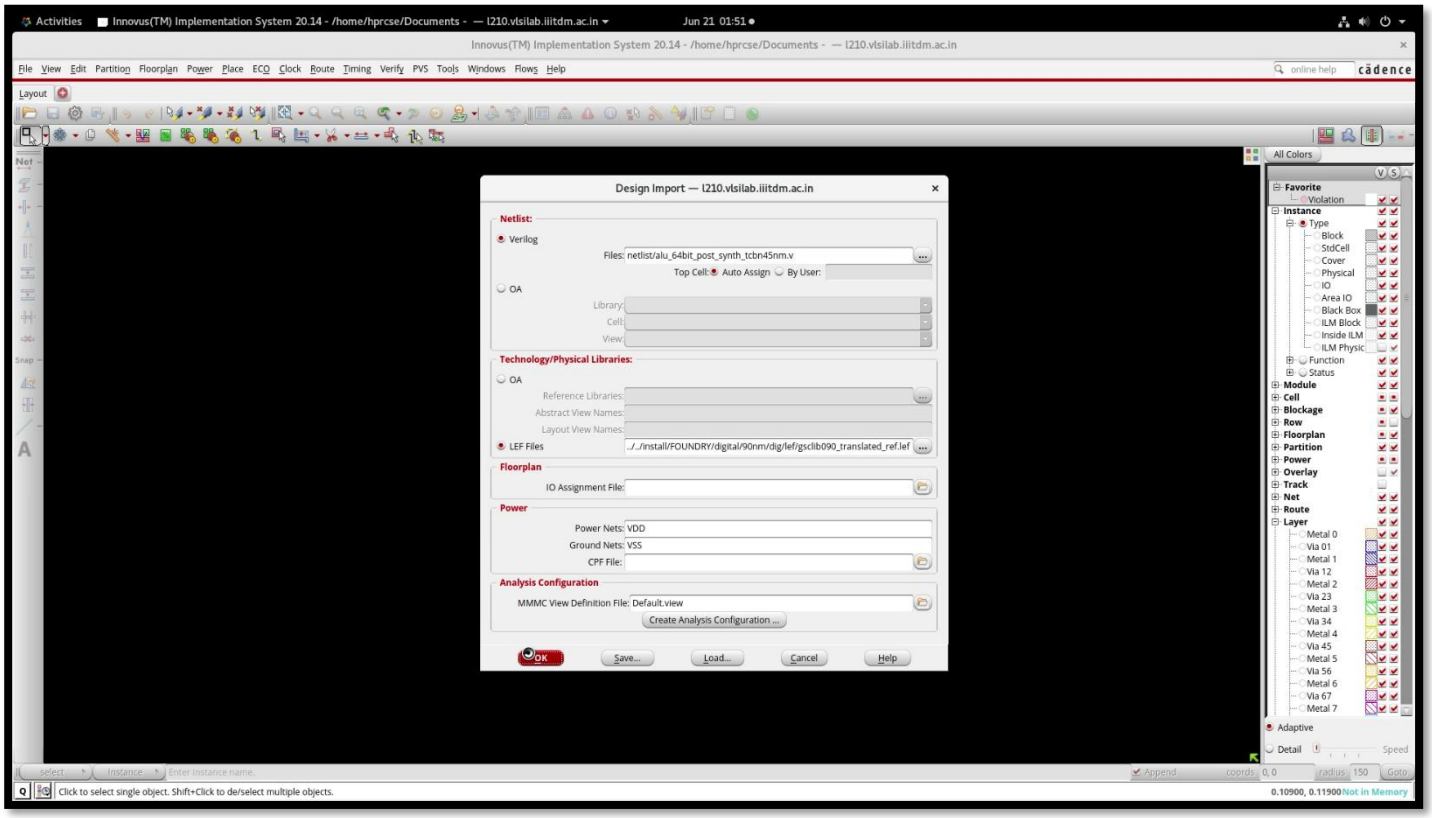


Figure 21: Final Routed Layout with MMMC Constraints Applied in Cadence Innovus

4.1Final Routed Layout with MMMC Constraints

After placement, clock tree synthesis, and routing, the final layout is generated. With MMMC constraints applied, Innovus ensures timing closure across all defined scenarios. The layout includes standard cells, macros, metal routing, and power stripes.

Reference: Figure 21 – Final Routed Layout with MMMC Constraints Applied in Cadence Innovus

create_delay_corner -name delay_typical -library_set typical -rc_corner rc_corner_typical

create_analysis_view -name func_view -delay_corner delay_typical -setup

set_analysis_view -setup func_view

Step 5: Floorplanning

Floorplanning is the first step in the physical implementation flow where the physical layout structure of the chip is defined. It sets the foundation for placement, power planning, routing, and overall chip performance.Think of it as architecting the floor plan of a building before construction.

This step sets the physical outline and guides the later placement/routing.

Steps:

floorPlan -site core -r 1.0 0.6 10 10 10 10

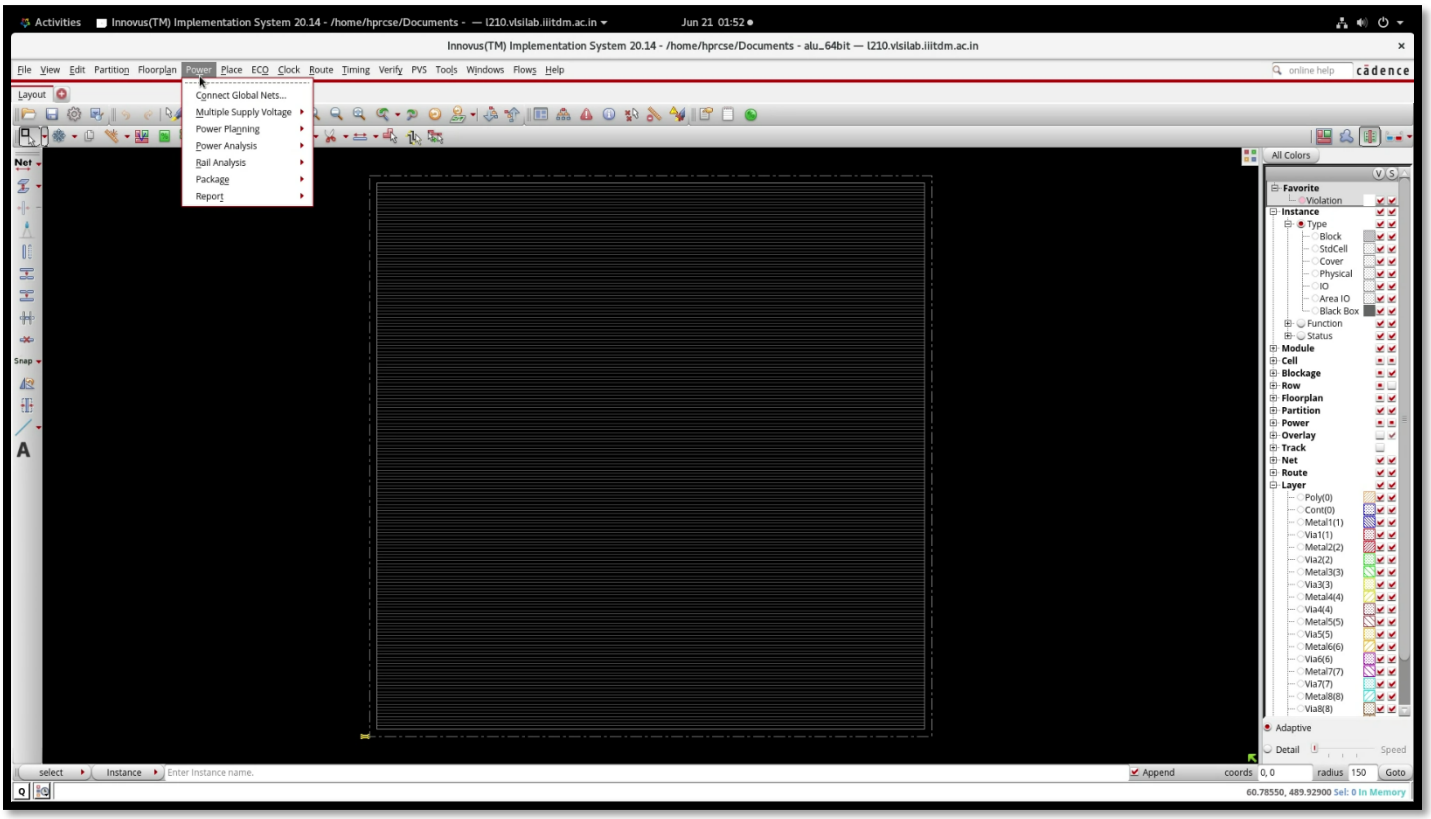


Figure 22: Floorplanning

5. **Floorplanning in Cadence Innovus**

Floorplanning defines the chip's core area, I/O pad placement, macro placement, and utilization. It's the foundation of physical design and directly impacts timing, congestion, and routing quality. You can perform it via GUI or using the floorPlan and placeIO commands.

Reference: Figure 22 – Floorplanning in Cadence Innovus

Task	Description
-site core	Uses core standard cell site
-r ratio	Aspect ratio of core (width/height)
10 10 10 10	Left, bottom, right, top core margins

Step 6: Power Planning

Power planning is the process of designing and implementing a robust power delivery network (PDN) to ensure all parts of the chip receive stable and noise-free power (VDD) and ground (VSS). This is critical for timing, reliability, and manufacturability.

Create power rings and power stripes to supply VDD/VSS to all cells.

Feature	Role in Power Plan
Rings	Global power delivery from IO
Stripes	Local power distribution inside core
Well Taps	Avoid latch-up and connect to substrate
Endcaps	Fill and protect row edges
connectPGNet	Links cells to VDD/VSS

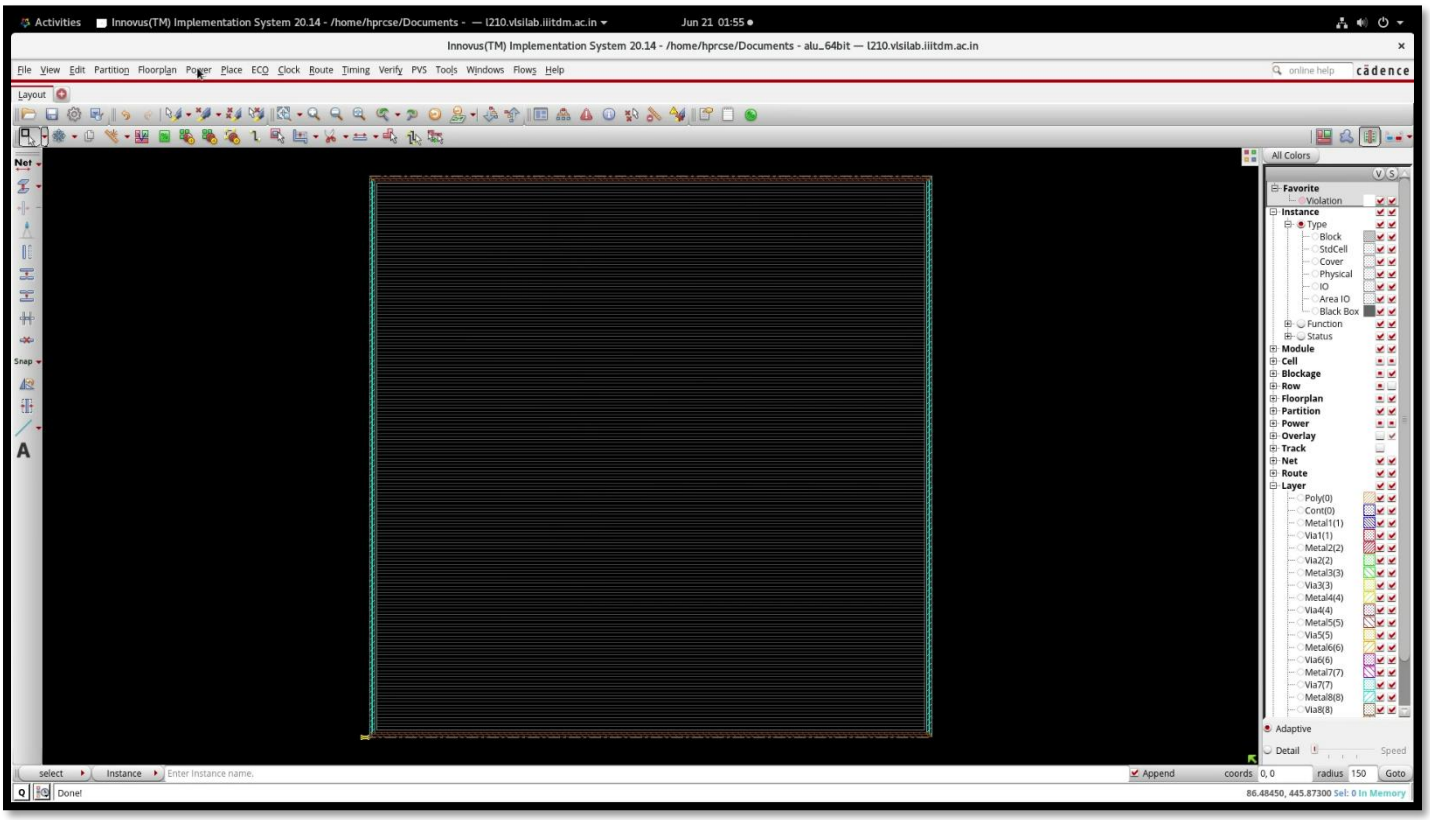


Figure 23: Powerplanning

Steps:

addRing -type core -nets {VDD VSS} -layer top -width 5 -spacing 2

addStripe -nets {VDD VSS} -layer M2 -width 2 -spacing 2 -pitch 20 -set_to_set_distance 40

connectPGNet

6. **Power Planning in Cadence Innovus**
Power planning involves defining power rings, stripes, and connections to ensure robust power delivery across the chip. This step includes commands like addRing, addStripe, and connectPower to distribute VDD and VSS uniformly, preventing IR drop and electromigration issues.
Reference: Figure 23 – Power Planning in Cadence Innovus

Task	Description
addRing	Adds metal rings around the core
addStripe	Vertical/horizontal metal lines to spread power
connectPGNet	Connects power nets (VDD/VSS) to standard cell rails

Step 7: Placement

Places standard cells and macros considering timing, congestion, and power.Placement is the stage where standard cells (logic gates, combinational and sequential elements) are physically arranged within the defined floorplan area without overlaps, while optimizing for **timing, congestion, power, and routability**.

Steps:

placeDesign

checkPlace

optDesign -preCTS

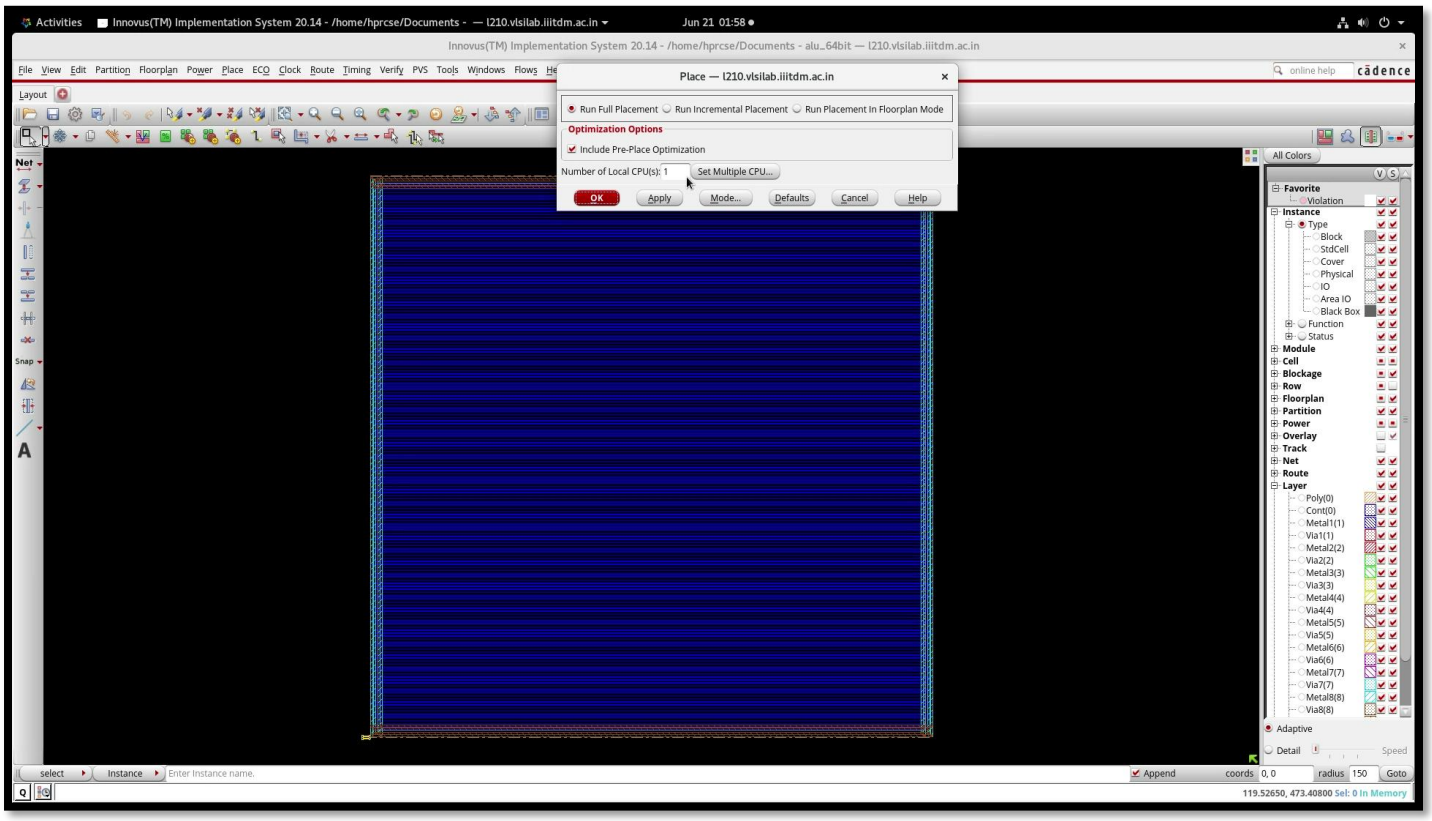


Figure 24: Placement

7.Standard Cell Placement in Cadence Innovus

Placement arranges synthesized standard cells within the defined floorplan while minimizing wirelength, congestion, and timing violations. This step is performed using the place_opt command, which includes pre-placement optimization for better timing and routability.

Reference: Figure 24 – Placement in Cadence Innovus

Task	Description
placeDesign	Places cells optimally
checkPlace	Verifies DRC, overlap, congestion
optDesign -preCTS	Optimizes timing and logic pre-CTS

Step 8:Clock Tree Synthesis (CTS)

Clock Tree Synthesis (CTS) is the process of distributing the clock signal from clock sources (e.g., PLL or clock port) to all sequential elements (flip-flops, latches) in the design while minimizing clock skew and insertion delay.Builds a robust and balanced clock network to minimize skew and latency.

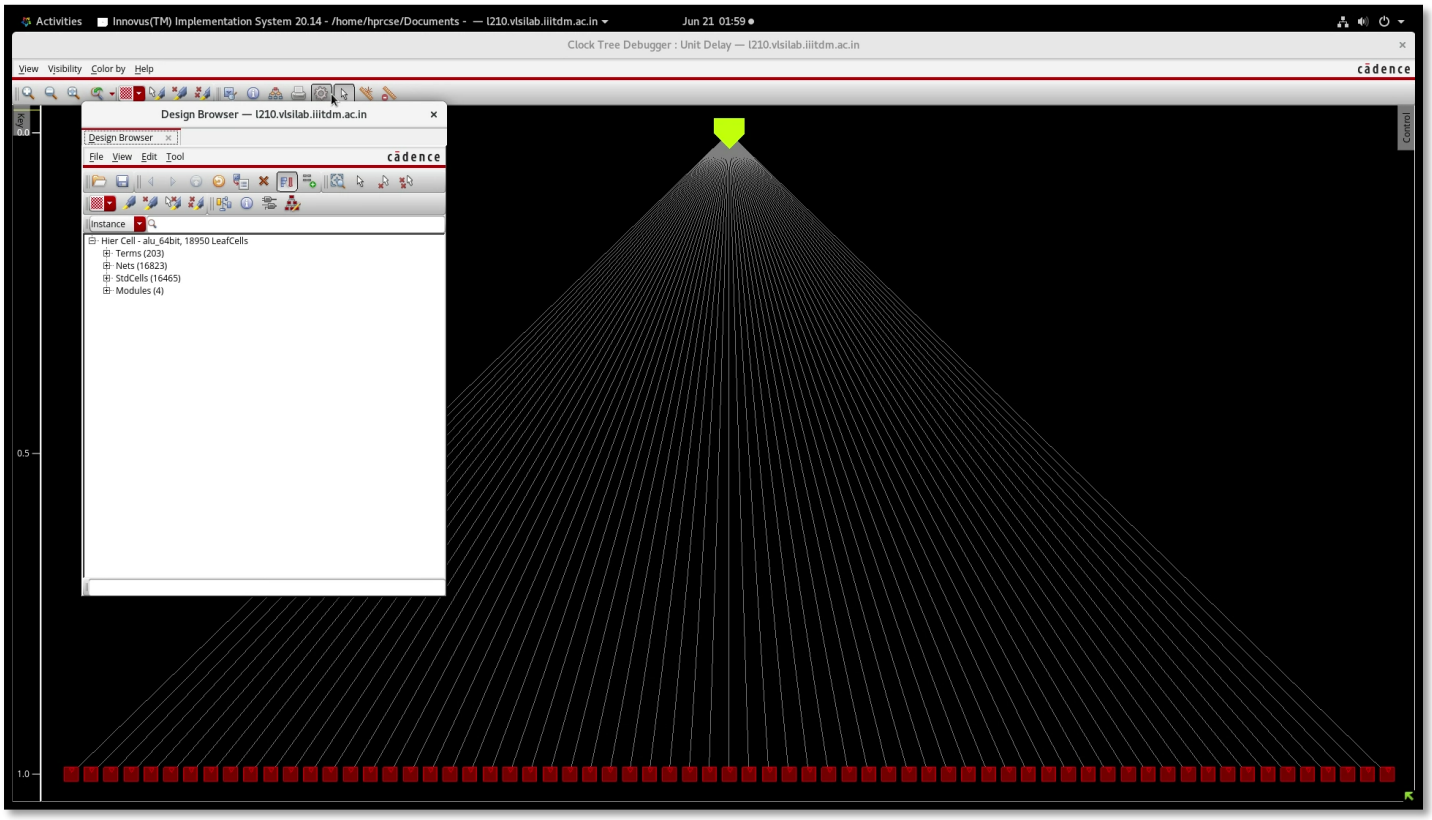


Figure 25: Clock Tree Synthesis(CTS)

8.Clock Tree Synthesis (CTS) in Cadence Innovus

CTS builds the clock distribution network to minimize skew and latency across the design. It balances the clock paths from source to all sequential elements using the clock_opt command. This step ensures synchronized data capture across all flip-flops.

Reference: Figure 25 – Clock Tree Synthesis (CTS) in Cadence Innovus

Steps:

setCTSMode -engine cts

clockDesign

timeDesign -postCTS -hold

check_clock_tree

Task	Description
clockDesign	Builds clock tree (buffers, clock paths)
timeDesign -postCTS	Runs STA after CTS to check slack
check_clock_tree	Verifies skew, fanout, and insertion delay

Step 9: Routing

Routing is the process of physically connecting the pins of standard cells, macros, and I/O ports using metal layers. After placement and Clock Tree Synthesis (CTS), routing defines the actual interconnections that allow the circuit to function.

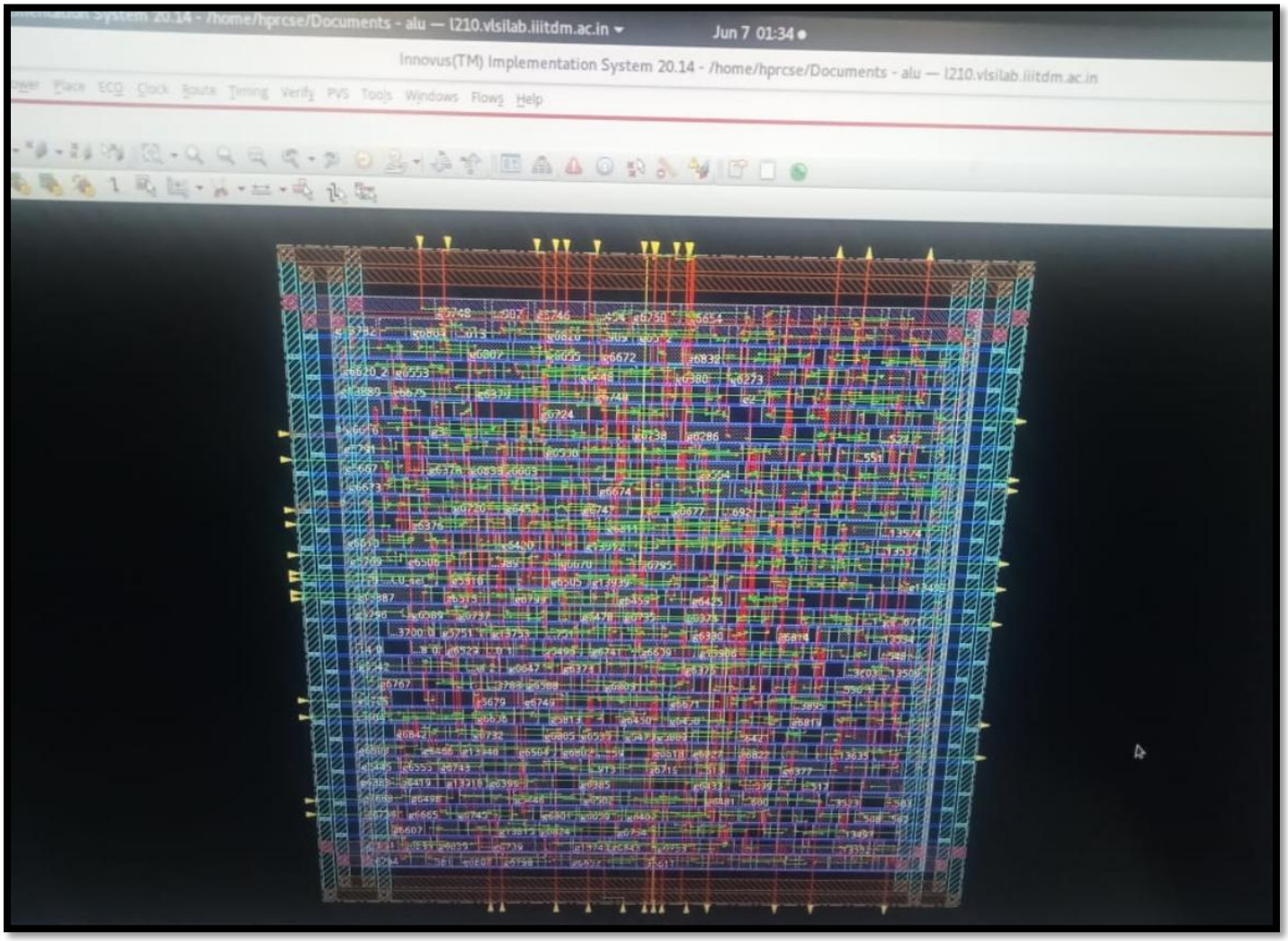


Figure 26: Routing

9.Routing in Cadence Innovus

Routing connects all placed standard cells and macros based on the netlist, completing the physical implementation. It consists of global routing followed by detailed routing using the route_opt command, ensuring DRC-compliant, timing-optimized interconnects.

Reference: Figure 26 – Routing in Cadence Innovus

Objectives of Routing

Goal	Description
Establish signal paths	**Connect all nets (inputs, outputs, clock, power)**
Meet design rules	**Ensure DRC-clean metal traces**
Preserve timing	**Avoid delay or crosstalk issues**
Minimize congestion	**Balance usage of routing tracks**
Ensure manufacturability	**Proper via usage and metal spacing**

Step 10: DRC & LVS Checks

Verifies physical and logical correctness of the layout.

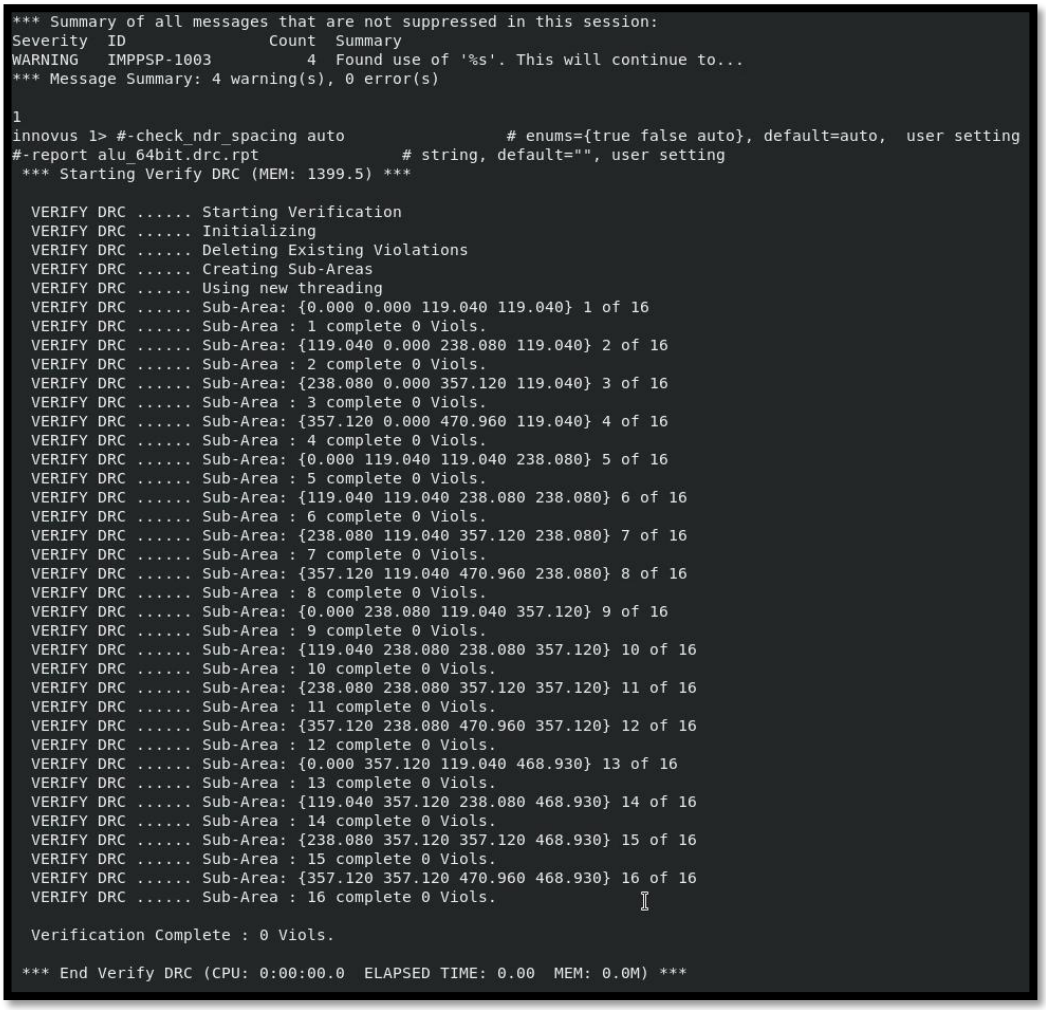


Figure 20: DRC/LVS Checks

Steps:

verifyGeometry -report geom.rpt

verifyConnectivity -type all -report conn.rpt

verify_drc -report drc.rpt

10.Perform DRC and LVS Checks

After routing, Design Rule Check (DRC) and Layout Versus Schematic (LVS) are run to ensure that the layout is manufacturable and matches the schematic. DRC verifies spacing, width, and design compliance with foundry rules, while LVS ensures logical correctness by comparing the layout netlist with the schematic netlist.

Reference: Figure 20 – DRC/LVS Checks in Cadence Innovus

Task	Description
verifyGeometry	Checks shape overlap, vias, antenna rules
verifyConnectivity	Ensures all power, ground, and signal nets ok
verify_drc	Checks for DRC violations
verify_lvs	Matches netlist with layout

Step 11 : **Signoff & GDSII Export**

This is the final stage of the digital IC design flow. It ensures that the design is fully verified, clean of violations, and is ready for fabrication by exporting the final layout in GDSII format (Graphic Data System II).

The Signoff & GDSII Export phase is the tapeout-ready milestone of your project. It certifies that your design is:

- Functionally correct
- Timing clean
- Physically DRC/LVS clean
- Ready for fabrication via the GDSII file

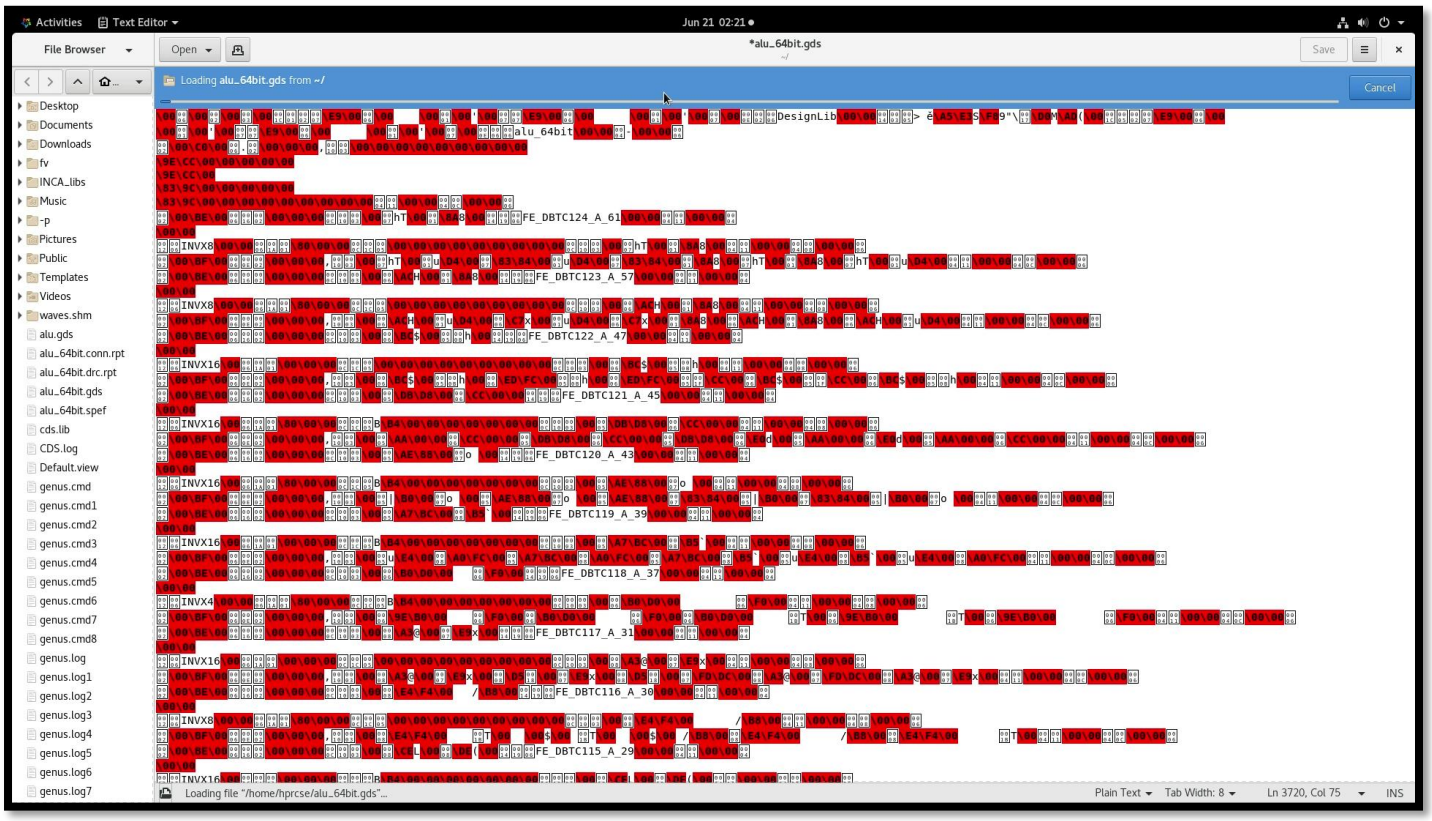


Figure 20: GDSII File Generation from Final Layout in Cadence Innovus

20.GDSII File Generation from Final Layout

Once DRC and LVS are clean, the final layout is exported in GDSII format using the streamOut command. This GDSII file is the standard output used for tape-out and manufacturing. It includes all geometric layers, metal routing, and mask-level data.

⇒ Example: streamOut final.gds -map layer.map -lib design_lib -cells top -merge

Reference: Figure 20 – GDSII File Generation from Final Layout in Cadence Innovus

Step 12: IO.SDC File

io.sdc

```
##################################################################
# CADENCE
# sdc.io – SDC File for 64-bit ALU
# Description: I/O and clock timing constraints for synthesis
##################################################################
# CLOCK DEFINITION
create_clock -name clk -period 10.0 [get_ports clk]
# INPUT DELAY CONSTRAINTS (Assumed: input arrives 2ns after clk edge)
set_input_delay 2.0 -clock clk [get_ports A]
set_input_delay 2.0 -clock clk [get_ports B]
set_input_delay 2.0 -clock clk [get_ports sel]
# OUTPUT DELAY CONSTRAINTS (Assumed: output required 2ns before clk edge)
set_output_delay 2.0 -clock clk [get_ports out]
set_output_delay 2.0 -clock clk [get_ports carryout]
set_output_delay 2.0 -clock clk [get_ports Zero]
set_output_delay 2.0 -clock clk [get_ports Overflow]
##################################################################
```

Summary of Flow

Floorplan	**floorPlan, addRing**
Power Plan	addStripe, connectPGNet
Placement	placeDesign, optDesign
Clock Tree	clockDesign, check_clock_tree
DRC/LVS	verify_drc, verify_lvs
GDSII Export	streamOut, saveDesign

5 CONCLUSION AND FUTURE SCOPE

Conclusion

The design and implementation of the 64-bit Arithmetic Logic Unit (ALU) using the **Cadence digital IC design flow** have demonstrated the complete semi-custom ASIC design methodology — from **RTL coding to GDSII generation**. The ALU was modeled in Verilog, verified using Incisive/Xcelium simulators, synthesized using Genus, and physically implemented using Innovus.

Key stages such as **floorplanning, power planning, placement, clock tree synthesis (CTS), routing**, and **signoff checks** were carried out systematically. Timing closure was successfully achieved, ensuring the ALU meets the required setup and hold constraints. The final **GDSII layout file** was generated, making the design ready for fabrication. This project showcases a robust understanding of VLSI physical design principles, tool flows, and design constraints involved in creating a high-performance arithmetic unit.

Future Scope

While the current 64-bit ALU performs essential arithmetic and logic operations efficiently, the design can be enhanced in several ways to meet advanced application requirements:

1. Pipelining & Parallelism

- Introduce pipelining stages to increase the throughput of the ALU.
- Implement parallel execution units to support superscalar architecture.

2. Low Power Techniques

- Apply power gating, clock gating, and multi-Vt cell usage to reduce dynamic and leakage power in portable and battery-powered SoCs.

3. Fault Tolerance

- Integrate ECC (Error Correcting Codes), parity checks, or triple modular redundancy (TMR) for high-reliability and aerospace-grade applications.

4. Integration in a Processor Core

- Embed the ALU into a larger processor datapath (RISC-V or custom ISA), supporting complete CPU functionality.