ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of a task would be incomplete without the mention of the people who made it possible and without their constant guidance and encouragement, success would not have been possible.

I am grateful to the institute **Acharya Institute of Technology** and management with its ideas and inspiration for having provided us with the good infrastructure, laboratory, facilities and inspiring staff which has made this internship complete successfully.

I would like to express my sincere gratitude to **Dr. C K Marigowda, Principal, AIT** for all the facilities that he has extended throughout my work.

I heartily thank and express my sincere gratitude to **Dr. Rajeswari, Professor and HOD, Dept. of ECE, AIT** for her valuable support and a constant source of enthusiastic inspiration to steer us forward.

I would like to express my sincere gratitude to the Internal Guide **Dr. Nagapushpa K.P Assistant Professor Department of ECE, AIT** for her invaluable guidance and support.

I would like to express my sincere gratitude to the Internship Coordinators **Mrs. Sumangala S J, Assistant Professor, Dept. of ECE, AIT** for her valuable guidance and support.

Finally, I would like to express my sincere gratitude to my parents, all teaching and non-teaching faculty members and friends for their moral support, encouragement and help throughout the completion of the internship.

Hemanth S

1AY21EC041

ABSTRACT

The internship provided comprehensive exposure to both front-end and back-end aspects of VLSI design. In the front-end phase, participants gained hands-on experience with Verilog HDL for RTL design, simulation, synthesis, and verification. Key concepts such as finite state machines, testbenches, and combinational/sequential circuit design were practiced using tools like Yosys and Xilinx ISE.

The back-end training involved physical design workflows including floorplanning, placement, clock tree synthesis, routing, and layout verification. Tools such as KLayout, and OpenROAD were used to perform tasks like GDSII generation. A mini-project on a 4 Bit Adder-Subtractor design helped integrate both front-end and back-end knowledge. Overall, the internship enhanced the intern's understanding of the complete VLSI flow and prepared them for industry roles in semiconductor design and verification.

TABLE OF CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

1.1 PURPOSE AND OBJECTIVES OF THE INTERNSHIP

Internships play a vital role in shaping a student's academic and professional path by offering real-world experiences beyond classroom education. The primary purpose of this internship was to gain hands-on exposure and practical understanding of VLSI (Very Large-Scale Integration) design, which forms the backbone of modern digital systems and electronics. This 460-hour comprehensive training, jointly conducted by **Rooman Technology Pvt Ltd** in collaboration with **Visvesvaraya Technological University (VTU)**, was tailored to build both foundational and advanced skills in VLSI front-end and back-end design flows.

The key objective was to bridge the knowledge gap between theoretical concepts taught in academic settings and practical implementations used in the semiconductor industry. The internship aimed to train participants on the complete chip design flow, beginning from the RTL (Register Transfer Level) modeling and behavioral design to synthesis, floorplanning, placement, routing, and final GDSII generation. Through industry-standard tools and real-time projects, we were encouraged to design digital components, simulate circuits, and carry out backend layout design with verification procedures like DRC (Design Rule Checking) and LVS (Layout vs. Schematic).

In addition to technical expertise, the internship had several other goals:

- To develop problem-solving abilities through design challenges.
- To instill good programming practices using Verilog HDL.
- To enhance understanding of ASIC and FPGA architectures.
- To offer exposure to EDA tools used in commercial IC development.
- To cultivate soft skills like teamwork, communication, time management, and documentation.

In essence, this internship was structured not just as a training program, but as an immersive learning experience designed to prepare us for the competitive and rapidly evolving VLSI domain.

1.2 OVERVIEW OF THE INDUSTRY AND THE SPECIFIC ORGANIZATION

The semiconductor industry is one of the fastest-growing and most critical sectors in the world, enabling advancements in computing, telecommunications, automotive, healthcare, and consumer electronics. At the core of this industry lies VLSI design, a process that involves integrating millions (and now billions) of transistors into a single chip. These integrated circuits (ICs) are used in almost every electronic device today,

from smartphones and laptops to satellites and medical equipment.

As devices become more compact, efficient, and powerful, the complexity of ICs has also increased exponentially. This necessitates a highly skilled workforce proficient in digital design methodologies, fabrication processes, and verification techniques. VLSI engineers are required to ensure high performance, low power consumption, and minimal area usage in ICs while maintaining design integrity and manufacturability.

In this context, **Rooman Technology Pvt Ltd** plays a pivotal role in nurturing future VLSI professionals. It is a recognized training and consulting organization specializing in advanced semiconductor and embedded system domains. Known for its practical, application-oriented training modules, Roman Technology has developed a reputation for industry-aligned curriculum, live projects, and mentorship by domain experts.

The organization offers a variety of training programs, including VLSI design, physical design, system-on-chip design, and embedded software development. Their collaboration with VTU ensures that engineering students from affiliated colleges receive quality exposure to industry-level tools, technologies, and workflows. Their training approach focuses not only on theoretical clarity but also on delivering robust, hands-on experiences that mirror real semiconductor industry practices.

During the internship, we used professional tools such as:

- **Yosys**: An open-source synthesis tool used for RTL synthesis.
- **OpenROAD**: For complete digital backend automation.
- **KLayout**: For physical verification and design rule checks.

The exposure to these tools provided a comprehensive view of how a chip is designed, verified, and physically realized—offering insights that go far beyond classroom simulations.

1.3 HOW THIS INTERNSHIP ALIGNS WITH OUR ACADEMIC AND CAREER GOALS

As an engineering student specializing in **Electronics and Communication Engineering**, my academic curriculum included courses like Digital Logic Design, Microelectronics, VLSI Design, and HDL programming. While these courses laid the theoretical foundation, the internship allowed me to apply these principles in real-world scenarios, thus enhancing my practical understanding.

This internship was not only aligned with but also integral to my academic goals. It reinforced subjects like:

- **Digital Design**: Designing multiplexers, encoders, decoders, adders, counters, and FSMs using Verilog.
- **HDL Simulation**: Writing synthesizable code and testbenches for simulation.
- **Synthesis and Optimization**: Understanding how high-level behavioral code is transformed into logic gates and flip-flops.
- **Backend Layout Design**: Learning placement, routing, parasitic extraction, and verification.

Furthermore, the training in tools like Vivado and Magic VLSI gave me a head-start in using the same EDA software that industry professionals use. The exposure to ASIC flow through tools like Yosys and OpenROAD helped me understand the nuances of timing constraints, clock tree synthesis, and the challenges involved in chip-level design and physical realization.

From a **career perspective**, this internship has helped me solidify my interest in the VLSI field. I now have a clearer picture of the roles and responsibilities of a VLSI design engineer. The hands-on experience has made me confident in my ability to pursue opportunities in areas such as:

- **ASIC Front-End Design**: RTL coding, synthesis, and simulation.
- **Physical Design**: Floorplanning, placement, routing, and GDSII generation.
- **FPGA Design**: Prototyping and real-time hardware testing.
- **Design Verification**: Using testbenches, coverage models, and formal techniques.

Moreover, the exposure has given me a competitive edge while applying for entry-level roles and internships at core semiconductor companies. It has also prepared me to attempt higher-level certifications and contribute to open-source VLSI projects. With the global semiconductor market experiencing continuous growth and India emerging as a potential design hub, this internship has positioned me at the forefront of a promising technical career path.

Lastly, the mentorship and professional interactions during the training helped improve my communication skills, teamwork, and presentation abilities. These are essential attributes not only for workplace success but also for academic excellence in group projects, seminars, and research presentations.

CHAPTER 2

ORGANIZATION PROFILE

2.1 OVERVIEW

Established in 1999, **Manish Kumar (CEO)**. Rooman Technologies is a premier Indian enterprise dedicated to skill development and IT training. With over two decades of experience, the company has trained more than 1.2 million students across 198 cities and partnered with 24 academic institutions. Rooman integrates cutting-edge technologies—such as Artificial Intelligence (AI), Data Analysis, and Machine Learning—into its curriculum and delivery systems to provide industry-relevant education.

2.2 MISSION AND VISION

- **Vision:** Integrate global innovation, technology, and skill to empower individuals, society, and businesses.
- **Mission:** Impart quality training for the empowerment of youth to make India the skill capital of the world; integrate global technologies to introduce innovative products and solutions; increase global presence through associations, collaborations, and partnerships; and establish a global education campus housing top university of the world.

2.3 SERVICES AND OFFERINGS

Rooman Technologies offers a diverse range of programs tailored to meet the evolving demands of the IT industry:

- **Job-Guaranteed Courses:** Programs in Networking & Cybersecurity, Cloud Computing & DevOps, Java Full Stack Development, Certified Ethical Hacker, and Data Science & Machine Learning.
- **Global IT Certifications:** Training for certifications such as Cisco Certified Network Associate (CCNA), AWS Certified Solutions Architect Associate, Java Certification Programming, Microsoft Azure Solution Architect, and Python Programming.
- **Academic Programs:** Minor degrees in specialized fields like AI-ML, VLSI, HPC and Cloud Computing, and Cyber Security.
- **Government-Sponsored Training:** Initiatives like PMKVY Regular and PMKVY FutureSkill, aimed at upskilling and reskilling the workforce in collaboration with the Government of India.

Rooman Technologies stands as a beacon in the realm of IT training and skill development, continually

evolving to meet industry demands and empower individuals with the knowledge and skills necessary for a successful career in the everchanging technological landscape.

CHAPTER 3
VERILOG AND SYSTEM VERILOG IN VLSI DESIGN

3.1 VERILOG

Verilog is a hardware description language (HDL) used to model and simulate digital systems. It allows designers to describe the behavior and structure of electronic circuits at various levels of abstraction, from high-level functional specifications to low-level gate representations. Originally developed in the 1980s, Verilog has become a standard language in the field of electronic design automation (EDA) for designing integrated circuits (ICs). Its syntax and constructs enable efficient design, verification, and synthesis of complex digital systems.

3.1.1 Fundamentals of VLSI Design and Verilog Basics

VLSI (Very Large-Scale Integration) technology is a critical advancement in electronics that allows the integration of millions or billions of transistors on a single chip. This technology has enabled the development of complex integrated circuits (ICs) that are smaller, faster, and more power-efficient than their predecessors.

The design process for VLSI circuits typically follows several stages:

1. **Specification**: This initial phase involves defining the requirements and functionality of the circuit, including performance metrics, power consumption, and area constraints. Specifications serve as a blueprint for the design process.
2. **Design**: This phase includes:
   - **Architectural Design**: High-level structure of the system, defining how different components interact.
   - **Logic Design**: Functional blocks that perform specific operations, often represented using Boolean algebra.
   - **Circuit Design**: Detailed implementation of the logic design, specifying the actual components (gates, flip-flops, etc.) and their interconnections.
3. **Verification**: Ensuring that the design meets specifications through simulation and testing. This phase is critical to identify and

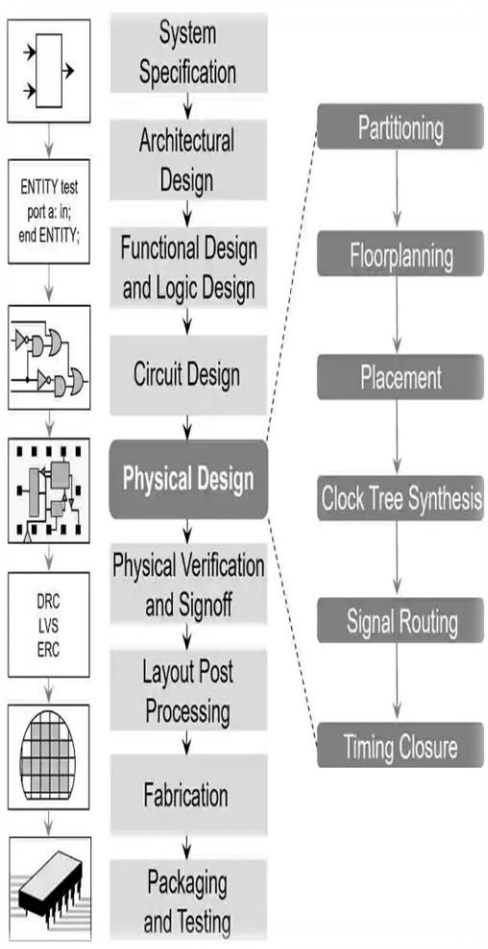

Figure 3. 1: VLSI Design Process

rectify errors before fabrication. Verification can be done using simulation tools that model the behavior of the design.

4. **Physical Design**: Involves floor planning (arranging major blocks), placement (positioning standard cells), and routing (connecting the cells). This stage translates the logical design into a physical layout that can be manufactured.
5. **Fabrication**: The final stage where the design is manufactured into a physical chip using semiconductor fabrication processes. This involves photolithography, etching, and other processes to create the integrated circuit.

3.1.2 VLSI Data Types, Signal Characteristics, and Continuous Assignments

In Verilog, various data types are used to represent different kinds of signals and variables:

- **Wires**: Used for connecting different components; they do not store values and are driven by continuous assignments. Wires can be thought of as physical connections in a circuit.
- **Registers (reg)**: Used to store values; they can hold data until explicitly changed, making them suitable for modeling memory elements. Registers are essential for sequential logic, where the output depends on past inputs.
- **Integers**: Used for arithmetic operations and can represent a range of values, typically 32 bits in size. They are useful for counting and indexing.
- **Real Numbers**: Used for floating-point calculations, allowing for more complex mathematical operations. Real numbers are often used in simulations that require precision.
- **Memories**: Arrays of registers that can store multiple values, useful for implementing RAM or ROM structures. Memories can be indexed to access specific data.

Signal characteristics include:

- **Strength**: Indicates how strong a signal is (e.g., strong, weak, high impedance), which is important in multi-driver scenarios where multiple sources may drive a signal.
- **Logic Values**: Represent the state of a signal (0, 1, x for unknown, z for high impedance). These values are crucial for modeling real-world digital circuits, where signals may not always be in a defined state.

Continuous assignments are used to model combinational logic, where the output is continuously updated based on the inputs. The syntax for continuous assignments is straightforward, using the **assign** keyword.

3.1.3 Syntax, Semantics, and Core Representations in Verilog

Verilog syntax

the rules for writing valid code, including the use of keywords, operators, and punctuation. Understanding syntax is essential for writing correct Verilog code.

- **Keywords**: Reserved words in Verilog that have special meaning (e.g., **module**, **end module**, **assign**, **always**, **if**, **else**).
- **Operators**: Symbols that perform operations on variables (e.g., arithmetic operators like +, logical operators like **&&**, and bitwise operators like **&**).

Semantics

It refers to the meaning of the code and how it translates to circuit behavior. Understanding semantics is crucial for effective design, as it determines how the written code will behave in simulation and in hardware.

Core representations in Verilog include:

- **Behavioral Level**: Describes the functionality of the circuit using high-level constructs, focusing on what the circuit does rather than how it does it. For example, using **if** statements to describe logic.
- **Dataflow Level**: Describes how data moves through the circuit using continuous assignments. This level is useful for modeling combinational logic without specifying the underlying gates.
- **Gate Level**: Represents the circuit using basic logic gates (AND, OR, NOT), providing a detailed view of the circuit's structure. This level is essential for timing analysis and synthesis.
- **Switch Level**: Uses transistors to represent circuits, allowing for detailed analog simulations. This level is important for understanding the behavior of circuits at the transistor level.

3.1.4 Gate-Level Modeling and Simulation Techniques

Gate-level modeling involves using primitive logic gates to represent the circuit. This level of abstraction is essential for detailed timing analysis and accurate simulation. Various simulation techniques are employed to verify the design:

- **Functional Simulation**: Checks the logical behavior of the design without considering timing. This type of simulation is useful for initial verification of the design, ensuring that it behaves as expected under various input conditions.
- **Timing Simulation**: Takes into account the delays associated with gates and signal propagation, providing a more accurate representation of how the circuit will behave in real-world conditions. Timing simulation is crucial for identifying timing violations.

- **Static Timing Analysis (STA)**: Analyzes the timing characteristics of the circuit without running a simulation, identifying potential timing violations such as setup and hold time violations. STA is essential for ensuring that the circuit meets its timing requirements.
- **Event-Driven Simulation**: Processes events (changes in signal values) to simulate the circuit's behavior over time. This method is efficient for large designs, as it only simulates changes rather than evaluating the entire circuit continuously.

3.1.5 Procedural Blocks, Assignments, and Control Flow

Procedural blocks:

It is in Verilog, such as **always** and **initial** blocks, are used to describe sequential and combinational logic. The **always** block executes whenever there is a change in the signals listed in its sensitivity list. The **initial** block runs once at the start of the simulation.

Assignments can be:

- **Blocking Assignments (=)**: Execute sequentially, meaning the next statement waits for the current assignment to complete. This is useful for modeling combinational logic where the order of execution matters.
- **Non-Blocking Assignments (<=)**: Execute concurrently, allowing multiple assignments to occur simultaneously. This is particularly useful in modeling sequential logic, where the order of execution does not affect the final outcome.

Control flow:

It constructs like **if-else** and **case** statements allow for conditional execution of code. These constructs enable designers to implement complex logic based on specific conditions.

3.1.6 Looping Constructs, Tasks, Functions, and Compiler Directives

Looping constructs

These are in Verilog, such as **for**, **while**, **repeat**, and **forever**, allow for the execution of code blocks multiple times based on specified conditions. These constructs are useful for generating repetitive structures or performing operations on arrays.

Tasks and functions are procedural blocks that facilitate code reuse:

- **Tasks**: Can have input, output, and inout arguments and can contain timing controls. Tasks are useful for encapsulating complex operations that may require multiple statements.

- **Functions**: Return a single value and cannot contain timing controls. Functions are typically used for calculations or operations that need to return a result.

Compiler directives

There are several Directives such as **include**, **define**, and **ifdef**, control the compilation process and enhance code organization. These directives allow for conditional compilation and code modularization, making it easier to manage large designs.

3.1.7 Switch Level Modeling and VCD Files

Switch-level modeling describes circuits using transistors (MOSFETs) and provides a low-level representation for detailed analog simulations. This modeling is essential for understanding how signals propagate through the circuit at the transistor level. Switch-level models can capture the behavior of circuits under various conditions, including noise and temperature variations.

VCD (Value Change Dump) files are generated during simulation to record signal changes over time. These files are useful for debugging and waveform viewing, allowing designers to visualize how signals change in response to inputs. VCD files can be imported into waveform viewers to analyze the timing and behavior of the circuit.

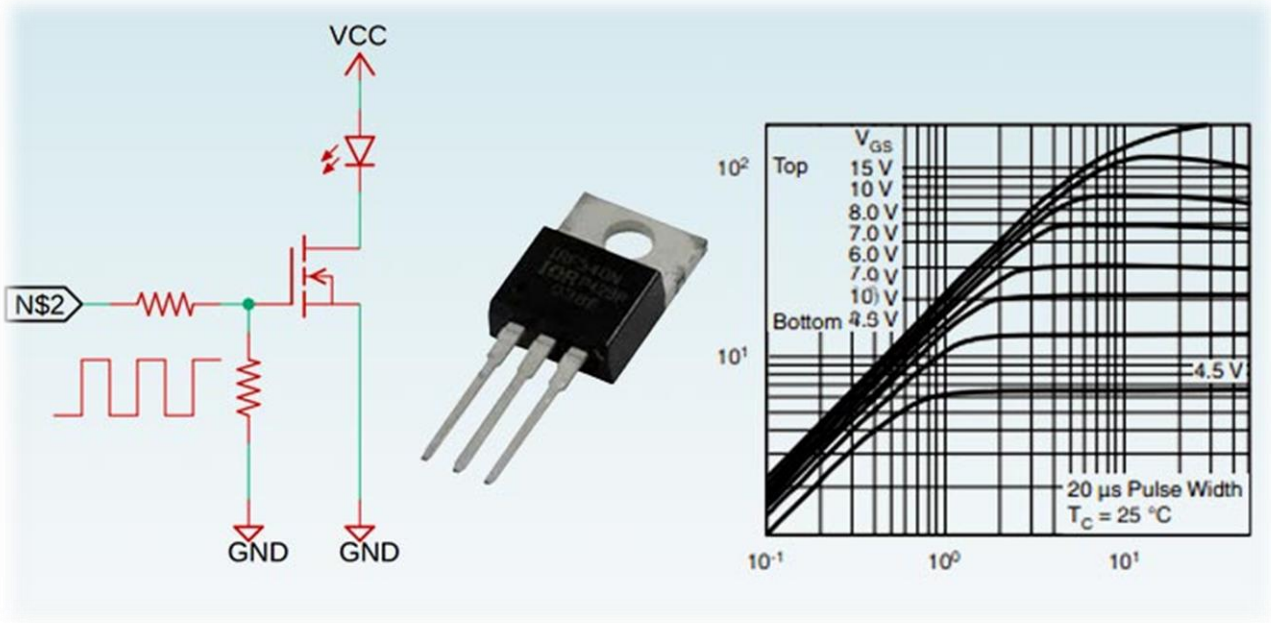

Figure 3. 2: MOSFET Switching Circuit

3.2 SYSTEM VERILOG

System Verilog is an extension of the Verilog hardware description language (HDL) that combines features for both design and verification of digital systems. It enhances traditional Verilog by introducing advanced constructs such as assertions, interfaces, and object-oriented programming (OOP) capabilities. Designed to

address the increasing complexity of integrated circuits (ICs), System Verilog improves the efficiency and effectiveness of the design and verification processes. As a result, it has become a standard language in the field of electronic design automation (EDA) for developing and validating complex hardware systems.

3.2.1 Introduction to System Verilog and Verification

System Verilog is a powerful hardware description and verification language that extends Verilog to address the complexities of modern digital design. It integrates features for both design and verification, making it a comprehensive tool for hardware engineers.

Key enhancements in System Verilog include:

- **Assertions**: These are used to specify expected behaviors in the design, allowing for automatic checking during simulation. Assertions help catch errors early in the design process and improve reliability.
- **Coverage**: System Verilog provides coverage constructs that measure how thoroughly a design has been tested, ensuring that all scenarios are considered and improving test quality.
- **Object-Oriented Programming (OOP)**: System Verilog introduces OOP concepts, enabling better code organization, reuse, and abstraction in testbenches. This allows for the creation of reusable components and more maintainable code.

Verification is a critical aspect of the design process, as it ensures that the hardware behaves as intended under various conditions. System Verilog provides a robust framework for creating testbenches, enabling designers to simulate and validate their designs effectively.

3.2.2 Data Types, Enumerations, and Constants

System Verilog introduces a rich set of data types that enhance the expressiveness and functionality of the language compared to traditional Verilog. Key data types include:

- **Logic**: A 4-state data type that can represent values 0, 1, x (unknown), and z (high impedance). It is preferred over the traditional **reg** type for better simulation accuracy and to avoid ambiguity in signal states.

- **Bit**: A 2-state data type that can represent only 0 and 1. It is useful for applications where unknown states are not needed, providing a more efficient representation.
- **Integer**: A signed 32-bit data type used for arithmetic operations. It is commonly used for counting and indexing.
- **Real**: A floating-point data type used for precise calculations, particularly in simulations that require high accuracy.
- **Enumerations**: User-defined data types that allow designers to create named constants, improving code readability and maintainability. For example, an enumeration for states in a finite state machine (FSM) can be defined as: typedef enum {IDLE, RUNNING, STOPPED} state_t;
- **Constants**: Defined using **parameter** and **localparam**, constants allow designers to create fixed values that can be reused throughout the code, enhancing maintainability and reducing errors.

3.2.3 Operators, Block Names, and Local Variables

System Verilog enhances the operator set available for designers, including new logical, bitwise, and relational operators. These operators allow for more expressive and concise code.

- **Block Names**: System Verilog allows designers to name blocks of code, such as **always** and **initial** blocks. This feature improves code readability and organization, making it easier to understand the structure of the design.

  For example: **always @** (posedge clk) begin : my_block
  // Code here
  end

- **Local Variables**: Local variables can be declared within procedural blocks, limiting their scope to that block. This prevents naming conflicts and enhances code modularity. Local variables can be declared using the **var** keyword, and they can be of any data type.

  For example: initial begin
  var int local_var; // Local variable
  local_var = 5;
  end

3.2.4 Working with Arrays, Queues, and Data Structures

System Verilog introduces advanced data structures that facilitate the management of collections of data:

- **Arrays**: System Verilog supports both fixed-size and dynamic arrays. Fixed-size arrays have a predetermined size, while dynamic arrays can grow or shrink during runtime.
  For example: int fixed_array[10]; // Fixed-size array
  int dynamic_array[]; // Dynamic array
- **Associative Arrays**: These are arrays indexed by arbitrary values (not just integers), allowing for more flexible data storage. They are particularly useful for implementing lookup tables.
  For example: int associative_array[string]; // Associative array indexed by strings
- **Queues**: A queue is a dynamic data structure that allows for the storage of elements in a first-in-first-out (FIFO) manner. Queues can be resized dynamically and are useful for managing data streams.
  For example: int queue[$]; // Dynamic queue
- **Structures and Unions**: System Verilog allows the creation of user-defined data types using structures and unions, enabling designers to group related data together. Structures can contain multiple fields of different data types, while unions allow for the storage of different data types in the same memory location.
  For example: typedef struct {
  int id;
  logic [7:0] data;
  } my_struct;

3.2.5 Multithreading, Interfaces, and Clocking Structures

System Verilog introduces features that facilitate concurrent execution and communication between different components:

- **Multithreading**: System Verilog supports concurrent execution through the use of **fork** and **join** constructs, allowing multiple processes to run simultaneously. This is particularly useful for modeling complex interactions in testbenches.
  For example: initial begin
  fork
  // Process 1
  begin
  // Code for process 1
  end

```
// Process 2
begin
// Code for process 2
end
join
end
```

- **Interfaces**: Interfaces in System Verilog provide a way to group related signals together, simplifying the connection between modules. They encapsulate the communication protocol and can include both signals and methods.

  For example:
  ```
  interface my_interface;
      logic clk;
      logic reset;
      logic [7:0] data;
  endinterface
  ```

- **Clocking Structures**: Clocking blocks are used to define the timing of signal sampling and driving, providing a clear way to manage timing in testbenches. They specify when signals should be sampled and driven, helping to avoid race conditions.

  For example:
  ```
  clocking cb @(posedge clk);
      input data;
      output control;
  endclocking
  ```

3.2.6 Advanced Programming Constructs and Object-Oriented Concepts

System Verilog incorporates advanced programming constructs and object-oriented programming (OOP) concepts that enhance code organization and reusability:

- **Classes**: System Verilog supports the creation of classes, allowing designers to define objects that encapsulate data and behavior. Classes can have properties (data members) and methods (functions) that operate on those properties.

  For example:
  ```
  class my_class;
      int value;
      function void set_value(int v);
      value = v;
  endfunction
  ```

endclass

- **Inheritance**: Classes can inherit properties and methods from other classes, promoting code reuse and reducing redundancy. This allows for the creation of specialized classes that extend the functionality of base classes.
- **Polymorphism**: System Verilog supports polymorphism, allowing methods to be overridden in derived classes. This enables the same method name to behave differently based on the object type.
- **Randomization**: System Verilog provides built-in support for randomization, allowing designers to generate random values for testing purposes. This is particularly useful in verification environments to explore a wide range of scenarios.
- **Functional Coverage**: System Verilog includes constructs for functional coverage, allowing designers to measure how well their tests cover the design's functionality. This helps ensure that all critical scenarios are tested.

3.3 OPEN ROAD

The OpenROAD project is an open-source initiative focused on automating the physical design stage of Very-Large-Scale Integration (VLSI) circuits. Its main goal is to develop a comprehensive, autonomous flow that converts Register Transfer Level (RTL) descriptions into the final manufacturing layout, known as GDSII. By offering a free and accessible suite of Electronic Design Automation (EDA) tools, OpenROAD seeks to democratize chip design, reduce barriers to entry for new designers, and encourage innovation within the VLSI industry. The project encompasses critical design phases, including synthesis, placement, clock tree synthesis, and routing, thereby contributing to a more open and collaborative ecosystem for hardware development.

3.3.1 Overview of the OpenROAD Project and ASIC Design Flow

The OpenROAD project is an open-source initiative aimed at automating the physical design stage of Application-Specific Integrated Circuit (ASIC) design. It provides a complete RTL-to-GDSII flow, enabling designers to transform high-level hardware descriptions into final manufacturing layouts. The project focuses on democratizing chip design by offering a suite of Electronic Design Automation (EDA) tools that are accessible to everyone.

ASIC Design Flow:

The ASIC design flow typically consists of several key stages:

1. **Specification**: Defining the requirements and functionality of the ASIC.
2. **RTL Design**: Writing the hardware description in a high-level language like Verilog or VHDL.
3. **Synthesis**: Converting the RTL code into a gate-level netlist using tools like Yosys.
4. **Floor Planning**: Organizing the layout of the chip, determining the placement of functional blocks.
5. **Placement**: Positioning standard cells and components within the defined floor plan.
6. **Clock Tree Synthesis (CTS)**: Designing the clock distribution network to ensure minimal skew and delay.
7. **Routing**: Connecting the various components and cells to form the complete circuit.
8. **Sign-off**: Performing final checks, including timing analysis and verification, before generating the GDSII layout for manufacturing.

The OpenROAD project integrates these stages into a cohesive flow, allowing for a streamlined design process that enhances efficiency and reduces time-to-market.

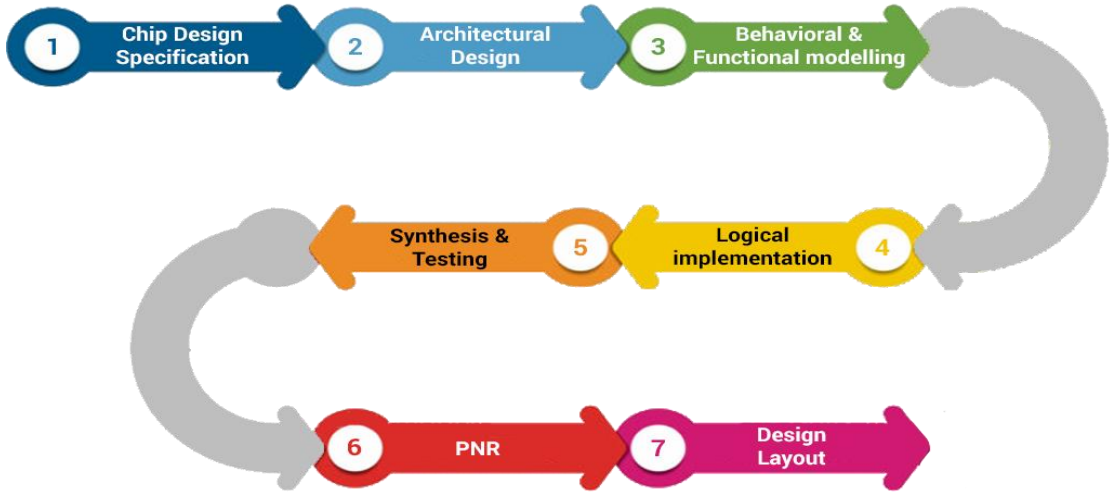

Figure 3. 3: Asic Design Process

3.3.2 Synthesis Using Yosys and Timing Analysis

Synthesis is the process of converting RTL code into a gate-level netlist, which represents the logical structure of the design using standard cells. The OpenROAD project utilizes Yosys, an open-source synthesis tool, to perform this task.

Yosys Synthesis:

- **Functionality**: Yosys supports various synthesis techniques, including technology mapping, optimization, and logic synthesis. It can handle multiple input formats and generate output in various formats compatible with downstream tools.
- **Optimization**: Yosys applies various optimization techniques to reduce area, power, and delay, ensuring that the synthesized netlist meets the design specifications.

Timing Analysis:

After synthesis, timing analysis is performed to ensure that the design meets its timing constraints. This involves:

- **Static Timing Analysis (STA)**: Evaluating the timing of the circuit without simulating it, checking for setup and hold time violations.
- **Timing Reports**: Generating reports that provide insights into critical paths, slack, and timing violations, allowing designers to make informed decisions about optimizations.

3.3.3 Floor Planning, IR Drop Analysis, and Placement

Floor planning is a crucial step in the physical design process, where the overall layout of the chip is defined. This stage involves organizing the placement of functional blocks and ensuring efficient use of space.

Floor Planning:

- **Block Arrangement**: Designers determine the arrangement of major functional blocks, considering factors like signal integrity, power distribution, and routing congestion.
- **Aspect Ratio and Area**: The floor plan must adhere to specific aspect ratios and area constraints to optimize performance and manufacturability.

IR Drop Analysis:

- **Power Integrity**: IR drop analysis assesses the voltage drop across power distribution networks due to current flow. It ensures that all components receive adequate voltage levels during operation.
- **Simulation Tools**: Tools are used to simulate power distribution and identify areas where voltage drops may exceed acceptable limits, allowing for design adjustments.

Placement:

- **Standard Cell Placement**: After floor planning, standard cells are placed within the defined regions. The placement process aims to minimize wire length and congestion while adhering to design rules.

- **Optimization**: Placement algorithms optimize the arrangement of cells to improve performance metrics such as timing and power consumption.

3.3.4 Clock Tree Synthesis (CTS) and Routing Design Flow

Clock Tree Synthesis (CTS) is a critical step in the design flow that focuses on creating a balanced clock distribution network to minimize skew and ensure reliable clock delivery to all components.

Clock Tree Synthesis (CTS):

- **Clock Network Design**: CTS involves designing a tree-like structure for the clock signal, ensuring that all flip-flops receive the clock signal simultaneously.
- **Skew Minimization**: The goal is to minimize clock skew, which can lead to timing violations. Techniques such as buffer insertion and wire sizing are used to achieve this.

Routing Design Flow:

- **Global Routing**: The initial routing phase determines the general paths for signal connections without considering detailed design rules.
- **Detailed Routing**: The detailed routing phase involves creating the actual connections between cells while adhering to design rules and constraints. This includes layer assignment, via placement, and ensuring signal integrity.
- **DRC Checks**: Design Rule Checks (DRC) are performed to ensure that the routing adheres to manufacturing constraints, preventing issues during fabrication.

3.3.5 Integration of OpenROAD Components in ASIC Design

The integration of OpenROAD components into the ASIC design flow allows for a seamless transition between different stages of the design process. Each component of OpenROAD is designed to work together, providing a cohesive environment for designers.

Component Integration:

- **Modular Architecture**: OpenROAD's modular architecture allows designers to select and integrate specific components based on their design requirements. This flexibility enables customization of the design flow.
- **Interoperability**: Components such as Yosys for synthesis, placement tools, and routing engines can communicate effectively, sharing data and results to streamline the design process.

Benefits of Integration:

- **Efficiency**: The integrated flow reduces the time and effort required to move between different design

stages, enhancing overall productivity.

- **Collaboration**: OpenROAD encourages collaboration among developers and users, allowing for continuous improvement and adaptation of tools based on user feedback and industry needs.
- **Open-Source Community**: The open-source nature of OpenROAD fosters a community-driven approach, where users can contribute to the development and enhancement of tools, driving innovation in the ASIC design space.

CHAPTER 4

PROJECT DETAILS

4.1 PROJECT-9: DESIGN OF AN 4 BIT ADDER-SUBTRACTOR USING VERILOG OR SYSTEM VERILOG OR VHDL

4.1.1 Introduction

This report presents the RTL design, functional verification, and physical layout generation of a 4-bit Adder-Subtractor using Verilog. The design is verified using UVM methodology and implemented in OpenROAD to generate the GDS-II layout. The analysis includes the evaluation of area, timing, and power consumption.

A	B	Mode	Operation	S	Cout
0000	0000	0	Addition	0000	0
0001	0001	0	Addition	0010	0
0010	0001	0	Addition	0011	0
0100	0011	0	Addition	0111	0
1000	0110	0	Addition	1110	0
0001	0001	1	Subtraction	0000	1
0100	0010	1	Subtraction	0010	1
1000	0011	1	Subtraction	0101	1
1111	0001	1	Subtraction	1110	1

Figure 4. 1: Truth table of 4 Bit Adder and Subtractor

4.1.2 Design Architecture

- **Full Adder Module**: Performs single-bit addition with carry propagation.
- **4-bit Adder-Subtractor Module:**

  Takes two 4-bit inputs and a mode selection bit.

  Uses XOR gates to compute the two's complement of B when performing Subtraction.

  Uses a chain of full adders to compute the sum or difference.

  Outputs the 4-bit result and carry/borrow flag.

4.1.3 Block Diagram

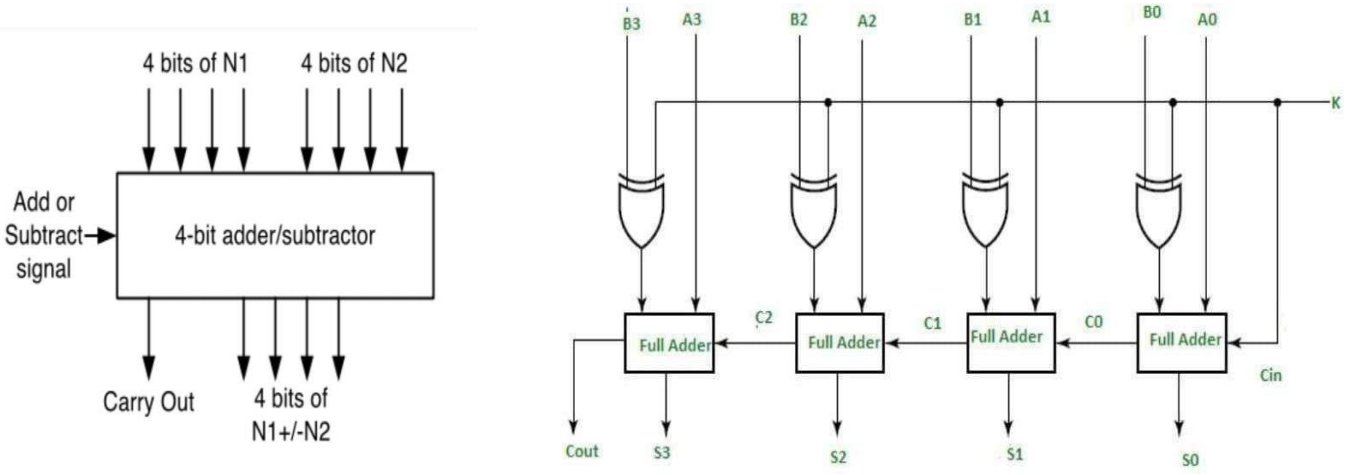

Figure 4. 2: Block Diagram of 4 Bit Adder-Subtractor

4.1.4 RTL Code

```
module full_adder (
    input A,  // Input bit
    A input B,  // Input
    bit B input Cin, //
    Carry-in output S,  //
    Sum
    output Cout // Carry-out

);

    assign S = A ^ B ^ Cin;  // Sum calculation

    assign Cout = (A & B) | (Cin & (A ^ B));  // Carry-out calculation

endmodule


module ripple_carry_adder_subtractor #(parameter SIZE = 4) (
    input [SIZE-1:0] A, B,
```

```
    input CTRL,

    output [SIZE-1:0] S, Cout

);

    bit [SIZE-1:0] Bc;

    genvar g;
    assign Bc[0] = B[0] ^ CTRL;
    full_adder fa0 (A[0], Bc[0], CTRL, S[0], Cout[0]);
    generate

        for (g = 1; g < SIZE; g++) begin
            assign Bc[g] = B[g] ^ CTRL;
            full_adder fa (A[g], Bc[g], Cout[g-1], S[g], Cout[g]);
        end
endgenerate
endmodule
```

Testbench Code

```
module RCAS_TB;

wire [3:0] S, Cout;

reg [3:0] A, B;

reg ctrl;

ripple_carry_adder_subtractor rcas (A, B, ctrl, S, Cout);

initial begin

$monitor("CTRL=%b: A = %d, B = %d --> S = %d, Cout[3] = %d", ctrl, A, B, S, Cout[3]);

ctrl = 0;

A = 1; B = 0; #10;
```

```
A = 2; B = 4; #10;


A = 4'hB; B = 4'h6; #10; A = 5;

B = 3; #10;

ctrl = 1;

A = 1; B = 0; #10;

A = 2; B = 4; #10;

A = 4'hB; B = 4'h6; #10; A = 5;

B = 3; #10;

#10; $finish;

end

initial begin

$dumpfile("waves.vcd");

$dumpvars;

end

endmodule
```

4.1.5 Simulation ,Verification And Simulation Results

Testbench Setup:

- Input values are provided for A, B, and Mode.
- Output values S and Cout are observed.

Expected Output:

A = 00000010, B = 00000100, Cin = 1, S = 00001011, Cout = 1

A = 00000100, B = 00000110, Cin = 0, S = 00001110, Cout = 0

A = 00001000, B = 00000011, Cin = 0, S = 00001011, Cout = 0

4.1.6 Simulated Input-Output Waveforms

Below are the simulated input-output waveforms:

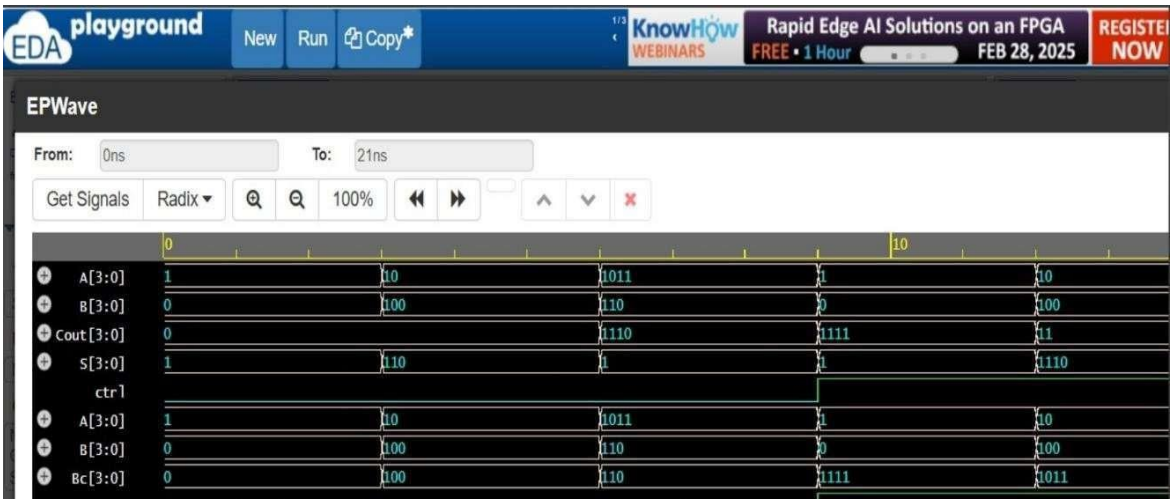

Figure 4. 3: Stimulation Input-Output Waveforrm of 4 Bit Adder-Subtractor

4.1.7 Results and Discussion

The 4-bit Adder cum Subtractor was successfully implemented and verified. The simulation results matched the expected behavior, confirming the correctness of the design.

EDA LINK: https://edaplayground.com/x/Grdp

4.2 EVALUATION CRITERIA FOR BLOCK-LEVEL VERIFICATION IN UVM

4.2.1 Testbench Architecture (50%)

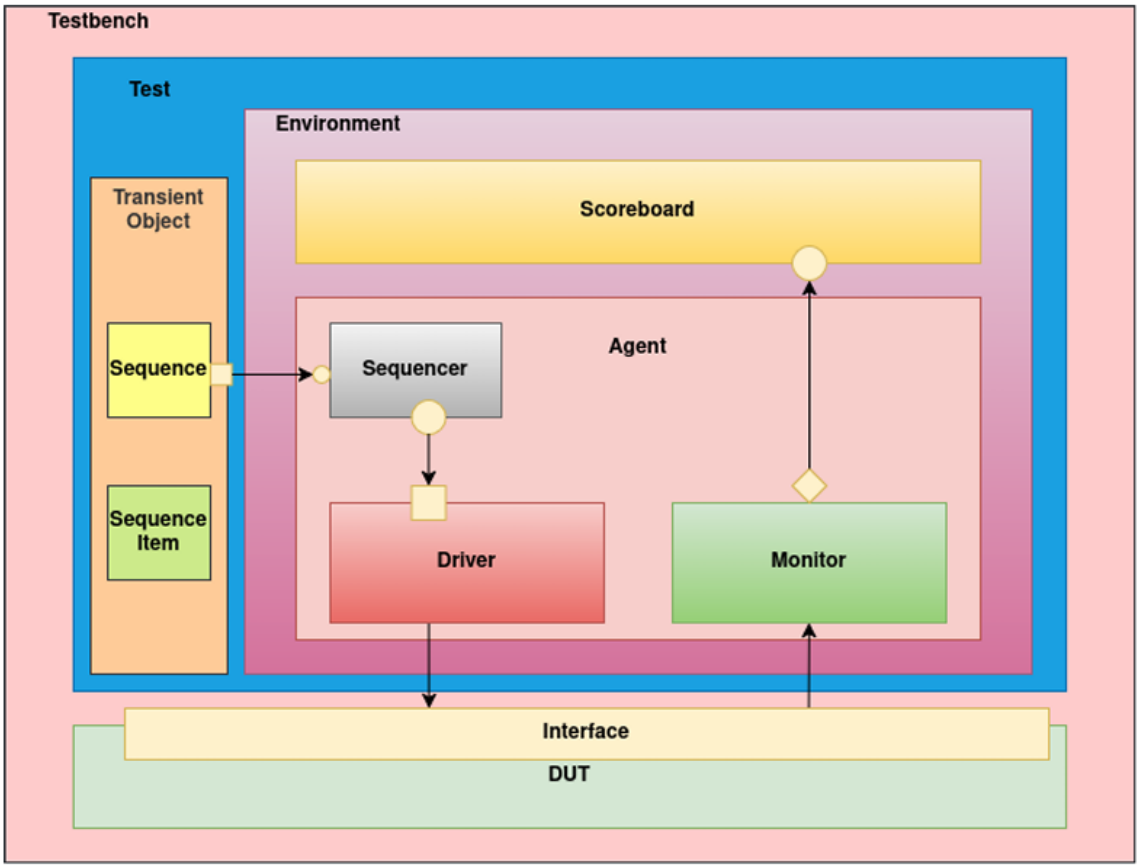

Figure 4. 4: Testbench Architecture

- Proper use of UVM components
- Adherence to the UVM factory and configuration mechanism.
- Proper use of virtual sequences and sequence layering if applicable.

4.2.2 Driver

```
class adder_subtractor_driver extends uvm_driver #(adder_subtractor_seq_item);

`uvm_component_utils(adder_subtractor_driver)

virtual adder_subtractor_if tb_if;

adder_subtractor_seq_item req;

function new(string name = "adder_subtractor_driver", uvm_component parent);

super.new(name, parent);

endfunction

virtual function void build_phase(uvm_phase phase);

super.build_phase(phase);

if (!uvm_config_db#(virtual adder_subtractor_if)::get(this, "", "vif", tb_if)) begin

`uvm_fatal("adder_subtractor_driver", "Failed to get interface")

end

 endfunction

virtual task run_phase(uvm_phase phase);

forever begin

seq_item_port.get_next_item(req);

tb_if.in1 = req.in1;

tb_if.in2 = req.in2;

tb_if.operation = req.operation;

#10;

seq_item_port.item_done();
```

```
end

endtask

endclass
```

4.2.3 Monitor

```
class adder_subtractor_monitor extends uvm_monitor;

`uvm_component_utils(adder_subtractor_monitor)

virtual adder_subtractor_if tb_if;

uvm_analysis_port #(adder_subtractor_seq_item) monitor_port;

adder_subtractor_seq_item tr;

function new(string name = "adder_subtractor_monitor", uvm_component parent);

super.new(name, parent); endfunction

virtual function void build_phase(uvm_phase phase);

super.build_phase(phase);

if (!uvm_config_db#(virtual adder_subtractor_if)::get(this, "", "vif", tb_if)) begin

`uvm_fatal("adder_subtractor_monitor", "Failed to get interface")

end

tr = adder_subtractor_seq_item::type_id::create("tr");

monitor_port = new("monitor_port", this);

 endfunction

virtual task run_phase(uvm_phase phase);

forever begin

#10; // Capture values after some delay tr.in1 =

tb_if.in1;
```

```
tr.in2 = tb_if.in2;
tr.operation = tb_if.operation;
tr.out = tb_if.out;
tr.cout = tb_if.cout;
tr.overflow = tb_if.overflow;
monitor_port.write(tr);
end
endtask
 endclass
```

4.2.4 Agent

```
class adder_agent extends uvm_agent;
`uvm_component_utils(adder_agent)
adder_driver drv; adder_monitor mon;
uvm_sequencer #(adder_transaction) seqr;
function new(string name = "adder_agent", uvm_component parent=null);
super.new(name, parent); endfunction
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
drv = adder_driver::type_id::create("drv", this);
mon =adder_monitor::type_id::create("mon",  this);

seqr = uvm_sequencer#(adder_transaction)::type_id::create("seqr", this);
 endfunction
```

```
virtual function void connect_phase(uvm_phase phase);
 super.connect_phase(phase);
drv.seq_item_port.connect(seqr.seq_item_export);


endfunction
 endclass
```

4.2.5 Environment

```
class adder_subtractor_env extends uvm_env;
`uvm_component_utils(adder_subtractor_env)
adder_subtractor_agent agent;
adder_subtractor_scoreboard scoreboard;
function new(string name = "adder_subtractor_env", uvm_component parent);
super.new(name, parent);
endfunction
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
agent = adder_subtractor_agent::type_id::create("agent", this);
scoreboard = adder_subtractor_scoreboard::type_id::create("scoreboard", this);
endfunction
virtual function void connect_phase(uvm_phase phase);
agent.mon.monitor_port.connect(scoreboard.monitor_imp);
endfunction
 endclass
```

4.2.6 Test

```
class adder_subtractor_test extends uvm_test;

`uvm_component_utils(adder_subtractor_test)


adder_subtractor_env env;

adder_subtractor_seq seq;


function new(string name = "adder_subtractor_test", uvm_component parent);

super.new(name, parent);

endfunction

virtual function void build_phase(uvm_phase phase);

super.build_phase(phase);

env = adder_subtractor_env::type_id::create ("env", this);

seq = adder_subtractor_seq::type_id::create("seq");

endfunction

virtual task run_phase(uvm_phase phase);

phase.raise_objection(this);

seq.start(env.agent.seqr);

#1;

phase.drop_objection(this);

endtask

endclass
```

4.2.7 Testbench

```
import uvm_pkg::*;

`include "uvm_macros.svh"

`define WIDTH 8

`include "package.sv"

module tb_adder_subtractor;


// Declare the virtual interface to connect with the DUT

adder_subtractor_if #(`WIDTH) vif();

// Instantiate the DUT

adder_subtractor #(

.WIDTH(8)

) dut (

.in1(vif.in1),

.in2(vif.in2),

.operation(vif.operation),

.out(vif.out),

.cout(vif.cout),

.overflow(vif.overflow)

);

initial begin

uvm_config_db#(virtual adder_subtractor_if)::set(null, "*", "vif", vif);

run_test("adder_subtractor_test");
```

```
end

endmodule
```

4.3 STIMULUS GENERATION(15%)

- Development of constrained-random and directed test sequences.
- Use of UVM sequences and transaction-based stimulus generation.
- Ability to generate different corner cases and invalid scenarios.
- Parameterization and reuse of sequences.

4.3.1 Sequence Item

```
class adder_subtractor_seq_item extends uvm_sequence_item;

rand bit [`WIDTH-1:0] in1; rand bit

[`WIDTH-1:0] in2; rand bit operation;

bit [`WIDTH-1:0] out;

bit cout;

bit overflow;

function new(string name = "adder_subtractor_seq_item");

super.new(name);

endfunction

`uvm_object_utils_begin(adder_subtractor_seq_item)

`uvm_field_int(in1, UVM_ALL_ON)

`uvm_field_int(in2, UVM_ALL_ON)

`uvm_field_int(operation, UVM_ALL_ON)

`uvm_field_int(out, UVM_ALL_ON)

`uvm_field_int(cout, UVM_ALL_ON)

`uvm_field_int(overflow, UVM_ALL_ON)
```

```
`uvm_object_utils_end

endclass
```

4.3.2 Sequence

```
class adder_subtractor_seq extends uvm_sequence #(adder_subtractor_seq_item);

`uvm_object_utils(adder_subtractor_seq)

adder_subtractor_seq_item req;

function new(string name = "adder_subtractor_seq");

super.new(name);

endfunction

virtual task body();

req = adder_subtractor_seq_item::type_id::create("req");

repeat(1) begin

start_item(req);
if (!req.randomize()) begin

`uvm_error("adder_subtractor_seq", "Randomization failed")

end finish_item(req);

end

endtask

endclass
```

4.4 SCOREBOARDING AND CHECKING(25%)

- Implementation of functional and self-checking scoreboard.
- Use of predictive models and golden reference comparison.
- Effective use of UVM phases for checking.

4.4.1 Scoreboard

```
class adder_subtractor_scoreboard extends uvm_scoreboard;

`uvm_component_utils(adder_subtractor_scoreboard)
uvm_analysis_imp#(adder_subtractor_seq_item, adder_subtractor_scoreboard)

monitor_imp;

function new(string name = "adder_subtractor_scoreboard", uvm_component parent);

super.new(name, parent);

monitor_imp = new("monitor_imp", this);

endfunction

virtual function void write(adder_subtractor_seq_item tr);

bit [`WIDTH-1:0] expected_out;

bit expected_cout;

bit expected_overflow;

if (tr.operation == 0) begin

{expected_cout, expected_out} = tr.in1 + tr.in2; end else

begin

{expected_cout, expected_out} = tr.in1 - tr.in2; end

if (tr.out !== expected_out) begin

`uvm_error("adder_subtractor_scoreboard", $sformatf("Mismatch: in1=%0d, in2=%0d,
op=%0b, expected_out=%0d, actual_out=%0d , expected_cout=%0d, actual_cout=%0d",
tr.in1, tr.in2, tr.operation, expected_out, tr.out,expected_cout,tr.cout))

end else begin

//`uvm_info("adder_subtractor_scoreboard", "Transaction matched expected

values", UVM_MEDIUM)

`uvm_info("adder_subtractor_scoreboard", $sformatf("Match: in1=%0d, in2=%0d,
```

```
op=%0b, expected_out=%0d, actual_out=%0d ,expected_cout=%0d, actual_cout=%0d",
tr.in1, tr.in2, tr.operation, expected_out, tr.out,tr.out,expected_cout,tr.cout),UVM_LOW)

end

endfunction
endclass
```

4.5 DEBUGGING AND LOGS(5%)

- Effective use of UVM messaging and verbosity levels.
- Debugging skills and ability to interpret waveforms and logs.
- Error detection.
- Documentation of issues and resolutions.

4.5.1 Waveform (In Testbench)

```
initial begin
   $dumpfile("dump.vcd");
   $dumpvars();
end
```

4.6 UVM REPORT

```
** Report counts by severity
UVM_INFO :    3
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
```

Figure 4. 5: UVM Report

4.7 CODE QUALITY AND BEST PRACTICES (5%)

- Consistency in naming conventions and coding style.
- Use of parameterized and reusable components.
- Proper comments and documentation within the code.

- Efficient and optimized coding practices.

EDA LINK: https://edaplayground.com/x/PvrG

4.8 GENERATE GDS USING OPENROAD

In this section, the layout of the RTL code has been generated using the OpenROAD software tool.
Technology/Platform utilized: nangate45

Instructions of the config.mk

export DESIGN_NICKNAME = adder_subtractor

export DESIGN_NAME = adder_subtractor export

PLATFORM = gf180

export VERILOG_FILES = $(sort $(wildcard

$(DESIGN_HOME)/src/$(DESIGN_NICKNAME)/ adder_subtractor.v))

export SDC_FILE =$(DESIGN_HOME)/$(PLATFORM)/$(DESIGN_NICKNAME)/constraint.sdc #export

PLACE_PINS_ARGS = -min_distance 4 -min_distance_in_tracks export

CORE_UTILIZATION = 0.5

#export CORE_ASPECT_RATIO = 1

#export CORE_MARGIN = 2 export

PLACE_DENSITY = 0.1 export

TNS_END_PERCENT = 100

#export EQUIVALENCE_CHECK ?= 0

#export REMOVE_CELLS_FOR_EQY = sky130_fd_sc_hd__tapvpwrvgnd*

#export FASTROUTE_TCL =$(DESIGN_HOME)/$(PLATFORM)/$(DESIGN_NICKNAME)/fastroute.tcl

export REMOVE_ABC_BUFFERS = 1

Instructions of the constraint.sdc

current_design adder_subtractor

set clk_name v_clk #set

clk_port_name clk set

clk_period 2.5

set clk_io_pct 0.2

#create_clock -name $clk_name -period $clk_period

set non_clock_inputs [lsearch -inline -all -not [all_inputs]]

set_input_delay [expr $clk_period * $clk_io_pct] -clock $clk_name $non_clock_inputs

set_output_delay [expr $clk_period * $clk_io_pct] -clock $clk_name [all_outputs]

4.9 LAYOUT OF THE DESIGN

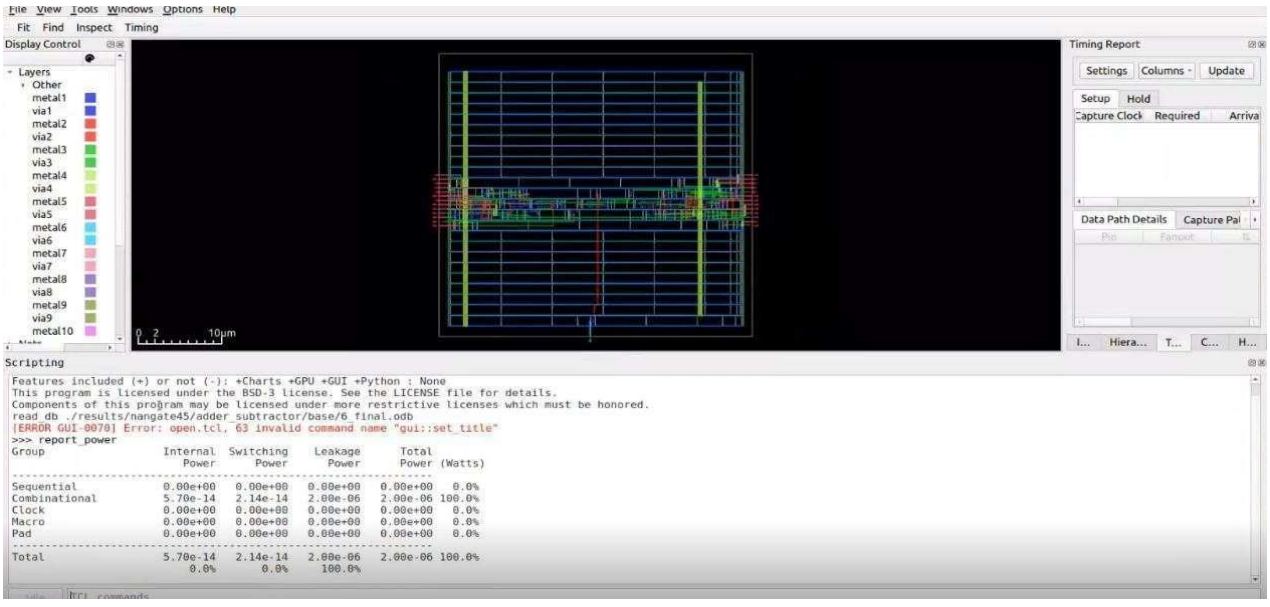

Figure 4. 6: Layout Design of the 4 Bit Adder-Subtractor

4.10 PERFORMANCE ANALYSIS

4.10.1 Power Measurement:

Group	Internal Power	Switching Power	Leakage Power	Total Power (Watts)
Sequential	0.00e+00	0.00e+00	0.00e+00	0.00e+00 0.0%
Combinational	5.70e-14	2.14e-14	2.00e-06	2.00e-06 100.0%
Clock	0.00e+00	0.00e+00	0.00e-00	0.00e+00 0.0%
Macro	0.00e+00	0.00e+00	0.00e+00	0.00e+00 0.0%
Pad	0.00e+00	0.00e+00	0.00e+00	0.00e+00 0.0%
Total	5.70e-14 0.0%	2.14e-14 0.0%	2.00e-06 100.0%	2.00e-06 100.0%

Figure 4. 7: Performance Analysis of 4 Bit Adder-Subtractor

4.10.2 Area Measurement:

- **Design Area**: 97 μ^2
- **Utilization**: 8%

4.10.3 Timing Information:

Delay (ns)	Time (ns)	Description
0.00	0.00	v input external delay
0.00	0.00	v A[0] (in)
0.00	0.00	v A[1] (in)
0.00	0.00	v A[2] (in)
0.10	0.10	^ inv1/Z (inverter cell)
0.12	0.22	^ and1/Z (AND gate output)
0.30	1.79	v Y[0] (out)
0.30	1.79	v carry_out (out)

Figure 4. 8: Timing Information of 4 Bit Adder-Subtractor

4.11 GENERATED GDS

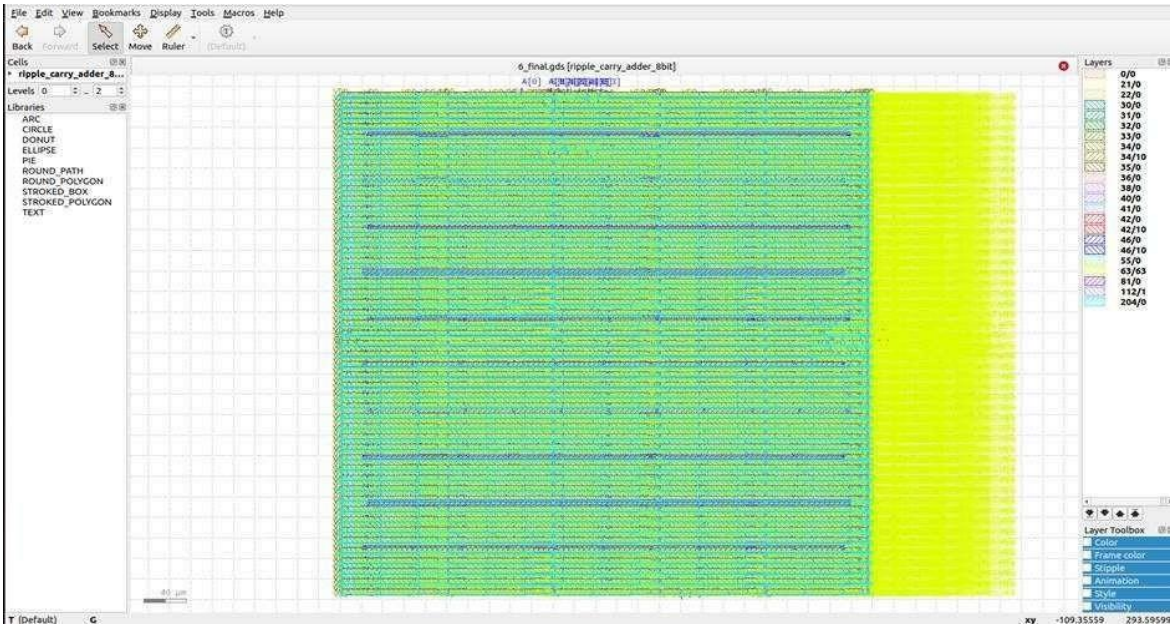

Figure 4. 9: Generated GDS

Google Drive Link of Tar file:

https://drive.google.com/drive/folders/1B4ZmyLTi849gBQYlzQli85uneZiN701o?usp=drive_link

In this report, the RTL code of the 4-bit Adder-Subtractor has been designed in Verilog. The code is successfully verified using UVM with a 100% test case pass. The design is further processed in the OpenROAD tool to generate its GDS-II layout using the GF180 platform. The generated layout consumes 108 nW power and occupies 4662 sq. µm area. There are no setup and hold violations, ensuring proper timing closure.

CHAPTER 5

LEARNING OUTCOMES

5.1 TECHNICAL, PROFESSIONAL, AND INTERPERSONAL SKILLS ACQUIRED

Over the course of the 460-hour internship, I gained a broad set of skills essential for a career in VLSI Design Engineering. Technically, I became proficient in using hardware description languages like Verilog HDL to design digital circuits at the register-transfer level (RTL). I developed skills in designing and simulating both combinational and sequential logic circuits, implementing finite state machines, and building functional blocks like adders, multiplexers, and ALUs. The use of tools such as Xilinx ISE, Vivado, and Yosys helped me understand how to synthesize RTL code into gate-level netlists.

On the back-end side, I learned the fundamentals of physical design, including floorplanning, standard cell placement, clock tree synthesis (CTS), routing, and parasitic extraction. I worked with tools like KLayout, and OpenROAD to carry out layout design, and GDSII generation. These experiences helped me understand how logical designs are mapped to physical silicon, and the importance of timing, area, and power optimization in layout planning.

In addition to technical skills, I developed key professional competencies. I learned how to manage tasks efficiently, maintain documentation, and troubleshoot design errors. Receiving regular mentor feedback and conducting peer reviews helped me improve my debugging approach, communication, and confidence. I also improved my interpersonal skills by collaborating with fellow interns in group projects, participating in discussions, and jointly solving design challenges.

5.2 KNOWLEDGE GAINED ABOUT THE INDUSTRY AND ITS PRACTICES

The internship bridged the gap between academic learning and industry demands. I became familiar with the end-to-end VLSI design flow, starting from specifications and RTL design to synthesis, place and route (PnR), and final sign-off processes. The internship emphasized the significance of structured and hierarchical design approaches that are commonly followed in industry.

I gained an understanding of how commercial chip design projects are managed in phases—starting from logic design and functional verification to timing closure, floorplanning, and tape-out. Important industry topics like Design for Testability (DFT), clock domain crossing (CDC), static timing analysis (STA), and

scan chain insertion were introduced. I also became aware of the challenges associated with real-world designs, such as managing multi-million gate circuits, power distribution networks, and signal integrity issues.

Moreover, the internship gave me exposure to the collaborative nature of VLSI projects. I learned that different teams often work simultaneously on RTL development, verification, physical design, and timing analysis, requiring constant coordination and version control. This understanding reinforced the importance of clear documentation, communication, and review mechanisms in professional semiconductor environments.

5.3 KEY TAKEAWAYS

- Developed end-to-end understanding of front-end and back-end VLSI design processes.
- Gained proficiency in industry-standard EDA tools for synthesis, simulation, and layout.
- Strengthened coding and debugging abilities in Verilog and digital logic design.
- Learned to apply practical knowledge to real-world design constraints and verification.
- Enhanced professional behavior, including time management, peer collaboration, and project documentation.
- Gained valuable insight into the structure, workflow, and expectations of the VLSI industry.
- Prepared for roles in ASIC/FPGA design, physical design engineering, and design verification.

CHAPTER 6

CONCLUSION

6.1 SUMMARY OF OVERALL EXPERIENCE

The internship in VLSI Design Engineering at Roman Technology Pvt Ltd, under the VTU collaboration, was a significant and transformative milestone in my academic and professional journey. It provided me with a rare opportunity to immerse myself in the practical world of VLSI—learning not only theoretical concepts but also how they are applied in real-time industrial environments. The structured curriculum, expert mentorship, and hands-on training made this internship both rigorous and rewarding.

In the front-end segment, I gained in-depth exposure to hardware description languages, specifically Verilog HDL, and understood how to model digital circuits at the RTL level. Designing and simulating various components such as ALUs, counters, FSMs, and datapaths enhanced my logic design skills. In the back-end part, I delved into the physical aspects of chip implementation, including floorplanning, placement and clock tree synthesischecks using tools such as OpenROAD, and KLayout.

Beyond technical learning, the internship sharpened my debugging abilities, code optimization techniques, and tool proficiency. It also encouraged critical thinking, initiative, and adaptability—qualities essential for success in high-performance semiconductor industries. Collaborating with peers and industry mentors exposed me to professional communication, project planning, and review practices that mirror real workplace environments. These engagements have helped me become more disciplined, focused, and prepared for industry expectations.

6.2 HIGHLIGHT THE VALUE ADDED TO OUR ACADEMIC JOURNEY AND CAREER PATH

From an academic standpoint, this internship has immensely reinforced and expanded my classroom learning. It connected the dots between abstract theoretical principles and their actual implementations in chip design and verification processes. For example, concepts like setup and hold time, logic synthesis, gate-level simulation, and parasitic extraction—often introduced briefly in coursework—were explored in detail with practical illustrations and real-world examples during the training.

Moreover, working on mini-projects such as the the 4 Bit Adder-Subtractor provided a holistic experience of designing a digital component from scratch and taking it through both simulation and layout stages. This

hands-on application of knowledge will directly benefit my final-year engineering projects, academic assessments, and even competitive technical events.

On the career front, the internship has built a solid foundation for my aspirations in the VLSI domain. It has helped me understand the roles and responsibilities associated with ASIC Design Engineer, Physical Design Engineer, and Verification Engineer profiles. The exposure to EDA tools and industrial workflows has also improved my employability and confidence in attending interviews and technical evaluations. Additionally, it has guided me in identifying my areas of strength—particularly in front-end RTL design and synthesis—which I now plan to pursue further through certification courses and research.

In conclusion, this internship was more than just a training program; it was a gateway into the professional world of semiconductor design. The knowledge, skills, and experience I gained will continue to influence my academic excellence and shape my long-term career in VLSI engineering.

REFERENCES

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Boston, MA, USA: Addison-Wesley, 2010.

[2] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[3] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. San Jose, CA, USA: Prentice Hall, 2003.

[4] M. Sarrafzadeh and C. K. Wong, *An Introduction to VLSI Physical Design*, 1st ed. New York, NY, USA: McGraw-Hill, 1996.

[5] K. Eshraghian, D. A. Pucknell, and S. Eshraghian, *Essentials of VLSI Circuits and Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

[6] Yosys Open SYnthesis Suite. [Online]. Available: https://yosyshq.net/yosys/

[7] OpenROAD Project. [Online]. Available: https://theopenroadproject.org/

[8] KLayout Layout Viewer and Editor. [Online]. Available: https://www.klayout.de/

[9] Rooman Technology Pvt Ltd – VLSI Internship Training Material, 2025. (Unpublished internal training resource)