UNIVERSITÀ DI PISA

CORDIC: Cartesian to Polar Coordinate Transformation

Authors:

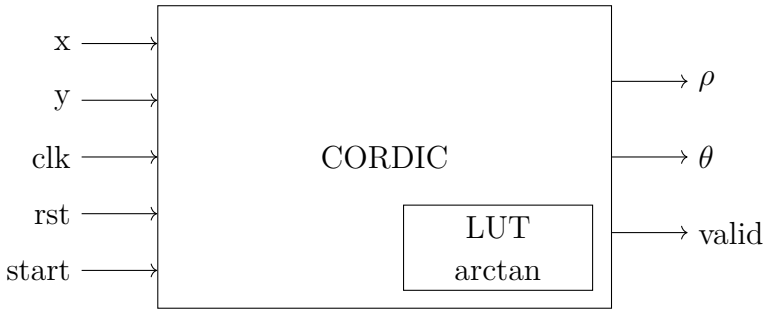
Andrea Di Matteo

Antonio Ciociola

ANNO ACCADEMICO 2024/2025

Contents

1 Introduction



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:

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \arctan(2^{-i}) \end{aligned} \tag{1.1}$$

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

$$x_n = A_n \cdot \sqrt{x_0^2 + y_0^2}$$

$$y_n = 0$$

$$z_n = z_0 + \arctan\left(\frac{y_0}{x_0}\right)$$

$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$

1.1 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:** the input vector will be rotated towards the x-axis until the y-component is reduced to zero. After a fixed number of iterations, the modulus of the vector can be retrieved on the x-axis, while its phase corresponds to the angle of rotation.

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

2.1 Fix Step

By default, the CORDIC algorithm converges only for input angles within the range $(-99.7°, 99.7°)$. 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.

2.2 Algorithm implementation

With reference to the Equations 1.1, the combinatorics of a single step is given by the following block diagram:

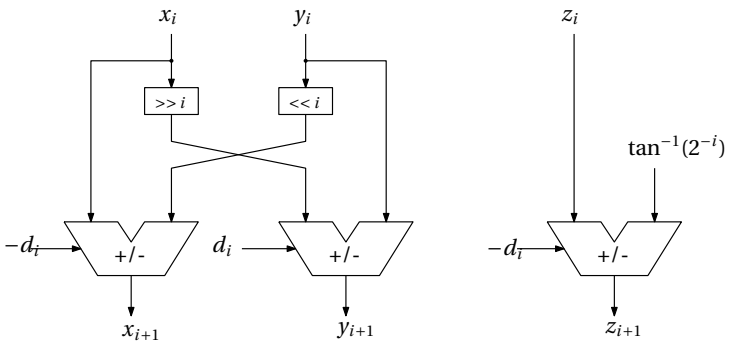


Figure 2.1: Block diagram of a single step of CORDIC

Since the CORDIC algorithm typically converges at a rate of one bit per iteration, the number of steps is predetermined at design time based on data representation. For this reason, a possible implementation of CORDIC involves a series of n combinatorics as shown in Figure 2.1. The resulting combinatorics between input and output registers will assume the following form:

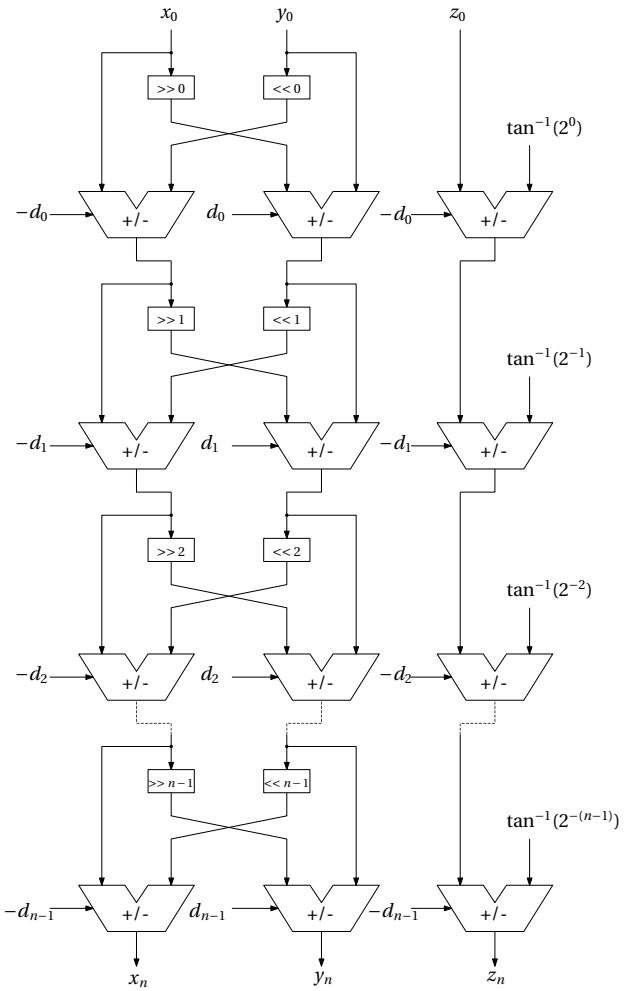


Figure 2.2: Block diagram of the full combinatorics of CORDIC algorithm

Assuming the shift operation has no cost, the logic required will be composed by $3 \cdot N$ adders, where N represents the number of iterations needed for convergence. Additionally, the function will go through N levels of logic. Solutions designed in this way tend to have a long critical path, resulting in a low clock frequency for the network.

To address this limitation, pipelined solutions are generally preferred. By introducing three registers between each step of the process, the throughput of the design remains unchanged. However, the critical path reduces to:

$$T_{\text{critical}} = T_{c \to q} + T_{\text{prop}} + T_{\text{setup}},$$

where $T_{c \to q}$ is the clock-to-Q delay of the flip-flop, T_{prop} is the propagation delay of the combinatorial logic which, this time, is just a single level of adder, and T_{setup} is the setup time of the next flip-flop.

Although this modification improves the clock speed, it introduces new trade-offs: The delay for processing a single input becomes $N \cdot T_{\text{clk}}$. More critically, the cost in terms of area occupation and number of components increases significantly, as $3 \cdot N$ additional registers are inserted into the design. This also negatively impacts power consumption. Hybrid solutions, where registers are placed every 2 or 3 levels of logic, are also commonly adopted nowadays to balance clock speed, area efficiency, and power consumption.

The solution we adopted takes advantage of the recursive nature of the CORDIC equations by slightly modifying the single step and incorporating a feedback loop. This design choice ensures a small critical path, allowing for a high clock frequency while minimizing area occupation, cost, and power consumption:

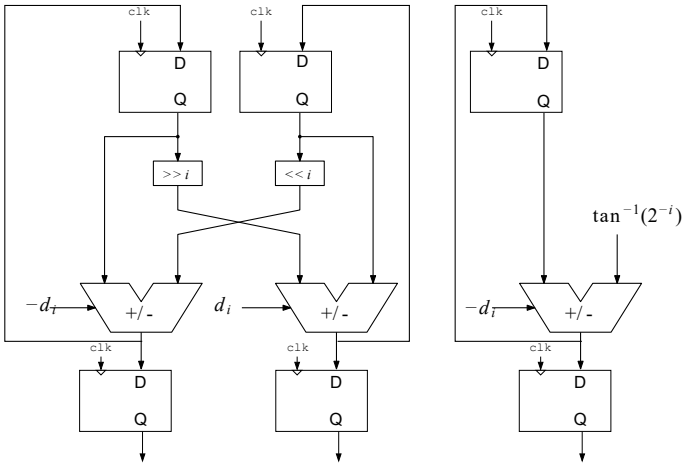


Figure 2.3: Block diagram of a single step CORDIC with feedback loop

This approach was chosen with a specific scenario in mind: the CORDIC algorithm acting as an on-board accelerator. In such a scenario, a smaller design with lower power consumption is highly desirable, as it facilitates integration without significantly affecting the board's utilization, heat dissipation, or energy requirements.

In order to achieve this, we will adopt a Finite State Machine (FSM) with a control part, which includes a status register, and an operational part dedicated to performing the necessary calculations.

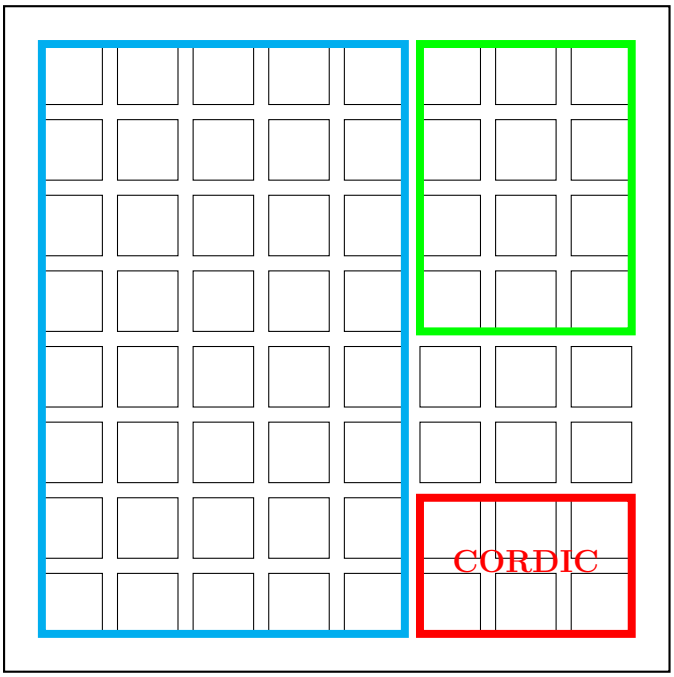


Figure 2.4: Schematic representation of an FPGA. The grid represents the available logic blocks, with coloured outlines indicating resource usage. The CORDIC block, highlighted in red, is a small portion of the total FPGA area.

It is true that this design results in a delay of $N \cdot T_{\text{clk}}$ for each operation and reduces throughput to $1/N$ (processing one input every N clock cycles). However, these trade-offs are justified by the need to prioritize compactness and efficiency over throughput, aligning with the intended use case of the CORDIC as an auxiliary accelerator.

2.3 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.

2.4 Module Precision

The parameter ρ is represented as unsigned with a UQ8.8 fixed-point format. During intermediate calculations, it uses a UQ10.14 format instead of UQ8.16 in order to prevent overflow. Specifically, the two additional integer bits account for cases like $x = 127$ and $y = 127$, where the output, before normalization, is

$$\text{Output} = A_n \cdot \sqrt{2} \cdot 127,$$

with $A_n \approx 1.6467605$. Substituting, the value is approximately

$$\text{Output} \approx 1.6467605 \cdot 1.4142135 \cdot 127 \approx 295.342.$$

This value exceeds the UQ8.16 range but fits within UQ10.14. After normalization, the output is scaled by $\frac{1}{A_n} \approx 0.607252$ and then converted to the Q8.8 format by truncation of the least significant bits (LSBs).

2.5 Phase Precision and Error Analysis

Focusing on the phase, θ us represented as signed with Q3.13 fixed-point format. During intermediate calculations it uses a Q3.21 fixed-point format. Assuming no error on the input, the values of $\text{atan}(2^{-1})$ in Q3.21 format have an absolute error ϵ_a of 2^{-21}. Over the course of 16 iterations, the accumulated error ϵ_M becomes:

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

Since the final result is truncated to Q3.13, which involves discarding the 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 fixed step for accepting inputs in all 4 quadrants, and also normalizes ρ. As said in Section 2.2, we followed a Finite State Machine (FSM) approach where the control part uses a status register represented as `current_state`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY CORDIC IS

  GENERIC (
    M : POSITIVE := 24; -- internal representation size
    N : POSITIVE := 16; -- input size
    ITERATIONS : POSITIVE := 16; -- CORDIC algorithm iterations
    ITER_BITS : POSITIVE := 4 -- number of bits needed to represent
    ↪  iterations
  );
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
    y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
    start : IN STD_LOGIC;
    rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
    theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
    valid : OUT STD_LOGIC
  );

END ENTITY;

ARCHITECTURE behavioral OF CORDIC IS

  CONSTANT k : UNSIGNED(M - 1 DOWNTO 0) :=
  ↪  to_unsigned(INTEGER(0.6072528458 * (2 ** (M - 1))), M);
```

```
28    CONSTANT HALF_PI : SIGNED(M - 1 DOWNTO 0) :=
      ↪  to_signed(INTEGER(1.570796327 * (2 ** (M - 3))), M);
29
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    -- ISTANTIATE 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    ------------------------------------------------
83    -- CONTROL PART
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   ------------------------------------------------
123   -- OPERATIONAL PART
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 and y_t have 2 more bits for the integer compared to
                ↪  x and y
150             -- this avoids overflows during the computation
151             x_t <= shift_left(resize(signed(x), M), (M - N - 2));
152             y_t <= shift_left(resize(signed(y), M), (M - N - 2));
153             z_t <= to_signed(0, M);
154             valid <= '1';
155             x_out <= x_out;
156             z_out <= z_out;
157
158           WHEN FIX =>
159             IF sign = '1' THEN
160               x_t <= - y_t;
161               y_t <= x_t;
162               z_t <= z_t - HALF_PI;
163             ELSE
164               x_t <= y_t;
165               y_t <= - x_t;
166               z_t <= z_t + HALF_PI;
167             END IF;
168
169             valid <= '0';
170             counter <= (OTHERS => '0');
171
172           WHEN COMPUTING =>
173             IF sign = '1' THEN
174               x_t <= x_t - shift_right(y_t, to_integer(counter));
175               y_t <= y_t + shift_right(x_t, to_integer(counter));
176               z_t <= z_t - signed(atan_out);
177             ELSE
178               x_t <= x_t + shift_right(y_t, to_integer(counter));
179               y_t <= y_t - shift_right(x_t, to_integer(counter));
180               z_t <= z_t + signed(atan_out);
181             END IF;
182
183             counter <= counter + 1;
184
```

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

Listing 3.1: `CORDIC.vhd`

3.2 Atan LUT

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

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

```
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 <=
      ↪  STD_LOGIC_VECTOR(to_signed(LUT(to_integer(unsigned(address))),
      ↪  24));
39  END PROCESS;
40
41  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 print 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 Coordinate;
53
54  -- Array of coordinates to test
55  CONSTANT Coordinates : CoordinateArray := (
56
57    (1.0, 1.0),
58    (10.0, 10.0),
59    (0.0, 0.0),
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 :
    ↪  INTEGER) RETURN real IS
70  BEGIN
71    RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
72  END FUNCTION;
73
74  -- function to print values in report
75  FUNCTION to_real(val : signed(N - 1 DOWNTO 0); fraction_bits :
    ↪  INTEGER) RETURN real IS
76  BEGIN
77    RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
78  END FUNCTION;
79
80  PROCEDURE echo(arg : IN STRING := "") IS
81  BEGIN
82    std.textio.write(std.textio.output, arg & LF); -- LF ensures a
      ↪  newline after each message
```

```
83    END PROCEDURE echo;
84  BEGIN
85    -- Instantiate the CORDIC component
86    cordic_inst : CORDIC
87    PORT MAP(
88      clk => clk,
89      rst => reset,
90      x => x,
91      y => y,
92      start => start,
93      rho => rho,
94      theta => theta,
95      valid => valid
96    );
97
98    -- todo fix behavior if cordic is not ready / does not work
99
100   clk <= (NOT(clk) AND run_simulation) AFTER T_clk / 2;
101
102   -- test process
103   test : PROCESS
104   BEGIN
105     reset <= '1';
106     start <= '0';
107     WAIT FOR 2 * T_clk;
108
109     reset <= '0';
110     FOR i IN 0 TO n_coordinates - 1 LOOP
111
112       IF valid = '0' THEN
113         WAIT UNTIL valid = '1';
114       END IF;
115
116       x <= STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).x * 2.0 **
          ↪  floating), N));
117       y <= STD_LOGIC_VECTOR(to_signed(INTEGER(Coordinates(i).y * 2.0 **
          ↪  floating), N));
118       start <= '1';
119
120       WAIT FOR 5 * T_clk;
121
122       start <= '0';
123       IF valid = '0' THEN
124         WAIT UNTIL valid = '1';
125       END IF;
126
127       WAIT FOR 1 * T_clk;
128       -- Osservazione del valore del modulo e della fase
129       echo("");
130       echo("Test " & INTEGER'image(i) & " X: " &
          ↪  real'image(Coordinates(i).x) & " Y: " &
          ↪  real'image(Coordinates(i).y));
131       echo("----------------------------------------");
```

```
132         echo("Module (Q8.8) = " & real'image(to_real(unsigned(rho), 8)));
133         echo("Phase  (Q3.13) = " & real'image(to_real(signed(theta),
            ↪  13)));
134         echo("----------------------------------------");
135
136         WAIT FOR 10 * T_clk;
137
138       END LOOP;
139
140       run_simulation <= '0';
141       WAIT;
142     END PROCESS;
143   END ARCHITECTURE;
```

Listing 4.1: `CORDIC_tb.vhd`

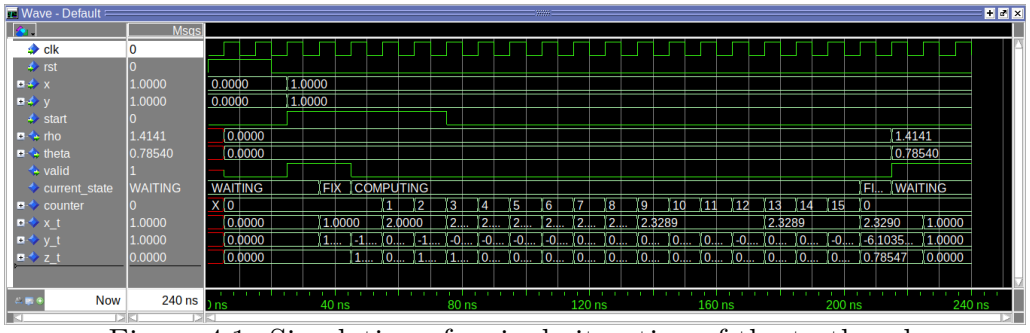


Figure 4.1: Simulation of a single iteration of the testbench

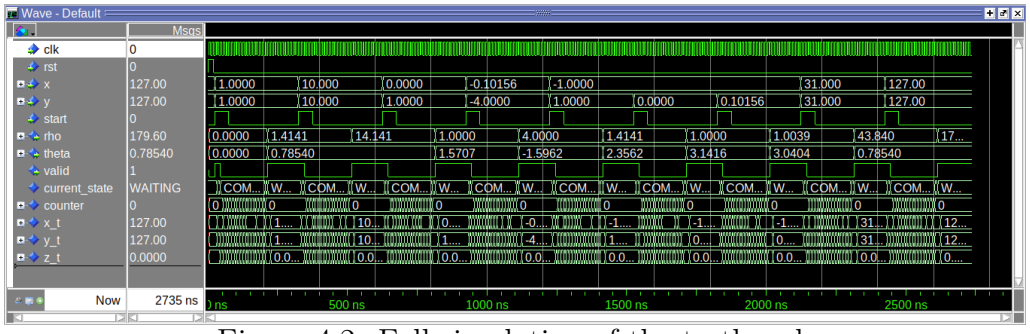


Figure 4.2: Full simulation of the testbench

4.2 Graph testbench

This testbench serves 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.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.math_real.ALL;
USE std.textio.ALL;

ENTITY GRAPH_tb IS
  GENERIC (
    N : POSITIVE := 16;
    floating : INTEGER := 8 -- (Q8.8)
  );
END ENTITY;

ARCHITECTURE Behavioral OF GRAPH_tb IS

  ------------------------------------------------------------
  -- CORDIC component
  ------------------------------------------------------------
  COMPONENT CORDIC
    PORT (
      clk : IN STD_LOGIC;
      rst : IN STD_LOGIC;
      x : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
      y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
      start : IN STD_LOGIC;
      rho : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
      theta : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
      valid : OUT STD_LOGIC
    );
  END COMPONENT;

  ------------------------------------------------------------
  -- Signals
  ------------------------------------------------------------
  SIGNAL clk : STD_LOGIC := '0';
  SIGNAL rst : STD_LOGIC := '0';
  SIGNAL x_in : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
  SIGNAL y_in : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
  SIGNAL start_in : STD_LOGIC := '0';
  SIGNAL rho_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
  SIGNAL theta_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
  SIGNAL valid_out : STD_LOGIC := '0';

  SIGNAL run_simulation : STD_LOGIC := '1';

  ------------------------------------------------------------
  -- Simulation parameters
  ------------------------------------------------------------

  CONSTANT T_clk : TIME := 20 ns; -- clock period = 20 ns
  CONSTANT SAMPLES : INTEGER := 512; -- samples per dimension

  ------------------------------------------------------------
```

```
54    -- Output file
55    ------------------------------------------------------------
56
57    FILE results_file : text OPEN write_mode IS
      ↪  "cordic_results_512x512.txt";
58
59    ------------------------------------------------------------
60    -- Conversion function from std_logic_vector to real
61    ------------------------------------------------------------
62
63    FUNCTION to_real(val : unsigned(N - 1 DOWNTO 0); fraction_bits :
      ↪  INTEGER) RETURN real IS
64    BEGIN
65      RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
66    END FUNCTION;
67
68    FUNCTION to_real(val : signed(N - 1 DOWNTO 0); fraction_bits :
      ↪  INTEGER) RETURN real IS
69    BEGIN
70      RETURN real(to_integer(val)) / 2.0 ** fraction_bits;
71    END FUNCTION;
72
73  BEGIN
74
75    ------------------------------------------------------------
76    -- CORDIC declaration
77    ------------------------------------------------------------
78    UUT : CORDIC
79    PORT MAP(
80      clk => clk,
81      rst => rst,
82      x => x_in,
83      y => y_in,
84      start => start_in,
85      rho => rho_out,
86      theta => theta_out,
87      valid => valid_out
88    );
89
90    ------------------------------------------------------------
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 + eral(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 * 2.0 **
            ↪  floating)), N));
132         y_in <= STD_LOGIC_VECTOR(to_signed(INTEGER(floor(real_y * 2.0 **
            ↪  floating)), N));
133
134         start_in <= '1';
135         WAIT FOR 2*T_clk;
136         start_in <= '0';
137
138         WAIT UNTIL valid_out = '1';
139         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 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:

$$\text{Module Error:} \quad \text{Max} = 0.004030\,\text{units}, \quad \text{Mean} = 0.001889\,\text{units}$$

$$\text{Phase Error:} \quad \text{Max} = 0.000154\,\text{rad}, \quad \text{Mean} = 0.000062\,\text{rad}$$

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

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

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 at most 1 bit error in the output.

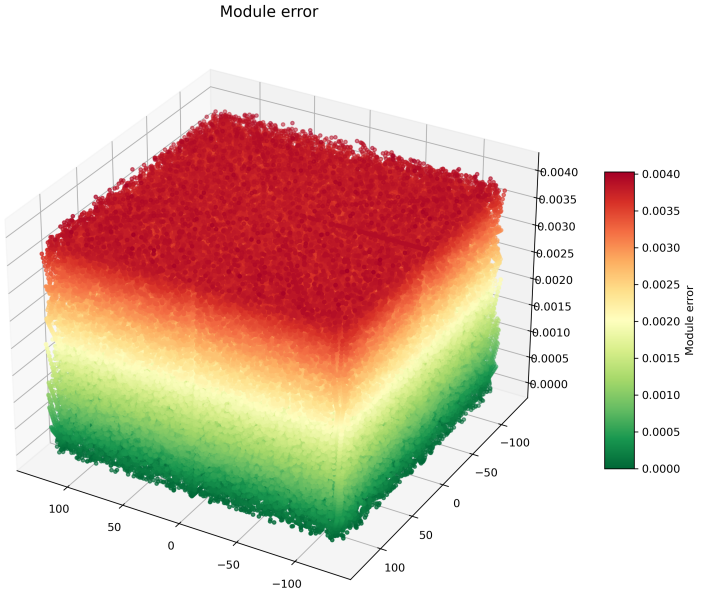


Figure 4.3: 3D plot of module error (ρ).

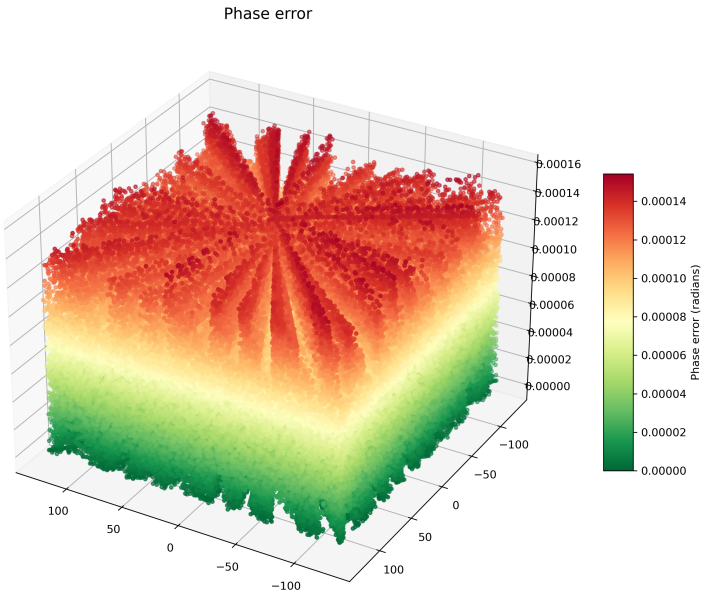


Figure 4.4: 3D plot of phase error (θ).

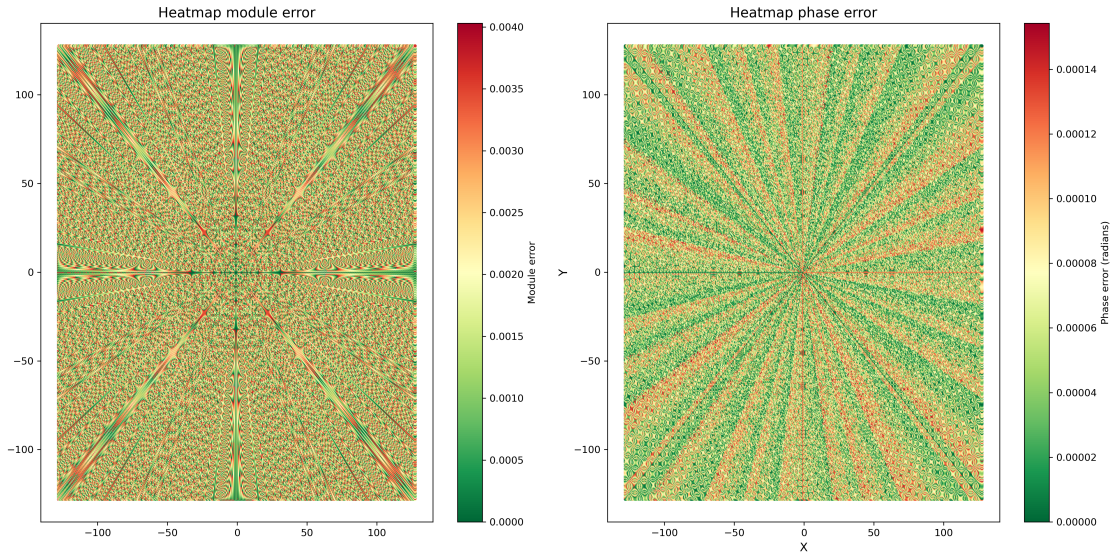


Figure 4.5: Heatmap for phase and module errors.

4.4 Corner cases

When the `valid` signal is set to 1, it indicates that the output data is valid and that the CORDIC module is ready to accept a new computation request. However, if CORDIC is currently performing a computation, as we can notice from the control part of the Code 3.1, any incoming `start` command will be ignored. This ensures that the ongoing computation is not interrupted. The new request will only be accepted once the CORDIC transitions back to the `WAITING` state, signaling with `valid = '1'` that the current computation has been completed.

Regarding the specific input case of the point $(0, 0)$, no modifications are required in the VHDL code. The output for this scenario is already defined as $\rho = 0$ and $\theta = 3.3139\,\text{rad}$. Since with zero modulus vectors ($\rho = 0$) the phase (θ) is irrelevant, this behavior is both mathematically consistent and functionally correct. This means the implementation handles such cases correctly without the need of extra logic.

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. 144 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. 802 nets (for connecting all the components)

The RTL Analysis generated the Elaborated Design shown in Figure 5.1, consistent with the expected structure of the system.

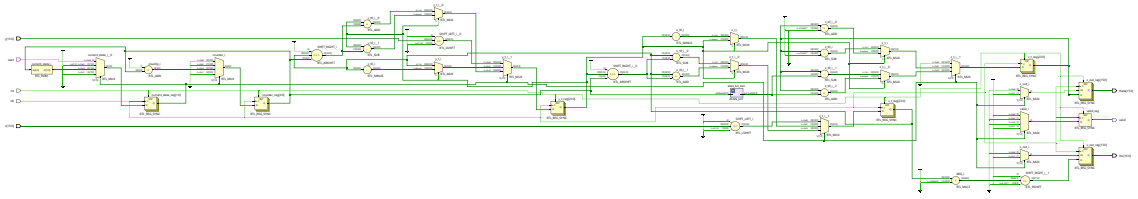


Figure 5.1: Elaborated RTL design.

5.3 Synthesis timing report

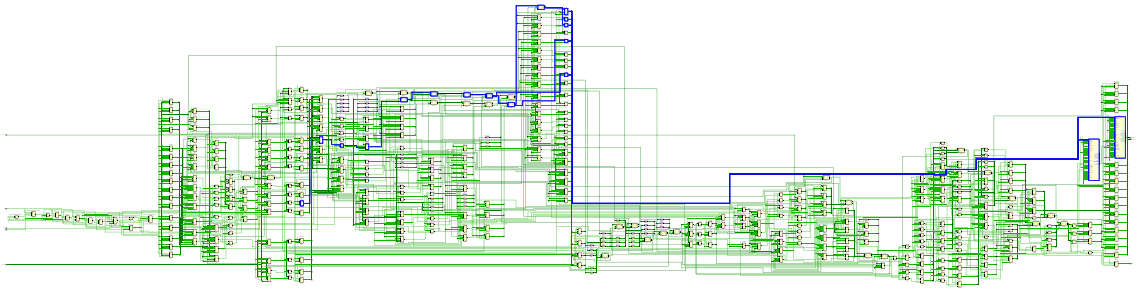


Figure 5.2: Figure showing the elaborated RTL design with the main critical paths for set-up-time-violation found in synthesis step highlighted in blue.

Name	Slack	Levels	Routes	From	To	Total Delay
Path 1	12.78	10	6	y_t_reg[14]/C	ARG/A[23]	6.77
Path 2	12.78	10	6	y_t_reg[14]/C	ARG___0/A[23]	6.77
Path 3	12.90	9	6	y_t_reg[14]/C	ARG/A[19]	6.66
Path 4	12.90	9	6	y_t_reg[14]/C	ARG___0/A[19]	6.66
Path 5	13.02	8	6	y_t_reg[14]/C	ARG/A[15]	6.54
Path 6	13.02	8	6	y_t_reg[14]/C	ARG___0/A[15]	6.54
Path 7	13.03	10	6	y_t_reg[14]/C	ARG/A[22]	6.53
Path 8	13.03	10	6	y_t_reg[14]/C	ARG___0/A[22]	6.53
Path 9	13.08	10	6	y_t_reg[14]/C	ARG/A[21]	6.47
Path 10	13.08	10	6	y_t_reg[14]/C	ARG___0/A[21]	6.47

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

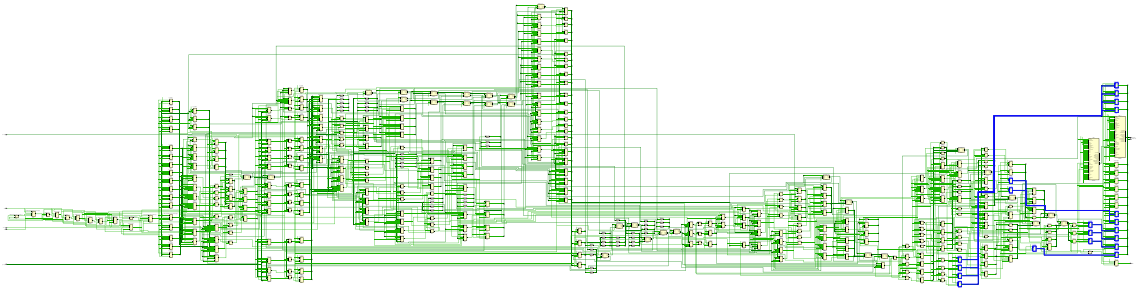


Figure 5.3: Figure showing the elaborated RTL design with the main critical paths for hold-time-violation found in synthesis step highlighted in blue.

Name	Slack	Levels	Routes	From	To	Total Delay
Path 11	0.28	0	1	z_t_reg[23]/C	z_out_reg[15]/D	0.30
Path 12	0.28	0	1	z_t_reg[8]/C	z_out_reg[0]/D	0.31
Path 13	0.28	0	1	z_t_reg[18]/C	z_out_reg[10]/D	0.31
Path 14	0.28	0	1	z_t_reg[19]/C	z_out_reg[11]/D	0.31
Path 15	0.28	0	1	z_t_reg[20]/C	z_out_reg[12]/D	0.31
Path 16	0.28	0	1	z_t_reg[21]/C	z_out_reg[13]/D	0.31
Path 17	0.28	0	1	z_t_reg[22]/C	z_out_reg[14]/D	0.31
Path 18	0.28	0	1	z_t_reg[9]/C	z_out_reg[1]/D	0.31
Path 19	0.28	0	1	z_t_reg[10]/C	z_out_reg[2]/D	0.31
Path 20	0.28	0	1	z_t_reg[11]/C	z_out_reg[3]/D	0.31

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

5.4 Implementation timing report

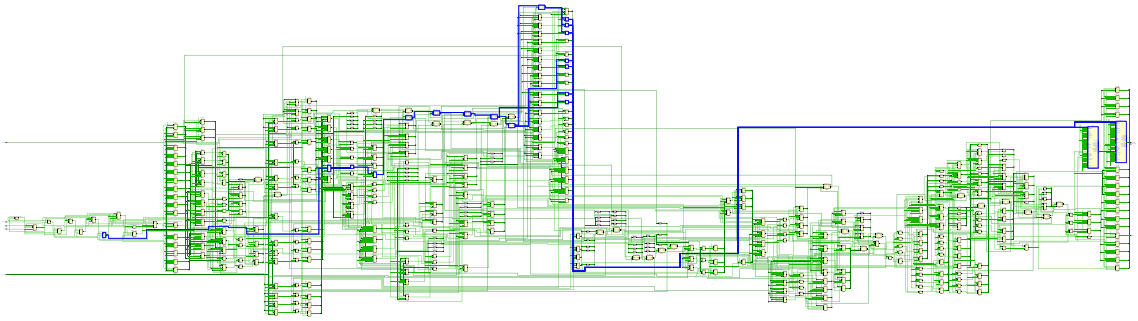


Figure 5.4: Figure showing the elaborated RTL design with the main critical paths for set-up-time-violation found during implementation step highlighted in blue.

Name	Slack	Levels	Routes	From	To	Total Delay
Path 1	10.19	10	6	counter_reg[2]/C	ARG/A[20]	9.16
Path 2	10.38	10	6	counter_reg[2]/C	ARG__0/A[20]	8.97
Path 3	10.43	10	6	counter_reg[2]/C	ARG__0/A[21]	9.13
Path 4	10.43	10	6	counter_reg[2]/C	ARG/A[22]	8.92
Path 5	10.57	8	6	counter_reg[2]/C	ARG__0/A[12]	8.78
Path 6	10.59	9	6	counter_reg[2]/C	ARG__0/A[17]	8.97
Path 7	10.60	9	6	counter_reg[2]/C	ARG/A[16]	8.75
Path 8	10.61	8	6	counter_reg[2]/C	ARG__0/A[13]	8.94
Path 9	10.61	10	6	counter_reg[2]/C	ARG/A[21]	8.94
Path 10	10.62	10	6	counter_reg[2]/C	ARG__0/A[22]	8.73

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

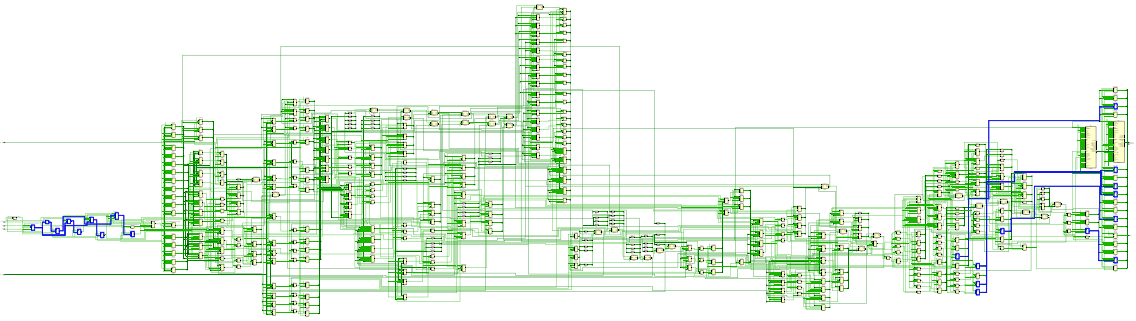


Figure 5.5: Figure showing the elaborated RTL design with the main critical paths for hold-time-violation found during implementation step highlighted in blue.

Name	Slack	Levels	From	To	Total Delay
Path 11	0.17	1	current_state_reg[1]/C	counter_reg[2]/D	0.33
Path 12	0.18	1	current_state_reg[1]/C	counter_reg[0]/D	0.32
Path 13	0.18	1	current_state_reg[1]/C	counter_reg[1]/D	0.32
Path 14	0.19	0	z_t_reg[22]/C	z_out_reg[14]/D	0.29
Path 15	0.22	0	z_t_reg[12]/C	z_out_reg[4]/D	0.29
Path 16	0.22	0	z_t_reg[14]/C	z_out_reg[6]/D	0.30
Path 17	0.23	0	z_t_reg[15]/C	z_out_reg[7]/D	0.26
Path 18	0.23	0	z_t_reg[10]/C	z_out_reg[2]/D	0.33
Path 19	0.24	1	counter_reg[0]/C	counter_reg[3]/D	0.38
Path 20	0.24	0	z_t_reg[18]/C	z_out_reg[10]/D	0.32

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

5.5 Utilization Report

Resource	Utilization (%)	Description
Slice LUTs	1.59%	Look-Up Tables used as logic
Slice Registers	0.27%	Registers used in the design
Slice	1.93%	Total slices utilized
LUT as Logic	1.59%	LUTs specifically used as logic
DSPs	2.50%	Digital Signal Processing blocks
Bonded IOB	0.00%	Bonded Input/Output Blocks
BUFGCTRL	0.00%	Global Clock Buffers

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

5.6 Power Report

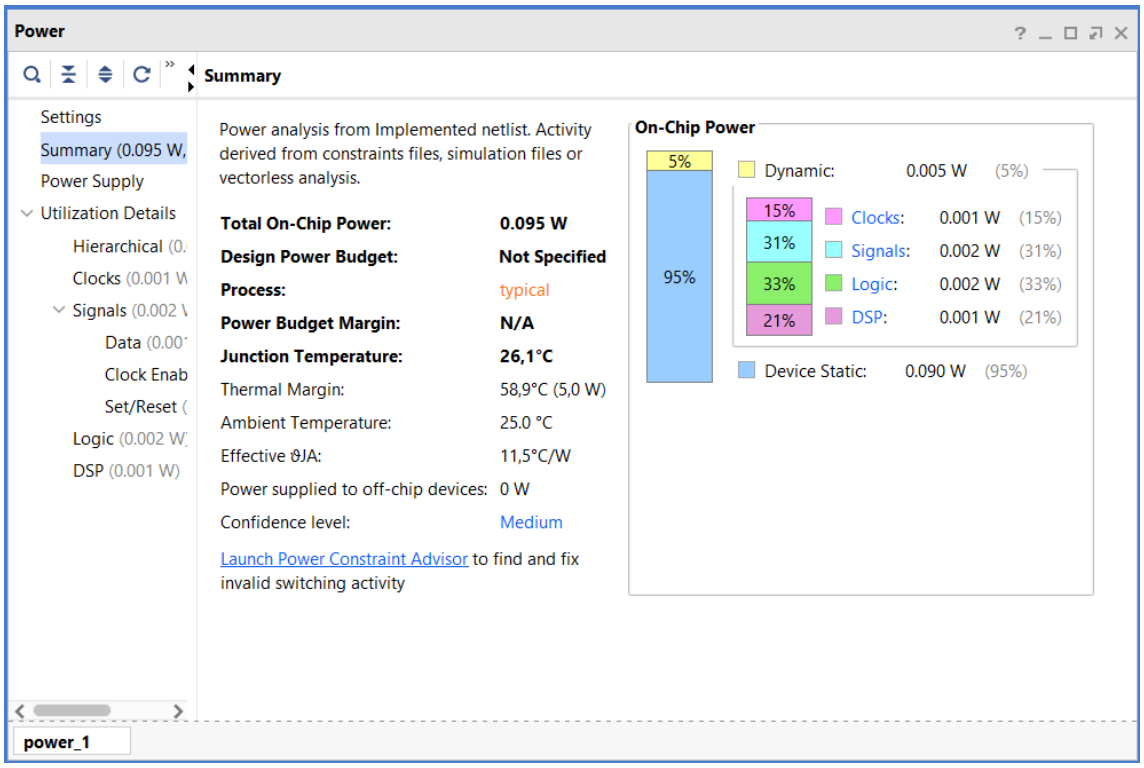


Figure 5.6: Vivado Power Report after implementation.

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.

5.7 Conclusions

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.