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

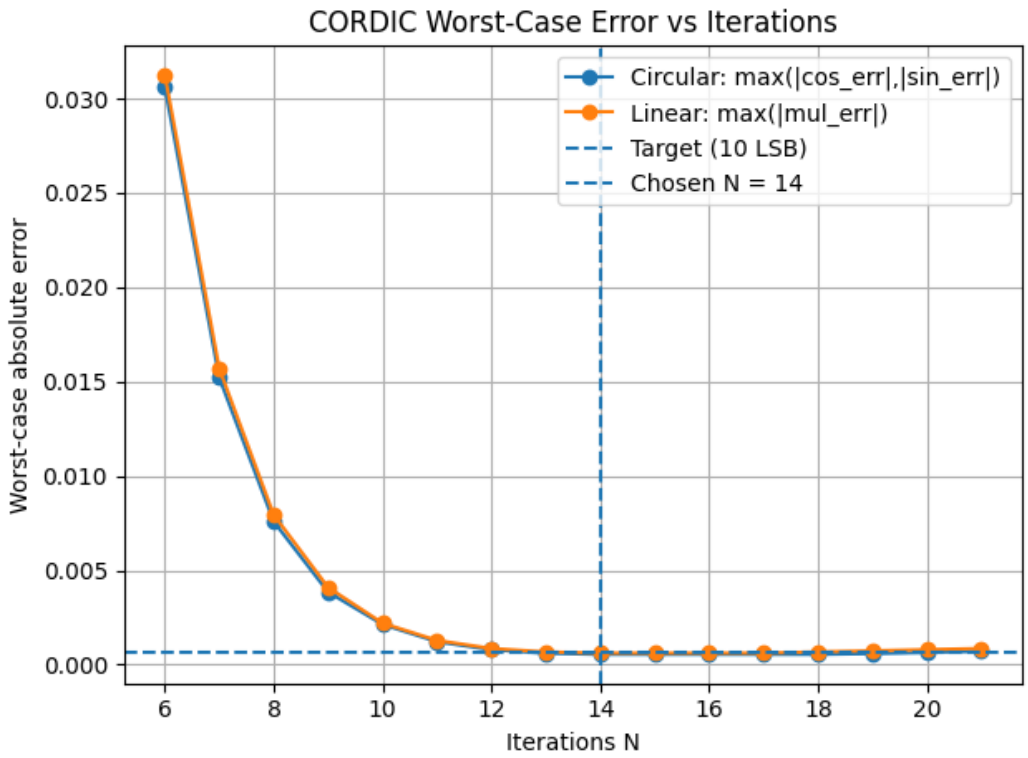

Figure 1: CORDIC Worst-Case Error vs Iterations

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

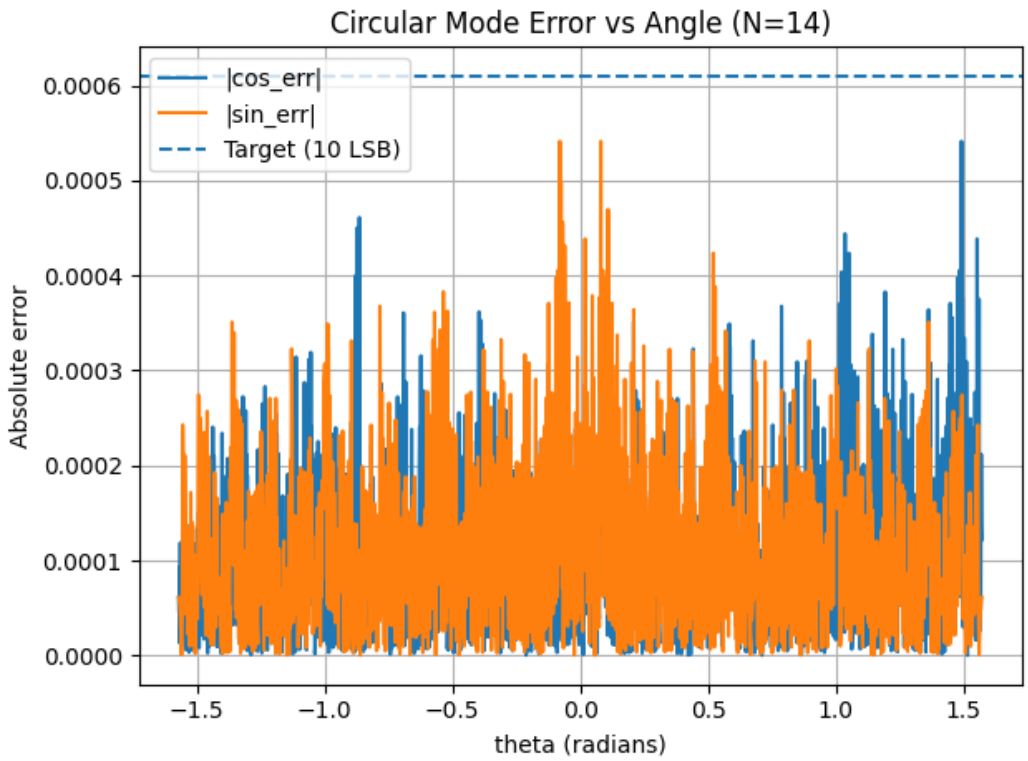

Figure 2: Circular Mode Error vs Angle (N=14)

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

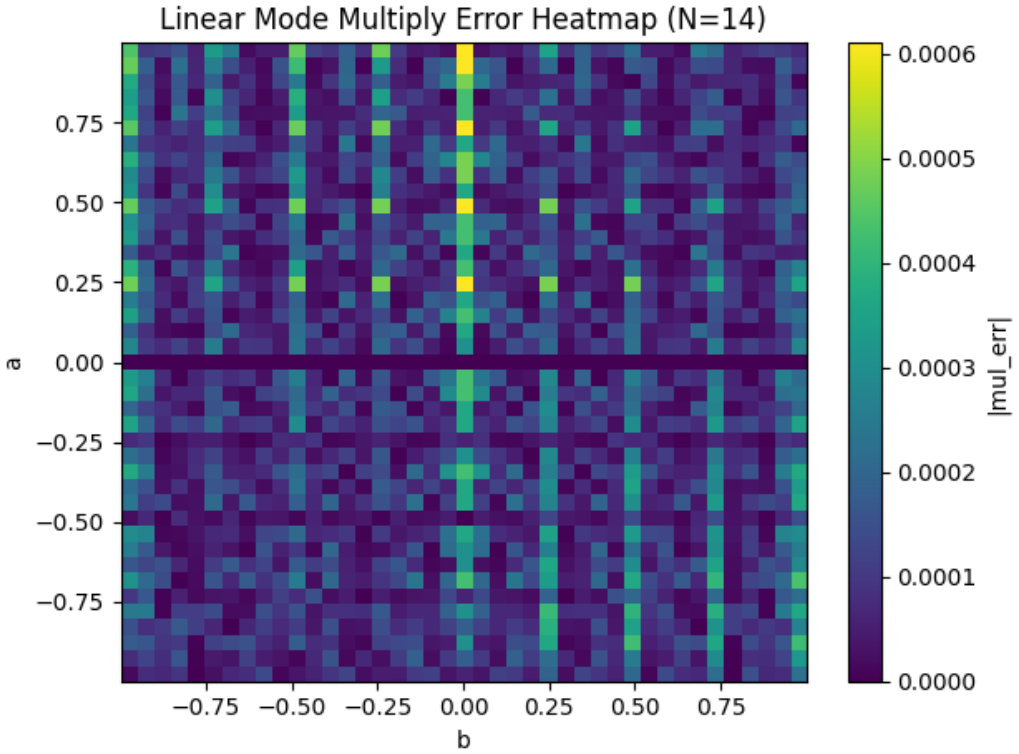

Figure 3: Linear Mode Multiplication Error Heatmap (N=14)

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

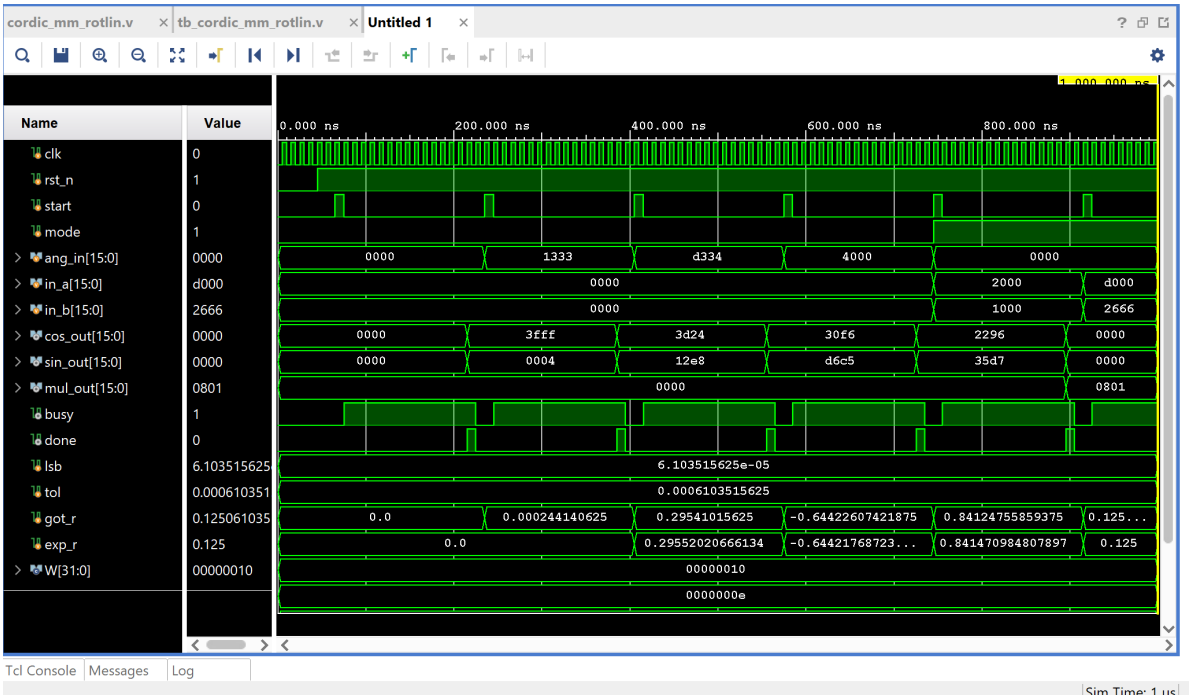

Figure 4: Behavioral Simulation Waveform

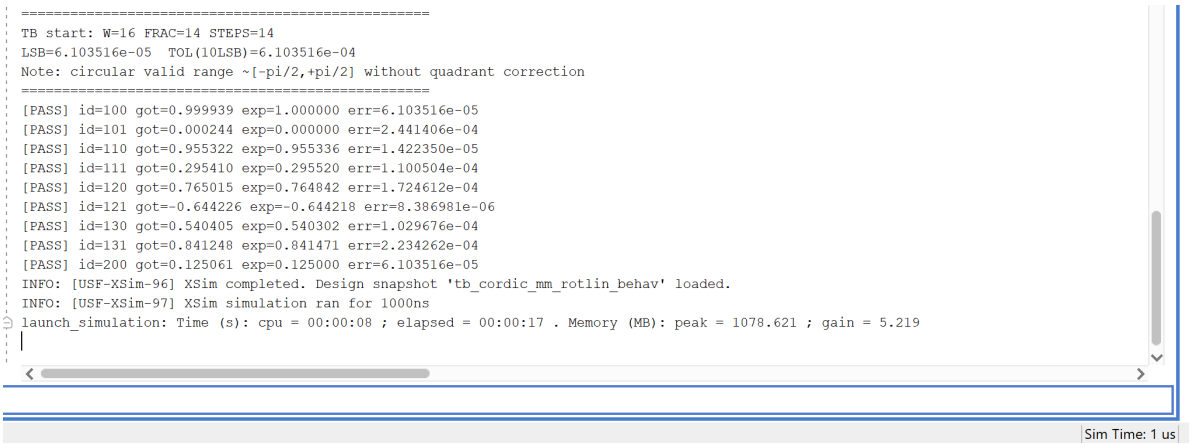

Figure 5: Simulation Console Output

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

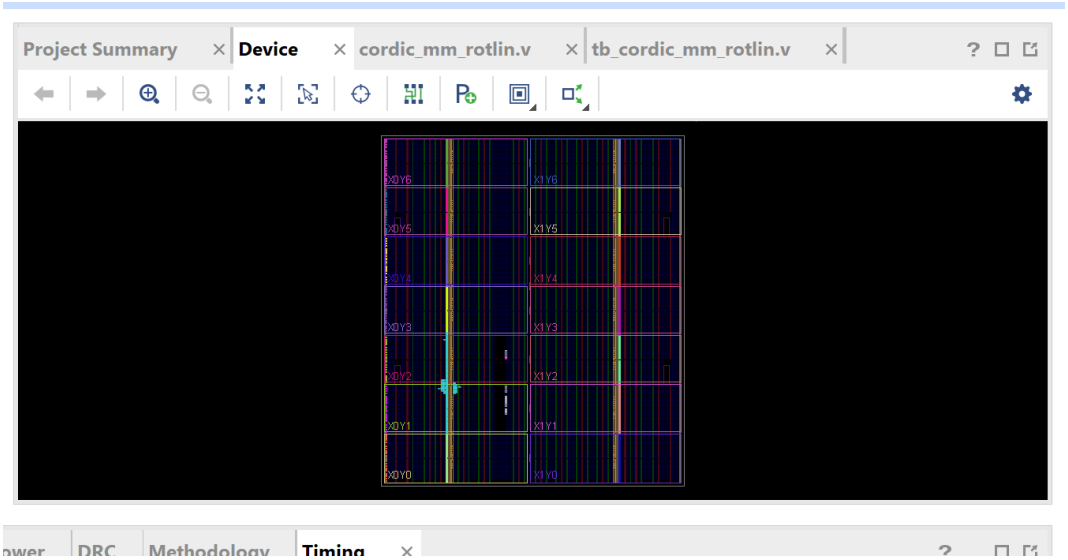

Figure 6: FPGA Device View

5.2 Schematic View

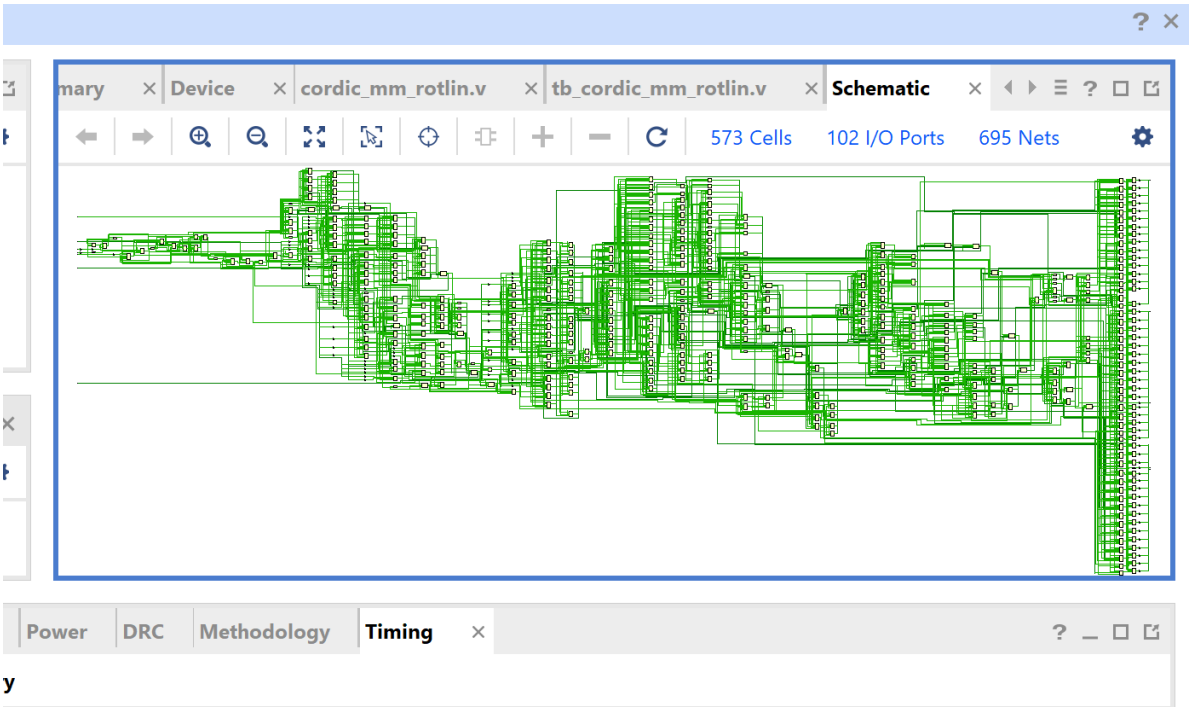

Figure 7: Post-Synthesis Schematic

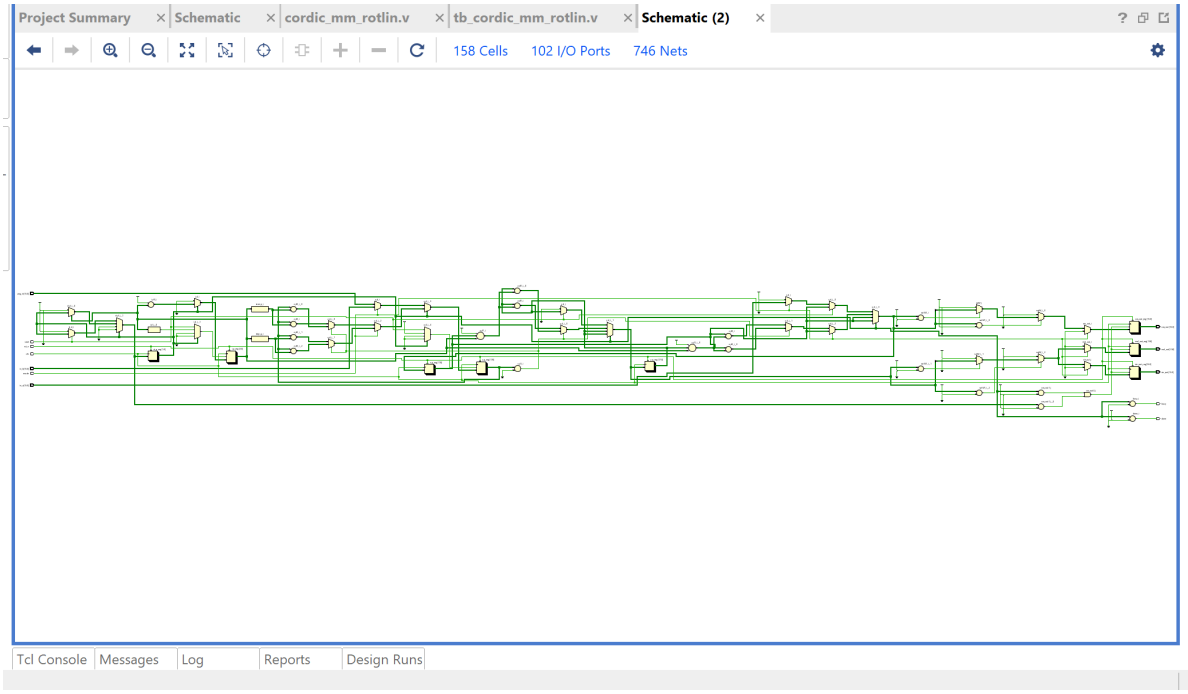

Figure 8: Detailed Schematic View

5.3 Resource Utilization

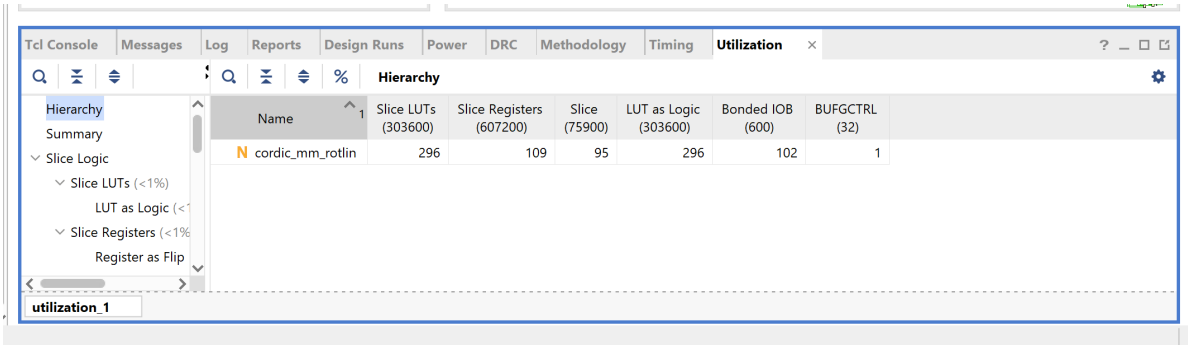

Figure 9: Resource Utilization Report

Key results:

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

5.4 Timing Analysis

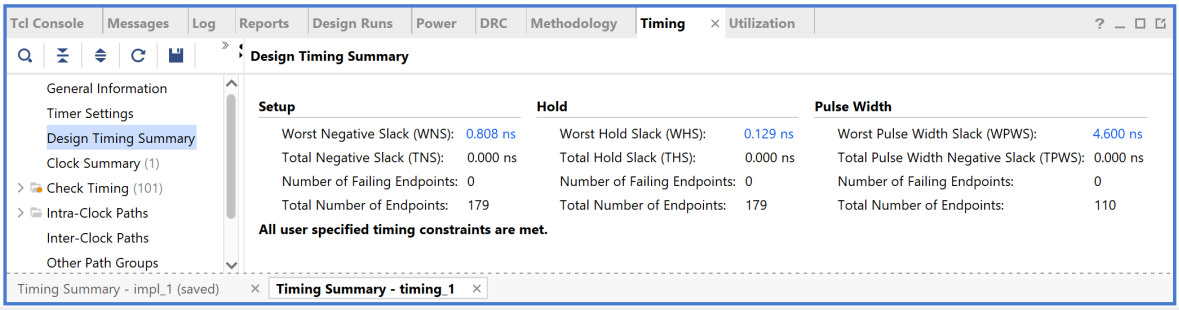

Figure 10: Timing Summary

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.

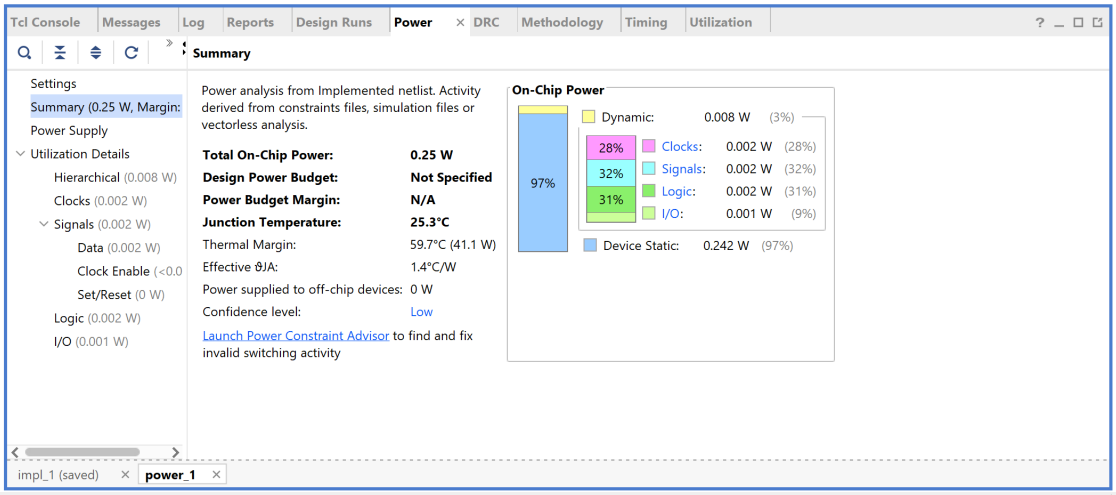

Figure 11: Vivado Power Report (power_1)

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	6.103516×10^{-5}	Yes
101	sin(0.0)	0.000000	0.000244	2.441406×10^{-4}	Yes
110	cos(0.3)	0.955336	0.955322	1.422350×10^{-5}	Yes
111	sin(0.3)	0.295520	0.295410	1.100504×10^{-4}	Yes
120	cos(-0.7)	0.764842	0.765015	1.724612×10^{-4}	Yes
121	sin(-0.7)	-0.644218	-0.644226	8.386981×10^{-6}	Yes
130	cos(1.0)	0.540302	0.540405	1.029676×10^{-4}	Yes
131	sin(1.0)	0.841471	0.841248	2.234262×10^{-4}	Yes
200	0.5×0.25	0.125000	0.125061	6.103516×10^{-5}	Yes

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

Parameter	Value
Fixed-point format	Q2.14
LSB (2^{-14})	6.103516×10^{-5}
Tolerance (10 LSB)	6.103516×10^{-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()
```