Student Name: Rami Omer
Student ID: 215855620
Course: EECS 4612 – Digital VLSI
Instructor: Dr. Amir M. Sodagar
TA: Mohsen Namavar
Term Project: Design of a 32-Bit ALU
Date: April 6th, 2025

1. Abstract / Summary

This project addresses the challenge of designing a scalable and functional Arithmetic Logic Unit (ALU) capable of performing arithmetic, logic, and shift operations at both the bit and word levels. The objective was to develop a modular and hierarchical digital system, beginning with a fully functional 1-bit ALU and extending it to a complete 32-bit design. The final goal was to implement, verify, and lay out the design using Cadence tools, and export a fabrication-ready GDSII file.

The solution involved designing a 1-bit ALU supporting arithmetic (addition, subtraction, increment, decrement), logic (AND, OR, XOR, NOT), and shift (left and right) operations. This unit was verified using structural Verilog and then expanded into two 32-bit ALU variants: (1) a structurally connected modular design and (2) a behavioral design described at the RTL level. Both versions underwent thorough functional verification and physical implementation including synthesis, place-and-route, DRC, and LVS.

Post-layout timing analysis revealed that the behavioral design achieved a maximum frequency of approximately 16.39 MHz, outperforming the modular design, which reached around 11.61 MHz. In terms of layout area, the behavioral ALU also proved more efficient, occupying approximately 975.726 μm^2, compared to the 1451.106 μm^2 area consumed by the modular design. These differences were attributed to the cleaner wiring, reduced logic depth, and more optimized synthesis achievable in behavioral RTL.

Based on this performance and area comparison, the behavioral ALU design was selected for final chip implementation. It was successfully instantiated into a top-level layout and integrated with a pre-designed pad frame. All necessary VDD, VSS, and signal pad connections were carefully routed, using appropriate metal layers and vias to ensure design rule compliance and physical connectivity.

In summary, the project achieved its core goals: creating a hierarchical ALU system from the 1-bit level to 32-bit, validating functionality and timing, and producing a GDSII layout suitable for fabrication. The successful integration of the behavioral ALU with a full pad frame underscores the project's completeness and industry-relevant design workflow.

2. Introduction

Digital systems form the backbone of modern computing, and at the heart of nearly every digital processor lies the Arithmetic Logic Unit (ALU). The ALU is a fundamental building block responsible for executing arithmetic and logic operations—tasks that are central to the functionality of microprocessors, embedded systems, and custom ASIC designs. As such, the design, verification, and physical implementation of a high-performance, scalable ALU represent a cornerstone in digital circuit design and system-on-chip (SoC) development.

This project focuses on the structured design and physical realization of a 32-bit ALU, built hierarchically from a reusable 1-bit slice. The approach adopted promotes modularity, reusability, and design scalability, allowing for flexible expansion or adaptation in future integrated circuit (IC) applications. The 1-bit ALU was designed to support a rich set of operations, including:

- Arithmetic operations: Transfer, Addition, Addition with Carry, subtraction, Subtraction with Borrow, increment, and decrement.
- Logic operations: AND, OR, XOR, and NOT.
- Shift operations: Logical shift left and logical shift right.

Once verified, this unit was instantiated 32 times to form a complete 32-bit ALU using both structural and behavioral design methodologies. Each version was functionally verified using self-checking Verilog testbenches to confirm correct outputs across all supported operation modes. Emphasis was placed on design verification, synthesis, and physical layout, using the Cadence Virtuoso and Innovus toolchains, ensuring that the final chip layout adheres to fabrication design rules.

In addition to logical correctness, key performance metrics such as layout area, maximum operating frequency, and design rule compliance were evaluated. These metrics informed the selection of the behavioral design for final chip integration, owing to its lower area footprint and higher achievable frequency post-route.

The final objective of the project was to integrate the verified ALU core into a pad frame layout, completing a full chip-level implementation. This required careful pin-to-pad assignment, metal layer routing, and via planning to ensure robust power delivery and I/O connectivity. The complete layout was exported in GDSII format, making the design ready for fabrication.

This project not only demonstrates mastery of hierarchical digital design and verification but also highlights the practical challenges of physical implementation, such as meeting design constraints, managing layout complexity, and interfacing with standardized I/O pads. Ultimately, the project prepares students for industry-grade VLSI workflows and provides insight into the end-to-end chip development lifecycle.

3. Project Objectives

This project is divided into two main parts. The first part focuses on designing, simulating, and implementing a 1-bit Arithmetic Logic Unit (ALU) using a modular, structural approach. The expectation is to verify the ALU's functionality, synthesize the design, and perform physical implementation using Place & Route (PnR) tools in Cadence Innovus.

The second part scales the ALU to a 32-bit version using both modular and behavioral design styles. The designs are simulated, synthesized, and compared in terms of area and speed. The more efficient design is selected for final chip integration, which involves connecting it to a pre-designed pad frame, generating a

complete layout, and exporting the chip in GDS format. Final deliverables include simulation waveforms, design files, synthesis reports, area/speed analysis, and the finalized GDSII file.

4. Design Description, Simulation, and Verification

This section is divided into two parts. The first part focuses on the structural design of the 1-bit ALU, including the design and verification of each submodule that composes it. The second part presents the design and simulation of the full 32-bit ALU implemented using two approaches: a modular instantiation of 1-bit ALUs and a behavioral model.

Part 1: 1-bit ALU Structural Design

The 1-bit ALU was developed using a modular and hierarchical approach. The submodules involved in this design were each designed structurally, verified individually, and then integrated to form the complete 1-bit ALU.

4.1 Submodule Designs

To build a functionally complete 1-bit ALU, each of its constituent submodules must be independently designed, simulated, and verified before integration. The three critical components that make up the ALU architecture are the 4-to-1 multiplexer, the arithmetic unit, and the logic unit. The following sections present the design methodology and simulation results for each submodule, starting with the 4-to-1 multiplexer.

4.1.1 4-to-1 Multiplexer

The 4-to-1 multiplexer is a foundational submodule in our ALU architecture, enabling the selection between multiple data inputs based on control signals. Its structural design and strategic placement allow for compact, flexible operation across both the arithmetic and logical domains of the ALU.

4-to-1 Multiplexer Design Description

The 4-to-1 multiplexer used in my design is implemented using basic logic gates (AND, OR, NOT), making it compatible with fully structural hardware design practices. A 4-to-1 multiplexer is a combinational circuit that selects one of four input signals (A, B, C, or D) and routes it to a single output (Y) based on the values of two control signals (S1 and S0). The selection logic is governed by the truth table shown below. Each row in **Table 1** corresponds to a unique combination of the select lines, determining which input is passed through to the output.

S1	S0	Input X	Output Y
0	0	A	A
0	1	B	B
1	0	C	C
1	1	D	D

Table 1: Truth Table for 4-to-1 Multiplexer
This table illustrates the selection behavior of a 4-to-1 multiplexer, where the output Y reflects the input (A, B, C, or D) selected by the control signals S1 and S0.

The Boolean expression representing the output of this multiplexer is:

Y = (S1′ & S0′ AND A) OR (S1′ AND S0 AND B) OR (S1 AND S0′ AND C) OR (S1 AND S0 AND D)

This structure enables precise control over signal flow, forming a critical part of the ALU's arithmetic logic routing.

Primary Usage of the 4-to-1 Multiplexer in the ALU Architecture:

1. **Final Output Selection in the 1-bit ALU:** At the final stage of the ALU data path, a 4-to-1 multiplexer is used to select the result of one of four functional operations: arithmetic output (Di), logic output (Ei), shift right, and shift left. **Figure 1** illustrates that this decision is made based on the two most significant bits of the 4-bit control signal (S[3:2]), effectively allowing the ALU to switch between operation modes.

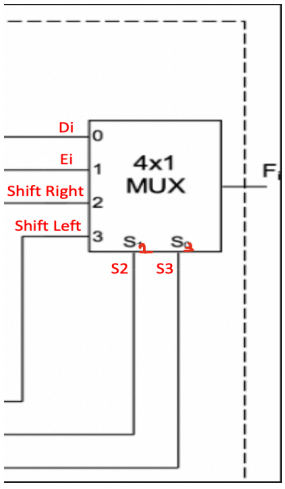


Figure 1: *4-to-1 Multiplexer for Final Output Selection in the 1-Bit ALU*

2. **Within the Arithmetic Unit:** A second use of the 4-to-1 multiplexer appears inside the **Arithmetic Unit**, where it is employed to select the second operand (Bi) for the full adder. The multiplexer chooses among four predefined inputs:
   a. 0 (for transfer or increment operations),
   b. Bi (for addition),
   c. ~Bi (for subtraction),
   d. 1 (for decrement or special cases).

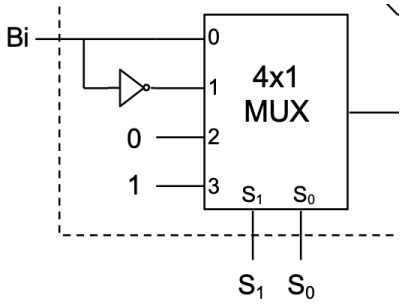


Figure 2: *4-to-1 Multiplexer for Operand Selection within the Arithmetic Unit*

As shown in **Figure 2**, this selection is driven by the lower two bits of the sel signal (S[1:0]), making the arithmetic unit flexible and capable of supporting multiple arithmetic operations with a single hardware implementation.

3. **Within the Logic Unit**

The logic unit also employs a 4-to-1 multiplexer to determine which logic operation result to forward as the logic output (Ei). The four logic expressions—AND (Ai & Bi), OR (Ai | Bi), XOR (Ai ^ Bi), and NOT (~Ai)—are precomputed, and the multiplexer, as depicted in **Figure 3,** selects among them based on S[1:0]. This setup enables compact, modular logic selection with minimal structural complexity.

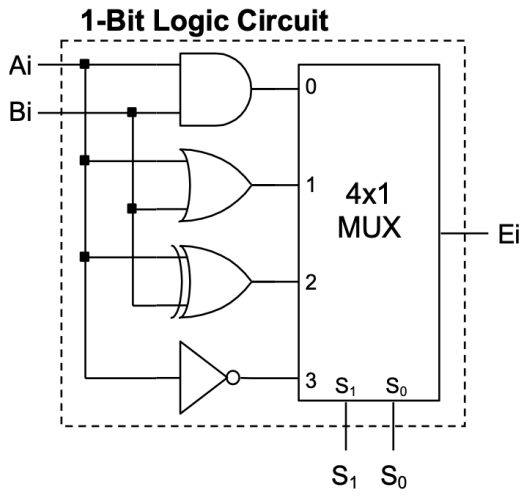


Figure 3: *4-to-1 Multiplexer in the Logic Unit of the 1-Bit ALU*

.

Multiplexer Inputs and Outputs in the Arithmetic Unit, Logic Unit and the 1-bit ALU
Inputs (Data):

- D0 = Arithmetic result (Di), or constant logic 0 (inside arithmetic unit), or the AND operation (inside the logic unit)
- D1 = Logic result (Ei), or operand Bi (inside arithmetic unit), or the OR operation (inside the logic unit)
- D2 = Shift right result or ~Bi (bitwise negation inside arithmetic unit) or XOR operation (inside the logic unit)
- D3 = Shift left result or constant logic 1 or the NOT operation (inside the logic unit)

Inputs (Control) S[3:0]:

- In the ALU output selection, S[3:2] is used to select the operation mode
- In the arithmetic unit, S[1:0] is used to control operand selection.
- In the logic unit, S[1:0] is used to select the logic operation.

Output: The selected signal passed to either the ALU output (Fi) or to the full adder as the second operand (Bi logic) or the logic unit output (Ei).

4-to-1 Multiplexer Simulation and Verification

To validate the correctness and structural integrity of the 4-to-1 multiplexer, an exhaustive functional verification was conducted using Xcelium, covering every possible scenario for a 2-bit select signal (sel[1:0]) in combination with all meaningful variations of the four data inputs: a, b, c, and d. The verification testbench was carefully designed to not only ensure logical correctness under normal conditions but also to strategically probe edge cases, such as all inputs being zero or one, and selective toggling of individual bits.

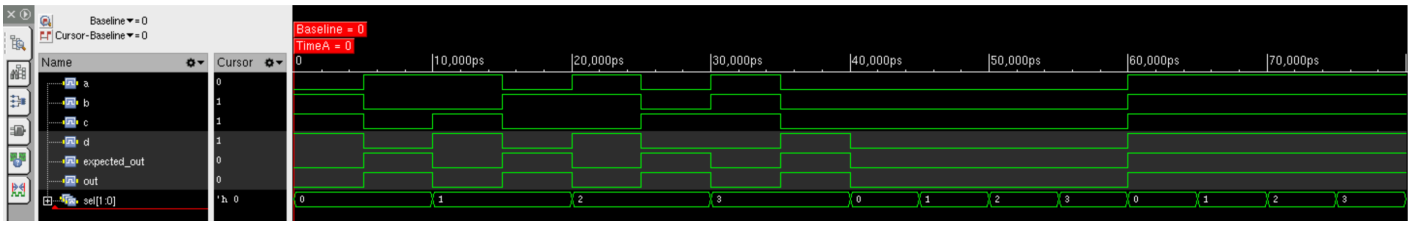


***Figure 4**: Timing waveform illustrating the functional simulation of the 4-to-1 multiplexer.*

Figure 4 shows that ach test case dynamically compared the expected output (expected_out) against the observed output (out), logging results with clear pass/fail criteria. All combinations were verified, with each case concluding with a [PASS] message confirming functional alignment.

Key Strategic Test Scenarios:

1. **Basic Routing Check**:
   a=1, b=0, c=0, d=0 with sel=2'b00 → Expected = a = 1 → Output = 1
   Validates correct routing of input a when sel=00.
2. **Middle Selector Tests**:
   a=0, b=1, c=0, d=0 with sel=2'b01 → Expected = b = 1 → Output = 1
   a=0, b=0, c=1, d=0 with sel=2'b10 → Expected = c = 1 → Output = 1
   Confirms that the MUX correctly handles middle control paths.
3. **Final Selector Test**:
   a=0, b=0, c=0, d=1 with sel=2'b11 → Expected = d = 1 → Output = 1
   Demonstrates that the most significant select case operates correctly.
4. **All Inputs Zero**:
   a=0, b=0, c=0, d=0 with all select lines → Output always 0
   Validates correct fallback behavior when no active input is high.
5. **All Inputs One**:
   a=1, b=1, c=1, d=1 with all select lines → Output always 1
   Ensures uniformity when all inputs are high and confirms select logic isolates the correct path without interference.
6. **Inverted Selector Edge Case**:
   a=1, b=1, c=0, d=0 with sel=2'b10 → Expected = c = 0 → Output = 0
   Proves the selector prioritizes correct path over value similarity.

These tests cover both the typical and critical edge conditions expected in digital circuits involving multiplexers. The waveform simulation, shown earlier, visually confirms the output transitions precisely at the moments expected based on selector values. Rising and falling edges are aligned exactly with selector changes, ensuring timing behavior is also acceptable. A full printout of all tested cases, along with their corresponding expected and obtained outputs, is included in the Appendix for reference.

4.1.2 Arithmetic Unit

The arithmetic unit of the 1-bit ALU is primarily responsible for executing all arithmetic operations such as addition, subtraction, increment, and decrement. This unit is constructed using two core structural components: a full adder and a 4-to-1 multiplexer. We begin by discussing the design and verification of the full adder, which serves as the computational engine for binary addition and forms the foundation for more complex arithmetic functions.

Full Adder

The design of the full adder is central to the operation of the arithmetic unit in the 1-bit ALU. Before diving into the detailed implementation of the arithmetic unit, it is important to understand the fundamental logic and purpose of the full adder circuit.

Full Adder Design Description

A full adder is a fundamental building block in arithmetic logic units (ALUs), designed to perform the addition of three binary inputs: A, B and a carry-in bit (Cin). The full adder produces a sum output (D) and a carry-out output (Cout). The logic equations for a full adder are as follows:

- Sum (D) = A XOR B XOR Cin
- Carry Out (Cout) = (A AND B) OR (A AND Cin) OR (B AND Cin)

A	B	CIN	D	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2: Full Adder Truth Table
This table outlines the complete set of input combinations for a full adder, showing how inputs A, B, and Cin produce the corresponding sum output (D) and carry-out (Cout).

These equations reflect the behavior summarized in the full adder truth table shown in **Table 2**, which includes all eight possible input combinations for A, B, and Cin, each producing the correct sum and carry-out. This operation forms the core of addition-based instructions in the ALU such as simple transfer of A addition of A and B, increment of A, and arithmetic with carry (addition with carry or subtraction with borrow.

In our structural 1-bit ALU design, the full adder is a critical component embedded within the arithmetic unit. It accepts two 1-bit operands and a carry-in signal, producing a 1-bit result and a carry-out. The design leverages basic logic gates (XOR, AND, OR) to meet synthesis and implementation requirements in a fully structural manner.

In the implemented Verilog code, the full adder is constructed as a structural module using gate-level primitives. Two XOR gates generate the sum output Di, while three AND gates and a final OR gate calculate the carry-out Couti. The input operands Ai and Bi are passed directly or selected via a multiplexer depending on the arithmetic operation being performed.

This full adder design is functionally flexible and supports a variety of ALU operations, including:

- **Transfer A** (Di = Ai)
- **Increment A** (Di = Ai + 1)
- **Addition** (Di = Ai + Bi)
- **Addition with carry** (Di = Ai + Bi + Cin)
- **Subtraction with borrow** (Di = Ai + ~Bi)
- **Subtraction** (Di = Ai + ~Bi + Cin)
- **Decrement A** (Di = Ai - 1)

Each of these operations is selected through a 4-to-1 multiplexer preceding the full adder (see **Figure 5**) based on control signals S1 and S0. The selected value is added to Ai using the full adder, with Cin controlling carry-based operations. The output Di is routed as the arithmetic result of the 1-bit ALU, and Couti is propagated as the carry to the next bit in a 32-bit cascade.

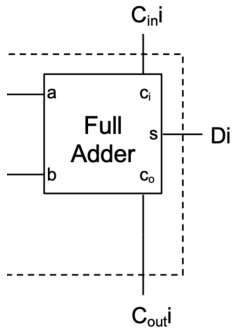


Figure 5: Gate-Level Symbol of Full Adder in Arithmetic Unit

Full Adder Simulation and Verification

The full adder module was functionally verified using exhaustive testing, where all eight possible input combinations of the three 1-bit inputs—Ai, Bi, and Cini—were applied. Each case was validated by comparing the output sum Di and carry-out Couti against the expected results. Simulation waveforms in **Figure 6** clearly show the propagation of signals and confirm the accuracy of the module. The output Di matches the expected Ai XOR Bi XOR Cini, and Couti corresponds to the carry logic ((Ai AND Bi) OR (Ai AND Cini) OR (Bi AND Cini)).

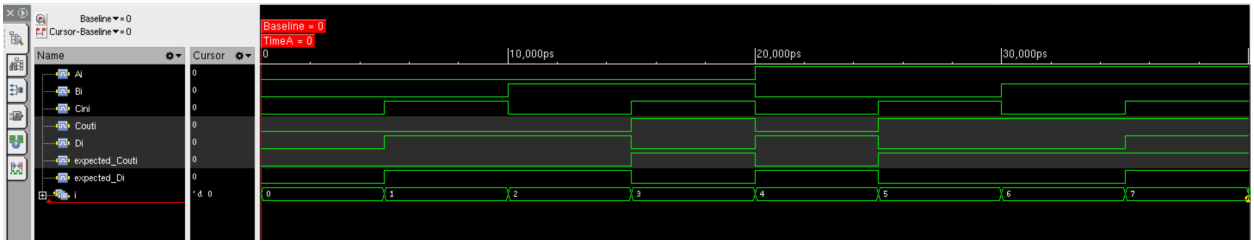


***Figure 6**: Simulation Waveform for Full Adder Functional Verification*

Table 3 confirms the correctness and completeness of the design. The exhaustive testing is outlined below, each representing an arithmetic operation:

Case	Ai	Bi	Cini	Di	Couti	Expected Di	Expected Couti
1	0	0	0	0	0	0	0
2	0	0	1	1	0	1	0
3	0	1	0	1	0	1	0
4	0	1	1	0	1	0	1
5	1	0	0	1	0	1	0
6	1	0	1	0	1	0	1
7	1	1	0	0	1	0	1
8	1	1	1	1	1	1	1

***Table 3**: Exhaustive Truth Table and Simulation Results for Full Adder*

This table outlines all possible input combinations for the full adder (Ai, Bi, Cini) along with the corresponding sum output (Di), carry-out (Couti), and their expected values. The results confirm that the full adder behaves correctly under every logical condition, matching the theoretical outputs.

4-to-1 Multiplexer for Arithmetic Unit

4-to-1 Multiplexer Design Description
See section4.1.1

4-to-1 Multiplexer Simulation and Verification
See section4.1.1

Complete Arithmetic Unit Design Description

The arithmetic unit in the 1-bit ALU is designed to support multiple arithmetic operations through a combination of a 4-to-1 multiplexer and a full adder. This modular architecture (see **Figure 7**) enables operation flexibility while maintaining a fully structural design.

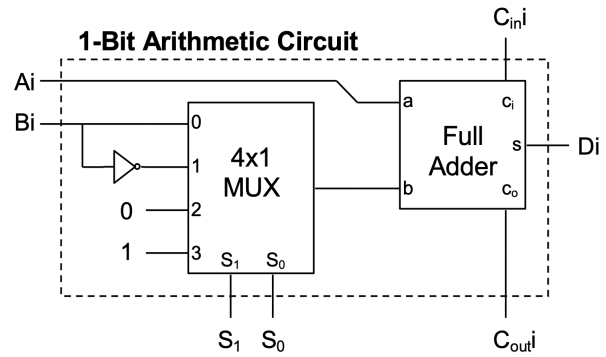


Figure 7. *Gate-Level Schematic of the 1-Bit Arithmetic Unit*

As previously described, the full adder computes the sum (Di) and carry-out (Couti) of three input bits: Ai, a modified version of Bi, and Cini. The second operand to the adder is determined using a 4-to-1 multiplexer, which selects among constant logic 0, the direct value of Bi, the bitwise complement ~Bi, or constant logic 1, based on the 2-bit select lines S1and S0.

The behavior of the arithmetic unit is summarized in the truth table below, showing how different combinations of S1, S0, and Cini produce operations such as Transfer A, Increment A, Addition, Subtraction with and without borrow, and Decrement A.

S1	S0	Cini	Operation	Function
0	0	0	F = A	Transfer A
0	0	1	F = A + 1	Increment A
0	1	0	F = A + B	Addition
0	1	1	F = A + B + 1	Add with carry
1	0	0	F = A + ~B	Subtract with borrow
1	0	1	F = A + ~B + 1	Subtraction
1	1	0	F = A - 1	Decrement A
1	1	1	F = A	Transfer A

Table 4. Control Signal Combinations and Their Corresponding Arithmetic Operations

This table summarizes the behavior of the arithmetic unit based on the values of the select signals S1, S0, and the carry-in input Cini. Each unique combination enables a specific operation such as transfer, increment, addition, subtraction, or decrement.

In the arithmetic unit, the operand that gets added to Ai is selected through a 4-to-1 multiplexer. This operand, denoted as Bi_after here, is derived based on the control inputs S1 and S0, as well as the actual input Bi. The multiplexer implements the following Boolean logic:

This expression enables the selection of different operand values (e.g., Bi, Bi_after, 1, or 0) to support a variety of arithmetic operations like addition, subtraction, and increment/decrement.

$$Bi\_after = (\sim S1 \text{ AND } S0 \text{ AND } Bi) \text{ OR } (S1 \text{ AND } \sim S0 \text{ AND } Bi) \text{ OR } (S1 \text{ AND } S0)$$

The output sum Di of the full adder within the arithmetic unit is determined using the exclusive-OR logic between the primary operand Ai, the selected operand Bi_after*, and the carry-in* C{in}. The Boolean expression is:

$$Di = Ai \text{ XOR } Bi\_after \text{ XOR } Cini$$

This formula ensures the proper generation of the sum bit for a wide range of arithmetic functions supported by the ALU.

The carry-out Couti from the full adder is crucial for ripple-carry propagation in multi-bit ALUs. It is generated based on the majority function of Ai, Bi_after, and Cini:

$$Couti = (Ai \text{ AND } Bi\_after) \text{ OR } (Ai \text{ AND } Cini\}) \text{ OR } (Bi\_after \text{ AND } Cini)$$

This logic ensures accurate carry propagation and is consistent with the behavior of standard full adders. These expressions comprehensively define the arithmetic unit's logic and validate its ability to support the required operations structurally. The implementation of the unit follows this logical design using only basic logic gates and hierarchical module instantiation in Verilog.

Complete Arithmetic Unit Simulation and Verification

To verify the correctness of the arithmetic unit, we performed exhaustive testing on all combinations of control and input signals. This includes evaluating each operation mode across all possible logical combinations of the inputs Ai, Bi, and Cini, under every configuration of the select lines S1 and S0. The results demonstrate the robustness and functional correctness of the design across arithmetic modes such as Transfer, Increment, Addition, Addition with Carry, Subtraction with Borrow, Subtraction, and Decrement.

Each test was run using a custom testbench and validated using functional waveform analysis and expected output comparison. **Table 5** below outlines a subset of these test cases, comparing actual outputs Di, Couti with their expected values.

sel1	sel0	Ai	Bi	Cini	Di	Couti	Expected_Di	Expected_Couti	Operation
0	0	0	0	0	0	0	0	0	Transfer A
0	0	0	0	1	1	0	1	0	Increment A
0	0	0	1	0	0	0	0	0	Transfer A
0	0	0	1	1	1	0	1	0	Increment A
0	0	1	0	0	1	0	1	0	Transfer A
0	0	1	0	1	0	1	0	1	Increment A
0	0	1	1	0	1	0	1	0	Transfer A
0	0	1	1	1	0	1	0	1	Increment A
0	1	0	0	0	0	0	0	0	Addition A+B
0	1	0	0	1	1	0	1	0	Addition A+B+1
0	1	0	1	0	1	0	1	0	Addition A+B
0	1	0	1	1	0	1	0	1	Addition A+B+1
0	1	1	0	0	1	0	1	0	Addition A+B
0	1	1	0	1	0	1	0	1	Addition A+B+1
0	1	1	1	0	0	1	0	1	Addition A+B
0	1	1	1	1	1	1	1	1	Addition A+B+1
1	0	0	0	0	1	0	1	0	Subtraction A-B-1
1	0	0	0	1	0	1	0	1	Subtraction A-B
1	0	0	1	0	0	0	0	0	Subtraction A-B-1
1	0	0	1	1	1	0	1	0	Subtraction A-B
1	0	1	0	0	0	1	0	1	Subtraction A-B-1
1	0	1	0	1	1	1	1	1	Subtraction A-B
1	0	1	1	0	1	0	1	0	Subtraction A-B-1
1	0	1	1	1	0	1	0	1	Subtraction A-B
1	1	0	0	0	1	0	1	0	Decrement A
1	1	0	0	1	0	1	0	1	Transfer A
1	1	0	1	0	1	0	1	0	Decrement A
1	1	0	1	1	0	1	0	1	Transfer A
1	1	1	0	0	0	1	0	1	Decrement A
1	1	1	0	1	1	1	1	1	Transfer A
1	1	1	1	0	0	1	0	1	Decrement A
1	1	1	1	1	1	1	1	1	Transfer A

Table 5. *Exhaustive Functional Verification of the Arithmetic Unit*
This table summarizes the results of all possible input combinations of the arithmetic unit. It includes actual and expected outputs for Di and Couti, along with the corresponding operation. The results confirm the correct behavior across all arithmetic modes (Transfer, Increment, Addition, Subtraction, and Decrement).

Key Verified Functional Cases:

- Transfer A (sel = 00, Cin = 0): Confirms that only Ai is propagated to Di, with Couti = 0.
- Increment A (sel = 00, Cin = 1): Confirms proper carry-in addition to Ai.
- Addition A + B (sel = 01, Cin = 0) and A + B + 1 (sel = 01, Cin = 1): Validates that Bi is correctly passed and summed with Ai.
- Subtraction A − B − 1 (sel = 10, Cin = 0) and A − B (sel = 10, Cin = 1): Validates the use of bitwise negation for subtraction with/without borrow.
- Decrement A (sel = 11, Cin = 0): Verifies subtracting one from Ai.
- Transfer A (sel = 11, Cin = 1): Ensures operation degenerates to pass-through for testing control consistency.

These strategic cases are supported with waveform outputs as shown in **Figure 8**, and additional console results from Xcelium are provided in the Appendix for completeness.

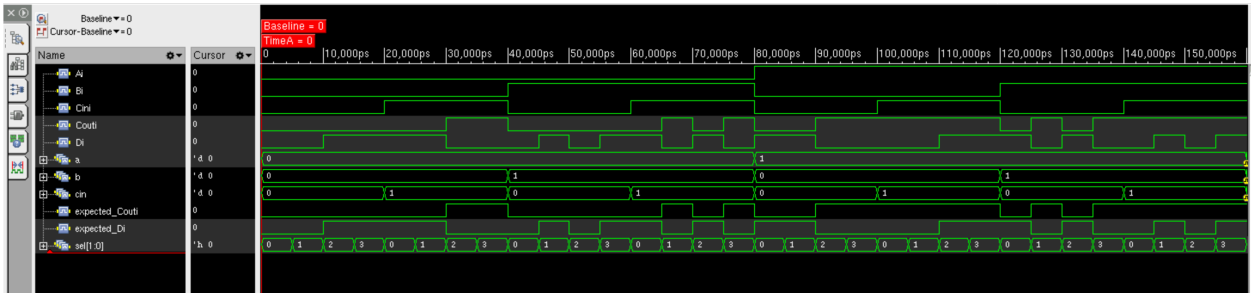


Figure 8: *Functional Simulation Waveform for the Arithmetic Unit*

All tests passed successfully as confirmed by both waveform results and Xcelium terminal logs (see Appendix). The simulation confirms the arithmetic unit's structural integrity and correctness of control signal interpretation across all eight functional arithmetic modes.

4.1.3 Logic Unit

The Logic Unit Design Description follows as a continuation of the modular 1-bit ALU architecture. Building on the arithmetic operations, this section outlines the construction of the logic unit responsible for performing bitwise logic functions. Using a combination of basic logic gates and a 4-to-1 multiplexer, the logic unit provides the ALU with essential logical capabilities such as AND, OR, XOR, and NOT, based on control signals S1 and S0.

Logic Unit Design Description

The logic unit within the 1-bit ALU is designed to perform basic bitwise operations between two 1-bit operands: Ai and Bi. To support modularity and reuse, the design implements each logic operation structurally using basic gates—AND, OR, XOR, and NOT—and then selects the result using a 4-to-1 multiplexer.

This selection is controlled by the 2-bit control signal S1S0, which chooses one of the four logic functions based on the operation required. **Figure 3** illustrates that each logic operation is performed in parallel and fed into a shared multiplexer.

The logical operations handled by the unit are as follows:

- **AND** operation (Ai AND Bi) when S1S0 = 00
- **OR** operation (Ai OR Bi) when S1S0 = 01
- **XOR** operation (Ai XOR Bi) when S1S0 = 10
- **NOT** operation (~Ai) when S1S0 = 11

These operations are computed structurally in the HDL code and routed into the 4-to-1 multiplexer, which then outputs the selected operation result Ei. The Boolean expression for Ei, using basic gates and control logic, is:

Ei = (~S1 AND ~S0 AND (Ai AND Bi)) OR (~S1 AND S0 AND (Ai OR Bi)) OR (S1 AND ~S0 AND (Ai XOR Bi)) OR (S1 AND S0 AND ~Ai)

Logic Unit Simulation and Verification

To verify the correctness and robustness of the logic unit, an exhaustive set of functional tests was conducted using the Xcelium simulator. The testbench iterates through all possible input combinations of control signals S1, S0 and data inputs Ai, Bi, validating the output Ei against the expected results derived from the implemented logical operations.

The logic unit supports four operations, selected using a 2-bit control signal:

- 00: AND (Ai AND Bi)
- 01: OR (Ai OR Bi)
- 10: XOR (Ai XOR Bi)
- 11: NOT (~Ai)

Key Simulation Results:

1. AND Operation (sel1, sel0 = 00)
    a. Case: Ai=1, Bi=1 → Expected Ei=1, Observed Ei=1
    b. Case: Ai=0, Bi=1 → Expected Ei=0, Observed Ei=0

2. OR Operation (sel1, sel0 = 01)
    a. Case: Ai=1, Bi=0 → Expected Ei=1, Observed Ei=1
    b. Case: Ai=0, Bi=0 → Expected Ei=0, Observed Ei=0

3. XOR Operation (sel1, sel0 = 10)
    a. Case: Ai=0, Bi=1 → Expected Ei=1, Observed Ei=1
    b. Case: Ai=1, Bi=1 → Expected Ei=0, Observed Ei=0

4. NOT Operation (sel1, sel0= 11)
    a. Case: Ai=0 → Expected Ei=1, Observed Ei=1
    b. Case: Ai=1 → Expected Ei=0, Observed Ei=0

These strategic test cases span both edge conditions (e.g., all-zero inputs, both inputs high) and typical functional combinations. The terminal output of the remaining test cases of the exhaustive testing process can be found in the Appendix for reference.

As shown in **Figure 9**, the logic unit consistently matched the expected outputs, confirming the correctness of its gate-level design.

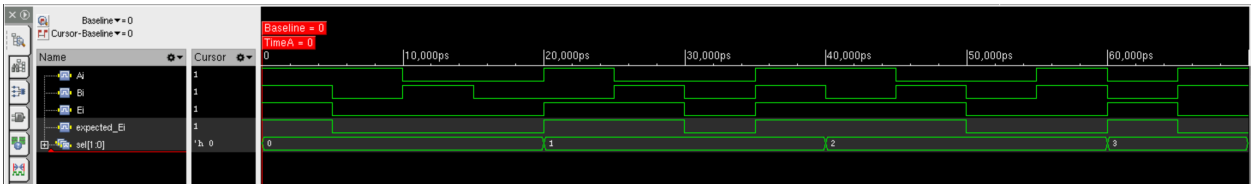


***Figure 9**: Simulation waveform of the logic unit showing the output Ei for various combinations of control inputs s1s0 and data inputs Ai and Bi.*

The table below shows the truth table of all control and input configurations of the logic unit. It also shows all test cases performed on the logical unit to verify the unit's functionality:

sel1	sel0	Ai	Bi	Operation	Ei	Expected_Ei
0	0	0	0	AND Operation (Ai & Bi)	0	0
0	0	0	1	AND Operation (Ai & Bi)	0	0
0	0	1	0	AND Operation (Ai & Bi)	0	0
0	0	1	1	AND Operation (Ai & Bi)	1	1
0	1	0	0	OR Operation (Ai | Bi)	0	0
0	1	0	1	OR Operation (Ai | Bi)	1	1
0	1	1	0	OR Operation (Ai | Bi)	1	1
0	1	1	1	OR Operation (Ai | Bi)	1	1
1	0	0	0	XOR Operation (Ai ^ Bi)	0	0
1	0	0	1	XOR Operation (Ai ^ Bi)	1	1
1	0	1	0	XOR Operation (Ai ^ Bi)	1	1
1	0	1	1	XOR Operation (Ai ^ Bi)	0	0
1	1	0	0	NOT Operation (~Ai)	1	1
1	1	0	1	NOT Operation (~Ai)	1	1
1	1	1	0	NOT Operation (~Ai)	0	0
1	1	1	1	NOT Operation (~Ai)	0	0

***Table 6**: Truth Table and Simulation Results for the Logic Unit.*
This table summarizes the functional behavior of the logic unit across all possible combinations of select lines (sel1, sel0) and inputs (Ai, Bi). It includes the operation type, actual output (Ei), and expected output (Expected_Ei), verifying the correct execution of AND, OR, XOR, and NOT operations.

4.2 ALU 1-bit Integration

In this section, we present the integration of the previously designed and verified submodules—namely the arithmetic unit, logic unit, and the 4-to-1 multiplexer—into a single unified 1-bit ALU design. The integration encapsulates core functional blocks responsible for arithmetic, logic, and shift operations, along with a multiplexer-based control mechanism that selects the appropriate output based on a 4-bit control signal.

ALU 1-bit Design Description

The 1-bit ALU integrates arithmetic, logic, and shift operations into a modular unit that can be chained to form a multi-bit ALU. It leverages Verilog-based structural design principles, reusing previously verified submodules and adding shift functionality through combinational logic.

Figure 10 below illustrates the internal structure of a 1-bit ALU. It includes a 1-bit Arithmetic Circuit for arithmetic operations (controlled by S1 and S0), a 1-bit Logic Circuit for logic operations (also controlled by S1 and S0), and combinational logic for shift-right and shift-left operations. The 4-to-1 multiplexer selects one of the four results—arithmetic (Di), logic (Ei), shift-right (A[i-1]), or shift-left (A[i+1])—based on the most significant bits of the control signal (S3, S2). The selected result is output as Fi, while the arithmetic carry-out (Couti) is propagated to the next ALU slice for multi-bit operation.

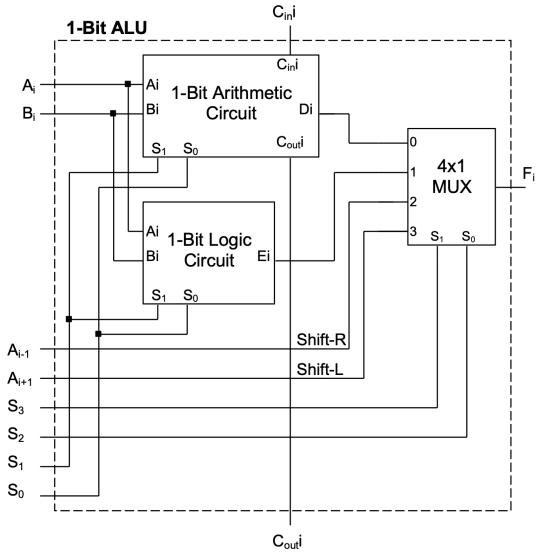


Figure 10: *1-Bit ALU Block Diagram*

As can be seen in **Figure 10,** the ALU takes the following inputs:

- Ai, Bi: 1-bit primary operands.
- A_prev (A[i-1]) , A_next (A[i+1]): Bits from adjacent ALU slices used for shift operations.
- Cini: Carry-in bit for arithmetic operations.
- sel[3:0]: 4-bit control signal for operation selection.

The 1-bit ALU structurally instantiates the following submodules:

1. **Arithmetic Unit:** Performs transfer, addition, subtraction, increment, and decrement. The operation mode the arithmetic unit is controlled by Cini and the operand Bi, which is controlled by sel[1:0]. The output of the arithmetic unit is Di (result) and Couti (carry out). Again, the Boolean expression for both outputs is given by:

$$Di = Ai \text{ XOR } Bi\_after \text{ XOR } Cini$$

$$Couti = (Ai \text{ AND } Bi\_after) \text{ OR } (Ai \text{ AND } Cini\}) \text{ OR } (Bi\_after \text{ AND } Cini)$$

Where Bi_after is the second operand in the arithmetic unit selected after the 4-to-1 multiplexers controlled by the first two bits of the control signal. The Boolean expression of Bi_after is given by:

$$Bi\_after = (\sim S1 \text{ AND } S0 \text{ AND } Bi) \text{ OR } (S1 \text{ AND } \sim S0 \text{ AND } Bi) \text{ OR } (S1 \text{ AND } S0)$$

2. **Logic Unit:** Performs bitwise AND, OR, XOR, and NOT operations. The operation mode is also Controlled by sel[1:0]. And the output of the logic unit is Ei (logic result). The Boolean expression for Ei is again given by :

   Ei = (~S1 AND ~S0 AND (Ai AND Bi)) OR (~S1 AND S0 AND (Ai OR Bi)) OR (S1 AND ~S0 AND (Ai XOR Bi)) OR (S1 AND S0 AND ~Ai)

   Note: The value of Couti doesn't matter (denoted x in the truth table of the ALU) if the operation mode selected is a logic unit operation.

3. **Shift Logic (direct combinational logic):** Unlike the arithmetic and logic units, the shift operations are not encapsulated as standalone submodules. Instead, they are implemented inline within the ALU using combinational logic.

   The two most significant bits of the selection signal sel[3:2] determine the type of output operation. If sel[3:2] == 10, the final output is assigned A_prev (A[i-1]), effectively shifting the input Ai one position to the right in a multi-bit ALU configuration. If sel[3:2] == 11, the output is set to A_next (A[i+1]), representing a shift left.
   The enable conditions are constructed using **AND** gates in combination with the control bits. This logic ensures that only the appropriate shift path is activated depending on the control value. The Boolean expressions of shift left, and shift right are given by:

   Shift_Right = sel[3] AND ~sel[2] AND A[i-1]
   Shift_Left = sel[3] AND sel[2] AND A[i+1]

4. **Output MUX**: The final output is selected using the most significant two bits sel[3:2] as control inputs to the 4-to-1 multiplexer:

   Fi = (~sel[3] AND sel[2] AND Di) + (sel[3] AND sel[2] AND Ei) + (sel[3] AND sel[2] AND A[i-1]) + (sel[3] AND sel[2] AND A[i+1])

5. **Table 7** below summarizes the full range of operations based on the control signal. The truth table defines how each combination of sel[3] to sel[0] and Cin corresponds to a specific ALU function.

Sel[3]	Sel[2]	Sel[1]	Sel[0]	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with Carry
0	0	1	0	0	F = A + B′	Subtract with Borrow
0	0	1	0	1	F = A + B′ + 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A

0	0	1	1	1	F = A	Transfer A
0	1	0	0	X	F = A · B	AND
0	1	0	1	X	F = A + B	OR
0	1	1	0	X	F = A ⊕ B	XOR
0	1	1	1	X	F = ~A	Complement A (NOT)
1	0	X	X	X	$F = A_{(i-1)}$	Shift Right A (1-bit) into F
1	1	X	X	X	$F = A_{(i+1)}$	Shift Left A (1-bit) into F

Table 7: *Functional Description of the 1-Bit ALU*

ALU 1-bit Simulation and Verification

To validate the correctness of the 1-bit ALU design, an exhaustive set of functional verification tests was conducted. These tests were implemented through a comprehensive Verilog testbench that iterates through all meaningful combinations of control signals (S3–S0) and inputs (Ai, Bi, Cin, A_prev, A_next) to evaluate each of the ALU's operations as specified in the functional truth table. **Table 8** (ALU Functional Truth Table) summarizes all possible control combinations and their expected operations. It serves as a reference point for verifying the ALU's behavior against the test results. This ensures coverage of:

- All 8 arithmetic operations (S3S2=00)
- All 4 logic operations (S3S2=01)
- Both shift operations (S3S2=10 and 11)

Sel3	**Sel2**	**Sel1**	**Sel0**	**Ai**	**Bi**	**Cin**	**F**	**Cout**	**Operation**
0	0	0	0	0	0	0	0	0	Transfer A
0	0	0	0	0	0	1	1	0	Increment A
0	0	0	0	0	1	0	0	0	Transfer A
0	0	0	0	0	1	1	1	0	Increment A
0	0	0	0	1	0	0	1	0	Transfer A
0	0	0	0	1	0	1	0	1	Increment A
0	0	0	0	1	1	0	1	0	Transfer A
0	0	0	0	1	1	1	1	0	Increment A
0	0	0	1	0	0	0	0	0	Addition A+B
0	0	0	1	0	0	1	1	0	Addition A+B+1
0	0	0	1	0	1	0	1	0	Addition A+B
0	0	0	1	0	1	1	1	1	Addition A+B+1
0	0	0	1	1	0	0	1	0	Addition A+B
0	0	0	1	1	0	1	0	1	Addition A+B+1
0	0	0	1	1	1	0	1	1	Addition A+B

0	0	0	1	1	1	1	1	1	Addition A+B+1
0	0	1	0	0	0	0	0	0	Subtraction A-B-1
0	0	1	0	0	0	1	0	1	Subtraction A-B
0	0	1	0	0	1	0	0	0	Subtraction A-B-1
0	0	1	0	0	1	1	1	1	Subtraction A-B
0	0	1	0	1	0	0	0	1	Subtraction A-B-1
0	0	1	0	1	0	1	1	1	Subtraction A-B
0	0	1	0	1	1	0	1	0	Subtraction A-B-1
0	0	1	0	1	1	1	1	0	Subtraction A-B
0	0	1	1	0	0	0	1	0	Decrement A
0	0	1	1	0	0	1	0	1	Transfer A
0	0	1	1	0	1	0	1	1	Transfer A
0	0	1	1	0	1	1	1	1	Transfer A
0	0	1	1	1	0	0	1	0	Decrement A
0	0	1	1	1	0	1	0	1	Transfer A
0	0	1	1	1	1	0	1	1	Transfer A
0	0	1	1	1	1	1	1	1	Transfer A
0	1	0	0	0	0	x	0	x	AND Operation
0	1	0	0	0	1	x	0	x	AND Operation
0	1	0	0	1	0	x	0	x	AND Operation
0	1	0	0	1	1	x	1	x	AND Operation
0	1	0	1	0	0	x	0	x	OR Operation
0	1	0	1	0	1	x	1	x	OR Operation
0	1	0	1	1	0	x	1	x	OR Operation
0	1	0	1	1	1	x	1	x	OR Operation
0	1	1	0	0	0	x	0	x	XOR Operation
0	1	1	0	0	1	x	1	x	XOR Operation
0	1	1	0	1	0	x	1	x	XOR Operation
0	1	1	0	1	1	x	0	x	XOR Operation
0	1	1	1	0	0	x	1	x	NOT Operation
0	1	1	1	1	0	x	0	x	NOT Operation
1	0	x	x	0	x	x	0	x	Shift Right A
1	0	x	x	1	x	x	0	x	Shift Right A

1	1	x	x	0	x	x	0	x	Shift Left A
1	1	x	x	1	x	x	1	x	Shift Left A

Table 8: ALU 1-bit Exhaustive Functional Test Cases
This table presents a comprehensive set of input combinations used to verify the 1-bit ALU. Each row corresponds to a unique test case defined by the 4-bit control signal (S3 S2 S1 S0), the carry-in input (Cin), and operands Ai and Bi. The resulting outputs F (ALU output) and Cout (carry-out) are compared against the expected behavior based on the selected operation. The table covers arithmetic operations (e.g., transfer, increment, addition with/without carry, subtraction with/without borrow, decrement), logic operations (AND, OR, XOR, NOT), and shift operations (left and right). This exhaustive testing ensures full validation of the ALU's functional correctness.

The ALU operations tested include arithmetic (addition, increment, decrement, subtraction with and without borrow), logical (AND, OR, XOR, NOT), and shift operations (shift right and shift left). Each operation was verified with selected edge and representative test cases that confirm the correct behavior under varying input scenarios.

Key strategic cases include:

- **Transfer A:** For control signal 0000 and Cin=0, the output Fi matches the value of Ai, confirming a direct pass-through of operand A.
  Test: Transfer A (sel=0000, Cin=0) sel=0000 Ai=0 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=0 Couti=0 | Obtained Fi=0 Couti=0
  [PASS] Arithmetic

- **Increment A:** For 0000 and Cin=1, the test confirms Fi = A + 1, ensuring carry-in is properly added.
  Test: Increment A (sel=0000, Cin=1) sel=0000 Ai=0 Bi=0 Cin=1 A_prev=0 A_next=0 → Expected Fi=1 Couti=0 | Obtained Fi=1 Couti=0
  [PASS] Arithmetic

- **Addition A + B:** Verified with 0001 and both Cin=0 and Cin=1, showing correct full adder operation.
  Test: Addition A+B (sel=0001, Cin=0) sel=0001 Ai=0 Bi=1 Cin=0 A_prev=0 A_next=0 → Expected Fi=1 Couti=0 | Obtained Fi=1 Couti=0
  [PASS] Arithmetic

  Test case: Addition with carry A+B+1 (sel=0001, Cin=1) sel=0001 Ai=1 Bi=1 Cin=1 A_prev=0 A_next=0 → Expected Fi=1 Couti=1 | Obtained Fi=1 Couti=1
  [PASS] Arithmetic

- **Subtraction A - B:** Tested with 0010, and both borrow and non-borrow cases (Cin=1 and Cin=0) show correct 2's complement subtraction.
  Test: Subtraction A-B (sel=0010, Cin=1) sel=0010 Ai=1 Bi=0 Cin=1 A_prev=0 A_next=0 → Expected Fi=1 Couti=1 | Obtained Fi=1 Couti=1
  [PASS] Arithmetic

  Test: Subtraction with borrow A-B-1 (sel=0010, Cin=0) sel=0010 Ai=1 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=0 Couti=1 | Obtained Fi=0 Couti=1
  [PASS] Arithmetic

- **Decrement A:** 0011 with Cin=0 successfully outputs A - 1, demonstrating borrow handling.
  Test: Decrement A-1 (sel=0011, Cin=0) sel=0011 Ai=1 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=0 Couti=1 | Obtained Fi=0 Couti=1
  [PASS] Arithmetic

- **Logic Operations (AND, OR, XOR, NOT):** With control signals in the range 0100-0111, logic functions were confirmed to be correct across multiple combinations of Ai and Bi.

  Test: Logic AND (sel=0100) sel=0100 Ai=1 Bi=1 Cin=0 A_prev=0 A_next=0 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

  Test: Logic OR (sel=0101) sel=0101 Ai=1 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

  Test: Logic XOR (sel=0110) sel=0110 Ai=1 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

  Test: Logic NOT (sel=0111) sel=0111 Ai=0 Bi=0 Cin=0 A_prev=0 A_next=0 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

- **Shift Operations:** For S3S2 = 10 (shift right) and 11 (shift left), the ALU correctly routed A_prev and A_next respectively to Fi, depending on Ai–1 and Ai+1, simulating a bitwise shift in a multi-bit ALU chain.
  Test: Shift Right A (sel=1000) sel=1000 Ai=0 Bi=0 Cin=0 A_prev=1 A_next=0 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

  Test: Shift Left A (sel=1100)
  sel=1100 Ai=1 Bi=0 Cin=0 A_prev=0 A_next=1 → Expected Fi=1 Couti=x | Obtained Fi=1 Couti=0
  [PASS] Logic/Shift

The simulation waveform (**Figure 11**) visually confirms that the output Fi matches the expected_Fi across the entire test duration for all control signal transitions. The control lines (sel[3:0]) are shown sweeping through all operational modes, with Fi and Couti reacting accordingly. The consistency of the expected_Fi and Fi traces is clear and confirms that each module (arithmetic, logic, shift, mux) was correctly triggered.

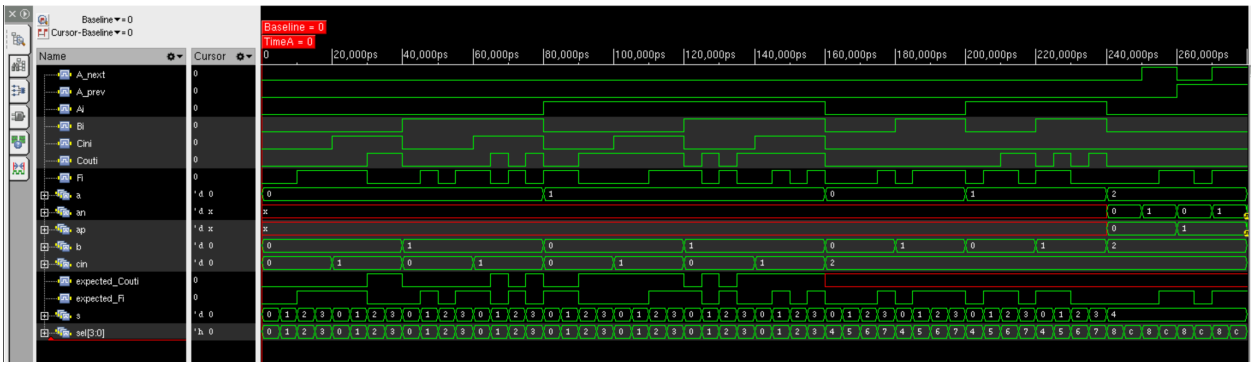


Figure 11: 1-Bit ALU Simulation Waveform.

All test case results were echoed in the simulation console output. For every test scenario, both the expected and actual outputs for Fi and Couti were printed and compared. In every case, a [PASS] confirmation was returned, indicating that the outputs matched precisely. These console logs are included in the **Appendix** for full traceability.

Part 2: 32-bit ALU Design

Building upon the verified 1-bit ALU design, Part 2 focuses on scaling the architecture to a 32-bit Arithmetic Logic Unit using two different implementation strategies: modular and behavioral. The modular implementation cascades thirty-two 1-bit ALU slices to form a structural design that mirrors real-world hardware, while the behavioral implementation simplifies the same functionality using high-level Verilog constructs for rapid simulation and validation. Together, these approaches enable both low-level control and high-level abstraction, ensuring full operational coverage across arithmetic, logic, and shift functionalities.

4.3 32-bit ALU (Modular Implementation)

To realize the 32-bit ALU using a modular architecture, the design builds upon the previously verified 1-bit ALU slices. The following design description outlines how these slices are instantiated and connected in a bit-slice fashion to form a complete 32-bit Arithmetic Logic Unit. Key considerations include carry propagation between slices, operand wiring, and control signal distribution to ensure consistent operation across all bits.

32-bit ALU Modular Implementation Design Description

The 32-bit ALU is architecturally constructed as a modular extension of the previously verified 1-bit ALU slice. The key concept of this design lies in its bit-slice approach, where thirty-two identical instances of the 1-bit ALU module are instantiated and cascaded to form a unified 32-bit data path capable of performing arithmetic, logic, and shift operations. This approach not only promotes scalability but also maintains modularity and ease of debugging.

Slice Instantiation and Wiring

Each slice of the 32-bit ALU handles a corresponding bit of the 32-bit operands A and B, with indexed connections such as A[i], B[i], and F[i] for inputs and outputs. Within the generate loop, the ALU slices are instantiated with appropriate wiring:

- **Carry Propagation**: Carries are passed from the least significant bit (LSB) to the most significant bit (MSB). The Cin signal is only connected to the first slice (i == 0), while subsequent slices receive their carry-in from the carry-out of the previous stage: .Cini(i == 0 ? Cin : carryin[i-1]) (in my Verilog code). This ensures proper chained carry behavior for arithmetic operations like addition and subtraction.

- **Shift Wiring**: For shift operations, each slice must know its neighboring bits: A_prev: Represents the bit to the right of the current slice (for shift-right operations). A_next: Represents the bit to the left (for shift-left operations). These are derived in my Verilog code as: a_prev = (i == 31) ? DinR : A[i + 1]; a_next = (i == 0) ? DinL : A[i - 1]; This boundary logic handles edge conditions for bits A[0] and A[31], using the input lines DinL and DinR to simulate circular or fixed bit shifting.

- **Operation Control**: The 4-bit select input sel[3:0] is broadcast to every 1-bit slice to define the ALU operation. Bits sel[1:0] choose the operation inside the 1-bit ALU, and sel[3:2] control the final MUX that determines the output, Fi.

Output Logic: Each 1-bit slice outputs its respective Fi bit, contributing to the final 32-bit result, F.

Carry Output (Cout): Only arithmetic operations require carry propagation. To handle this, a logic condition is used at the end of the module: assign Cout = (sel[3:2] == 2'b00) ? carryin[31] : 1'b0; This ensures Cout reflects the arithmetic carry only when sel[3:2] == 00, which includes arithmetic operations like addition, subtraction, increment, and decrement.

Visual Representation

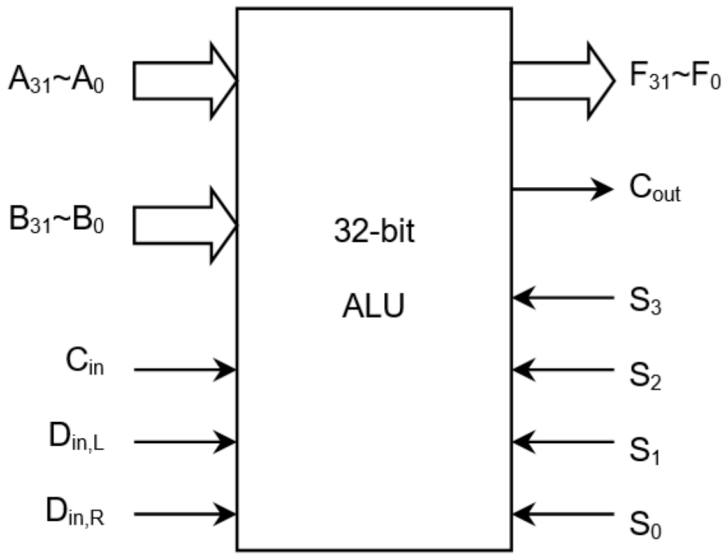


***Figure 12**: Modular 32-bit ALU Input/Output Interface*

This structure is mirrored in the input/output diagram (**Figure 12**), where:

- A and B are 32-bit inputs,
- Cin is the initial carry-in for arithmetic operations,

- DinL and DinR support shift-left and shift-right,
- The select lines S3–S0 determine the ALU operation,
- F is the 32-bit function output,
- Cout is the final carry-out.

32-bit ALU Modular Implementation Simulation and Verification

To validate the functionality of the modular 32-bit ALU design, a comprehensive testbench was developed and executed. The verification strategy involved applying a broad set of strategic test cases covering all categories of operations — arithmetic, logic, shift, and random combinations — and comparing the obtained outputs against the expected results. The output waveform and simulation logs confirm the correctness of the ALU's behavior across all scenarios.

Strategic Test Cases and Results

The verification was structured to include both edge cases and representative use cases. Below is a summary of critical operations tested:

- **Transfer A**:
  Test Case: sel=0000, A=0x00000000, Cin=0
  Expected: F=0x00000000, Cout=0
  Obtained: F=0x00000000, Cout=0
  [PASS] Confirmed that Fi passes Ai without modification.

- **Increment A**:
  Test Case: sel=0000, A=0xffffffff, Cin=1
  Expected: F=0x00000000, Cout=1
  Obtained: F=0x00000000, Cout=1
  [PASS] Wrap-around with carry-out correctly triggered.

- **Addition A + B**:
  Test Case: sel=0001, A=0xffffffff, B=0xffffffff, Cin=0
  Expected: F=0xfffffffe, Cout=1
  Obtained: F=0xfffffffe, Cout=1
  [PASS] Validated full-adder logic for unsigned overflow.

- **Addition A + B + 1 (with carry)**:
  Test Case: sel=0001, A=0x00000001, B=0x00000001, Cin=1
  Expected: F=0x00000003, Cout=0
  Obtained: F=0x00000003, Cout=0
  [PASS] Carry-in correctly included.

- **Subtraction A – B – 1**:
  Test Case: sel=0010, A=0x00000004, B=0x00000003, Cin=0
  Expected: F=0x00000000, Cout=1
  Obtained: F=0x00000000, Cout=1
  [PASS] Confirmed borrow handling.

- **Subtraction A – B (with borrow)**:
  Test Case: sel=0010, A=0x00000004, B=0x00000003, Cin=1

Expected: F=0x00000001, Cout=1
Obtained: F=0x00000001, Cout=1
[PASS] Two's complement subtraction works as expected.

- **Subtraction A – B (B > A)**:
  Test Case: sel=0010, A=0x00000003, B=0x00000004, Cin=0
  Expected: F=0xffffffff, Cout=0
  Obtained: F=0xffffffff, Cout=0
  [PASS] Confirmed underflow is properly computed.

- **Decrement A**:
  Test Case: sel=0011, A=0x00000000, Cin=0
  Expected: F=0xffffffff, Cout=0
  Obtained: F=0xffffffff, Cout=0
  [PASS] Wrap-around correctly occurs on underflow.

Logic Operations (AND, OR, XOR, NOT):

- **AND Operation:**
  Test Case: sel=0100, A=0x0f0f0f0f, B=0xf0f0f0f0
  Expected: F=0x00000000, Cout=0
  Obtained: F=0x00000000, Cout=0
  [PASS]

- **OR Operation:**
  Test Case: sel=0101, A=0x0f0f0f0f, B=0xf0f0f0f0
  Expected: F=0xffffffff, Cout=0
  Obtained: F=0xffffffff, Cout=0
  [PASS]

- **XOR Operation:**
  Test Case: sel=0110, A=0xaaaaaaaa, B=0x55555555
  Expected: F=0xffffffff, Cout=0
  Obtained: F=0xffffffff, Cout=0
  [PASS]

- **NOT Operation:**
  Test Case: sel=0111, A=0x00000000, B=0x00000000
  Expected: F=0xffffffff, Cout=0
  Obtained: F=0xffffffff, Cout=0
  [PASS]

Shift Operations:

- **Right Shift:**
  Test Case: sel=1000, A=0x12345678, DinR=0
  Expected: F=0x091a2b3c, Cout=0
  Obtained: F=0x091a2b3c, Cout=0
  [PASS] Confirmed logical right shift by 1.

- **Left Shift:**
  Test Case: sel=1100, A=0x12345678, DinL=0
  Expected: F=0x2468acf0, Cout=0
  Obtained: F=0x2468acf0, Cout=0
  [PASS] Confirmed logical left shift by 1.

- **Boundary Shift Tests:**
  Test Case (Right Shift Edge): sel=1000, A=0x12345678, DinR=1
  Expected: F=0x891a2b3c, Cout=0
  Obtained: F=0x891a2b3c, Cout=0
  [PASS]

  Test Case (Left Shift Edge): sel=1100, A=0x12345678, DinL=1
  Expected: F=0x2468acf1, Cout=0
  Obtained: F=0x2468acf1, Cout=0
  [PASS]

  Test Case (Both ends active): sel=1100, DinL=1, DinR=1, A=0x12345678
  Expected: F=0x2468acf1, Cout=0
  Obtained: F=0x2468acf1, Cout=0
  [PASS]

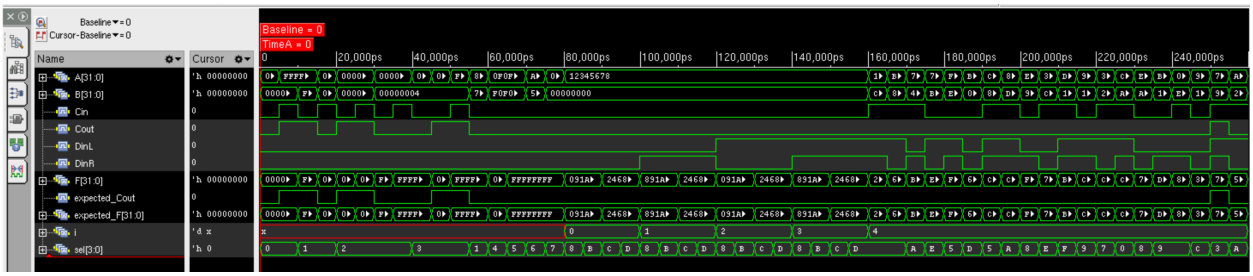


Figure 13: *Simulation Waveform of the 32-bit Modular ALU*

As shown in the simulation waveform (**Figure 13**), the output F[31:0] transitions precisely in sync with the control signals sel[3:0] and the input operand changes. The waveform also shows correct propagation of Cout, especially during overflow and borrow cases. The expected output traces (expected_Fi and expected_Cout) overlap exactly with their corresponding obtained values, validating functional correctness at every clock edge.

4.4 32-bit ALU (Behavioral Implementation)

To begin exploring the behavioral model of the 32-bit ALU, this section presents the design details, including the internal logic that mirrors the functionality of the modular version. By modeling operations at a higher level of abstraction, we ensure the same operational coverage while improving code readability and simulation efficiency.

32-bit ALU Behavioral Implementation Design

The behavioral implementation of the 32-bit ALU models the entire functionality using high-level procedural constructs within a single module. Unlike the modular implementation which instantiates and

wires 32 separate 1-bit ALU slices, the behavioral version centralizes control flow and output logic, simplifying structural overhead and improving simulation efficiency for testing.

This ALU accepts the following 32-bit wide inputs:

- A and B: Operand buses.
- CIN: Carry-in for arithmetic operations.
- DL and DR: Inputs for shift-left and shift-right operations, respectively.
- S[3:0]: 4-bit control signal used to determine the operation type.

The output includes:

- F: The 32-bit result of the ALU operation.
- COUT: A 1-bit flag representing the carry-out (only meaningful during arithmetic).

Internal Logic

The design is organized into four main operation categories, selected using the two most significant bits of the control signal S[3:2]:

1. **Arithmetic Operations (S[3:2] = 2'b00)**
   - A secondary multiplexer behavior is embedded using case (S[1:0]) to choose from:
     - ➔ 2'b00: Transfer A → mux_in = 32'h00000000
     - ➔ 2'b01: Addition (A + B) → mux_in = B
     - ➔ 2'b10: Subtraction (A - B) → mux_in = ~B (2's complement)
     - ➔ 2'b11: Decrement A → mux_in = 32'hFFFFFFFF
   - The final arithmetic output is computed as: temp_sum = A + mux_in + CIN
   - Carry-out is extracted from temp_sum[32].

2. **Logic Operations (S[3:2] = 2'b01)**
   - Direct bitwise operations based on S[1:0]:
     - ➔ 2'b00: F = A & B
     - ➔ 2'b01: F = A | B
     - ➔ 2'b10: F = A ^ B
     - ➔ 2'b11: F = ~A
   - Carry-out is set to a default high (COUT = 1'b0) since it's irrelevant here.

3. **Shift Right (S[3:2] = 2'b10)**
   - Logical right shift by 1 bit using the expression: F = {DR, A[31:1]}The most significant bit is filled by DR.

4. **Shift Left (S[3:2] = 2'b11)**
   - Logical left shift by 1 bit using the expression: F = {A[30:0], DL} The least significant bit is filled by DL.

32-bit ALU Behavioral Implementation Simulation and Verification

To verify the correctness of the behavioral implementation of the 32-bit ALU, an extensive testbench was developed to cover a wide range of arithmetic, logic, and shift operations. All tests were conducted using Xcelium, and simulation waveforms were captured to visually validate the correctness of each operation.

The waveform below (**Figure 14**) illustrates the behavioral ALU output F[31:0] and carry-out Cout across several operations. The results are compared in real time against expected_F and expected_Cout for direct

confirmation. As seen, the outputs of the behavioral design consistently match the expected results for all test cases.

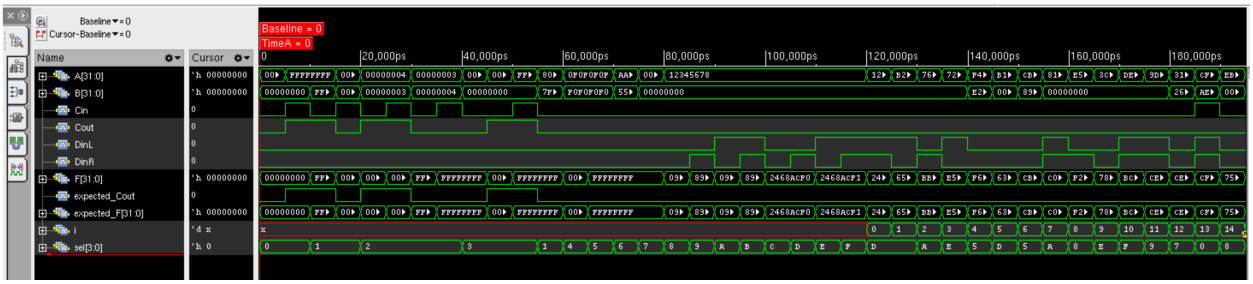


***Figure 14**: Behavioral ALU waveform simulation showing F, Cout, and match with expected outputs over time.*

Strategic Test Cases and Observations

A collection of strategic test cases was employed to ensure complete functional coverage:

- **Transfer A**: sel=0000, A=0x00000000, Cin=0
  Expected: F=0x00000000, Cout=0
  Obtained: F=0x00000000, Cout=0

- **Increment A**: sel=0000, A=0xFFFFFFFF, Cin=1
  Expected: F=0x00000000, Cout=1
  Obtained: F=0x00000000, Cout=1

- **Addition A + B (max values)**:
  sel=0001, A=0xFFFFFFFF, B=0xFFFFFFFF, Cin=0
  Expected: F=0xFFFFFFFE, Cout=1
  Obtained: F=0xFFFFFFFE, Cout=1

- **Addition A + B + 1**:
  sel=0001, A=0x00000001, B=0x00000001, Cin=1
  Expected: F=0x00000003, Cout=0
  Obtained: F=0x00000003, Cout=0

- **Subtraction A - B - 1**:
  sel=0010, A=0x00000004, B=0x00000003, Cin=0
  Expected: F=0x00000000, Cout=1
  Obtained: F=0x00000000, Cout=1

- **Subtraction A - B**:
  sel=0010, A=0x00000004, B=0x00000003, Cin=1
  Expected: F=0x00000001, Cout=1
  Obtained: F=0x00000001, Cout=1

- **Subtraction A - B - 1 (B > A)**:
  sel=0010, A=0x00000003, B=0x00000004, Cin=0
  Expected: F=0xFFFFFFFF, Cout=0
  Obtained: F=0xFFFFFFFF, Cout=0

- **Decrement A (Underflow)**:
  sel=0011, A=0x00000000, Cin=0
  Expected: F=0xFFFFFFFF, Cout=0
  Obtained: F=0xFFFFFFFF, Cout=0

Logic Operations (sel=0100–0111)

- **AND**:
  A=0x0F0F0F0F, B=0xF0F0F0F0
  Expected & Obtained: F=0x00000000

- **OR**:
  A=0x0F0F0F0F, B=0xF0F0F0F0
  Expected & Obtained: F=0xFFFFFFFF

- **XOR**:
  A=0xAAAAAAAA, B=0x55555555
  Expected & Obtained: F=0xFFFFFFFF

- **NOT A**:
  A=0x00000000
  Expected & Obtained: F=0xFFFFFFFF

Shift Operations (sel=10xx for right, 11xx for left)

- **Shift Right**:
  sel=1000, A=0x12345678, DinR=0
  Expected & Obtained: F=0x091A2B3C

  sel=1001, A=0x12345678, DinR=1
  Expected & Obtained: F=0x891A2B3C

- **Shift Left**:
  sel=1100, A=0x12345678, DinL=0
  Expected & Obtained: F=0x2468ACF0

  sel=1101, A=0x12345678, DinL=1
  Expected & Obtained: F=0x2468ACF1

The behavioral ALU correctly implemented all intended operations and passed all functional tests. All values for F and Cout matched expected_F and expected_Cout as confirmed by the waveform (see **Figure 14**) which further confirms temporal correctness and signal transitions for each operation cycle. Full simulation logs and test case outputs are available in the Appendix for detailed examination.

32-bit ALU Behavioral Implementation VS 32-bit ALU modular Implementation Simulation and Verification

To ensure the functional equivalence between the behavioral and modular designs of the 32-bit ALU, an extensive simulation-based comparison was conducted. A dedicated testbench (tb_modular_VS_behavioral.v) was developed to apply identical inputs to both ALUs simultaneously and verify that their outputs matched across all operation types, including arithmetic, logic, and shift operations.

The comparison focused on validating that the modular ALU, constructed from thirty-two 1-bit slices, and the behavioral ALU, written using high-level RTL constructs, produce consistent results under identical conditions. Both output result F and carry-out Cout (where applicable) were monitored and compared. The results were logged in detail and visually confirmed via waveform analysis.

Strategic Verification Results:

Arithmetic Operation Tests:

- **Transfer A:**
  sel=0000, A=0x00000000, B=0x00000000, Cin=0
  Modular: F=0x00000000, Cout=0
  Behavioral: F=0x00000000, Cout=0
  [PASS] Arithmetic Match

- **Increment A:**
  sel=0000, A=0xFFFFFFFF, B=0x00000000, Cin=1
  Modular: F=0x00000000, Cout=1
  Behavioral: F=0x00000000, Cout=1
  [PASS] Arithmetic Match

- **Addition A + B (with and without carry):**
  sel=0001, A=0xFFFFFFFF, B=0xFFFFFFFF, Cin=0
  Modular: F=0xFFFFFFFE, Cout=1
  Behavioral: F=0xFFFFFFFE, Cout=1
  [PASS] Arithmetic Match

  sel=0001, A=0x00000001, B=0x00000001, Cin=1
  Modular: F=0x00000003, Cout=0
  Behavioral: F=0x00000003, Cout=0
  [PASS] Arithmetic Match

- **Subtraction A – B and A – B – 1 (with and without borrow):**
  sel=0010, A=0x00000004, B=0x00000003, Cin=0
  Modular: F=0x00000000, Cout=1
  Behavioral: F=0x00000000, Cout=1
  [PASS] Arithmetic Match

  sel=0010, A=0x00000004, B=0x00000003, Cin=1
  Modular: F=0x00000001, Cout=1
  Behavioral: F=0x00000001, Cout=1
  [PASS] Arithmetic Match

- **Decrement A and Underflow Cases:**
  sel=0011, A=0x00000000, Cin=0
  Modular: F=0xFFFFFFFF, Cout=0
  Behavioral: F=0xFFFFFFFF, Cout=0
  [PASS] Arithmetic Match

Logic Operation Tests (sel = 0100–0111):

- **AND:**
  A = 0x0F0F0F0F, B = 0xF0F0F0F0

Modular & Behavioral Output: F = 0x00000000
[PASS] Match

- **OR:**
  A = 0x0F0F0F0F, B = 0xF0F0F0F0
  Modular & Behavioral Output: F = 0xFFFFFFFF
  [PASS] Match

- **XOR:**
  A = 0xAAAAAAAA, B = 0x55555555
  Modular & Behavioral Output: F = 0xFFFFFFFF
  [PASS] Match

- **NOT A:**
  A = 0x00000000
  Modular & Behavioral Output: F = 0xFFFFFFFF
  [PASS] Match

- **Shift Operation Tests:**
  Logical Shift Right (sel = 10xx):
  A = 0x12345678, DinR = 0 → F = 0x091A2B3C
  A = 0x12345678, DinR = 1 → F = 0x891A2B3C
  Match confirmed in both designs

- **Logical Shift Left (sel = 11xx):**
  A = 0x12345678, DinL = 0 → F (modular & behavioral) = 0x2468ACF0
  A = 0x12345678, DinL = 1 → F = 0x2468ACF1
  [PASS] Match in all tested variations

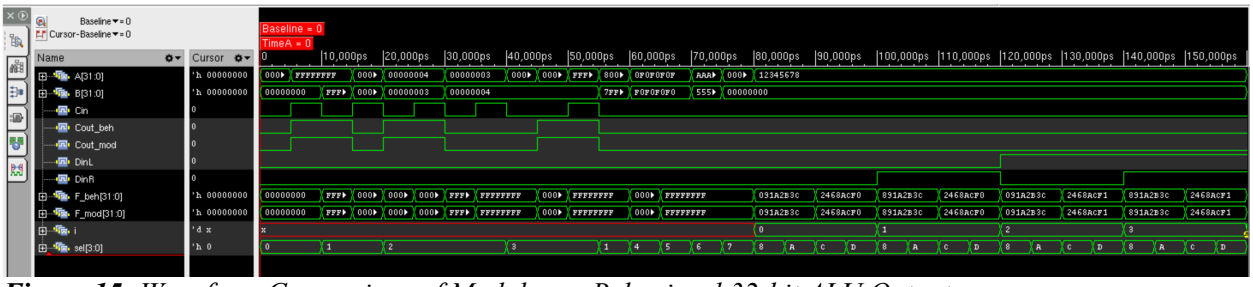


***Figure 15**: Waveform Comparison of Modular vs Behavioral 32-bit ALU Outputs*

The waveform capture (see **Figure 15**) visually confirms the correctness of each output (F) for both designs. The F_mod[31:0] and F_beh[31:0] signals remain identical at every timestamp, reinforcing the equivalence. Even during transitions involving shifts and arithmetic carry-over, both outputs remain in perfect sync, as do the Cout_mod and Cout_beh values for arithmetic functions.

The full simulation output, including expected and obtained results for every test case, is available in the **Appendix**. These include both standard test cases (e.g., addition, subtraction, shifts) and several randomly directed tests to probe edge cases.

5. Synthesis Results

Following the successful simulation and functional verification of both the modular and behavioral 32-bit ALU designs, the next step involved evaluating the hardware efficiency of each implementation through synthesis. This section presents the synthesis results, highlighting key performance metrics including area utilization, power consumption, and timing. We begin by outlining the tools used during synthesis.

5.1 Tools used

All synthesis tasks were performed using Cadence Genus Synthesis Solution, an RTL synthesis tool widely used in industry for ASIC design. Genus translates high-level RTL (Verilog HDL) descriptions into gate-level netlists optimized for area, power, and timing performance based on a given technology library.

The synthesized netlists were analyzed to extract performance metrics such as total cell area, dynamic and leakage power, and worst negative slack (WNS) or maximum frequency. These metrics provide a quantitative comparison between the 1-bit ALU, the 32-bit modular ALU, and the 32-bit behavioral ALU designs in terms of implementation cost and suitability for large-scale digital systems.

5.2 Area, power, and timing reports

To assess the hardware efficiency of our ALU implementations, synthesis was performed on three designs: the 1-bit ALU, the 32-bit ALU using modular composition of 1-bit slices, and the 32-bit ALU using a behavioral description. Each design was synthesized individually using the same technology and tool flow to ensure consistency and fairness in the evaluation.

The following subsections present the synthesis reports for each design, detailing area utilization, total power consumption (including dynamic and leakage power), and key timing characteristics such as critical path delay and worst negative slack. These reports offer insight into the trade-offs between structural modularity and behavioral abstraction in RTL design.

5.2.1 1-bit ALU

The 1-bit ALU was synthesized using Cadence Genus Synthesis Solution. The synthesis report confirms that the design is fully combinational, and area, power, and timing metrics were extracted.

Power Report

```
Instance: /alu_1bit
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       logic     2.11523e-09  1.68016e-08  1.18341e-06  1.20232e-06 100.00%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     2.11523e-09  1.68016e-08  1.18341e-06  1.20232e-06 100.00%
  Percentage           0.18%        1.40%       98.43%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Figure 16: *Power breakdown report for 1-bit ALU from Genus, highlighting switching as the primary source of power consumption.*

According to the Genus-generated power breakdown (see **Figure 16**), the total power consumed by the 1-bit ALU is 1.20232 µW, with switching power dominating the profile at 98.43%. Leakage power is minimal (0.18%), and internal power accounts for 1.40%, indicating that the design is highly efficient from a dynamic standpoint.

Area Report

The total synthesized area of the 1-bit ALU is 53.694 µm², as shown in **Figure 17**. The area usage is distributed across 15 logic cells, including various arithmetic and control gates such as:

- One full adder cell (ADDFHXL) consuming 7.524 µm²
- Multiplexers (MX2X1, MX4XL)
- AND/OR/NOR/NAND gates

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.17-s066_1
  Generated on:           Apr 01 2025  04:59:06 pm
  Module:                 alu_1bit
  Operating conditions:   PVT_0P9V_125C (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================
```

Gate	Instances	Area	Library
ADDFHXL	1	7.524	slow_vdd1v0
AND2XL	1	1.368	slow_vdd1v0
AND3XL	1	2.052	slow_vdd1v0
CLKBUFX20	2	16.416	slow_vdd1v0
CLKMX2X12	1	6.498	slow_vdd1v0
INVXL	1	0.684	slow_vdd1v0
MX2X1	1	2.394	slow_vdd1v0
MX2XL	2	4.788	slow_vdd1v0
MX4XL	1	7.182	slow_vdd1v0
NAND2X1	1	1.026	slow_vdd1v0
NOR2BX1	1	1.368	slow_vdd1v0
NOR2X1	1	1.026	slow_vdd1v0
OR2XL	1	1.368	slow_vdd1v0
total	15	53.694	

Type	Instances	Area	Area %
inverter	1	0.684	1.3
buffer	2	16.416	30.6
logic	12	36.594	68.2
physical_cells	0	0.000	0.0
total	15	53.694	100.0

Figure 17: *Area utilization report showing total area (53.694 µm²) and breakdown across standard cells (inverter, logic, buffer).*

Logic occupies 68.2% of the total area, followed by buffers (30.6%) and inverters (1.3%), reinforcing that the design is logic-dominant with minimal control overhead.

Gate-Level Schematic

The synthesized netlist was inspected visually (see **Figure 18**), showing the correct instantiation and interconnection of the arithmetic unit, logic unit, and shift control logic with muxes. The inputs (Ai, Bi, Cin, sel, A_prev, A_next) feed into respective modules, while the final output is routed through a 4-to-1 multiplexer, as expected.

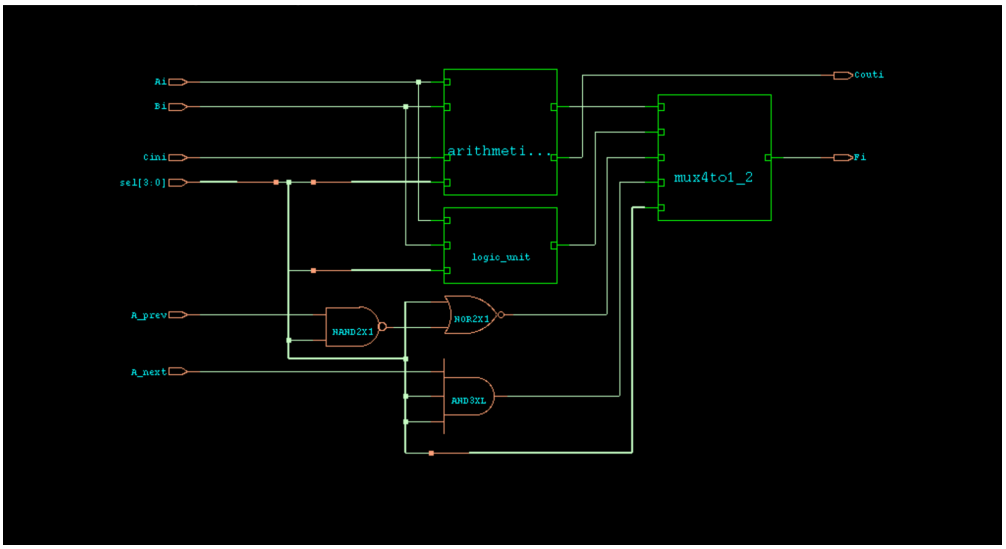


Figure 18: Gate-level synthesized schematic of the 1-bit ALU, illustrating proper data path wiring between arithmetic, logic, and shift units.

5.2.2 32-bit modular ALU

The 32-bit modular ALU was synthesized using Cadence Genus by instantiating thirty-two 1-bit ALU slices in a ripple-carry configuration. Each slice includes an arithmetic unit, logic unit, shift logic, and a 4-to-1 multiplexer. The modules are fully interconnected to pass carry and shift information through the chain.

Power Report

The total power consumption of the modular ALU is 25.5943 µW, which breaks down into:

- Leakage Power: 49.60 nW (0.19%)
- Internal Power: 439.08 nW (1.72%)
- Switching Power: 25.1057 µW (98.09%)

This power profile shown in **Figure 19** shows that switching activity dominates power consumption, which is expected due to heavy operand and control signal transitions throughout the cascaded slices.

```
Instance: /alu_32bit_modular
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        logic     4.96023e-08  4.39078e-07  2.51057e-05  2.55943e-05 100.00%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     4.96023e-08  4.39078e-07  2.51057e-05  2.55943e-05 100.00%
  Percentage           0.19%        1.72%       98.09%      100.00% 100.00%
  ------------------------------------------------------------------------
```

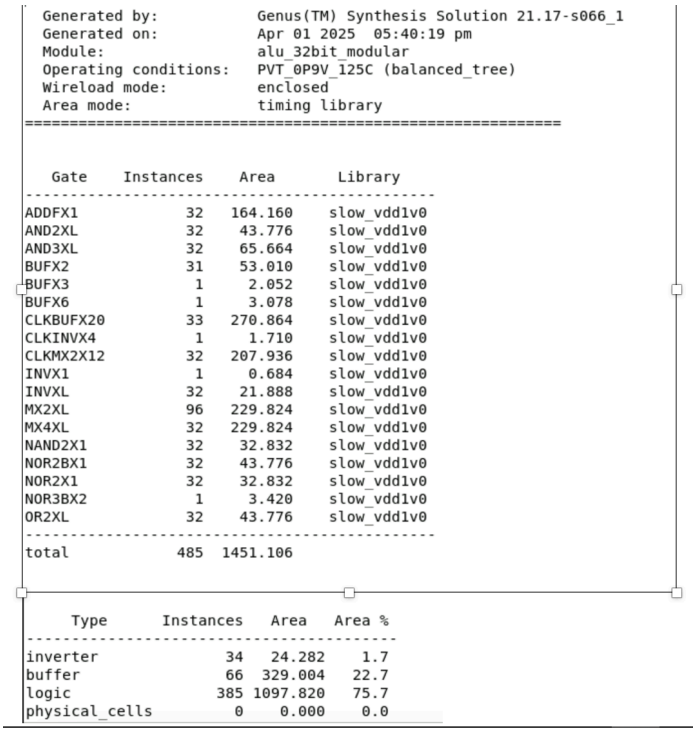
Figure 19: *Power report for the modular 32-bit ALU, showing that dynamic (switching) power is the major contributor.*

Area Report

The synthesized area of the 32-bit modular ALU totaled 1451.106 μm², composed of:

- 385 logic cells accounting for 75.7% of the total area
- 66 buffer cells occupying 22.7%
- 34 inverters using 1.7%

The area breakdown depicted in **Figure 20** highlights that the majority of the area is dedicated to logic computation, which aligns with the expected behavior of a structurally replicated design with significant interconnect logic.



```
  Generated by:           Genus(TM) Synthesis Solution 21.17-s066_1
  Generated on:           Apr 01 2025  05:40:19 pm
  Module:                 alu_32bit_modular
  Operating conditions:   PVT_0P9V_125C (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
==========================================================


   Gate    Instances    Area       Library
-------------------------------------------------
ADDFX1            32   164.160    slow_vdd1v0
AND2XL            32    43.776    slow_vdd1v0
AND3XL            32    65.664    slow_vdd1v0
BUFX2             31    53.010    slow_vdd1v0
BUFX3              1     2.052    slow_vdd1v0
BUFX6              1     3.078    slow_vdd1v0
CLKBUFX20         33   270.864    slow_vdd1v0
CLKINVX4           1     1.710    slow_vdd1v0
CLKMX2X12         32   207.936    slow_vdd1v0
INVX1              1     0.684    slow_vdd1v0
INVXL             32    21.888    slow_vdd1v0
MX2XL             96   229.824    slow_vdd1v0
MX4XL             32   229.824    slow_vdd1v0
NAND2X1           32    32.832    slow_vdd1v0
NOR2BX1           32    43.776    slow_vdd1v0
NOR2X1            32    32.832    slow_vdd1v0
NOR3BX2            1     3.420    slow_vdd1v0
OR2XL             32    43.776    slow_vdd1v0
-------------------------------------------------
total            485  1451.106


     Type      Instances   Area   Area %
-----------------------------------------
inverter              34   24.282    1.7
buffer                66  329.004   22.7
logic                385 1097.820   75.7
physical_cells         0    0.000    0.0
```

Figure 20*: Synthesized layout view of the 32-bit modular ALU showing cascading 1-bit slices in a ripple-carry arrangement.*

Gate-Level Schematic

The 32-bit modular ALU was constructed by cascading thirty-two 1-bit ALU slices, each handling one bit of the operands and contributing to the overall result F[31:0]. The design seen in **Figure 21** uses a ripple-carry configuration, where carry-out from each slice is connected to the carry-in of the next higher bit slice. Due to this chaining, the design exhibits a straightforward, regular structure but incurs a longer critical path, which affects timing and area utilization.

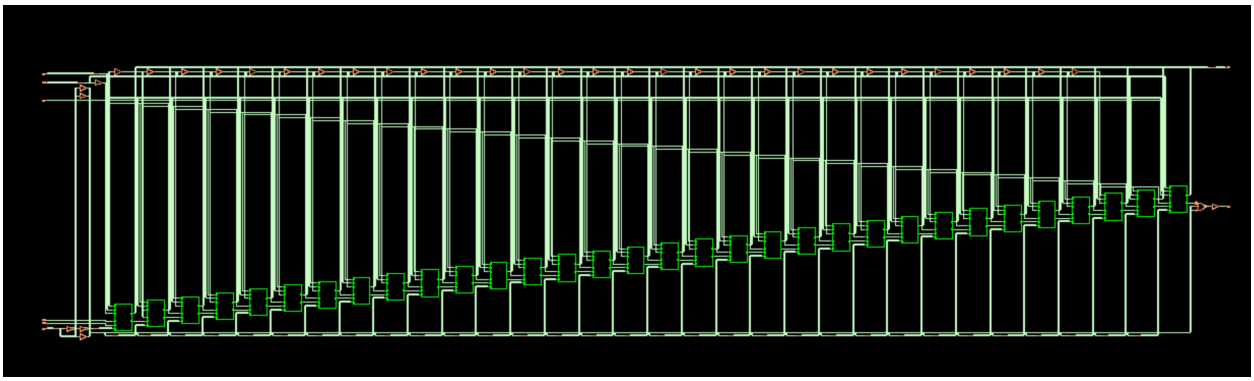


Figure 21*: Area report generated by Cadence Genus detailing gate-level utilization across inverter, buffer, and logic instances in the 32-bit modular ALU.*

5.2.3 32-bit behavioral ALU

The behavioral implementation of the 32-bit ALU was synthesized using Cadence Genus, and the resulting metrics for power, area, and gate-level structure are reported here. Unlike the modular version, the behavioral ALU is described using high-level Verilog constructs, allowing the synthesis tool more flexibility in optimization.

Power Report

```
Instance: /alu_32bit_behavioral
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
      memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       logic     3.61007e-08  3.28942e-07  2.49149e-05  2.52799e-05 100.00%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     3.61007e-08  3.28942e-07  2.49149e-05  2.52799e-05 100.00%
  Percentage           0.14%        1.30%       98.56%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Figure 22: *Power report for the behavioral ALU generated by Cadence Genus, detailing logic switching and internal power contributions.*

As shown in **Figure 22**, the total dynamic power consumed by the behavioral ALU is approximately 25.2799 µW, broken down as follows:

- Switching Power dominates the profile at 98.56% of total consumption.
- Internal Power accounts for 1.30%, while Leakage Power is minimal at 0.14%.
- All power is attributed to logic elements, with no contribution from memory, clock, or I/O pads.

Area Report

The area analysis (shown in **Figure 23**) reveals a total utilization of 975.726 µm², with:

- Logic gates occupying the majority at 64.2% (626.202 µm² across 267 instances).
- Buffers at 33.4% (325.584 µm² across 65 instances).
- Inverters at 2.5% (23.940 µm² across 35 instances).

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.17-s066_1
  Generated on:           Apr 01 2025  01:53:19 am
  Module:                 alu_32bit_behavioral
  Operating conditions:   PVT_0P9V_125C (balanced_tree)
  Wireload mode:          enclosed
  Area mode:              timing library
============================================================
```

Gate	Instances	Area	Library
ADDFX1	32	164.160	slow_vdd1v0
AND2X1	2	2.736	slow_vdd1v0
AO21X1	32	76.608	slow_vdd1v0
AOI22X1	32	65.664	slow_vdd1v0
AOI32X1	32	76.608	slow_vdd1v0
BUFX2	32	54.720	slow_vdd1v0
CLKAND2X8	1	5.814	slow_vdd1v0
CLKBUFX20	33	270.864	slow_vdd1v0
INVX1	35	23.940	slow_vdd1v0
MX2XL	32	76.608	slow_vdd1v0
NAND2X1	1	1.026	slow_vdd1v0
NAND2XL	32	32.832	slow_vdd1v0
NOR2BX2	1	2.736	slow_vdd1v0
NOR2X1	1	1.026	slow_vdd1v0
NOR2X4	1	3.078	slow_vdd1v0
OA21X2	1	3.078	slow_vdd1v0
OAI211X1	64	109.440	slow_vdd1v0
OR2X1	1	1.368	slow_vdd1v0
OR2X2	2	3.420	slow_vdd1v0
total	367	975.726	

Type	Instances	Area	Area %
inverter	35	23.940	2.5
buffer	65	325.584	33.4
logic	267	626.202	64.2
physical_cells	0	0.000	0.0
total	367	975.726	100.0

Figure 23: *Area breakdown of the behavioral ALU showing contributions from inverters, buffers, and logic gates across 367 instances.*

Gate-Level Schematic

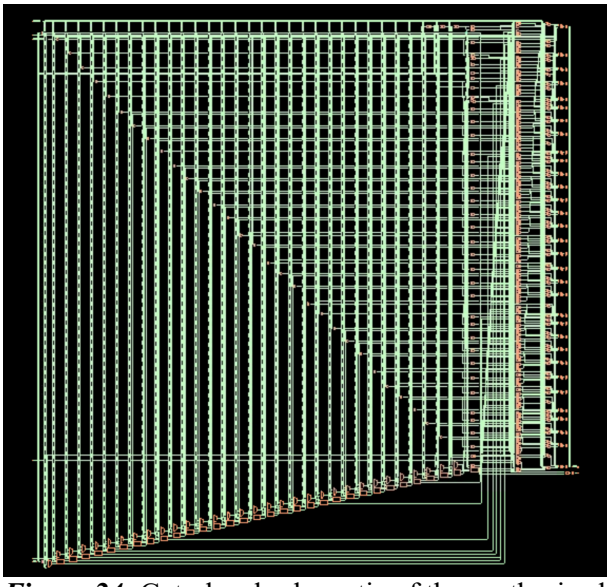


Figure 24: Gate-level schematic of the synthesized 32-bit behavioral ALU. The layout is more compact, and irregular compared to the modular design due to synthesis-level optimizations.

The corresponding gate-level layout, depicted in **Figure 24**, shows a vertically stacked and highly parallel structure. Each column corresponds to a bit-slice of the operation, and the design appears optimized for compactness and minimum routing complexity. Compared to the modular layout, this one demonstrates less regularity but better area efficiency due to reduced overhead from structural instantiations.

6. Physical Design

Having completed the synthesis stage for all ALU designs, the next step involves transitioning to physical implementation using the Place and Route (PnR) process in Cadence Innovus. This section details how each synthesized netlist is transformed into a physical layout, verified through Design Rule Check (DRC) and connectivity checks, and evaluated in terms of area and timing performance.

6.1 Layout of 1-bit ALU

We begin with the physical layout of the 1-bit ALU slice, the core building block of the modular ALU architecture. This subsection presents the Innovus-generated layout, verification results from DRC and LVS, and an analysis of the area footprint and estimated critical path delay.

Innovus PnR screenshots

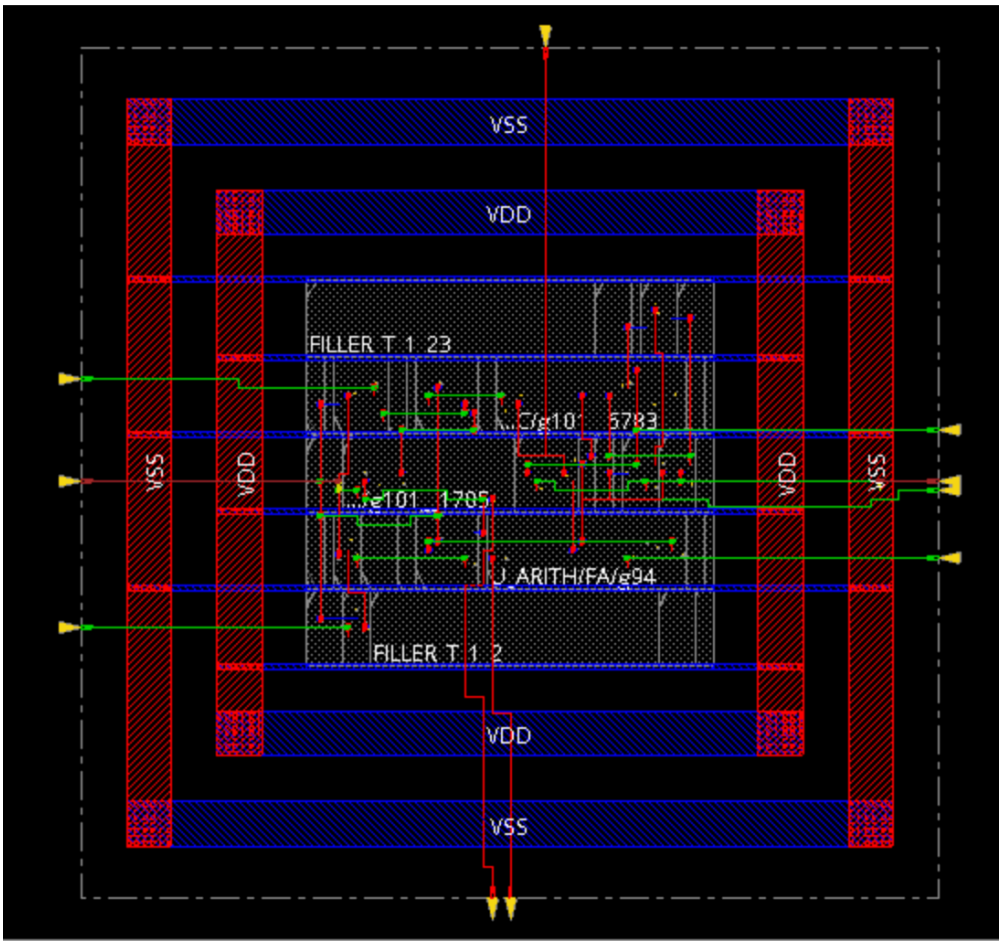


Figure 25: *Physical Layout View of the 1-bit ALU Design.*

Figure 25 shows the post-layout implementation of the 1-bit ALU, including standard cell placement, metal routing, power (VDD/VSS) rails, and signal connections. The layout has been optimized for area and timing, ensuring functional correctness and manufacturability.

DRC and connectivity check results

```
0
innovus 3> innovus 3> **ERROR: (IMPSYT-16284):  Enter a filler cell name.
innovus 3> **WARN: (IMPSP-5217):       addFiller command is running on a postRoute database. It is recommended to be followed by ecoRoute -target command to mak
e the DRC clean.
Type 'man IMPSP-5217' for more detail.
*INFO: Adding fillers to top-module.
*INFO:   Added 0 filler inst  (cell FILL64 / prefix FILLER).
*INFO:   Added 2 filler insts (cell FILL32 / prefix FILLER).
*INFO:   Added 0 filler inst  (cell FILL16 / prefix FILLER).
*INFO:   Added 0 filler inst  (cell FILL8 / prefix FILLER).
*INFO:   Added 6 filler insts (cell FILL4 / prefix FILLER).
*INFO:   Added 10 filler insts (cell FILL2 / prefix FILLER).
*INFO:   Added 8 filler insts (cell FILL1 / prefix FILLER).
*INFO: Total 26 filler insts added - prefix FILLER (CPU: 0:00:00.0).
For 26 new insts, innovus 3> #-check_ndr_spacing auto               # enums={true false auto}, default=auto,  user setting
#-check_same_via_cell true              # bool, default=false, user setting
#-exclude_pg_net true                   # bool, default=false, user setting
#-report alu_1bit.drc.rpt               # string, default="", user setting
 *** Starting Verify DRC (MEM: 2812.1) ***

  VERIFY DRC ..... Starting Verification
  VERIFY DRC ..... Initializing
  VERIFY DRC ..... Deleting Existing Violations
  VERIFY DRC ..... Creating Sub-Areas
  VERIFY DRC ..... Using new threading
  VERIFY DRC ..... Sub-Area: {0.000 0.000 19.000 18.810} 1 of 1
  VERIFY DRC ..... Sub-Area : 1 complete 0 Viols.

  Verification Complete : 0 Viols.

 *** End Verify DRC (CPU: 0:00:00.0  ELAPSED TIME: 0.00  MEM: 264.1M) ***
```

Figure 26: *Design Rule Check (DRC) Verification Report for the 1-bit ALU Design.*

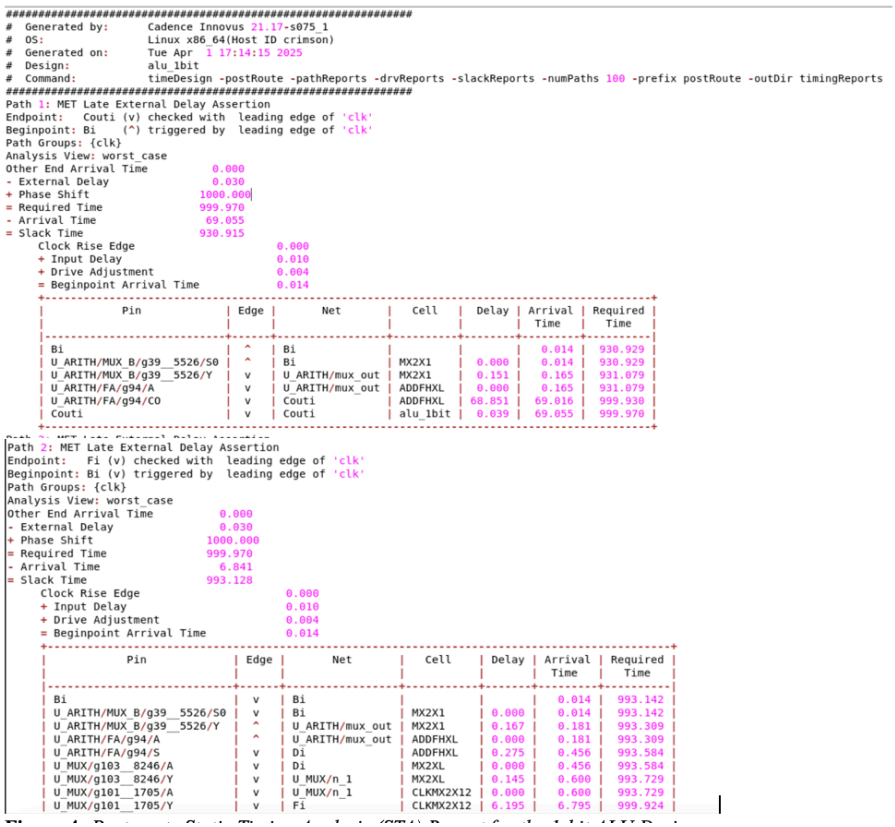
In Figure 26, the DRC verification log can be seen. The verification process completed with zero violations, confirming that the post-layout design of the 1-bit ALU is DRC clean and compliant with the foundry's design rules.

```
innovus 3> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Apr  1 17:24:37 2025

Design Name: alu_1bit
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (19.0000, 18.8100)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Apr  1 17:24:37 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Figure 27: *Connectivity Verification Report for the 1-bit ALU Design.*

Figure 27 shows the connectivity verification summary of the 1-bit ALU design. The report confirms that no connectivity violations or warnings were detected, ensuring that all nets in the design are properly connected and there are no floating or unconnected signals.

Final layout estimated max speed



```
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID crimson)
#  Generated on:      Tue Apr  1 17:14:15 2025
#  Design:            alu_1bit
#  Command:           timeDesign -postRoute -pathReports -drvReports -slackReports -numPaths 100 -prefix postRoute -outDir timingReports
Path 1: MET Late External Delay Assertion
Endpoint:   Couti (v) checked with  leading edge of 'clk'
Beginpoint: Bi    (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay               0.030
+ Phase Shift               1000.000
= Required Time             999.970
- Arrival Time               69.055
= Slack Time                930.915
     Clock Rise Edge                     0.000
     + Input Delay                       0.010
     + Drive Adjustment                  0.004
     = Beginpoint Arrival Time           0.014
     +-------------------------------------------------------------------------------------+
     |            Pin             | Edge |       Net       |   Cell   | Delay | Arrival | Required |
     |                            |      |                 |          |       |  Time   |   Time   |
     |----------------------------+------+-----------------+----------+-------+---------+----------|
     | Bi                         |  ^   | Bi              |          |       |   0.014 |  930.929 |
     | U_ARITH/MUX_B/g39__5526/S0 |  ^   | Bi              | MX2X1    | 0.000 |   0.014 |  930.929 |
     | U_ARITH/MUX_B/g39__5526/Y  |  v   | U_ARITH/mux_out | MX2X1    | 0.151 |   0.165 |  931.079 |
     | U_ARITH/FA/g94/A           |  v   | U_ARITH/mux_out | ADDFHXL  | 0.000 |   0.165 |  931.079 |
     | U_ARITH/FA/g94/CO          |  v   | Couti           | ADDFHXL  | 68.851 |  69.016 |  999.930 |
     | Couti                      |  v   | Couti           | alu_1bit | 0.039 |  69.055 |  999.970 |
     +-------------------------------------------------------------------------------------+
Path 2: MET Late External Delay Assertion
Endpoint:   Fi (v) checked with  leading edge of 'clk'
Beginpoint: Bi (v) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay               0.030
+ Phase Shift               1000.000
= Required Time             999.970
- Arrival Time                6.841
= Slack Time                993.128
     Clock Rise Edge                     0.000
     + Input Delay                       0.010
     + Drive Adjustment                  0.004
     = Beginpoint Arrival Time           0.014
     +-------------------------------------------------------------------------------------+
     |            Pin             | Edge |       Net       |   Cell   | Delay | Arrival | Required |
     |                            |      |                 |          |       |  Time   |   Time   |
     |----------------------------+------+-----------------+----------+-------+---------+----------|
     | Bi                         |  v   | Bi              |          |       |   0.014 |  993.142 |
     | U_ARITH/MUX_B/g39__5526/S0 |  v   | Bi              | MX2X1    | 0.000 |   0.014 |  993.142 |
     | U_ARITH/MUX_B/g39__5526/Y  |  ^   | U_ARITH/mux_out | MX2X1    | 0.167 |   0.181 |  993.309 |
     | U_ARITH/FA/g94/A           |  ^   | U_ARITH/mux_out | ADDFHXL  | 0.000 |   0.181 |  993.309 |
     | U_ARITH/FA/g94/S           |  v   | Di              | ADDFHXL  | 0.275 |   0.456 |  993.584 |
     | U_MUX/g103__8246/A         |  v   | Di              | MX2XL    | 0.000 |   0.456 |  993.584 |
     | U_MUX/g103__8246/Y         |  v   | U_MUX/n_1       | MX2XL    | 0.145 |   0.600 |  993.729 |
     | U_MUX/g101__1705/A         |  v   | U_MUX/n_1       | CLKMX2X12 | 0.000 |  0.600 |  993.729 |
     | U_MUX/g101__1705/Y         |  v   | Fi              | CLKMX2X12 | 6.195 |  6.795 |  999.924 |
```

Figure 28: *Post-route Static Timing Analysis (STA) Report for the 1-bit ALU Design.*

Figure 28 shows two critical timing paths identified by Cadence Innovus timing analysis tool. Path 1 traces the delay from input Bi to output Couti, with a total delay of 69.055 ns and positive slack of 930.915 ns. Path 2 traces the delay also from input Bi to output Fi, with a total delay of 6.841 ns and slack of 993.1280 ns. These results confirm that the timing constraints are met, and the circuit operates reliably at the target clock period without timing violations.

Estimation of highest speed of the 1-bit ALU:

Estimated Maximum Speed: The timing analysis shows the worst-case slack time is positive, indicating timing closure and meeting time constraint.

The worst arrival time corresponds to the critical path delay of the circuit. In my 1-bit ALU circuit, the critical path is path 1 in **Figure 28**, and it starts from the second bit of the 4-bit control signal sel (sel[1]) to the carry out bit (Couti). The critical path delay is measured to be t = 69.053 ns. Based on this information, we can use the following formula:

Maximum Operating Frequency: f_{max} = 1 / Critical Path Delay

to calculate the circuit's highest speed. And for my the 1-bit ALU between our hands, the circuit's highest speed is given by $f_{max} = 1 / (69.055 \times 10^{-9}$ seconds) = 14.48 MHz

> Therefore, the circuit can operate at a maximum speed of approximately **14.48 MHz** under typical conditions.

6.2 Layout of 32-bit ALUs

Following the detailed examination of the 1-bit ALU layout, we extend the physical design process to the complete 32-bit implementations. Section 6.2 showcases the layouts of both the modular and behavioral ALUs, comparing their area utilization, timing performance, and compliance with DRC rules, offering insight into the trade-offs between structural and behavioral design methodologies.

6.2.1 Modular ALU layout

This subsection presents the Innovus-generated layout, verification results from DRC and LVS, and an analysis of the area footprint and estimated critical path delay.

Innovus PnR screenshots

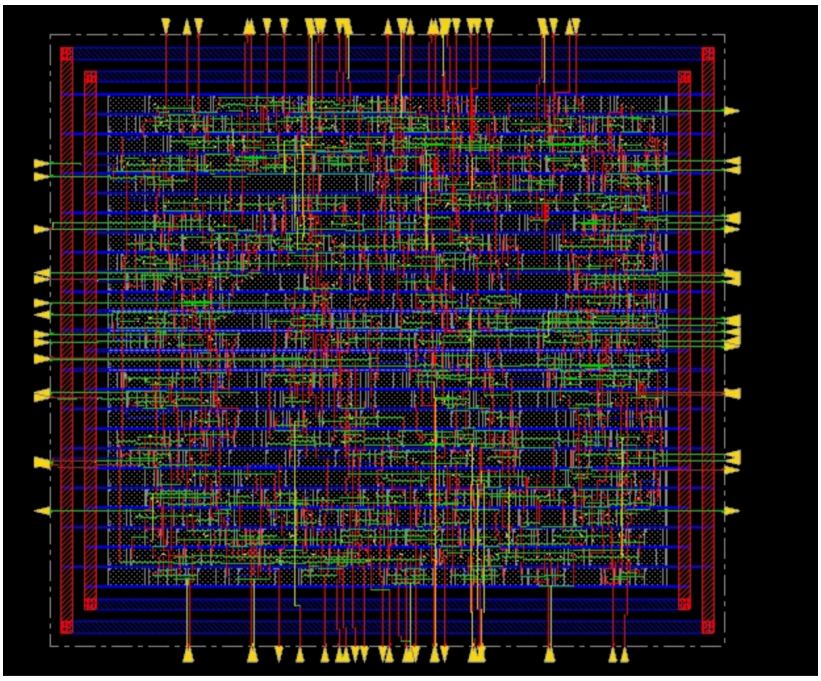


Figure 29: *Final Layout View of the 32-bit Modular ALU*

Figure 29 shows the final physical layout of the 32-bit modular ALU after place and route, as displayed in the Cadence Innovus tool. The layout includes the standard cell placement, routing of interconnects, and filler cells to ensure proper utilization of the layout area. The red and blue metal layers represent the vertical and horizontal routing tracks, respectively, while the yellow markers denote the input and output pins of the ALU. The dense routing and placement of standard cells reflect the complexity of the design, with clear power (VDD) and ground (VSS) rails framing the boundaries of the layout. The successful generation of this layout confirms that the ALU has been physically implemented, routed, and optimized, ready for fabrication or further analysis.

DRC and connectivity check results

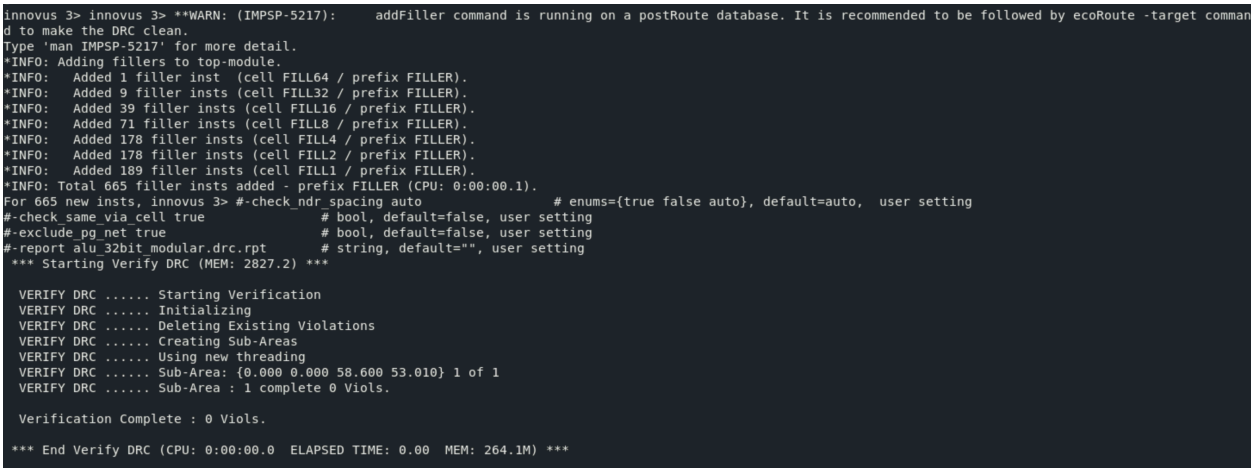


Figure 30: *Design Rule Check (DRC) Report after Place and Route of 32-bit Modular ALU*

Figure 30 shows the DRC (Design Rule Check) report generated after the place and route process of the 32-bit modular ALU. The DRC verification process confirms that there are zero violations in the final layout, meaning that the design complies fully with the foundry's manufacturing rules. This indicates that the layout is clean, manufacturable, and ready for tape-out or further verification steps.

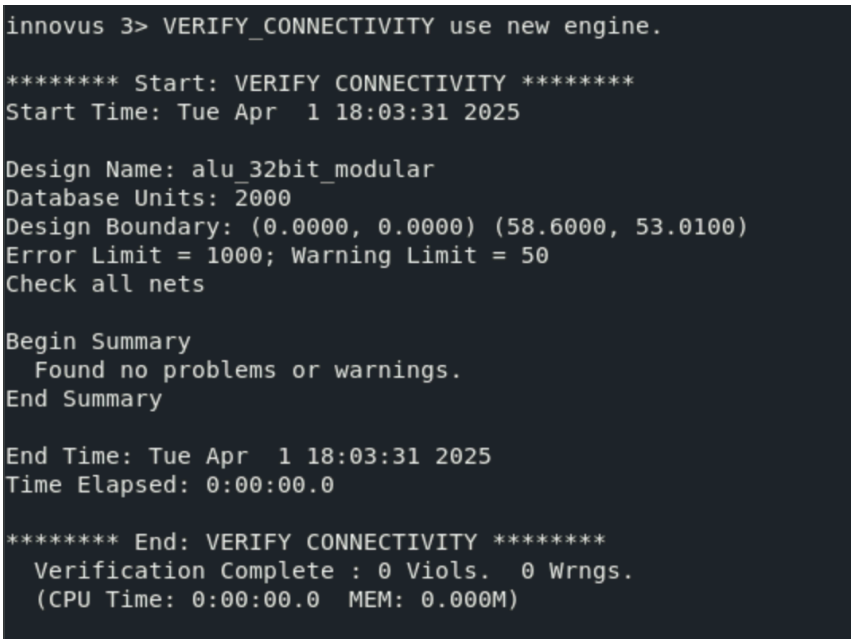


Figure 31: *Connectivity Check Report for the 32-bit Modular ALU Design*

Figure 31 presents the results of the VERIFY CONNECTIVITY check performed after the place and route of the 32-bit modular ALU. The verification process confirms that there are no connectivity errors or warnings within the design. This indicates a successful place and route phase with valid net connections across the entire ALU structure.

Final layout estimated max speed

```
#####################################################################
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID crimson)
#  Generated on:      Tue Apr  1 17:53:30 2025
#  Design:            alu_32bit_modular
#  Command:           timeDesign -postRoute -pathReports -drvReports -slackReports -
#####################################################################
Path 1: MET Late External Delay Assertion
Endpoint:   Cout   (^) checked with  leading edge of 'clk'
Beginpoint: sel[1] (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay                0.030
+ Phase Shift                1000.000
= Required Time               999.970
- Arrival Time                 86.157
= Slack Time                  913.812
     Clock Rise Edge                      0.000
     + Input Delay                        0.010
     + Drive Adjustment                   0.112
     = Beginpoint Arrival Time            0.122
     +--------------------------------------------------------------------------------
Path 2: MET Late External Delay Assertion
Endpoint:   F[31]  (v) checked with  leading edge of 'clk'
Beginpoint: sel[1] (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay                0.030
+ Phase Shift                1000.000
= Required Time               999.970
- Arrival Time                 13.318
= Slack Time                  986.652
     Clock Rise Edge                      0.000
     + Input Delay                        0.010
     + Drive Adjustment                   0.112
     = Beginpoint Arrival Time            0.122
     +--------------------------------------------------------------------------------
Path 33: MET Late External Delay Assertion
Endpoint:   F[0]   (v) checked with  leading edge of 'clk'
Beginpoint: sel[1] (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay                0.030
+ Phase Shift                1000.000
= Required Time               999.970
- Arrival Time                  7.067
= Slack Time                  992.903
     Clock Rise Edge                      0.000
     + Input Delay                        0.010
     + Drive Adjustment                   0.112
```

Figure 32: *Post-Route Static Timing Analysis Report for the 32-bit Modular ALU*

The timing analysis report shown in **Figure 32** produced by Cadence Innovus of the 32-bit modular ALU reveal important insights regarding its performance. It identifies the critical path of the circuit as the carry

propagation chain that begins from the selection signal sel[1] and ends at the final carry-out signal Cout. The total delay along this path is approximately 86.157 nanoseconds (ns), making it the worst-case delay of the design and, consequently, the limiting factor in determining the maximum clock frequency of the ALU. The critical path has a positive slack of 913.812 ns, confirming that the design comfortably meets the specified timing constraint. This critical path behavior is characteristic of ripple-carry architectures, where each bit's carry-out signal is forwarded to the next bit's carry-in, creating a dependency chain that spans all 32 bits. The progressive propagation of carry through every bit slice results in cumulative delay.

A clear trend is observed when analyzing the delay of output signals across the ALU. The delay for computing F[0] is minimal because it depends only on the primary inputs A[0], B[0], and the external carry-in, without any internal carry dependency. In contrast, the delay for any intermediate output F[0+k], where $k \in [1,31]$ increases proportionally with the number of preceding bit slices because its carry-in relies on the computed carry of all previous bits. The longest delay is observed at F[31], which depends on the propagation of the carry signal through all thirty-one lower-order bits. Therefore, while F[0] is computed almost instantaneously, F[31] experiences the accumulated delay of the entire chain, leading to a significant difference in arrival times.

Estimation of highest speed of the 32-bit ALU modular design:

Estimated Maximum Speed: The worst arrival time corresponds to the critical path delay of the circuit. In my 32-bit ALU modular circuit design, the critical path is path 1 in **Figure 32**, and it starts from the second bit of the 4-bit control signal sel (sel[1]) to the carry out (Cout). The critical path delay is measured to be t = 165.300 ns. Based on this information, we can use the following formula:

Maximum Operating Frequency: f_{max} = 1 / Critical Path Delay

to calculate the circuit's highest speed. And for my the 1-bit ALU between our hands, the circuit's highest speed is given by $f_{max} = 1 / (86.157 \times 10^{-9}$ seconds) = **11.61 MHz**.

> Therefore, the circuit can operate at a maximum speed of approximately **11.61 MHz** under typical conditions.

6.2.2 Behavioral ALU layout

This subsection presents the Innovus-generated layout for the 32-bit ALU behavioral implementation, verification results from DRC and LVS, and an analysis of the area footprint and estimated critical path delay.

Innovus PnR screenshots

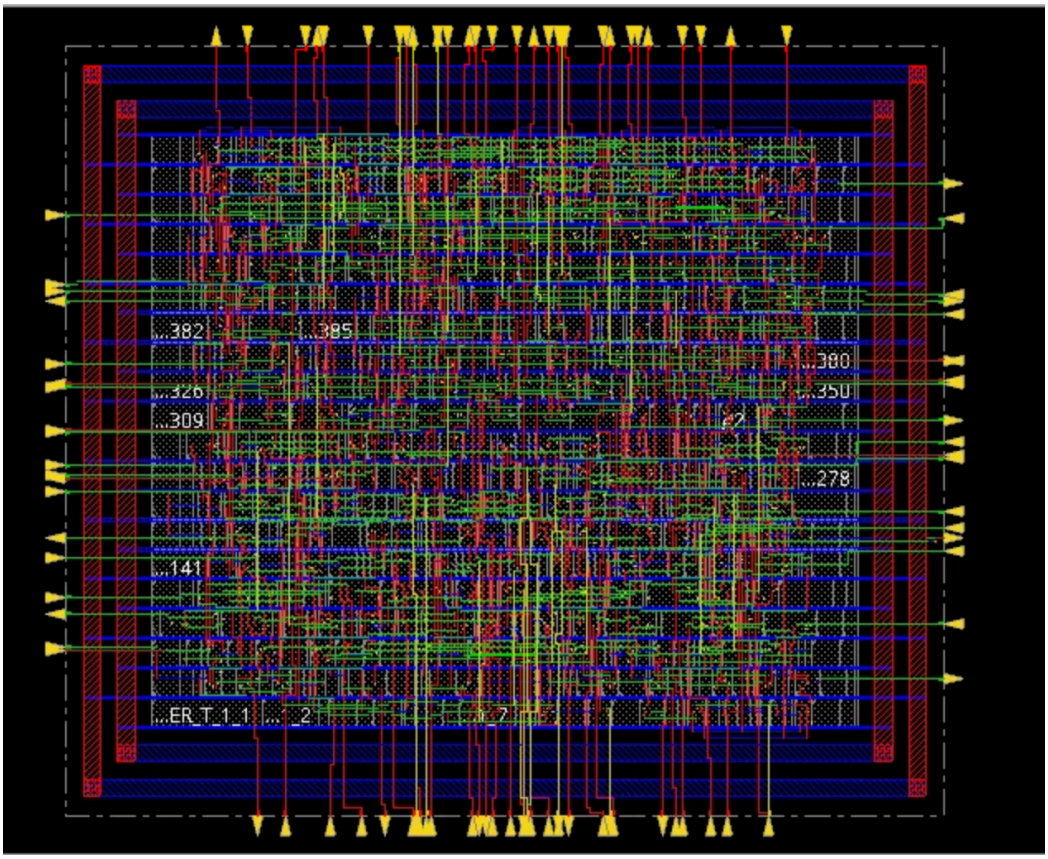


Figure 33: *Final Post-Route Layout of the 32-bit Behavioral ALU Design in Cadence Innovus*

Figure 33 shows the completed physical implementation of the 32-bit behavioral ALU design after place-and-route using Cadence Innovus. The layout includes all standard cells, routing of input/output signals, and power rails (VDD and VSS).

DRC and connectivity check results

```
innovus 3> innovus 3> **WARN: (IMPSP-5217):     addFiller command is running on a postRoute database. It is recommended to be followed by ecoRoute -target comman
d to make the DRC clean.
Type 'man IMPSP-5217' for more detail.
*INFO: Adding fillers to top-module.
*INFO:   Added 0 filler inst  (cell FILL64 / prefix FILLER).
*INFO:   Added 1 filler inst  (cell FILL32 / prefix FILLER).
*INFO:   Added 15 filler insts (cell FILL16 / prefix FILLER).
*INFO:   Added 96 filler insts (cell FILL8 / prefix FILLER).
*INFO:   Added 172 filler insts (cell FILL4 / prefix FILLER).
*INFO:   Added 145 filler insts (cell FILL2 / prefix FILLER).
*INFO:   Added 161 filler insts (cell FILL1 / prefix FILLER).
*INFO: Total 590 filler insts added - prefix FILLER (CPU: 0:00:00.1).
For 590 new insts, innovus 3> #-check_ndr_spacing auto              # enums={true false auto}, default=auto,  user setting
#-check_same_via_cell true              # bool, default=false, user setting
#-exclude_pg_net true                   # bool, default=false, user setting
#-report alu_32bit_behavioral.drc.rpt    # string, default="", user setting
 *** Starting Verify DRC (MEM: 2824.7) ***

  VERIFY DRC ..... Starting Verification
  VERIFY DRC ..... Initializing
  VERIFY DRC ..... Deleting Existing Violations
  VERIFY DRC ..... Creating Sub-Areas
  VERIFY DRC ..... Using new threading
  VERIFY DRC ..... Sub-Area: {0.000 0.000 50.800 44.460} 1 of 1
  VERIFY DRC ..... Sub-Area : 1 complete 0 Viols.

  Verification Complete : 0 Viols.

 *** End Verify DRC (CPU: 0:00:00.0  ELAPSED TIME: 0.00  MEM: 264.1M) ***
```

Figure 34: *Design Rule Check (DRC) Report for the 32-bit Behavioral ALU Design*

Figure 34 shows the DRC verification results of the 32-bit behavioral ALU design after the place-and-

route process in Cadence Innovus. The message **"Verification Complete: 0 Viols."** indicates that the final layout is DRC-clean, meaning it meets all manufacturing constraints and is ready for tape-out.

```
innovus 3> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Apr  1 13:43:44 2025

Design Name: alu_32bit_behavioral
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (50.8000, 44.4600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Apr  1 13:43:44 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Figure 35: *Connectivity Verification Report for the 32-bit Behavioral ALU Design*

Figure 35 shows the connectivity check results for the final layout of the 32-bit behavioral ALU design in Cadence Innovus. The report indicates **0 Violations and 0 Warnings**, ensuring that the interconnections between the ALU core and the pad frame are complete and accurate. This is a mandatory verification step before generating the final GDSII for fabrication.

Final layout estimated max speed

```
#############################################################
#  Generated by:      Cadence Innovus 21.17-s075_1
#  OS:                Linux x86_64(Host ID crimson)
#  Generated on:      Tue Apr  1 02:17:54 2025
#  Design:            alu_32bit_behavioral
#  Command:           timeDesign -postRoute -pathReports -drvReports -slackReports -numPaths 100 -
prefix postRoute -outDir timingReports
#############################################################
Path 1: MET Late External Delay Assertion
Endpoint:   F[31]  (v) checked with  leading edge of 'clk'
Beginpoint: sel[0] (^) triggered by  leading edge of 'clk'
Path Groups: {clk}
Analysis View: worst_case
Other End Arrival Time          0.000
- External Delay               0.030
+ Phase Shift                1000.000
= Required Time              999.970
- Arrival Time                61.008
= Slack Time                 938.962
     Clock Rise Edge                     0.000
     + Input Delay                       0.010
     + Drive Adjustment                  0.059
     = Beginpoint Arrival Time           0.069

+-------------------------------------------------------------------------------------------------
```

Figure 36: *Post-Route Static Timing Analysis (STA) Report for Behavioral 32-bit ALU Design*

Figure 36 presents the STA timing reports for the post-route analysis of the behavioral 32-bit ALU design. The first path identifies the critical path delay ending at output F[31], which exhibits an arrival

time of approximately 61.008 ns with a slack of 938.962 ns. This indicates that the F[31] output meets timing constraints with ample slack.

Estimation of highest speed of the 32-bit ALU behavioral design:

Estimated Maximum Speed: The worst arrival time corresponds to the critical path delay of the circuit. In my 32-bit ALU behavioral circuit design, the critical path is path 1 in **Figure 36**, and it starts from the first bit of the 4-bit control signal sel (sel[0]) to the 31st bit of the output (F[31]). The critical path delay is measured to be t = 61.008 ns. Based on this information, we can use the following formula:

Maximum Operating Frequency: f_{max} = 1 / Critical Path Delay

to calculate the circuit's highest speed. And for my the 1-bit ALU between our hands, the circuit's highest speed is given by $f_{max} = 1 / (61.008 \times 10^{-9}$ seconds) = 16.39 MHz

> Therefore, the circuit can operate at a maximum speed of approximately **16.39 MHz** under typical conditions.

6.2.3 Comparison of area, power and speed between ALU modular and ALU behavioral

To evaluate the efficiency of both 32-bit ALU implementations, we compared the modular and behavioral designs across three primary metrics: layout area, total power consumption, and maximum operating frequency. These metrics were obtained post-synthesis and place-and-route using Cadence Genus and Innovus.

Area

- Modular ALU:
  - ➔ Total area: 1451.106 µm²
  - ➔ Logic utilization: 75.7% of total area
- Behavioral ALU:
  - ➔ Total area: 975.726 µm²
  - ➔ Logic utilization: 64.2% of total area

The behavioral design achieved a 32.8% reduction in layout area compared to the modular design. This is primarily due to RTL-level optimizations allowed by behavioral synthesis, which reduces interconnect complexity and gate count.

Power

- Modular ALU:
  - ➔ Total power: 2.55943e–05 W
  - ➔ Switching power dominates at 98.09%
- Behavioral ALU:
  - ➔ Total power: 2.52799e–05 W
  - ➔ Switching power also dominates at 98.56%

The power consumption of both designs is nearly identical. However, the behavioral ALU uses slightly less power (by ~1.2%), suggesting minor improvements in signal toggling and reduced capacitive loads due to a more compact layout.

Maximum Speed (Post-Layout)

- Modular ALU: Estimated maximum frequency: 11.61 MHz
- Behavioral ALU: Estimated maximum frequency: 16.39 MHz

The behavioral design operates at a 41% higher frequency. This can be attributed to the reduced combinational path delays in behavioral coding and the absence of long carry chains present in the modular ripple-carry structure.

Summary Table: Modular vs Behavioral 32-bit ALU

Metric	**Modular ALU**	**Behavioral ALU**	**Improvement (Behavioral)**
Total Area (µm²)	1451.106	975.726	32.8%
Total Power (W)	2.55943e–05	2.52799e–05	1.2%
Max Frequency	11.61 MHz	16.39 MHz	41.2%

Table 9: *Summary of area, power, and speed comparison between the 32-bit modular and behavioral ALU designs. The behavioral ALU achieves a 32.8% reduction in area, 1.2% lower power consumption, and 41.2% higher maximum operating frequency compared to the modular implementation.*

The behavioral ALU design outperforms the modular version in all key metrics: smaller area, lower power, and higher speed. These gains make it a better candidate for integration into the final chip design, particularly for applications where performance and area optimization are critical.

8. Final Chip Integration

With the functional verification and synthesis of the 1-bit and 32-bit ALU designs complete, the final phase involved integrating the core design into a complete chip using a pad frame. This step prepares the design for fabrication by adding I/O pads, power/ground rings, and ensuring robust layout verification. The integration process was carried out using Cadence Innovus, which enabled the placement of the core ALU block within a predefined pad frame and subsequent execution of physical design rule checks and layout-versus-schematic (LVS) verification.

Core layout inserted into pad frame

Following the completion of the individual ALU designs, the final step involved integrating the synthesized and placed-and-routed core into a standard pad frame using Cadence Innovus.

This integration included:

1. Pad frame selection: A predefined pad frame was chosen to accommodate all I/Os of the 32-bit ALU.
2. Core-to-pad pin alignment: The I/O pins of the ALU core were correctly mapped to the corresponding pad cells.
3. Core placement: The 32-bit ALU core was positioned within the pad ring, and routing was completed to connect all core outputs/inputs to the pad frame.
4. Power ring and routing: VDD and GND rails were extended around the core to match the pad frame’s power grid.

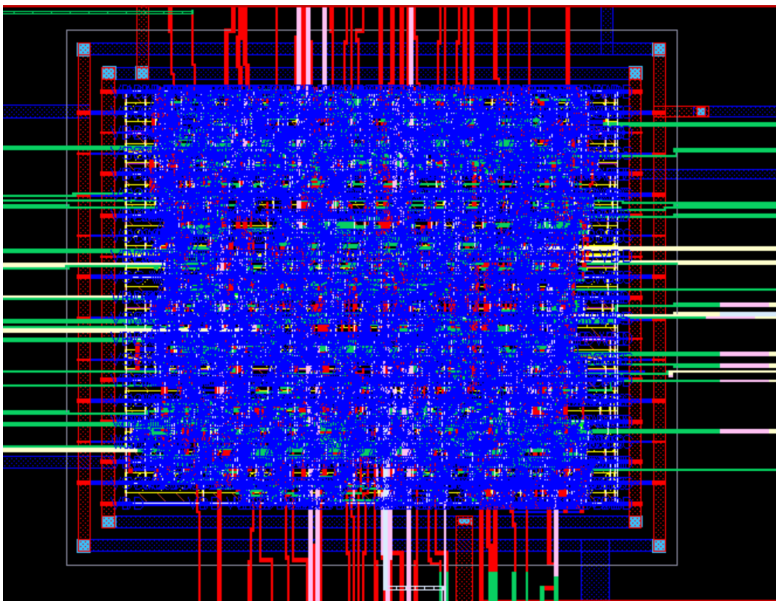


Figure 37: *Integrated layout of the 32-bit ALU core within the pad frame, showing metal layers, power routing, and global signal connections after full place and route (PnR).*

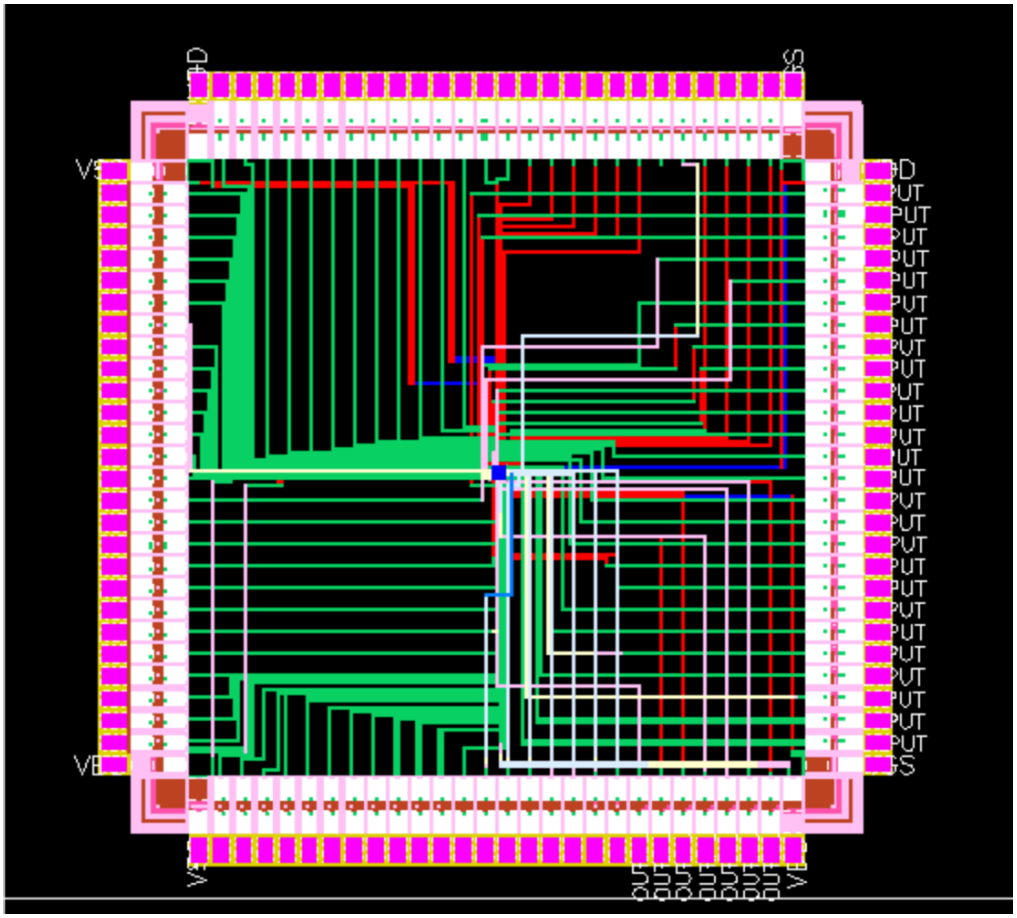


Figure 38: *Annotated physical layout view of the chip post-integration, highlighting the labeled pad cells (VDD, GND, and I/Os), confirming successful core-to-pad connectivity.*

DRC verification

Upon running DRC and connectivity checks in Cadence Virtuoso after inserting the synthesized ALU core into the provided pad frame, a number of DRC violations were reported (see **Figure 39**). However, a detailed inspection of these violations revealed that none of them originated from the ALU core or the integration process. Instead, all violations were tied to pre-existing issues in the third-party pad frame design, specifically in the layout and spacing of the BONDPAD structures as shown in **Figure 40**.

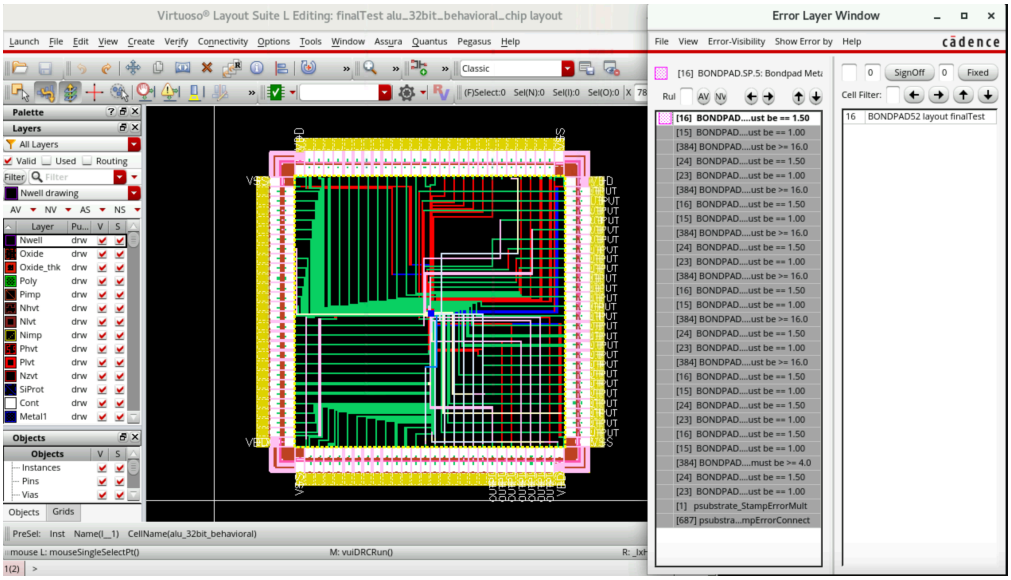


***Figure 39**: Virtuoso layout view showing DRC violations in the pad frame after integrating the ALU core. The violations listed in the Error Layer Window pertain solely to the BONDPAD design rules and not to the ALU core or its connectivity.*

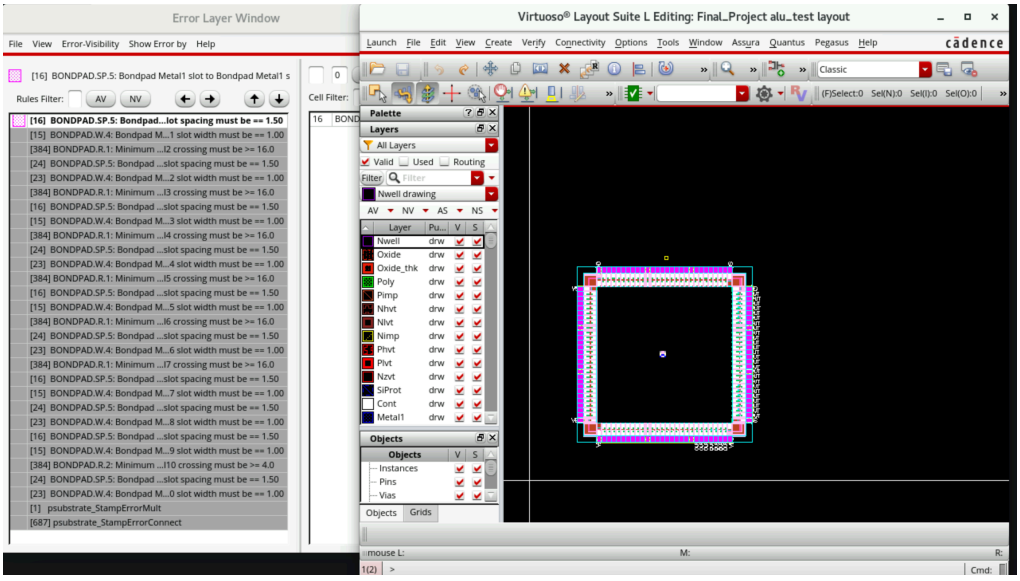


***Figure 40**: Final integrated layout of the 32-bit behavioral ALU core within the pad frame in Virtuoso. The layout passes DRC for the core, and all listed violations originate from the pad frame design.*

This distinction is critical, as the ALU core itself had already passed DRC and LVS cleanly during the synthesis and layout phase in Cadence Innovus, confirming that the design met all necessary fabrication constraints independently.

Therefore, the DRC errors observed in Virtuoso do not reflect any connectivity or placement issues in the ALU core or its integration, but rather limitations in the pad frame's compliance with certain rule decks, which are outside the scope of this project.

Total layout area

The area analysis of the alu_behavioral core (shown in **Figure 23**) reveals a total utilization of 975.726 µm², with:

- Logic gates occupying the majority at 64.2% (626.202 µm² across 267 instances).
- Buffers at 33.4% (325.584 µm² across 65 instances).
- Inverters at 2.5% (23.940 µm² across 35 instances).

This report (see **Figure 41**), generated using the *Query → Area and Density Calculator* command in Cadence Virtuoso, displays the total layout area of the behavioral 32-bit ALU after full chip integration with the pad frame. The total layout area is approximately 5,444,778.61 µm², and the core metal layer (Metal3) area is 498,236.63 µm², yielding a layout density of 0.0915.

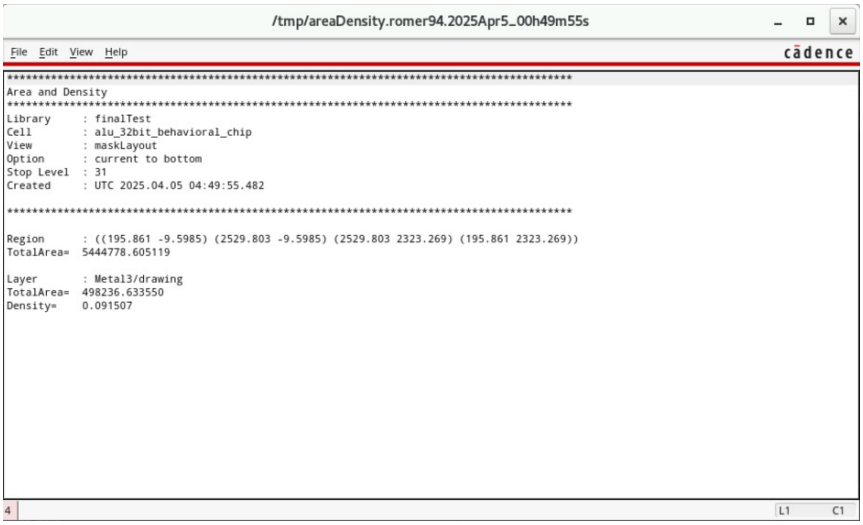


Figure 41: *Total Layout Area and Density Report of the Final Chip Integration*

Max speed using post-route timing analysis (STA result from Innovus)

As explained in **section 6.2.2,** the maximum speed of the circuit can be calculated from the maximum operating frequency of the circuit, which relies on the critical path delay time. The critical path delay was obtained from the timing report generated in the PnR Innovus stage. The following is a reminder of that calculation.

Estimation of highest speed of the 32-bit ALU behavioral design:

Estimated Maximum Speed: The worst arrival time corresponds to the critical path delay of the circuit. In my 32-bit ALU behavioral circuit design, the critical path is path 1 in **Figure 36**, and it starts from the

first bit of the 4-bit control signal sel (sel[0]) to the 31st bit of the output (F[31]). The critical path delay is measured to be t = 61.008 ns. Based on this information, we can use the following formula:

Maximum Operating Frequency: f_{max} = 1 / Critical Path Delay

to calculate the circuit's highest speed. And for my the 1-bit ALU between our hands, the circuit's highest speed is given by $f_{max} = 1 / (61.008 \times 10^{-9}$ seconds) = 16.39 MHz

> Therefore, the circuit can operate at a maximum speed of approximately **16.39 MHz** under typical conditions.

Pad Configuration Diagram

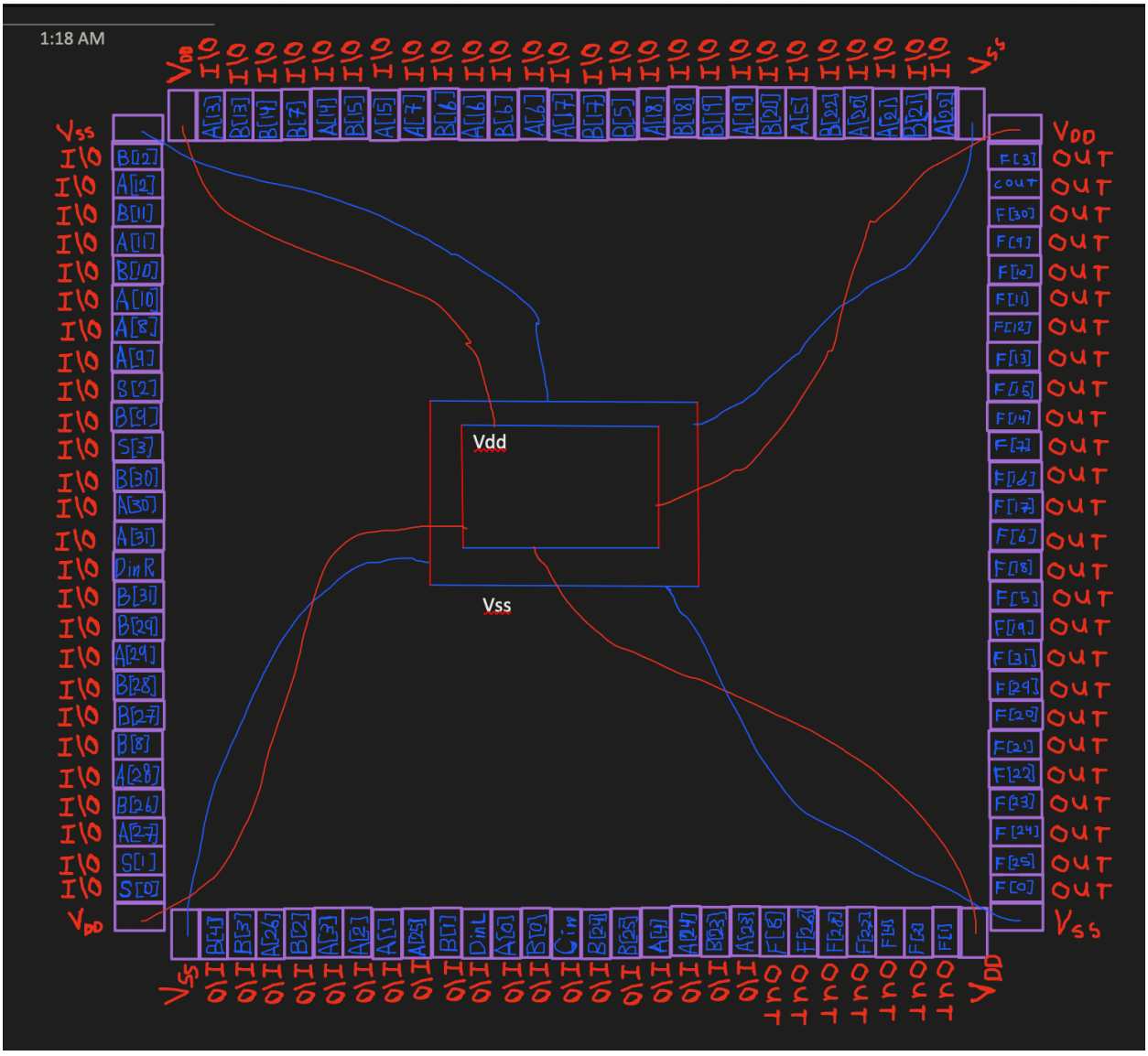


Figure 42: *Pad Configuration Diagram of the Final Chip Integration*

Figure 42 shows the pad frame configuration for the final integrated chip containing the behavioral 32-bit ALU core. Each pad cell (outlined in blue) is connected to a specific signal in the ALU design, with the corresponding internal signal name labeled inside the square structure of the pad cell (e.g., A[0], B[31], F[0], sel[3:0], Cout, Cin, etc.). This naming convention ensures traceability between I/O pads and internal signals of the ALU.

For clarity, only the wiring between VDD and VSS pads and the ALU core is shown here (red and blue lines), highlighting power distribution paths. All functional I/O connections are present in the pad frame but are omitted from the visualization to reduce clutter.

9. Appendix

Please check the PDF file titled Appendix to see the Appendix of this term project report