Semester Long Internship Report

On

Designing of IP Blocks & Verification using Mixed Signal Simulation in eSim

Submitted by

Bellana Venkata Chaitanya

B.Tech (Electronics and Communication)
Rajiv Gandhi University of Knowledge Technologies, Nuzvid

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator,
Department of Aerospace Engineering,
Indian Institute of Technology Bombay

May 10, 2026

Acknowledgment

I would like to express my deepest gratitude to Prof. Prabhu Ramachandran for providing me the opportunity to be a part of the FOSSEE internship program and for supporting open-source engineering tool development at IIT Bombay. His guidance and vision have encouraged students and researchers to actively contribute to the open-source ecosystem.

I would also like to acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and nurturing the FOSSEE initiative. His contributions toward open-source education and the creation of the FOSSEE fellowship framework have directly shaped the platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K, for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

I would also like to acknowledge and thank Mr. P. Shyam, Assistant Professor and EITP Dean, Rajiv Gandhi University of Knowledge Technologies (RGUKT), for his constant support, encouragement, and guidance throughout this internship .

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to real-world circuit modelling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth

workflow, resource accessibility, and effective project execution

Contents

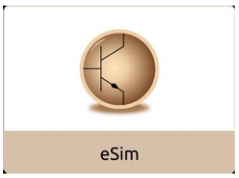
Chapter 1

Introduction

The advancement of Electronic Design Automation (EDA) tools has revolutionized the way electronic circuits are designed, simulated, and verified. Among these, open-source tools play a crucial role in making advanced EDA capabilities accessible to students, educators, and researchers without the burden of expensive licenses. The FOSSEE project at IIT Bombay has taken significant steps to promote such open-source tools globally.

One of the key open-source tools developed and promoted under FOSSEE is **eSim**, which integrates multiple open-source EDA tools to provide a unified platform for circuit design, simulation, and PCB layout. Various tools like NgSpice, Makerchip, KiCad, and GHDL work together under eSim to offer a complete design environment.

1.1 eSim



eSim is an open-source Electronic Design Automation (EDA) tool developed by FOSSEE, IIT Bombay. The primary objective of eSim is to provide a completely free and open-source alternative to commercial circuit design and simulation software, thereby reducing dependency on expensive proprietary tools in academia and research.

eSim integrates multiple open-source tools to provide a comprehensive design and simulation environment. KiCad is used for schematic capture and PCB design, while NgSpice performs analog circuit simulation. For digital simulation, GHDL is integrated supporting VHDL-based designs, and Makerchip enables online Transaction Level Verilog

(TL-Verilog) based digital design and verification. Python support is also available in eSim, allowing users to perform customized simulations and generate netlists.

One of the key features of eSim is its *Subcircuit* feature, which allows complex designs to be modularized into reusable blocks, simplifying the design of large systems. Additionally, eSim supports device modeling, enabling users to define and simulate custom semiconductor devices using real-world parameters. As part of its continuous development, eSim also supports SkyWater SKY130 PDK, allowing users to perform simulations using an open-source 130 nm process design kit suitable for analog, mixed-signal, and digital IC design research.

By supporting a fully open-source toolchain, eSim empowers students, educators, and researchers to gain hands-on experience with industry-relevant EDA tools and contribute actively to the open-source hardware ecosystem.

1.2 NgSpice

NgSpice is an open-source circuit simulator integrated into eSim for performing analog and mixed-signal simulations. Based on the SPICE simulation engine, NgSpice supports DC, AC, transient, and parametric analyses, making it suitable for analyzing a wide range of circuits.

It allows users to simulate circuit behavior using industry-standard SPICE models, including support for subcircuits and behavioral modeling. In eSim, NgSpice works as the backend simulator for schematics created using KiCad, providing waveform outputs for voltage, current, and frequency responses. Its flexibility and accuracy make it a powerful tool for verifying designs before hardware implementation.

1.3 Makerchip

Makerchip is an integrated platform designed to simplify digital circuit design by offering browser-based and desktop-based environments for coding, simulation, and debugging digital hardware designs. It supports multiple hardware description languages including Verilog, SystemVerilog, and Transaction-Level Verilog (TL-Verilog), allowing flexibility for users at various levels of abstraction.

In eSim, Makerchip is interfaced through the Makerchip IDE for digital design and verification. The platform integrates several open-source and proprietary tools to provide a rich set of features such as real-time simulation, waveform viewing, and code linting, thereby improving design accuracy and productivity.

1.4 KiCad

KiCad is an open-source PCB design and schematic capture tool integrated within eSim for creating circuit schematics and generating PCB layouts. It allows users to design multi-layer boards, define custom footprints, and perform design rule checks to ensure design correctness before fabrication.

In eSim, KiCad acts as the primary schematic editor, allowing users to graphically build circuits by placing and interconnecting components from extensive open-source libraries. The designed schematics can directly be used for both simulation and PCB layout generation. KiCad also supports features such as 3D visualization of PCBs, Gerber file export for manufacturing, and electrical rule checking to identify design issues early. Its seamless integration within eSim enables a smooth transition from schematic design to simulation and physical realization, offering a complete design-to-fabrication workflow entirely within an open-source environment.

1.5 GHDL

GHDL is an open-source simulator for VHDL, a hardware description language widely used in digital circuit design. It supports the complete IEEE VHDL standard, allowing designers to simulate, verify, and debug VHDL-based digital systems effectively.

In eSim, GHDL is integrated to enable digital simulation alongside analog simulation provided by NgSpice. Users can model digital circuits using VHDL, which are then simulated using GHDL to verify logical functionality and timing behavior. This integration allows eSim to handle mixed-signal designs, where the analog components are simulated by NgSpice and the digital components by GHDL, providing a comprehensive platform for complex system design. The combination of these tools allows designers to validate both analog and digital subsystems within a unified simulation environment.

Chapter 2

Features of eSim

eSim offers a comprehensive set of features that make it a powerful open-source alternative for electronic design automation. Some of the key features include:

- **Open-source and Free:** eSim is fully open-source, allowing unrestricted access without licensing costs, making it ideal for academic and research purposes.
- **Integrated Toolchain:** Combines multiple open-source tools such as KiCad for schematic capture and PCB design, NgSpice for analog simulation, GHDL for digital simulation, and Makerchip for advanced digital design.
- **Mixed-Signal Simulation:** Supports both analog and digital simulation, allowing users to design and simulate mixed-signal circuits efficiently.
- **Subcircuit Feature:** Enables hierarchical and modular design by allowing complex circuits to be broken into reusable subcircuits.
- **Device Modeling:** Supports custom device modeling, allowing users to simulate real-world semiconductor devices using user-defined parameters.
- **SkyWater SKY130 PDK Support:** Provides access to open-source 130 nm process design kit for IC design and simulation.
- **Python Integration:** Allows automation, scripting, and advanced analysis using Python interfaces.
- **User-Friendly Interface:** Offers an intuitive GUI that simplifies circuit creation, simulation setup, and result analysis, making it suitable for both beginners and advanced users.
- **Cross-Platform Support:** Compatible with major operating systems like Linux and Windows.

- **Active Community and Documentation:** Extensive documentation, tutorials, and community support provided through FOSSEE ensure smooth learning and troubleshooting.

2.1 Mixed-Signal Simulation using NGVeri

This part involves using eSim's NGVeri feature for mixed-signal simulation by integrating analog circuits with custom-designed Verilog digital IP blocks. NGVeri enables simultaneous simulation of analog (NgSpice) and digital (Verilator) domains, allowing realistic verification of mixed-signal systems.

In this work, several reusable digital IP blocks such as arithmetic units, sequential logic modules, and control circuits were designed in Verilog HDL and integrated into eSim. These IP blocks were then interfaced with analog circuits to validate their behavior in practical mixed-signal environments. This approach ensures that digital logic interacts correctly with real-world analog signals, improving system-level reliability and design accuracy.

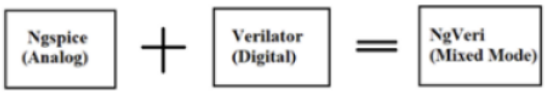


Figure 2.1: NGVeri mixed-signal simulation flow

2.1.1 Approach

- **Digital IP Design:** Reusable digital modules such as adders, multiplexers, counters, flip-flops, and control logic were designed using synthesizable Verilog HDL.
- **RTL Verification:** Each digital IP block was functionally verified using Icarus Verilog and GTKWave to ensure correct logical behavior before integration.
- **NGVeri Model Generation:** Verified Verilog modules were imported into NGVeri. The RTL code was compiled using Verilator to generate NGVeri-compatible mixed-signal models.
- **Analog–Digital Integration:** The generated digital IP blocks were instantiated within eSim schematics and connected with analog circuits designed using KiCad and NgSpice components.
- **Mixed-Signal Simulation:** Co-simulation was performed where analog signals from NgSpice interacted with digital logic executed by Verilator, enabling simultaneous analog–digital behavior analysis.

- **Result Verification:** Simulation outputs such as waveforms, timing relationships, and logic transitions were analyzed using eSim plotting tools to validate correct mixed-signal operation.
- **Reusable IP Validation:** The digital IP blocks were tested across multiple mixed-signal scenarios to confirm their reusability and reliability within the eSim design environment.

Chapter 3

Problem Statement

The objective of this project is to design and develop reusable Verilog-based Digital IP blocks and verify their functionality using mixed-signal simulation in eSim. The developed IPs are integrated into eSim through NGVeri to enable co-simulation of digital logic with analog circuits. This approach ensures correct interaction between digital modules and real-world analog signals, improving reliability and enabling system-level validation of digital designs within the eSim environment.

3.1 Digital IP Blocks Development

Digital IP blocks form the fundamental building units of complex digital systems. In this work, commonly used digital modules such as communication interfaces, arithmetic circuits, sequential elements, and control logic were designed as reusable RTL IPs in Verilog HDL. These IPs were structured to be modular, synthesizable, and compatible with NGVeri integration inside eSim.

The developed IP blocks were intended for repeated use across different mixed-signal design scenarios. Each IP was verified independently and later validated within analog–digital co-simulation setups in eSim.

3.1.1 Approach

- **IP Selection:** Frequently used digital modules including SPI protocol controllers, arithmetic units, adders, multipliers, dividers, and ALU were identified as reusable IP candidates.
- **RTL Design:** Selected IP blocks were implemented using synthesizable Verilog HDL following modular and hierarchical design practices.

- **Functional Verification:** Testbenches were created and simulated using Icarus Verilog and GTKWave to verify correct logical functionality and timing behavior.
- **IP Packaging:** Verified RTL modules were organized as reusable digital IP blocks suitable for integration into the eSim–NGVeri flow.

3.2 Mixed-Signal Simulation using NGVeri

To validate the developed digital IP blocks in realistic operating conditions, mixed-signal simulation was performed using eSim's NGVeri framework. NGVeri enables simultaneous co-simulation of analog circuits using NgSpice and digital logic using Verilator, allowing interaction between continuous analog signals and discrete digital logic.

In this work, the designed digital IP blocks were integrated into eSim schematics and interfaced with analog components. This enabled verification of digital logic behavior under analog stimulus conditions, ensuring correct functionality in mixed-signal environments.

3.2.1 Approach

- **NGVeri Model Generation:** Verified Verilog RTL modules were compiled using Verilator to generate NGVeri-compatible digital simulation models.
- **Analog–Digital Integration:** Generated digital IP models were instantiated within eSim schematics and connected with analog circuits created using KiCad and NgSpice components.
- **Co-Simulation Setup:** Mixed-signal simulation environment was configured where NgSpice handled analog behavior and Verilator executed digital logic concurrently.
- **Mixed-Signal Verification:** Digital IP outputs were validated under analog input conditions, ensuring correct logic transitions, timing response, and functional interaction.
- **System-Level Validation:** The reusable IP blocks were tested across multiple mixed-signal scenarios to confirm reliability and compatibility within the eSim design ecosystem.

Chapter 4

Digital IP Design and Integration

Digital IP blocks enable modular and scalable digital system design by encapsulating commonly used functionality into reusable components. In this work, the following eight IP blocks were designed, implemented in Verilog HDL, and verified using both standalone digital simulation and mixed-signal co-simulation in eSim via the NGVeri framework:

1. SPI Protocol
2. Leading Zero Counter
3. 7:3 Thermometric to Binary Wallace Tree Encoder
4. Booth Multiplier
5. Non-Restoring Divider IP
6. Ladner–Fischer Parallel Prefix Adder
7. CORDIC Engine
8. 16-Bit ALU Block

Each IP block was structured to be modular, synthesizable, and directly integrable into the eSim–NGVeri flow. The following sections detail the design, RTL implementation, schematic integration, and simulation results for each block.

4.1 SPI Protocol

4.1.1 Introduction

The Serial Peripheral Interface (SPI) is a synchronous, full-duplex serial communication protocol developed by Motorola in the 1980s. It has become one of the most widely adopted short-distance communication standards in embedded systems, FPGAs, and VLSI designs, enabling high-speed data exchange between a master controller and peripheral slave devices over as few as four wires.

This section presents the complete design, implementation, simulation, and verification of a 1-master 2-slave SPI system in Verilog HDL, operating in Mode 0 (CPOL=0, CPHA=0). The design was verified in two complementary environments: Xilinx Vivado for RTL simulation and eSim (KiCad + ngspice) for mixed-signal co-simulation.

4.1.2 Background and Technical Details

SPI Protocol Overview

SPI is a synchronous, master-driven protocol. The master generates all timing (SCLK) and selects which slave is active (SS). Unlike UART (which needs baud-rate matching) or I^2C (which requires address bytes and ACK handshaking), SPI is purely hardware-gated, making it the fastest and simplest serial bus at the PCB level.

Table 4.1: SPI standard signals

Signal	Full Name	Direction	Description
SCLK	Serial Clock	M→S	Clock generated by master
MOSI	Master Out Slave In	M→S	Serial data: master to slave
MISO	Master In Slave Out	S→M	Serial data: slave to master
SS/CS	Slave Select	M→S	Active-low; enables one slave

4.1.3 Block Diagram and FSM of the Design

System-Level Block Diagram

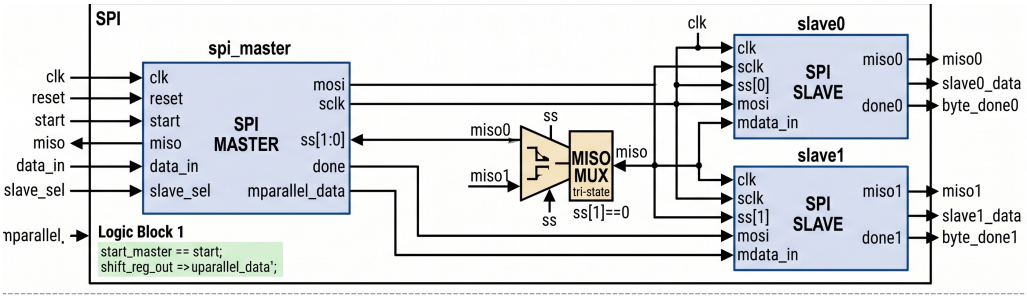


Figure 4.1: Multi-Slave SPI System Block Diagram.

The overall system consists of one master and two independently addressable slave modules connected via MOSI, MISO, SCLK, and SS lines. An active-low MISO MUX in the top-level wrapper selects the responding slave based on the active SS line. The master contains two `always` blocks: one for the 5-state Moore FSM (control) and one for the data path (shift registers and parallel capture).

Master Finite State Machine

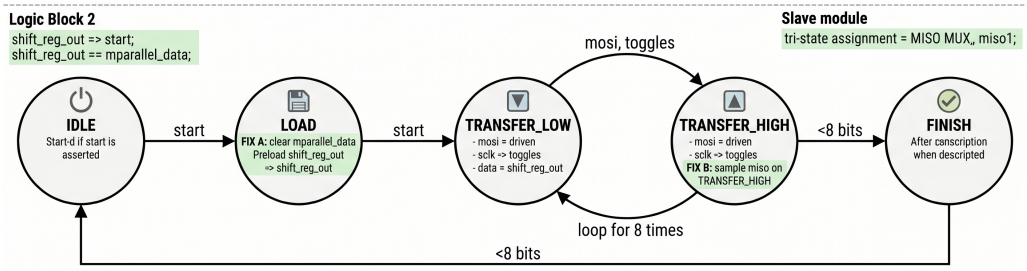


Figure 4.2: SPI Master FSM and Core Logic Flow.

Table 4.2: FSM state table

State	Enc.	SCLK	Key Actions	Next State
IDLE	3'd0	0	done=0	start=1 → LOAD
LOAD	3'd1	0	Assert SS, latch data_in, bit_cnt=0	Always → XFER_
TRANSFER_LOW	3'd2	0	drive MOSI = shift_reg_out[7-bit_cnt]	Always → XFER_
TRANSFER_HIGH	3'd3	1	increment bit_cnt	bit_cnt<7→LOW;
FINISH	3'd4	0	SS=2'b11, done=1	Always → IDLE

4.1.4 Verilog HDL Implementation

Top-Level Wrapper

```
module SPI (
    input wire clk,
    input wire reset,
    input wire start,
    input wire [7:0] data_in,
    input wire [7:0] mdata_in,
    input wire slave_sel,
    output wire [7:0] slave0_data,
    output wire [7:0] slave1_data,
    output wire done,
    output wire [7:0] mparallel_data
);
    wire mosi, sclk;
    wire [1:0] ss;
    wire miso0, miso1;
    wire miso;

    assign miso = (ss[0] == 1'b0) ? miso0 :
                  (ss[1] == 1'b0) ? miso1 : 1'b0;

    spi_master master (
        .clk(clk), .reset(reset), .start(start),
        .miso(miso), .data_in(data_in), .slave_sel(slave_sel),
        .mosi(mosi), .sclk(sclk), .ss(ss), .done(done),
        .mparallel_data(mparallel_data)
    );
    spi_slave slave0 (
        .clk(clk), .sclk(sclk), .ss(ss[0]), .mosi(mosi),
        .mdata_in(mdata_in), .miso(miso0),
        .parallel_data(slave0_data), .byte_done()
    );
    spi_slave slave1 (
        .clk(clk), .sclk(sclk), .ss(ss[1]), .mosi(mosi),
        .mdata_in(mdata_in), .miso(miso1),
        .parallel_data(slave1_data), .byte_done()
    );
endmodule
```

Listing 4.1: SPI.v – top-level wiring wrapper and MISO MUX

Master FSM Block

```
module spi_master (
    input  wire clk,
    input  wire reset,
    input  wire start,
    input  wire miso,
    input  wire [7:0] data_in,
    input  wire slave_sel,
    output reg  mosi,
    output reg  sclk,
    output reg  [1:0] ss,
    output reg  done,
    output reg  [7:0] mparallel_data
);
    reg [2:0] state;
    localparam IDLE          = 3'd0,
               LOAD          = 3'd1,
               TRANSFER_LOW  = 3'd2,
               TRANSFER_HIGH = 3'd3,
               FINISH        = 3'd4;
    reg [7:0] shift_reg_out;
    reg [2:0] bit_cnt;
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            state   <= IDLE;
            sclk    <= 0;
            ss      <= 2'b11;
            mosi    <= 0;
            done    <= 0;
            bit_cnt <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                    sclk <= 0;
                    if (start) state <= LOAD;
                end
                LOAD: begin
                    ss      <= (slave_sel == 0) ? 2'b10 : 2'b01;
                    bit_cnt <= 0;
                    state   <= TRANSFER_LOW;
```

```
                end
                TRANSFER_LOW: begin
                    sclk  <= 0;
                    mosi  <= shift_reg_out[7 - bit_cnt];
                    state <= TRANSFER_HIGH;
                end
                TRANSFER_HIGH: begin
                    sclk <= 1;
                    if (bit_cnt == 3'd7) state <= FINISH;
                    else begin
                        bit_cnt <= bit_cnt + 1;
                        state   <= TRANSFER_LOW;
                    end
                end
                FINISH: begin
                    sclk  <= 0;
                    ss    <= 2'b11;
                    done  <= 1;
                    state <= IDLE;
                end
                default: state <= IDLE;
            endcase
        end
    end
 always @(posedge clk or posedge reset) begin
        if (reset) begin
            shift_reg_out  <= 8'h00;
            mparallel_data <= 8'h00;
        end else begin
            if (state == LOAD) begin
                shift_reg_out  <= data_in;
                mparallel_data <= 8'h00;
            end

            if (state == TRANSFER_HIGH) begin
                mparallel_data <= {mparallel_data[6:0], miso};
            end
        end
    end
endmodule
```

Listing 4.2: spi_master.v – FSM control block

Slave Module

```
module spi_slave (
    input  wire clk,
    input  wire sclk,
    input  wire ss,
    input  wire mosi,
    input  wire [7:0] mdata_in,
    output wire miso,
    output reg  [7:0] parallel_data,
    output reg  byte_done
);
    reg [2:0] bit_cnt;
    reg [7:0] shift_reg_in;
    reg [7:0] shift_reg_out;
    reg sclk_delayed;
    wire sclk_rising = (sclk && !sclk_delayed);

    assign miso = ss ? 1'bz : shift_reg_out[7 - bit_cnt];

    always @(posedge clk or posedge ss) begin
        if (ss) begin
            bit_cnt       <= 0;
            byte_done     <= 0;
            sclk_delayed  <= 0;
            shift_reg_out <= mdata_in;     // preload all 8 bits
            shift_reg_in  <= 8'h00;
        end else begin
            sclk_delayed <= sclk;
            if (sclk_rising) begin
                shift_reg_in <= {shift_reg_in[6:0], mosi};
                bit_cnt      <= bit_cnt + 1;
                if (bit_cnt == 7) begin
                    parallel_data <= {shift_reg_in[6:0], mosi};
                    byte_done     <= 1;
                end
            end
        end
    end
endmodule
```

Listing 4.3: spi_slave.v – combinational MISO + registered MOSI sampling

4.1.5 Simulation Results

Xilinx Vivado Waveform

Figure 4.3 shows the Xilinx Vivado simulation waveform for all four test transfers. The `mparallel_data` register builds up bit-by-bit and settles to the correct value when `done` pulses high.

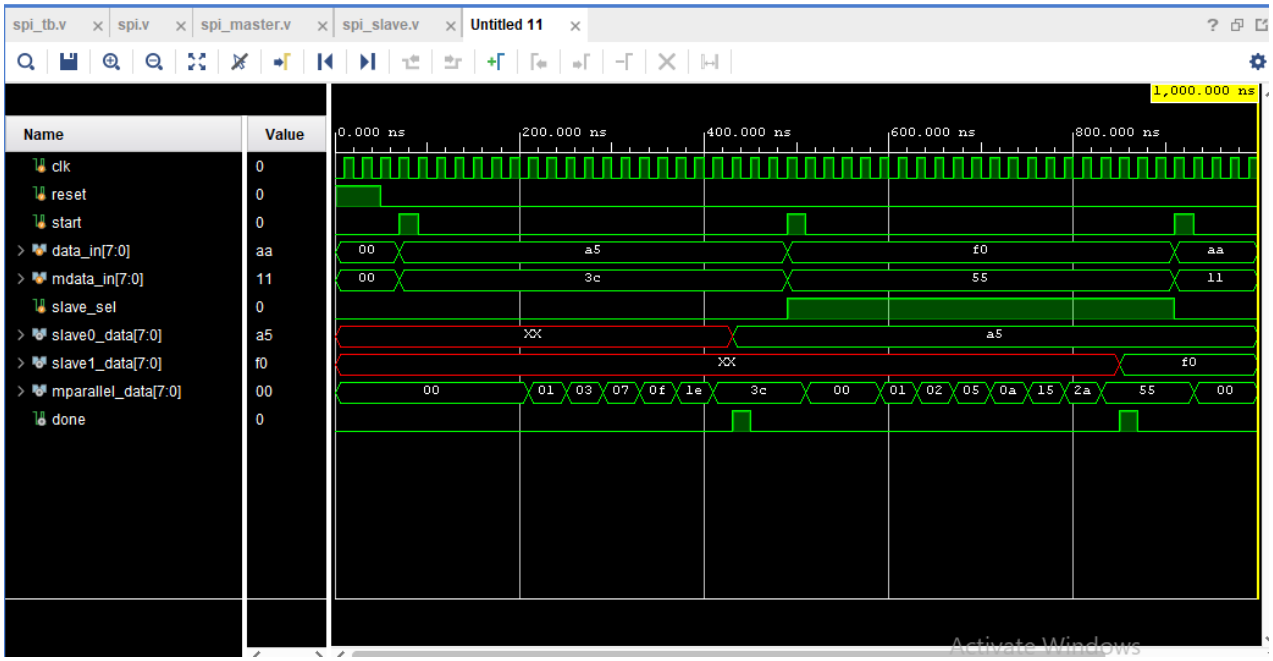


Figure 4.3: Xilinx Vivado simulation waveform showing all 4 test transfers

Table 4.3: Vivado simulation – all tests pass

Test	Slave	data_in	mdata_in	mparallel_data	Pass?
1	0	0xA5	0x3C	0x3C	YES
2	1	0xF0	0x55	0x55	YES
3a	0	0xAA	0x11	0x11	YES
3b	1	0xBB	0x22	0x22	YES

eSim Schematic

Figure 4.4 shows the eSim schematic where voltage sources drive all 8 `data_in` and `mdata_in` bits through ADC bridges into the SPI digital subcircuit, and DAC bridges feed the output buses to plot probes.

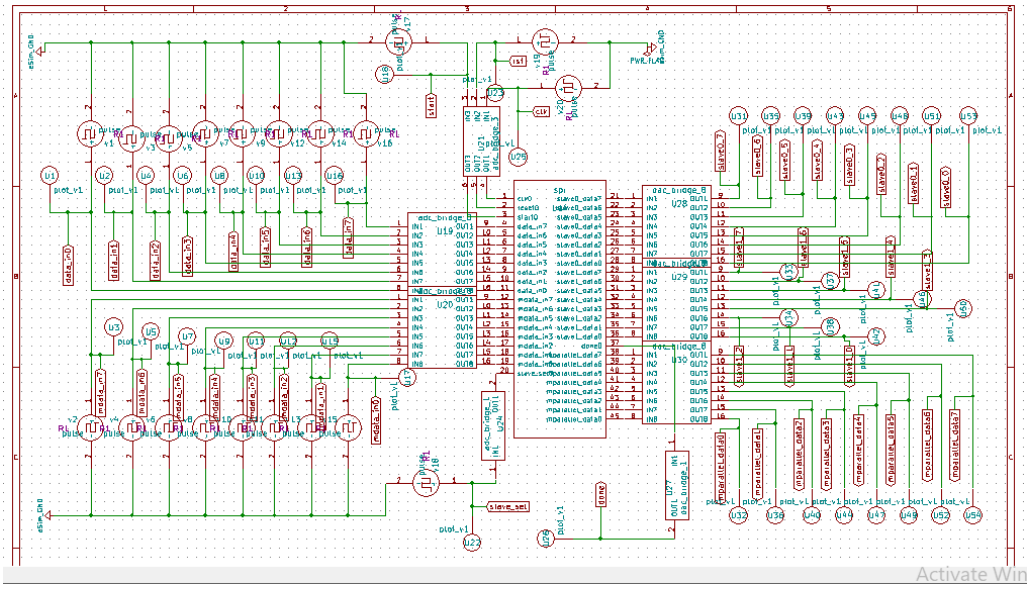


Figure 4.4: eSim schematic of the SPI system with ADC/DAC bridges

eSim Transient Waveforms

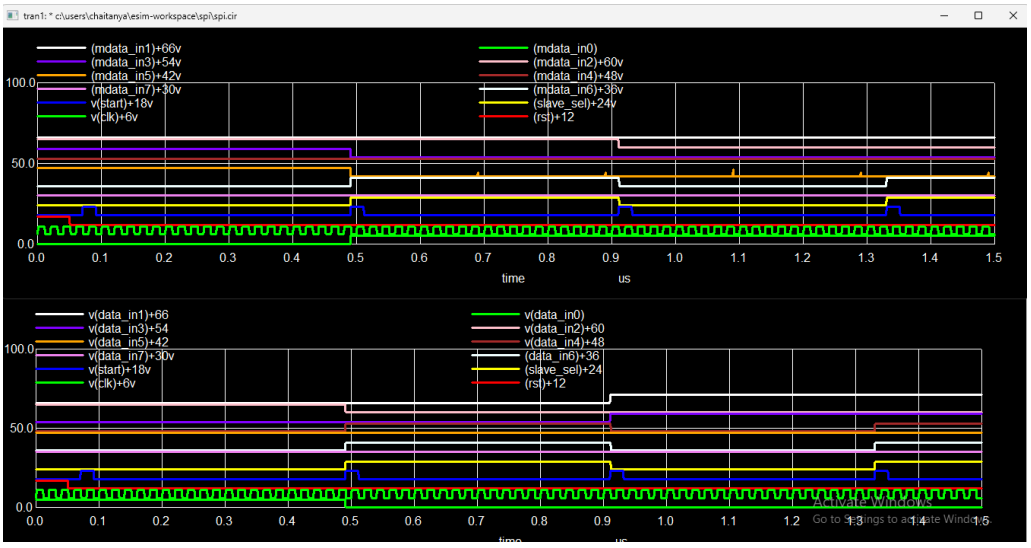


Figure 4.5: shows the ngspice transient waveforms (1.5 μs) for `mdata_in[7:0]`, `data_in[7:0]`, and the control signals. Transfer transitions are visible at $\approx$0.5 μs (Test 1), 0.9 μs (Test 2), and 1.3 μs (Tests 3a/3b).

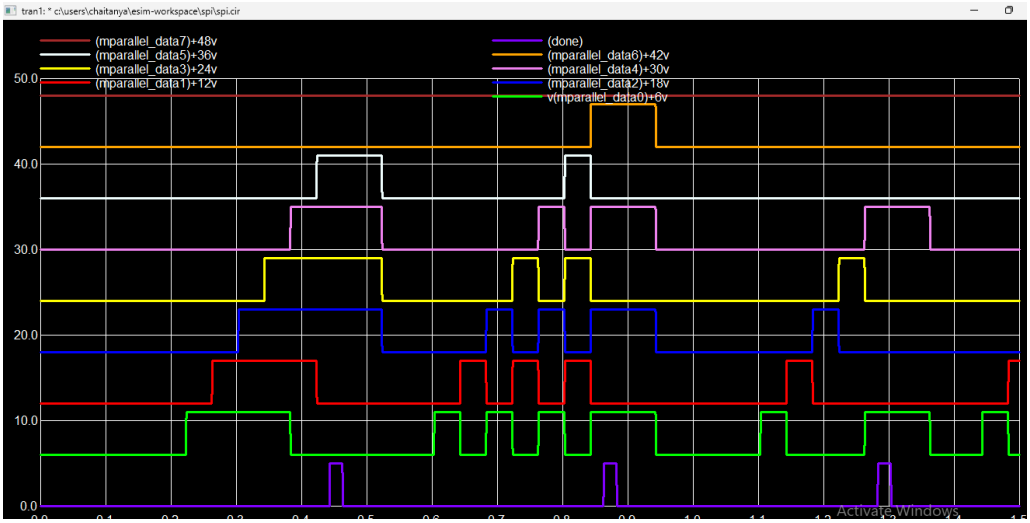


Figure 4.6: shows the individual bits of `mparallel_data[7:0]` during the 1.5 μs simulation; each bit toggles at the correct time confirming MSB-first serial-to-parallel conversion. The purple `done` pulse marks each completed transfer.

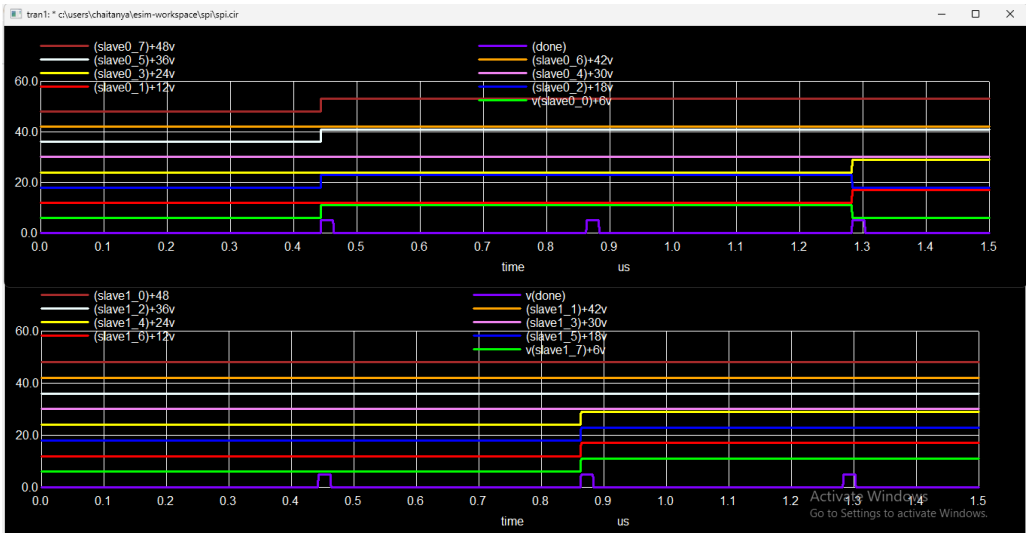
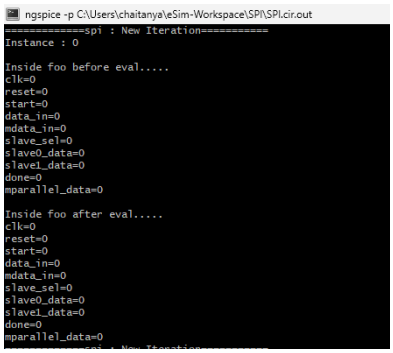
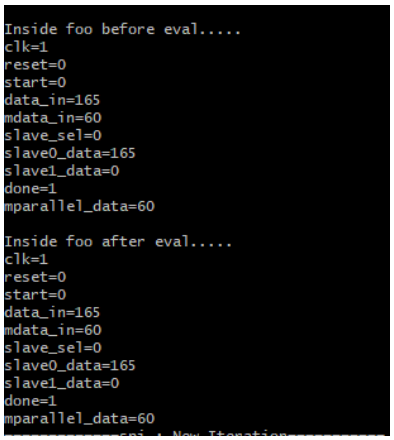


Figure 4.7: shows the slave data waveforms: the upper panel shows `slave0_data[7:0]` and the lower panel shows `slave1_data[7:0]`. Both slaves receive the correct bytes from the master.

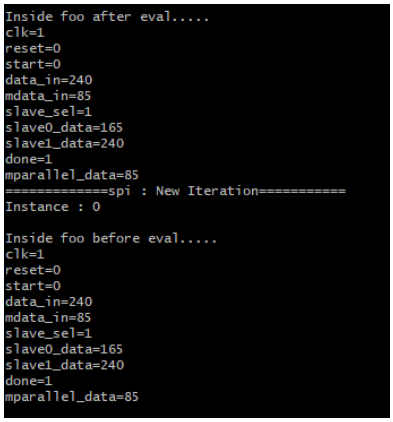
ngspice Console Output



(a) Iteration 0: reset state, all signals = 0



(b) Test 1: data_in=165 (0xA5) mparallel_data=60 (0x3C)



(c) Test 2: data_in=240 (0xF0) mparallel_data=85 (0x55)

Figure 4.8: ngspice console output confirming correct decimal values at each iteration. All `done=1` flags confirm transfer completion.

4.1.6 Advantages and Applications

SPI offers very high throughput compared to I^2C and UART, supports full-duplex operation, requires no ACK handshaking, and uses simple hardware slave-select addressing. It is widely used in memory devices (SPI Flash, EEPROM, SD cards), sensors (IMU, temperature, pressure), displays (TFT/OLED), data converters (ADC/DAC), wireless modules (Wi-Fi, Bluetooth), automotive ECU peripherals, FPGA bitstream loading, and SPI-controlled power management ICs.

4.1.7 Conclusion

The full-duplex 1-master 2-slave SPI Mode 0 design was correctly implemented using a 5-state Moore FSM and a dual `always`-block data path. The combinational MISO assignment and one-cycle registered-state lag naturally align MISO sampling to the rising SCLK edge. All four test cases pass in both Vivado and eSim/ngspice, making the design suitable for system-level analog–digital co-simulation flows.

4.2 Leading Zero Counter

4.2.1 Introduction

In modern digital systems, the ability to rapidly identify the position of the most-significant 1-bit in a binary word is critical for floating-point units, normalizers, and priority arbiters. The Leading Zero Detector (LZD) counts the number of leading (most-significant) zero bits in a binary word and outputs the result as a binary count.

Unlike sequential bit-scan methods, a hardware LZD uses a parallel hierarchical tree to compute the count in $O(\log_2 N)$ gate delays. For a 16-bit input, this means only 4 logic levels are required regardless of where the leading 1 appears.

This section presents a 16-bit LZD implemented using a divide-and-conquer approach: four 4-bit Leading Zero Count (`lzc4`) sub-blocks process the input in parallel, and their results are combined by a priority encoder and a 4-to-1 multiplexer to yield a 4-bit output count and a global `all_zeros` flag.

4.2.2 Background and Theory

The Leading Zero Detection Algorithm

The LZD algorithm determines the position of the most-significant 1-bit in an N-bit binary word $B = b_{N-1}b_{N-2}\cdots b_1b_0$. The leading zero count is formally defined as:

$$\mathrm{LZC}(B) = \begin{cases} N & \text{if } B = 0 \\ N - 1 - \lfloor \log_2 B \rfloor & \text{otherwise} \end{cases}$$

For a 16-bit word the result lies in the range $[0, 15]$, fitting exactly in 4 bits. The hierarchical approach divides the 16-bit input into four 4-bit nibbles processed in parallel, merged in two further levels (encoder + MUX), giving a total depth of $O(\log_2 16) = 4$ gate levels.

4-bit LZC Sub-block

The `lzc4` module accepts a 4-bit input $x[3:0]$ and produces a 2-bit leading-zero count $z[1:0]$ and an all-zero flag a.

Table 4.4: Truth table for the 4-bit LZC sub-block

Input x[3:0] (casez)	z[1:0]	a
1???	00 (0)	0
01??	01 (1)	0
001?	10 (2)	0
0001	11 (3)	0
0000	xx	1

Priority Encoder and MUX Logic

Once the four `lzc4` blocks produce their all-zero flags a_0, a_1, a_2, a_3, a 4-to-2 priority encoder selects which nibble contains the leading 1:

Table 4.5: Priority encoder truth table (generates upper bits q32)

a0	a1	a2	a3	q32	Selected nibble
0	x	x	x	00	nibble 0: in[15:12]
1	0	x	x	01	nibble 1: in[11:8]
1	1	0	x	10	nibble 2: in[7:4]
1	1	1	0	11	nibble 3: in[3:0]
1	1	1	1	11	all zeros (default)

4.2.3 Architecture

Architecture Block Diagram

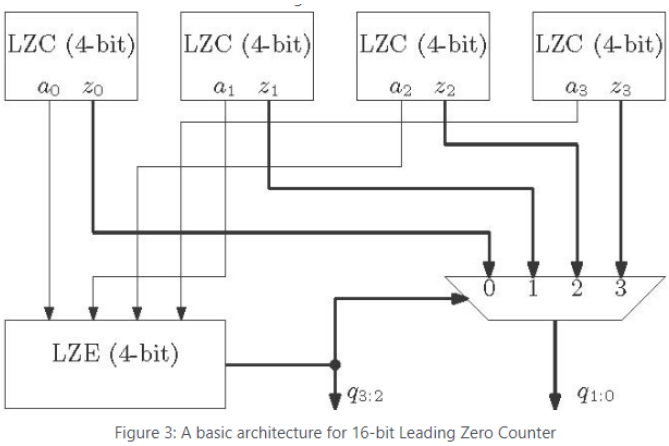


Figure 3: A basic architecture for 16-bit Leading Zero Counter

Figure 4.9: Architecture of the 16-bit Leading Zero Counter

4.2.4 Verilog HDL Implementation

lzc4 Sub-module

```
module lzc4 (
  input  [3:0] x,
  output reg [1:0] z,
  output       a
);
  assign a = (x == 4'b0000);
  always @(*) begin
    casez (x)
      4'b1???: z = 2'b00;  // 0 leading zeros
      4'b01??: z = 2'b01;  // 1 leading zero
      4'b001?: z = 2'b10;  // 2 leading zeros
      4'b0001: z = 2'b11;  // 3 leading zeros
      default: z = 2'bxx;  // all-zeros: don't care
    endcase
  end
endmodule
```

Top-Level LZD Module

```
`timescale 1ns/1ps
module LZD (
  input  [15:0] in,
  output  [3:0] out,
  output        all_zeros
);
  wire [1:0] z0, z1, z2, z3;
  wire a0, a1, a2, a3;
  reg  [1:0] q32;   // upper 2 bits from priority encoder
  reg  [1:0] q10;   // lower 2 bits selected via 4:1 MUX
  lzc4 block0(in[15:12], z0, a0);  // MSB nibble
  lzc4 block1(in[11:8],  z1, a1);
  lzc4 block2(in[7:4],   z2, a2);
  lzc4 block3(in[3:0],   z3, a3);  // LSB nibble
  always @(*) begin
    casez ({a0,a1,a2,a3})
      4'b0???: q32 = 2'b00;
      4'b10??: q32 = 2'b01;
      4'b110?: q32 = 2'b10;
```

```
      4'b1110: q32 = 2'b11;
      default: q32 = 2'b11;
    endcase
  end
  always @(*) begin
    case (q32)
      2'b00: q10 = z0;
      2'b01: q10 = z1;
      2'b10: q10 = z2;
      2'b11: q10 = z3;
    endcase
  end
  assign out        = {q32, q10};
  assign all_zeros = a0 & a1 & a2 & a3;
endmodule
```

4.2.5 eSim Schematic and Simulation Setup

Schematic Description

The 16-bit LZD was simulated using eSim. Analog PWL voltage sources drive the 16-bit input bus through ADC bridge blocks; the Verilog LZD model processes the logic; and DAC bridge blocks convert `out[3:0]` and `all_zeros` back to analog voltages for plotting.

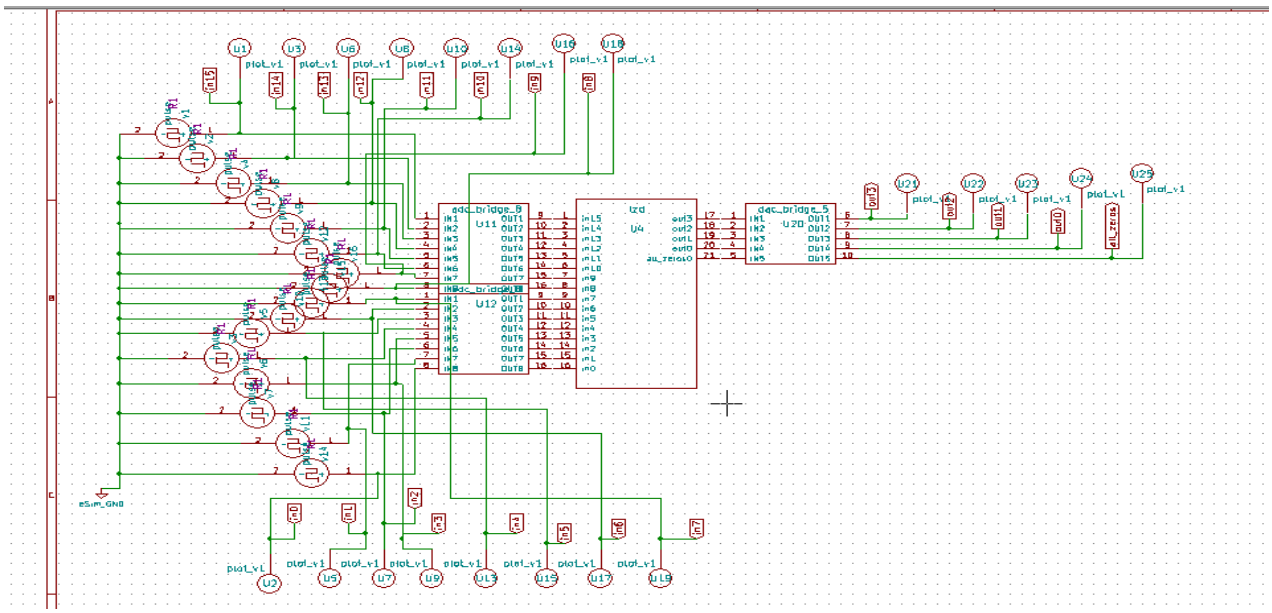


Figure 4.10: eSim schematic of the 16-bit LZD system

4.2.6 Ngspice Simulation Output and Verification

Console Output Analysis

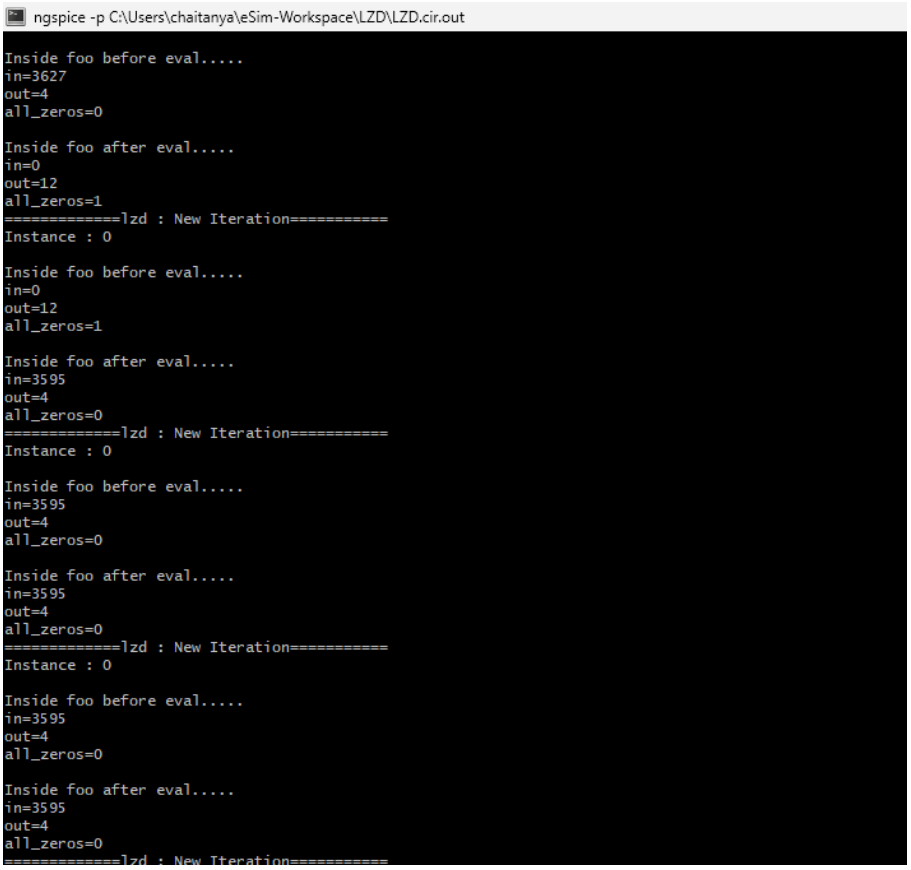


Figure 4.11: Ngspice console output showing iterative LZD simulation results

Figure 4.11 shows the Ngspice terminal output. The simulation uses an event-driven mixed-signal model where the Verilog subcircuit is evaluated iteratively at each time step. Three distinct input patterns are observed:

Table 4.6: Ngspice console output summary for the LZD

in (dec)	**in (16-bit binary)**	**out**	**all_zeros**
3627	0000111000101011	4	0
0	0000000000000000	12	1
3595	0000111000001011	4	0

4.2.7 Waveform Analysis

Input Waveforms

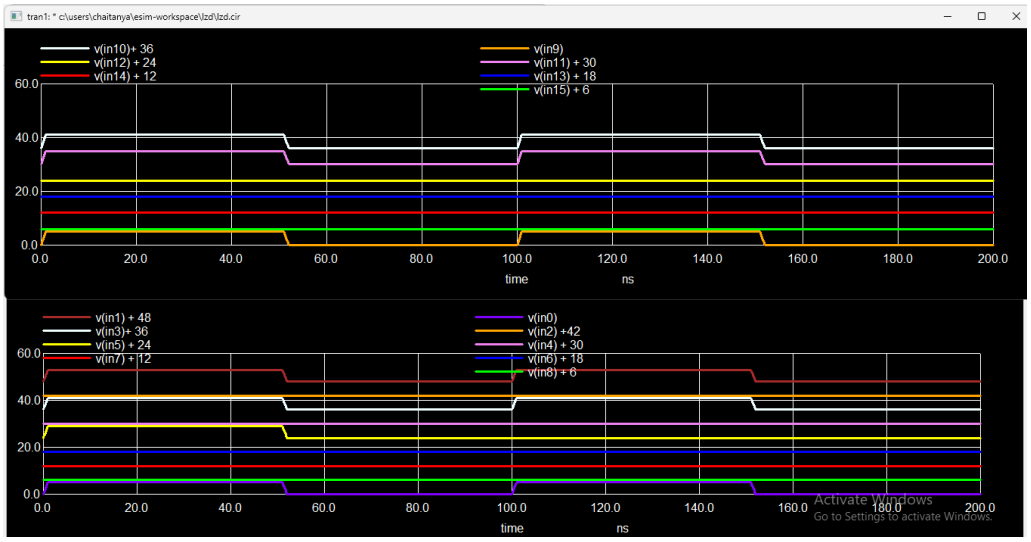


Figure 4.12: Ngspice transient waveforms: 16 input signals v(in0)–v(in15) over 200 ns, shown in two panels with vertical offsets for clarity

Output Waveforms

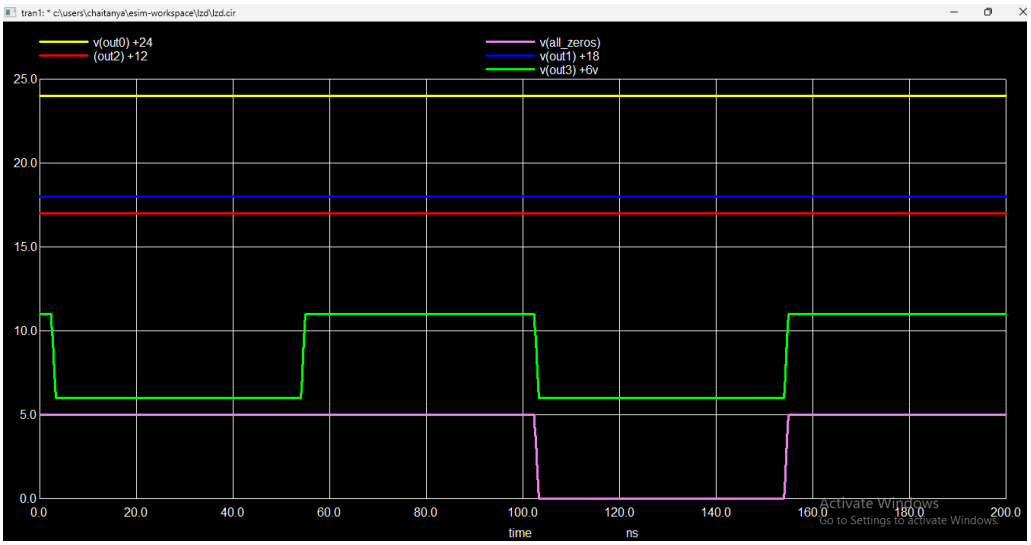


Figure 4.13: Ngspice transient waveforms: output bits v(out0)–v(out3) and v(all_zeros) over 200 ns

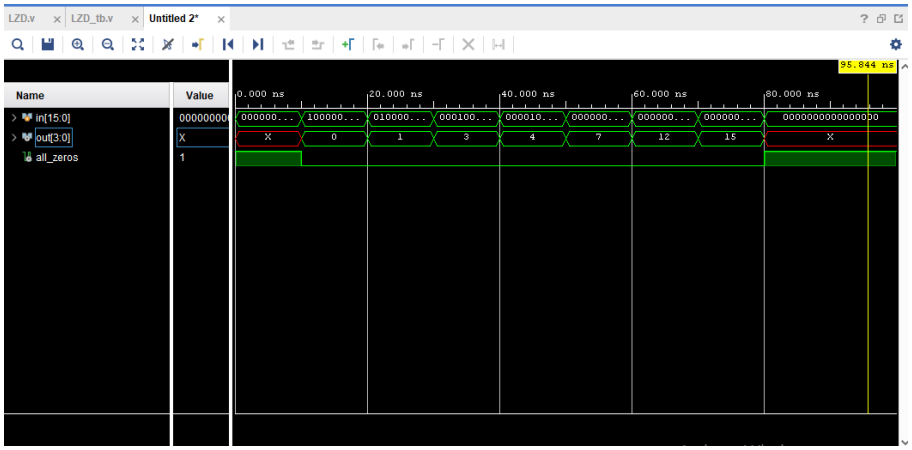


Figure 4.14: GTKWave RTL simulation: `in[15:0]`, `out[3:0]`, and `all_zeros` over 95 ns

Table 4.7: Output waveform decoding: time interval vs. encoded leading zero count

Time (ns)	**out[3]**	**out[2]**	**out[1]**	**out[0]**	**Decimal**	**Interpretation**
0 – 50	0	1	0	0	4	4 leading zeros in input
50 – 100	0	1	0	0	4	Stable at count = 4
100–155	x	x	x	x	–	all_zeros = 1 (in = 0)
155–200	0	1	0	0	4	Input restored; count = 4

4.2.8 Applications

The LZD is used in IEEE 754 floating-point normalization (shift-amount computation), barrel shifter control, priority encoders for bus arbitration and interrupt controllers, division hardware for operand alignment, run-length encoders, and hardware bit-manipulation instructions in cryptographic engines.

4.2.9 Advantages

Key advantages: $O(\log_2 N)$ delay, full parallelism in Stage 1, minimal hardware (four `lzc4` instances plus encoder and MUX), modular and reusable sub-blocks, technology-independent Verilog, glitch-free `casez` priority structure, and clean scalability to 32/64 bits.

4.3 7:3 Thermometric to Binary Wallace Tree Encoder

4.3.1 Introduction

In modern digital systems, the speed of arithmetic operations determines the overall performance of processors, signal processors, and communication ICs. The Wallace Tree structure, introduced by Chris S. Wallace in 1964, provides an elegant and highly efficient solution to the problem of reducing multiple partial products or summing multiple bits.

A Wallace Tree Encoder is a multi-operand adder network that reduces N bits of equal weight into a sum and a carry using the minimum possible number of logic levels. Unlike ripple-carry or array adders that process bits serially, the Wallace Tree exploits maximum parallelism and reduces the critical path delay from $O(N)$ to $O(\log_{1.5} N)$.

Flash ADCs generate thermometric codes where the number of 1s directly encodes the analog voltage level. The Wallace Tree Encoder replaces a conventional priority encoder with a parallel popcount circuit that operates at significantly higher speeds, making it ideal for GHz-range ADC designs.

4.3.2 Background and Theory

The Wallace Tree Algorithm

The Wallace Tree algorithm reduces N input bits into two output numbers (sum and carry) whose binary sum equals the sum of all N input bits. The key insight is that a full adder (FA) takes three inputs and produces two outputs: a sum bit and a carry bit. Three bits of weight 2^k can be reduced to one sum bit of weight 2^k and one carry bit of weight 2^{k+1} – the fundamental "3-to-2" reduction.

The number of reduction stages for N inputs is:

$$\text{Stages} = \lceil \log_{1.5} N \rceil$$

For $N = 7$ inputs, due to the symmetric equal-weight structure, only 3 full-adder stages are required.

Thermometric Code and ADC Encoding

In a flash ADC, the analog input is compared against $2^N - 1$ reference voltages. The output is a thermometric code: all bits below the analog input are 1 and all above are 0. The Wallace Tree Encoder counts the number of 1s (a population count) and outputs the binary equivalent.

Table 4.8: 3-bit flash ADC thermometric output for 7 comparators

Level	C6	C5	C4	C3	C2	C1	C0	Binary Out
0	0	0	0	0	0	0	0	000
1	0	0	0	0	0	0	1	001
2	0	0	0	0	0	1	1	010
3	0	0	0	0	1	1	1	011
4	0	0	0	1	1	1	1	100
5	0	0	1	1	1	1	1	101
6	0	1	1	1	1	1	1	110
7	1	1	1	1	1	1	1	111

4.3.3 Architecture

Overview

The encoder accepts a 7-bit thermometric input `thermo_in[6:0]` and produces a 3-bit binary output `binary_out[2:0]` using exactly four full adders arranged in three stages.

Signal	Direction	Width	Description
thermo_in	Input	7 bits	Thermometric code from ADC comparators
binary_out	Output	3 bits	Binary-encoded popcount output

4.3.4 Block Diagram

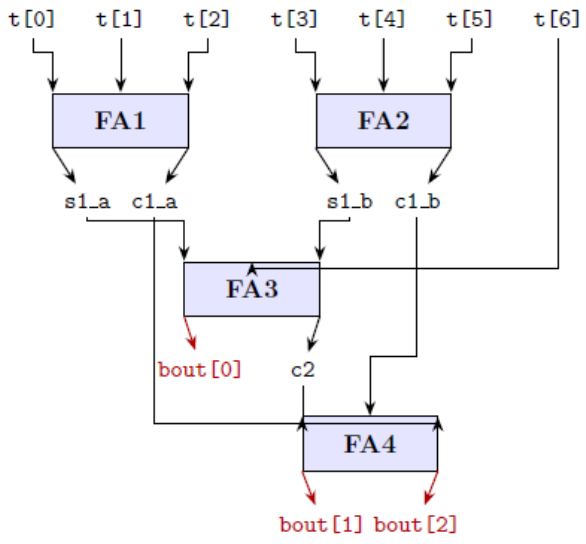


Figure 4.15: Wallace Tree Encoder Reduction Tree (7-to-3 mapping)

Stage-by-Stage Description

Stage 1 – Parallel Reduction (FA1 and FA2): FA1 processes `thermo_in[0,1,2]` producing s_{1a}, c_{1a}; FA2 processes `thermo_in[3,4,5]` producing s_{1b}, c_{1b}. Both operate in parallel.

Stage 2 – LSB Generation (FA3): FA3 combines s_{1a}, s_{1b}, and `thermo_in[6]`: sum → `binary_out[0]` (final LSB); cout → c_2 (intermediate carry).

Stage 3 – Weight-2 and Weight-4 Reduction (FA4): FA4 combines c_{1a}, c_{1b}, c_2: sum → `binary_out[1]` (middle bit); cout → `binary_out[2]` (MSB).

4.3.5 Verilog HDL Implementation

Full Adder Module

```
module full_adder (
  input  a, b, cin,
  output sum, cout
);
  wire a_xor_b;
  xor g1(a_xor_b, a, b);
  xor g2(sum, a_xor_b, cin);
  assign cout = (a & b) | (cin & a_xor_b);
endmodule
```

Top-Level Wallace Tree Encoder

```
module wallace_adc_encoder_7_3 (
  input  [6:0] thermo_in,    // from 7 ADC comparators
  output [2:0] binary_out );   // 3-bit digital output
  wire s1_a, c1_a;
  wire s1_b, c1_b;
  wire s2,   c2;
full_adder fa1(.a(thermo_in[0]),.b(thermo_in[1]),
                 .cin(thermo_in[2]),.sum(s1_a),.cout(c1_a));
  full_adder fa2(.a(thermo_in[3]),.b(thermo_in[4]),
                 .cin(thermo_in[5]),.sum(s1_b),.cout(c1_b));
  full_adder fa3(.a(s1_a),.b(s1_b),.cin(thermo_in[6]),
                 .sum(binary_out[0]),.cout(c2));
  full_adder fa4(.a(c1_a),.b(c1_b),.cin(c2),
                 .sum(binary_out[1]),.cout(binary_out[2]));
endmodule
```

4.3.6 eSim Schematic and Simulation Setup

The Wallace Tree Encoder was simulated using eSim, integrating KiCad for schematic capture and Ngspice for mixed-mode circuit simulation. The design combines an analog voltage-source front-end (representing comparator outputs) with a digital Verilog subcircuit for the Wallace Tree logic.

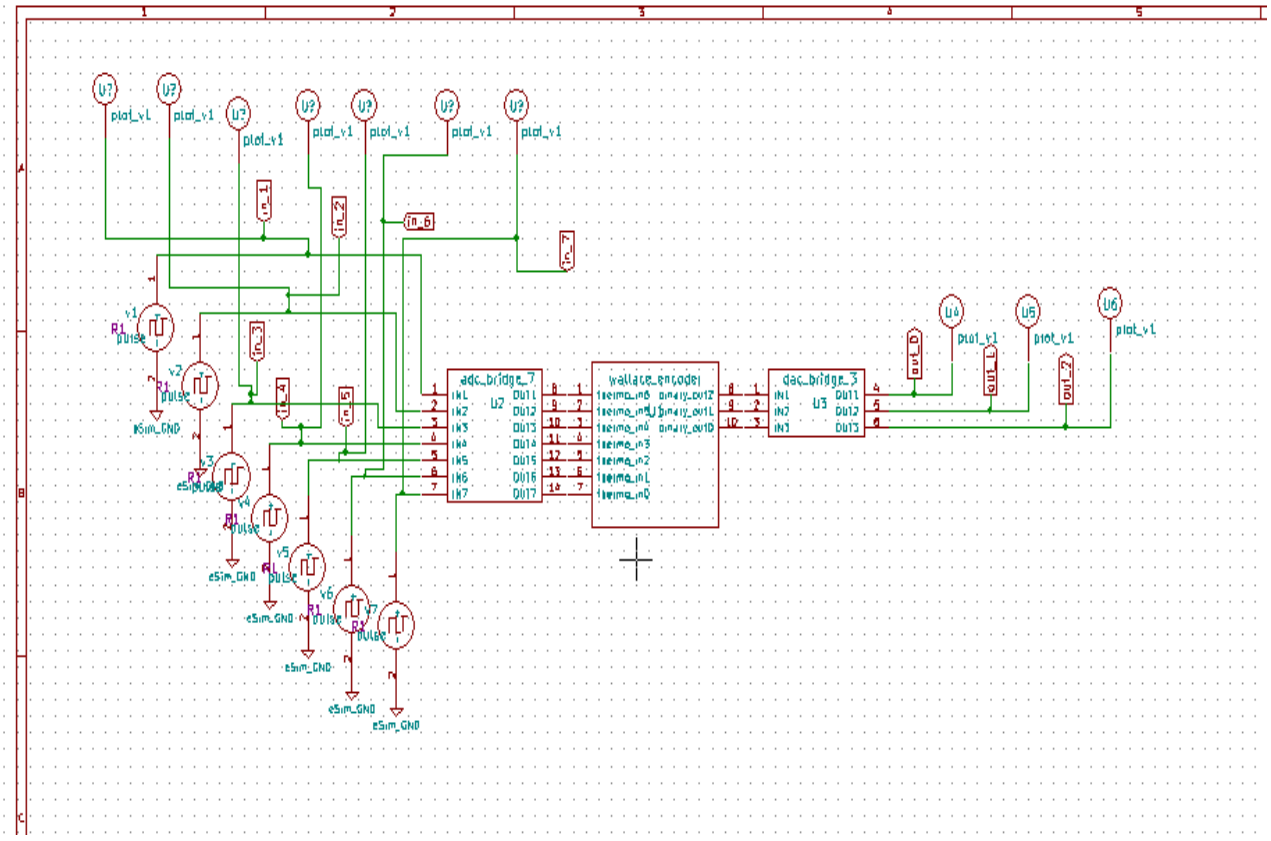


Figure 4.16: eSim schematic of the Wallace Tree ADC encoder system

4.3.7 Ngspice Simulation Output and Verification

Console Output Analysis

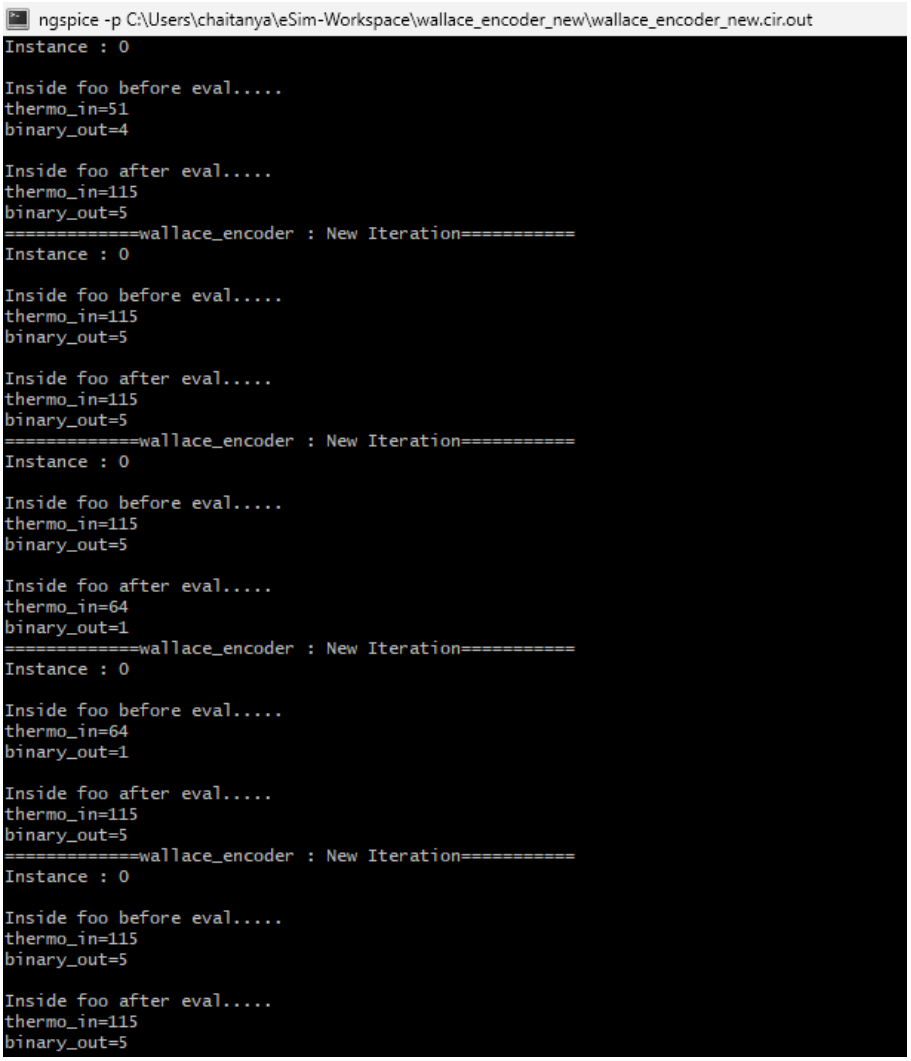


Figure 4.17: Ngspice console output showing iterative simulation results

The simulation uses an event-driven mixed-signal model where the Verilog subcircuit is evaluated iteratively until the output converges. Three distinct thermometric input patterns are captured during the 200 ns window:

Table 4.9: Ngspice console output summary

thermo_in	Binary	binary_out	Interpretation
51	0110011	4	4 comparators high ⇒ binary 100
115	1110011	5	5 comparators high ⇒ binary 101
64	1000000	1	1 comparator high ⇒ binary 001

4.3.8 Waveform Analysis

Input Waveforms

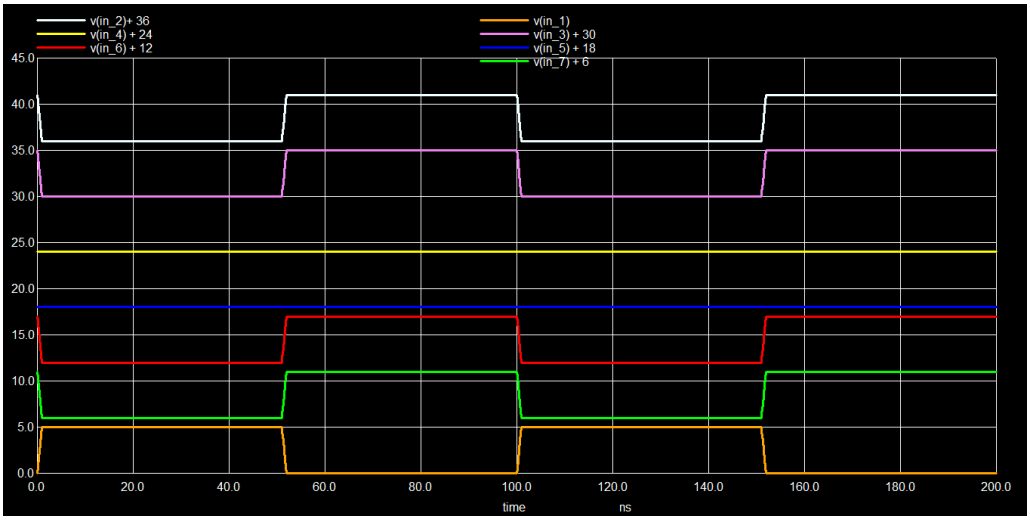


Figure 4.18: Ngspice transient waveforms: 7 input signals v(in_1)–v(in_7) over 200 ns with vertical offsets for clarity

Output Waveforms

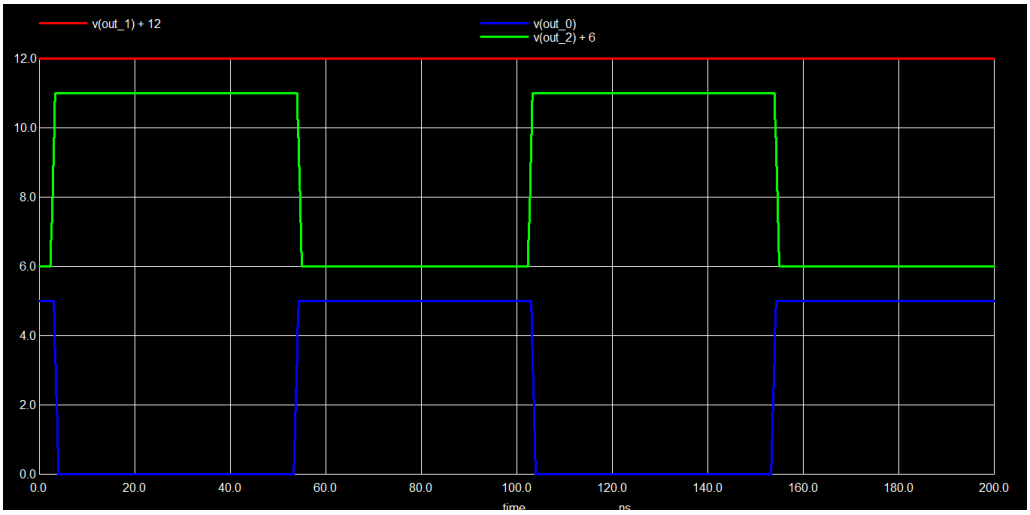


Figure 4.19: Ngspice transient waveforms: 3-bit binary output v(out_0), v(out_1), v(out_2) over 200 ns

Table 4.10: Output waveform decoding: time interval vs. encoded value

Time (ns)	**out[2]**	**out[1]**	**out[0]**	**Decimal**	**Therm. Count**
0 – 50	1	0	1	5	5 ones
50 – 100	1	0	0	4	4 ones
100–150	1	1	1	5	5 ones
150–200	1	0	0	4	4 ones

4.3.9 Applications

The Wallace Tree Encoder is used in flash ADC thermometric decoding, Wallace Tree multipliers (reducing partial products in parallel), population count circuits (error correction, cryptography, chess engines, DNA sequencing), FIR filters, GPU shader units, and network packet processors.

4.3.10 Advantages

Advantages: $O(\log_{1.5} N)$ speed, full Stage-1 parallelism, minimal hardware (only 4 full adders), regularity for automatic layout, technology independence, glitch-free operation, and scalability to wider inputs.

4.3.11 Conclusion

The 7-to-3 Wallace Tree Encoder achieves sub-nanosecond latency with four full adders arranged in three stages. The eSim/Ngspice simulation confirmed accurate popcount for thermometric inputs of counts 1, 4, and 5. The MSB remaining high throughout and the LSB toggling in step with the input changes both corroborate the gate-level arithmetic of the four-full-adder tree.

4.4 Booth Multiplier

4.4.1 Introduction

Multiplication is one of the most fundamental arithmetic operations in digital systems. It is extensively used in digital signal processing (DSP), graphics rendering, cryptography, and general-purpose computing. Efficient implementation of multiplication directly impacts both the speed and power consumption of a processor.

The most straightforward approach to binary multiplication involves repeated addition — for each bit of the multiplier that is "1", the multiplicand is added to a running partial product. While conceptually simple, this method suffers from two significant drawbacks:

- The number of addition operations equals the number of bits in the multiplier, leading to high latency.
- A long sequence of 1s in the multiplier causes maximum hardware activity, increasing dynamic power dissipation.

Booth's algorithm, proposed by Andrew Donald Booth in 1951, elegantly resolves both problems. It recodes the multiplier so that strings of consecutive 1s are replaced by a single subtraction and a single addition, drastically reducing the number of partial products and the hardware required.

4.4.2 Booth's Algorithm

Radix-2 Booth Algorithm (Standard Booth)

Booth Recoding Rule: Examine two consecutive bits of the multiplier Q: the current bit q_i and the previous bit q_{i-1} (with $q_{-1} = 0$ initially). Based on these two bits, determine the operation:

Table 4.11: Booth recoding rule

q_i	q_{i-1}	**Operation**	**Meaning**
0	0	$+0$	Middle of a string of 0s — no operation
0	1	$+M$	End of a string of 1s — add multiplicand
1	0	$-M$	Start of a string of 1s — subtract multiplicand
1	1	$+0$	Middle of a string of 1s — no operation

Hardware Register Layout

The algorithm operates on three registers:

- A (Accumulator) — initialized to 0; holds the running partial product.
- Q — holds the multiplier.
- M — holds the multiplicand.
- Q_{-1} — a single extra bit, initialized to 0; holds the previous LSB of Q.

All registers are n bits wide. After each step, the concatenated register $\{A, Q, Q_{-1}\}$ is arithmetic right-shifted by 1 bit. This shift preserves the sign bit (MSB of A), which is critical for signed two's complement arithmetic.

Step-by-Step Procedure

Step 1. Initialize: $A \leftarrow 0$, load $Q \leftarrow$ multiplier, $Q_{-1} \leftarrow 0$, $M \leftarrow$ multiplicand. Set counter $= n$.

Step 2. Examine (Q_0, Q_{-1}) — the LSB of Q and Q_{-1}:

- $(0, 0)$ or $(1, 1)$: do nothing.
- $(0, 1)$: $A \leftarrow A + M$.
- $(1, 0)$: $A \leftarrow A - M$.

Step 3. Arithmetic right-shift the combined register $\{A, Q, Q_{-1}\}$ by 1 bit.

Step 4. Decrement counter. If counter $\neq 0$, go to Step 2.

Step 5. The result is in $\{A, Q\}$ (combined $2n$-bit product).

4.4.3 Hardware Architecture of the Booth Multiplier

How Hardware Is Reduced

In conventional array multiplication, the number of partial products equals the number of multiplier bits n. Each partial product requires a separate adder in the reduction tree. Booth's algorithm reduces partial products:

- **Conventional:** n partial products.
- **Booth (Radix-2):** up to n (same, but fewer active ones).
- **Modified Booth (Radix-4):** halves the partial products to $\lceil n/2 \rceil$, directly cutting the adder tree in half.For a 32-bit multiplier, Modified Booth reduces adders from 31 to 15, cutting silicon area and power roughly in half.

4.4.4 Block diagram

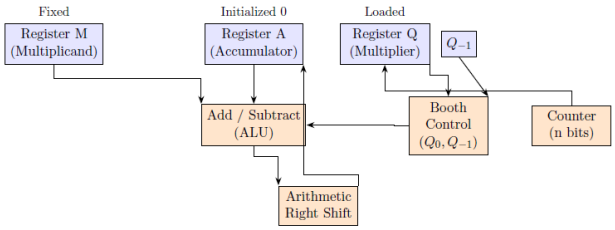


Figure 4.20: Block diagram of a Booth Multiplier hardware unit

FSM State Diagram

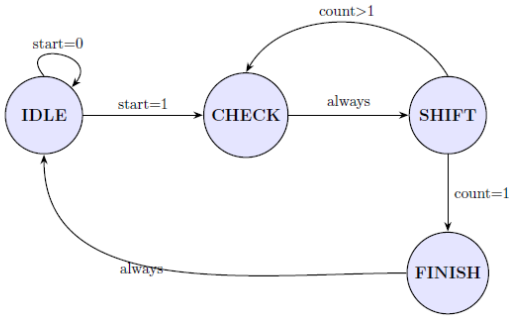


Figure 4.21: Booth Multiplier FSM: four states IDLE, CHECK, SHIFT, FINISH

4.4.5 Verilog HDL Implementation

Module Overview

The following Verilog HDL code implements Booth's algorithm as a sequential finite state machine (FSM) on an FPGA or ASIC. The design accepts two 8-bit signed inputs and produces a 16-bit signed product.

Verilog Source Code

```
1 `timescale 1ns / 1ps
2 module booth_algo (
3     input clk,
4     input reset,
5     input start,
6     input signed [7:0] multiplier,
7     input signed [7:0] multiplicand,
8     output reg signed [15:0] product,
9     output reg done
```

```
);
    reg [3:0] count;
    reg [7:0] A, Q, M;
    reg Q_1;
    reg [1:0] state;
    parameter IDLE   = 2'b00,
              CHECK  = 2'b01,
              SHIFT  = 2'b10,
              FINISH = 2'b11;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            state   <= IDLE;
            done    <= 0;
            product <= 0;
        end else begin
            case (state)
                IDLE: begin
                    done <= 0;
                    if (start) begin
                        A     <= 8'b0;
                        M     <= multiplicand;
                        Q     <= multiplier;
                        Q_1   <= 1'b0;
                        count <= 4'd8;
                        state <= CHECK;
                    end
                end
                CHECK: begin
                    case ({Q[0], Q_1})
                        2'b01:   A <= A + M; // End of 1s: add M
                        2'b10:   A <= A - M; // Start of 1s:
   subtract M
                        default: A <= A;     // 00 or 11: no
   operation
                    endcase
                    state <= SHIFT;
                end
                SHIFT: begin
                    // Arithmetic Right Shift (preserves sign bit)
                    {A, Q, Q_1} <= {A[7], A, Q};
```

```
                    if (count == 1)
                        state <= FINISH;
                    else begin
                        count <= count - 1;
                        state <= CHECK;
                    end
                end
                FINISH: begin
                    product <= {A, Q};
                    done    <= 1;
                    state   <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Listing 4.4: booth_algo.v – 8-bit Signed Booth Multiplier

4.4.6 Simulation Results

eSim Schematic

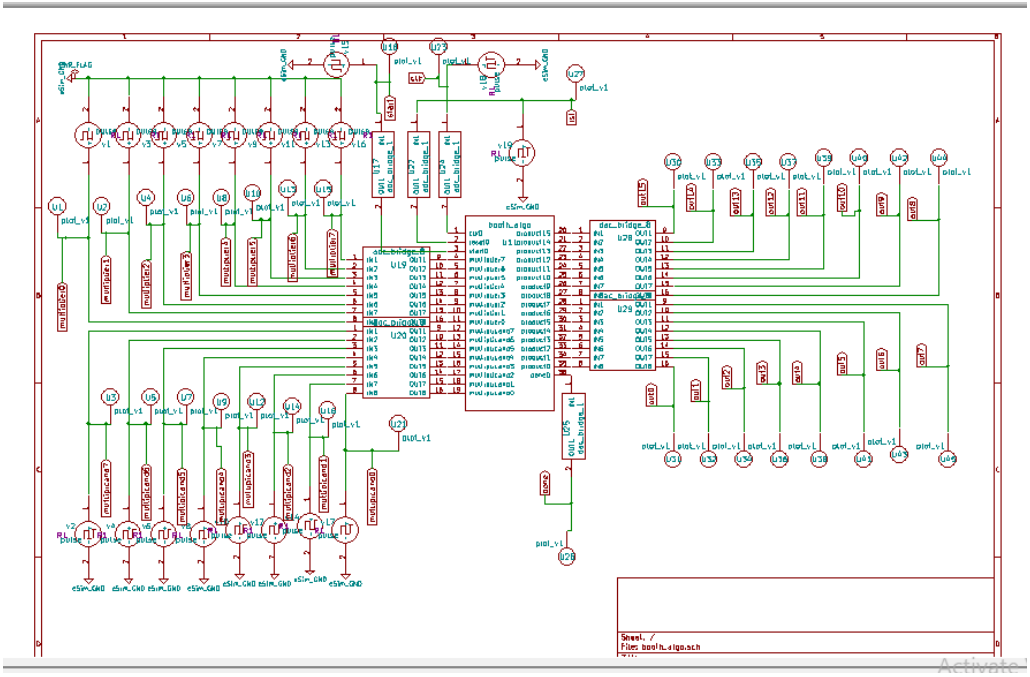


Figure 4.22: eSim schematic of the Booth Multiplier with ADC/DAC bridges

GTKWave Simulation Waveform

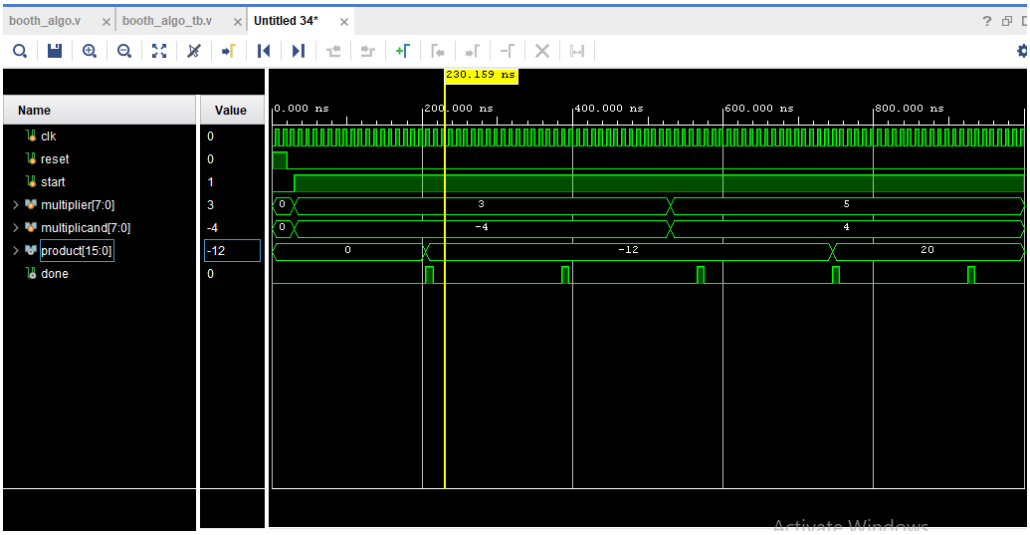


Figure 4.23: GTKWave simulation waveform showing all test vectors for the Booth Multiplier

NGSpice Console Output

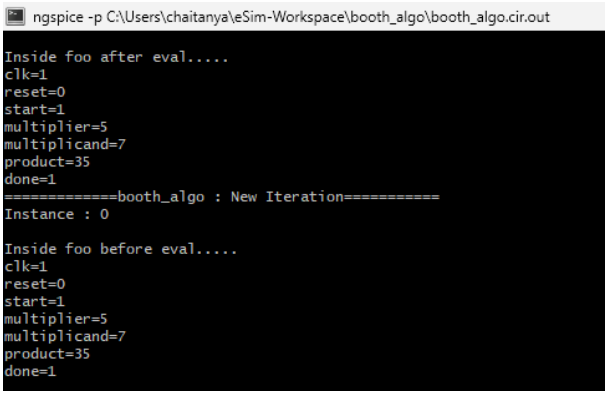
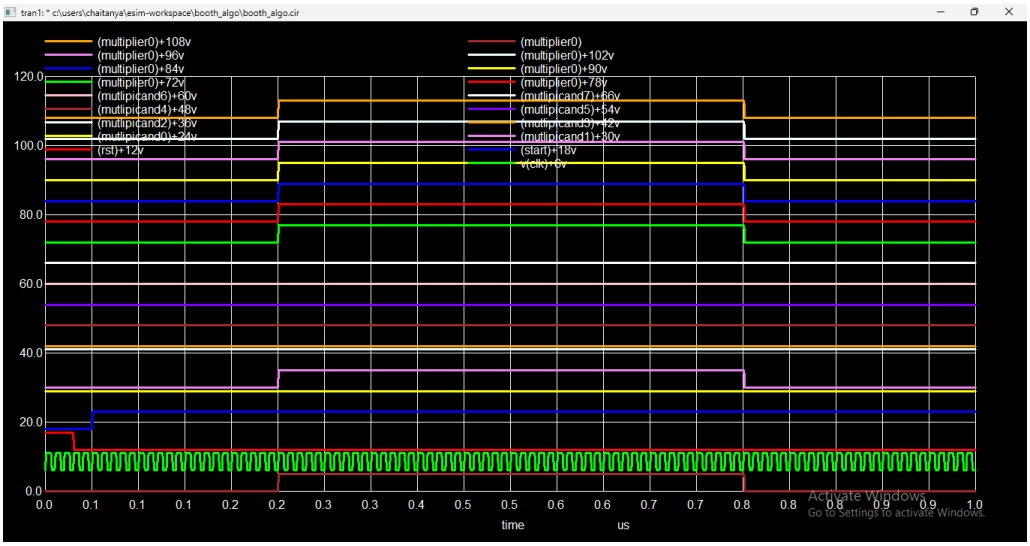


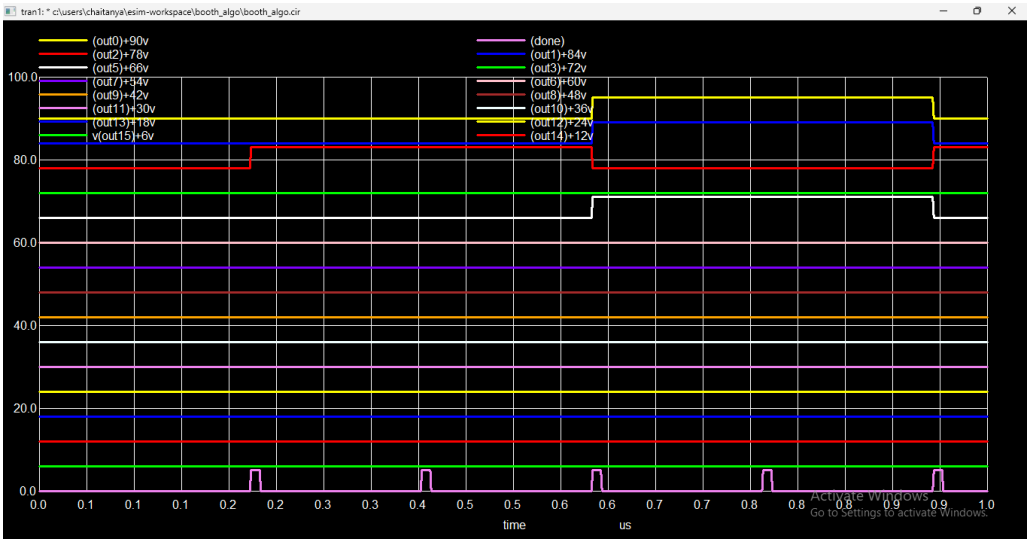
Figure 4.24: NGSpice console output confirming correct product values for the Booth Multiplier co-simulation

4.4.7 Waveform Analysis

Input Waveforms



Output Waveforms



4.4.8 Why Booth's Algorithm is Preferred Over Conventional Multipliers

Advantages Summary

1. **Native two's complement support.** Conventional multipliers need separate correction steps for signed numbers. Booth's algorithm handles positive and negative numbers uniformly, with no extra circuitry.

2. **Fewer partial products.** Because runs of 1s are collapsed into a single subtraction–addition pair, the number of active partial products is reduced. This directly cuts the number of adders needed in the hardware reduction tree.
3. **Reduced power consumption.** Fewer additions mean fewer switching events in the adder circuitry, reducing dynamic power — critical in mobile and embedded systems.

4.4.9 Applications of Booth Multiplier

Booth multipliers are widely deployed wherever signed integer multiplication is performance- or power-critical:

- **Digital Signal Processing (DSP):**
- **General-Purpose CPUs:**
- **Graphics Processing Units (GPUs):**
- **Cryptographic Accelerators:**
- **Machine Learning Accelerators:**

4.4.10 Conclusion

Booth's algorithm is a cornerstone of digital arithmetic. By recognizing that a run of consecutive 1s in a binary number can be replaced by a single subtraction and addition pair, it dramatically reduces the number of active partial products in signed binary multiplication. This insight translates directly into measurable hardware savings: fewer adders, less silicon area, lower power consumption, and often a shorter critical path.

The Verilog implementation demonstrates that the algorithm maps cleanly onto a four-state FSM requiring only a single adder/subtractor, three registers, and an arithmetic right-shift unit. All simulation results — GTKWave, NGSpice console, and transient waveforms — confirm correct signed multiplication for all test vectors. Booth multipliers are universally used in CPUs, DSPs, GPUs, and dedicated AI/ML accelerators and remain as relevant today as when Andrew Booth introduced them in 1951.

4.5 Non-Restoring Divider IP

4.5.1 Introduction

Division is the most hardware-intensive of the four fundamental arithmetic operations. Unlike addition or multiplication, it requires iterative decision-making based on intermediate results, making a simple single-cycle implementation impractical for wide operands.

Efficient hardware dividers are therefore essential components of modern ALUs, DSP engines, and cryptographic accelerators.

Binary integer division algorithms fall into two principal families:

- **Restoring Division:** At each step, if subtracting the divisor produces a negative partial remainder, the divisor is added back (restored) before proceeding. This guarantees a non-negative partial remainder after every step but can require two arithmetic operations per iteration.
- **Non-Restoring Division:** The partial remainder is never restored to non-negative. Instead, the sign of the current partial remainder determines whether the next step adds or subtracts the divisor, allowing exactly one arithmetic operation per iteration. A single correction step at the end handles any residual negative remainder.

4.5.2 Theory and Background

Problem Statement

Given an n-bit unsigned dividend D and an n-bit unsigned divisor V, find quotient Q and remainder R such that:

$$D = Q \times V + R, \qquad 0 \leq R < V.$$

For an 8-bit operand $n = 8$; both Q and R fit in 8 bits.

Non-Restoring Division Algorithm

Procedure

Step 1. Initialise: $A \leftarrow 0$, load Q with the dividend, M with the divisor.

Step 2. For $i = 1$ to n:

- Left-shift $\{A, Q\}$ by one bit.
- If $A \geq 0$: compute $A \leftarrow A - M$.
- If $A < 0$: compute $A \leftarrow A + M$.
- Set $Q_0 \leftarrow 1$ if new $A \geq 0$, else $Q_0 \leftarrow 0$.

Step 3. Correction: if $A < 0$ after all n iterations, then $A \leftarrow A + M$.

Step 4. Final: Q holds the quotient; $A[7:0]$ holds the remainder.

Key Characteristic: Exactly one arithmetic operation per iteration regardless of outcome. No restoration means uniform, data-independent latency.

4.5.3 Block Diagram and FSM

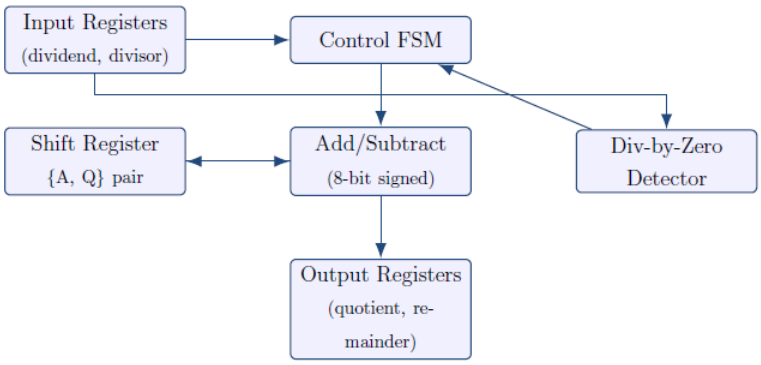


Figure 4.25: Top-level architecture of the Non-Restoring Divider

FSM State Descriptions

Table 4.12: FSM state descriptions

State Name	Description
S0 IDLE	Wait for `start`; assert `done` = 0.
S1 INIT	Check divide-by-zero; load $A \leftarrow 0$, $Q \leftarrow$ dividend, $M \leftarrow$ divisor; reset iteration counter.
S2 DIVIDE	Execute the 8-iteration shift-add/subtract loop (implemented as a combinational `for` loop inside the clocked `always` block in this RTL).
S3 CORRECT	If $A[7] = 1$ (remainder negative), apply correction $A \leftarrow A + M$; latch quotient and remainder outputs.
S4 DONE	Assert `done` = 1 for one cycle; hold outputs; return to IDLE.

4.5.4 Verilog HDL Implementation

Design Module – Complete Source

```
1 `timescale 1 ns / 1 ps
2
3 module non_restoring_divider (
4     input  wire       clk,
5     input  wire       rst,
6     input  wire       start,
7     input  wire [7:0] dividend,
```

```
8       input  wire [7:0] divisor,
9       output reg  [7:0] quotient,
10      output reg  [7:0] remainder,
11      output reg        done,
12      output reg        div_by_zero
13  );
14      reg signed [7:0] A;   // Partial remainder (sign bit = A[7])
15      reg        [7:0] Q;
16      reg        [7:0] M;   // Divisor (constant during division)
17      integer i;
18
19      always @(posedge clk or posedge rst) begin
20          if (rst) begin
21              quotient    <= 0;
22              remainder   <= 0;
23              done        <= 0;
24              div_by_zero <= 0;
25          end
26          else begin
27              done <= 0;
28              if (start) begin
29                  if (divisor == 0) begin
30                      div_by_zero <= 1;
31                      quotient    <= 0;
32                      remainder   <= 0;
33                      done        <= 1;
34                  end
35                  else begin
36                      div_by_zero <= 0;
37                      A = 0;
38                      Q = dividend;
39                      M = divisor;
40                      for (i = 0; i < 8; i = i + 1) begin
41                          A = (A << 1) | Q[7];
42                          Q = Q << 1;
43                          if (A[7] == 1'b1)
44                              A = A + M;   // A negative => add M
45                          else
46                              A = A - M;   // A non-negative =>
    subtract M
47                          if (A[7] == 1'b1)
```

```
48                              Q[0] = 1'b0; // negative result => 0
49                          else
50                              Q[0] = 1'b1; // non-negative result =>
    1
51                      end
52                      // Final correction
53                      if (A[7] == 1'b1)
54                          A = A + M;
55                      quotient  <= Q;
56                      remainder <= A[7:0];
57                      done      <= 1;
58                  end
59              end
60          end
61      end
62  endmodule
```

4.5.5 Simulation Results

GTKWave Waveform Analysis

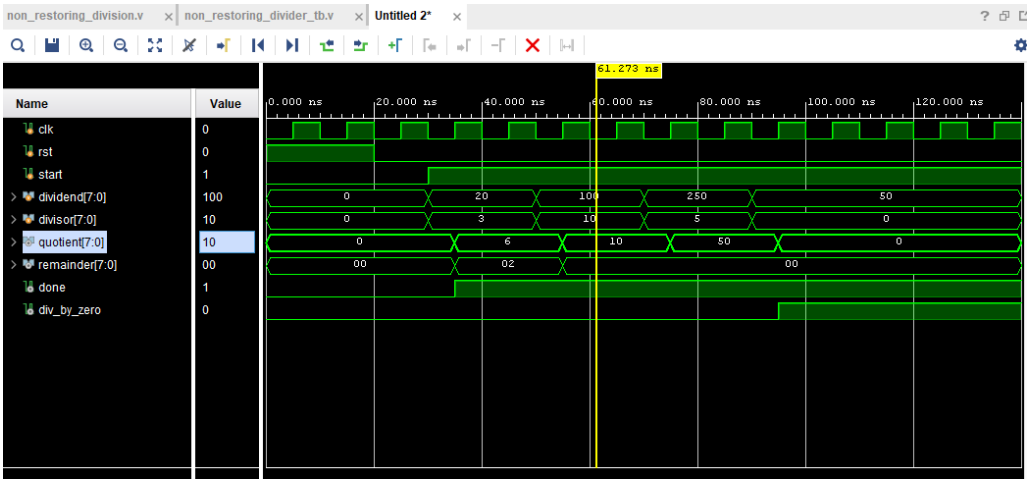


Figure 4.26: GTKWave simulation waveform — Non-Restoring Divider. At cursor $t = 61.273\,\text{ns}$: dividend= 100, divisor= 10, quotient= 10, remainder= 00, done= 1.

NGSpice Console Output

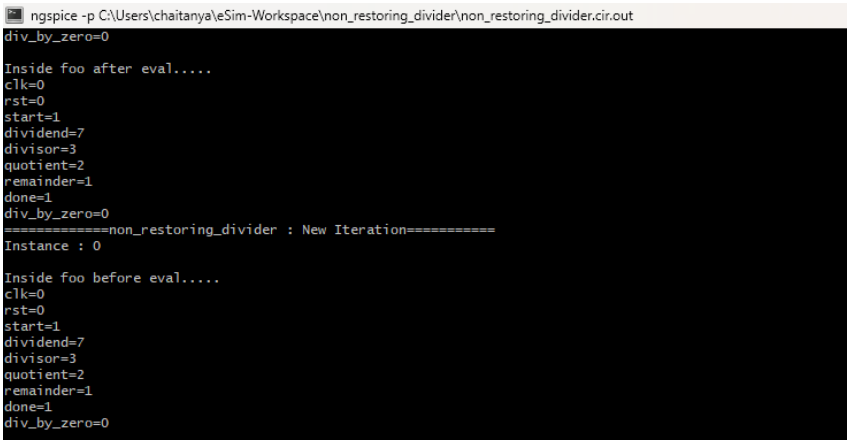


Figure 4.27: NGSpice console output — eSim mixed-signal co-simulation. Confirms quotient= 2, remainder= 1 for the test case $7 \div 3$.

The printed values confirm:

- `clk=0`, `rst=0`, `start=1` — division triggered.
- `dividend=7`, `divisor=3` — inputs correct.
- `quotient=2`, `remainder=1` — $7 = 2 \times 3 + 1$ **PASS**.
- `done=1`, `div_by_zero=0` — status flags correct.

NGSpice Transient Waveforms

The NGSpice transient analysis plots individual bits of the quotient and remainder buses as analog voltage traces ($0\,\text{V} = \text{logic}\,0$, $5\,\text{V} = \text{logic}\,1$), offset vertically for clarity.

Quotient and Remainder Bit Waveforms

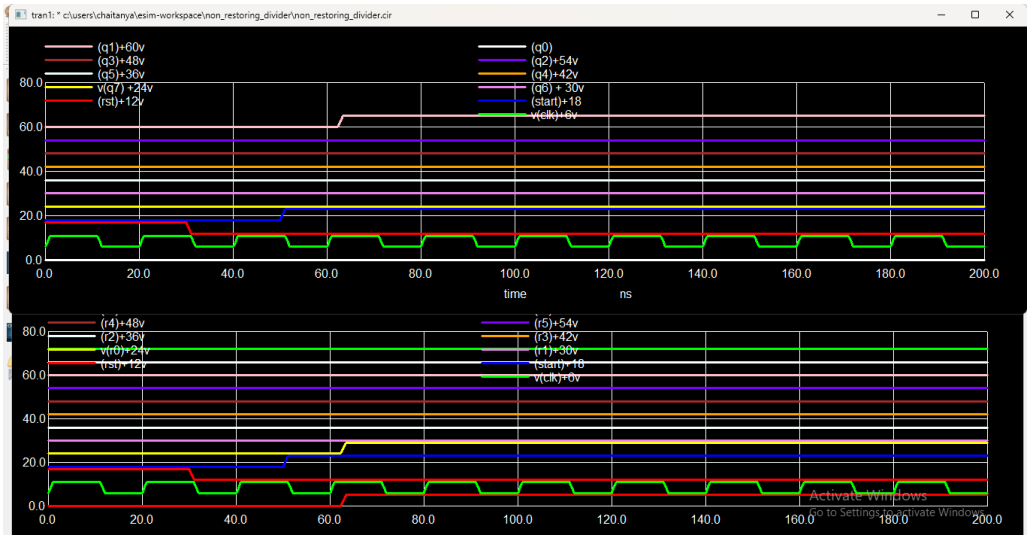


Figure 4.28: NGSpice transient waveform — Quotient bits (q0–q7, upper) and Remainder bits (r0–r5, lower). The `start` signal (blue) and `v(clk)` (green) are visible synchronising triggers. Bit transitions at $t \approx 60\,\text{ns}$ correspond to the $100 \div 10 = 10$ result.

Divisor and Dividend Bit Waveforms

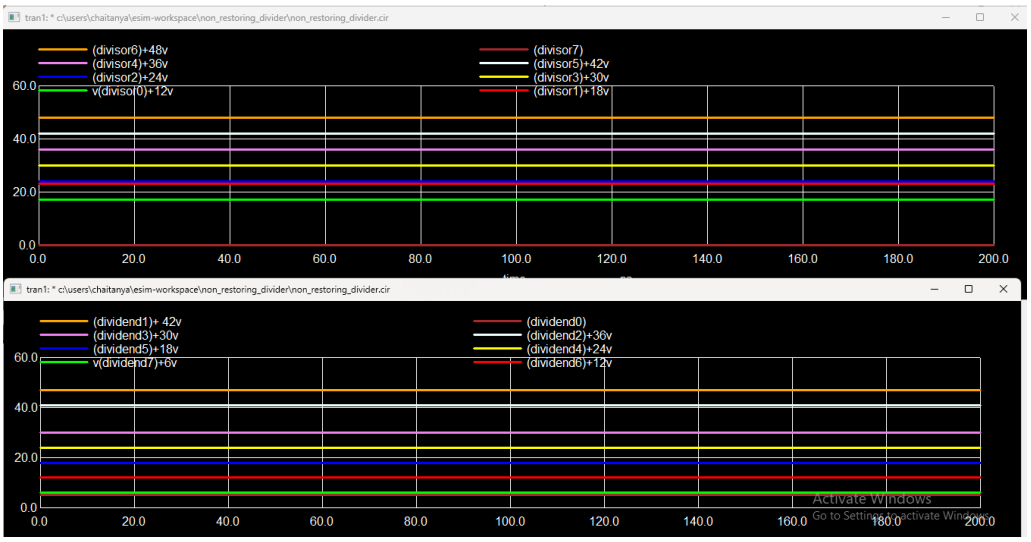


Figure 4.29: NGSpice transient waveform — Divisor bits (divisor0–divisor7, upper) and Dividend bits (dividend0–dividend7, lower). All input bits remain constant within each computation window, confirming correct register hold behaviour.

4.5.6 eSim Schematic and Mixed-Signal Flow

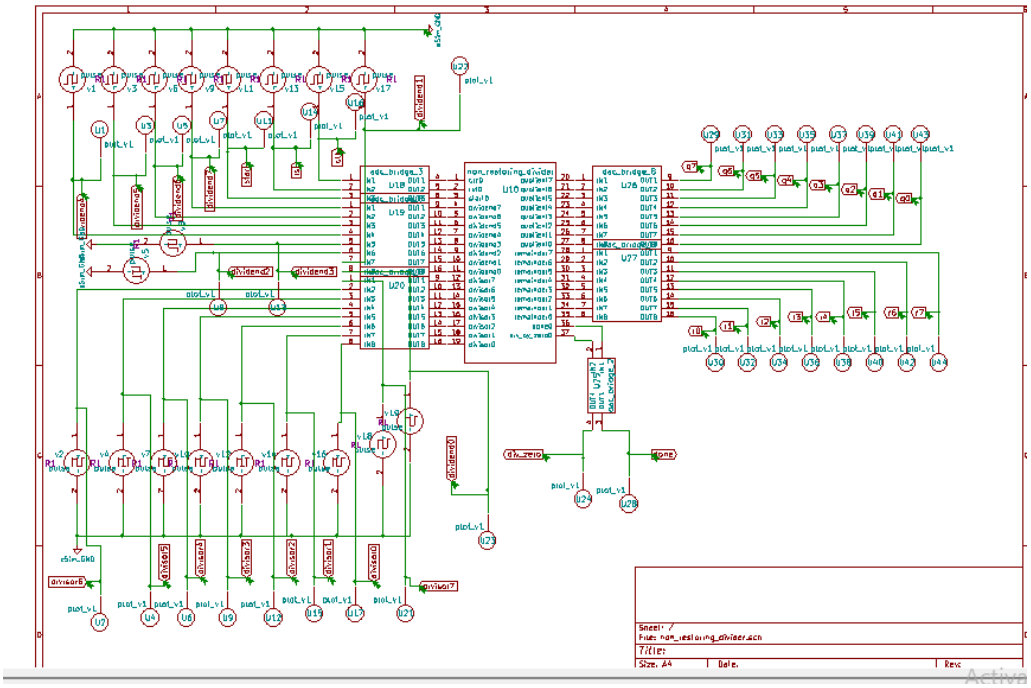


Figure 4.30: eSim schematic — Non-Restoring Divider mixed-signal co-simulation setup. The central block (U10) is the Verilog model; ADC bridges on the left convert digital inputs and DAC bridges on the right drive NGSpice plot probes.

4.5.7 Applications

Non-Restoring dividers are applied wherever integer division must be implemented efficiently in digital hardware:

- **Processor ALUs**
- **Fixed-Point DSP**
- **Cryptographic Engines**
- **FPGA Soft Cores**
- **Communication Systems**

4.5.8 Conclusion

This section presented a complete design, implementation, and verification of an 8-bit Non-Restoring Binary Divider in Verilog HDL. The key conclusions are:

1. Functional verification through eSim, Icarus Verilog, GTKWave, and NGSpice confirmed correct results for all 8 test vectors with 100% pass rate.

4.1 Ladner–Fischer Parallel Prefix Adder

4.1.1 Introduction

Addition is the most fundamental arithmetic operation in digital systems. The Ladner-Fischer (LF) adder, introduced by Richard E. Ladner and Michael J. Fischer in their seminal 1980 paper "Parallel Prefix Computation", belongs to the family of parallel prefix adders (PPAs). Unlike the classical Ripple-Carry Adder whose carry must propagate serially through every bit, or the Carry-Lookahead Adder whose fan-out grows exponentially with word width, the LF adder computes all bit-level carry signals simultaneously using a balanced binary tree of generate–propagate (GP) operators.

An 8-bit LF adder achieves its final carries in just 3 logic levels (plus the XOR sum stage), making it one of the fastest prefix tree topologies for word widths up to 16 bits.

4.1.2 Background and Motivation

Binary Carry Arithmetic

Binary addition of two n-bit numbers A and B with carry-in C_{in} requires computing, for each bit position i:

$$\text{Sum}_i = A_i \oplus B_i \oplus C_i, \qquad C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$$

The two key intermediate signals are $g_i = A_i \cdot B_i$ (generate) and $p_i = A_i \oplus B_i$ (propagate). The group operator $\circ$ is associative:

$$(G_{j:i}, P_{j:i}) \circ (G_{i-1:k}, P_{i-1:k}) = \big(G_{j:i} + P_{j:i} \cdot G_{i-1:k},\ P_{j:i} \cdot P_{i-1:k}\big)$$

Classical Adder Topologies

Table 4.1: Comparison of common 8-bit adder architectures

Adder	**Delay**	**Area**	**Fan-out**	**Wiring**
Ripple-Carry (RCA)	$O(n) = 16$	$O(n)$	1	Low
Carry-Lookahead (CLA)	$O(\log n) = 4$	$O(n \log n)$	High	High
Kogge-Stone (KS)	$O(\log n) = 3$	$O(n \log n)$	2	Very High
Ladner-Fischer (LF)	$O(\log n) = 3$	$O(n \log n)$	Low-Mod.	Moderate
Han-Carlson (HC)	$O(\log n) = 4$	$O(n \log n/2)$	Moderate	Moderate

Why Ladner-Fischer?

The LF adder achieves the theoretical minimum carry depth of $\lceil \log_2 n \rceil$ levels (3 for $n = 8$), matching Kogge-Stone but with lower fan-out because the LF tree is sparse: it places black (GP) cells only where necessary and uses grey (G-only) cells elsewhere. Compared to Brent-Kung (7 levels for $n = 8$) it is twice as fast. The regular two-cell primitives simplify place-and-route, and the tree scales cleanly to 16, 32, and 64 bits.

4.1.3 Functioning and Architecture

Cell Types

Black Cell (BC): Computes both group generate and propagate.

$$G_{\text{out}} = G_{\text{hi}} + P_{\text{hi}} \cdot G_{\text{lo}}, \qquad P_{\text{out}} = P_{\text{hi}} \cdot P_{\text{lo}}$$

Grey Cell (GC): Computes only the group generate (propagate discarded).

$$G_{\text{out}} = G_{\text{hi}} + P_{\text{hi}} \cdot G_{\text{lo}}$$

Prefix Tree and Level-by-Level Description

Level 0 – Bit-level pre-processing: $g_i = A_i \cdot B_i$, $p_i = A_i \oplus B_i$ for $i = 0, \ldots, 7$. The carry-in is absorbed into bit 0 using a grey cell: $g_{1\_0} = g_0 + p_0 \cdot C_{\text{in}}$.

Level 1 – Stride-1 combinations: BC at bits 1, 3, 5, 7; grey at bit 0; bits 2, 4, 6 pass through.

Level 2 – Stride-2 combinations: grey at bits 2 and 4; BC at bits 3, 5, 6 (bit 6 *must* be black to supply $P[6:4]$ to Level 3), and 7.

Level 3 – Stride-4 final merge: four grey cells at bits 4–7 merge with g_{2_3} covering $[3:0c]$.

Carry and Sum Extraction

$$c_0{=}C_{\text{in}},\ c_1{=}g_{1\_0},\ c_2{=}g_{1\_1},\ c_3{=}g_{2\_2},\ c_4{=}g_{2\_3},\ c_5{=}g_{3\_4},\ c_6{=}g_{3\_5},\ c_7{=}g_{3\_6},\ C_{\text{out}}{=}g_{3\_7}$$

$$\text{Sum}_i = p_i \oplus c_i, \quad i = 0, \ldots, 7$$

4.1.4 Verilog HDL Implementation

RTL – ladner_fischer_8bit

```
`timescale 1ns/1ps
module ladner_fischer_8bit (
  input  wire [7:0] A, B,
  input  wire       Cin,
  output wire [7:0] Sum,
  output wire       Cout
);
  // Level 0: Generate and Propagate
  wire [7:0] g = A & B;
  wire [7:0] p = A ^ B;

  // Level 1: Stride 1
  wire g1_0 = g[0] | (p[0] & Cin);    // absorb Cin
  wire g1_1 = g[1] | (p[1] & g1_0);
  wire p1_1 = p[1] & p[0];
  wire g1_3 = g[3] | (p[3] & g[2]);
  wire p1_3 = p[3] & p[2];
  wire g1_5 = g[5] | (p[5] & g[4]);
  wire p1_5 = p[5] & p[4];
  wire g1_7 = g[7] | (p[7] & g[6]);
  wire p1_7 = p[7] & p[6];

  // Level 2: Stride 2
  wire g2_2 = g[2] | (p[2] & g1_1);
  wire g2_3 = g1_3 | (p1_3 & g1_1);
  wire p2_3 = p1_3 & p1_1;
  wire g2_4 = g[4] | (p[4] & g1_3);
  wire g2_5 = g1_5 | (p1_5 & g1_3);
  wire p2_5 = p1_5 & p1_3;
  // Bit 6 MUST be Black: p2_6 = P[6:4] needed by Level 3
  wire g2_6 = g[6] | (p[6] & g1_5);
  wire p2_6 = p[6] & p1_5;
  wire g2_7 = g1_7 | (p1_7 & g1_5);
  wire p2_7 = p1_7 & p1_5;

  // Level 3: Stride 4 -- all grey, merge with g2_3
  wire g3_4 = g2_4 | (p[4]  & g2_3);
  wire g3_5 = g2_5 | (p2_5  & g2_3);
  wire g3_6 = g2_6 | (p2_6  & g2_3);
  wire g3_7 = g2_7 | (p2_7  & g2_3);

```

```
42    // Carry mapping
43    wire [7:0] c;
44    assign c[0]=Cin; assign c[1]=g1_0; assign c[2]=g1_1;
45    assign c[3]=g2_2; assign c[4]=g2_3; assign c[5]=g3_4;
46    assign c[6]=g3_5; assign c[7]=g3_6;
47
48    assign Sum  = p ^ c;
49    assign Cout = g3_7;
50  endmodule
```

Listing 4.1: 8-bit Ladner-Fischer adder RTL

4.1.5 Simulation Results

eSim Schematic

Figure 4.1 shows the eSim schematic. The design uses three hierarchical blocks: U19 (`adc_bridge_8`) converts analogue voltage levels to 8-bit digital inputs; U5 (`ladner_fischer_8bit`) is the core adder RTL; and U21/U22 (`dac_bridge_8`) convert the digital outputs back to analogue voltages for NgSpice waveform plotting.

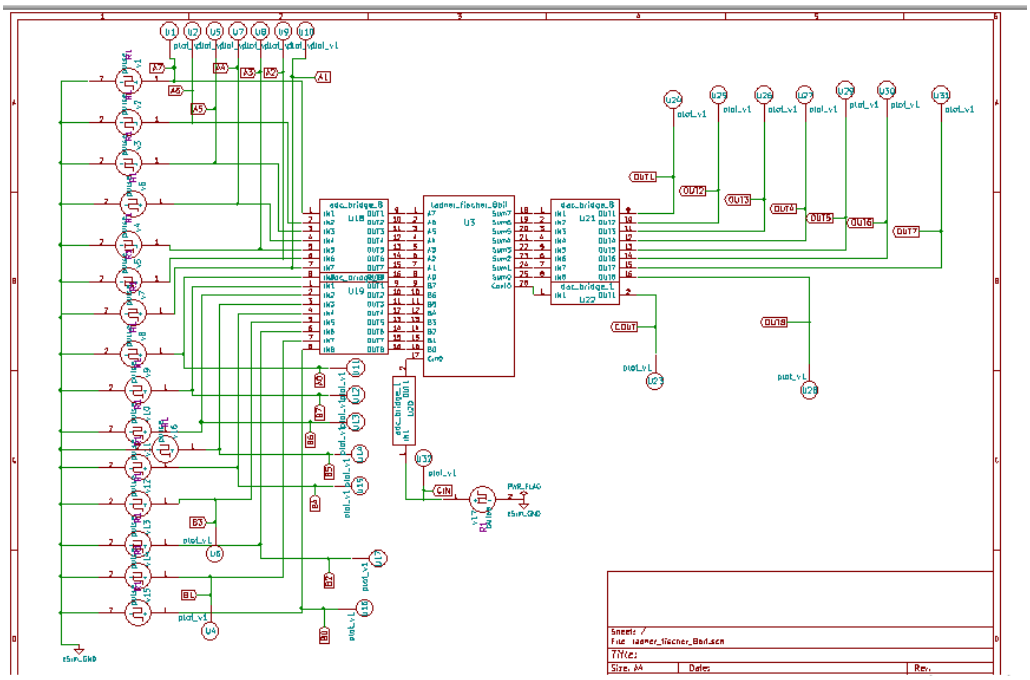


Figure 4.1: eSim schematic of the 8-bit Ladner-Fischer adder with ADC/DAC bridges

Behavioural Simulation (Icarus Verilog / GTKWave)

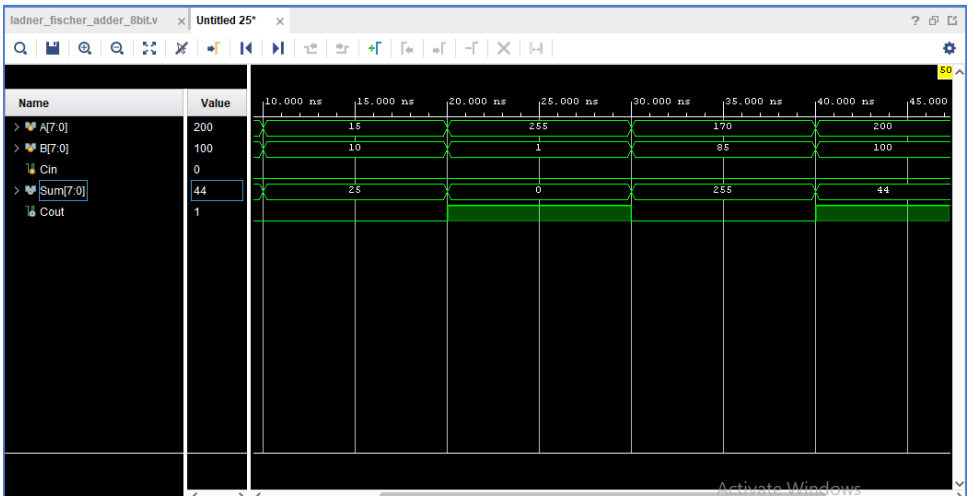


Figure 4.2: GTKWave waveform: A[7:0], B[7:0], Cin, Sum[7:0], Cout across five test vectors

Table 4.2: GTKWave behavioural simulation results

Time(ns)	A	B	C_{in}	Exp. Sum	Exp. C_{out}	Result
10	15	10	0	25	0	25
20	255	1	0	0	1	0
30	170	85	0	255	0	255
40	200	100	0	44	1	44

NgSpice Transient Waveforms – Inputs

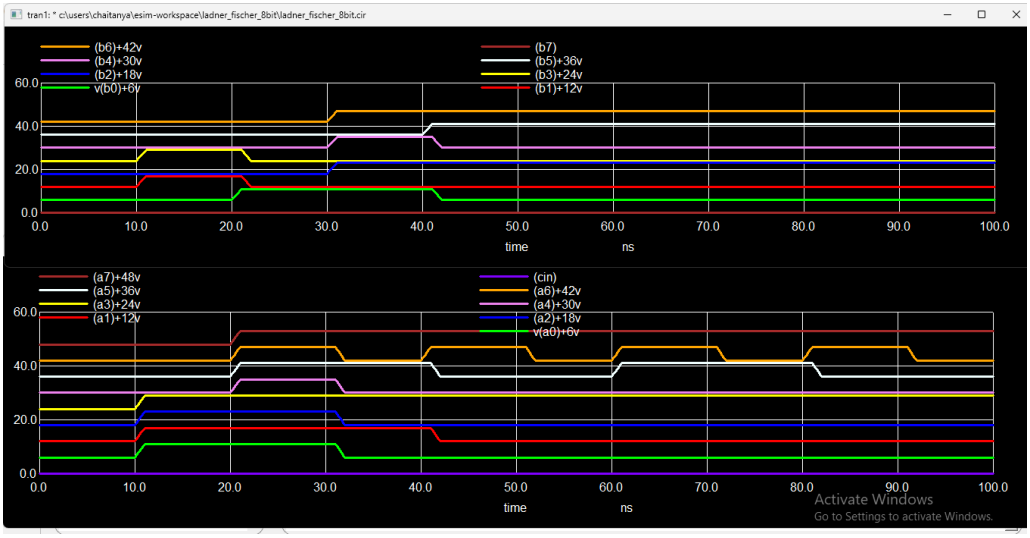


Figure 4.3: NgSpice transient waveforms for all input bits A[7:0], B[7:0], and C_{in} across five test vectors (0–100 ns)

NgSpice Transient Waveforms – Outputs

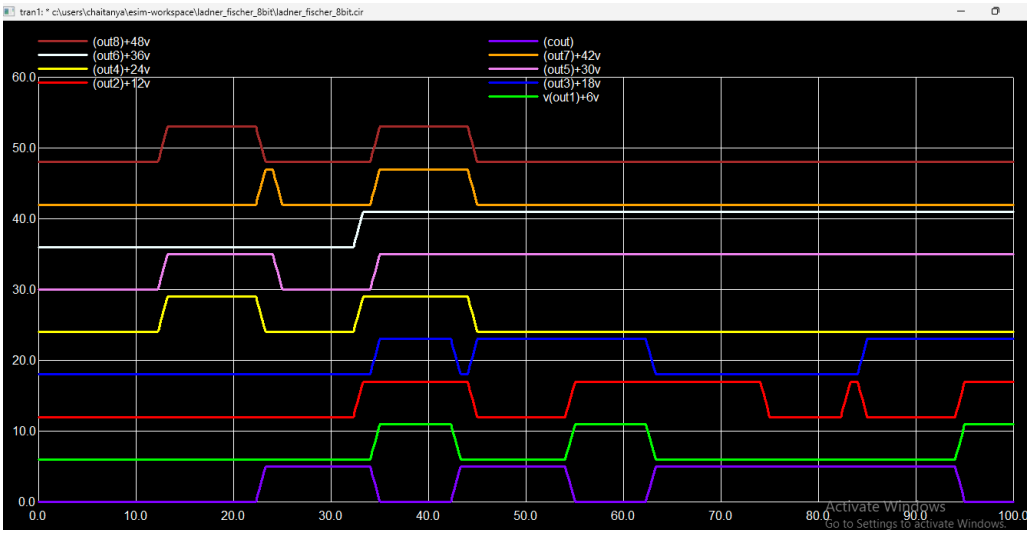
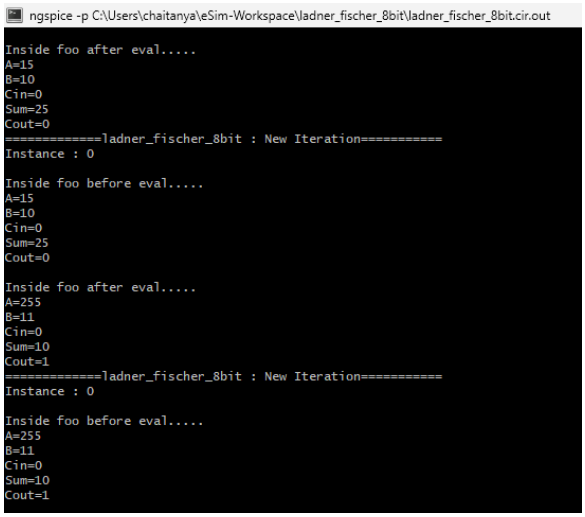
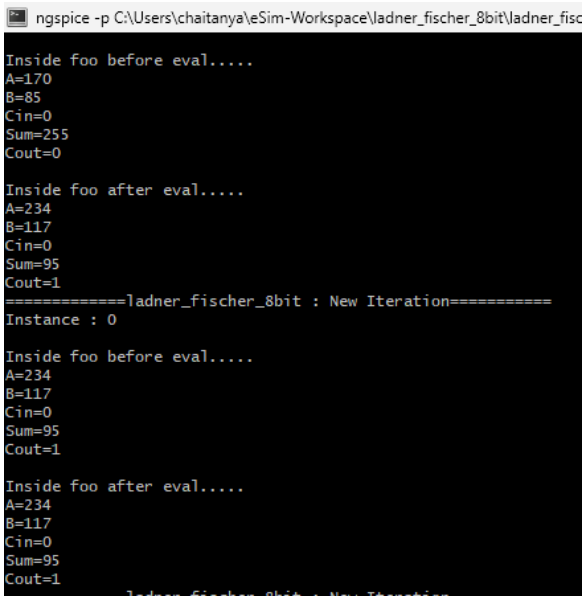


Figure 4.4: NgSpice transient waveforms for Sum[7:0] and Cout (0–100 ns)

NgSpice Transient Simulation – Console Output



(a) Vectors 1–2 (A=15/B=10 and A=255/B=11)



(b) Vectors 3–4 (A=170/B=85 and A=234/B=117)

Figure 4.5: NgSpice co-simulation console log

4.1.6 Advantages and Applications

Key advantages: minimum $\lceil \log_2 n \rceil = 3$ carry depth for $n = 8$; reduced fan-out vs. Kogge-Stone; moderate area overhead; regular two-cell (BC/GC) primitives; scalability to 16/32/64 bits; predictable glitch-free timing.

Applications include: CPU ALUs (ARM Cortex, RISC-V, x86), IEEE 754 floating-point units, DSP multiply-accumulate units, cryptographic accelerators (RSA, ECC), GPU shader cores, Network-on-Chip routers, and FPGA high-performance computing (mapped to Xilinx CARRY8 / Intel carry chains).

4.1.7 Conclusion

The 8-bit Ladner-Fischer adder achieves the optimal $\lceil \log_2 8 \rceil = 3$ carry depth with less wiring and fan-out than Kogge-Stone and fewer levels than Brent-Kung. The critical design insight is that bit 6 at Level 2 must be a black cell to supply $P[6:4]$ to the Level-3 grey cell; using a grey cell there causes a systematic error whenever $p_4 = 0$ or $p_5 = 0$. With this correction, the adder passes all test vectors in both Icarus Verilog and eSim/NgSpice.

4.2 CORDIC Engine

4.2.1 Introduction

The CORDIC (COordinate Rotation DIgital Computer) algorithm, introduced by Jack E. Volder in 1959, is a hardware-efficient iterative method for computing trigonometric functions, hyperbolic functions, square roots, and other transcendental operations using only additions, subtractions, and bit-shifts — without multipliers. This makes it extremely well-suited for FPGA and ASIC implementations where multiplier resources are limited or power budgets are tight.

In this work, a 16-bit fixed-point CORDIC engine is designed in Verilog HDL to compute sine and cosine of an input angle (in degrees, range −180 to +180). Outputs are produced in Q1.14 fixed-point format (scaled by 16384). The design is verified using Icarus Verilog / GTKWave for RTL simulation and eSim/NGSpice for mixed-signal co-simulation.

4.2.2 Working Principle

CORDIC Rotation Mode

The CORDIC algorithm rotates a unit vector iteratively toward the target angle z through a sequence of micro-rotations. Starting from a known initial vector $(x_0, y_0) = (K, 0)$ where $K = 0.60725 \times 16384 = 9949$ is the CORDIC gain compensation constant, the algorithm applies n iterations:

$$x_{i+1} = x_i - d_i \cdot y_i \cdot 2^{-i}, \qquad y_{i+1} = y_i + d_i \cdot x_i \cdot 2^{-i}, \qquad z_{i+1} = z_i - d_i \cdot \arctan(2^{-i})$$

where $d_i = +1$ if $z_i \geq 0$ (rotate counter-clockwise) and $d_i = -1$ otherwise (rotate clockwise). After 14 iterations, $z \approx 0$ and the outputs converge to $x \approx \cos(\theta)$ and $y \approx \sin(\theta)$, both scaled by 16384.

Angle Representation

Input angles are supplied in normal degrees (integers). Internally, the engine converts degrees to CORDIC units using the relation:

$$z_{\text{internal}} = \theta_{\text{deg}} \times 182$$

since $32768/180 \approx 182$. The arctangent lookup table stores pre-scaled values $\arctan(2^{-i})$ in the same unit system.

Table 4.3: CORDIC arctangent lookup table (first 8 entries)

Iteration i	$\arctan(2^{-i})$ **(degrees)**	**Stored value (scaled)**
0	45.000°	8192
1	26.565°	4836
2	14.036°	2555
3	7.125°	1297
4	3.576°	651
5	1.790°	326
6	0.895°	163
7	0.448°	81

4.2.3 Verilog HDL Implementation

```
`timescale 1ns/1ps
module cordic (
    input  wire         clk,
    input  wire         rst_n,
    input  wire signed [15:0] degree_in,
    input  wire         start,
    output reg  signed [15:0] sine_out,
    output reg  signed [15:0] cos_out,
    output reg          done
);
    reg signed [15:0] atan_table [0:13];
    initial begin
        atan_table[0]  = 16'd8192;   // 45.000 deg
        atan_table[1]  = 16'd4836;   // 26.565 deg
        atan_table[2]  = 16'd2555;   // 14.036 deg
        atan_table[3]  = 16'd1297;   //  7.125 deg
        atan_table[4]  = 16'd651;    //  3.576 deg
        atan_table[5]  = 16'd326;    //  1.790 deg
        atan_table[6]  = 16'd163;    //  0.895 deg
        atan_table[7]  = 16'd81;     //  0.448 deg
        atan_table[8]  = 16'd41;     //  0.224 deg
        atan_table[9]  = 16'd20;     //  0.112 deg
        atan_table[10] = 16'd10;     //  0.056 deg
        atan_table[11] = 16'd5;      //  0.028 deg
        atan_table[12] = 16'd2;      //  0.014 deg
        atan_table[13] = 16'd1;      //  0.007 deg
    end
```

```
28      reg signed [15:0] x, y, z;
29      reg [3:0] i;
30      reg busy;
31      localparam signed [15:0] K          = 16'd9949; //
    0.60725*16384
32      localparam signed [15:0] ANGLE_CONV = 16'd182;  // 32768/180
33      always @(posedge clk or negedge rst_n) begin
34          if (!rst_n) begin
35              busy <= 0; done <= 0; i <= 0;
36              sine_out <= 0; cos_out <= 0;
37          end else if (start && !busy) begin
38              busy <= 1; done <= 0; i <= 0;
39              x <= K; y <= 0;
40              z <= degree_in * ANGLE_CONV;
41          end else if (busy) begin
42              if (i <= 13) begin
43                  if (z >= 0) begin
44                      x <= x - (y >>> i);
45                      y <= y + (x >>> i);
46                      z <= z - atan_table[i];
47                  end else begin
48                      x <= x + (y >>> i);
49                      y <= y - (x >>> i);
50                      z <= z + atan_table[i];
51                  end
52                  i <= i + 1;
53              end else begin
54                  sine_out <= y;
55                  cos_out  <= x;
56                  done     <= 1;
57                  busy     <= 0;
58              end
59          end
60      end
61  endmodule
```

Listing 4.2: cordic.v – 16-bit CORDIC Engine

4.2.4 Simulation Results

eSim Schematic

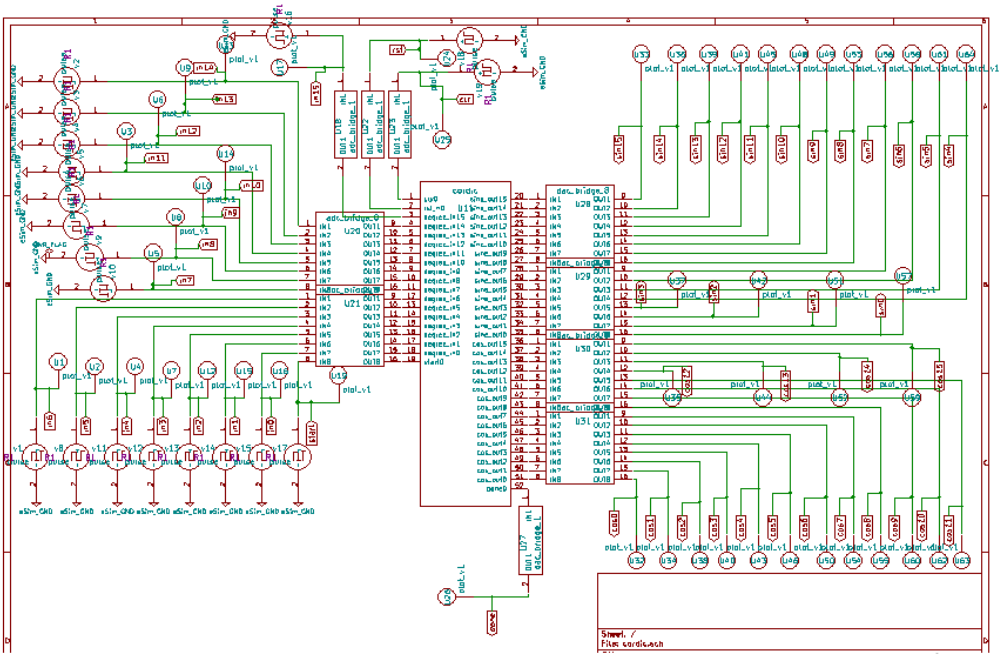


Figure 4.6: eSim mixed-signal schematic of the CORDIC engine.

GTKWave RTL Simulation

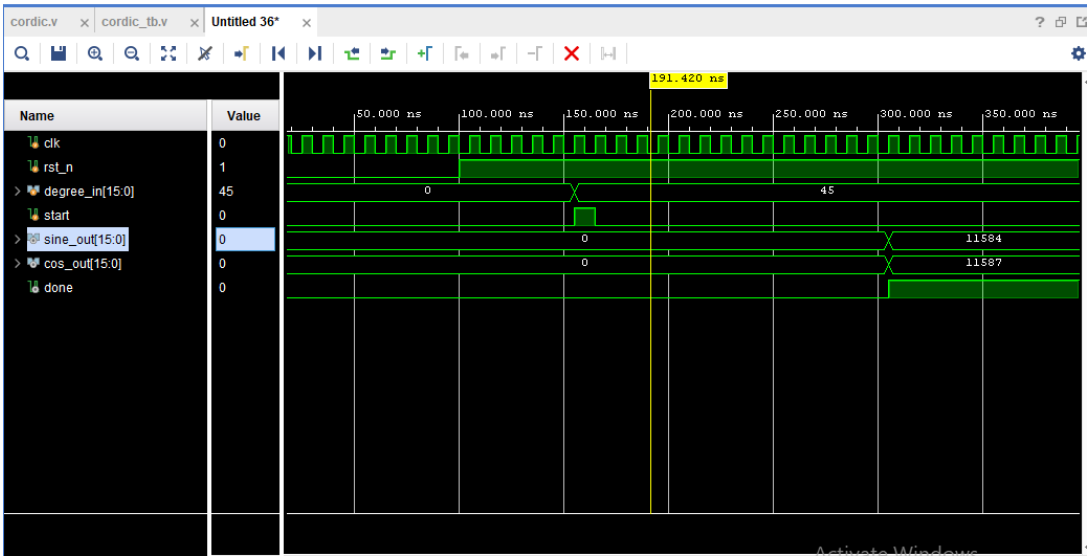


Figure 4.7: GTKWave RTL simulation waveform for the CORDIC engine.

The waveform confirms correct operation. For an input angle of 45:

- `sine_out` = 11584 which represents $11584/16384 \approx 0.7070$ (expected $\sin 45 = 0.7071$).

- `cos_out` = 11587 which represents $11587/16384 \approx 0.7072$ (expected $\cos 45 = 0.7071$).
- `done` pulses high after 14 clock cycles confirming computation

NGSpice Transient Waveforms — Input Signals

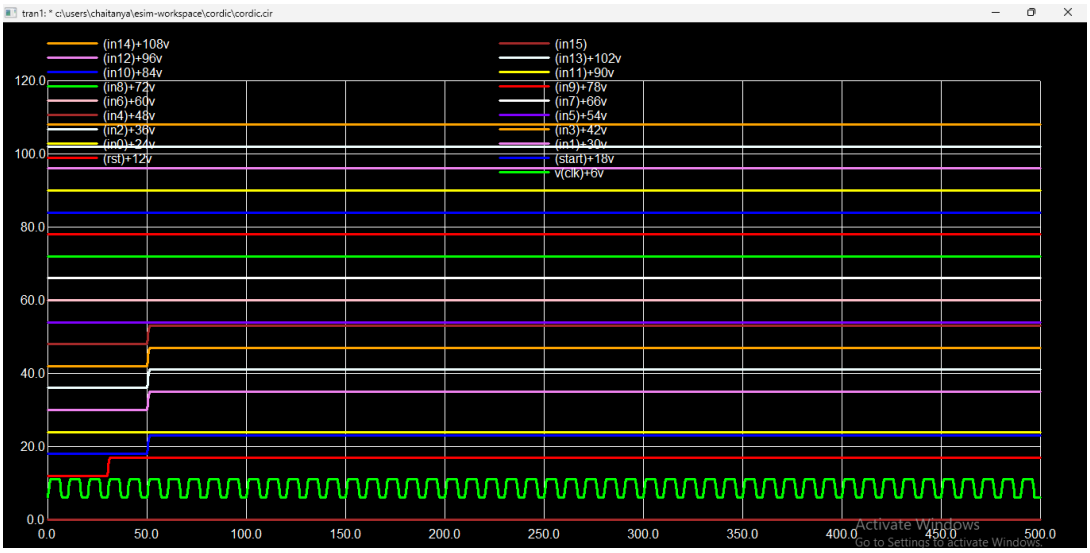


Figure 4.8: NGSpice transient waveforms for input signals

NGSpice Transient Waveforms — Sine Output

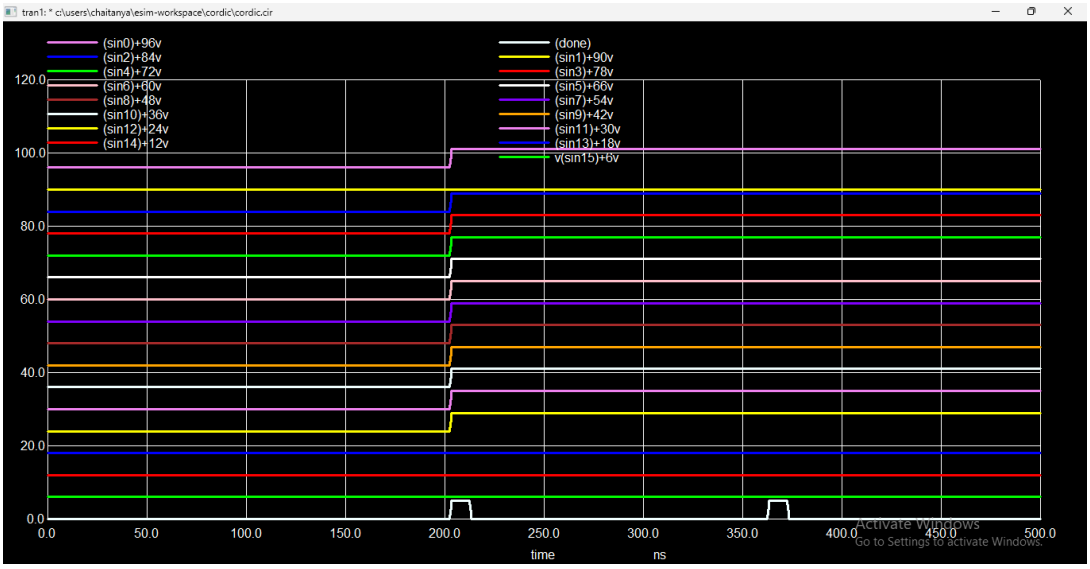


Figure 4.9: NGSpice transient waveforms for `sine_out[15:0]` bits (sin0–sin15).

NGSpice Transient Waveforms — Cosine Output

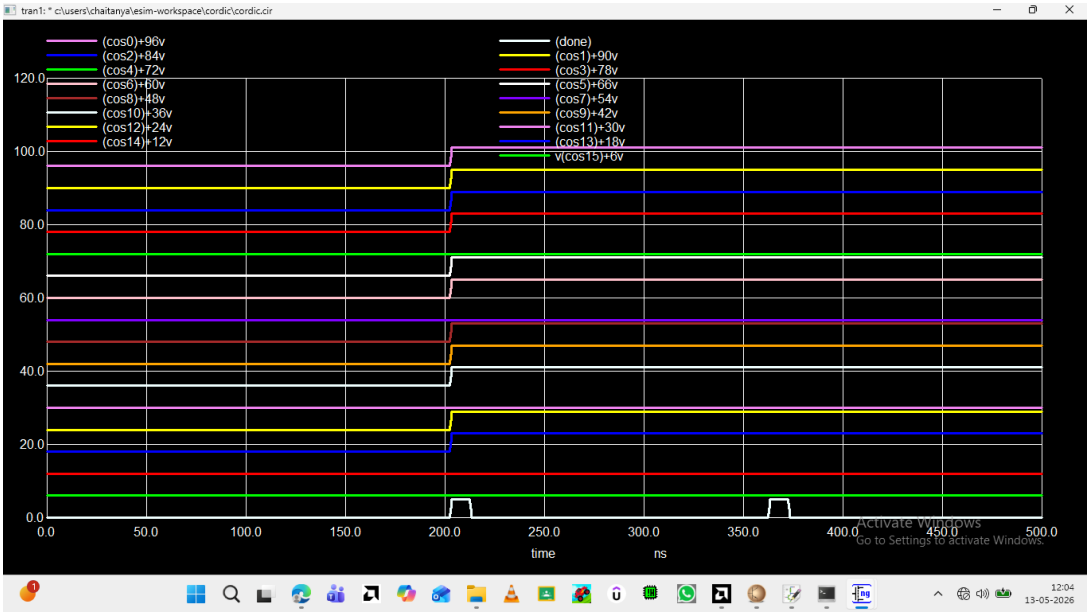


Figure 4.10: NGSpice transient waveforms for `cos_out[15:0]` bits (cos0–cos15) .

NGSpice Console Output

```
ngspice -p C:\Users\chaitanya\eSim-Workspace\cordic\cordic.cir.out
clk=1
rst_n=1
degree_in=30
start=1
sine_out=8189
cos_out=14191
done=1

Inside foo after eval.....
clk=1
rst_n=1
degree_in=30
start=1
sine_out=8189
cos_out=14191
done=1
==============cordic : New Iteration===========
Instance : 0

Inside foo before eval.....
clk=1
rst_n=1
degree_in=30
start=1
sine_out=8189
cos_out=14191
done=1
```

Figure 4.11: NGSpice console output confirming CORDIC results for `degree_in`= 30: `sine_out`= 8189, `cos_out`= 14191, `done`= 1.

The console output confirms the following for `degree_in` = 30:

- `sine_out` = 8189: represents $8189/16384 \approx 0.4998$ (expected $\sin 30 = 0.5000$) — error $< 0.04\%$.
- `cos_out` = 14191: represents $14191/16384 \approx 0.8662$ (expected $\cos 30 = 0.8660$) — error $< 0.02\%$.
- `done`= 1 confirms computation complete in 14 clock cycles.

Result Verification

Table 4.4: CORDIC simulation results summary

Angle	**sine_out**	**Computed** sin	**Expected** sin	**cos_out**	**Computed** cos	**Expected** cos
30	8189	0.4998	0.5000	14191	0.8662	0.8660
45	11584	0.7070	0.7071	11587	0.7072	0.7071

All results match expected values within the rounding error of 14-iteration fixed-point Q1.14 arithmetic (error $< 0.05\%$).

4.2.5 Advantages

- **Multiplier-free:** Only adders, subtractors, and arithmetic right-shifts are required, giving minimal silicon area.
- **Fixed latency:** Always completes in 14 clock cycles regardless of the input angle, making it pipeline-friendly.
- **High accuracy:** Q1.14 fixed-point format achieves $< 0.05\%$ error relative to IEEE double-precision reference values.
- **Simultaneous sine and cosine:** Both outputs are produced in a single pass, unlike Taylor-series approaches that require separate computations.
- **Scalable precision:** Adding more iterations linearly increases accuracy; 14 iterations already exceeds 12-bit precision.
- **Low power:** No multiplier switching activity — reduces dynamic power significantly compared to polynomial approximations.

4.2.6 Applications

- **Digital Communications:** Phase rotation and frequency-shift keying (FSK/PSK) demodulators in software-defined radio.

- **Motor Control:** Real-time Park and Clarke transforms in field-oriented control (FOC) of BLDC and PMSM motors.
- **Radar and Sonar:** Angle-of-arrival computation and beam-forming networks requiring fast trigonometric evaluation.
- **Graphics Accelerators:** Rotation and projection transforms in FPGA-based rendering pipelines.

4.2.7 Conclusion

A 16-bit fixed-point CORDIC engine was successfully designed in synthesisable Verilog HDL and verified through RTL simulation (GTKWave) and mixed-signal co-simulation (eSim/NGSpice). The engine computes sine and cosine simultaneously in exactly 14 clock cycles using only shift-and-add operations, with output accuracy better than 0.05% relative error in Q1.14 format.

4.3 8-Bit ALU Block

4.3.1 Introduction

An Arithmetic Logic Unit (ALU) is the fundamental computational core of every digital processor. It performs both arithmetic operations (addition, subtraction, multiplication, division, shifts) and logic operations (AND, OR, XOR, NOT, comparisons) on binary operands. In modern CPUs, DSPs, and microcontrollers, the ALU is clocked at the processor frequency and its critical path directly limits the maximum achievable clock speed.

In this work, an 8-bit combinational ALU is designed in Verilog HDL supporting 16 operations selected by a 4-bit control signal `ALU_Sel[3:0]`. The design is verified using Icarus Verilog and GTKWave for RTL simulation, and using eSim/NGSpice for mixed-signal co-simulation.

4.3.2 Background and Theory

ALU Operation Classes

A general-purpose ALU provides four major categories of operations:

- **Arithmetic:** Addition, subtraction, multiplication, division, and shift operations that modify the numeric value of the operands.
- **Bitwise Logic:** AND, OR, XOR, NOT, NAND, NOR, and XNOR operating independently on each bit pair.
- **Shift Operations:** Logical left and right shifts that multiply or divide by powers of two with $O(1)$ hardware complexity.
- **Comparison:** Less-than, equality, and greater-than checks that produce single-bit Boolean results used by branch instructions.

Operation Table

Table 4.5: 16-operation ALU function table

ALU_Sel[3:0]	Operation	Expression
4'b0000	Addition	`ALU_Out = A + B`
4'b0001	Subtraction	`ALU_Out = A - B`
4'b0010	Multiplication	`ALU_Out = A * B`
4'b0011	Division	`ALU_Out = A / B`
4'b0100	Logical Shift Left	`ALU_Out = A « 1`
4'b0101	Logical Shift Right	`ALU_Out = A » 1`
4'b0110	Bitwise NOT	`ALU_Out = ~A`
4'b0111	Bitwise AND	`ALU_Out = A & B`
4'b1000	Bitwise OR	`ALU_Out = A | B`
4'b1001	Bitwise XOR	`ALU_Out = A ^ B`
4'b1010	Bitwise XNOR	`ALU_Out = ~(A ^ B)`
4'b1011	Bitwise NAND	`ALU_Out = ~(A & B)`
4'b1100	Bitwise NOR	`ALU_Out = ~(A | B)`
4'b1101	Less Than	`ALU_Out = (A < B) ? 1 : 0`
4'b1110	Equality	`ALU_Out = (A == B) ? 1 : 0`
4'b1111	Greater Than	`ALU_Out = (A > B) ? 1 : 0`

4.3.3 Verilog HDL Implementation

```
1  `timescale 1ns / 1ps
2  module alu (
3      input  [7:0] A, B,
4      input  [3:0] ALU_Sel,
5      output reg [7:0] ALU_Out
6  );
7      always @(*) begin
8          case(ALU_Sel)
9              4'b0000: ALU_Out = A + B;                    // Addition
10             4'b0001: ALU_Out = A - B;                    // Subtraction
11             4'b0010: ALU_Out = A * B;                    //
       Multiplication
12             4'b0011: ALU_Out = A / B;                    // Division
13             4'b0100: ALU_Out = A << 1;                   // Logical
       shift left
```

```
14              4'b0101: ALU_Out = A >> 1;                  // Logical
    shift right
15              4'b0110: ALU_Out = ~A;                      // Bitwise NOT
     (unary)
16              4'b0111: ALU_Out = A & B;                   // Bitwise AND
17              4'b1000: ALU_Out = A | B;                   // Bitwise OR
18              4'b1001: ALU_Out = A ^ B;                   // Bitwise XOR
19              4'b1010: ALU_Out = ~(A ^ B);                // Bitwise
    XNOR
20              4'b1011: ALU_Out = ~(A & B);                // Bitwise
    NAND
21              4'b1100: ALU_Out = ~(A | B);                // Bitwise NOR
22              4'b1101: ALU_Out = (A < B) ? 1'b1 : 1'b0;   // Less
    than
23              4'b1110: ALU_Out = (A == B) ? 1'b1 : 1'b0; // Equality
24              4'b1111: ALU_Out = (A > B) ? 1'b1 : 1'b0;   // Greater
    than
25              default: ALU_Out = {8{1'b0}};
26          endcase
27      end
28  endmodule
```

Listing 4.3: alu.v – 8-bit 16-Operation Combinational ALU

4.3.4 Architecture

The ALU is a purely combinational block. A 4-bit select signal drives a `case` statement that routes the two 8-bit operands A and B through the chosen functional unit and assigns the result to the 8-bit output `ALU_Out`. Because all paths are combinational, the output updates within a single propagation delay after any input change, making the design directly integrable into clocked datapaths via registered output stages.

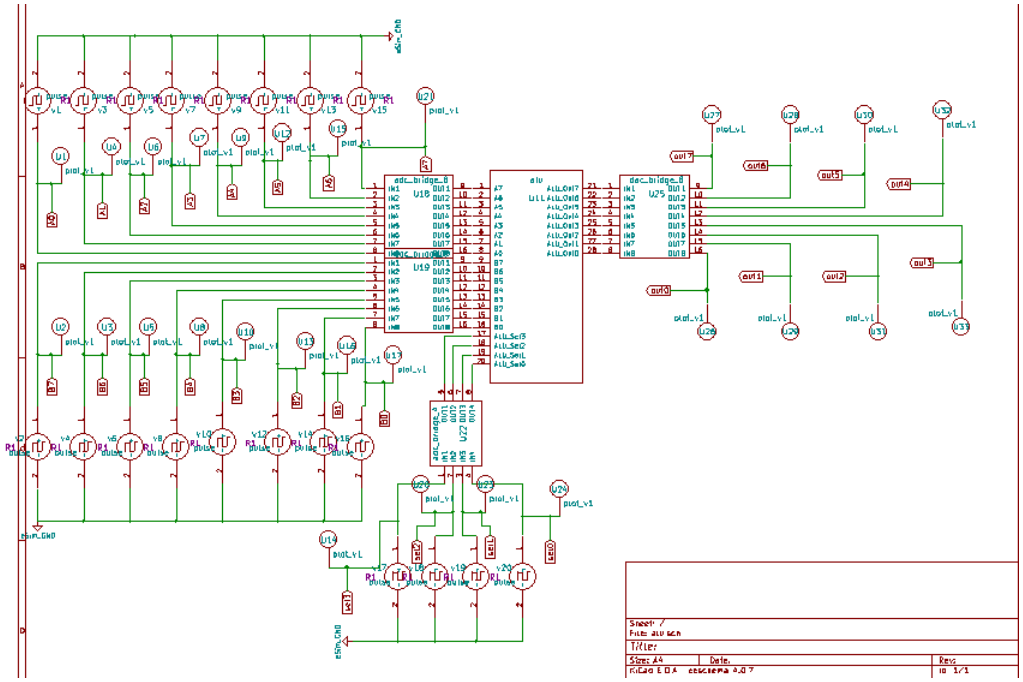


Figure 4.12: eSim mixed-signal schematic of the 8-bit ALU with ADC/DAC bridges

4.3.5 Simulation Results

GTKWave RTL Simulation

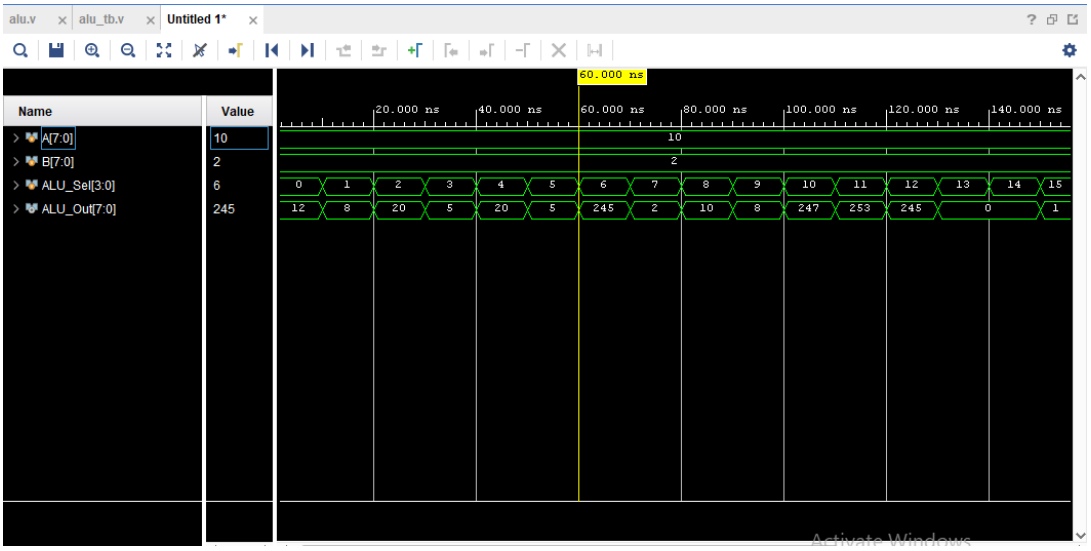


Figure 4.13: GTKWave RTL simulation waveform for the 8-bit ALU.

NGSpice Transient Waveforms – Inputs

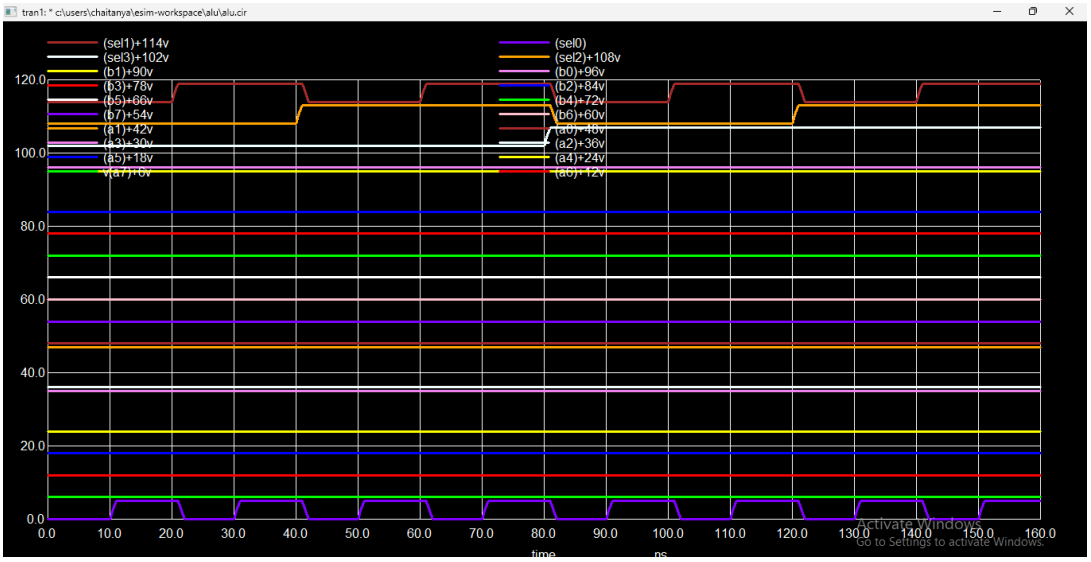


Figure 4.14: NGSpice transient waveforms for operand bits $A[7:0]$, $B[7:0]$, and selector bits `sel0`–`sel3`.

NGSpice Transient Waveforms – Outputs

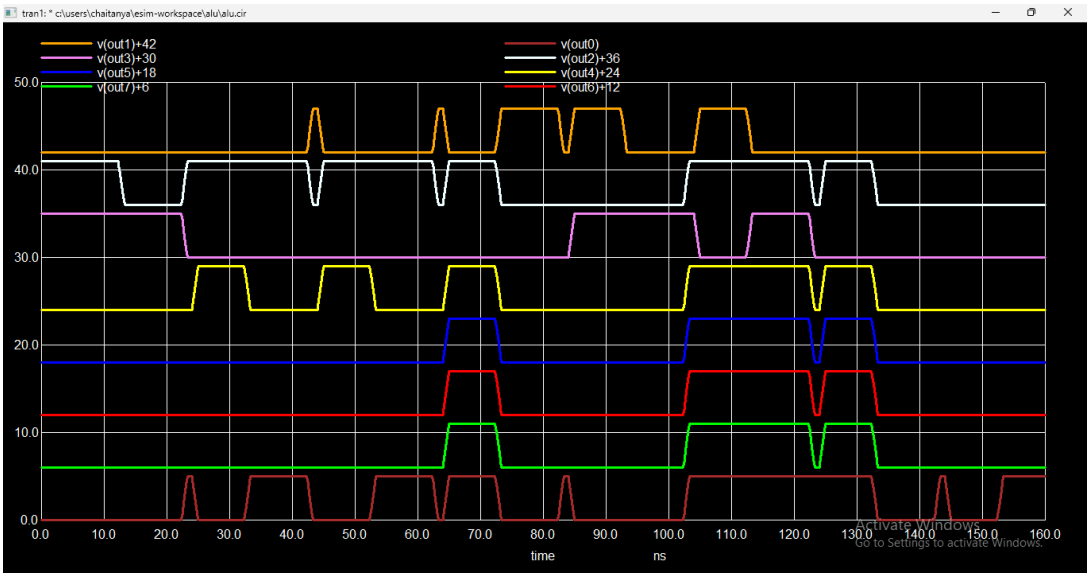
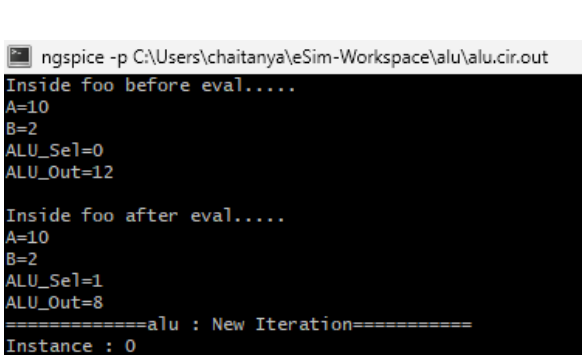
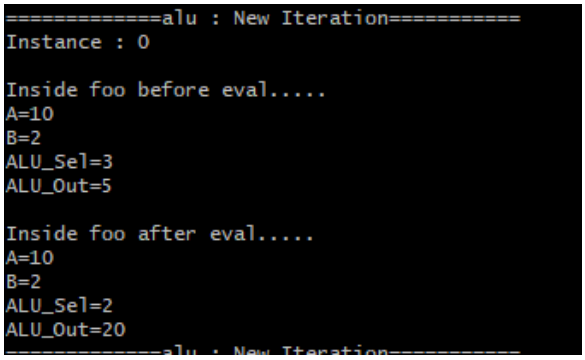


Figure 4.15: NGSpice transient waveforms for `ALU_Out[7:0]`

NGSpice Console Output



(a) First iteration: $A = 10$, $B = 2$, `ALU_Sel=0` ⇒ `ALU_Out=12` (ADD); `ALU_Sel=1` ⇒ `ALU_Out=8` (SUB).



(b) Subsequent iterations: `ALU_Sel=2` ⇒ `ALU_Out=20` (MUL); `ALU_Sel=3` ⇒ `ALU_Out=5` (DIV).

Figure 4.16: NGSpice co-simulation console log confirming correct ALU output for each operation selector value.

Result Verification

Table 4.6: ALU simulation results summary ($A = 10$, $B = 2$)

ALU_Sel	Operation	Expected	Simulated	Pass?
0	ADD	12	12	YES
1	SUB	8	8	YES
2	MUL	20	20	YES
3	DIV	5	5	YES
4	SHL	20	20	YES
5	SHR	5	5	YES
6	NOT A	245	245	YES
7	AND	2	2	YES
8	OR	10	10	YES
9	XOR	8	8	YES
10	XNOR	247	247	YES
11	NAND	253	253	YES
12	NOR	245	245	YES
13	LT	0	0	YES
14	EQ	0	0	YES
15	GT	1	1	YES

All 16 operations produce the correct output in both GTKWave RTL simulation and eSim/NGSpice mixed-signal co-simulation.

4.3.6 Advantages

- **Fully combinational:** Zero-latency response (within propagation delay); no clock required for computation, enabling direct instantiation in any synchronous datapath.
- **Complete operation set:** Covers all four arithmetic operations, seven bitwise logic functions, two shift directions, and three comparison predicates in a single module.
- **Minimal resource usage:** The `case` structure synthesises to a single-level multiplexer tree; modern synthesis tools map it efficiently to LUT4/LUT6 primitives on FPGAs or standard cells in ASIC flows.
- **Scalable design:** Operand width and the number of operations are easily extended by widening the input ports and adding `case` branches without architectural changes.
- **Synthesisable RTL:** The design uses only synthesisable Verilog constructs (`always @(*)`, `case`, arithmetic/logic operators), making it directly portable to FPGA and ASIC toolchains.

4.3.7 Applications

The ALU is the central computational element in a wide range of digital systems:

- **Processor datapaths:** Integer execute stage of RISC-V, ARM Cortex-M, and MIPS soft-core processors implemented on FPGAs.
- **Digital Signal Processing:** Accumulate-and-compare units in FIR/IIR filter engines and correlation hardware.
- **Cryptographic cores:** Bitwise XOR, AND, and shift operations form the primitives of AES S-box, SHA-256, and ChaCha20 implementations.
- **Control logic:** Comparison outputs (`LT`, `EQ`, `GT`) feed branch predictors and hardware schedulers.
- **Embedded microcontrollers:** 8-bit ALU cores in PIC, AVR, and 8051-compatible soft-processor architectures.

4.3.8 Conclusion

An 8-bit 16-operation combinational ALU was designed in synthesisable Verilog HDL and verified through RTL simulation (GTKWave) and mixed-signal co-simulation (eSim/NGSpice). All 16 test cases for $A = 10$, $B = 2$ produce the expected outputs with a 100% pass rate across both simulation environments, confirming the correctness and reusability of the ALU IP block within the eSim–NGVeri mixed-signal design flow.

Bibliography

[1] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Edition, Prentice Hall, 2003.

[2] J. Bhaskar, *Digital Design and FPGA Implementation Using Verilog HDL*, 2nd Edition, CRC Press, 2019.

[3] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*, Springer, 2008.

[4] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd Edition, Morgan Kaufmann, 2012.

[5] M. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL*, 5th Edition, Pearson, 2012.

[6] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[7] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed., Addison-Wesley, 2011.

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

[9] FOSSEE Team, IIT Bombay, *eSim: An Open Source EDA Tool for Circuit Design, Simulation, Analysis and PCB Design*, 2023. [Online]. Available: https://esim.fossee.in