

APPENDIX 1

Implementation of a RISC-V ALU (RV32I Subset)

INTERNSHIP PROJECT REPORT

by

MANNEM SHAMILI

(Reg. No.: NIELIT/NOI/OL/B67/23634)



Centre of Excellence in Chip Design
National Institute of Electronics and Information Technology
Noida, Uttar Pradesh
July & 2025

APPENDIX 2

Declaration by Author(s)

This is to declare that this report has been written by me/us. No part of the report is plagiarized from other sources. All information included from other sources have been duly acknowledged. I/We aver that if any part of the report is found to be plagiarized, I/we are shall take full responsibility for it.

Signature of author

MANNEM SHAMILI

Reg. No.: NIELIT/NOI/OL/B67/23634

Place:

Date:

ABSTRACT

This project presents the design and implementation of a RISC-V processor core based on the RV32I instruction subset. The main objective was to develop a simple, efficient, and synthesizable hardware description of the core using Verilog HDL. Emphasizing alignment with the open-source hardware movement, the project demonstrates the relevance of RISC-V in modern System-on-Chip (SoC) architectures. The processor was simulated, synthesized, and verified using a complete RTL-to-GDSII flow involving open-source EDA tools. The report concludes with insights into the processor's performance, potential for integration, and directions for future enhancement.

APPENDIX 3

TABLE OF CONTENTS

	PAGE NO.
ABSTRACT	ii
NOMENCLATURE	1
1. Introduction	
1.1 Problem addressed	2
1.2 Related Literature	3
1.3 Scope of the Project	4
2. Approach Used for Determining	
2.1 Architectural Design	6
2.2 RTL Design(Verilog)	8
2.3 Toolchain Used	9
3. Results and Discussion	11
3.1 Simulation Results	11
3.2 Synthesis Results	13
4. Conclusions and Recommendations	15
5. References	17
6. Appendices	18

NOMENCLATURE

Abbreviation / Term	Full Form / Description
RISC	Reduced Instruction Set Computer
RISC-V	An open standard Instruction Set Architecture based on RISC principles
RV32I	RISC-V 32-bit base Integer instruction set
HDL	Hardware Description Language
RTL	Register Transfer Level – abstraction for digital design
ALU	Arithmetic Logic Unit – performs arithmetic and logical operations
SoC	System on Chip – integrates all components of a computer or system on a chip
Verilog	Hardware Description Language used for digital circuit design
Testbench	Simulation setup used to verify hardware module functionality
Yosys	Open-source logic synthesis tool for Verilog designs
GrayWolf	Placement tool used in digital design flow
QRouter	Open-source routing tool for standard cell layout
Magic	Open-source VLSI layout tool
KLayout	IC layout viewer and editor
GDSII	Graphic Data System II – standard format for IC layout data
Synthesizable	Capable of being translated into gate-level hardware

CHAPTER 1: INTRODUCTION

1.1 Problem Addressed

In today's digital age, the demand for processors is exponentially increasing with the growth of embedded systems, mobile devices, IoT, smart appliances, and custom computing platforms. These systems require efficient, low-power, and customizable processing units that can be integrated seamlessly with different types of hardware environments. Traditionally, processor development has relied on proprietary Instruction Set Architectures (ISAs) such as x86 and ARM. While these ISAs have served the industry for decades, they pose significant challenges, particularly in the context of education, research, and innovation.

One of the major problems associated with proprietary ISAs is the restriction imposed by licensing and intellectual property rights. Developing, modifying, or integrating such cores into custom designs is often associated with high costs, legal complexities, and limitations in accessibility.

Among its various subsets, the RV32I base integer instruction set is the most fundamental. It includes essential instructions required for arithmetic, logical operations, data movement, and basic control flow. This subset is highly suitable for educational purposes, introductory processor design, and lightweight embedded applications. Implementing the RV32I core provides foundational insight into how modern processors fetch, decode, and execute instructions.

The core problem that this project addresses is the absence of a simple, open, and pedagogically valuable RISC-V core that students and beginners can study, modify, and test on open-source toolchains. While high-performance cores exist, they are often overly complex for learning. This project attempts to fill this gap by creating a clean, understandable, and Verilog-based implementation of the RV32I core that can be synthesized, simulated using accessible EDA tools.

1.2 Related Literature

The RISC-V architecture, launched by the University of California, Berkeley, has sparked significant academic and industrial interest due to its open-source and modular nature. Since its inception, RISC-V has been embraced as a revolutionary step toward democratizing processor development, offering flexibility and openness that proprietary architectures fail to provide. The literature around RISC-V has grown exponentially over recent years, with multiple textbooks, whitepapers, journal articles, and conference proceedings addressing its capabilities, performance, and implementation strategies.

The foundational literature for RISC-V includes “The RISC-V Instruction Set Manual” by Andrew Waterman and Krste Asanović, which outlines the specifications of different subsets including RV32I, RV64I, and their extensions. This manual is considered the definitive source for understanding RISC-V's philosophy, syntax, and structure. Another vital resource is *Computer Organization and Design: RISC-V Edition* by David Patterson and John Hennessy, which provides a comprehensive look at processor design using RISC-V as the base ISA.

Open-source projects such as PicoRV32 and VexRiscv serve as real-world implementations of the RISC-V architecture. PicoRV32 is a lightweight, bit-serial RISC-V core written in Verilog. It is suitable for FPGAs with limited resources and serves as an ideal example for beginners. It supports the RV32I instruction set and is known for its simplicity and clarity of design. VexRiscv, on the other hand, is a high-performance, pipelined, and configurable RISC-V implementation written in SpinalHDL. It allows users to customize the processor architecture for various use cases ranging from low-power embedded applications to high-performance computing.

Several research efforts have focused on the comparative analysis of RISC-V with other ISAs such as ARM and MIPS. These studies consistently show that RISC-V offers comparable performance with added flexibility and significantly reduced development cost. Papers published in IEEE, ACM, and Elsevier have provided in-depth studies on

RISC-V-based SoC design, hardware/software co-design, power optimization, and verification techniques.

In the context of education, RISC-V has enabled a new wave of hands-on processor design courses. Universities around the world have started incorporating RISC-V into their computer architecture labs. The availability of low-cost FPGA boards, open-source toolchains (like Yosys, GTKWave, and Qflow), and HDL simulators has made it easier for students to implement, simulate, and synthesize their own RISC-V cores.

1.3 Scope of the Project

The scope of this internship project is to design, implement, verify, and synthesize a basic RISC-V processor core that supports the RV32I instruction subset using Verilog HDL. This instruction set architecture (ISA) forms the core foundation of the RISC-V family, focusing on basic integer operations. The reason for focusing on RV32I is that it provides a manageable subset of RISC-V that includes essential arithmetic, logical, memory access, and control flow instructions—sufficient for building and understanding a working processor without the added complexity of advanced features.

The project involves multiple stages of processor development: starting from high-level

architectural design, followed by RTL (Register Transfer Level) modeling, simulation for verification, synthesis for gate-level implementation, and optional physical design layout using open-source EDA tools. Each stage is aimed at building a functional processor capable of executing a predefined set of instructions correctly.

The scope includes the development of:

- A Program Counter (PC) to fetch sequential instructions from memory.
- An Instruction Memory module for storing machine-level RISC-V programs.
- An Instruction Decoder, Control Unit to generate control signals based on opcodes.
- A Register File to store temporary computation data.
- An Arithmetic Logic Unit (ALU) to perform basic arithmetic and logical

operations.

- A Data Memory module to support LW (load word) and SW (store word) instructions.

The RV32I instruction subset is chosen not only for its simplicity but also for its educational value, enabling students and entry-level engineers to understand the fundamentals of processor design. This project ensures that the processor can fetch, decode, and execute instructions such as ADD, SUB, AND, OR, XOR, LW, SW, BEQ, and JAL, which are foundational to any computing architecture.

The project is implemented entirely using Verilog HDL. Verification is done through simulation of testbenches using tools such as Icarus Verilog and GTKWave. These testbenches check the logical correctness of individual modules and their integration. Synthesis is performed using Yosys, an open-source synthesis tool, which generates gate-level netlists and reports area and timing estimates

Importantly, to make the project feasible within a short internship duration, the scope explicitly excludes complex architectural enhancements such as pipelining, out-of-order execution, interrupt handling, cache design, and virtual memory support. These can be considered for future work, as they significantly increase the complexity of the design.

By defining a clear scope focused on the RV32I core, this project aims to provide a balance between depth and manageability. It allows exploration of key architectural concepts such as datapath design, instruction execution, and control signal generation while enabling hands-on experience with industry-grade, open-source tools. Furthermore, this work sets the foundation for scaling to more advanced cores or integrating with peripheral modules in future system-on-chip (SoC) designs.

CHAPTER 2: APPROACH USED

2.1 Architectural Design

The architectural design of a processor forms the blueprint that defines how it interprets and executes instructions. In this project, we follow a classic, single-cycle architecture tailored specifically to support the RISC-V RV32I subset. The architecture has been chosen to ensure clarity, simplicity, and pedagogical effectiveness. It consists of fundamental components including the Program Counter (PC), Instruction Memory, Register File, Control Unit, Arithmetic Logic Unit (ALU), and Data Memory.

The **Program Counter (PC)** holds the address of the current instruction and is updated every cycle. In the case of control transfer instructions like branches and jumps, the PC is updated based on the immediate offset.

The **Instruction Memory (IMEM)** contains a predefined set of machine-level instructions encoded according to the RV32I format. These instructions are fetched sequentially unless modified by control flow operations like JAL or BEQ.

The **Instruction Decoder and Control Unit** interprets the fetched instruction and generates control signals that guide the operations of the processor. These control signals include ALU operation selection, memory read/write enable, register write enable, and multiplexer control for operand selection.

The **Register File** includes 32 general-purpose registers, each 32-bits wide. The register file supports reading two operands and writing one result in a single cycle. The source and destination register indices are extracted from the instruction fields.

The **Arithmetic Logic Unit (ALU)** executes arithmetic and logical instructions such as ADD, SUB, AND, OR, and XOR. The ALU operation is determined by the control unit based on the instruction's function code.

- A Data Memory module to support LW (load word) and SW (store word) instructions.

The RV32I instruction subset is chosen not only for its simplicity but also for its educational value, enabling students and entry-level engineers to understand the fundamentals of processor design. This project ensures that the processor can fetch, decode, and execute instructions such as ADD, SUB, AND, OR, XOR, LW, SW, BEQ, and JAL, which are foundational to any computing architecture.

The project is implemented entirely using Verilog HDL. Verification is done through simulation of testbenches using tools such as Icarus Verilog and GTKWave. These testbenches check the logical correctness of individual modules and their integration. Synthesis is performed using Yosys, an open-source synthesis tool, which generates gate-level netlists and reports area and timing estimates.

Importantly, to make the project feasible within a short internship duration, the scope explicitly excludes complex architectural enhancements such as pipelining, out-of-order execution, interrupt handling, cache design, and virtual memory support. These can be considered for future work, as they significantly increase the complexity of the design.

By defining a clear scope focused on the RV32I core, this project aims to provide a balance between depth and manageability. It allows exploration of key architectural concepts such as datapath design, instruction execution, and control signal generation while enabling hands-on experience with industry-grade, open-source tools. Furthermore, this work sets the foundation for scaling to more advanced cores or integrating with peripheral modules in future system-on-chip (SoC) designs.

2.2 RTL Design (Verilog)

The RTL (Register Transfer Level) design is the critical phase of converting architectural specifications into synthesizable Verilog code. The RTL design of the RISC-V RV32I processor in this project was modular and hierarchical, promoting reusability and clarity. Each component of the architecture such as the Program Counter, Register File, ALU, Control Unit, and Memory units was implemented as an individual Verilog module.

1. Program Counter (PC):

Implemented as a simple 32-bit register that updates every clock cycle. For regular instructions, it increments by 4 (byte-addressed). In case of branch or jump, it is updated based on the immediate offset.

2. Instruction Memory:

This module provides the instruction corresponding to the current PC. For simulation purposes, a simple array was used to store pre-defined 32-bit RISC-V instructions.

3. Control Unit:

The control unit was designed to decode the opcode and function codes of the instruction and generate the necessary control signals for the ALU, register write enable, memory access, and multiplexers. It distinguishes between instruction types such as R-type, I-type, S-type, B-type, and J-type and adapts the control logic accordingly.

4. ALU (Arithmetic Logic Unit):

The ALU performs arithmetic and logic operations like addition, subtraction, bitwise AND, OR, XOR, and comparisons. It accepts two 32-bit inputs and produces a 32-bit result based on the control signal. The ALU also sets a zero flag used in branch decision logic.

5. Register File:

This module implements 32 registers, each 32-bits wide. It supports two read ports and one write port. Read and write operations are determined by register addresses specified in the instruction fields. The design ensures that register x0

always returns zero, conforming to RISC-V ISA rules.

6. Immediate Generator:

Immediate values were extracted from instruction fields and extended to 32-bits based on the instruction type. This unit is essential for handling immediate arithmetic, memory addressing, and control flow instructions.

7. Data Memory:

A byte-addressable memory array was implemented to support load and store instructions (LW and SW).

8. Multiplexers and Adders:

Multiplexers were used extensively to select between operands (register or immediate), next PC values (sequential or branch), and ALU inputs. Simple adders were used for PC+4 and branch target calculations.

2.3 Toolchain Used

The development of the RISC-V RV32I core in this project was carried out entirely using open-source electronic design automation (EDA) tools. The use of these tools not only aligns with the philosophy of the open-source RISC-V initiative but also ensures accessibility, cost-effectiveness, and transparency for students, researchers, and professionals alike.

1. Verilog Editor (GVim/Vim):

Coding the processor was performed in GVim, a graphical version of Vim, known for its efficiency and versatility in handling HDL files. GVim allows for syntax highlighting, code navigation, and quick editing, making it suitable for hardware description languages such as Verilog.

2. Icarus Verilog:

Icarus Verilog is an open-source Verilog simulation and synthesis tool. It was used to simulate individual modules and the top-level processor design. The .v files were compiled using iverilog, and the generated VCD (Value Change Dump) files were used for waveform visualization. Icarus Verilog provided

support for synthesizable Verilog constructs, making it ideal for early-stage functional validation.

3. GTKWave:

GTKWave was used in conjunction with Icarus Verilog for waveform visualization. This graphical viewer reads `.vcd` files generated during simulation and displays signal transitions over time. It was crucial for debugging and validating instruction execution, control signals, memory operations, and ALU results at every clock cycle.

4. Yosys:

Yosys is an open-source framework for RTL synthesis. It takes Verilog source code as input and produces a gate-level netlist in formats such as BLIF or JSON. Yosys was used to analyze synthesis results including area usage and logic elements. It also allows constraint definitions and supports various output flows compatible with downstream tools.

5. Qflow:

Qflow is a complete digital synthesis and layout toolchain based on open-source tools. It includes Yosys for synthesis, Graywolf for placement, Qrouter for routing, and Magic for layout editing. Qflow was used to perform the physical design process, including logic synthesis, technology mapping, placement, and routing. The tool generates layout files compatible with Magic and produces GDSII files for fabrication.

6. Magic:

Magic is a layout tool that allows users to view, edit, and verify VLSI layouts. It was used to inspect the final physical layout of the synthesized RISC-V processor. Magic provides visualization for design rule checks (DRC) and layout versus schematic (LVS) checks. The output layout was exported as a GDSII file.

7. KLayout:

KLayout is a GDSII viewer used to inspect the final layout generated by Magic. It provides advanced navigation, measurement, and hierarchy tools that allow verification of placement and connectivity.

CHAPTER 3: RESULTS AND DISCUSSION

3.1 Simulation Results

Simulation is a crucial part of the digital design process as it allows verification of functionality before the design is committed to synthesis or fabrication. In this project, simulation was performed using Icarus Verilog, and the waveforms were analyzed using GTKWave. The simulation helped verify that each module of the RISC-V processor behaves as expected and that the full processor can execute instructions from the RV32I subset correctly.

The initial simulation phase involved unit-level testing. Each major component of the processor—Program Counter, Instruction Decoder, Register File, ALU, Data Memory, and Control Unit—was independently tested using custom Verilog testbenches. For example, the ALU module was tested with various combinations of operands and operation codes (ADD, SUB, AND, OR, XOR) to verify correctness.

Once the modules passed individual simulations, they were integrated into a top-level module representing the full processor datapath. A complete testbench was developed to load a sequence of RV32I instructions into the instruction memory. These instructions were designed to verify the processor's ability to:

- Perform arithmetic operations (ADD, SUB, AND, OR, etc.)
- Execute memory operations (LW, SW)
- Handle control flow (BEQ, JAL)

Simulation was run using the iverilog command, and the generated .vcd files were viewed in GTKWave. The waveforms clearly showed the instruction flow through the processor. For example, when an ADD instruction was executed, the waveforms reflected register fetch, ALU operation, and register write-back.

Similarly, for a LW instruction, the data memory access was visible.

In particular, one simulation test sequence included:

1. Writing values to registers x1 and x2
2. Performing an ADD operation to store the sum in x3
3. Storing the result from x3 to memory
4. Loading the result back into x4
5. Using BEQ to conditionally jump to another instruction

The waveforms confirmed the correct execution of each of these steps, thereby validating the end-to-end functionality of the RV32I core.

The RTL design was also tested for corner cases, such as:

- Zero register (x0) always remaining 0
- Invalid opcodes not affecting internal signals
- Branch taken vs not taken
- Negative immediate values handled properly

Overall, simulation provided the confidence that the design operates as intended and forms the basis for further steps such as synthesis and layout. The testbenches developed can also be reused for regression testing when making future enhancements like pipelining or adding new instruction support.

3.2 Synthesis Results

Synthesis is the process of converting a high-level hardware description, typically written in a Hardware Description Language (HDL) such as Verilog, into a gate-level representation using standard cells. This step is essential in bridging the gap between abstract design and physical realization. In this project, synthesis was performed using the open-source tool Yosys, which generated a netlist that was further used in placement and routing via Qflow.

The Verilog source files were passed into Yosys using a synthesis script. The script specified the top module, read all RTL files, and defined the technology library to be used. In our case, we used the osu035 standard cell library, which is compatible with Qflow and Magic. The command `yosys -s synth.ys` initiated the synthesis process.

Yosys performed various synthesis steps:

- **Parsing and Elaboration:** The Verilog files were read and parsed into internal representations.
- **Optimization:** Redundant logic was removed, constant propagation was performed, and signal dependencies were simplified.
- **Technology Mapping:** The logic expressions were mapped onto gates available in the osu035 cell library.
- After synthesis, Yosys generated a gate-level netlist, typically in BLIF format. The synthesis statistics provided valuable insights such as
 - **Number of logic gates used**
 - **Number of flip-flops and latches**
 - **Fanout of internal signals**
 - **Critical path information**

The Qflow synthesis report generated the following important files:

- **.blif:** BLIF netlist file from Yosys
- **.sel.v:** Verilog file with synthesized logic

- **.area.rpt**: Report with area and gate usage

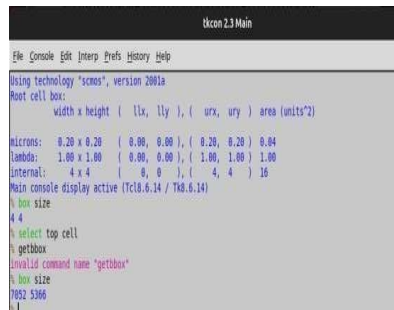
The synthesis also provided D flip-flop and combinational logic mapping, highlighting how sequential logic like the Program Counter or Register File is built from primitives.

- **Reset logic** was preserved.
- **Control signals** were logically simplified.
- **ALU and memory control** paths were implemented as expected.

Yosys also provided a graphical view of the synthesized netlist using the `show` command. This helped visualize connectivity between logic gates, improving debugging.

Finally, the synthesized design was simulated again using a post-synthesis netlist to verify that the functionality matched the RTL simulation results. The output `.vcd` file from this simulation was compared against the original to confirm correctness.

The layout of the RV32I ALU module was analyzed using the Magic VLSI tool. The bounding box measured **7052 λ \times 5366 λ** , which corresponds to **1,057.8 μm \times 805 μm** in the Sky130 technology ($\lambda = 0.15 \mu\text{m}$). This results in a total layout area of approximately **851,529 μm^2 or 0.8515 mm^2** .



```

Ukon 2.3 Main
File Console Edit Interp Prefs History Help
Using technology "scmos", version 2001a
Root cell box:
  width x height ( lx, lly ), ( urx, ury ) area (units^2)
microns:  0.20 x 0.20  ( 0.00, 0.00 ), ( 0.20, 0.20 ) 0.04
lambda:   1.00 x 1.00  ( 0.00, 0.00 ), ( 1.00, 1.00 ) 1.00
internal:  4 x 4      ( 0, 0 ), ( 4, 4 ) 16
Main console display active (Tcl8.6.14 / Tk8.6.14)
% box size
4 4
% select top cell
% getbbox
Invalid command name "getbbox"
% box size
7052 5366

```

CHAPTER 4: CONCLUSIONS AND RECOMMENDATIONS

The project titled “Design and Implementation of RISC-V Core (RV32I Subset)” has been a comprehensive and enriching experience in the domain of digital system design and open- source hardware. The journey from understanding the RISC-V instruction set to developing a fully functional and synthesizable core using Verilog HDL encompassed several critical phases including architectural planning, RTL coding, simulation, synthesis, placement, routing, and layout generation.

The successful implementation and simulation of the RV32I processor core highlight the potential of RISC-V as a versatile and scalable instruction set architecture. The processor was able to correctly execute arithmetic, logical, memory access, and control flow instructions defined by the RV32I base ISA, validating the correctness of the implementation.

One of the key achievements of this project is the use of fully open-source EDA tools such as Icarus Verilog, GTKWave, Yosys, Qflow, Magic, and KLayout. These tools provided a cost- effective and accessible environment for processor design and VLSI backend flow, demonstrating that robust and industry-relevant digital designs can be realized without proprietary toolchains. The integration of these tools not only provided functional verification and gate-level synthesis but also enabled physical layout and GDSII generation, simulating the complete ASIC development cycle.

This project also reinforced the importance of rigorous testing and verification at each design stage. The step-by-step simulation of individual modules, followed by integrated system testing, ensured that the design was robust and free of major functional bugs. The debugging process through GTKWave provided deep insights into how signals propagate and interact across clock cycles, which is crucial for understanding timing issues in real-world hardware.

From an educational and professional standpoint, this project provided hands-on experience in digital design, RTL development, simulation, synthesis, layout design, and verification using industry-standard methodologies. It bridges the academic understanding of computer architecture with practical implementation, making it especially valuable for roles in hardware design, embedded systems, and SoC development.

Recommendations:

1. **Pipelining for Performance:** Future work may involve extending the design to a pipelined architecture. This would improve instruction throughput and better represent modern processor designs.
2. **Toolchain Automation:** Shell scripts or Makefiles can be created to automate the entire process from simulation to GDSII generation, improving reproducibility and reducing manual errors.
3. **FPGA Implementation:** The synthesized netlist can be adapted and mapped onto an FPGA for real-time testing, providing a bridge between simulation and hardware prototyping.
4. **ISA Extension:** Support for other RISC-V subsets like RV32M (multiplication/division) or RV64I (64-bit) can be considered to extend the functionality of the processor.

CHAPTER 5: REFERENCES

1. A. Waterman, Y. Lee, D. Patterson, and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, Version 2.1, EECS Department, University of California, Berkeley, May 2016, pp. 1–129.
2. S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed., Pearson Education, 2003, ch. 4–7.3. N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed., Pearson/Addison-Wesley, 2011, ch. 9–12.
3. J. Bhasker, *A Verilog HDL Primer*, 3rd ed., Star Galaxy Publishing, 2005, ch. 3, pp. 45–97.
4. W. Wolf, *FPGA-Based System Design*, Prentice Hall, 2004, ch. 5, pp. 145–186.

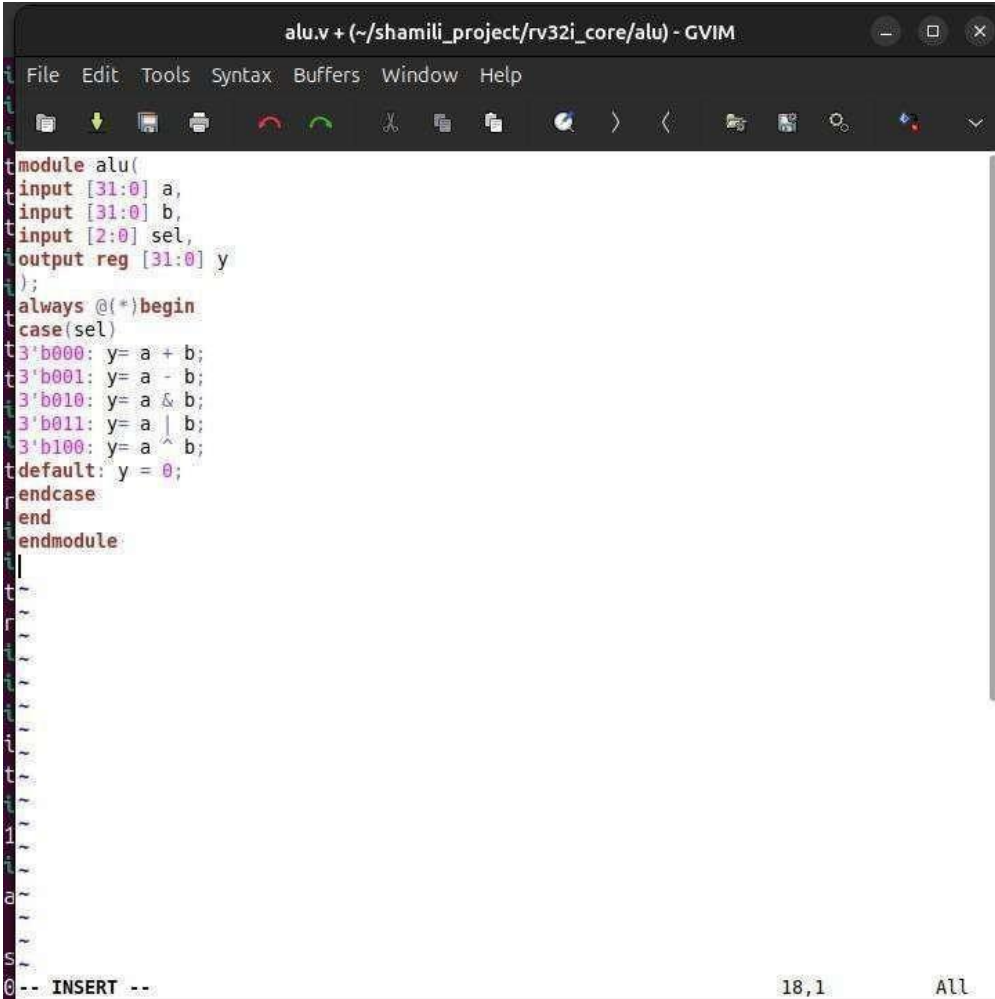
CHAPTER 6: APPENDICES

Below are the relevant images, code snippets, and associated theoretical descriptions for key stages of the RISC-V RV32I core implementation:

6.1 RTL Design File – alu.v

The RTL (Register Transfer Level) design file defines the behavior of the Arithmetic Logic Unit (ALU) in Verilog HDL. It is a critical block that performs computations in response to instruction decode logic in a processor.

- Description: Performs operations based on control signals. The ALU supports logical, arithmetic, and comparative functions.



```
alu.v + (~/shamili_project/rv32i_core/alu) - GVIM
File Edit Tools Syntax Buffers Window Help
[Icons]
module alu(
input [31:0] a,
input [31:0] b,
input [2:0] sel,
output reg [31:0] y
);
always @(*)begin
case(sel)
3'b000: y= a + b;
3'b001: y= a - b;
3'b010: y= a & b;
3'b011: y= a | b;
3'b100: y= a ^ b;
default: y = 0;
endcase
end
endmodule
-- INSERT -- 18,1 ALL
```

6.2 Verilog Testbench – alu_tb.v

A testbench simulates the functional behavior of a design module by applying stimulus and monitoring output. It is essential for validating the correctness of each RTL block through comprehensive scenarios.

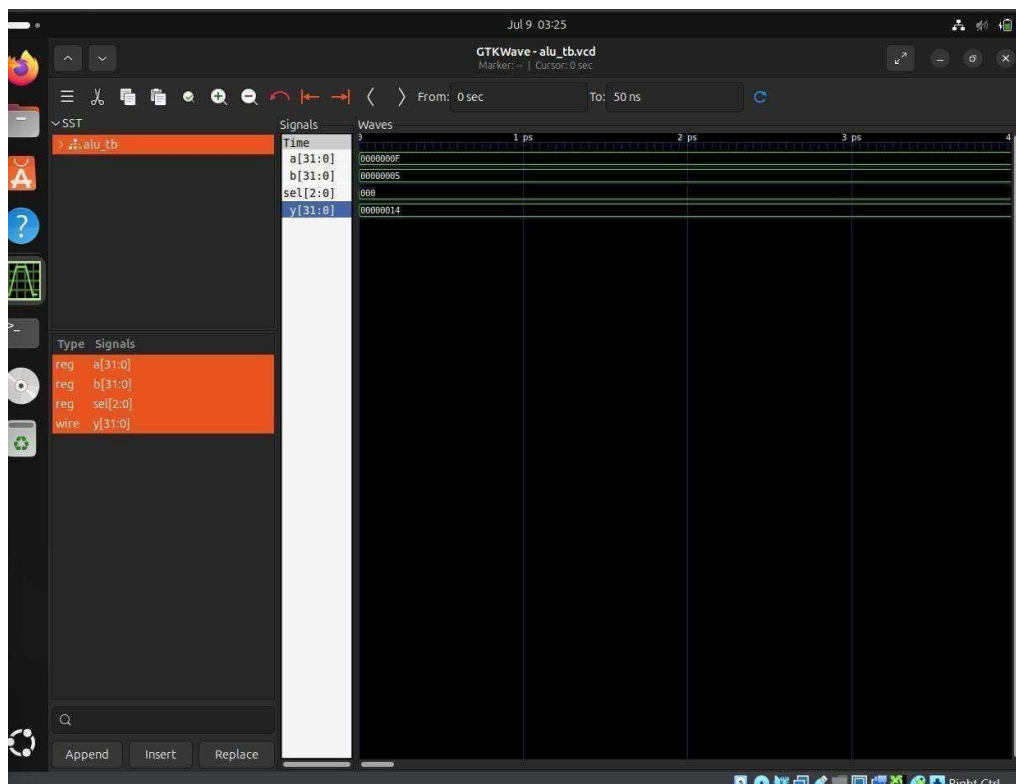
- Description: alu_tb.v contains test cases for all ALU operations including addition, subtraction, AND, OR, and comparison.

```
alu_tb.v (~shamili_project/rv32i_core/alu) - GVIM
File Edit Tools Syntax Buffers Window Help
timescale 1ns / 1ps
module alu_tb;
//inputs
reg [31:0] a;
reg [31:0] b;
reg [2:0] sel;
//output
wire [31:0] y;
//instantiate the ALU
alu uut (
    .a(a),
    .b(b),
    .sel(sel),
    .y(y)
);
initial begin
    $dumpfile("alu_tb.vcd");
    $dumpvars(0, alu_tb);
    //test case 1: addition
    a = 32'd15; b = 32'd5; sel = 3'b000; #10;
    //test case 2: subtraction
    a = 32'd15; b = 32'd7; sel = 3'b001; #10;
    //test case 3: and
    a = 32'hFFF0000; b = 32'h00FF00FF; sel = 3'b010; #10;
    //test case 4: or
    a = 32'hAAA0000; b = 32'h5550000; sel = 3'b011; #10;
    //test case 5: xor
    a = 32'hFFFFFFF; b = 32'h0000FFFF; sel = 3'b100; #10;
    //end simulation
    $finish;
end
endmodule
~
~
"alu_tb.v" 33L, 638B 9,13 All
```

6.3 GTKWave Viewer (Simulation)

GTKWave is a digital waveform viewer used for analyzing simulation output in VCD (Value Change Dump) format. It helps observe the behavior of signals over time, detect logic errors, and verify correct timing and control signal interactions.

- Tool Used: GTKWave
- Description: After simulation with Icarus Verilog, GTKWave displayed the VCD waveform.



6.4 Synthesis Step

Synthesis is a fundamental process in digital design where high-level behavioral RTL code is translated into a technology-specific gate-level representation. It ensures that the logic is realizable in silicon and complies with the constraints of the chosen standard cell library.

The synthesis stage also performs optimizations to reduce area, power, and delay.

- Tool Used: Yosys
- Description: The Verilog RTL code was passed to Yosys which optimized and mapped it to standard cells.

```
shamili@shamili: ~/shamili_project/rv32i_core/alu
shamili@shamili: ~/batch7 x shamili@shamili: ~/shamili_project/rv32i_core...
cellrex = re.compile('[ \t]*cell[ \t]*\(((^)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:679: SyntaxWarning: invalid escape
sequence '\('
pinrex = re.compile('[ \t]*pin[ \t]*\(((^)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:680: SyntaxWarning: invalid escape
sequence '\('
busrex = re.compile('[ \t]*bus[ \t]*\(((^)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:681: SyntaxWarning: invalid escape
sequence '\('
lat1rex = re.compile('[ \t]*latch[ \t]*\(((^)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:682: SyntaxWarning: invalid escape
sequence '\('
lat2rex = re.compile('[ \t]*latch[ \t]*\(((^, \t)+)[ \t]*,[ \t]*\(((^),)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:683: SyntaxWarning: invalid escape
sequence '\('
ff1rex = re.compile('[ \t]*ff[ \t]*\(((^)+)\)')
/usr/local/share/qflow/scripts/spi2xspice.py:684: SyntaxWarning: invalid escape
sequence '\('
ff2rex = re.compile('[ \t]*ff[ \t]*\(((^, \t)+)[ \t]*,[ \t]*\(((^),)+)\)')
Reading liberty netlist /usr/local/share/qflow/tech/osu035/osu035_stdcells.lib
Reading spice netlist alu.spc
Writing xspice netlist alu.xspice
Writing xspice file
Done.
shamili@shamili:~/shamili_project/rv32i_core/alu$
```

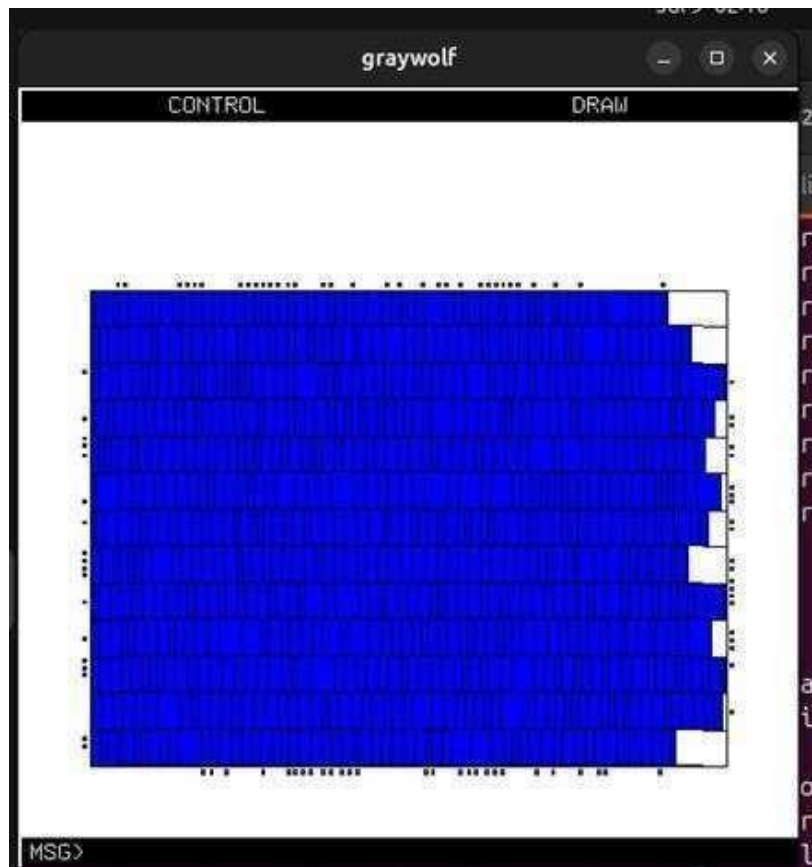


synth.log

6.5 Placement Step

Placement arranges logic cells on a predefined floorplan area so that the netlist connections can be routed efficiently. It directly affects signal delay, power consumption, and routing complexity. An optimized placement reduces overall wirelength and improves performance.

- Tool Used: GrayWolf (via Qflow)
- Description: GrayWolf arranges cells based on connectivity to minimize wire length.



place.log

6.6 Routing Step

Routing establishes the physical connections between standard cells placed during the placement stage. It determines the actual paths of signals using metal layers and vias. Good routing ensures signal integrity, meets timing constraints, and avoids congestion and DRC errors.

- Tool Used: QRouter
- Description: The router creates metal wires to connect all the logic gates as per the netlist.

```
shamili@shamili: ~/shamili_project/rv32i_core/alu
shamili@shamili: ~/batch7
*** Running stage3 routing with defaults, 2nd round
Nets remaining: 800
Nets remaining: 700
Nets remaining: 600
Nets remaining: 500
Nets remaining: 400
Nets remaining: 300
Nets remaining: 200
Nets remaining: 90
Nets remaining: 80
Nets remaining: 70
Nets remaining: 50
Nets remaining: 30
Nets remaining: 20
Nets remaining: 10
Nets remaining: 9
Nets remaining: 8
Nets remaining: 6
Nets remaining: 5
Nets remaining: 3
Progress: Stage 3 total routes completed: 4999
No failed routes!
*** Writing DEF file alu_route.def
Final: No failed routes!
*** Writing RC file alu_route.rc
Running annotate.tcl antenna.out /home/shamili/shamili_project/rv32i_core/alu/alu.rtno
pwr.v
/home/shamili/shamili_project/rv32i_core/alu/alu.spc /home/shamili/shamili_project/rv
32i_core/alu/alu.rtnopwr.anno.v
/home/shamili/shamili_project/rv32i_core/alu/alu.anno.spc /usr/local/share/qflow/tec
h/osu035/osu035_stdcells.sp /home/shamili/shamili_project/rv32i_core/alu/alu_powergroun
d
Running annotate.tcl
Found cell FILL pinlist vdd gnd
Done with annotate.tcl
shamili@shamili: ~/shamili_project/rv32i_core/alu$
```



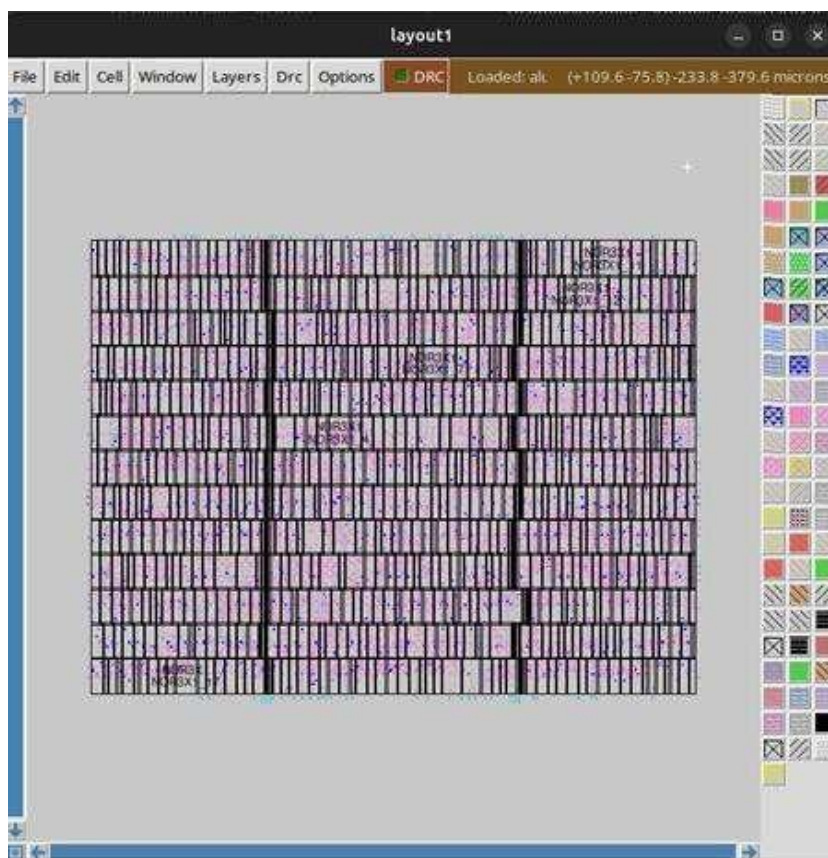
route.log

6.7 Magic Layout

Magic is a widely used open-source VLSI layout editor and viewer. It allows designers to inspect and validate the physical layout of the chip. Magic supports design rule checks (DRC), layout versus schematic checks (LVS), and GDSII generation, making it integral to the physical design flow.

- Tool Used: Magic VLSI

Description: Layout created from synthesis, placement, and routing is viewed and checked for DRC errors



6.8 GDSII File Generation

GDSII (Graphic Data System II) is the standard format used by the semiconductor industry to describe the layout of integrated circuits. It contains geometric shapes, layer information, and design hierarchy required for photomask creation in chip fabrication.

- Tool Used: Magic or Qflow (via Magic)
- Description: GDSII file was generated from the final layout to simulate fabrication data.

```

shamili@shamili: ~/shamili_project/rv32i_core/alu
shamili@shamili: ~/batch?
shamili@shamili: ~/shamili_project/rv32i_core/alu
Moving label "Y" from space to metal1 in cell XNOR2X1.
Reading "LATCH".
Moving label "D" from space to metal1 in cell LATCH.
Moving label "Q" from space to metal1 in cell LATCH.
Moving label "gnd" from space to metal1 in cell LATCH.
Moving label "vdd" from space to metal1 in cell LATCH.
Moving label "CLK" from space to metal1 in cell LATCH.
Reading "DFFSR".
Moving label "gnd" from space to metal1 in cell DFFSR.
Moving label "vdd" from space to metal1 in cell DFFSR.
Moving label "D" from space to metal1 in cell DFFSR.
Moving label "S" from space to metal1 in cell DFFSR.
Moving label "R" from space to metal1 in cell DFFSR.
Moving label "Q" from space to metal1 in cell DFFSR.
Moving label "CLK" from space to metal1 in cell DFFSR.
Reading "CLKBUF1".
Moving label "A" from space to metal1 in cell CLKBUF1.
Moving label "vdd" from space to metal1 in cell CLKBUF1.
Moving label "gnd" from space to metal1 in cell CLKBUF1.
Moving label "Y" from space to metal1 in cell CLKBUF1.
Reading "CLKBUF2".
Moving label "vdd" from space to metal1 in cell CLKBUF2.
Moving label "gnd" from space to metal1 in cell CLKBUF2.
Moving label "A" from space to metal1 in cell CLKBUF2.
Moving label "Y" from space to metal1 in cell CLKBUF2.
Reading "CLKBUF3".
Moving label "gnd" from space to metal1 in cell CLKBUF3.
Moving label "vdd" from space to metal1 in cell CLKBUF3.
Moving label "A" from space to metal1 in cell CLKBUF3.
Moving label "Y" from space to metal1 in cell CLKBUF3.
Reading "iit_stdcells".
Scaled magic input cell alu geometry by factor of 2
Processing timestamp mismatches: OAI21X1, INVX1, NOR2X1, NAND3X1, NAND2X1, INVX2, AND2X
2, NOR3X1, BUF2, FILL, OR2X2, AOI21X1, XNOR2X1, AOI22X1, BUF4, OAI22X1, INVX8, XOR2X1
shamili@shamili: ~/shamili_project/rv32i_core/alu$
    
```

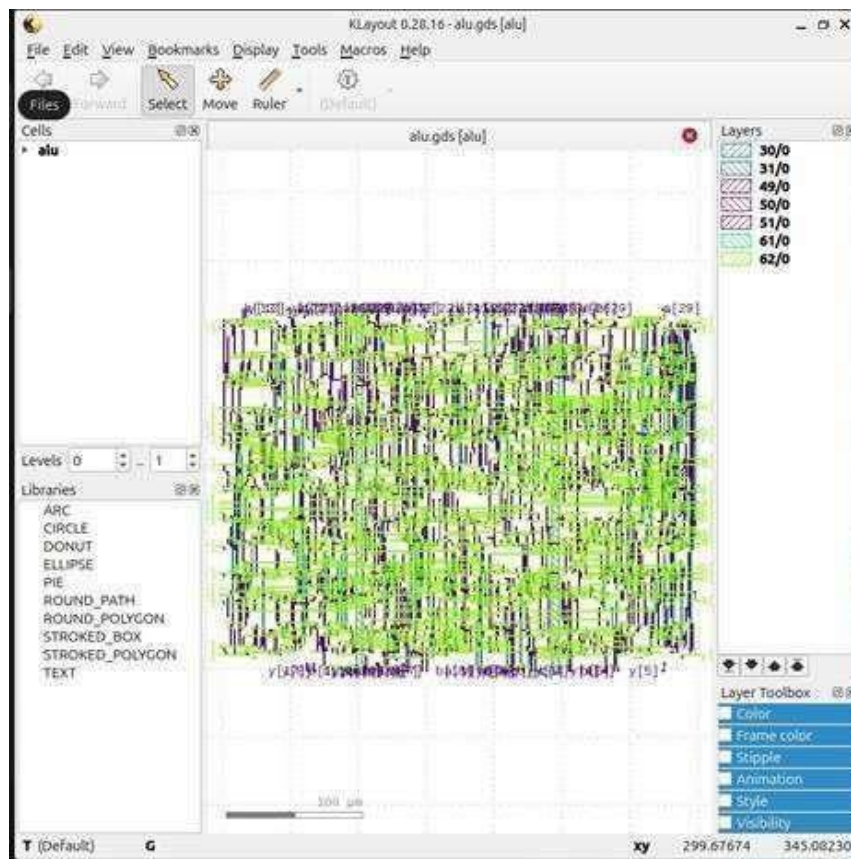


qflow.log

6.9 KLayout Viewer

KLayout is a high-performance viewer and editor for GDSII and OASIS files. It allows precise inspection of the layout geometry, hierarchy, and layer assignments. Designers use it to verify and validate final layouts before tape-out.

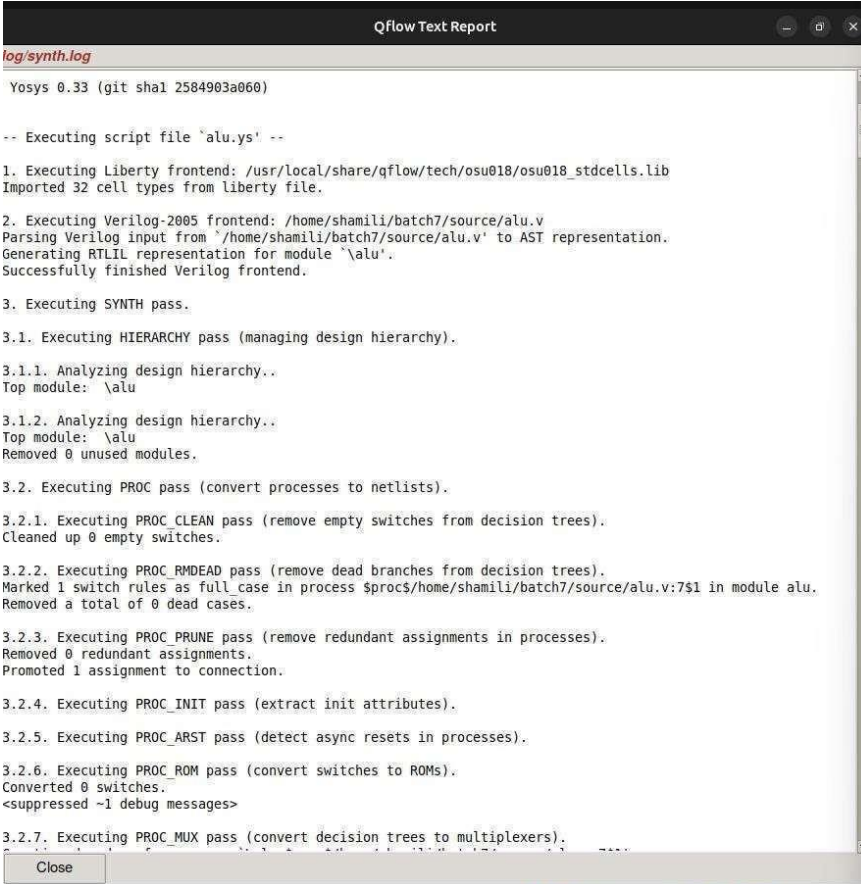
- Tool Used: KLayout
- Description: The generated GDSII was loaded into KLayout for inspection and verification.



6.10 Qflow GUI Screenshot

Qflow GUI provides a graphical interface for managing the digital design flow. It enables synthesis, placement, routing, DRC, and LVS from a centralized interface, simplifying the execution of the VLSI toolchain.

- Description: The Qflow GUI was used to run synthesis, placement, and routing steps in an organized and visual manner.



The screenshot shows a window titled "Qflow Text Report" with a file named "log/synth.log". The log content is as follows:

```

Yosys 0.33 (git sha1 2584903a060)

-- Executing script file 'alu.ys' --

1. Executing Liberty frontend: /usr/local/share/qflow/tech/osu018/osu018_stdcells.lib
Imported 32 cell types from liberty file.

2. Executing Verilog-2005 frontend: /home/shamili/batch7/source/alu.v
Parsing Verilog input from '/home/shamili/batch7/source/alu.v' to AST representation.
Generating RTLIL representation for module 'alu'.
Successfully finished Verilog frontend.

3. Executing SYNTH pass.

3.1. Executing HIERARCHY pass (managing design hierarchy).

3.1.1. Analyzing design hierarchy..
Top module: 'alu'

3.1.2. Analyzing design hierarchy..
Top module: 'alu'
Removed 0 unused modules.

3.2. Executing PROC pass (convert processes to netlists).

3.2.1. Executing PROC_CLEAN pass (remove empty switches from decision trees).
Cleaned up 0 empty switches.

3.2.2. Executing PROC_RMDEAD pass (remove dead branches from decision trees).
Marked 1 switch rules as full case in process $proc$/home/shamili/batch7/source/alu.v:7$1 in module alu.
Removed a total of 0 dead cases.

3.2.3. Executing PROC_PRUNE pass (remove redundant assignments in processes).
Removed 0 redundant assignments.
Promoted 1 assignment to connection.

3.2.4. Executing PROC_INIT pass (extract init attributes).

3.2.5. Executing PROC_ARST pass (detect async resets in processes).

3.2.6. Executing PROC_ROM pass (convert switches to ROMs).
Converted 0 switches.
<suppressed ~1 debug messages>

3.2.7. Executing PROC_MUX pass (convert decision trees to multiplexers).
    
```

At the bottom of the window, there is a "Close" button.