Bangladesh University of Engineering and Technology

Department of Electrical and Electronic Engineering

EEE 468 (July 2024)

VLSI Circuit and Design Laboratory  

**Final Project Report**

**Section: G3 Group: 03**

16-bit Carry-Lookahead Adder

**Course Instructors:**

**Nafis Sadik, Lecturer**

**Rafid Hassan Palash, Part-Time Lecturer**

**Signature of Instructor: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Academic Honesty Statement:**

**IMPORTANT! Please carefully read and sign the Academic Honesty Statement, below. Type the student ID and name, and put your signature. *You will not receive credit for this project experiment unless this statement is signed in the presence of your lab instructor.***

*“In signing this statement, We hereby certify that the work on this project is our own and that we have not copied the work of any other students (past or present), and cited all relevant sources while completing this project. We understand that if we fail to honor this agreement, We will each receive a score of ZERO for this project and be subject to failure of this course.”*

|  |  |  |
| --- | --- | --- |
| **Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Full Name: Anindya Kishore Choudhury**  **Student ID: 1906081** | **Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Full Name: MD. Toky Tazwar**  **Student ID: 1906174** |  |
| **Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Full Name: Akif Hamid**  **Student ID: 1906192** | **Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Full Name: Nusrat Jahan Anila**  **Student ID: 1906193** |  |

Table of Contents

[1 Abstract 1](#_Toc136895398)

[2 Introduction 1](#_Toc136895399)

[3 Design 1](#_Toc136895400)

[3.1 Problem Formulation (PO(b)) 1](#_Toc136895401)

[3.1.1 Identification of Scope 1](#_Toc136895402)

[3.1.2 Literature Review 1](#_Toc136895403)

[3.1.3 Formulation of Problem 1](#_Toc136895404)

[3.1.4 Analysis 1](#_Toc136895405)

[3.2 Design Method (PO(a)) 1](#_Toc136895406)

[3.3 Circuit Diagram 1](#_Toc136895407)

[3.4 Simulation Model 2](#_Toc136895408)

[3.5 CAD/Hardware Design 2](#_Toc136895409)

[3.6 PCB Design 2](#_Toc136895410)

[3.7 Full Source Code of Firmware 2](#_Toc136895411)

[3.8 2](#_Toc136895412)

[4 Implementation 2](#_Toc136895413)

[4.1 Description 2](#_Toc136895414)

[5 Design Analysis and Evaluation 3](#_Toc136895415)

[5.1 Novelty 3](#_Toc136895416)

[5.2 Design Considerations (PO(c)) 3](#_Toc136895417)

[5.2.1 Considerations to public health and safety 3](#_Toc136895418)

[5.2.2 Considerations to environment 3](#_Toc136895419)

[5.2.3 Considerations to cultural and societal needs 3](#_Toc136895420)

[5.3 Investigations (PO(d)) 3](#_Toc136895421)

[5.3.1 Design of Experiment 3](#_Toc136895422)

[5.3.2 Data Collection 3](#_Toc136895423)

[5.3.3 Results and Analysis 3](#_Toc136895424)

[5.3.4 Interpretation and Conclusions on Data 3](#_Toc136895425)

[5.4 Limitations of Tools (PO(e)) 3](#_Toc136895426)

[5.5 Impact Assessment (PO(f)) 3](#_Toc136895427)

[5.5.1 Assessment of Societal and Cultural Issues 4](#_Toc136895428)

[5.5.2 Assessment of Health and Safety Issues 4](#_Toc136895429)

[5.5.3 Assessment of Legal Issues 4](#_Toc136895430)

[5.6 Sustainability Evaluation (PO(g)) 4](#_Toc136895431)

[5.7 Ethical Issues (PO(h)) 4](#_Toc136895432)

[6 Reflection on Individual and Team work (PO(i)) 4](#_Toc136895433)

[6.1 Individual Contribution of Each Member 4](#_Toc136895434)

[6.2 Mode of TeamWork 4](#_Toc136895435)

[6.3 Diversity Statement of Team 4](#_Toc136895436)

[6.4 Log Book of Project Implementation 4](#_Toc136895437)

[7 Communication to External Stakeholders (PO(j)) 5](#_Toc136895438)

[7.1 Executive Summary 5](#_Toc136895439)

[7.2 User Manual 5](#_Toc136895440)

[7.3 Github Link 5](#_Toc136895441)

[7.4 YouTube Link 5](#_Toc136895442)

[8 Project Management and Cost Analysis (PO(k)) 5](#_Toc136895443)

[8.1 Bill of Materials 5](#_Toc136895444)

[8.2 Calculation of Per Unit Cost of Prototype 5](#_Toc136895445)

[8.3 Calculation of Per Unit Cost of Mass-Produced Unit 5](#_Toc136895446)

[8.4 Timeline of Project Implementation 5](#_Toc136895447)

[9 Future Work (PO(l)) 5](#_Toc136895448)

[10 References 5](#_Toc136895449)

# Abstract

|  |
| --- |
| The performance of digital systems heavily relies on efficient arithmetic operations, with binary addition being one of the most fundamental tasks. One of the primary challenges in binary addition is the carry propagation delay, which slows down addition, particularly in large bit-width operands. This project focuses on the design and implementation of a 16-bit Carry Lookahead Adder (CLA) to address this issue. The Carry Lookahead Adder significantly improves upon traditional adder designs, such as the Ripple Carry Adder (RCA), by reducing carry propagation delay through the parallel generation and propagation of carry bits. The CLA uses two key concepts, carry generate and carry propagate, to predict carry values ahead of time and perform addition in parallel, thereby improving speed. This report presents the detailed design and implementation process of the 16-bit CLA, including the construction of the adder using basic logic gates and the analysis of its performance. The results demonstrate a substantial reduction in addition time compared to the Ripple Carry Adder, making the CLA a suitable choice for high-speed computational applications. The project highlights the advantages of the Carry Lookahead Adder in enhancing the speed of arithmetic operations in digital circuits. |

# Introduction

In digital systems, arithmetic operations such as addition are fundamental to the execution of various tasks in processors, memory units, and computational devices. Among these, binary addition is a key operation that is widely used in applications ranging from basic calculations to complex signal processing. As digital systems scale in complexity and size, optimizing the performance of binary adders becomes increasingly important, especially in high-speed applications where performance and efficiency are critical.

A common challenge in the design of binary adders is the carry propagation delay, which occurs when adding two binary numbers. In traditional adder designs like the Ripple Carry Adder (RCA), each bit of the sum relies on the carry bit from the previous stage, resulting in a sequential carry propagation across all bit positions. This sequential delay becomes significant as the operand size increases, leading to slower performance and limiting the overall speed of the system.

To address this issue, the Carry Lookahead Adder (CLA) was developed in the 1960s. The CLA is designed to reduce the carry propagation delay by using additional logic to predict the carry values for each bit position in parallel, rather than waiting for the carry from the previous stage. This enables faster addition as the carry bits are computed ahead of time, thus reducing the overall delay in the addition process.

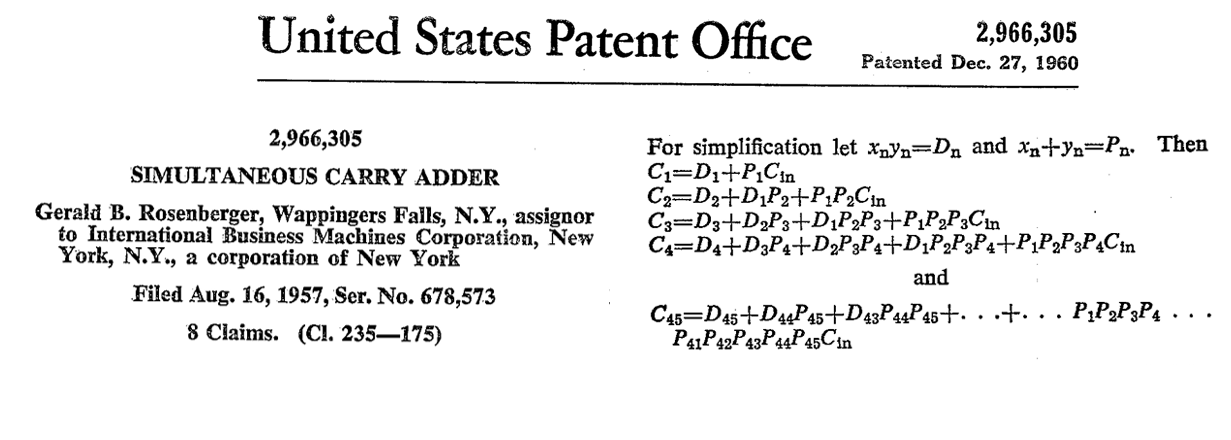
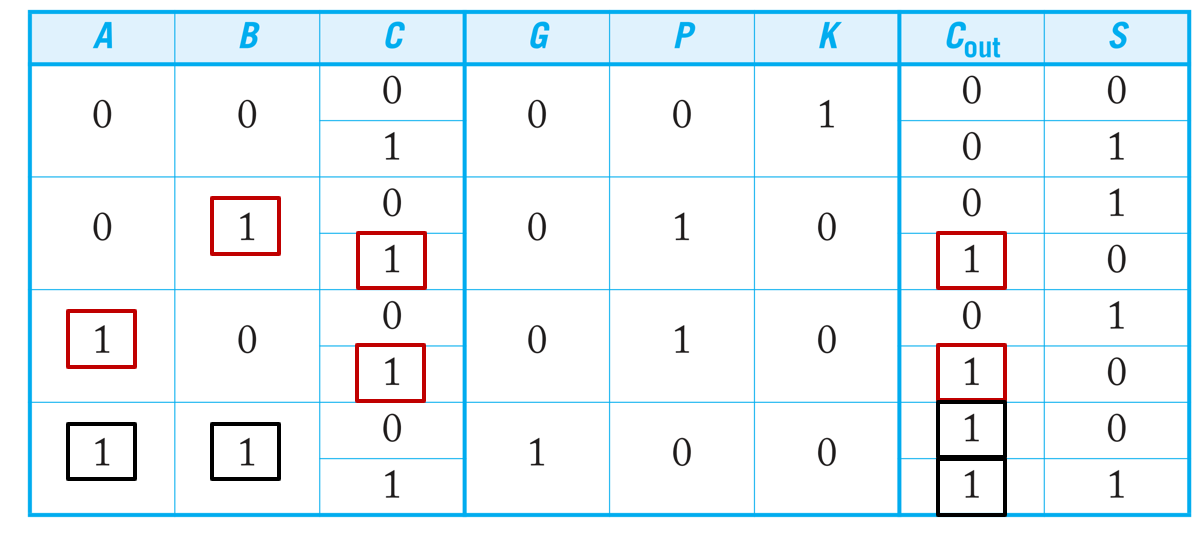
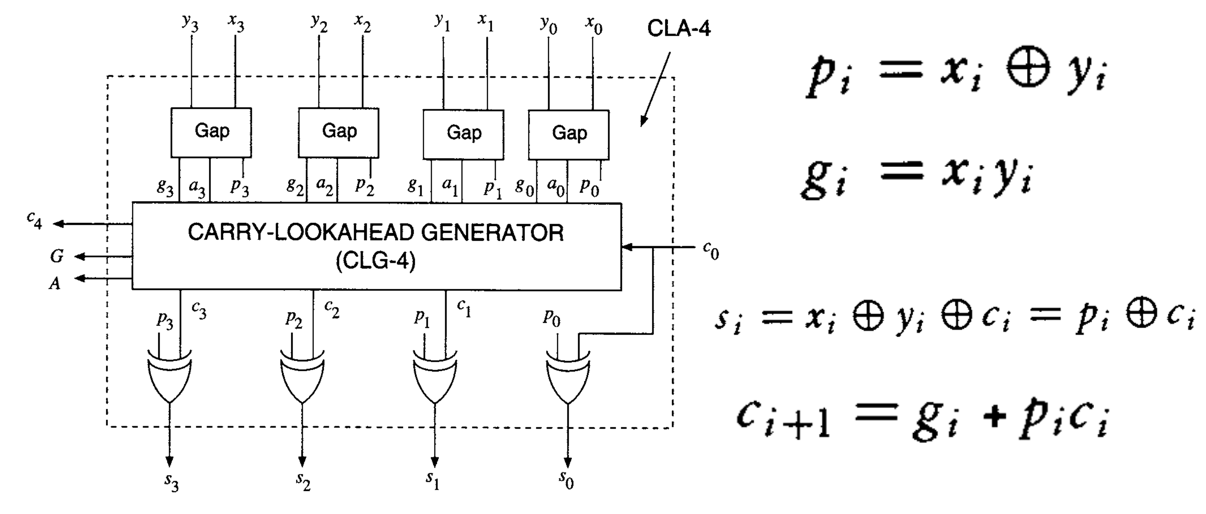


Fig: History of Simultaneous Carry Adder [1]

Simultaneous Carry Adder was first patented in 1960 . CLA was designed based on the truth table observations.

 Fig: Truth table for CLA [2]

Carry generation and propagation signals were determined from this table. The propagation signal is the XOR of the the two inputs and the generation is the AND of the two inputs. The Formulas are presented in the next page.

Fig: CLA basic structure and formulas [3]

The 16-bit Carry Lookahead Adder project focuses on implementing a CLA for 16-bit binary numbers to demonstrate how this design can improve the speed of addition in comparison to simpler adder architectures like the Ripple Carry Adder. The CLA utilizes two primary functions: carrygenerate and carry propagate, which allow for the parallel generation and propagation of carries, significantly reducing the critical path delay. This design is particularly useful in applications where high-speed computation is essential, such as in processors and digital signal processors.

This project explores the design and implementation of the 16-bit Carry Lookahead Adder, emphasizing the logic behind the carry generation and propagation functions. The report details the step-by-step construction of the adder, the use of logic gates, and the challenges encountered in ensuring that the adder performs efficiently with minimal delay. The results demonstrate the advantages of the Carry Lookahead Adder in terms of speed compared to the Ripple Carry Adder, and the potential applications in high-performance computing systems.

# Design

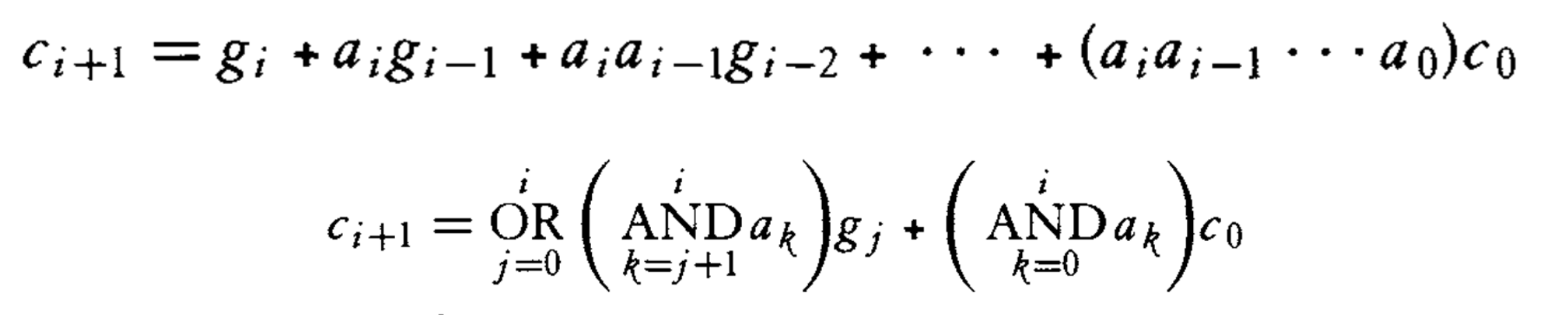
## Problem Formulation (PO(b))

A 16-bit carry lookahead adder takes two 16-bit binary numbers a and b as inputs, along with a carry-in (cin) bit and a clock signal. The adder performs fast binary addition by calculating all carries simultaneously instead of waiting for each carry to ripple through. The outputs include a 16-bit sum and a final carry-out bit, which together represent the addition result. This is the base problem we have been tasked to carry out in this project.

### Identification of Scope

The identification of scope for this 16-bit carry lookahead adder centers on managing the complex fan-in requirements shown in the theoretical equations. The first equation demonstrates how carry generation becomes increasingly complex with higher bit positions, requiring more AND-OR terms. The second equation presents this in a condensed mathematical form using AND-OR operations across multiple terms.

The equations for illustrating the excessive fan in requirements are as follows:

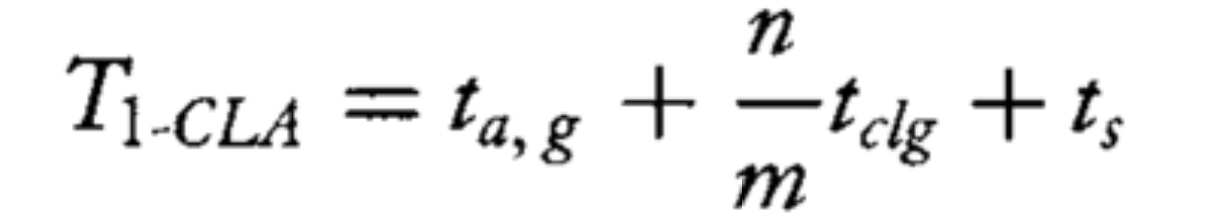


we can see the carry equation becomes very complex as the number of bits increases. For bit position 15, we would need:

* One 16-input OR gate
* 15 AND gates with input sizes ranging from 2 to 16.

This makes the implementation impractical due to large fan-in requirements.

By breaking down the 16-bit adder into four 4-bit groups, we can effectively balance speed and complexity. This approach aligns with the optimization equation T1-CLA, where ta,g represents AND-gate delay, tclg represents carry-lookahead generation time, and ts represents sum calculation time. The ratio n/m influences the overall delay, where n represents the total bits and m represents the group size.



For 4-bit x 4 CLA implementation:

* Total bits (n) = 16
* Group size (m) = 4
* Number of groups = n/m = 4

T1-CLA = ta,g + (n/m)tclg + ts = ta,g + (16/4)tclg + ts = ta,g + 4tclg + ts

Where:

* ta,g = Time to compute generate and propagate signals
* tclg = Time for carry lookahead generator (much smaller for 4-bit groups)
* ts = Time to compute final sum

This grouping strategy significantly reduces the fan-in burden while maintaining faster performance than a ripple carry design. Each 4-bit group manages its internal carries using lookahead logic, while group-level carries are calculated separately. This hierarchical structure creates a practical balance between the theoretical speed advantages of carry lookahead and the hardware constraints of RTL implementation.

### Literature Review

1. A New Carry Look-Ahead Adder Architecture Optimized for Speed and Energy by Padmanabhan Balasubramanian and Douglas L. Maskell

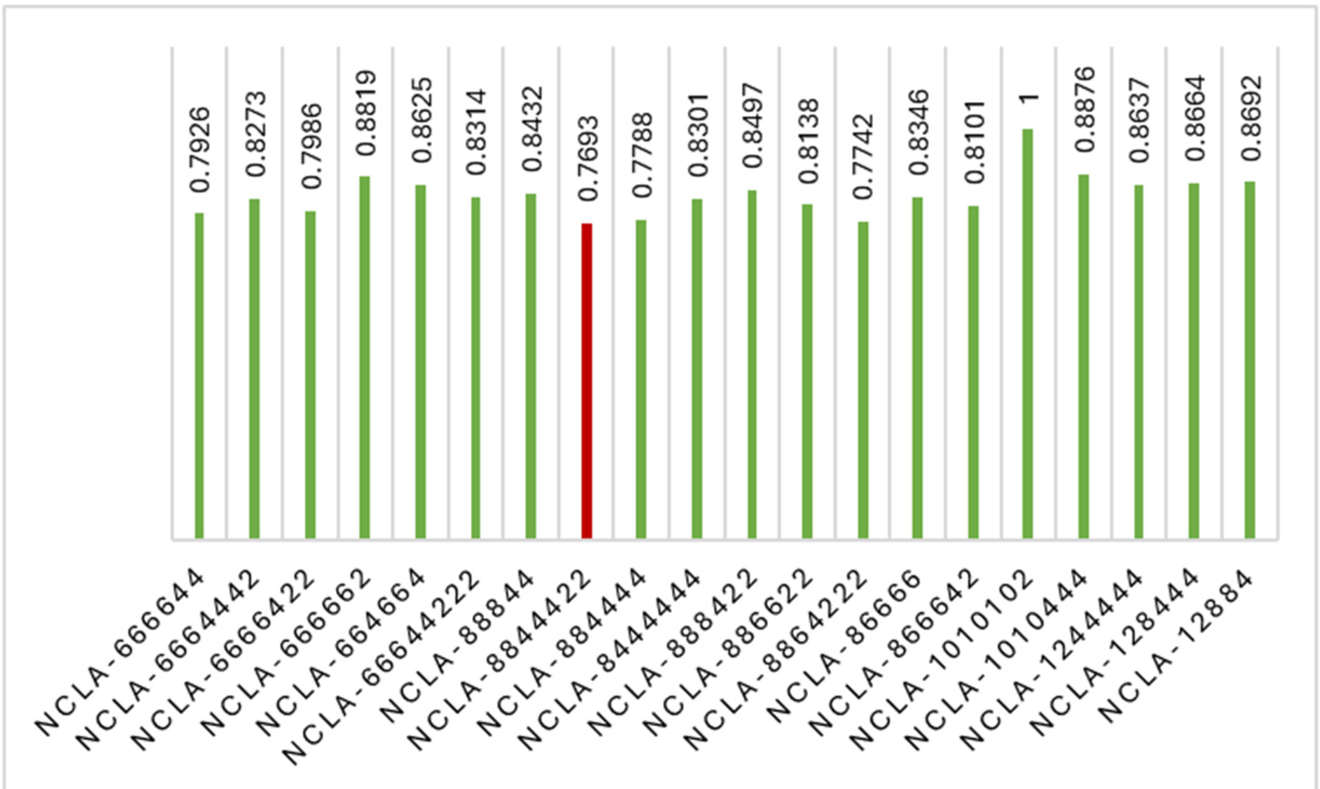


Figure : NCLA Architecture Performance Graph for Different Group Sizes. [4]

The research paper showcases a non-uniform-size carry lookahead adder (NCLA) architecture as an improvement over conventional uniform-size CLA designs. Their implementation using 32-bit addition in 28nm CMOS technology showed impressive results with significant reductions in delay, energy consumption, and area compared to other high-speed adders like Kogge-Stone, conditional sum, and carry select adders.

However, my practical implementation of a 16-bit NCLA2248 architecture did not achieve the power and area improvements seen in the paper. This difference could be attributed to several factors: the smaller bit width of my design (16-bit vs. 32-bit), different technology nodes, variations in synthesis constraints, or specific implementation details. The contrasting results highlight how theoretical improvements in architecture may not always translate directly to all practical implementations due to varying design parameters and constraints.

### Formulation of Problem

The problem formulation for this digital design project begins with a foundational 16-bit carry lookahead adder that accepts two 16-bit inputs (a and b), along with carry-in (Cin) and clock signals, producing a 16-bit sum and carry-out. The core challenge lies in optimizing the inherent speed-complexity trade-off in carry lookahead architectures.

Initial analysis led to breaking down the design into four 4-bit groups to manage fan-in requirements while maintaining speed advantages over ripple carry approaches. This decision was supported by theoretical equations showing carry propagation and the optimization relationship between delay components.

A thorough literature review of a recent paper implementing non-uniform sized CLA modules (NCLA) in 32-bit design showed promising improvements in delay, energy, and area compared to conventional approaches. However, my implementation of NCLA2248 architecture in 16-bit design did not achieve similar power and area improvements, highlighting the practical challenges in translating theoretical architectures across different design specifications.

This project thus focuses on balancing theoretical carry lookahead concepts with practical RTL implementation constraints to achieve optimal performance in a 16-bit adder design.

### Analysis

The analysis of this 16-bit carry lookahead adder project encompasses multiple verification stages and performance evaluations. The RTL design implements a hierarchical carry lookahead structure divided into four 4-bit groups, optimizing the balance between propagation delay and hardware complexity.

The verification strategy follows a comprehensive three-tier approach. The initial directed testbench provides basic functional verification through specific test cases, allowing visual inspection of waveforms to confirm proper timing and functionality. The self-checking testbench builds upon this by automating the verification process, comparing actual outputs against expected results for each test case.

The layered testbench, implemented in SystemVerilog, provides the most thorough verification coverage. It generates randomized test scenarios for the 16-bit inputs A and B, and the carry-in bit. With three inputs (a[15:0], b[15:0], cin), the possible combinations are extensive. The testbench examines critical corner cases, including zero-zero inputs and overflow conditions. Coverage metrics track both individual signal coverage (a, b, and cin) and cross-coverage between all three inputs, ensuring comprehensive testing of the design's functionality.

Performance analysis includes synthesis results examining area utilization and power consumption. The design undergoes detailed timing analysis to verify meeting clock constraints. Floorplanning analysis evaluates the physical implementation aspects, including cell placement and routing considerations. This multi-faceted analysis approach ensures both functional correctness and efficient implementation of the carry lookahead adder design, providing a complete picture of its performance and reliability.

The systematic verification methodology progressively builds confidence in the design's correctness while the synthesis and implementation analysis confirms its practicality in hardware.

## Design Method (PO(a))

### RTL Design:

The design method employs a modular approach to implement a 16-bit carry lookahead adder using SystemVerilog. The architecture is structured into four main modules: a top-level module (cla\_top\_sv) and three supporting modules (gen\_prop\_unit, base4\_carry\_unit, and summation\_unit). This hierarchical organization improves code readability and maintainability.

The generate-propagate unit creates fundamental carry lookahead signals using bitwise operations. The base4\_carry\_unit handles carry calculations for 4-bit groups, implementing the core carry lookahead logic. The design splits the 16-bit addition into four 4-bit sections, each managed by a separate base4\_carry\_unit instance, with inter-section carries properly connected.

The summation\_unit implements sequential logic using always\_ff block, synchronizing the final sum calculation with the clock signal. The design uses SystemVerilog-specific features like logic type declarations and generate blocks for cleaner code structure. The modular approach allows independent testing and verification of each functional block while maintaining the overall system integration through well-defined interfaces between modules.

### Directed and Self-Checking Testbench:

The testbench implementation combines directed testing with self-checking capabilities to verify the 16-bit carry lookahead adder design. The testbench generates a 10ns clock period and systematically tests various input combinations through a directed approach with automated result verification.

The test scenarios include specific input combinations carefully chosen to verify different aspects of the adder's functionality. These test cases range from small numbers (10, 12) to large values (65000, 65535), including corner cases like zero inputs and values near the maximum 16-bit range. The testbench also tests carry propagation by including scenarios with carry-in bits.

A check\_output task automates the verification process by comparing the adder's output with expected results calculated using built-in addition. This self-checking mechanism provides immediate pass/fail feedback for each test case. The testbench includes comprehensive result display functionality, showing binary and decimal representations of inputs and outputs, along with timing information. The implementation also includes VCD dumping capability for waveform analysis.

The combined directed and self-checking approach provides thorough verification while maintaining simplicity in the test structure. Each test case is synchronized with the clock's negative edge to ensure stable input transitions, and sufficient waiting time is included to allow for output stabilization.

### Layered Testbench:

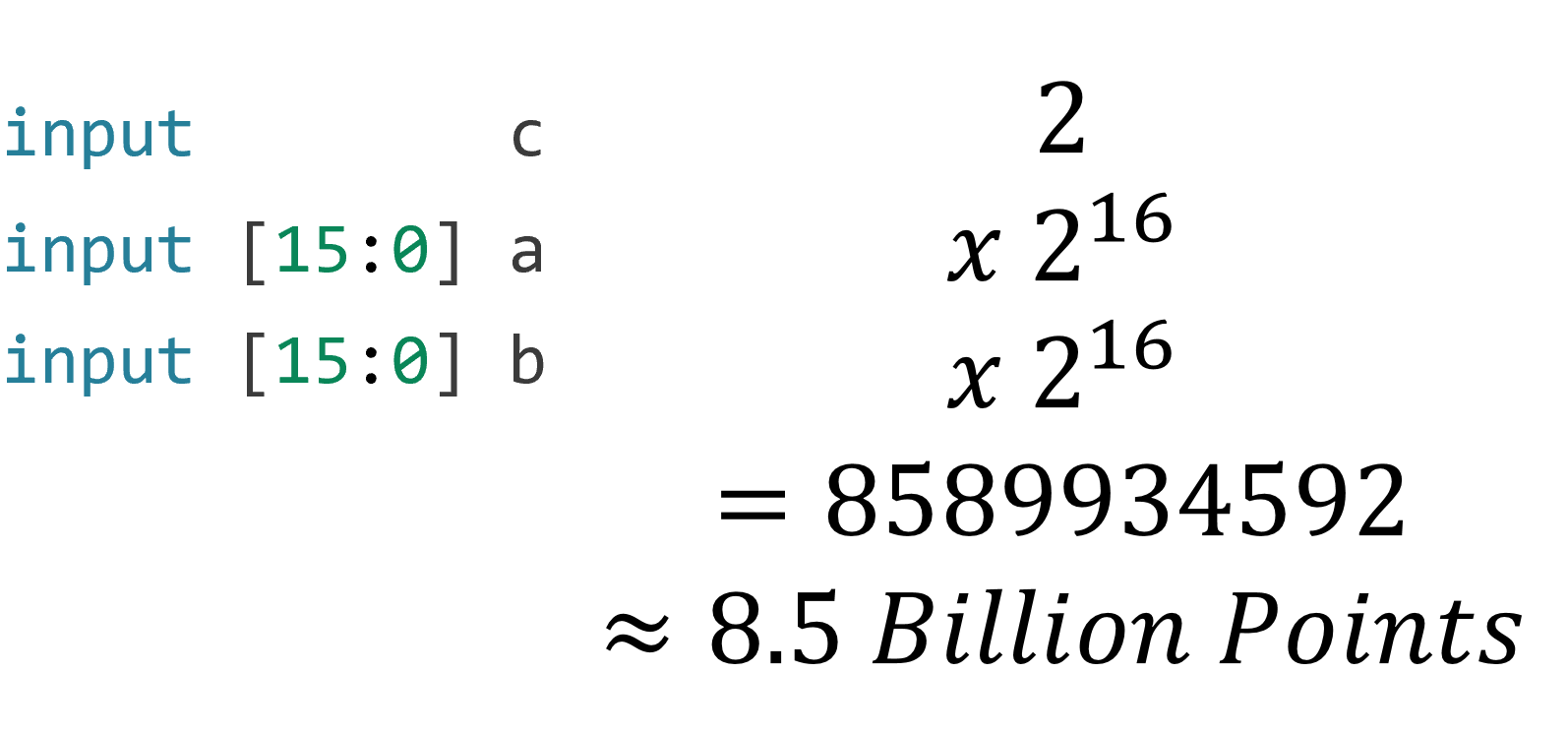
The layered testbench implements a comprehensive verification environment using SystemVerilog's object-oriented features. The testbench architecture consists of several key components working together to verify the 16-bit carry lookahead adder design systematically.

A custom interface (cla\_if) establishes the connection points between the testbench and the design under test (DUT), with separate clocking blocks for driver and monitor. The verification environment is built using five main classes: transaction, generator, driver, monitor, and scoreboard, all coordinated by an environment class.

The transaction class encapsulates the test data, including randomized inputs (a, b, cin) and expected outputs. The generator creates test scenarios, starting with a specific zero-input case followed by randomized transactions. The driver applies these test vectors to the DUT through the interface, while the monitor captures the DUT's responses.

The scoreboard implements sophisticated coverage tracking by dividing the 16-bit input space into 1024 bins, providing fine-grained analysis of test coverage. It tracks individual coverage for inputs a and b, carry-in coverage, cross coverage between carry-in and inputs, and complete 3D coverage across all inputs. Special cases like zero-zero inputs and overflow conditions are specifically monitored.

We can see as it follows to analyze all possible combination of input, there are around 8.5 billion points possible.



However, the dividing the range looks like the following:

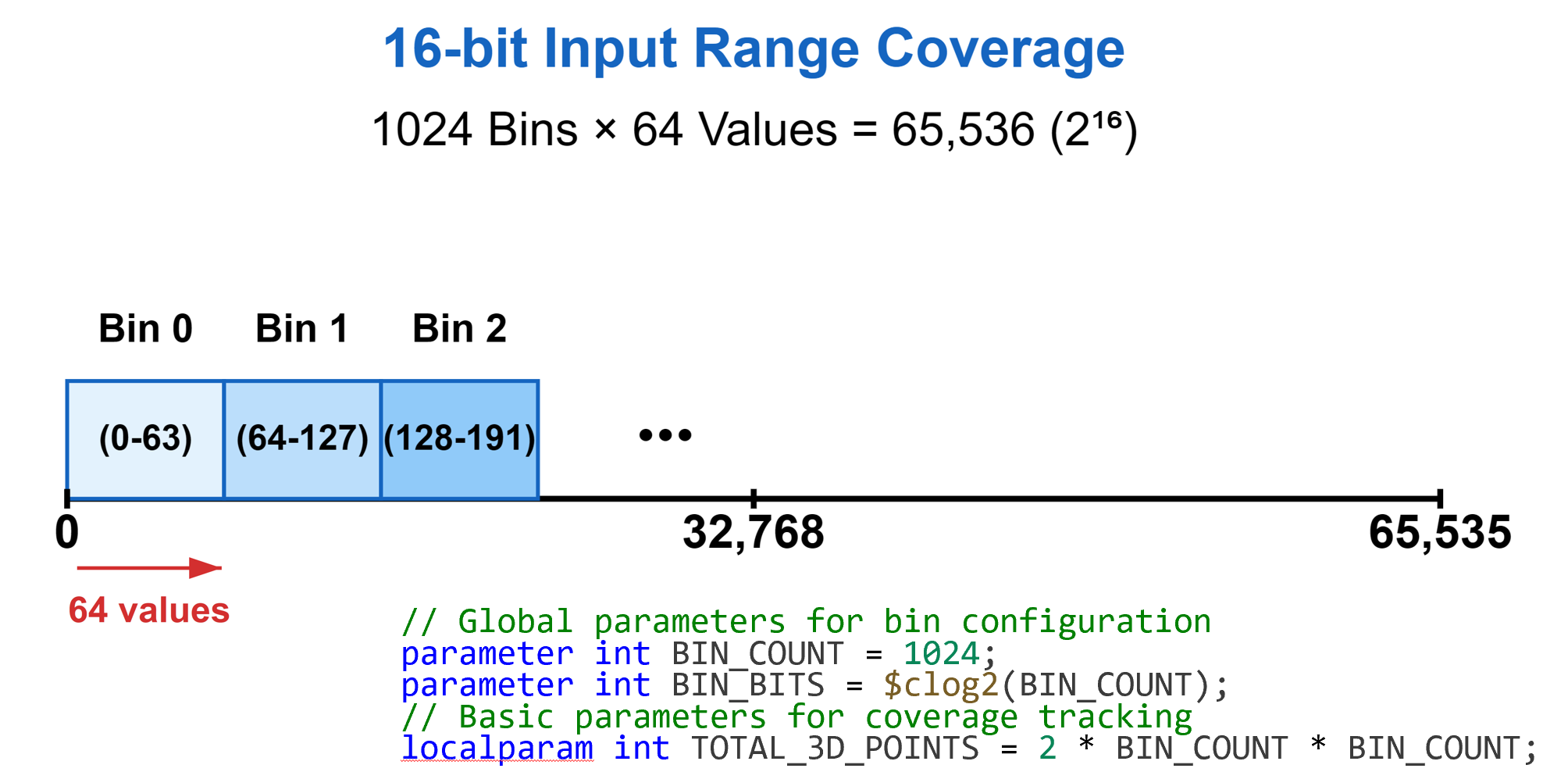


Figure : Dividing the Whole Range in 1024 Bins for the Input Values

The test program implements an incremental simulation approach, starting with 5 transactions and gradually increasing to 240,000 transactions with varying step sizes. This progressive increase helps identify any issues early while ensuring thorough verification at higher transaction counts. Coverage metrics are continuously calculated and stored in a CSV file for analysis.

The environment class orchestrates the entire verification process by creating and connecting all components through mailboxes and events. The top-level module ties everything together, generating the clock signal, instantiating the interface, test program, and DUT, and enabling waveform dumping for debugging purposes. This structured approach ensures comprehensive verification of the carry lookahead adder's functionality across its entire operating range.

## Circuit Diagram

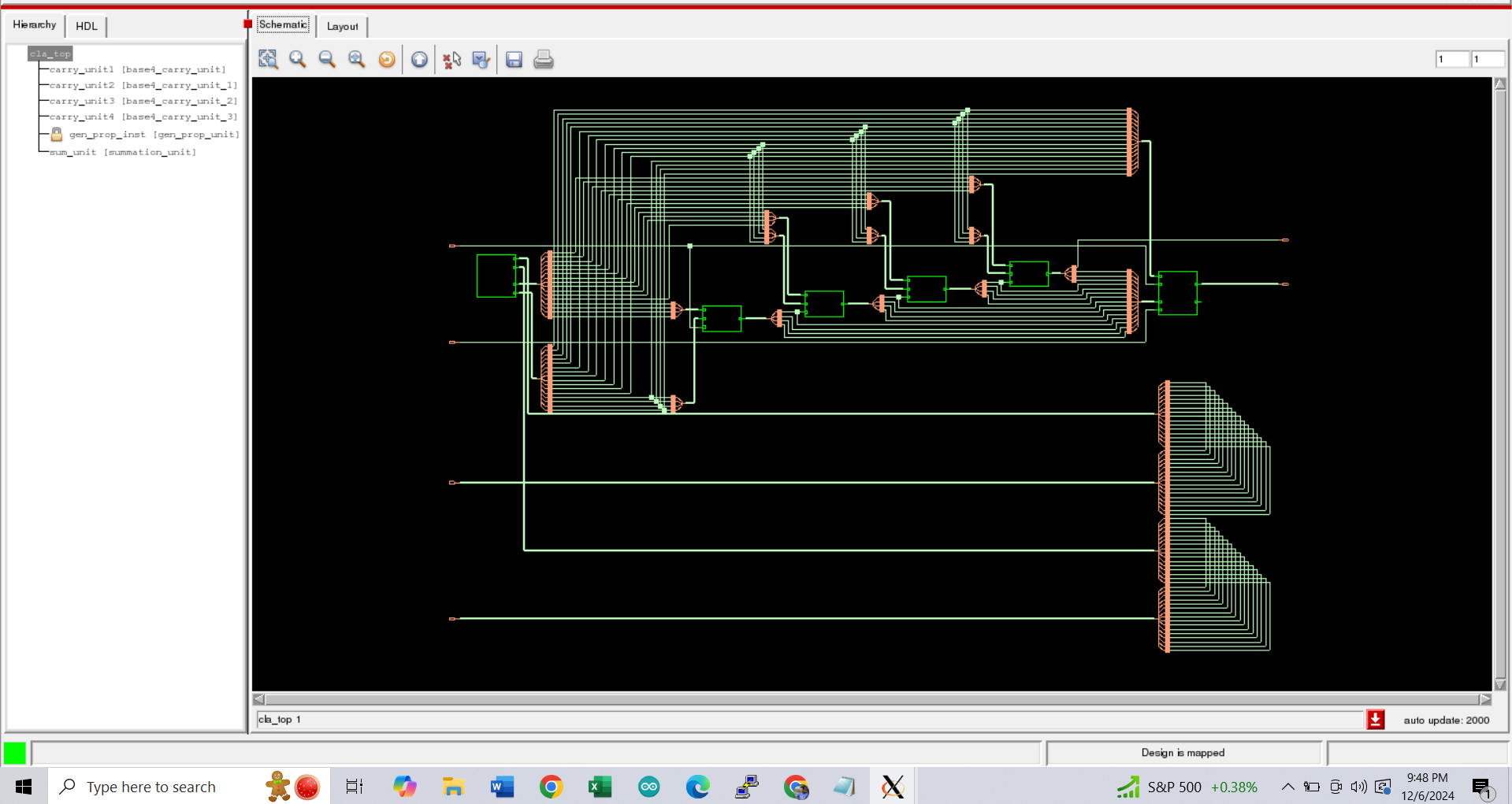


Fig: Schematic of Carry Lookahead Adder

## Full Source Code of Firmware

### RTL Code:

// SystemVerilog module for generate and propagate unit

module gen\_prop\_unit (

    input  logic [15:0] a, b,

    output logic [15:0] g, p  // Changed to logic type

);

    assign g = a & b;  // bitwise generator

    assign p = a ^ b;  // bitwise propagator

endmodule

// SystemVerilog module for base4 carry unit

module base4\_carry\_unit (

    input  logic [3:0] g,     // Generate signals

    input  logic [3:0] p,     // Propagate signals

    input  logic       cin,   // Carry-in

    output logic [4:1] cout   // Carry-out vector

);

    // Simplified carry chain implementation

    assign cout[1] = g[0] | (p[0] & cin);

    assign cout[2] = g[1] | (p[1] & g[0] | (p[0] & cin));

    assign cout[3] = g[2] | (p[2] & g[1] | (p[1] & g[0] | (p[0] & cin)));

    assign cout[4] = g[3] | (p[3] & g[2] | (p[2] & g[1] | (p[1] & g[0] | (p[0] & cin))));

endmodule

// SystemVerilog module for summation unit

module summation\_unit (

    input  logic        clk,         // Clock signal

    input  logic [15:0] p,          // Input vector p

    input  logic        cin,        // Carry-in signal

    input  logic [16:1] cout,       // Carry-out vector

    output logic [15:0] sum,        // Output sum vector

    output logic        carry\_out16  // Changed to logic type

);

    always\_ff @(posedge clk) begin  // Using always\_ff for sequential logic

        sum[0] <= p[0] ^ cin;  // First bit sum using carry-in

        for (int i = 1; i < 16; i++) begin  // Using int type for loop counter

            sum[i] <= (p[i] ^ cout[i]);  // Remaining bits sum using cout[i]

        end

    end

    assign carry\_out16 = cout[16];

endmodule

// SystemVerilog top module for 16-bit CLA

module cla\_top\_sv (

    input  logic        clk,        // Clock signal

    input  logic [15:0] a, b,       // 16-bit input operands

    input  logic        cin,        // Carry-in input

    output logic [15:0] sum,        // 16-bit output sum

    output logic        carry\_out16  // Carry-out signal

);

    // Internal signals declared as logic

    logic [15:0] g, p;                           // Generate and Propagate signals

    logic [4:1]  cout1, cout2, cout3, cout4;     // Carry-out for 4-bit sections

    logic        cout\_mid1, cout\_mid2, cout\_mid3; // Inter-section carries

    // Instantiate the generate and propagate unit

    gen\_prop\_unit gen\_prop\_inst (

        .a   (a),

        .b   (b),

        .g   (g),

        .p   (p)

    );

    // Instantiate carry units for each 4-bit section

    base4\_carry\_unit carry\_unit1 (

        .g    (g[3:0]),

        .p    (p[3:0]),

        .cin  (cin),

        .cout (cout1)

    );

    assign cout\_mid1 = cout1[4];

    base4\_carry\_unit carry\_unit2 (

        .g    (g[7:4]),

        .p    (p[7:4]),

        .cin  (cout\_mid1),

        .cout (cout2)

    );

    assign cout\_mid2 = cout2[4];

    base4\_carry\_unit carry\_unit3 (

        .g    (g[11:8]),

        .p    (p[11:8]),

        .cin  (cout\_mid2),

        .cout (cout3)

    );

    assign cout\_mid3 = cout3[4];

    base4\_carry\_unit carry\_unit4 (

        .g    (g[15:12]),

        .p    (p[15:12]),

        .cin  (cout\_mid3),

        .cout (cout4)

    );

    // Instantiate the summation unit

    summation\_unit sum\_unit (

        .clk         (clk),

        .p           (p),

        .cin         (cin),

        .cout        ({cout4[4:1], cout3[4:1], cout2[4:1], cout1[4:1]}),

        .sum         (sum),

        .carry\_out16 (carry\_out16)

    );

endmodule

### Directed and Self-Checking Testbench Code:

module cla\_top\_tb;

    // Test inputs

    logic [15:0] a, b;   // 8-bit input operands

    logic       cin;          // Carry-in input

    logic       clk;          // Clock signal

    logic [15:0] sum;   // Output sum

    logic       carry\_out16;

    // Instantiate the cla\_top module

    cla\_top dut (

        .a(a),

        .b(b),

        .cin(cin),

        .clk(clk),

        .sum(sum),

        .carry\_out16(carry\_out16)

    );

    // Clock generation

    initial begin

        clk = 0;

        forever #5 clk = ~clk; // Toggle clock every 5 time units

    end

    // Test sequence

    initial begin

        // Display header

        $display("--------------- CLA Testbench Results ---------------");

        $display("Time\t a (Binary)\t             b (Binary)\t       cin\t sum (Binary)\t sum (Decimal)\t carry\_out\t");

        // Test Case 1: a = 10, b = 12, cin = 0

        @(negedge clk);  //$display("Time is %t", $time);

        a = 16'b00001010; // 10

        b = 16'b00001100; // 12

        cin = 0;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin); // Expected sum = 10 + 12 = 22

        @(negedge clk);  //$display("Time is %t", $time);

        a = 500;

        b = 600;

        cin = 0;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        @(negedge clk);  //$display("Time is %t", $time);

        a = 30500;

        b = 20499;

        cin = 1;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        @(negedge clk);  //$display("Time is %t", $time);

        a = 65000;

        b = 10;

        cin = 1;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        @(negedge clk);  //$display("Time is %t", $time);

        a = 0;

        b = 0;

        cin = 1;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        @(negedge clk);  //$display("Time is %t", $time);

        a = 65120;

        b = 3000;

        cin = 1;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        @(negedge clk);  //$display("Time is %t", $time);

        a = 65535;

        b = 0;

        cin = 1;

        #10; // Wait for one clock cycle

        check\_output(a+b+cin);

        #10;

        $finish;

    end

    // Task to check output

    task check\_output(input [16:0] expected\_sum);

        // Display current results

        $display("%0t\t %b\t %b\t %b\t %b\t %0d\t          %b\t       ", $time, a, b, cin, sum, sum, carry\_out16);

        // Compare actual and expected output

        if ((sum !== expected\_sum[15:0]) | (carry\_out16 !== expected\_sum[16])) begin

            $display("\033[1;31mFAIL: Expected sum = %b, got sum = %b\033[0m", expected\_sum[15:0], sum);

            $display("\033[1;31m---------------------------------------------------------\033[0m");

        end else begin

            $display("\033[1;32mPASS: sum = %b\033[0m", sum);

            $display("\033[1;32m---------------------------------------------------------\033[0m");

        end

    endtask

      // VCD dump

    initial begin

    $dumpfile("cla\_directed\_tb.vcd");

    $dumpvars;

    end

endmodule

### Layered Testbench Code

/\*

Description

Author : Anindya Kishore Choudhury (anindyakchoudhury@gmail.com)

\*/

// Parameters for simulation control

parameter int SIM\_COUNT\_START = 5;

parameter int SIM\_COUNT1 = 240000;

parameter int SIM\_INCREMENT\_5\_20 = 5;

parameter int SIM\_INCREMENT\_20\_50 = 10;

parameter int SIM\_INCREMENT\_50\_100 = 25;

parameter int SIM\_INCREMENT\_100\_300 = 50;

parameter int SIM\_INCREMENT\_300\_600 = 100;

parameter int SIM\_INCREMENT\_600\_1000 = 200;

parameter int SIM\_INCREMENT\_1000\_1400 = 300;

parameter int SIM\_INCREMENT\_1400\_2000 = 800;

int SIM\_COUNT;

int increment;

// Global parameters for binning configuration

parameter int BIN\_COUNT = 1024;  // Can be changed to 256, 128 etc.

parameter int BIN\_BITS = $clog2(BIN\_COUNT);

// Interface definition

interface cla\_if(input clk);

    logic [15:0] a, b;

    logic cin;

    logic [15:0] sum;

    logic carry\_out16;

    clocking driver\_cb @(negedge clk);

        default input #1 output #1;

        output a, b, cin;

    endclocking

    clocking mon\_cb @(negedge clk);

        default input #1 output #1;

        input a, b, cin;

        input sum, carry\_out16;

    endclocking

    modport DRIVER  (clocking driver\_cb, input clk);

    modport MONITOR (clocking mon\_cb, input clk);

endinterface

// Transaction class

class transaction;

    rand bit [15:0] a;

    rand bit [15:0] b;

    rand bit cin;

    bit [15:0] sum;

    bit carry\_out16;

    function bit compare(transaction other);

        return (this.a == other.a) && (this.b == other.b) &&

               (this.cin == other.cin) && (this.sum == other.sum) &&

               (this.carry\_out16 == other.carry\_out16);

    endfunction

endclass

// Generator class

class generator;

    mailbox gen2driv;

    transaction g\_trans;

    function new(mailbox gen2driv);

        this.gen2driv = gen2driv;

    endfunction

    task main(input int count);

    // First transaction: Force a=0, b=0, cin=0

    g\_trans = new();

    g\_trans.a = 16'h0000;

    g\_trans.b = 16'h0000;

    g\_trans.cin = 1'b0;

    gen2driv.put(g\_trans);

    // Remaining transactions: Random

    repeat(count-1) begin  // count-1 because we already did one transaction

            g\_trans = new();

            assert(g\_trans.randomize());

            gen2driv.put(g\_trans);

        end

    endtask

endclass

// Driver class

class driver;

    mailbox gen2driv, driv2sb;

    virtual cla\_if.DRIVER claif;

    transaction d\_trans;

    event driven;

    function new(mailbox gen2driv, driv2sb, virtual cla\_if.DRIVER claif, event driven);

        this.gen2driv = gen2driv;

        this.claif = claif;

        this.driven = driven;

        this.driv2sb = driv2sb;

    endfunction

    task main(input int count);

        repeat(count) begin

            d\_trans = new();

            gen2driv.get(d\_trans);

            @(claif.driver\_cb);

            claif.driver\_cb.a <= d\_trans.a;

            claif.driver\_cb.b <= d\_trans.b;

            claif.driver\_cb.cin <= d\_trans.cin;

            driv2sb.put(d\_trans);

            -> driven;

        end

    endtask

endclass

// Monitor class

class monitor;

    mailbox mon2sb;

    virtual cla\_if.MONITOR claif;

    transaction m\_trans;

    event driven;

    function new(mailbox mon2sb, virtual cla\_if.MONITOR claif, event driven);

        this.mon2sb = mon2sb;

        this.claif = claif;

        this.driven = driven;

    endfunction

    task main(input int count);

        @(driven);

        @(claif.mon\_cb);

        repeat(count) begin

            m\_trans = new();

            @(posedge claif.clk);

            m\_trans.sum = claif.mon\_cb.sum;

            m\_trans.carry\_out16 = claif.mon\_cb.carry\_out16;

            m\_trans.a = claif.mon\_cb.a;

            m\_trans.b = claif.mon\_cb.b;

            m\_trans.cin = claif.mon\_cb.cin;

            mon2sb.put(m\_trans);

        end

    endtask

endclass

// Scoreboard class with overflow and zero coverage

class scoreboard;

    // Basic parameters for coverage tracking

    localparam int TOTAL\_3D\_POINTS = 2 \* BIN\_COUNT \* BIN\_COUNT;

    // Transaction handling

    mailbox driv2sb, mon2sb;

    transaction d\_trans, m\_trans;

    event driven;

    int pass, fail;

    // Individual distribution tracking arrays

    int a\_distribution[BIN\_COUNT];

    int b\_distribution[BIN\_COUNT];

    int cin\_distribution[2];

    // Cross coverage tracking arrays

    int cin\_cross\_a\_distrib[2][BIN\_COUNT];

    int cin\_cross\_b\_distrib[2][BIN\_COUNT];

    // 3D coverage tracking using a bit array

    bit coverage\_3d\_space[2][BIN\_COUNT][BIN\_COUNT];

    int total\_unique\_combinations;

    // Special case coverage tracking

    bit zero\_zero\_covered;    // Track if a=0 and b=0 case occurred

    bit overflow\_covered;     // Track if a+b > 2^16-1

    real zero\_zero\_coverage;

    real overflow\_coverage;

    // Coverage percentage storage

    real coverage\_a, coverage\_b, coverage\_cin;

    real cross\_coverage\_a[2];

    real cross\_coverage\_b[2];

    real coverage\_3d;

    // Constructor

    function new(mailbox driv2sb, mon2sb);

        this.driv2sb = driv2sb;

        this.mon2sb = mon2sb;

        this.total\_unique\_combinations = 0;

        this.zero\_zero\_covered = 0;

        this.overflow\_covered = 0;

    endfunction

    // Helper function to map 16-bit value to bin

    function bit [BIN\_BITS-1:0] get\_bin(logic [15:0] value);

        return value[15:16-BIN\_BITS];  // Take upper bits based on bin size

    endfunction

    // Update 3D coverage space

    function void update\_3d\_coverage(bit cin, bit [BIN\_BITS-1:0] a\_bin, bit [BIN\_BITS-1:0] b\_bin);

        if (!coverage\_3d\_space[cin][a\_bin][b\_bin]) begin

            coverage\_3d\_space[cin][a\_bin][b\_bin] = 1;

            total\_unique\_combinations++;

        end

    endfunction

    // Calculate final 3D coverage percentage

    function real calculate\_3d\_coverage();

        return (total\_unique\_combinations \* 100.0) / TOTAL\_3D\_POINTS;

    endfunction

    // Main task to process transactions

    task main(input int count);

        $display("------------------Scoreboard Test Starts--------------------");

        $display("Using %0d bins for coverage analysis", BIN\_COUNT);

        repeat(count) begin

            m\_trans = new();

            mon2sb.get(m\_trans);

            check\_result();

            if(!m\_trans.compare(d\_trans)) begin

                fail++;

                $display("\033[31mTest No.%d Failed\033[0m : a=%h b=%h cin=%b Expected: sum=%h cout=%b, Got: sum=%h cout=%b",

                        pass + fail, d\_trans.a, d\_trans.b, d\_trans.cin,

                        d\_trans.sum, d\_trans.carry\_out16,

                        m\_trans.sum, m\_trans.carry\_out16);

            end

            else begin

                pass++;

            end

        end

        calculate\_coverage();

        display\_coverage();

        write\_coverage\_report();

        $display("\033[1;33mData flow: Passed %0d out of Total %0d Tests\033[0m", pass, pass + fail);

        $display("------------------Scoreboard Test Ends--------------------");

        pass = 0; fail = 0;

    endtask

    // Process each transaction and update coverage

    task check\_result();

        bit [BIN\_BITS-1:0] a\_bin, b\_bin;

        bit [16:0] total\_sum;  // 17-bit to check overflow

        d\_trans = new();

        driv2sb.get(d\_trans);

        // Calculate total sum for overflow check

        total\_sum = d\_trans.a + d\_trans.b + d\_trans.cin;

        // Calculate expected results

        {d\_trans.carry\_out16, d\_trans.sum} = total\_sum;

        // Check for zero+zero case

        if (d\_trans.a == '0 && d\_trans.b == '0) begin

            zero\_zero\_covered = 1;

        end

        // Check for overflow (sum exceeds 16 bits)

        if (total\_sum > 16'hFFFF) begin

            overflow\_covered = 1;

        end

        // Get bins for coverage tracking

        a\_bin = get\_bin(d\_trans.a);

        b\_bin = get\_bin(d\_trans.b);

        // Update coverage metrics

        a\_distribution[a\_bin]++;

        b\_distribution[b\_bin]++;

        cin\_distribution[d\_trans.cin]++;

        cin\_cross\_a\_distrib[d\_trans.cin][a\_bin]++;

        cin\_cross\_b\_distrib[d\_trans.cin][b\_bin]++;

        update\_3d\_coverage(d\_trans.cin, a\_bin, b\_bin);

    endtask

    // Calculate all coverage metrics

    task calculate\_coverage();

        int zeros;

        // Calculate input 'a' coverage

        zeros = 0;

        foreach(a\_distribution[i])

            if(a\_distribution[i] == 0) zeros++;

        coverage\_a = 100.0 \* (BIN\_COUNT - zeros) / BIN\_COUNT;

        // Calculate input 'b' coverage

        zeros = 0;

        foreach(b\_distribution[i])

            if(b\_distribution[i] == 0) zeros++;

        coverage\_b = 100.0 \* (BIN\_COUNT - zeros) / BIN\_COUNT;

        // Calculate input 'cin' coverage

        zeros = 0;

        foreach(cin\_distribution[i])

            if(cin\_distribution[i] == 0) zeros++;

        coverage\_cin = 100.0 \* (2 - zeros) / 2;

        // Calculate cross coverage for cin×a

        for(int i = 0; i < 2; i++) begin

            zeros = 0;

            foreach(cin\_cross\_a\_distrib[i][j])

                if(cin\_cross\_a\_distrib[i][j] == 0) zeros++;

            cross\_coverage\_a[i] = 100.0 \* (BIN\_COUNT - zeros) / BIN\_COUNT;

        end

        // Calculate cross coverage for cin×b

        for(int i = 0; i < 2; i++) begin

            zeros = 0;

            foreach(cin\_cross\_b\_distrib[i][j])

                if(cin\_cross\_b\_distrib[i][j] == 0) zeros++;

            cross\_coverage\_b[i] = 100.0 \* (BIN\_COUNT - zeros) / BIN\_COUNT;

        end

        // Calculate 3D coverage

        coverage\_3d = calculate\_3d\_coverage();

        // Calculate special case coverages

        zero\_zero\_coverage = zero\_zero\_covered ? 100.0 : 0.0;

        overflow\_coverage = overflow\_covered ? 100.0 : 0.0;

    endtask

    // Display coverage results

    task display\_coverage();

        $display("\n=== Coverage Results (Using %0d bins) ===", BIN\_COUNT);

        $display("Input a coverage: %0.2f%%", coverage\_a);

        $display("Input b coverage: %0.2f%%", coverage\_b);

        $display("Input cin coverage: %0.2f%%", coverage\_cin);

        $display("\nCross Coverage (cin cross a):");

        $display("cin=0: %0.2f%%", cross\_coverage\_a[0]);

        $display("cin=1: %0.2f%%", cross\_coverage\_a[1]);

        $display("\nCross Coverage (cin cross b):");

        $display("cin=0: %0.2f%%", cross\_coverage\_b[0]);

        $display("cin=1: %0.2f%%", cross\_coverage\_b[1]);

        $display("\n3D Coverage (cin cross a cross b): %0.2f%%", coverage\_3d);

        $display("\nSpecial Cases Coverage:");

        $display("Zero + Zero case covered: %0.2f%%", zero\_zero\_coverage);

        $display("Overflow case covered: %0.2f%%", overflow\_coverage);

    endtask

    // Write coverage data to CSV file

    task write\_coverage\_report();

        int file = $fopen("cla\_coverage\_report.csv", "a");

        if(file) begin

            if($feof(file)) begin

                $fwrite(file, "SIM\_COUNT,a\_Coverage,b\_Coverage,cin\_Coverage,3D\_Coverage,Zero\_Zero\_Coverage,Overflow\_Coverage\n");

            end

            $fwrite(file, "%0d,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f,%0.2f\n",

                   SIM\_COUNT, coverage\_a, coverage\_b, coverage\_cin, coverage\_3d,

                   zero\_zero\_coverage, overflow\_coverage);

            $fclose(file);

        end

        else begin

            $display("Error: Unable to open coverage report file");

        end

    endtask

endclass

// Environment class

class environment;

    mailbox gen2driv, driv2sb, mon2sb;

    generator gen;

    driver drv;

    monitor mon;

    scoreboard scb;

    event driven;

    virtual cla\_if claif;

    function new(virtual cla\_if claif);

        this.claif = claif;

        reset();

    endfunction

    function void reset();

        gen2driv = new();

        driv2sb = new();

        mon2sb = new();

        gen = new(gen2driv);

        drv = new(gen2driv, driv2sb, claif.DRIVER, driven);

        mon = new(mon2sb, claif.MONITOR, driven);

        scb = new(driv2sb, mon2sb);

    endfunction

    task main(input int count);

        fork

            gen.main(count);

            drv.main(count);

            mon.main(count);

            scb.main(count);

        join

    endtask

endclass

// Update the test program to use incremental simulation counts:

program test(cla\_if claif);

  environment env;

  initial begin

    $display("Test: initial block started");

    env = new(claif);

    for (int i = SIM\_COUNT\_START; i <= SIM\_COUNT1;) begin

      SIM\_COUNT = i;

      $display("\n\nTest: Running simulation with SIM\_COUNT = %0d", SIM\_COUNT);

      if (env != null)

          env.main(SIM\_COUNT);

      else

          $display("Error: env is null");

      // Set the increment based on the current count

      if (i < 20)

          increment = SIM\_INCREMENT\_5\_20;

      else if (i < 50)

          increment = SIM\_INCREMENT\_20\_50;

      else if (i < 100)

          increment = SIM\_INCREMENT\_50\_100;

      else if (i < 300)

          increment = SIM\_INCREMENT\_100\_300;

      else if (i < 600)

          increment = SIM\_INCREMENT\_300\_600;

      else if (i < 1000)

          increment = SIM\_INCREMENT\_600\_1000;

      else if (i < 1400)

          increment = SIM\_INCREMENT\_1000\_1400;

      else if (i < 2000)

          increment = SIM\_INCREMENT\_1400\_2000;

      else

          increment = SIM\_INCREMENT\_1400\_2000 \* 2;

      i += increment;

    end

    $display("Test: All simulations completed");

    $finish;

  end

endprogram

// Top module

module cla\_top\_layered\_parambin\_tb;

  bit clk;

  // Clock generation

  initial begin

    forever #5 clk = ~clk;

  end

  // Interface instantiation

  cla\_if claif(clk);

  // Test instantiation

  test test\_prog(claif);

  // VCD dump

  initial begin

    $dumpfile("cla\_tb.vcd");

    $dumpvars;

  end

  // DUT instantiation

  cla\_top\_sv DUT (

    .a(claif.a),

    .b(claif.b),

    .cin(claif.cin),

    .sum(claif.sum),

    .carry\_out16(claif.carry\_out16),

    .clk(clk)

  );

endmodule

# Implementation

## Description

The implementation phase involved comprehensive testing and verification across multiple industry-standard tools and platforms. The RTL design was implemented using both Verilog and SystemVerilog, demonstrating compatibility and portability across different hardware description languages. Initial verification was performed using GTKWave for waveform analysis, providing visual confirmation of the design's timing behavior and functional correctness.

The design was then validated using Vivado, leveraging its synthesis and simulation capabilities to ensure hardware feasibility. Further verification was conducted using Cadence's NCSIM simulator, accessed through the irun command with a structured filelist for organized compilation and simulation. This multi-platform verification approach helped ensure the robustness of the carry lookahead adder implementation.

The test results were systematically collected through the testbench's coverage reporting mechanism, which generated detailed CSV files containing coverage metrics across various test scenarios. These data files were then processed using MATLAB to create visual representations of the coverage results, allowing for clear analysis of test coverage progression and identification of any potential gaps in the verification process.

The implementation successfully demonstrated the design's functionality across all platforms, with consistent results between directed tests, self-checking testbenches, and the comprehensive layered testbench. The ability to visualize and analyze the coverage data through MATLAB plots provided valuable insights into the verification process and confirmed the thoroughness of the testing strategy.

# Design Analysis and Evaluation

## Novelty

### RTL and Testbenches:

The design introduces novel approaches in both RTL implementation and verification methodology. In the RTL design, the carry lookahead adder is structured with a unique hierarchical organization using four 4-bit modules, optimizing the balance between speed and hardware complexity. The modular approach with separate generate-propagate, base4-carry, and summation units provides clear separation of concerns while maintaining efficient carry calculation and propagation.

The verification framework presents significant innovations in its layered testbench architecture and coverage-driven methodology. The testbench implements an intelligent transaction count scaling mechanism, starting with just 5 transactions and progressively increasing to 240,000 transactions. This scaling uses non-uniform increments - smaller steps (5 transactions) at lower counts where bugs are more likely to appear, and larger steps (up to 800 transactions) at higher counts. This approach optimizes verification time while maintaining thoroughness in critical regions of testing.

A particularly novel aspect is the implementation of a comprehensive coverage model using 1024 bins for analyzing the 16-bit input space. This fine-grained binning strategy provides detailed insights into test coverage across the entire input range. The coverage tracking encompasses multiple dimensions - individual input coverage, cross coverage between carry-in and inputs, and complete 3D coverage analysis. Special attention is paid to corner cases like zero-zero inputs and overflow conditions, ensuring these critical scenarios are thoroughly tested.

Key Novel Features:

- RTL Design:

* Hierarchical 4-bit module structure
* Separate functional units for generate-propagate, carry, and sum
* Efficient synchronous design with clock integration

- Verification Strategy:

* Progressive transaction scaling from 5 to 240,000
* Non-uniform increment steps based on simulation count
* 1024-bin coverage model for 16-bit input space

- Coverage Analysis:

* Individual input coverage tracking
* Cross coverage between carry-in and inputs
* 3D coverage analysis across all inputs
* Special case detection for zero-zero and overflow

- Data Collection and Analysis:

* Automated CSV file generation
* Real-time coverage metric calculation
* Integration with external analysis tools
* Quantitative verification completeness metrics

- Implementation Features:

* Cross-platform compatibility
* Multiple HDL support
* Comprehensive tool integration
* Automated result analysis

## Design Considerations (PO(c))

This section is mandatory to write if the course outcomes address PO(c) Read the PO Statement in the website first, before writing this section ([Program Outcomes and Program Educational Objectives | Department of EEE, BUET](https://eee.buet.ac.bd/academics/undergraduate/peo-po)) If not needed, remove this entire section. If needed, DO NOT Change the title of this section.

## Investigations (PO(d))

This section is mandatory to write if the course outcomes address PO(d) Read the PO Statement in the website first, before writing this section ([Program Outcomes and Program Educational Objectives | Department of EEE, BUET](https://eee.buet.ac.bd/academics/undergraduate/peo-po)) . If not needed, remove this entire section. If needed, DO NOT Change the title of this section.

### Design of Experiment

### Data Collection

**RTL and Testbenches:**

In the Data Collection phase, a multi-faceted approach was implemented to gather comprehensive simulation and coverage data. The SystemVerilog testbench incorporated automated data collection mechanisms through multiple channels to ensure thorough result analysis and verification.

The primary data collection was implemented through the scoreboard's write\_coverage\_report task, which systematically wrote coverage metrics to a CSV file named "cla\_coverage\_report.csv". This file captured critical metrics including input coverage for a and b, carry-in coverage, 3D cross coverage, and special case coverage for zero-zero inputs and overflow conditions. The CSV format was chosen for its compatibility with post-processing tools and ease of data analysis.

Real-time data visualization was achieved through strategic placement of $display() statements throughout the testbench. These displayed immediate feedback about test progress, coverage metrics, and pass/fail status of individual test cases in the terminal. The display statements were color-coded using ANSI escape sequences to highlight critical information, making it easier to identify test failures and coverage achievements during simulation.

For detailed signal analysis, the testbench generated Value Change Dump (VCD) files through $dumpfile and $dumpvars system tasks. These VCD files were utilized in both GTKWave and Simvision for waveform analysis, allowing detailed examination of signal transitions and timing relationships. The waveform viewers provided crucial insights into the design's behavior at different simulation timestamps.

The collected data provided three complementary views of the design's operation:

* Structured coverage data in CSV format for statistical analysis
* Real-time terminal output for immediate feedback during simulation
* Detailed signal waveforms for timing and functional verification

This comprehensive data collection approach enabled thorough analysis of the design's functionality and verification coverage, supporting both immediate debugging needs and long-term performance evaluation.

### Results and Analysis

**RTL Design:**

AKIF PUT THE RTL PICTURE here and describe

**Self-Checking Testbench:**

The directed testbench waveform demonstrates the successful operation of the 16-bit carry lookahead adder through various test cases. The simulation shows clear signal transitions synchronized with the clock signal, verifying proper timing behavior.

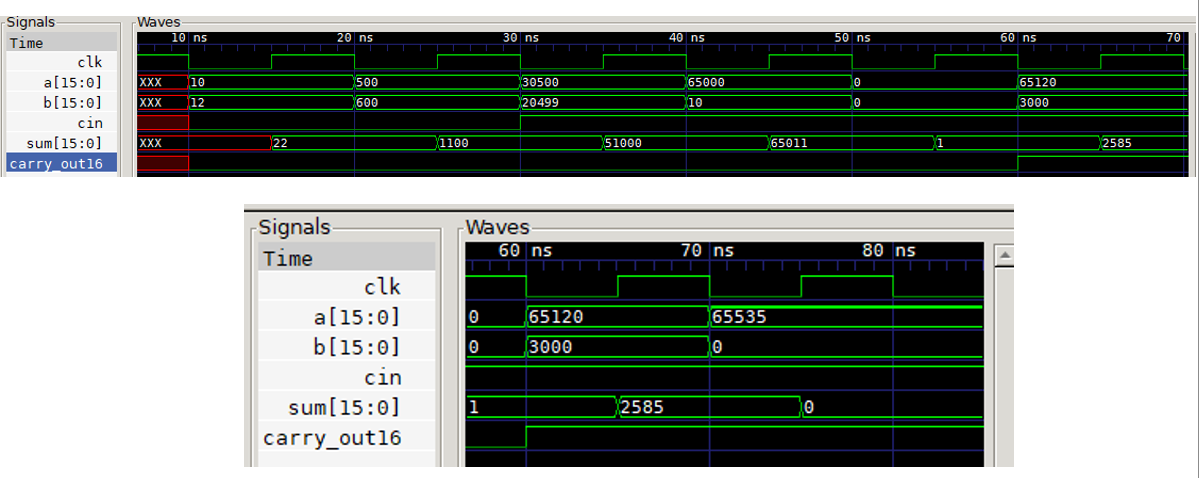


Figure : Directed Testbench Output

The test cases progress through several carefully selected input combinations:

* First test: a=10, b=12, cin = 0 resulting in sum=22
* Second test: a=500, b=600, cin = 0 resulting in sum=1100
* Third test: a=30500, b=20499, cin =1 resulting in sum=51000
* Fourth test: a=65000, b=10, cin = 1 resulting in sum=65011
* Fifth test: a=0, b=0, cin = 1 testing the zero-input case
* Sixth test: a=65120, b=3000, cin = 1 resulting in sum = 2585
* Final test: a=65535, b=0, cin = 1 testing maximum input value

Each test case shows correct addition results with proper carry propagation. The carry\_out16 signal behaves as expected, asserting when the sum exceeds 16 bits. The synchronous behavior is evident as the sum updates on the positive clock edge after inputs change.

The waveform verifies several important aspects of the design:

* Zero handling
* Large number addition
* Near-maximum value operation
* Proper carry generation
* Clock synchronization
* Input to output timing relationships

These results confirm that the carry lookahead adder correctly implements the required addition operations across a range of input values, from small numbers to near-maximum 16-bit values.

**Layered Testbench:**

The output of the layered testbench is as follows:

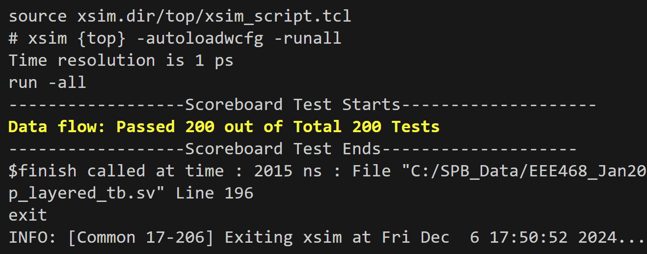


Figure : Terminal Output of the Layered Testbench without Parameterized Bins

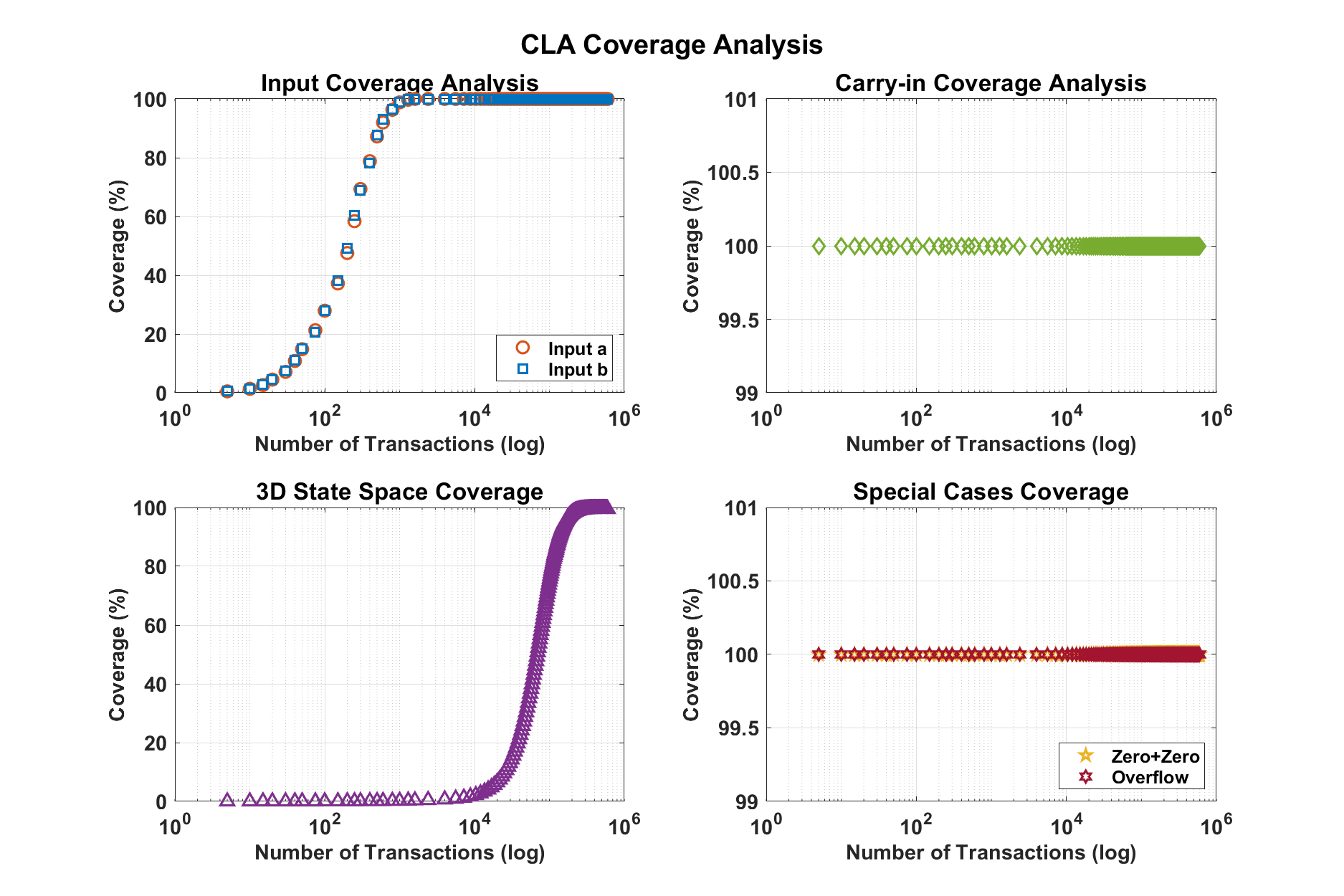


Figure : Rand Function Coverage Analysis Individually for Each Space and 3D Space

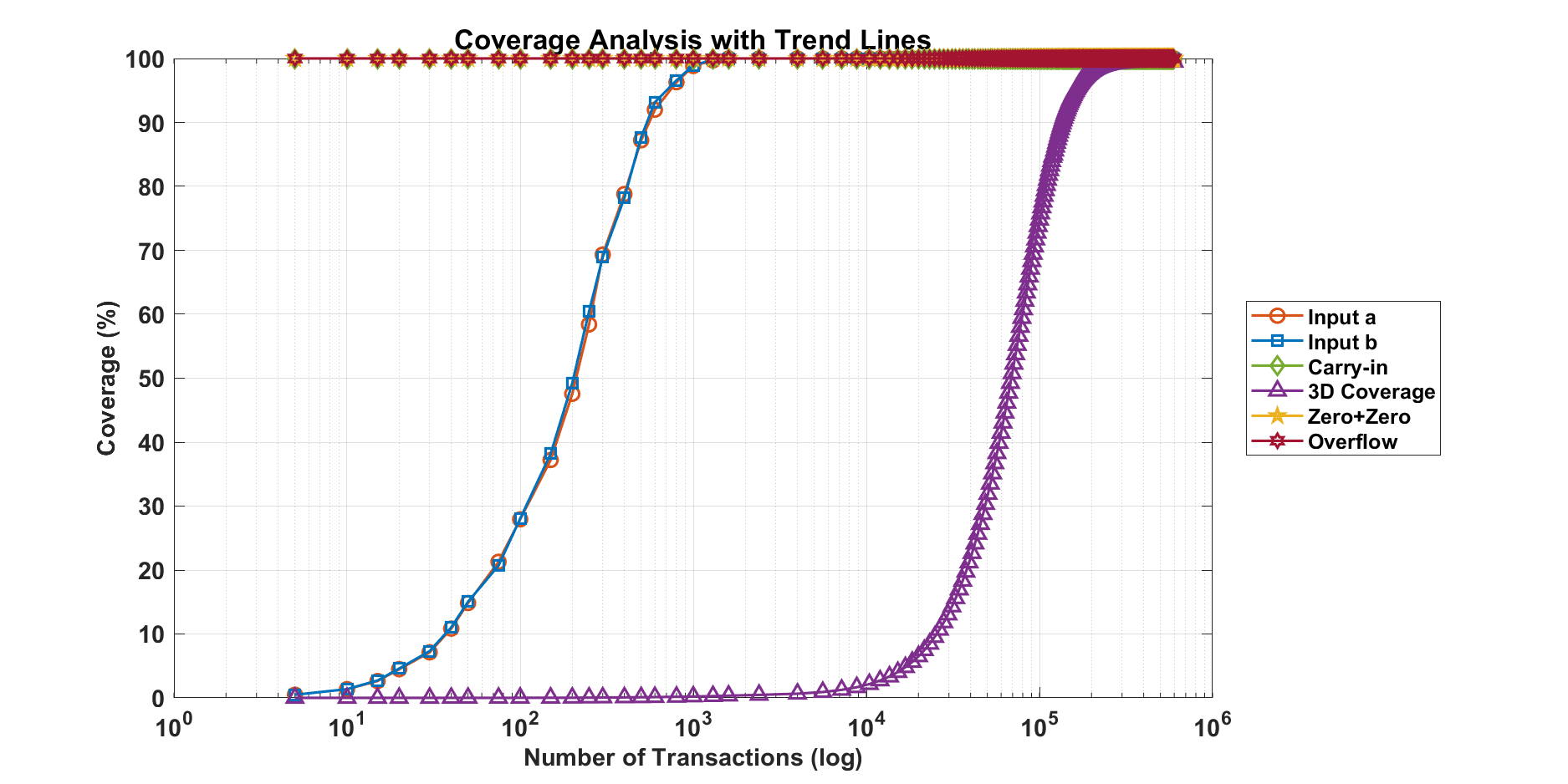


Figure : Coverage Analysis All in One Plot with Trend Lines

The coverage analysis graphs demonstrate comprehensive verification results from the layered testbench implementation. The data is plotted on logarithmic scales for transaction counts, providing clear visualization of coverage progression from 5 to 240,000 transactions.

The input coverage analysis shows rapid convergence for both inputs a and b, reaching nearly 100% coverage by 1000 transactions. The matching curves for inputs a and b indicate balanced testing of both operands. The steep rise between 100 and 1000 transactions demonstrates the effectiveness of the non-uniform transaction increment strategy in achieving broad input space coverage.

The carry-in coverage reaches 100% very early in the simulation, shown by the flat green line in the carry-in analysis plot. This indicates complete testing of both possible carry-in states (0 and 1) from the earliest test cases.

The 3D state space coverage, representing the cross-coverage of all inputs, shows a distinct sigmoid curve. The coverage remains low until about 10,000 transactions, then rises sharply to reach nearly 100% by 240,000 transactions. This behavior is expected due to the exponentially larger state space when considering all input combinations.

Special cases coverage (zero-zero inputs and overflow conditions) achieves 100% coverage early and maintains it throughout the simulation, indicated by the overlapping flat lines at 100%. This confirms that the testbench successfully identifies and verifies these critical corner cases from the beginning of the simulation.

The consolidated trend lines graph effectively shows the relationship between different coverage metrics. While individual input coverages converge quickly, the 3D coverage requires significantly more transactions to reach completion, validating the need for extensive random testing to achieve comprehensive verification coverage.

These results demonstrate that the layered testbench successfully achieves thorough verification coverage across all aspects of the design, with the progressive transaction count approach effectively balancing verification thoroughness with simulation efficiency.

### Interpretation and Conclusions on Data

**RTL and Testbenches:**

The comprehensive analysis of the 16-bit carry lookahead adder implementation demonstrates successful verification across multiple testing approaches. The RTL design, implemented with a hierarchical 4-bit module structure, shows correct functionality in adding 16-bit numbers with proper carry propagation and timing behavior.

The directed testbench results confirm proper operation across carefully selected test cases. The waveform analysis shows correct addition results for various scenarios, including small numbers (10+12), large numbers (65000+10), and corner cases like zero inputs and maximum values. The synchronous behavior with the clock and proper carry generation are clearly visible in the waveforms.

The layered testbench provides the most thorough verification coverage through its sophisticated 1024-bin analysis approach. The coverage metrics reveal several key insights:

Individual input coverage for a and b achieves rapid convergence, reaching near 100% by 1000 transactions. The carry-in coverage shows complete testing of both states very early in the simulation. The 3D state space coverage demonstrates more gradual progression, requiring approximately 240,000 transactions to reach complete coverage, reflecting the complexity of verifying all possible input combinations.

Special case coverage, including zero-zero inputs and overflow conditions, maintains 100% throughout the simulation, confirming robust handling of corner cases. The non-uniform transaction increment strategy proves effective, with smaller steps at lower counts catching early issues and larger steps efficiently handling extensive random testing.

These results validate both the correctness of the carry lookahead adder design and the effectiveness of the multi-layered verification approach. The combination of directed testing, self-checking capabilities, and comprehensive coverage analysis provides high confidence in the design's functionality across its entire operating range.

## Limitations of Tools (PO(e))

We used Cadence Tools for the RTL design and simulation. Cadence is an industry standard tool. So, it is one of the most updated tools available. Cadence tools, including Virtuoso and Genus, are powerful and widely used in the design and verification of integrated circuits, but they come with certain limitations. One of the main challenges is the complexity and steep learning curve associated with these tools, as they offer a vast array of features that require considerable expertise and experience to use effectively. Additionally, resource consumption is significant, especially for larger designs, as these tools demand high computational power and memory, which can slow down the design process. The **cost** of licensing Cadence tools is another limitation, as they tend to be expensive, particularly for smaller design teams or academic institutions.

## Impact Assessment (PO(f)).

### Assessment of Societal and Cultural Issues

Important societal and cultural challenges are brought up by the design and execution of a Carry Lookahead Adder (CLA), which we must take into account. Quick data processing is essential in sectors including healthcare, finance, and telecommunications, and the CLA's contribution to faster and more efficient computation benefits society as a whole. Improvements like speedier transaction processing and real-time diagnostics are made possible by this enhanced performance. Performance and sustainability must be balanced, though, as more powerful adders may have greater environmental effects due to their higher energy requirements.

Culturally, inclusive engineering practices are vital to ensure diverse teams are involved in technology development, leading to solutions that are equitable and consider the needs of different cultural contexts. We must ensure that their designs are accessible and beneficial to diverse populations, avoiding biases in computational models and improving the inclusivity of technological advancements. Balancing performance, sustainability, and inclusivity is an essential responsibility in the engineering of such systems.

### Assessment of Health and Safety Issues

The design and implementation of a Carry Lookahead Adder (CLA) generally present minimal direct health and safety risks. However, we must ensure the reliability and safety of the final product, especially when integrated into larger systems used in critical applications such as medical devices or automotive systems. Failures in these systems could jeopardize user safety. Additionally, we must consider potential risks related to electromagnetic interference (EMI) or heat dissipation in high-speed circuits, which can affect the performance and safety of electronic devices. Proper testing, adherence to safety standards, and quality control are essential to mitigate these concerns.

### Assessment of Legal Issues

The design of a Carry Lookahead Adder (CLA) involves several legal issues that we must address. First, intellectual property (IP) laws are important, as the design may be subject to patents, requiring engineers to ensure that their work does not infringe on existing patents. Additionally, compliance with industry standards and regulations, such as those set by the IEEE for electronic components, is crucial for legal acceptance and safety.

## Sustainability Evaluation (PO(g))

The sustainability of the Carry Lookahead Adder (CLA) design can be evaluated in terms of its environmental, economic, and social impact. From an environmental perspective, while the CLA improves performance and reduces delay, it may lead to higher energy consumption due to the increased complexity and speed of the circuit. This raises concerns about the long-term environmental impact. Economically, the design offers cost-effective improvements in speed, which can enhance the efficiency of larger systems, offering value in high-performance computing. Socially, the CLA contributes to advancements in sectors like healthcare and finance, benefiting society through faster and more reliable technology, though sustainability should balance these advancements with energy efficiency considerations.

## Ethical Issues (PO(h))

The design of a Carry Lookahead Adder (CLA) raises several ethical issues that we had addressed. First, ensuring transparency in the design process is essential, especially when the technology is used in critical applications like healthcare and finance, where errors could have significant consequences. We had a responsibility to ensure accuracy and reliability, preventing failures that could harm users. Additionally, the design should avoid biases in computational models, ensuring fairness and inclusivity.

# Reflection on Individual and Team work (PO(i))

## Individual Contribution of Each Member

|  |  |  |
| --- | --- | --- |
| **Name** | **ID** | **Contribution** |
| Anindya Kishore Choudhury | 1906081 | * Coordination * Literature Review * RTL Architecture Selection * RTL Design Code * Directed Testbench Code * Self-Checking Testbench Code * Parameterized Layered Testbench * Testbench Coverage Data Analaysis * Report Preparation |
| Toky Tazwar | 1906174 | * Synthesis File Generation * Power, Area and Timing calculation * Troubleshooting * SDC file and TCL file generation * Report Preparation |
| Akif Hamid | 1906192 | * Design Iterations * Troubleshooting * Fill as u did |
| Nusrat Jahan | 1906193 | * Core Implementation * Fill as u did |

## Mode of TeamWork

## Diversity Statement of Team

## Log Book of Project Implementation (Toky Fill this up and show me)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 1 | Group Formed | Anindya |  | Formed a team | Glad to be working in the same team |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 2 |  | Anindya |  | Motivated to Work Together | Everyone joined the meeting timely |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 3 | Project Finalisation | Anindya |  | Team worked Effectively |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 4 | Proposal Submission | Anindya |  | Team Work in done by cordinating in online setting |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week5 | Proposal Presentation | Anindya |  | The team worked overnight to present the project together |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 6 | Simulation Completed | Anindya |  | Team worked together in order to bring our expertise on the table to design a circuit and debug it for errors. |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 7 | Simulation Debugging | Anindya |  | Team has gathered the knowledge in debug the simulation to make it more effective in real life |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 8 | Components Listing | Anindya |  | Team tried to work under time pressure to pull it off |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 9 | Component testing | Anindya |  | Team worked at the central Library to do it |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 10 | Modular Structure Build up | Anindya |  | The team worked side by side to other project teams in this week to build the basic structure |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 11 | Debugging the Built Circuits Separately | Anindya |  | The team went above and beyond this week to make the circuit properly work |  |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 12 | Connecting all the modules and Verifying | Anindya |  | The team energetically pulled off the final demonstration. | The project is successful/ |
| Akif |  |
| Anila |  |
| Toky |  |
| **Date** | **Milestone achieved** | **Individual Role** | | **Team Role** | **Comments** |
| Week 13 | Final Presentation Preparation | Anindya |  | This is yet to be evaluated. | Yay! |
| Akif |  |
| Anila |  |
| Toky |  |

# Communication to External Stakeholders (PO(j))

This section is mandatory to write if the course outcomes address PO(j) Read the PO Statement in the website first, before writing this section ([Program Outcomes and Program Educational Objectives | Department of EEE, BUET](https://eee.buet.ac.bd/academics/undergraduate/peo-po)). . If not needed, remove this entire section. If needed, DO NOT Change the title of this section. Also, communication marks would consist writing this report properly with correct formatting.

## Executive Summary

Write a 100 word press release for your project that would be easy to understand by the masses.

## User Manual

## Github Link

Github Link: <https://github.com/anindyakchoudhury/ASIC-Project-Carry-Look-Ahead-Adder>

# Future Work (PO(l))

This section is mandatory to write if the course outcomes address PO(l) Read the PO Statement in the website first, before writing this section ([Program Outcomes and Program Educational Objectives | Department of EEE, BUET](https://eee.buet.ac.bd/academics/undergraduate/peo-po))

# References

|  |  |
| --- | --- |
| [1] | G. B. Rosenberger, "Simultaneous Carry Adder". US Patent 2966305, 27 December 1960. |
| [2] | N. H. E. W. a. D. M. Harris, CMOS VLSI design : a circuits and systems perspective., Noida: Pearson, 2015. |
| [3] | E. D. a. LangT, Digital arithmetic, San Francisco: Morgan Kaufmann Publishers, 2004. |
| [4] | P. B. a. D. L. Maskell, " “A New Carry Look-Ahead Adder Architecture Optimized for Speed and Energy,”," in *Electronics*, Sep. 2024. |
| [5] | G. B. Rosenberger.US Patent 2966305, 1960. |