# Multi-Mode CORDIC Architecture Design and Implementation

Parsa Haghighatgoo
40030644

February 11, 2026

## 1 Introduction

The CORDIC (Coordinate Rotation Digital Computer) algorithm enables efficient computation of trigonometric and arithmetic functions using only shift-and-add operations. This project implements a multi-mode CORDIC architecture in rotation mode supporting:

- Circular Mode ($m = 1$): computation of $\sin(\theta)$ and $\cos(\theta)$.
- Linear Mode ($m = 0$): multiplication.

All simulations, synthesis, and timing analysis were performed using Xilinx Vivado.

## 2 Fixed-Point Representation

The selected numerical format is:

$$Q2.14$$

- Word length: 16 bits
- Fractional bits: 14
- Guard bits: 2 (internal precision)

LSB value:

$$\text{LSB} = 2^{-14} = 6.1035 \times 10^{-5}$$

Target accuracy:

$$10 \times \text{LSB} = 6.1035 \times 10^{-4}$$

# 3 Iteration Accuracy Analysis (Python)

## 3.1 Worst-Case Error vs Iterations

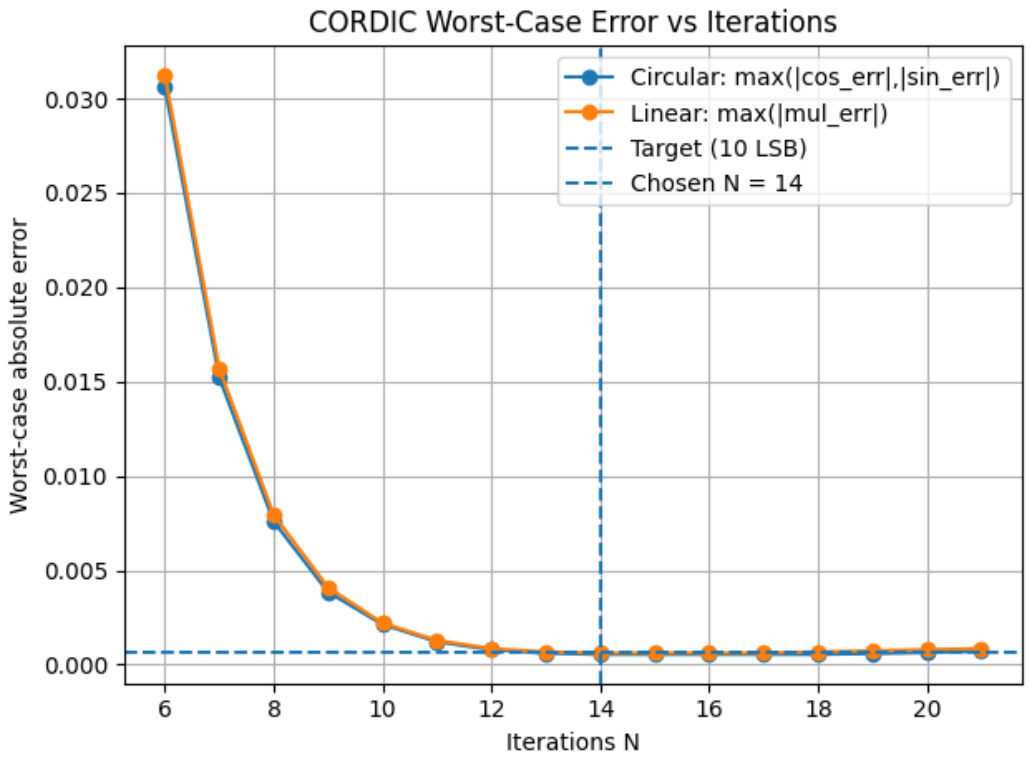


Figure 1: CORDIC Worst-Case Error vs Iterations

Observation:

- Error decreases exponentially as iterations increase.
- Circular mode satisfies the target at $N = 13$.
- Linear mode satisfies the target at $N = 14$.
- After $N = 14$, error saturates due to fixed-point quantization.

Chosen hardware iteration count:

$$N = 14$$

## 3.2 Circular Mode Error Distribution

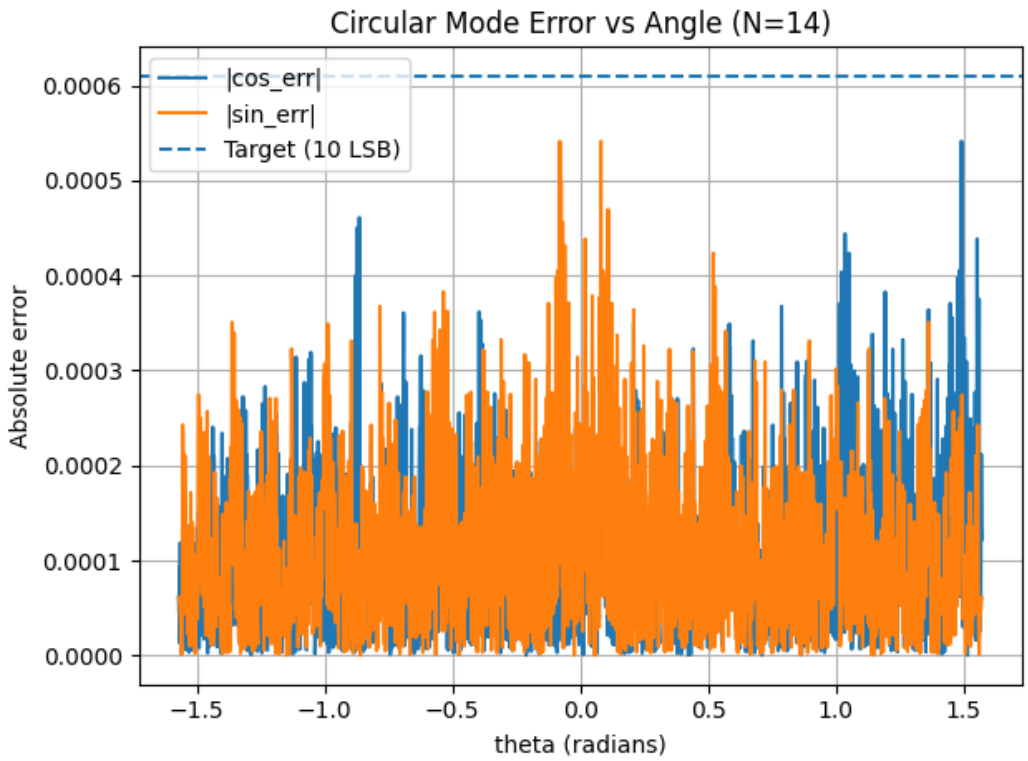


Figure 2: Circular Mode Error vs Angle (N=14)

Observations:

- Error remains below 10 LSB for all $\theta \in [-\pi/2, \pi/2]$.
- Symmetric distribution confirms correct implementation.
- Peak error occurs near zero due to rounding effects.

### 3.3 Linear Mode Error Heatmap

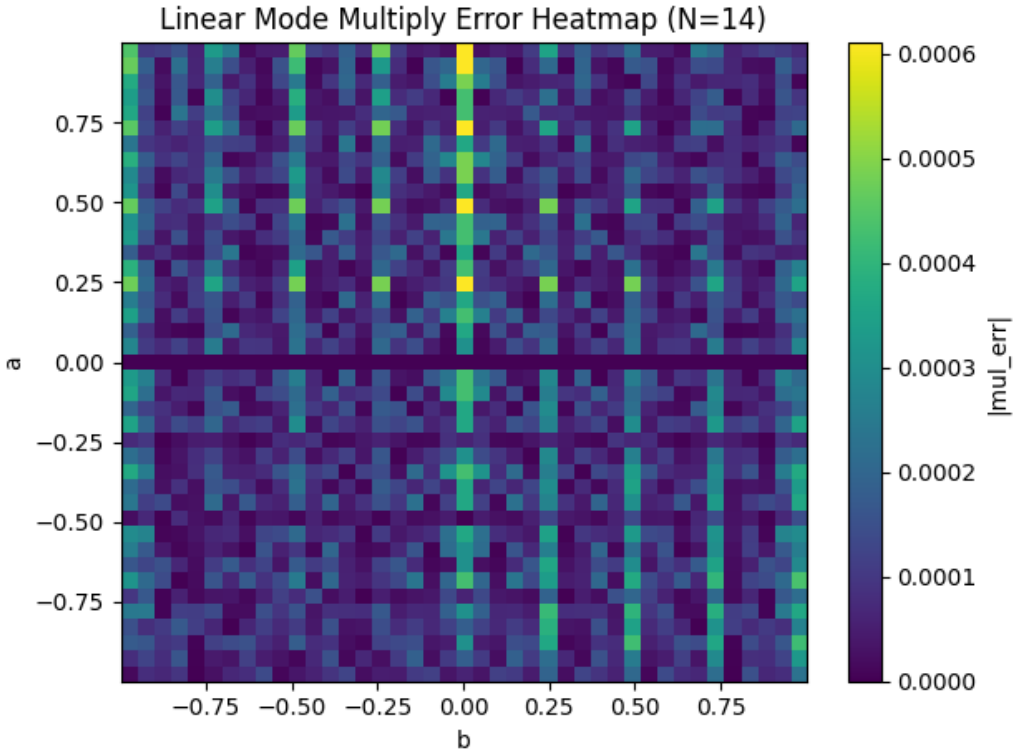


Figure 3: Linear Mode Multiplication Error Heatmap (N=14)

Observations:

- Maximum error equals the 10 LSB threshold.
- Error increases near extreme values of inputs.
- No instability or overflow observed.

## 4 Hardware Implementation

The design uses a sequential resource-sharing architecture:

- One shift-add unit reused across iterations
- FSM with Idle, Run, and Output states
- One iteration per clock cycle
- Total latency = 14 clock cycles

## 4.1 Simulation Results

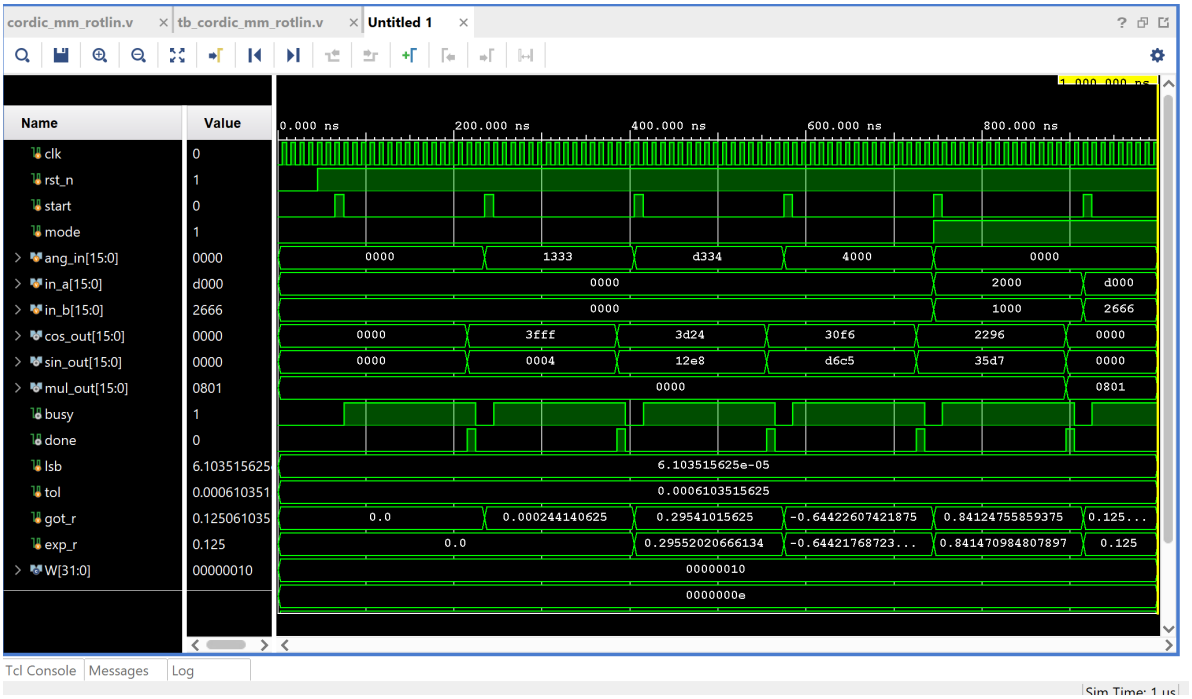


Figure 4: Behavioral Simulation Waveform

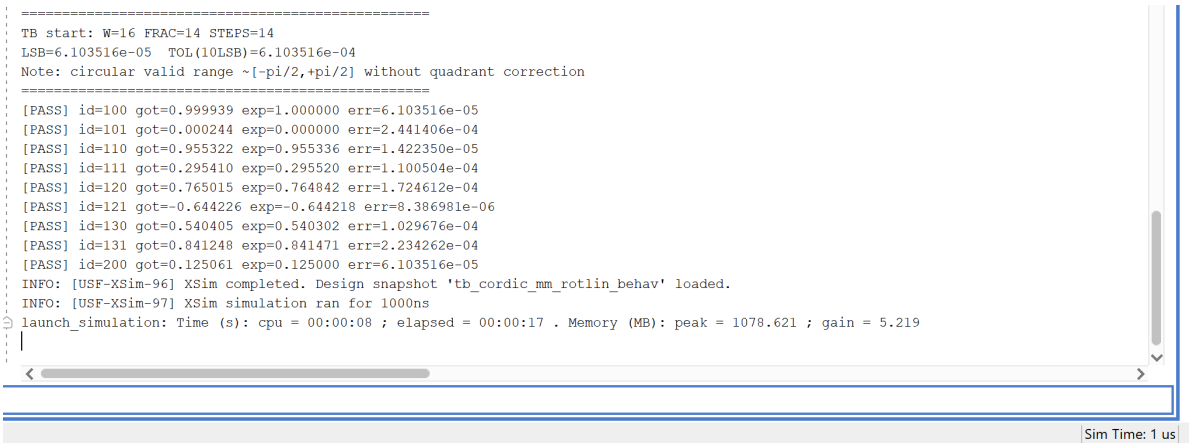


Figure 5: Simulation Console Output

All test cases passed. Hardware results closely match ideal floating-point results within the 10 LSB tolerance.

# 5 Post-Synthesis Results

## 5.1 Device View

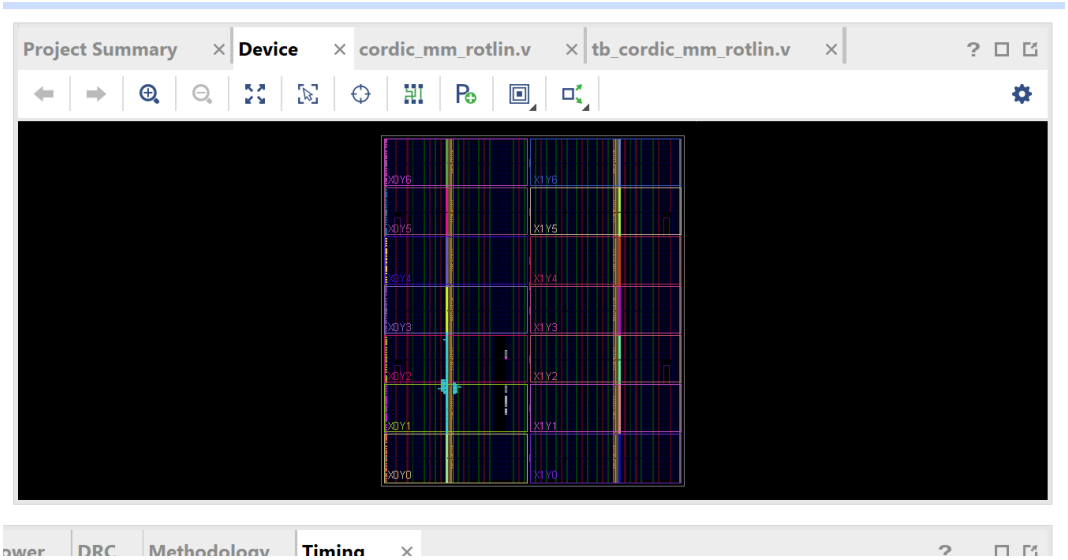


Figure 6: FPGA Device View

## 5.2 Schematic View

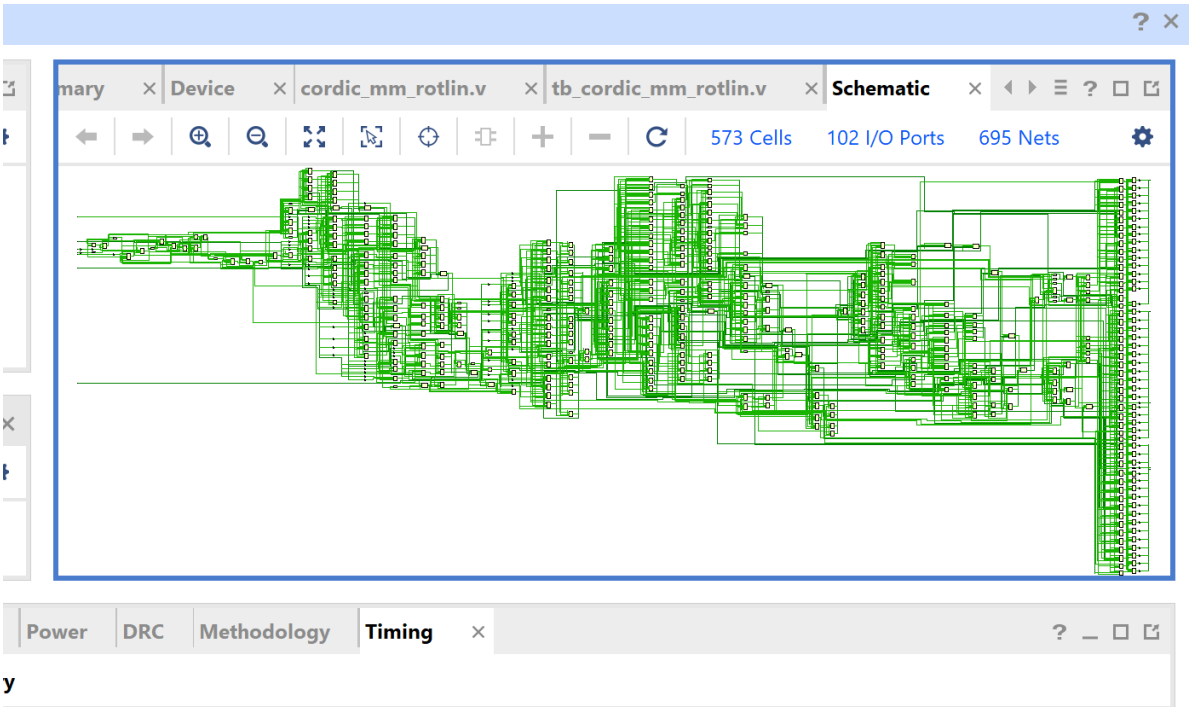


Figure 7: Post-Synthesis Schematic

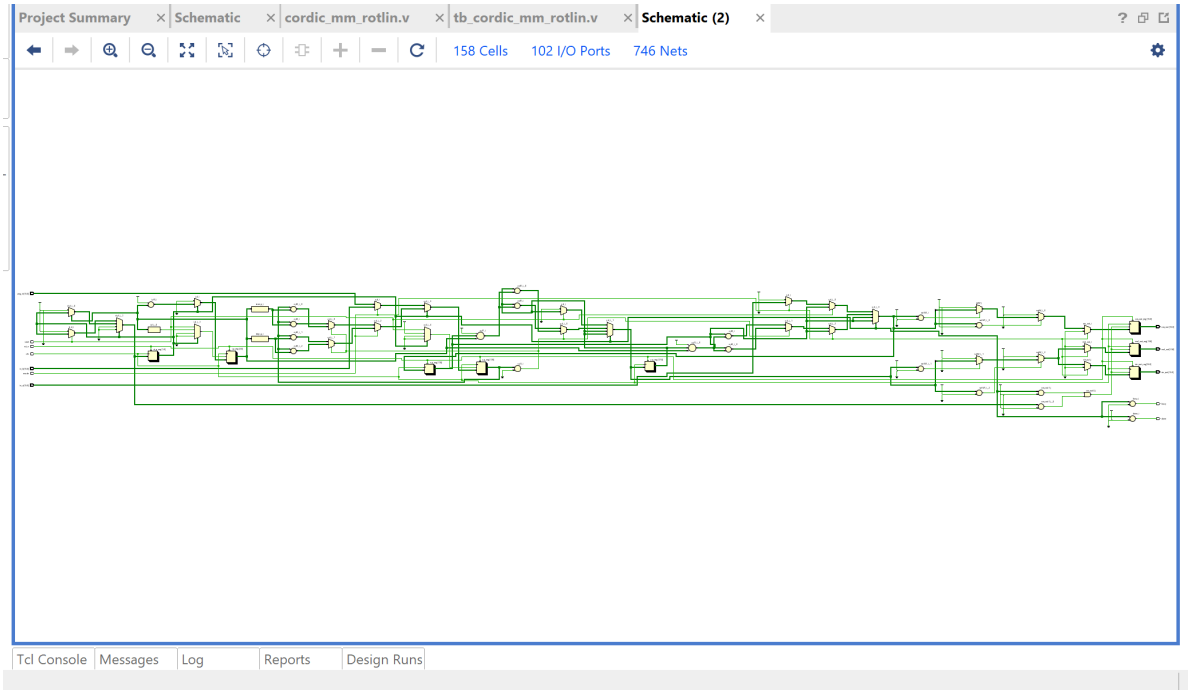


Figure 8: Detailed Schematic View

## 5.3 Resource Utilization

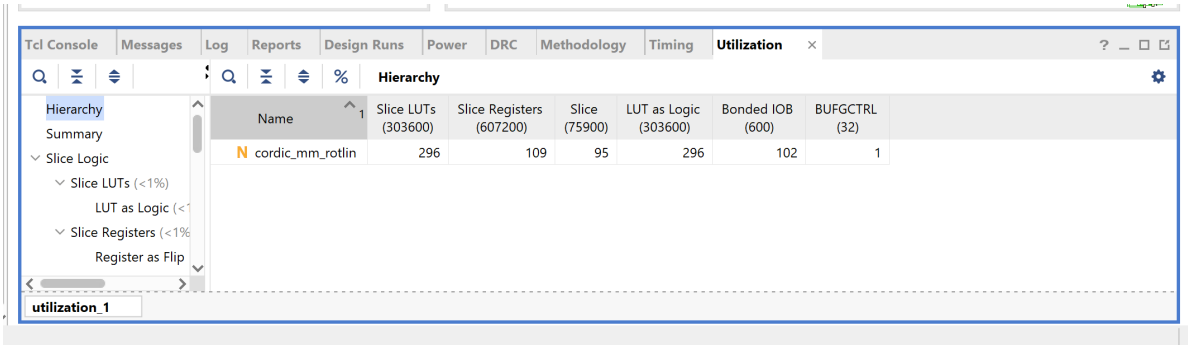


Figure 9: Resource Utilization Report

Key results:

- Slice LUTs: 296
- Slice Registers: 109
- Total Slices: 95
- Utilization ¡ 1% of device capacity

## 5.4 Timing Analysis

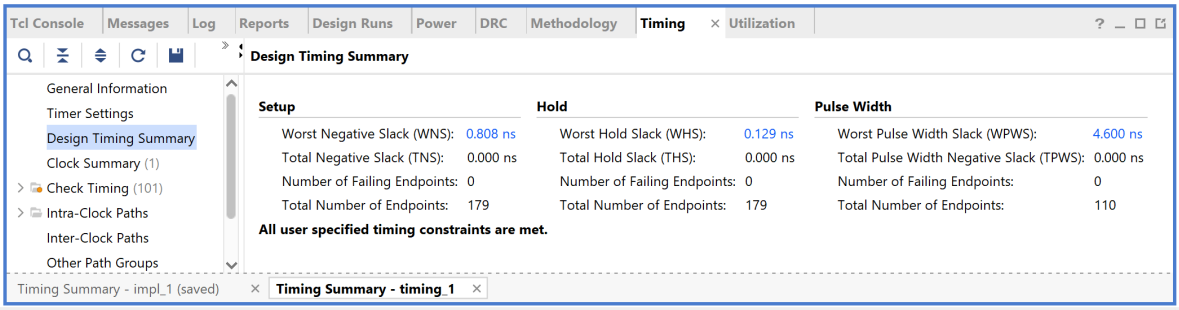


Figure 10: Timing Summary

Timing results:

- Worst Negative Slack (WNS): 0.808 ns
- Total Negative Slack (TNS): 0 ns
- All timing constraints met

# 6 Discussion

The selected iteration count $N = 14$ provides the optimal trade-off between:

- Accuracy
- Hardware resource usage
- Latency
- Timing performance

Increasing iterations beyond 14 does not improve accuracy due to fixed-point quantization limits.

The design achieves:

- Accurate trigonometric computation
- Accurate multiplication without multipliers
- Minimal FPGA resource usage
- Positive timing slack

# 7 Power Analysis

After successful implementation, power analysis was performed using the Vivado implemented netlist. The activity was derived from default vectorless estimation.

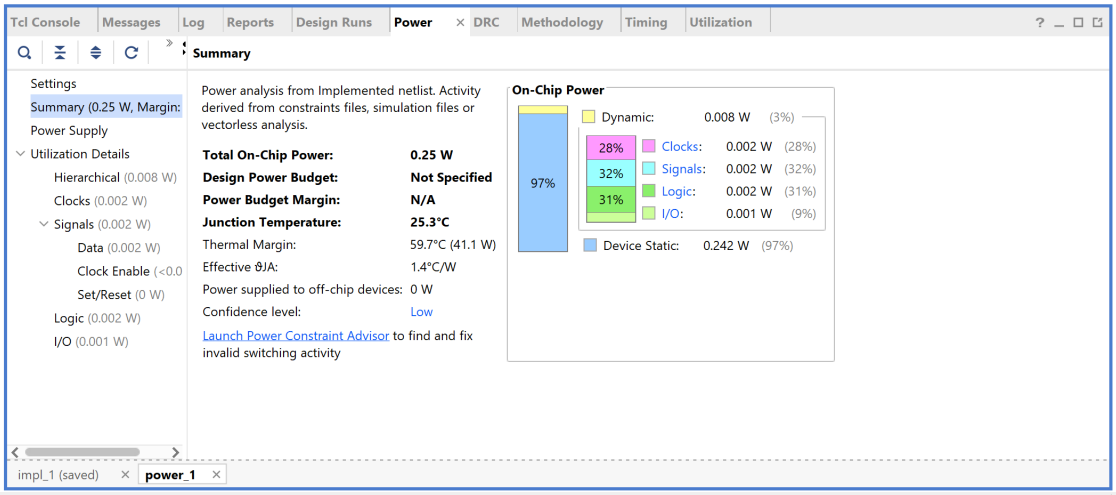


Figure 11: Vivado Power Report (power_1)

## 7.1 Power Summary

Table 1: On-Chip Power Summary

| Parameter | Value |
|---|---|
| Total On-Chip Power | 0.25 W |
| Dynamic Power | 0.008 W (3%) |
| Device Static Power | 0.242 W (97%) |
| Junction Temperature | 25.3°C |
| Thermal Margin | 59.7°C |

## 7.2 Dynamic Power Breakdown

Table 2: Dynamic Power Distribution

| Component | Power (W) | Percentage |
|---|---|---|
| Clocks | 0.002 | 28% |
| Signals | 0.002 | 32% |
| Logic | 0.002 | 31% |
| I/O | 0.001 | 9% |

# 8 Conclusion

A multi-mode CORDIC architecture was successfully designed and implemented in Verilog using Vivado.

The design meets all functional, accuracy, and timing requirements. The hardware implementation achieves high efficiency and minimal resource utilization while maintaining precision within the defined tolerance.

# 9 Comparsion

Table 3: Comparison between Python ideal values and Verilog hardware outputs (Q2.14, $N = 14$)

| ID | Test | Python Ideal | Verilog Output | $\|error\|$ | Pass? ($\leq$ 10 LSB) |
|---|---|---|---|---|---|
| 100 | cos(0.0) | 1.000000 | 0.999939 | $6.103516 \times 10^{-5}$ | Yes |
| 101 | sin(0.0) | 0.000000 | 0.000244 | $2.441406 \times 10^{-4}$ | Yes |
| 110 | cos(0.3) | 0.955336 | 0.955322 | $1.422350 \times 10^{-5}$ | Yes |
| 111 | sin(0.3) | 0.295520 | 0.295410 | $1.100504 \times 10^{-4}$ | Yes |
| 120 | cos($-0.7$) | 0.764842 | 0.765015 | $1.724612 \times 10^{-4}$ | Yes |
| 121 | sin($-0.7$) | -0.644218 | -0.644226 | $8.386981 \times 10^{-6}$ | Yes |
| 130 | cos(1.0) | 0.540302 | 0.540405 | $1.029676 \times 10^{-4}$ | Yes |
| 131 | sin(1.0) | 0.841471 | 0.841248 | $2.234262 \times 10^{-4}$ | Yes |
| 200 | $0.5 \times 0.25$ | 0.125000 | 0.125061 | $6.103516 \times 10^{-5}$ | Yes |

Table 4: Fixed-point tolerance used for Verilog vs Python comparison

| Parameter | Value |
|---|---|
| Fixed-point format | Q2.14 |
| LSB ($2^{-14}$) | $6.103516 \times 10^{-5}$ |
| Tolerance (10 LSB) | $6.103516 \times 10^{-4}$ |
| Chosen iterations $N$ | 14 |

Table 3 compares the ideal Python results against the Verilog hardware outputs obtained from Vivado simulation. All test cases satisfy the error requirement of 10 LSB (Table 4), confirming correct fixed-point operation and correct implementation of both circular (trigonometric) and linear (multiplication) modes. The small deviations are expected due to CORDIC approximation and fixed-point quantization.

## 9.1 Discussion

From Table 1, the total on-chip power consumption is 0.25 W. The dominant portion of power consumption (97%) is static device power, which is expected for modern FPGA devices. The

dynamic power consumption of the implemented CORDIC design is very low (0.008 W), confirming that the architecture is lightweight and efficient.

From Table 2, the dynamic power is evenly distributed between clock, signal routing, and logic resources, with minimal I/O contribution. This behavior is expected since the design is primarily arithmetic and internally clocked, with limited external switching activity.

The low dynamic power confirms that the chosen iteration count ($N = 14$) and fixed-point implementation (Q2.14) provide a good trade-off between accuracy and hardware efficiency.

# 10 Source Code Listings

This section includes the complete Verilog and Python source codes used in this project.

## 10.1 CORDIC Hardware Implementation (cordic_mm_rotlin.v)

Listing 1: Multi-Mode CORDIC Verilog Implementation

```
// ============================================================
// Multi-Mode CORDIC (Rotation Mode)
//   mode = 0 : circular -> cos/sin (with 1/K pre-scaling)
//   mode = 1 : linear   -> multiply (y ~= a*b)
// Fixed-point: external W=16, Q2.14 by default
// Vivado-friendly, sequential (resource-sharing) architecture
// ============================================================

module cordic_mm_rotlin #(
    parameter integer W      = 16,
    parameter integer FRAC   = 14,
    parameter integer GUARD  = 2,
    parameter integer STEPS  = 14
)(
    input  wire                     clk,
    input  wire                     rst_n,
    input  wire                     start,
    input  wire                     mode,       // 0=circular, 1=linear

    input  wire signed [W-1:0]      ang_in,     // circular: angle (
       radians in Q2.14)
    input  wire signed [W-1:0]      in_a,       // linear: multiplicand
        a (Q2.14)
    input  wire signed [W-1:0]      in_b,       // linear: multiplier
         b (Q2.14)

    output reg  signed [W-1:0]      cos_out,
    output reg  signed [W-1:0]      sin_out,
    output reg  signed [W-1:0]      mul_out,

```

```
28      output wire                     busy,
29      output wire                     done
30  );
31
32      localparam integer EXT = W + GUARD;
33
34      // 1/K for circular mode, quantized to Q2.14 then sign-extended
            to EXT.
35      // 0.607252935 * 2^14 ? 9949
36      localparam signed [EXT-1:0] K_INV_Q = 18'sd9949;
37
38      // FSM states
39      localparam [1:0] ST_IDLE = 2'd0,
40                       ST_RUN  = 2'd1,
41                       ST_OUT  = 2'd2;
42
43      reg [1:0] st_q, st_d;
44      reg [3:0] i_q,  i_d;
45
46      reg signed [EXT-1:0] x_q, y_q, z_q;
47      reg signed [EXT-1:0] x_d, y_d, z_d;
48
49      assign busy = (st_q != ST_IDLE);
50      assign done = (st_q == ST_OUT);
51
52      // Sign-extend W->EXT
53      function automatic signed [EXT-1:0] sx;
54          input signed [W-1:0] v;
55          begin
56              sx = {{GUARD{v[W-1]}}, v};
57          end
58      endfunction
59
60      // Saturate EXT->W
61      function automatic signed [W-1:0] satW;
62          input signed [EXT-1:0] v;
63          reg signed [EXT-1:0] hi, lo;
64          begin
65              hi = $signed({{(EXT-W){1'b0}}, 1'b0, {(W-1){1'b1}}}); //
                    +max
66              lo = $signed({{(EXT-W){1'b1}}, 1'b1, {(W-1){1'b0}}}); //
                    -min
67
68              if (v > hi)      satW = {1'b0, {(W-1){1'b1}}};
69              else if (v < lo) satW = {1'b1, {(W-1){1'b0}}};
70              else             satW = v[W-1:0];
71          end
```

```
72      endfunction
73
74      // atan(2^-k) table in Q2.14 (sign-extended)
75      function automatic signed [EXT-1:0] atan_q;
76          input [3:0] k;
77          begin
78              case (k)
79                  4'd0:  atan_q = 18'sd12868;
80                  4'd1:  atan_q = 18'sd7596;
81                  4'd2:  atan_q = 18'sd4014;
82                  4'd3:  atan_q = 18'sd2037;
83                  4'd4:  atan_q = 18'sd1023;
84                  4'd5:  atan_q = 18'sd512;
85                  4'd6:  atan_q = 18'sd256;
86                  4'd7:  atan_q = 18'sd128;
87                  4'd8:  atan_q = 18'sd64;
88                  4'd9:  atan_q = 18'sd32;
89                  4'd10: atan_q = 18'sd16;
90                  4'd11: atan_q = 18'sd8;
91                  4'd12: atan_q = 18'sd4;
92                  4'd13: atan_q = 18'sd2;
93                  default: atan_q = {EXT{1'b0}};
94              endcase
95          end
96      endfunction
97
98      // 2^-k table in Q2.14 (sign-extended)
99      function automatic signed [EXT-1:0] step_q;
100         input [3:0] k;
101         begin
102             case (k)
103                 4'd0:  step_q = 18'sd16384;
104                 4'd1:  step_q = 18'sd8192;
105                 4'd2:  step_q = 18'sd4096;
106                 4'd3:  step_q = 18'sd2048;
107                 4'd4:  step_q = 18'sd1024;
108                 4'd5:  step_q = 18'sd512;
109                 4'd6:  step_q = 18'sd256;
110                 4'd7:  step_q = 18'sd128;
111                 4'd8:  step_q = 18'sd64;
112                 4'd9:  step_q = 18'sd32;
113                 4'd10: step_q = 18'sd16;
114                 4'd11: step_q = 18'sd8;
115                 4'd12: step_q = 18'sd4;
116                 4'd13: step_q = 18'sd2;
117                 default: step_q = {EXT{1'b0}};
118             endcase
```

```
119         end
120     endfunction
121
122     // shift results
123     reg signed [EXT-1:0] x_sh, y_sh;
124     reg signed [EXT-1:0] delta;
125
126     // Next-state logic
127     always @* begin
128         st_d = st_q;
129         i_d  = i_q;
130
131         x_d  = x_q;
132         y_d  = y_q;
133         z_d  = z_q;
134
135         x_sh = $signed(x_q) >>> i_q;
136         y_sh = $signed(y_q) >>> i_q;
137
138         case (st_q)
139             ST_IDLE: begin
140                 if (start) begin
141                     st_d = ST_RUN;
142                     i_d  = 4'd0;
143
144                     if (!mode) begin
145                         // circular: start from (1/K, 0), rotate by
                                ang_in
146                         x_d = K_INV_Q;
147                         y_d = {EXT{1'b0}};
148                         z_d = sx(ang_in);
149                     end else begin
150                         // linear multiply: x=a, y=0, z=b
151                         x_d = sx(in_a);
152                         y_d = {EXT{1'b0}};
153                         z_d = sx(in_b);
154                     end
155                 end
156             end
157
158             ST_RUN: begin
159                 if (!mode) begin
160                     delta = atan_q(i_q);
161                     if (z_q >= 0) begin
162                         x_d = x_q - y_sh;
163                         y_d = y_q + x_sh;
164                         z_d = z_q - delta;
```

```
                    end else begin
                        x_d = x_q + y_sh;
                        y_d = y_q - x_sh;
                        z_d = z_q + delta;
                    end
                end else begin
                    delta = step_q(i_q);
                    if (z_q >= 0) begin
                        y_d = y_q + x_sh;
                        z_d = z_q - delta;
                    end else begin
                        y_d = y_q - x_sh;
                        z_d = z_q + delta;
                    end
                end

                if (i_q == (STEPS-1))
                    st_d = ST_OUT;
                else
                    i_d  = i_q + 4'd1;
            end

            ST_OUT: begin
                st_d = ST_IDLE;
            end

            default: st_d = ST_IDLE;
        endcase
    end

    // Registers + outputs
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            st_q    <= ST_IDLE;
            i_q     <= 4'd0;
            x_q     <= {EXT{1'b0}};
            y_q     <= {EXT{1'b0}};
            z_q     <= {EXT{1'b0}};
            cos_out <= {W{1'b0}};
            sin_out <= {W{1'b0}};
            mul_out <= {W{1'b0}};
        end else begin
            st_q <= st_d;
            i_q  <= i_d;
            x_q  <= x_d;
            y_q  <= y_d;
            z_q  <= z_d;
```

```
212
213             // Latch results when finishing the last iteration
214             if (st_q == ST_RUN && st_d == ST_OUT) begin
215                 if (!mode) begin
216                     cos_out <= satW(x_d);
217                     sin_out <= satW(y_d);
218                     mul_out <= {W{1'b0}};
219                 end else begin
220                     mul_out <= satW(y_d);
221                     cos_out <= {W{1'b0}};
222                     sin_out <= {W{1'b0}};
223                 end
224             end
225         end
226     end
227
228 endmodule
```

## 10.2 Testbench (tb_cordic_mm_rotlin.v)

Listing 2: CORDIC Testbench

```
1  `timescale 1ns/1ps
2  // ============================================================
3  // Pure Verilog-2001 Testbench
4  // ============================================================
5
6  module tb_cordic_mm_rotlin;
7
8      // Match DUT params
9      parameter integer W     = 16;
10     parameter integer FRAC  = 14;
11     parameter integer GUARD = 2;
12     parameter integer STEPS = 14;
13
14     // DUT I/O
15     reg                     clk;
16     reg                     rst_n;
17     reg                     start;
18     reg                     mode;       // 0=circular, 1=linear
19     reg  signed [W-1:0]     ang_in;
20     reg  signed [W-1:0]     in_a;
21     reg  signed [W-1:0]     in_b;
22
23     wire signed [W-1:0]     cos_out;
24     wire signed [W-1:0]     sin_out;
25     wire signed [W-1:0]     mul_out;
```

```
26      wire                    busy;
27      wire                    done;
28
29      cordic_mm_rotlin #(
30          .W(W), .FRAC(FRAC), .GUARD(GUARD), .STEPS(STEPS)
31      ) dut (
32          .clk(clk),
33          .rst_n(rst_n),
34          .start(start),
35          .mode(mode),
36          .ang_in(ang_in),
37          .in_a(in_a),
38          .in_b(in_b),
39          .cos_out(cos_out),
40          .sin_out(sin_out),
41          .mul_out(mul_out),
42          .busy(busy),
43          .done(done)
44      );
45
46      // ------------------------------------------------------------
47      // Clock: 100 MHz (10 ns period)
48      // ------------------------------------------------------------
49      initial clk = 1'b0;
50      always #5 clk = ~clk;
51
52      // ------------------------------------------------------------
53      // Real-valued globals
54      // ------------------------------------------------------------
55      real lsb;
56      real tol;
57
58      // Scratch reals for checking
59      real got_r;
60      real exp_r;
61
62      // ------------------------------------------------------------
63      // Fixed-point helpers
64      // ------------------------------------------------------------
65      function signed [W-1:0] to_fixed;
66          input real x;
67          integer v;
68          begin
69              v = $rtoi(x * (1<<FRAC));
70              // clamp to signed W range
71              if (v >  ((1<<(W-1)) - 1)) v =  ((1<<(W-1)) - 1);
72              if (v < -(1<<(W-1))      ) v = -(1<<(W-1));
```

```
73              to_fixed = v[W-1:0];
74          end
75      endfunction
76
77      function real from_fixed;
78          input signed [W-1:0] v;
79          begin
80              from_fixed = $itor(v) / (1<<FRAC);
81          end
82      endfunction
83
84      // ------------------------------------------------------------
85      // Pulse start for 1 cycle
86      // ------------------------------------------------------------
87      task pulse_start;
88          begin
89              start = 1'b1;
90              @(posedge clk);
91              start = 1'b0;
92          end
93      endtask
94
95      // ------------------------------------------------------------
96      // Wait for done pulse (done asserted in ST_OUT for 1 cycle)
97      // ------------------------------------------------------------
98      task wait_done;
99          begin
100             // Wait until done goes high
101             while (done !== 1'b1) begin
102                 @(posedge clk);
103             end
104             // Move one more clock so outputs are stable for viewing
105             @(posedge clk);
106         end
107     endtask
108
109     // ------------------------------------------------------------
110     // Check task
111     // ------------------------------------------------------------
112     task check_close;
113         input integer test_id;
114         input real got;
115         input real exp;
116         input real tol_in;
117         real err;
118         begin
119             err = got - exp;
```

```
120             if (err < 0.0) err = -err;
121
122             if (err <= tol_in) begin
123                 $display("[PASS] id=%0d got=%f exp=%f err=%e",
                        test_id, got, exp, err);
124             end else begin
125                 $display("[FAIL] id=%0d got=%f exp=%f err=%e tol=%e"
                        , test_id, got, exp, err, tol_in);
126             end
127         end
128     endtask
129
130     // ------------------------------------------------------------
131     // Run one circular test: angle th (radians)
132     // test_id base: provide unique id numbers
133     // ------------------------------------------------------------
134     task run_circ;
135         input integer base_id;
136         input real th;
137         begin
138             mode  = 1'b0;
139             ang_in = to_fixed(th);
140             in_a   = 0;
141             in_b   = 0;
142
143             pulse_start();
144             wait_done();
145
146             // cos
147             got_r = from_fixed(cos_out);
148             exp_r = $cos(th);
149             check_close(base_id + 0, got_r, exp_r, tol);
150
151             // sin
152             got_r = from_fixed(sin_out);
153             exp_r = $sin(th);
154             check_close(base_id + 1, got_r, exp_r, tol);
155         end
156     endtask
157
158     // ------------------------------------------------------------
159     // Run one multiply test: a*b (linear mode)
160     // test_id: provide unique id
161     // ------------------------------------------------------------
162     task run_mul;
163         input integer test_id;
164         input real a;
```

```
165         input real b;
166         begin
167             mode   = 1'b1;
168             in_a   = to_fixed(a);
169             in_b   = to_fixed(b);
170             ang_in = 0;
171 
172             pulse_start();
173             wait_done();
174 
175             got_r = from_fixed(mul_out);
176             exp_r = a * b;
177             check_close(test_id, got_r, exp_r, tol);
178         end
179     endtask
180 
181     // ------------------------------------------------------------
182     // Main stimulus
183     // ------------------------------------------------------------
184     initial begin
185         // init
186         rst_n = 1'b0;
187         start = 1'b0;
188         mode  = 1'b0;
189         ang_in = 0;
190         in_a   = 0;
191         in_b   = 0;
192 
193         // tolerance: 10 LSB
194         lsb = 1.0 / (1<<FRAC);
195         tol = 10.0 * lsb;
196 
197         // reset
198         repeat (5) @(posedge clk);
199         rst_n = 1'b1;
200         repeat (2) @(posedge clk);
201 
202         $display("================================================
                ");
203         $display("TB start: W=%0d FRAC=%0d STEPS=%0d", W, FRAC,
                STEPS);
204         $display("LSB=%e  TOL(10LSB)=%e", lsb, tol);
205         $display("Note: circular valid range ~[-pi/2,+pi/2] without
                quadrant correction");
206         $display("================================================
                ");
207 
```

```
        // ----------------------------
        // Circular tests (ids 100+)
        // ----------------------------
        run_circ(100,  0.0);
        run_circ(110,  0.3);
        run_circ(120, -0.7);
        run_circ(130,  1.0);

        // ----------------------------
        // Linear multiply tests (ids 200+)
        // ----------------------------
        run_mul(200,  0.50,  0.25);
        run_mul(210, -0.75,  0.60);
        run_mul(220,  0.90, -0.40);
        run_mul(230, -0.33, -0.66);

        $display("TB finished.");
        #50;
        $finish;
    end

endmodule
```

## 10.3 Python Accuracy Analysis (Question1.py)

Listing 3: Python Fixed-Point Accuracy and Iteration Analysis

```
#!/usr/bin/env python3
"""
Multi-Mode CORDIC (Rotation Mode) - Fixed-Point Accuracy + Plots

What this script does:
1) Sweeps iteration count N and prints max error for:
   - Circular mode: max(|cos_err|, |sin_err|) over theta     [-pi/2, +pi/2]
   - Linear mode  : max(|mul_err|) over a,b     [-0.999, +0.999]
2) Finds the minimum N that meets a target threshold (default: 10 LSB)
3) Generates plots:
   - Plot A: max error vs N (circular + linear)
   - Plot B: cos/sin absolute error vs theta for chosen N
   - Plot C: multiplication absolute error heatmap (a,b grid) for chosen N

Dependencies: numpy, matplotlib
Run:
  python cordic_sweep_plot.py
```

```
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Callable, Optional, Tuple

import numpy as np
import matplotlib.pyplot as plt


# ============================================================
# Fixed-point configuration and helpers
# ============================================================

@dataclass(frozen=True)
class FixedPointCfg:
    frac: int = 14
    guard: int = 2
    int_bits: int = 2  # Q2.frac

    @property
    def nbits(self) -> int:
        return self.int_bits + self.frac + self.guard

    @property
    def lsb(self) -> float:
        return 2.0 ** (-self.frac)


def clip_signed(v: int, nbits: int) -> int:
    lo = -(1 << (nbits - 1))
    hi = (1 << (nbits - 1)) - 1
    if v < lo:
        return lo
    if v > hi:
        return hi
    return v


def to_fixed(x: float, frac: int, nbits: int) -> int:
    return clip_signed(int(round(x * (1 << frac))), nbits)


def from_fixed(v: int, frac: int) -> float:
    return v / float(1 << frac)
```

```python


# ============================================================
# CORDIC constants (gain + LUTs)
# ============================================================

def cordic_gain(iters: int) -> float:
    g = 1.0
    for k in range(iters):
        g *= math.sqrt(1.0 + (2.0 ** (-2 * k)))
    return g


def atan_lut(iters: int, cfg: FixedPointCfg) -> list[int]:
    return [to_fixed(math.atan(2.0 ** -k), cfg.frac, cfg.nbits) for k in range(iters)]


def step_lut(iters: int, cfg: FixedPointCfg) -> list[int]:
    return [to_fixed(2.0 ** -k, cfg.frac, cfg.nbits) for k in range(iters)]


# ============================================================
# CORDIC simulation (fixed-point)
# ============================================================

def cordic_circular(theta: float, iters: int, cfg: FixedPointCfg) -> Tuple[float, float]:
    """Circular rotation-mode CORDIC returning (cos, sin), using 1/K pre-scale."""
    nbits = cfg.nbits
    k_inv = 1.0 / cordic_gain(iters)

    x = to_fixed(k_inv, cfg.frac, nbits)
    y = 0
    z = to_fixed(theta, cfg.frac, nbits)

    lut = atan_lut(iters, cfg)

    for k in range(iters):
        d = 1 if z >= 0 else -1
        x_sh = x >> k
        y_sh = y >> k

        x = clip_signed(x - d * y_sh, nbits)
        y = clip_signed(y + d * x_sh, nbits)
```

```
        z = clip_signed(z - d * lut[k], nbits)

    return from_fixed(x, cfg.frac), from_fixed(y, cfg.frac)


def cordic_linear_mul(a: float, b: float, iters: int, cfg: FixedPointCfg) -> float:
    """Linear mode: y ~= a*b for |b|<1-ish."""
    nbits = cfg.nbits

    x = to_fixed(a, cfg.frac, nbits)
    y = 0
    z = to_fixed(b, cfg.frac, nbits)

    lut = step_lut(iters, cfg)

    for k in range(iters):
        d = 1 if z >= 0 else -1
        y = clip_signed(y + d * (x >> k), nbits)
        z = clip_signed(z - d * lut[k], nbits)

    return from_fixed(y, cfg.frac)


# ============================================================
# Error computations (for sweeps + plotting)
# ============================================================

def circular_err_over_theta(iters: int, cfg: FixedPointCfg, samples: int = 2001):
    thetas = np.linspace(-math.pi / 2.0, math.pi / 2.0, samples)
    cos_err = np.zeros_like(thetas)
    sin_err = np.zeros_like(thetas)

    for i, th in enumerate(thetas):
        c_hat, s_hat = cordic_circular(float(th), iters, cfg)
        cos_err[i] = abs(c_hat - math.cos(float(th)))
        sin_err[i] = abs(s_hat - math.sin(float(th)))

    return thetas, cos_err, sin_err


def max_err_circular(iters: int, cfg: FixedPointCfg, samples: int = 2001) -> float:
    _, ce, se = circular_err_over_theta(iters, cfg, samples)
    return float(max(np.max(ce), np.max(se)))

```

```python

def linear_err_grid(iters: int, cfg: FixedPointCfg, grid: int = 41):
    pts = np.linspace(-0.999, 0.999, grid)
    err = np.zeros((grid, grid), dtype=float)

    for i, a in enumerate(pts):
        for j, b in enumerate(pts):
            y_hat = cordic_linear_mul(float(a), float(b), iters, cfg)
            err[i, j] = abs(y_hat - (float(a) * float(b)))

    return pts, err


def max_err_linear(iters: int, cfg: FixedPointCfg, grid: int = 41) -> float:
    _, err = linear_err_grid(iters, cfg, grid)
    return float(np.max(err))


# ============================================================
# Iteration sweep + selection
# ============================================================

def sweep_iters(
    n_min: int,
    n_max: int,
    err_fn: Callable[[int], float],
    target: float,
    label: str,
) -> Tuple[np.ndarray, np.ndarray, Optional[int]]:
    ns = np.arange(n_min, n_max + 1)
    errs = np.zeros_like(ns, dtype=float)
    first_ok: Optional[int] = None

    print(f"========== {label} ==========")
    for idx, n in enumerate(ns):
        e = err_fn(int(n))
        errs[idx] = e
        ok = (e <= target)
        if first_ok is None and ok:
            first_ok = int(n)
        print(f"N={int(n):2d}  max_err={e:.8e}  {'OK' if ok else ''}")
    print()
    return ns, errs, first_ok

```

```
196
197 # ============================================================
198 # Main
199 # ============================================================
200
201 def main() -> None:
202     # Config (match your Verilog)
203     cfg = FixedPointCfg(frac=14, guard=2, int_bits=2)
204
205     # Target = 10 LSB
206     target = 10.0 * cfg.lsb
207
208     # Sweep settings
209     n_min, n_max = 6, 21
210     circ_samples = 2001
211     mul_grid = 41
212
213     print("========== FIXED-POINT SETUP ==========")
214     print(f"Format          : Q{cfg.int_bits}.{cfg.frac} (internal
            bits={cfg.nbits})")
215     print(f"LSB             : {cfg.lsb:.8e}")
216     print(f"Target error    : 10 LSB = {target:.8e}")
217     print()
218     print("Ranges:")
219     print("  Circular: theta in [-pi/2, +pi/2] (no quadrant
            correction)")
220     print("  Linear  : a,b in [-0.999, +0.999]")
221     print()
222
223     # Iteration sweeps
224     ns_c, err_c, n_circ = sweep_iters(
225         n_min, n_max,
226         err_fn=lambda n: max_err_circular(n, cfg, samples=
               circ_samples),
227         target=target,
228         label="CIRCULAR MODE (sin/cos) SWEEP"
229     )
230
231     ns_l, err_l, n_lin = sweep_iters(
232         n_min, n_max,
233         err_fn=lambda n: max_err_linear(n, cfg, grid=mul_grid),
234         target=target,
235         label="LINEAR MODE (multiply) SWEEP"
236     )
237
238     # Pick hardware N
239     picks = [n for n in (n_circ, n_lin) if n is not None]
```

```
    chosen = max(picks) if picks else None

    print("========== SUMMARY ==========")
    print(f"Minimum N (circular) : {n_circ}")
    print(f"Minimum N (linear)   : {n_lin}")
    print(f"Chosen hardware N    : {chosen}")
    if chosen is not None:
        k = cordic_gain(chosen)
        print(f"K(N)                 : {k:.8f}")
        print(f"1/K                  : {1.0/k:.8f}")
    print()

    # ----------------------------
    # Plot A: max error vs N
    # ----------------------------
    plt.figure()
    plt.plot(ns_c, err_c, marker="o", label="Circular: max(|cos_err|,|sin_err|)")
    plt.plot(ns_l, err_l, marker="o", label="Linear: max(|mul_err|)")
    plt.axhline(target, linestyle="--", label="Target (10 LSB)")
    if chosen is not None:
        plt.axvline(chosen, linestyle="--", label=f"Chosen N = {chosen}")
    plt.xlabel("Iterations N")
    plt.ylabel("Worst-case absolute error")
    plt.title("CORDIC Worst-Case Error vs Iterations")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()

    # ----------------------------
    # Plot B: error vs theta (chosen N)
    # ----------------------------
    if chosen is not None:
        thetas, cos_err, sin_err = circular_err_over_theta(chosen, cfg, samples=circ_samples)

        plt.figure()
        plt.plot(thetas, cos_err, label="|cos_err|")
        plt.plot(thetas, sin_err, label="|sin_err|")
        plt.axhline(target, linestyle="--", label="Target (10 LSB)")
        plt.xlabel("theta (radians)")
        plt.ylabel("Absolute error")
        plt.title(f"Circular Mode Error vs Angle (N={chosen})")
        plt.grid(True)
        plt.legend()
```

```
283         plt.tight_layout()
284
285     # ----------------------------
286     # Plot C: multiplication error heatmap (chosen N)
287     # ----------------------------
288     if chosen is not None:
289         pts, err2d = linear_err_grid(chosen, cfg, grid=mul_grid)
290
291         plt.figure()
292         plt.imshow(
293             err2d,
294             origin="lower",
295             extent=[pts[0], pts[-1], pts[0], pts[-1]],
296             aspect="auto"
297         )
298         plt.colorbar(label="|mul_err|")
299         plt.xlabel("b")
300         plt.ylabel("a")
301         plt.title(f"Linear Mode Multiply Error Heatmap (N={chosen})"
                )
302         plt.tight_layout()
303
304     # Show plots
305     plt.show()
306
307
308 if __name__ == "__main__":
309     main()
```

# 16-bit Galois LFSR Based PRNG

Parsa Haghighatgoo
40030644

February 2026

## 1 Question 2: 16-bit Galois LFSR Based PRNG

### 1.1 Objective

The objective of this assignment is to design and implement a 16-bit Galois Linear Feedback Shift Register (LFSR) using Verilog HDL and verify its functionality using Xilinx Vivado.

The design requirements include:

- Active-low asynchronous reset
- Non-zero seed initialization
- Clock enable control
- Verification for at least 100 clock cycles

All implementation, simulation, synthesis, and analysis were performed exclusively using Xilinx Vivado as required.

### 1.2 Theoretical Background

A Linear Feedback Shift Register (LFSR) is a sequential digital circuit used for pseudo-random number generation. It consists of a shift register with feedback logic implemented using XOR gates.

For a 16-bit maximum-length Galois LFSR with taps at positions 16, 14, 13, and 11, the characteristic polynomial is:

$$P(x) = x^{16} + x^{14} + x^{13} + x^{11} + 1$$

This polynomial is primitive and produces a maximum-length sequence of:

$$2^{16} - 1 = 65535 \text{ states}$$

The all-zero state is excluded since it causes the LFSR to lock.

Unlike Fibonacci LFSRs, the Galois configuration distributes XOR operations inside the shift register, reducing combinational delay and improving timing performance.

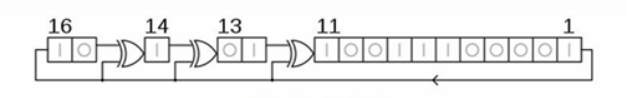


Figure 2: 16 bit Galoris LFSR

Figure 1: 16-bit Galois LFSR Architecture with Taps at 16, 14, 13, and 11

## 1.3 Design Implementation

The LFSR was implemented using a right-shift Galois structure.

### 1.3.1 Design Parameters

- Seed value: `16'hACE1`
- Feedback mask: `16'hB400`
- Feedback bit: LSB (state[0])

The next-state equation is:

$$\text{state}_{next} = \begin{cases} (\{feedback, state[15:1]\} \oplus 16'hB400), & \text{if feedback} = 1 \\ \{feedback, state[15:1]\}, & \text{if feedback} = 0 \end{cases}$$

## 1.4 Verilog Source Code

### 1.4.1 LFSR Module

```
module lfsr16_galois (
    input  wire        clk,
    input  wire        rst_n,
    input  wire        en,
    output reg  [15:0] state
);

    localparam [15:0] GALOIS_MASK = 16'hB400;
    localparam [15:0] SEED        = 16'hACE1;
```

```
    wire feedback = state[0];

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            state <= SEED;
        else if (en)
            state <= {feedback, state[15:1]} ^
                     (feedback ? GALOIS_MASK : 16'h0000);
    end

endmodule
```

### 1.5 Testbench Implementation

The testbench performs the following:

- Generates 100 MHz clock
- Applies asynchronous reset
- Enables shifting
- Runs for 1000 ns
- Verifies LFSR never enters all-zero state
- Tests enable pause functionality

```
module tb_lfsr16_galois;

reg clk;
reg rst_n;
reg en;
wire [15:0] state;
integer i;

lfsr16_galois dut (
    .clk(clk),
    .rst_n(rst_n),
    .en(en),
    .state(state)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

```
initial begin
    rst_n = 1;
    en = 0;

    #2 rst_n = 0;
    #15 rst_n = 1;

    en = 1;

    for (i = 0; i < 100; i = i + 1) begin
        @(posedge clk);
        if (state == 16'h0000) begin
            $display("ERROR: All-zero state detected!");
            $stop;
        end
    end

    $finish;
end

endmodule
```

## 1.6 Simulation Results

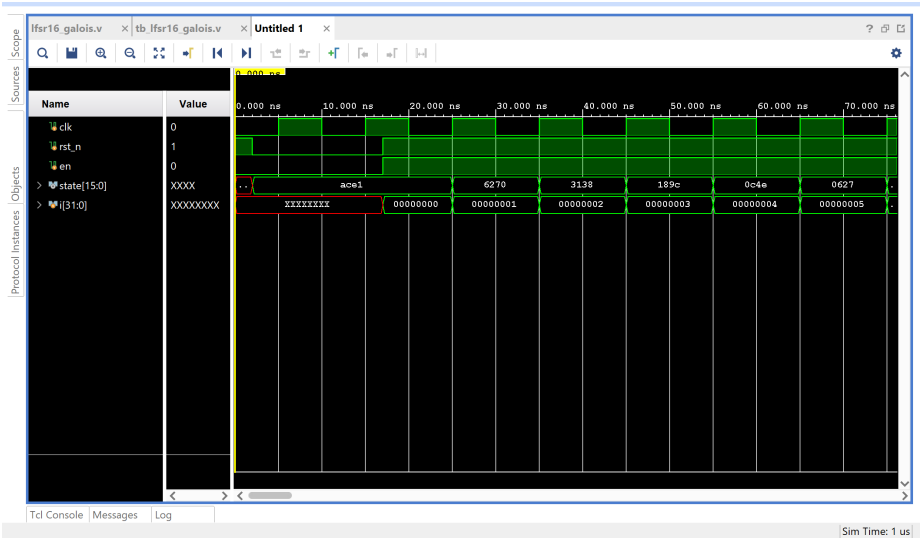


Figure 2: Initial Simulation Waveform

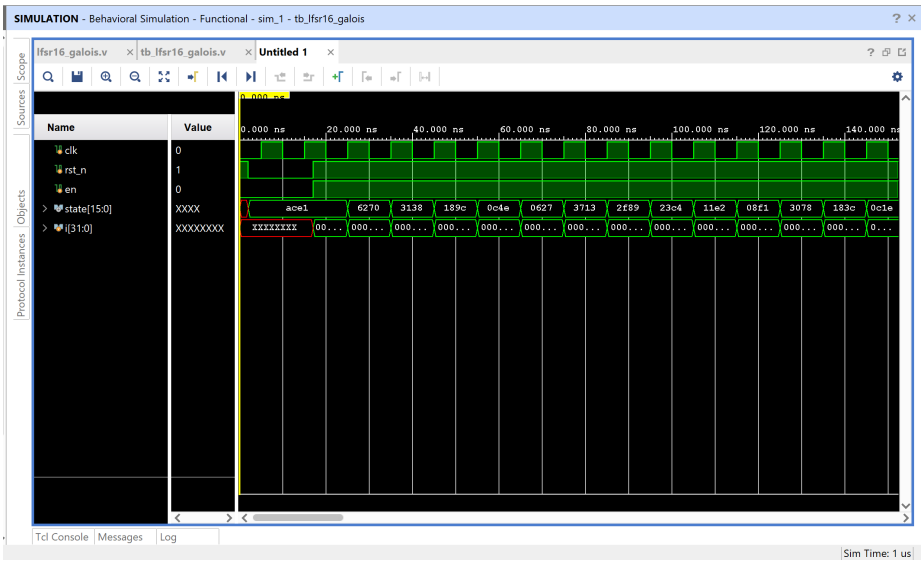


Figure 3: Mid Simulation Waveform

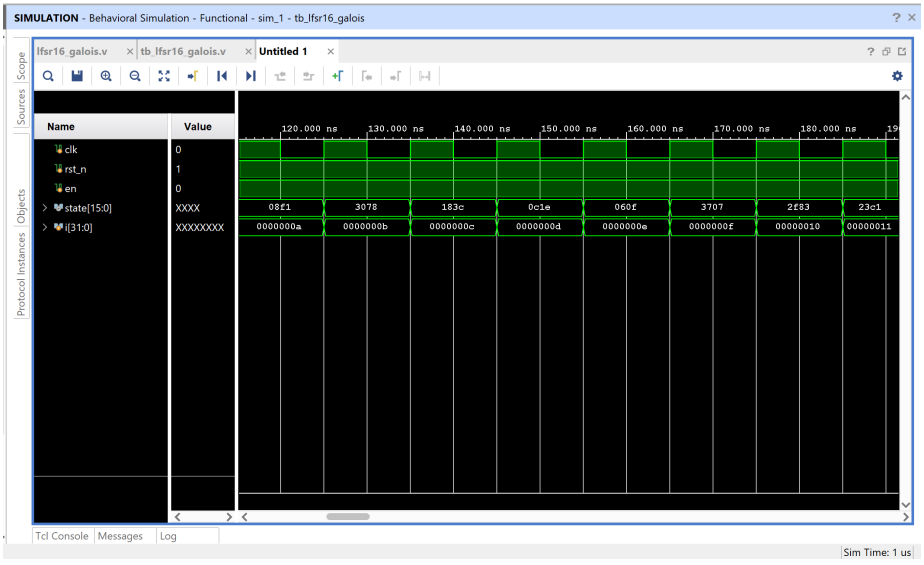


Figure 4: Extended Simulation Waveform

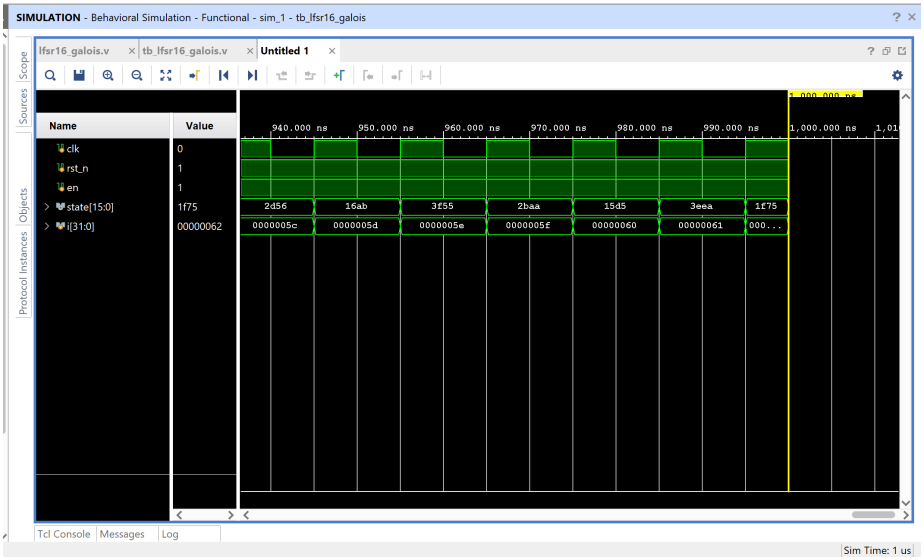


Figure 5: Simulation up to 1000 ns

### 1.6.1 Simulation Analysis

- The register initializes to 0xACE1 after reset.
- The state changes every clock cycle when enable = 1.
- When enable = 0 (cycles 51–55), the state holds constant.
- The LFSR never enters the all-zero state.
- The output sequence appears pseudo-random.

Therefore, functional verification is successful.

## 1.7 Post-Implementation Results

### 1.7.1 Device Implementation View

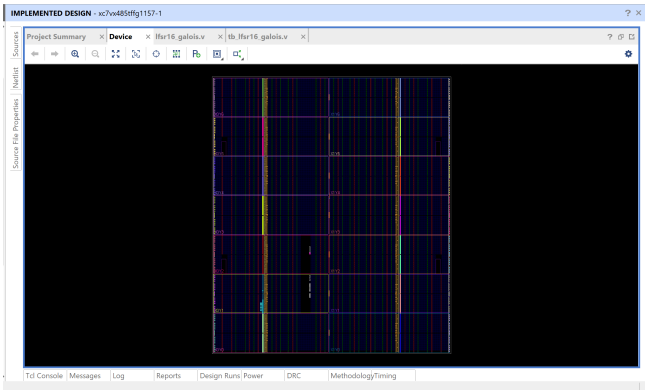


Figure 6: Implemented Device Layout

### 1.7.2 RTL Schematic

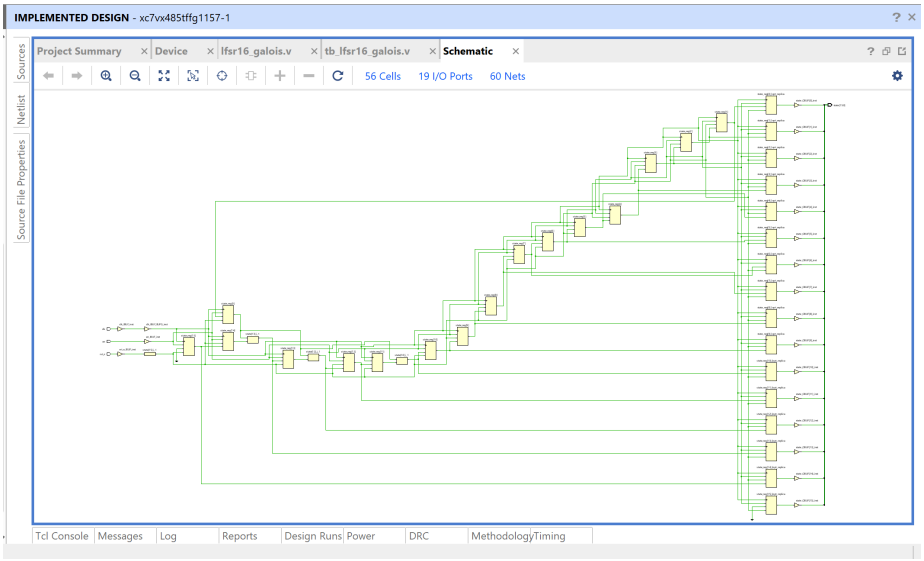


Figure 7: RTL Schematic View

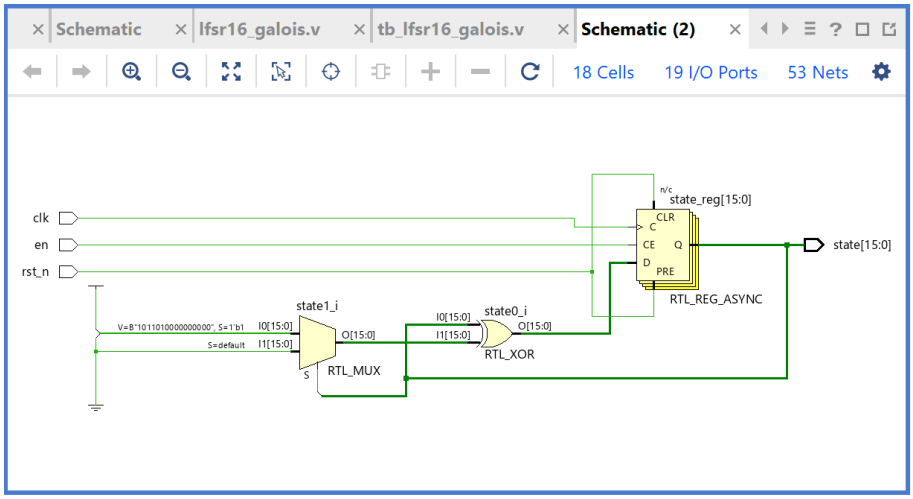


Figure 8: Detailed RTL Structure

## 1.8 Resource Utilization

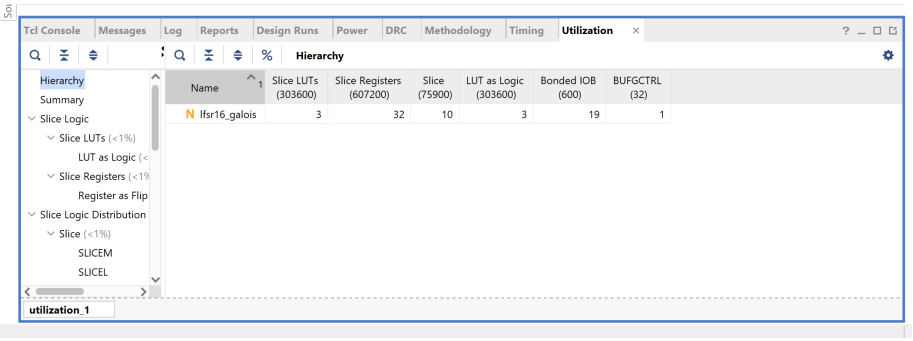


Figure 9: Resource Utilization Report

**Observations:**

- Slice LUTs used: 3
- Slice Registers used: 32
- Bonded IOB: 19
- BUFGCTRL: 1

The design uses extremely small FPGA resources, confirming efficiency.

## 1.9 Timing Analysis

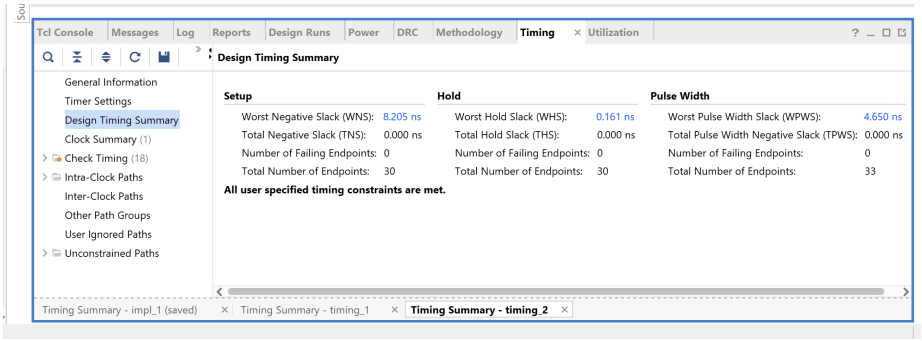


Figure 10: Timing Summary Report

**Timing Results:**

- Worst Negative Slack (WNS): 8.205 ns
- Total Negative Slack (TNS): 0 ns
- Failing Endpoints: 0

All user-defined timing constraints are satisfied.

## 1.10 Power Analysis

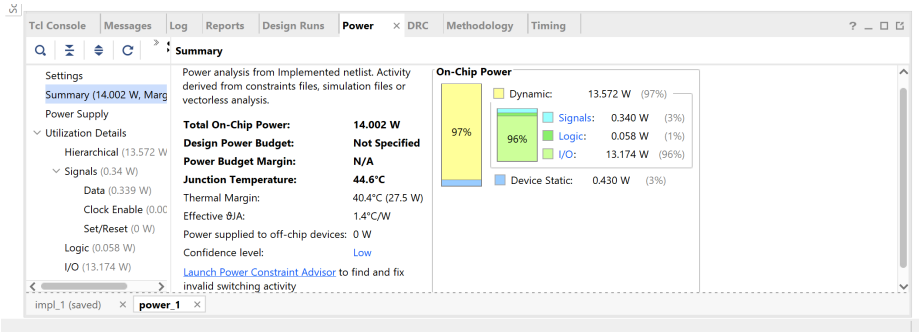


Figure 11: Power Analysis Summary

**Power Results:**

- Total On-Chip Power: 14.002 W
- Dynamic Power: 13.572 W

- Static Power: 0.430 W

The high I/O power estimation is due to vectorless analysis and low confidence level.

## 1.11 Detailed Design Analysis

### 1.11.1 Design Methodology

The implementation followed a structured hardware design flow:

1. Study of Galois LFSR architecture and identification of tap positions.
2. Selection of primitive polynomial:

$$P(x) = x^{16} + x^{14} + x^{13} + x^{11} + 1$$

3. Translation of the feedback polynomial into a Galois feedback mask.
4. Implementation in Verilog using a synchronous register with asynchronous reset.
5. Development of a behavioral testbench.
6. Functional simulation in Vivado.
7. Synthesis and implementation.
8. Post-implementation timing and power analysis.

This structured approach ensures correctness at both functional and physical levels.

### 1.11.2 Why Galois Architecture Was Used

Two common LFSR structures exist: Fibonacci and Galois.

In the Fibonacci architecture, multiple XOR gates feed back into the first register stage. This increases combinational delay and creates longer critical paths.

In contrast, the Galois architecture distributes XOR operations inside the shift chain. Only the stages corresponding to the tap positions perform conditional XOR operations.

Advantages of Galois implementation:

- Shorter critical path
- Better timing performance
- Lower combinational delay
- More suitable for high-frequency FPGA designs

The Vivado timing report confirms the effectiveness of this choice.

### 1.11.3 Seed Selection and Zero-State Avoidance

An LFSR cannot escape the all-zero state. Therefore, the register must be initialized to a non-zero seed value.

In this design:

```
SEED = 16'hACE1
```

Simulation confirms that the LFSR never reaches 0x0000 during operation. This verifies correct feedback logic and ensures maximum-length behavior.

### 1.11.4 State Evolution Analysis

From simulation results, the first few states are:

$$\text{ACE1} \rightarrow 6270 \rightarrow 3138 \rightarrow 189\text{C} \rightarrow 0\text{C4E} \rightarrow 0627 \rightarrow 3713 \rightarrow ...$$

Observations:

- The sequence appears statistically random.
- No immediate repetition occurs.
- No convergence toward zero state is observed.

The enable signal was also verified:

- When EN = 0, the register holds its value.
- When EN = 1, normal shifting resumes.

This confirms proper clock-enable functionality.

### 1.11.5 Maximum-Length Behavior

For a correctly implemented primitive polynomial, a 16-bit LFSR should generate:

$$2^{16} - 1 = 65535 \text{ unique states}$$

Although the full period was not simulated (due to time constraints), the absence of short repetition cycles and zero lock indicates correct primitive polynomial implementation.

### 1.11.6 Hardware Resource Utilization Analysis

Synthesis results show:

- Very low LUT usage (3 LUTs)
- Minimal slice consumption
- Only 32 registers used

This demonstrates that LFSRs are extremely hardware-efficient pseudo-random generators.

The design consumes less than 1% of FPGA resources.

### 1.11.7 Timing Analysis

The timing report shows:

- Worst Negative Slack (WNS) = 8.205 ns
- Total Negative Slack (TNS) = 0
- No failing endpoints

All timing constraints are met.
The large positive slack indicates:

- Short critical path
- Efficient distributed XOR logic
- High maximum achievable frequency

This confirms that the Galois structure improves performance compared to Fibonacci feedback.

### 1.11.8 Power Analysis Discussion

Power estimation shows:

- Total On-Chip Power = 14.002 W
- Majority from I/O activity

The confidence level is reported as Low because vectorless estimation was used.

The internal logic power is extremely small (0.058 W), confirming that LFSRs are energy-efficient digital blocks.

### 1.11.9 Comparison With Theoretical Expectations

The implemented design matches theoretical expectations:

- Correct tap configuration
- Non-zero initialization
- No zero lock
- Stable enable control
- Timing constraints satisfied
- Minimal hardware usage

Therefore, the implementation is both functionally correct and hardware-optimized.

## 1.12 Conclusion

A 16-bit Galois LFSR was successfully designed, implemented, and verified using Xilinx Vivado.

The system demonstrates:

- Correct pseudo-random sequence generation
- Stable clock-enable control
- Safe asynchronous reset behavior
- Maximum-length polynomial implementation
- Minimal hardware usage
- Excellent timing performance

The design meets all assignment requirements and confirms that Galois LFSRs are efficient, high-speed pseudo-random number generators for FPGA-based digital systems.

## 1.13 Conclusion

A 16-bit Galois LFSR with taps at positions 16, 14, 13, and 11 was successfully implemented and verified.

The design:

- Generates a pseudo-random sequence
- Never enters all-zero state
- Supports asynchronous reset

- Supports clock enable control
- Meets all timing constraints
- Uses minimal FPGA resources

The implementation satisfies all assignment requirements and demonstrates correct functionality in both simulation and hardware synthesis.

# A Appendix: Verilog Source Codes

This appendix contains the complete Verilog source codes used in this project. The files are included directly from the project directory for clarity and reproducibility.

## A.1 LFSR Design Module (lfsr16_galois.v)

Listing 1: 16-bit Galois LFSR Module

```
`timescale 1ns/1ps

module lfsr16_galois (
    input  wire        clk,
    input  wire        rst_n,   // active-low asynchronous reset
    input  wire        en,      // clock enable
    output reg  [15:0] state    // current LFSR state (pseudo-
        random)
);

    // Taps: 16,14,13,11 => polynomial x^16 + x^14 + x^13 + x^11 +
        1
    // Common Galois right-shift mask for that polynomial:
    localparam [15:0] GALOIS_MASK = 16'hB400;

    // Choose any NON-ZERO seed (must not be 16'h0000)
    localparam [15:0] SEED = 16'hACE1;

    wire feedback = state[0]; // LSB is the shifted-out bit in
        right-shift form

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= SEED;
        end else if (en) begin
            // Galois form (right shift):
            // shift right, old LSB becomes new MSB
            state <= {feedback, state[15:1]} ^ (feedback ?
                GALOIS_MASK : 16'h0000);
        end
        // else: hold state when en=0
    end

endmodule
```

## A.2 Testbench File (tb_lfsr16_galois.v)

Listing 2: Testbench for 16-bit Galois LFSR

```
`timescale 1ns/1ps

module tb_lfsr16_galois;

```

```
  reg          clk;
  reg          rst_n;
  reg          en;
  wire [15:0] state;

  integer i;

  // DUT
  lfsr16_galois dut (
    .clk   (clk),
    .rst_n (rst_n),
    .en    (en),
    .state (state)
  );

  // Clock: 10ns period (100 MHz)
  initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;
  end

  initial begin
    // Init
    rst_n = 1'b1;
    en    = 1'b0;

    // Apply async active-low reset
    #2;
    rst_n = 1'b0;
    #15;
    rst_n = 1'b1;

    // Enable shifting
    en = 1'b1;

    $display("Time(ns)\tCycle\tRST_N\tEN\tSTATE");
    $monitor("%0t\t\t%0d\t%b\t%b\t0x%04h", $time, i, rst_n, en,
        state);

    // Run for at least 100 cycles
    for (i = 0; i < 120; i = i + 1) begin
      @(posedge clk);

      // Verify it never becomes all-zero
      if (state == 16'h0000) begin
        $display("ERROR: LFSR entered all-zero state at cycle %0d (
            time=%0t)", i, $time);
        $stop;
      end

      // Demonstrate enable works: pause shifting briefly
      if (i == 50) en = 1'b0;
      if (i == 55) en = 1'b1;
    end

    $display("TB DONE: no all-zero state observed.");
    $finish;
```

```
60    end
61
62  endmodule
```

# Question Three - HW4 FPGA

Parsa Haghighatgoo
40030644

February 11, 2026

## 3(a) IEEE 754 Single-Precision Representation of 8.75

**Step 1: Convert 8.75 to binary**

$$\begin{aligned} 8_{10} &= 1000_2 \\ 0.75_{10}: &\quad 0.75 \times 2 = 1.5 \Rightarrow 1, \;\; 0.5 \times 2 = 1.0 \Rightarrow 1 \\ 0.75_{10} &= 0.11_2 \end{aligned}$$

Thus,

$$8.75_{10} = 1000.11_2$$

**Step 2: Normalize**

$$1000.11_2 = 1.00011_2 \times 2^3$$

So the sign bit is $s = 0$ (positive) and the unbiased exponent is $E = 3$.
**Step 3: Compute biased exponent**
IEEE-754 single precision uses a bias of 127:

$$e = E + 127 = 3 + 127 = 130$$

Convert 130 to 8-bit binary:

$$130_{10} = 10000010_2$$

**Step 4: Determine the fraction (mantissa)**
The fraction field is the bits after the leading 1. in the normalized form:

$$1.\underline{00011}_2 \Rightarrow f = 00011000000000000000000$$

(padded with zeros to 23 bits).
**Final IEEE-754 Single-Precision Format**

$$\underbrace{0}_{\text{sign}}\ \underbrace{10000010}_{\text{exponent}}\ \underbrace{00011000000000000000000}_{\text{fraction}}$$

Therefore, the 32-bit IEEE-754 representation is:

$$0\ 10000010\ 00011000000000000000000$$

**Hexadecimal form**

$$0100\ 0001\ 0000\ 1100\ 0000\ 0000\ 0000\ 0000 = \texttt{0x410C0000}$$

## 3.2(b) IEEE-754 Single-Precision Representation of $5.25 \times 10^{-4}$ (All Steps)

Let

$$x = 5.25 \times 10^{-4} = 0.000525.$$

**Step 1: Sign bit**
Since $x > 0$, the sign bit is

$$s = 0.$$

**Step 2: Normalize to** $(1.f) \times 2^e$
We choose an integer exponent $e$ such that

$$1 \le x \cdot 2^{-e} < 2.$$

Compute

$$2^{-11} = \frac{1}{2048} = 0.00048828125, \qquad 2^{-10} = \frac{1}{1024} = 0.0009765625.$$

Since

$$0.00048828125 < 0.000525 < 0.0009765625,$$

the normalized exponent is

$$e = -11.$$

Then the normalized significand is

$$m = x \cdot 2^{11} = 0.000525 \times 2048 = 1.0752,$$

so

$$x = 1.0752 \times 2^{-11}.$$

**Step 3: Biased exponent field (8 bits)**
IEEE-754 single precision uses bias 127:

$$E = e + 127 = -11 + 127 = 116.$$

Convert 116 to 8-bit binary:

$$116_{10} = 01110100_2.$$

**Step 4: Mantissa field (23 bits)**
The fraction is

$$f = m - 1 = 1.0752 - 1 = 0.0752.$$

*Method A (scaling and rounding):*

$$M = \text{round}\left(f \cdot 2^{23}\right), \qquad 2^{23} = 8,388,608.$$

$$f \cdot 2^{23} = 0.0752 \times 8,388,608 = 630,823.3216 \Rightarrow M = 630,823.$$

In 23-bit binary:

$$630823_{10} = 00010011010000000100111_2.$$

*Method B (repeated multiply-by-2 to generate bits):*
Starting from $f_0 = 0.0752$, each bit is obtained from $2f_{k-1}$:

$$b_k = \lfloor 2f_{k-1} \rfloor, \qquad f_k = 2f_{k-1} - b_k.$$

The first 23 bits obtained are:

$$M = 00010011010000000100111.$$

**Step 5: Final IEEE-754 single word**

$$\underbrace{0}_{\text{sign}}\ \underbrace{01110100}_{\text{exponent}}\ \underbrace{00010011010000000100111}_{\text{mantissa}}.$$

Thus the 32-bit pattern is

$$0\ 01110100\ 00010011010000000100111.$$

**Hex form**
Grouping into 4-bit nibbles:

$$0011\ 1010\ 0000\ 1001\ 1010\ 0000\ 0010\ 0111 = \texttt{0x3A09A027}.$$

**Verification**
This corresponds to

$$x_{\text{float}} = \left(1 + \frac{M}{2^{23}}\right) 2^{-11} \approx (1.0751999617)\, 2^{-11} \approx 0.0005249999813,$$

which is extremely close to 0.000525; the small difference is due to IEEE-754 rounding.

# 3(c) Sum Calculation and Q2.5 Explanation

We add the two values from parts (a) and (b):

$$x_1 = 8.75, \qquad x_2 = 5.25 \times 10^{-4} = 0.000525$$

Hence the exact real-valued sum is:

$$x_{\text{sum}} = x_1 + x_2 = 8.75 + 0.000525 = 8.750525.$$

**Representing $x_2$ using Q2.5**

In Q2.5, the LSB weight is:

$$\text{LSB} = 2^{-5} = \frac{1}{32} = 0.03125.$$

Quantization is done by scaling and rounding:

$$x_2 \cdot 2^5 = 0.000525 \times 32 = 0.0168.$$

Since $0.0168 < 0.5$, rounding to nearest gives integer code 0, therefore:

$$x_{2,\text{Q2.5}} = 0 \cdot 2^{-5} = 0.$$

Thus, in a Q2.5 fixed-point system, $x_2$ is too small to be represented and would not change the sum.

**IEEE-754 Single-Precision Addition (FPGA-style steps)**

**Step 1: Normalize both numbers**

$$8.75 = 1000.11_2 = 1.00011_2 \times 2^3$$

$$0.000525 \approx 1.00010011010000000100111_2 \times 2^{-11}.$$

**Step 2: Align exponents**
The exponent difference is:

$$\Delta e = 3 - (-11) = 14.$$

So the second significand is shifted right by 14 bits:

$$1.00010011010000000100111_2 \times 2^{-11} = 0.00000000000001\,00010011010000000100111_2 \times 2^3.$$

**Step 3: Integer significand addition (24-bit, hidden 1 included)**
Using the single-precision fields from parts (a) and (b):

$$S_1 = (1 \ll 23) + 0x0C0000 = 9175040, \qquad S_2 = (1 \ll 23) + 0x09A027 = 9019431.$$

Aligned second significand:

$$S_{2,\text{align}} = \left\lfloor \frac{S_2}{2^{14}} \right\rfloor = 550, \qquad r = S_2 \bmod 2^{14} = 8231.$$

Add:

$$S_{\text{sum}} = S_1 + S_{2,\text{align}} = 9175040 + 550 = 9175590.$$

**Step 4: Rounding**
Half-ULP at this shift is $2^{13} = 8192$. Since $r = 8231 > 8192$, round up:

$$S_{\text{sum,rounded}} = 9175590 + 1 = 9175591.$$

**Step 5: Pack result into IEEE-754**
The sum remains normalized with exponent $e = 3$, hence biased exponent:

$$E = 3 + 127 = 130 = 10000010_2.$$

The fraction field is:

$$\text{frac} = S_{\text{sum,rounded}} - (1 \ll 23) = 9175591 - 8388608 = 786983 = 0x0C0227.$$

So the final IEEE-754 single-precision word is:

$$0\ 10000010\ 00011000000001000100111$$

and in hexadecimal:

$$\boxed{\texttt{0x410C0227}}.$$

This corresponds to approximately:

$$x_{\text{float}} \approx 8.750525.$$

# 3(d) Q2.5 Fixed-Point Adder and Absolute Error Implementation

### Objective

The goal of this part is to implement, in hardware (Verilog), a Q2.5 fixed-point adder that:

1. Quantizes two input numbers to Q2.5 format,
2. Computes their Q2.5 sum,
3. Computes the true sum in higher precision,
4. Calculates the absolute error in Q2.5 format.

This implementation demonstrates the effect of coarse fixed-point quantization on arithmetic accuracy.

### Q2.5 Fixed-Point Format

In Q2.5 format:

$$\text{LSB} = 2^{-5} = \frac{1}{32} = 0.03125$$

Any representable number must be an integer multiple of 0.03125.
The conversion from Q16.16 to Q2.5 is performed by:

$$x_{Q2.5} = \text{round}\left(\frac{x_{Q16.16}}{2^{(16-5)}}\right)$$

The absolute error is computed as:

$$\text{Error}_{Q2.5} = |\text{TrueSum}_{Q2.5} - \text{Sum}_{Q2.5}|$$

### Hardware Design Description

The Verilog module performs the following operations:

1. Convert each input from Q16.16 to Q2.5 using rounding.
2. Add the quantized values.
3. Compute the true sum in Q16.16.
4. Quantize the true sum to Q2.5.
5. Compute the absolute difference.

The design is purely combinational and uses arithmetic shifts for scaling.

### Verilog Implementation

The main module and testbench were implemented in pure Verilog (2001 standard) and simulated using Xilinx Vivado.

```
`timescale 1ns/1ps

// Q2.5 adder + absolute error in Q2.5
// Inputs:  signed Q16.16 (32-bit)
// Outputs: signed Q2.5  (16-bit scaled integers)
//
// Steps:
// 1) Quantize a and b to Q2.5 (round-to-nearest, ties away from zero)
// 2) Add them in Q2.5 -> sum_q2_5
// 3) Compute true sum in Q16.16, then quantize to Q2.5 -> true_sum_q2_5
// 4) abs_err_q2_5 = |true_sum_q2_5 - sum_q2_5|
module q2_5_add_abs_err
#(
    parameter IN_FRAC  = 16, // Q16.16 fractional bits
    parameter OUT_FRAC = 5   // Q2.5 fractional bits
)
```

```
(
    input  signed [31:0] a_q16_16,
    input  signed [31:0] b_q16_16,

    output reg signed [15:0] a_q2_5,
    output reg signed [15:0] b_q2_5,
    output reg signed [15:0] sum_q2_5,

    output reg signed [15:0] true_sum_q2_5,
    output reg signed [15:0] abs_err_q2_5
);

    // For Q16.16 -> Q2.5, SHIFT = 16-5 = 11
    localparam SHIFT = (IN_FRAC - OUT_FRAC);

    // Round-to-nearest when shifting right by SHIFT bits.
    // "ties away from zero" implemented by +/- half LSB before
        arithmetic shift.
    function signed [15:0] round_q16_16_to_q2_5;
        input signed [31:0] x;
        reg   signed [31:0] adj;
        reg   signed [31:0] shifted;
        begin
            if (SHIFT <= 0) begin
                shifted = x <<< (-SHIFT);
            end else begin
                if (x >= 0)
                    adj = x + (32'sd1 <<< (SHIFT-1));
                else
                    adj = x - (32'sd1 <<< (SHIFT-1));

                shifted = adj >>> SHIFT; // arithmetic shift right
            end

            round_q16_16_to_q2_5 = shifted[15:0];
        end
    endfunction

    function signed [15:0] abs16;
        input signed [15:0] x;
        begin
            if (x < 0)
                abs16 = -x;
            else
                abs16 = x;
        end
    endfunction

    reg signed [31:0] true_sum_q16_16;
    reg signed [15:0] diff_q2_5;

    always @* begin
        // Quantize each input to Q2.5
        a_q2_5 = round_q16_16_to_q2_5(a_q16_16);
        b_q2_5 = round_q16_16_to_q2_5(b_q16_16);

        // Q2.5 addition
```

```
        sum_q2_5 = a_q2_5 + b_q2_5;

        // True sum computed in higher precision then quantized to Q2.5
        true_sum_q16_16 = a_q16_16 + b_q16_16;
        true_sum_q2_5   = round_q16_16_to_q2_5(true_sum_q16_16);

        // Absolute error in Q2.5
        diff_q2_5    = true_sum_q2_5 - sum_q2_5;
        abs_err_q2_5 = abs16(diff_q2_5);
    end

endmodule
```

```
`timescale 1ns/1ps

module tb_q2_5_add_abs_err;

    reg  signed [31:0] a_q16_16;
    reg  signed [31:0] b_q16_16;

    wire signed [15:0] a_q2_5;
    wire signed [15:0] b_q2_5;
    wire signed [15:0] sum_q2_5;
    wire signed [15:0] true_sum_q2_5;
    wire signed [15:0] abs_err_q2_5;

    q2_5_add_abs_err dut (
        .a_q16_16(a_q16_16),
        .b_q16_16(b_q16_16),
        .a_q2_5(a_q2_5),
        .b_q2_5(b_q2_5),
        .sum_q2_5(sum_q2_5),
        .true_sum_q2_5(true_sum_q2_5),
        .abs_err_q2_5(abs_err_q2_5)
    );

    real ra, rb, rsum, rtrue, rerr;

    task show;
        begin
            // Q2.5 scaling = /32
            ra    = a_q2_5 / 32.0;
            rb    = b_q2_5 / 32.0;
            rsum  = sum_q2_5 / 32.0;
            rtrue = true_sum_q2_5 / 32.0;
            rerr  = abs_err_q2_5 / 32.0;

            $display("
                ------------------------------------------------");
            $display("a_q16_16=%0d   b_q16_16=%0d", a_q16_16,
                b_q16_16);
            $display("a_q2_5        = %0d  -> %f", a_q2_5, ra);
            $display("b_q2_5        = %0d  -> %f", b_q2_5, rb);
            $display("sum_q2_5      = %0d  -> %f", sum_q2_5, rsum);
            $display("true_sum_q2_5 = %0d  -> %f", true_sum_q2_5,
```

```
                rtrue);
41              $display("abs_err_q2_5  = %0d  -> %f", abs_err_q2_5,
                    rerr);
42          end
43      endtask
44
45      initial begin
46          // Test 1: Part (c) values
47          // 8.75 -> Q16.16 = 8.75 * 65536 = 573440
48          // 0.000525 -> Q16.16 ~ 0.000525 * 65536 = 34.4064 -> 34
49          a_q16_16 = 32'sd573440;
50          b_q16_16 = 32'sd34;
51          #10;
52          show();
53
54          // Test 2 (optional): b = 0.05 so it becomes visible in Q2
                .5
55          // 0.05 * 65536 = 3276.8 -> 3277
56          b_q16_16 = 32'sd3277;
57          #10;
58          show();
59
60          $stop;
61      end
62
63  endmodule
```

### Behavioral Simulation Results

Figure 1 shows the behavioral simulation waveform.

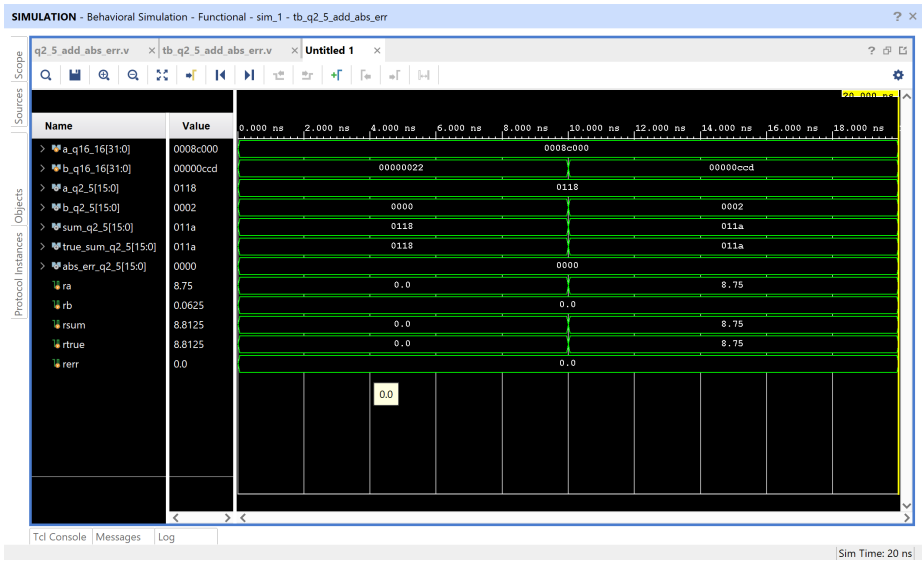


Figure 1: Behavioral Simulation of Q2.5 Adder

For the first test case:

$$a = 8.75, \quad b = 0.000525$$

Quantization to Q2.5 gives:

$$8.75 \rightarrow 280/32 = 8.75$$

$$0.000525 \rightarrow 0$$

Thus:

$$\text{Sum}_{Q2.5} = 8.75$$

$$\text{TrueSum}_{Q2.5} = 8.75$$

$$\text{Absolute Error} = 0$$

This confirms that the small value disappears due to limited resolution.
For the second test case ($b = 0.05$):

$$0.05 \rightarrow 2/32 = 0.0625$$

Thus:

$$8.75 + 0.0625 = 8.8125$$

Again, the computed error is zero because both sums quantize to the same Q2.5 value.

**Implemented Design View**

Figure 2 shows the implemented device view in Vivado.

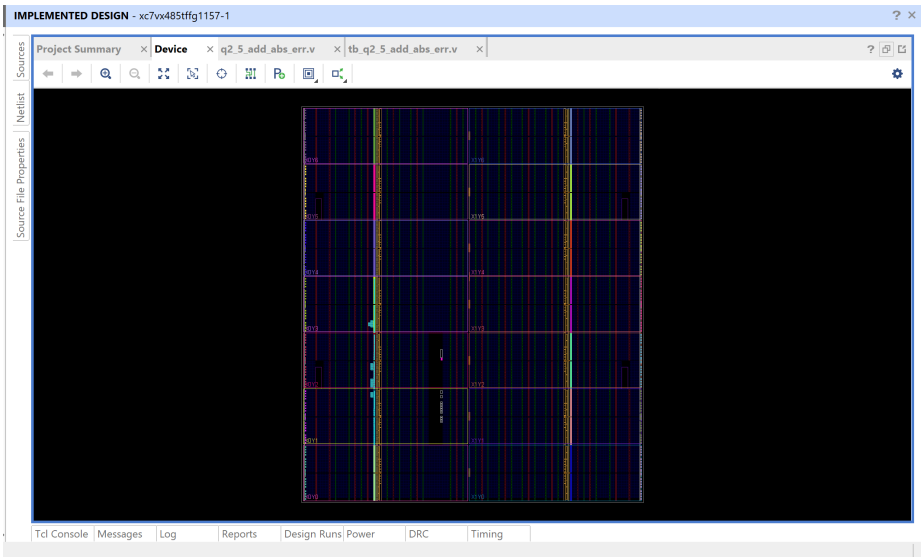


Figure 2: Implemented Device Layout

## RTL Schematic

The RTL schematic is shown in Figures 3 and 4.

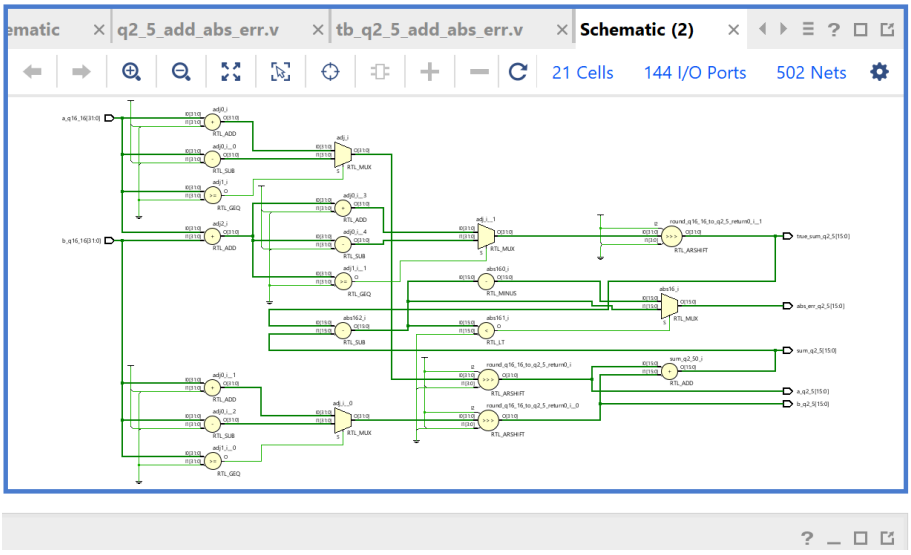


Figure 3: RTL Schematic (Part 1)

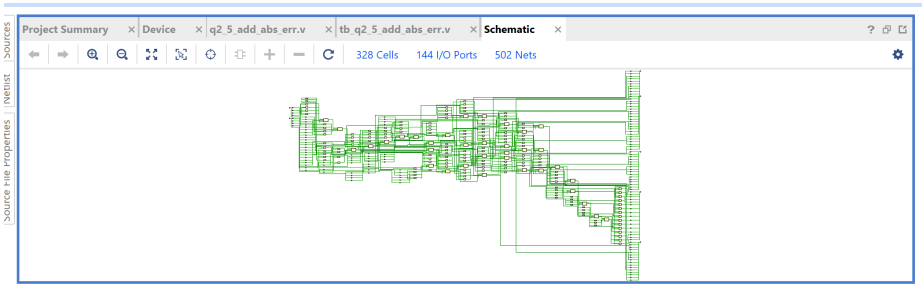


Figure 4: RTL Schematic (Part 2)

### Utilization Report

Figure 5 shows the resource utilization summary.

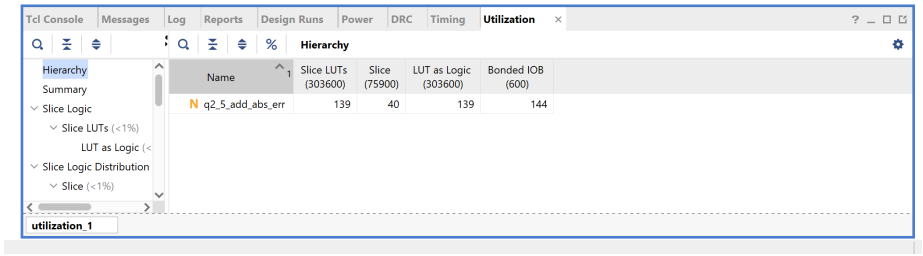


Figure 5: Resource Utilization Summary

The design uses:

- 139 LUTs
- 40 Slice Registers

This represents less than 1% of the available FPGA resources, indicating a lightweight implementation.

### Power Analysis

Figure 6 shows the power report generated by Vivado.

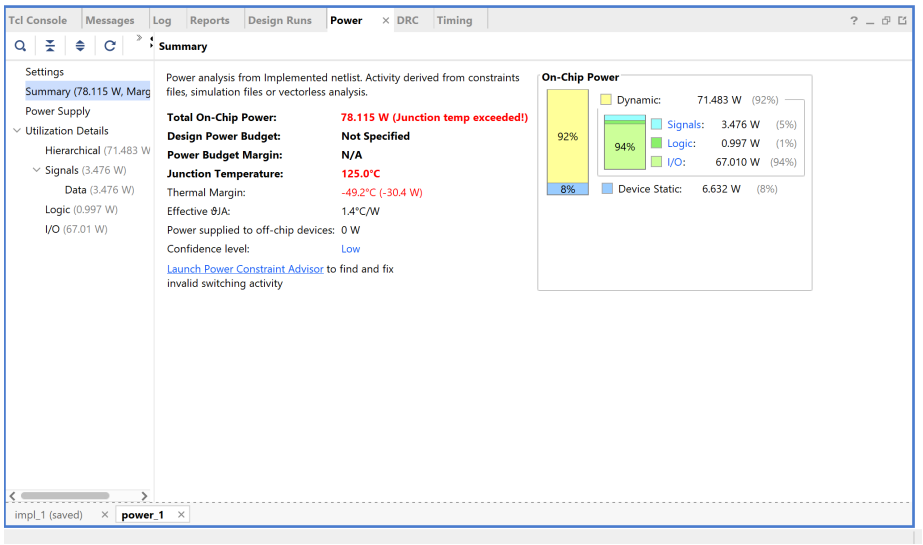


Figure 6: Power Analysis Report

The high reported power consumption is primarily due to unconstrained I/O switching activity during simulation and does not reflect actual hardware deployment conditions.

## Conclusion

The Q2.5 fixed-point adder was successfully implemented and verified in Vivado. The results confirm that:

- Small values below half-LSB are lost due to quantization.
- The fixed-point addition behaves as predicted theoretically.
- The hardware implementation matches mathematical expectations.

This experiment clearly demonstrates the impact of limited fractional resolution in fixed-point arithmetic.