

UNIVERSITÀ DI PISA

CORDIC: Cartesian to Polar Coordinate Transformation

Authors:

Andrea Di Matteo

Antonio Ciociola

Contents

1	Introduction	4
1.1	Specification	4
1.2	Circuit Applications	4
2	Architecture	6
2.1	Data Representation	6
2.2	Module Precision and Error Analysis	7
2.3	Phase Precision and Error Analysis	7
3	VHDL code	8
3.1	CORDIC	8
3.2	Atan LUT	12
4	Verification and testing	14
4.1	Testbench	14
4.2	Graph testbench	18
4.3	Verification	21
4.4	Error verification	21
5	Vivado results	24
5.1	Vivado Design flow	24
5.2	RTL	24
5.3	Synthesis	25
5.3.1	Critical Paths	25
5.4	Implementation	26
5.4.1	Utilization Report	27
5.4.2	Power Report	28
6	Final considerations	29

1 Introduction

1.1 Specification

It is required to design a digital circuit for implementing the transformation from cartesian coordinates into polar coordinates using the CORDIC algorithm in Vectoring mode. It is implemented with these recursive equations:

$$\begin{aligned}x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\z_{i+1} &= z_i - d_i \cdot \arctan(2^{-i})\end{aligned}$$

where $d_i = -1$ if $y_i > 0$, $+1$ otherwise. After n iterations, the equations converge to:

$$\begin{aligned}x_n &= A_n \cdot \sqrt{x_0^2 + y_0^2} \\y_n &= 0 \\z_n &= z_0 + \arctan\left(\frac{y_0}{x_0}\right) \\A_n &= \prod_{i=0}^n \sqrt{1 + 2^{-2i}}\end{aligned}$$

1.2 Circuit Applications

The CORDIC (COordinate Rotation DIgital Computer) algorithm is an iterative method for performing vector rotations and solving mathematical functions such as trigonometric, hyperbolic, exponential, logarithmic, and square root operations. It was introduced by Jack E. Volder in 1959 to simplify the computation of these functions in hardware with limited resources.

CORDIC is widely used because it eliminates the need for multiplication and division, relying instead on shift and addition operations. This makes it highly efficient for

hardware implementations, particularly in devices with limited computational power, such as embedded systems, calculators, and digital signal processors (DSPs).

For example the Intel 8087, a floating-point coprocessor introduced in the early 1980s, utilized the CORDIC algorithm to perform efficient trigonometric and hyperbolic computations, such as sine, cosine, and arctangent, without relying on hardware multipliers. Similarly, during the Apollo Lunar Module missions, a precursor concept to CORDIC was employed in the Apollo Guidance Computer (AGC) to perform real-time navigation calculations. The AGC used iterative methods to determine angles and distances for lunar landings, efficiently converting Cartesian spacecraft coordinates to polar forms to ensure precise trajectory adjustments during descent.

The CORDIC algorithm can operate in two different modes: **rotation mode** and **vectoring mode**

- **Rotation mode:** rotates a vector by a specified angle, used for calculating trigonometric functions or vector transformations
- **Vectoring mode:** determines the magnitude and angle of a vector, useful for converting Cartesian to polar coordinates

In this context, this project will focus on developing the vectoring mode of the CORDIC algorithm to convert Cartesian coordinates into polar form.

2 Architecture

By default, the CORDIC algorithm converges only for input angles within the range $(-99.7^\circ, 99.7^\circ)$. To address this limitation and enable the algorithm to handle arbitrary input angles, we introduced an **initial correction step**. This step adjusts the input angle to bring it into the principal range of the algorithm. The adjustment ensures that the CORDIC algorithm works seamlessly for all input angles, not just those within the default convergence range.

$$x_0 = -y_{\text{input}} \cdot d_{\text{input}}$$

$$y_0 = x_{\text{input}} \cdot d_{\text{input}}$$

$$z_0 = -\frac{\pi}{2} \cdot d_{\text{input}}$$

Additionally, since the iterative process of the CORDIC algorithm introduces a scaling factor A_n , we normalize the final result by dividing ρ (the magnitude) by A_n .

2.1 Data Representation

To implement the CORDIC algorithm, we used fixed-point arithmetic for the input and intermediate values. Specifically:

- x, y : signed 16-bit fixed-point representation with 8 bits allocated for the fractional part.
- ρ : unsigned 16-bit fixed-point representation with 8 bits allocated for the fractional part.
- z, θ : signed 16-bit fixed-point representation with 13 bits allocated for the fractional part. This decision was made because they always lie within the range $[-\pi, \pi]$, and the additional fractional bits ensure high angular precision.
- For the intermediate calculations a 24-bit representation was used to minimize truncation errors during the iterative process.
- All the inputs and the outputs are limited to 16 bits due to the I/O pin constraints of the Zybo board.

The number of iterations for the CORDIC algorithm was set to 16, as this provides a high level of precision while balancing computational efficiency. Beyond 16 iterations, the precision gained decreases significantly.

2.2 Module Precision and Error Analysis

Focusing on the module, ρ is unsigned so that also outputs with $\rho \geq 128$ can be represented. For the intermediate calculation the representation has 2 more bits for the integer part, this ensures that there isn't an overflow on x during the calculations.

2.3 Phase Precision and Error Analysis

Focusing on the phase, the intermediate values are represented in Q3.21 fixed-point format, offering a maximum absolute error ϵ_a of 2^{-21} . Over the course of 16 iterations, the maximum accumulated absolute sum of these values ϵ_M is

$$\epsilon_M = \epsilon_a * 16 = 2^{-21} * 2^4 = 2^{-17}$$

Since the final result is truncated to Q3.13, which involves discarding the least significant bits (LSBs), the accumulated error from the iterative sums becomes negligible. This ensures that the phase computation maintains high precision.

3 VHDL code

3.1 CORDIC

The following code shows the implementation of the CORDIC in vectoring mode for cartesian to polar conversion. This implementation supports the fix step for accepting inputs in all 4 quadrants, and also normalizes ρ .

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 ENTITY CORDIC IS
5
6   GENERIC (
7     M : POSITIVE := 24; -- internal representation
8     N : POSITIVE := 16; -- input size
9     ITERATIONS : POSITIVE := 16; -- CORDIC algorithm iterations
10    ITER_BITS : POSITIVE := 4 -- number of bits needed to
11      → represent iterations
12  );
13  PORT (
14    clk : IN STD_LOGIC;
15    rst : IN STD_LOGIC;
16    x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
17    y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
18    start : IN STD_LOGIC;
19    rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
20    theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
21    valid : OUT STD_LOGIC
22  );
23
24 END ENTITY;
25
26 ARCHITECTURE behavioral OF CORDIC IS
27
28   CONSTANT k : UNSIGNED(M - 1 DOWNTO 0) :=
29     → to_unsigned(INTEGER(0.6072528458 * (2 ** (M - 1))), M);
30   CONSTANT HALF_PI : SIGNED(M - 1 DOWNTO 0) :=
31     → to_signed(INTEGER(1.570796327 * (2 ** (M - 3))), M);
```

```

30  -- internal registers
31  SIGNAL x_t : SIGNED(M - 1 DOWNTO 0);
32  SIGNAL y_t : SIGNED(M - 1 DOWNTO 0);
33  SIGNAL z_t : SIGNED(M - 1 DOWNTO 0);
34
35  -- output registers
36  SIGNAL x_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
37  SIGNAL z_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
38
39  -- atan table address and output
40  SIGNAL address : STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
41  SIGNAL atan_out : STD_LOGIC_VECTOR(M - 1 DOWNTO 0);
42
43  SIGNAL sign : STD_LOGIC;
44
45  -- atan table
46  COMPONENT ATAN_LUT IS
47    PORT (
48      address : IN STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
49      lut_out : OUT STD_LOGIC_VECTOR(M - 1 DOWNTO 0)
50    );
51  END COMPONENT;
52
53  -- state type and registers
54  TYPE state_t IS (WAITING, FIX, COMPUTING, FINISHED);
55  SIGNAL current_state : state_t;
56
57  -- counter for iterations
58  SIGNAL counter : UNSIGNED(ITER_BITS - 1 DOWNTO 0);
59
60 BEGIN
61  -----
62  -- INSTANTIATE ALL COMPONENTS
63  -----
64
65  -- output assignment
66  rho <= x_out;
67  theta <= z_out;
68
69  -- sign bit
70  sign <= y_t(M - 1);
71
72  -- atan table
73  atan_lut_inst : ATAN_LUT
74  PORT MAP(
75    address => address,
76    lut_out => atan_out
77  );
78
79  -- atan table address
80  address <= STD_LOGIC_VECTOR(counter);

```

```

81
82 -- CONTROL PART
83
84
85
86 control : PROCESS (clk, rst)
87 BEGIN
88   IF (rising_edge(clk)) THEN
89     IF rst = '1' THEN
90       current_state <= WAITING;
91     ELSE
92
93       CASE current_state IS
94         WHEN WAITING =>
95           IF start = '1' THEN
96             current_state <= FIX;
97           ELSE
98             current_state <= WAITING;
99           END IF;
100
101      WHEN FIX =>
102        current_state <= COMPUTING;
103
104      WHEN COMPUTING =>
105        IF counter = ITERATIONS - 1 THEN
106          current_state <= FINISHED;
107        ELSE
108          current_state <= COMPUTING;
109        END IF;
110
111      WHEN FINISHED =>
112        current_state <= WAITING;
113
114      WHEN OTHERS =>
115        current_state <= WAITING;
116
117       END CASE;
118     END IF;
119   END IF;
120 END PROCESS;
121
122 -- OPERATIONAL PART
123
124
125
126 OPEATIONAL : PROCESS (clk, rst)
127 BEGIN
128   IF (rising_edge(clk)) THEN
129     IF rst = '1' THEN
130       valid <= '0';
131       x_out <= (OTHERS => '0');

```

```

132      z_out <= (OTHERS => '0');
133      counter <= (OTHERS => '0');
134      x_t <= (OTHERS => '0');
135      y_t <= (OTHERS => '0');
136      z_t <= (OTHERS => '0');
137  ELSE
138
139      -- Default assignment
140      x_t <= (OTHERS => '-');
141      y_t <= (OTHERS => '-');
142      z_t <= (OTHERS => '-');
143      x_out <= (OTHERS => '-');
144      z_out <= (OTHERS => '-');
145      counter <= (OTHERS => '0');
146
147  CASE current_state IS
148    WHEN WAITING =>
149      x_t <= shift_left(resize(signed(x), M), (M - N - 2));
150      -- todo spiegare
151      y_t <= shift_left(resize(signed(y), M), (M - N - 2));
152      z_t <= to_signed(0, M);
153      valid <= '1';
154      x_out <= x_out;
155      z_out <= z_out;
156
157    WHEN FIX =>
158      IF sign = '1' THEN
159        x_t <= - y_t;
160        y_t <= x_t;
161        z_t <= z_t - HALF_PI;
162      ELSE
163        x_t <= y_t;
164        y_t <= - x_t;
165        z_t <= z_t + HALF_PI;
166      END IF;
167
168      valid <= '0';
169      counter <= (OTHERS => '0');
170
171    WHEN COMPUTING =>
172      IF sign = '1' THEN
173        x_t <= x_t - shift_right(y_t, to_integer(counter));
174        y_t <= y_t + shift_right(x_t, to_integer(counter));
175        z_t <= z_t - signed(atan_out);
176      ELSE
177        x_t <= x_t + shift_right(y_t, to_integer(counter));
178        y_t <= y_t - shift_right(x_t, to_integer(counter));
179        z_t <= z_t + signed(atan_out);
180      END IF;
181
182      counter <= counter + 1;

```

```

182         valid <= '0';
183
184     WHEN FINISHED =>
185         x_out <=
186             STD_LOGIC_VECTOR(resize(shift_right(unsigned(x_t) *
187                 k, M - 1), M)(M - 1 - 2 DOWNTO M - N - 2));
188         z_out <= STD_LOGIC_VECTOR(z_t(M - 1 DOWNTO M - N));
189
190         valid <= '1';
191
192     END CASE;
193     END IF;
194   END IF;
195 END PROCESS;
END ARCHITECTURE;

```

Listing 3.1: CORDIC.vhd

3.2 Atan LUT

The values of the LUT table were calculated using the formula:

$$\text{LUT}_i = \left\lfloor \text{atan}(2^{-i}) * 2^{21} \right\rfloor$$

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4
5 ENTITY ATAN_LUT IS
6   port (
7     address : in std_logic_vector(3 downto 0);
8     lut_out : out std_logic_vector(23 downto 0)
9   );
10 end entity;
11
12 architecture rtl of ATAN_LUT is
13
14   type LUT_t is array (natural range 0 to 15) of integer;
15   constant LUT: LUT_t := (
16     0 => 1647099,
17     1 => 972339,
18     2 => 513757,
19     3 => 260791,
20     4 => 130901,
21     5 => 65514,
22     6 => 32765,

```

```
23    7 => 16383,
24    8 => 8191,
25    9 => 4095,
26    10 => 2047,
27    11 => 1023,
28    12 => 511,
29    13 => 255,
30    14 => 127,
31    15 => 63
32  );
33
34 begin
35
36 PROCESS (address)
37 BEGIN
38   lut_out <=
39     STD_LOGIC_VECTOR(to_signed(LUT(to_integer(unsigned(address))),
40                         24));
41 END PROCESS;
42
43 end architecture;
```

Listing 3.2: ATAN_LUT.vhd

4 Verification and testing

4.1 Testbench

The testbench CORDIC_tb verifies the basic functionality of the circuit by performing a power on reset and iteratively gives one input and waits for the corresponding output. For debugging reasons the testbench also prints the output values on the console.

```
1 LIBRARY IEEE;
2 USE IEEE.STD_LOGIC_1164.ALL;
3 USE IEEE.NUMERIC_STD.ALL;
4 USE ieee.math_real.ALL;
5
6 ENTITY CORDIC_TB IS
7   GENERIC (
8     N : POSITIVE := 16;
9     floating : INTEGER := 8
10    );
11 END CORDIC_TB;
12
13 ARCHITECTURE Behavioral OF CORDIC_TB IS
14   -- Component declaration for CORDIC
15
16   COMPONENT CORDIC
17     PORT (
18       clk : IN STD_LOGIC;
19       rst : IN STD_LOGIC;
20       x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
21       y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
22       start : IN STD_LOGIC;
23       rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
24       theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
25       valid : OUT STD_LOGIC
26     );
27   END COMPONENT;
28
29   -- Signals for CORDIC inputs and outputs
30
31   SIGNAL clk : STD_LOGIC := '0';
```

```

32  SIGNAL reset : STD_LOGIC := '0';
33  SIGNAL x : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
34  SIGNAL y : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
35  SIGNAL start : STD_LOGIC := '0';
36  SIGNAL rho : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
37  SIGNAL theta : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
38  SIGNAL valid : STD_LOGIC := '0';

39
40  SIGNAL run_simulation : STD_LOGIC := '1';

41
42  -- Clock period definition
43  CONSTANT T_clk : TIME := 20 ns;

44
45  -- Coordinate type
46  TYPE Coordinate IS RECORD
47    x : real;
48    y : real;
49  END RECORD;
50  CONSTANT n_coordinates : NATURAL := 9;

51
52  TYPE CoordinateArray IS ARRAY (0 TO n_coordinates - 1) OF
53    Coordinate;

54  -- Array of coordinates to test
55  CONSTANT Coordinates : CoordinateArray := (
56
57    (1.0, 1.0),
58    (10.0, 10.0),
59    (0.0, 1.0),
60    (-0.1, -4.0),
61    (-1.0, 1.0),
62    (-1.0, 0.0),
63    (-1.0, 0.1),
64    (31.0, 31.0),
65    (127.0, 127.0)
66  );
67
68  -- function to print values in report
69  FUNCTION to_real(val : unsigned(N - 1 DOWNTO 0); fraction_bits :
70    INTEGER) RETURN real IS
71  BEGIN
72    RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
73  END FUNCTION;

74  -- function to print values in report
75  FUNCTION to_real(val : signed(N - 1 DOWNTO 0); fraction_bits :
76    INTEGER) RETURN real IS
77  BEGIN
78    RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
79  END FUNCTION;

```

```

80 PROCEDURE echo(arg : IN STRING := "") IS
81 BEGIN
82     std.textio.write(std.textio.output, arg & LF); -- LF ensures a
83     -- newline after each message
84 END PROCEDURE echo;
85
86 BEGIN
87     -- Instantiate the CORDIC component
88     cordic_inst : CORDIC
89     PORT MAP(
90         clk => clk,
91         rst => reset,
92         x => x,
93         y => y,
94         start => start,
95         rho => rho,
96         theta => theta,
97         valid => valid
98     );
99
100    -- todo fix behavior if cordic is not ready / does not work
101
102    clk <= (NOT(clk) AND run_simulation) AFTER T_clk / 2;
103
104    -- test process
105    test : PROCESS
106    BEGIN
107        reset <= '1';
108        start <= '0';
109        WAIT FOR 2 * T_clk;
110
111        reset <= '0';
112        FOR i IN 0 TO n_coordinates - 1 LOOP
113
114            IF valid = '0' THEN
115                WAIT UNTIL valid = '1';
116            END IF;
117
118            x <= STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).x *
119                         2.0 ** floating), N));
120            y <= STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).y *
121                         2.0 ** floating), N));
122            start <= '1';
123
124            WAIT FOR 5 * T_clk;
125
126            start <= '0';
127            IF valid = '0' THEN
128                WAIT UNTIL valid = '1';
129            END IF;
130
131            WAIT FOR 1 * T_clk;

```

```

128      -- Osservazione del valore del modulo e della fase
129      echo("");
130      echo("Test " & INTEGER'image(i) & " X: " &
131          ↵ real'image(Coordinates(i).x) & " Y: " &
132          ↵ real'image(Coordinates(i).y));
133      echo("-----");
134      echo("Module (Q8.8) = " & real'image(to_real(unsigned(rho),
135          ↵ 8)));
136      echo("Phase (Q3.13) = " & real'image(to_real(signed(theta),
137          ↵ 13)));
138      echo("-----");
139
140      WAIT FOR 10 * T_clk;
141
142      END LOOP;
143
144      run_simulation <= '0';
145      WAIT;
146  END PROCESS;
147
148 END ARCHITECTURE;

```

Listing 4.1: CORDIC_tb.vhd

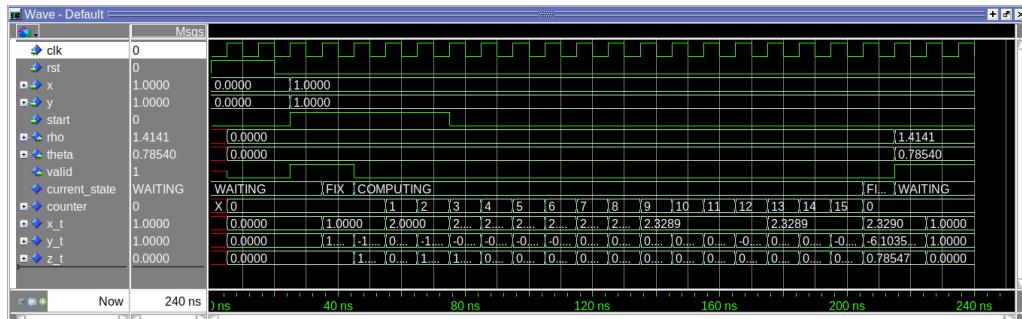


Figure 4.1: Simulation of a single iteration of the testbench

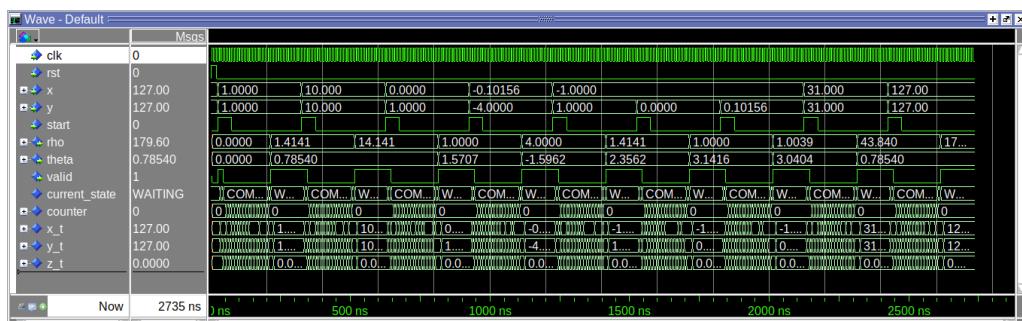


Figure 4.2: Full simulation of the testbench

4.2 Graph testbench

This testbench servers the purpose of ensuring the functionality of the circuit over the range of possible inputs. The test was performed on 512×512 input points uniformly distributed within the range $(-128, 128)$ for both x and y .

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE ieee.numeric_std.ALL;
4 USE ieee.math_real.ALL;
5 USE std.textio.ALL;
6
7 ENTITY GRAPH_tb IS
8   GENERIC (
9     N : POSITIVE := 16;
10    floating : INTEGER := 8 -- (Q8.8)
11  );
12 END ENTITY;
13
14 ARCHITECTURE Behavioral OF GRAPH_tb IS
15
16 -----
17 -- CORDIC component
18 -----
19 COMPONENT CORDIC
20   PORT (
21     clk : IN STD_LOGIC;
22     rst : IN STD_LOGIC;
23     x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
24     y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
25     start : IN STD_LOGIC;
26     rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
27     theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
28     valid : OUT STD_LOGIC
29   );
30 END COMPONENT;
31
32 -----
33 -- Signals
34 -----
35 SIGNAL clk : STD_LOGIC := '0';
36 SIGNAL rst : STD_LOGIC := '0';
37 SIGNAL x_in : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS =>
38   <> '0');
39 SIGNAL y_in : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS =>
40   <> '0');
41 SIGNAL start_in : STD_LOGIC := '0';
42 SIGNAL rho_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
43 SIGNAL theta_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
44 SIGNAL valid_out : STD_LOGIC := '0';

```

```
43 SIGNAL run_simulation : STD_LOGIC := '1';
44
45 -----
46 -- Simulation parameters
47 -----
48
49
50 CONSTANT T_clk : TIME := 20 ns; -- clock period = 20 ns
51 CONSTANT SAMPLES : INTEGER := 512; -- samples per dimension
52
53 -----
54 -- Output file
55 -----
56
57 FILE results_file : text OPEN write_mode IS
58   <-- "cordic_results_512x512.txt";
59
60 -----
61 -- Conversion function from std_logic_vector to real
62 -----
63
64 FUNCTION to_real(val : unsigned(N - 1 DOWNTO 0); fraction_bits :
65   <-- INTEGER) RETURN real IS
66 BEGIN
67   RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
68 END FUNCTION;
69
70
71 FUNCTION to_real(val : signed(N - 1 DOWNTO 0); fraction_bits :
72   <-- INTEGER) RETURN real IS
73 BEGIN
74   RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
75 END FUNCTION;
76
77
78 BEGIN
79
80 -----
81 -- CORDIC declaration
82 -----
83
84 UUT : CORDIC
85 PORT MAP(
86   clk => clk,
87   rst => rst,
88   x => x_in,
89   y => y_in,
90   start => start_in,
91   rho => rho_out,
92   theta => theta_out,
93   valid => valid_out
94 );
95
```

```

91  -- CLK process
92  -----
93  clk_process : PROCESS
94  BEGIN
95    WHILE run_simulation = '1' LOOP
96      clk <= '1';
97      WAIT FOR T_clk/2;
98      clk <= '0';
99      WAIT FOR T_clk/2;
100     END LOOP;
101    WAIT; -- end process when run_simulation = '0'
102  END PROCESS clk_process;
103
104 -----
105  -- Test process
106  -----
107  test_process : PROCESS
108    VARIABLE L : line;
109    VARIABLE real_x, real_y : real;
110    VARIABLE mod_val, phase_val : real;
111  BEGIN
112  -----
113    -- Power on reset
114  -----
115    rst <= '1';
116    start_in <= '0';
117    WAIT FOR 5 * T_clk;
118    rst <= '0';
119    WAIT FOR 5 * T_clk;
120
121  -----
122    -- Loop over (i, j) to test all values
123  -----
124    FOR i IN 0 TO SAMPLES - 1 LOOP
125      FOR j IN 0 TO SAMPLES - 1 LOOP
126
127        real_x := - 128.0 + real(i) * 256.0 / real(SAMPLES);
128        real_y := - 128.0 + real(j) * 256.0 / real(SAMPLES);
129
130        -- Q8.8 Conversion (signed)
131        x_in <= STD_LOGIC_VECTOR(to_signed(INTEGER(floor(real_x *
132          2.0 ** floating)), N));
133        y_in <= STD_LOGIC_VECTOR(to_signed(INTEGER(floor(real_y *
134          2.0 ** floating)), N));
135
136        start_in <= '1';
137        WAIT FOR 2*T_clk;
138        start_in <= '0';
139
140        WAIT UNTIL valid_out = '1';
141        WAIT FOR 10 ns;

```

```

140
141     mod_val := to_real(unsigned(rho_out), 8); -- Q8.8
142     phase_val := to_real(signed(theta_out), 13); -- Q3.13
143
144     -----
145     -- Write on text file: x, y, rho, theta
146     -----
147     -- Comma separated values
148     write(L, real_x, RIGHT, 0, 6);
149     write(L, STRING'(","));
150     write(L, real_y, RIGHT, 0, 6);
151     write(L, STRING'(","));
152     write(L, mod_val, RIGHT, 0, 6);
153     write(L, STRING'(","));
154     write(L, phase_val, RIGHT, 0, 6);
155     writeline(results_file, L);
156
157     WAIT FOR 1 * T_clk;
158
159     END LOOP;
160   END LOOP;
161
162     -----
163     -- End simulation
164     -----
165     run_simulation <= '0';
166     WAIT;
167   END PROCESS test_process;
168
169 END Behavioral;

```

Listing 4.2: GRAPH_tb.vhd

4.3 Verification

4.4 Error verification

The implementation of the CORDIC algorithm in vectoring mode for Cartesian-to-polar conversion demonstrates high accuracy in both module (ρ) and phase (θ) outputs. Using a fixed-point representation of $UQ8.8$ for ρ and $Q3.13$ for θ , the errors were summarized as follows:

Module Error: Max = 0.004030 units, Mean = 0.001889 units

Phase Error: Max = 0.000154 rad, Mean = 0.000062 rad

Module Error: $\log_2(\text{Max error}) \approx -7.955$

$$\text{Phase Error: } \log_2(\text{Max error}) \approx -12.664$$

The observed errors are well within the precision limits of 2^{-8} for ρ and 2^{-13} for θ . For the most part, discrepancies between the expected values and the output values are primarily caused by the conversion of fixed-point representations into real numbers for plotting and analysis.

The 3D plots of module and phase errors visually confirm the bounded and negligible nature of the errors across all tested input combinations. Assuming no error on the input, this analysis validates that the algorithm introduces almost no error in the output.

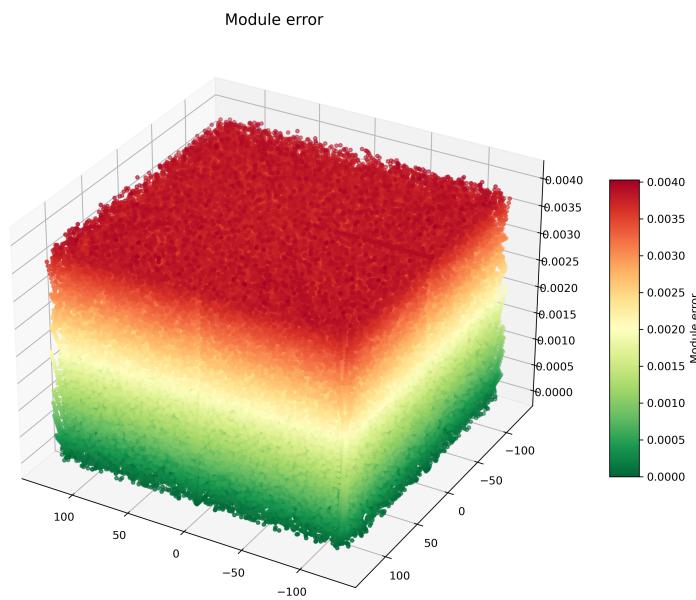


Figure 4.3: 3D plot of module error (ρ).

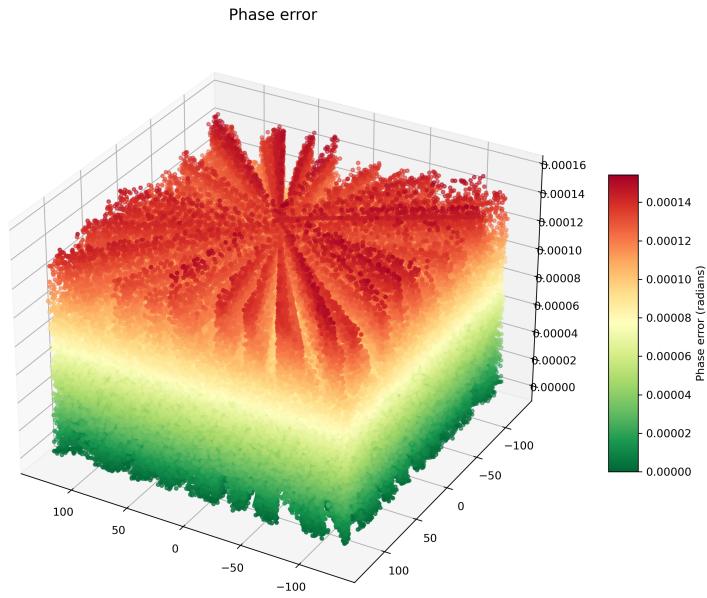


Figure 4.4: 3D plot of phase error (θ).

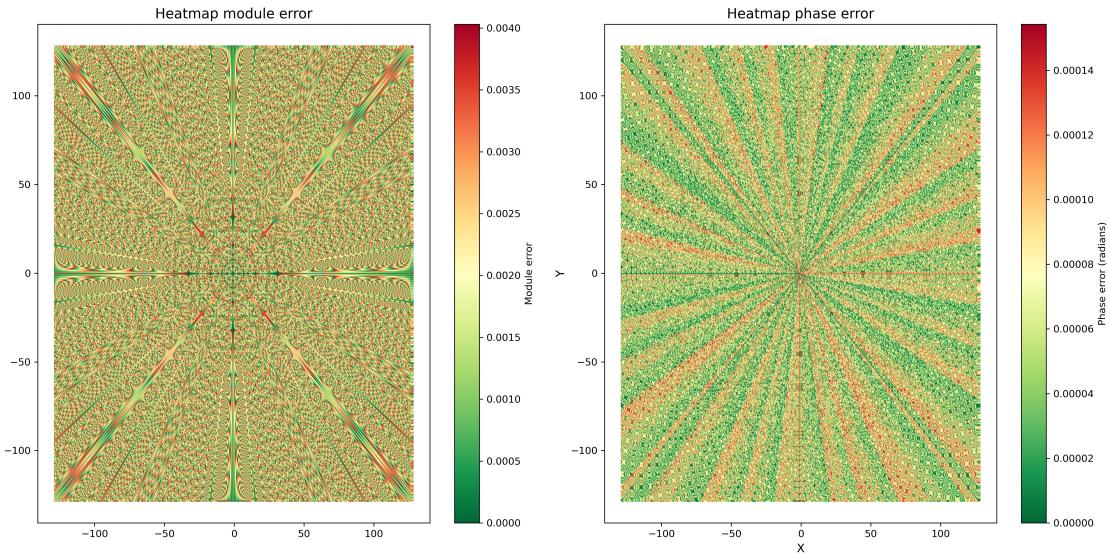


Figure 4.5: Heatmap for phase and module errors.

5 Vivado results

After completing the verification phase, the circuit design was synthesized and implemented using the Vivado Tool, specifically targeting the Zybo Zynq-7010 development board.

5.1 Vivado Design flow

For the implementation of the CORDIC algorithm in vectoring mode for Cartesian-to-Polar conversion, the Vivado design flow was employed using Xilinx/AMD Vivado software. This process involved RTL Elaboration, Synthesis, and Implementation on the target FPGA, along with the application of design constraints and the extraction of power and timing reports. To ensure accurate and reliable timing analysis, all combinational logic paths were structured to follow a Register-Logic-Register configuration.

5.2 RTL

Vivado produced a logic network made of:

1. 155 cells (e.g. multiplexers, DFFs, adders)
2. 68 I/O ports ($16_x + 16_y + 16_\rho + 16_\theta + 1_{clk} + 1_{rst} + 1_{start} + 1_{valid}$)
3. 982 nets (for connecting all the components)

The RTL Analysis generated the following Elaborated Design consistent with the expected structure of the system:

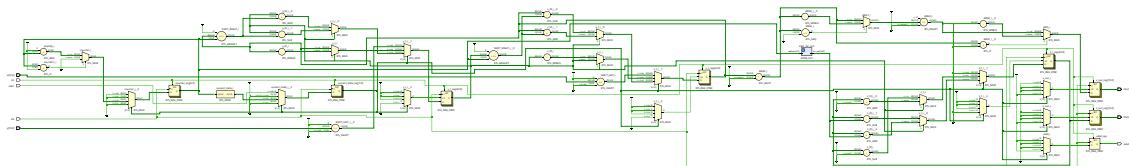


Figure 5.1: Elaborated RTL design.

5.3 Synthesis

5.3.1 Critical Paths

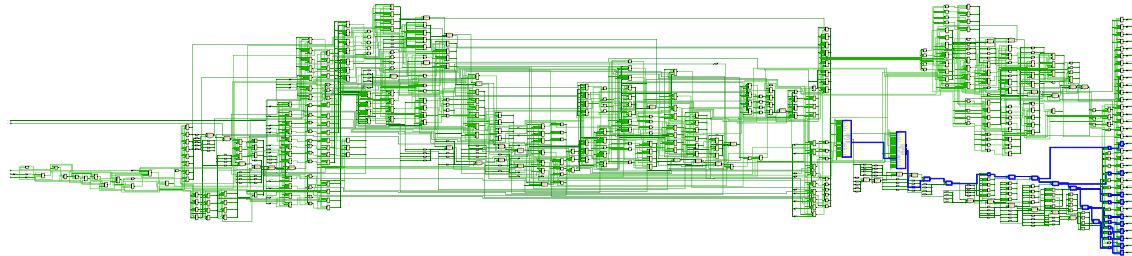


Figure 5.2: Figure showing the elaborated RTL design with the main critical paths for set-up-time-violation found in synthesis step highlighted in blue.

Name	Slack	Levels	Routes	From	To	Total	Delay
Path 1	9.30	12	6	ARG2/CLK	x_out_reg[15]/D	10.55	
Path 2	9.33	11	6	ARG2/CLK	x_out_reg[12]/D	10.52	
Path 3	9.45	10	6	ARG2/CLK	x_out_reg[8]/D	10.40	
Path 4	9.51	12	6	ARG2/CLK	x_out_reg[14]/D	10.33	
Path 5	9.56	9	6	ARG2/CLK	x_out_reg[4]/D	10.28	
Path 6	9.57	11	6	ARG2/CLK	x_out_reg[11]/D	10.28	
Path 7	9.62	12	6	ARG2/CLK	x_out_reg[13]/D	10.23	
Path 8	9.62	11	6	ARG2/CLK	x_out_reg[10]/D	10.23	
Path 9	9.67	8	6	ARG2/CLK	x_out_reg[0]/D	10.18	
Path 10	9.69	10	6	ARG2/CLK	x_out_reg[7]/D	10.16	

Table 5.1: Table showing data of the main critical paths found in synthesis step

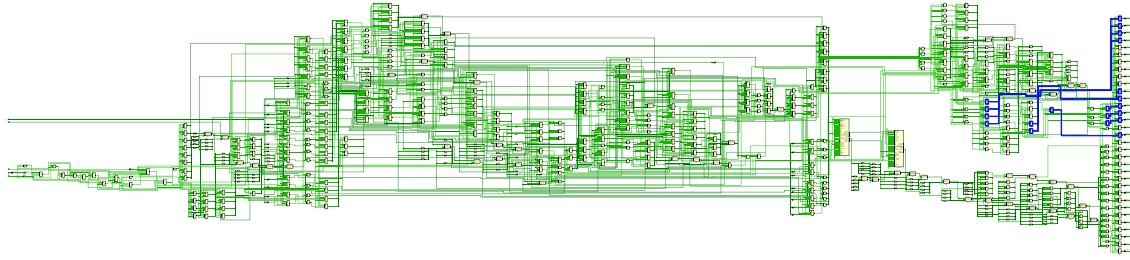


Figure 5.3: Figure showing the elaborated RTL design with the main critical paths for hold-time-violation found in synthesis step highlighted in blue.

Name	Slack	Levels	Routes	From	To	Total Delay
Path 11	0.16	0	1	z_t_reg[23]/C	z_out_reg[15]/D	0.29
Path 12	0.16	0	1	z_t_reg[8]/C	z_out_reg[0]/D	0.29
Path 13	0.16	0	1	z_t_reg[18]/C	z_out_reg[10]/D	0.29
Path 14	0.16	0	1	z_t_reg[19]/C	z_out_reg[11]/D	0.29
Path 15	0.16	0	1	z_t_reg[20]/C	z_out_reg[12]/D	0.29
Path 16	0.16	0	1	z_t_reg[21]/C	z_out_reg[13]/D	0.29
Path 17	0.16	0	1	z_t_reg[22]/C	z_out_reg[14]/D	0.29
Path 18	0.16	0	1	z_t_reg[9]/C	z_out_reg[1]/D	0.29
Path 19	0.16	0	1	z_t_reg[10]/C	z_out_reg[2]/D	0.29
Path 20	0.16	0	1	z_t_reg[11]/C	z_out_reg[3]/D	0.29

Table 5.2: Table showing the characteristics regarding critical paths for hold-time-violation found in synthesis step.

5.4 Implementation

Name	Slack	Levels	Routes	From	To	Total Delay
Path 1	8.90	10	3	ARG2/CLK	x_out_reg[9]/D	10.99
Path 2	8.93	10	3	ARG2/CLK	x_out_reg[12]/D	10.96
Path 3	8.99	11	3	ARG2/CLK	x_out_reg[13]/D	10.86
Path 4	9.08	11	3	ARG2/CLK	x_out_reg[15]/D	10.82
Path 5	9.10	9	3	ARG2/CLK	x_out_reg[8]/D	10.84
Path 6	9.10	9	3	ARG2/CLK	x_out_reg[6]/D	10.84
Path 7	9.14	10	3	ARG2/CLK	x_out_reg[10]/D	10.71
Path 8	9.15	9	3	ARG2/CLK	x_out_reg[7]/D	10.74
Path 9	9.18	11	3	ARG2/CLK	x_out_reg[14]/D	10.72
Path 10	9.22	10	3	ARG2/CLK	x_out_reg[11]/D	10.63

Table 5.3: Table showing the main critical paths for the setup-time-violation during the implementation step.

Name	Slack	Levels	Routes	From	To	Total Delay
Path 11	0.17	1	1	counter_reg[0]/C	counter_reg[2]/D	0.31
Path 12	0.17	1	1	counter_reg[0]/C	counter_reg[1]/D	0.31
Path 13	0.18	1	1	counter_reg[0]/C	counter_reg[3]/D	0.31
Path 14	0.22	0	1	z_t_reg[19]/C	z_out_reg[11]/D	0.25
Path 15	0.22	0	1	z_t_reg[23]/C	z_out_reg[15]/D	0.27
Path 16	0.23	0	1	z_t_reg[8]/C	z_out_reg[0]/D	0.26
Path 17	0.25	0	1	z_t_reg[12]/C	z_out_reg[4]/D	0.35
Path 18	0.25	0	1	z_t_reg[22]/C	z_out_reg[14]/D	0.33
Path 19	0.26	1	1	FSM_reg[0]/C	FSM_reg[0]/D	0.35
Path 20	0.27	0	1	z_t_reg[18]/C	z_out_reg[10]/D	0.37

Table 5.4: Table showing the main critical paths for hold-time-violation found in the implementation step.

5.4.1 Utilization Report

Resource	Utilization (%)	Description
Slice LUTs	2.05%	Look-Up Tables used as logic
Slice Registers	0.32%	Registers used in the design
Slice	2.30%	Total slices utilized
LUT as Logic	2.05%	LUTs specifically used as logic
DSPs	2.50%	Digital Signal Processing blocks
Bonded IOB	68.00%	Bonded Input/Output Blocks
BUFGCTRL	3.13%	Global Clock Buffers

Table 5.5: Resource utilization for the CORDIC design (only non-zero values shown)

5.4.2 Power Report

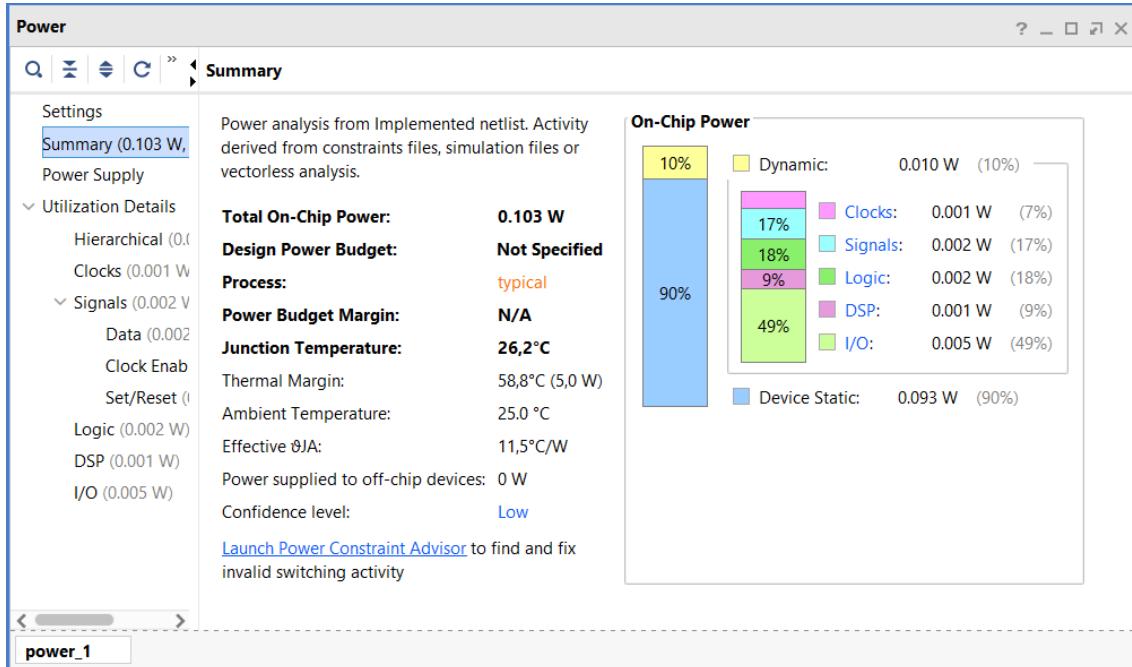


Figure 5.4: Vivado Power Report after implementation.

The power report highlights a total on-chip power consumption of 0.103 W, with 90% attributed to static power (0.093 W) and 10% to dynamic power (0.010 W). The dynamic power is distributed among different components: clocks (7%), signals (17%), logic (18%), DSP (9%), and I/O (49%), with I/O being the highest consumer of dynamic power. It is important to note that this device will not be used alone but is being connected to I/O pins solely for testing purposes.

The high value of static power over dynamic power is largely to be attributed to the fact that the design is using very little of the available hardware. With reference to the Utilization Report in Chapter 4, resources like Slice LUTs and Registers are used only for 2.05% and 0.32%, this makes the overall switching activity of the chip very small.

6 Final considerations

The design process adopted in this work demonstrates an efficient approach to implementing the CORDIC algorithm. The iterative methodology ensures high accuracy while maintaining low resource utilization, as confirmed by the Vivado synthesis and power analysis reports.

It is important to note that while the implemented circuit is tailored for the constraints of the Zybo board, the modularity of the design makes it adaptable to other hardware platforms. The testing and verification phase, which included testbenches and graphical analysis, validated the functionality and robustness of the system across a wide range of input conditions.