UNIVERSITÀ DI PISA

CORDIC: Cartesian to Polar Coordinate Transformation

Authors:

Andrea Di Matteo

Antonio Ciociola

ANNO ACCADEMICO 2024/2025

Contents

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 $(-99.7°, 99.7°)$. 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.

2.1 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

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

  GENERIC (
    M : POSITIVE := 24; -- internal representation
    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 : SIGNED(N - 1 DOWNTO 0) := to_signed(1304065748,
  ↪ N); -- 1/(Gain factor) multiplied by 2^N-1
  CONSTANT k : UNSIGNED(M - 1 DOWNTO 0) :=
  ↪ to_unsigned(INTEGER(0.6072528458 * (2 ** (M - 1))), M); --
  ↪ todo se in futuro vivado si lamenta usare M-2
  CONSTANT HALF_PI : SIGNED(M - 1 DOWNTO 0) :=
  ↪ to_signed(INTEGER(1.570796327 * (2 ** (M - 3))), M); -- todo
  ↪ documentare meglio il N-3

```

```
31    -- internal registers
32    SIGNAL x_t : SIGNED(M - 1 DOWNTO 0);
33    SIGNAL y_t : SIGNED(M - 1 DOWNTO 0);
34    SIGNAL z_t : SIGNED(M - 1 DOWNTO 0);
35
36    -- output registers
37    SIGNAL x_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
38    SIGNAL z_out : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
39
40    -- atan table address and output
41    SIGNAL address : STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
42    SIGNAL atan_out : STD_LOGIC_VECTOR(M - 1 DOWNTO 0);
43
44    SIGNAL sign : STD_LOGIC;
45
46    -- atan table
47    COMPONENT ATAN_LUT IS
48      PORT (
49        address : IN STD_LOGIC_VECTOR(ITER_BITS - 1 DOWNTO 0);
50        lut_out : OUT STD_LOGIC_VECTOR(M - 1 DOWNTO 0)
51      );
52    END COMPONENT;
53
54    -- state type and registers
55    TYPE state_t IS (WAITING, FIX, COMPUTING, FINISHED);
56    SIGNAL current_state : state_t;
57
58    -- counter for iterations
59    SIGNAL counter : UNSIGNED(ITER_BITS - 1 DOWNTO 0);
60
61  BEGIN
62    ------------------------------------------------
63    -- ISTANTIATE ALL COMPONENTS
64    ------------------------------------------------
65
66    -- output assignment
67    rho <= x_out;
68    theta <= z_out;
69
70    -- todo trovare nome migliore tipo d
71    -- sign bit
72    sign <= y_t(M - 1);
73    -- sign <= '0' WHEN y_t > 0 ELSE '1';
74
75    -- atan table
76    atan_lut_inst : ATAN_LUT
77    PORT MAP(
78      address => address,
79      lut_out => atan_out
80    );
81
```

```
82    -- atan table address
83    address <= STD_LOGIC_VECTOR(counter);
84
85    -- todo decidere se tenere o togliere gli assegnamenti stupidi
86    -- aggiustare meglio spiegazione e codice per i 29 bit di atan
87
88    ------------------------------------------------
89    -- CONTROL PART
90    ------------------------------------------------
91
92    control : PROCESS (clk, rst)
93    BEGIN
94      IF (rising_edge(clk)) THEN
95        IF rst = '1' THEN
96          current_state <= WAITING;
97        ELSE
98
99          CASE current_state IS
100           WHEN WAITING =>
101             IF start = '1' THEN
102               current_state <= FIX;
103             ELSE
104               current_state <= WAITING;
105             END IF;
106
107           WHEN FIX =>
108             current_state <= COMPUTING;
109
110           WHEN COMPUTING =>
111             IF counter = ITERATIONS - 1 THEN
112               current_state <= FINISHED;
113             ELSE
114               current_state <= COMPUTING;
115             END IF;
116
117           WHEN FINISHED =>
118             current_state <= WAITING;
119
120           WHEN OTHERS =>
121             current_state <= WAITING;
122
123         END CASE;
124       END IF;
125     END IF;
126   END PROCESS;
127
128   ------------------------------------------------
129   -- OPEATIONAL PART
130   ------------------------------------------------
131
132   OPEATIONAL : PROCESS (clk, rst)
```

```
133   BEGIN
134     IF (rising_edge(clk)) THEN
135       IF rst = '1' THEN
136         valid <= '0';
137         x_out <= (OTHERS => '0');
138         z_out <= (OTHERS => '0');
139         counter <= (OTHERS => '0');
140         x_t <= (OTHERS => '0');
141         y_t <= (OTHERS => '0');
142         z_t <= (OTHERS => '0');
143       ELSE
144
145         -- Default assignment
146         -- todo vedere se tenere o togliere
147         -- x_t <= (OTHERS => '-');
148         -- y_t <= (OTHERS => '-');
149         -- z_t <= (OTHERS => '-');
150         -- x_out <= (OTHERS => '-');
151         -- z_out <= (OTHERS => '-');
152         counter <= (OTHERS => '0');
153
154         CASE current_state IS
155           WHEN WAITING =>
156             x_t <= shift_left(resize(signed(x), M), (M - N - 2));
                ↪  -- Internal representation has 14 bits of fraction
157             y_t <= shift_left(resize(signed(y), M), (M - N - 2));
158             z_t <= to_signed(0, M);
159             valid <= '1';
160             x_out <= x_out;
161             z_out <= z_out;
162
163           WHEN FIX =>
164             IF sign = '1' THEN
165               x_t <= - y_t;
166               y_t <= x_t;
167               z_t <= z_t - HALF_PI;
168             ELSE
169               x_t <= y_t;
170               y_t <= - x_t;
171               z_t <= z_t + HALF_PI;
172             END IF;
173
174             valid <= '0';
175             counter <= (OTHERS => '0');
176
177           WHEN COMPUTING =>
178             IF sign = '1' THEN
179               -- x_t <= x_t - y_t/(2 ** to_integer(counter));
180               x_t <= x_t - shift_right(y_t, to_integer(counter));
181               -- y_t <= y_t + x_t/(2 ** to_integer(counter));
182               y_t <= y_t + shift_right(x_t, to_integer(counter));
```

```
183               z_t <= z_t - signed(atan_out);
184             ELSE
185               -- x_t <= x_t + y_t/(2 ** to_integer(counter));
186               x_t <= x_t + shift_right(y_t, to_integer(counter));
187               -- y_t <= y_t - x_t/(2 ** to_integer(counter));
188               y_t <= y_t - shift_right(x_t, to_integer(counter));
189               z_t <= z_t + signed(atan_out);
190             END IF;
191
192             IF counter < ITERATIONS - 1 THEN
193               counter <= counter + 1;
194             ELSE
195               counter <= (OTHERS => '0');
196             END IF;
197
198             valid <= '0';
199
200           WHEN FINISHED =>
201             x_out <= STD_LOGIC_VECTOR(resize(unsigned(x_t) * k / (2
                  ↪  ** (M - 1)), M)(M - 1 - 2 DOWNTO M - N - 2));
202             z_out <= STD_LOGIC_VECTOR(z_t(M - 1 DOWNTO M - N));
203
204             valid <= '1';
205
206         END CASE;
207       END IF;
208     END IF;
209   END PROCESS;
210 END ARCHITECTURE;
```

Listing 3.1: CORDIC.vhd

3.2 Atan LUT

```
1
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4  use IEEE.numeric_std.all;
5
6  ENTITY ATAN_LUT is
7    port (
8      address  : in  std_logic_vector(3 downto 0);
9      lut_out : out std_logic_vector(23 downto 0)
10   );
11 end entity;
12
13 architecture rtl of ATAN_LUT is
14
15   type LUT_t is array (natural range 0 to 15) of integer;
```

```
16    constant LUT: LUT_t := (
17    0 => 1647099,
18    1 => 972339,
19    2 => 513757,
20    3 => 260791,
21    4 => 130901,
22    5 => 65514,
23    6 => 32765,
24    7 => 16383,
25    8 => 8191,
26    9 => 4095,
27    10 => 2047,
28    11 => 1023,
29    12 => 511,
30    13 => 255,
31    14 => 127,
32    15 => 63
33      );
34
35  begin
36
37  PROCESS (address)
38  BEGIN
39    lut_out <=
      ↪  STD_LOGIC_VECTOR(to_signed(LUT(to_integer(unsigned(address))),
      ↪  24));
40  END PROCESS;
41
42  end architecture;
```

Listing 3.2: `ATAN_LUT.vhd`

4 Verification and testing

4.1 Testbench

```
LIBRARY IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.NUMERIC_STD.ALL;
    USE ieee.math_real.ALL;

ENTITY CORDIC_TB IS
    GENERIC (
        N : POSITIVE := 16;
        floating : INTEGER := 8
    );
END CORDIC_TB;

ARCHITECTURE Behavioral OF CORDIC_TB IS
    -- Component declaration for CORDIC

    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 for CORDIC inputs and outputs

    SIGNAL clk : STD_LOGIC := '0';
    SIGNAL reset : STD_LOGIC := '0';
    SIGNAL x : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL y : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');
    SIGNAL start : STD_LOGIC := '0';
    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, 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
69      -- function to print values in report
70      function to_real(val : unsigned(N-1 downto 0); fraction_bits :
        ↪ integer) return real is
71      begin
72          return  real(to_integer(val)) / 2.0**fraction_bits;
73      end function;
74
75      -- function to print values in report
76      function to_real(val : signed(N-1 downto 0); fraction_bits :
        ↪ integer) return real is
77          begin
78              return  real(to_integer(val)) / 2.0**fraction_bits;
79          end function;
80
81      procedure echo(arg : in string := "") is
82          begin
83              std.textio.write(std.textio.output, arg & LF);
84      end procedure echo;
```

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

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

Listing 4.1: CORDIC_tb.vhd

```
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;  -- Per scrivere su file di testo
6  
7  entity GRAPH_tb is
8      generic (
9          N : positive := 16;      -- larghezza del bus di ingresso
           ↪  e uscita
10         floating : integer := 8  -- numero di bit frazionari
           ↪  (Q8.8)
11     );
12 end entity;
13 
14 architecture Behavioral of GRAPH_tb is
15 
16 
       ↪  ------------------------------------------------------------------------
17     -- Dichiarazione componente CORDIC
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      -- Segnali locali
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 => '0');
38      signal y_in         : std_logic_vector(N - 1 downto 0) :=
        ↪ (others => '0');
39      signal start_in     : std_logic := '0';
40      signal rho_out      : std_logic_vector(N - 1 downto 0);
41      signal theta_out    : std_logic_vector(N - 1 downto 0);
42      signal valid_out    : std_logic := '0';
43
44      signal run_simulation : std_logic := '1';
45
46
        ↪ ------------------------------------------------------------------------
47      -- Parametri per la simulazione
48
        ↪ ------------------------------------------------------------------------
49      constant T_clk      : time := 20 ns;  -- periodo di clock = 20
        ↪ ns
50      constant SAMPLES    : integer := 512; -- numero di campioni in
        ↪ ogni dimensione
51
52
        ↪ ------------------------------------------------------------------------
53      -- File di output
54
        ↪ ------------------------------------------------------------------------
55      file results_file : text open write_mode is
        ↪ "cordic_results_512x512.txt";
56
57
        ↪ ------------------------------------------------------------------------
58      -- Funzione di conversione da std_logic_vector (signed) a
        ↪ real,
59      -- tenendo conto del numero di bit frazionario.
60
        ↪ ------------------------------------------------------------------------
61      function to_real(val : unsigned(N-1 downto 0); fraction_bits :
        ↪ integer) return real is
62      begin
63          return real(to_integer(val)) / 2.0**fraction_bits;
64      end function;
65
```

```
66      function to_real(val : signed(N-1 downto 0); fraction_bits :
        ↪  integer) return real is
67      begin
68          return real(to_integer(val)) / 2.0**fraction_bits;
69      end function;
70
71  begin
72
73
        ↪  ------------------------------------------------------------------------
74      -- Istanza del CORDIC
75
        ↪  ------------------------------------------------------------------------
76      UUT : CORDIC
77          port map(
78              clk   => clk,
79              rst   => rst,
80              x     => x_in,
81              y     => y_in,
82              start => start_in,
83              rho   => rho_out,
84              theta => theta_out,
85              valid => valid_out
86          );
87
88
        ↪  ------------------------------------------------------------------------
89      -- Generazione del clock
90
        ↪  ------------------------------------------------------------------------
91      clk_process : process
92      begin
93          while run_simulation = '1' loop
94              clk <= '1';
95              wait for T_clk/2;
96              clk <= '0';
97              wait for T_clk/2;
98          end loop;
99          wait;  -- fermo il processo quando run_simulation = '0'
100     end process clk_process;
101
102
        ↪  ------------------------------------------------------------------------
103     -- Processo di test
104
        ↪  ------------------------------------------------------------------------
105     test_process : process
106     variable L : line;
107     variable real_x, real_y : real;
108     variable mod_val, phase_val : real;
109     begin
```

```
110
        ↪ ------------------------------------------------------------------
111     -- Reset iniziale
112
        ↪ ------------------------------------------------------------------
113     rst <= '1';
114     start_in <= '0';
115     wait for 5*T_clk;
116     rst <= '0';
117     wait for 5*T_clk;
118
119
        ↪ ------------------------------------------------------------------
120     -- Loop su (i, j) per testare tutti i valori
121
        ↪ ------------------------------------------------------------------
122     for i in 0 to SAMPLES-1 loop
123         for j in 0 to SAMPLES-1 loop
124
125             -- Calcolo dei valori reali corrispondenti
126             real_x := -128.0 + real(i) * 256.0 / real(SAMPLES);
127             real_y := -128.0 + real(j) * 256.0 / real(SAMPLES);
128
129             -- Conversione in Q8.8 (signed)
130             x_in <=
                ↪  std_logic_vector(to_signed(integer(floor(real_x
                ↪  * 2.0**floating)), N));
131             y_in <=
                ↪  std_logic_vector(to_signed(integer(floor(real_y
                ↪  * 2.0**floating)), N));
132
133             -- Avvio CORDIC
134             start_in <= '1';
135             wait for T_clk;
136             start_in <= '0';
137
138             -- Attendiamo che valid_out passi a '1'
139             wait until valid_out = '1';
140             wait for 10 ns;
141
142             -- Acquisisco il risultato
143             mod_val   := to_real(unsigned(rho_out), floating);
                ↪  -- UQ8.8
144             phase_val := to_real(signed(theta_out), 13);
                ↪  -- Q3.13
145
146
                ↪ ------------------------------------------------------------------
147             -- Stampo sul file di testo: x, y, rho, theta
148
                ↪ ------------------------------------------------------------------
```

```
149                 -- Scrivi i valori separati da virgola
150                 write(L, real_x, RIGHT, 0, 6);
151                 write(L, string'(","));
152                 write(L, real_y, RIGHT, 0, 6);
153                 write(L, string'(","));
154                 write(L, mod_val, RIGHT, 0, 6);
155                 write(L, string'(","));
156                 write(L, phase_val, RIGHT, 0, 6);
157                 writeline(results_file, L);
158
159                 -- Piccola attesa fra un test e il successivo
160                 wait for 1*T_clk;
161
162             end loop;
163         end loop;
164
165
        ↪  ------------------------------------------------------------------------
166         -- Fine simulazione
167
        ↪  ------------------------------------------------------------------------
168         run_simulation <= '0';
169         wait;
170     end process test_process;
171
172 end Behavioral;
```

Listing 4.2: `GRAPH_tb.vhd`

4.2 Verification

4.3 Error verification

The implementation of the CORDIC algorithm in vectoring mode for Cartesian-to-polar conversion demonstrates exceptional accuracy in both the module (ρ) and phase (θ) outputs. Using a fixed-point representation of $Q8.8$ for ρ and $Q3.13$ for θ, the errors were summarized as follows:

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

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

The test was performed on 512×512 input points uniformly distributed within the range $(-128, 128)$ for both x and y. This range ensures a comprehensive evaluation of the algorithm over a high range of valid inputs while maintaining consistency with the precision of the fixed-point formats used. The fixed-point representation

employed provides a precision of 2^{-8} for $Q8.8$ (module) and 2^{-13} for $Q3.13$ (phase).

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

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

The observed errors are well within these precision limits, with all measured deltas smaller than the respective numerical precision of the formats. Any discrepancies between the input and output values are solely due to the conversion of these 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 no error in the output, demonstrating the precision and robustness of CORDIC algorithm thanks to the 24 bits internal logic.

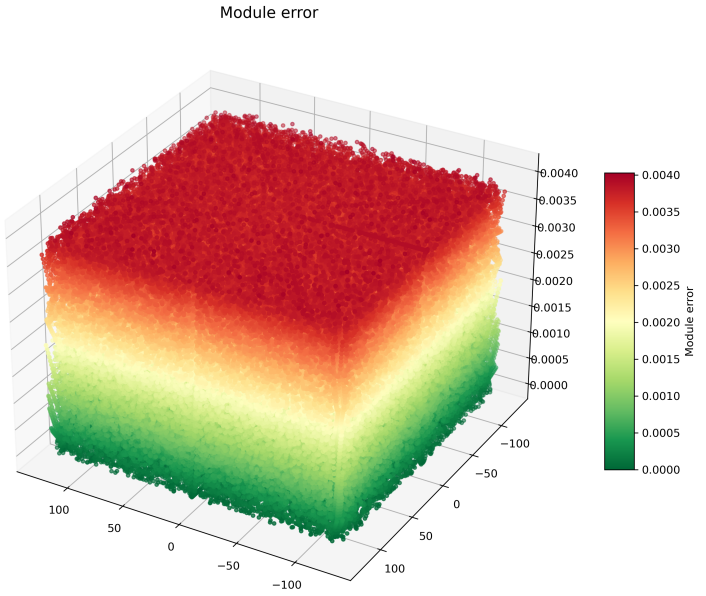

Figure 4.1: 3D plot of module error (ρ).

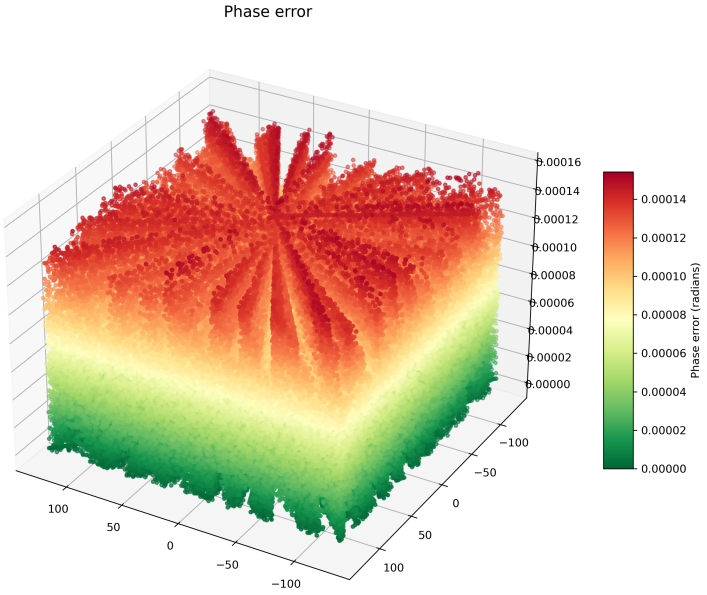

Figure 4.2: 3D plot of phase error (θ).

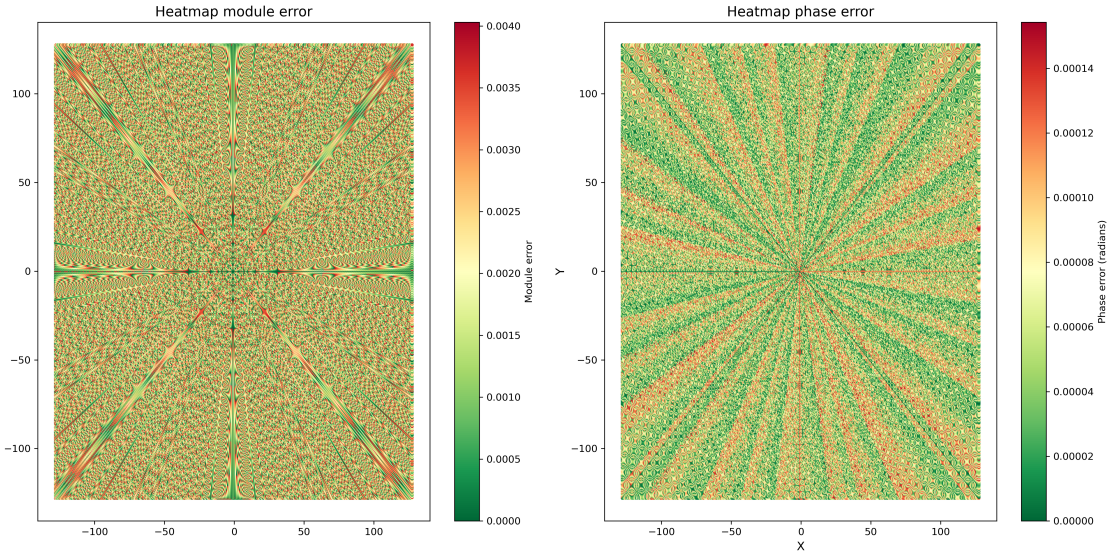

Figure 4.3: Heatmap for phase and module errors.

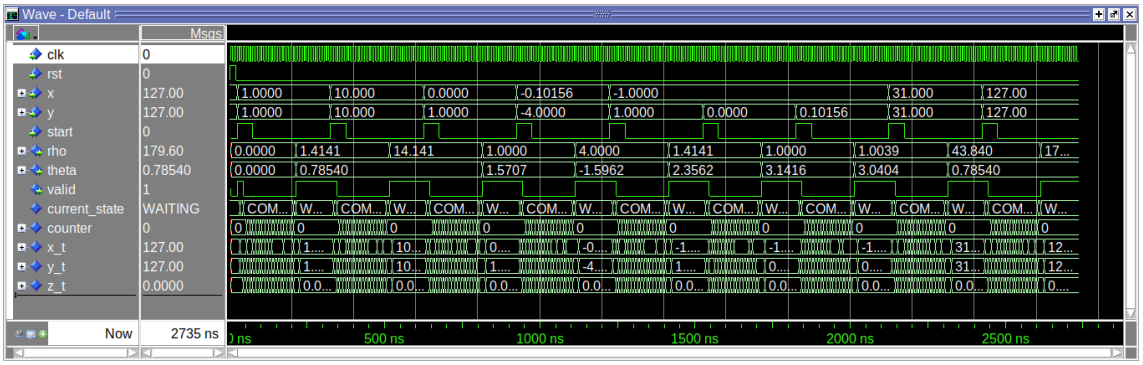

Figure 4.4: ...

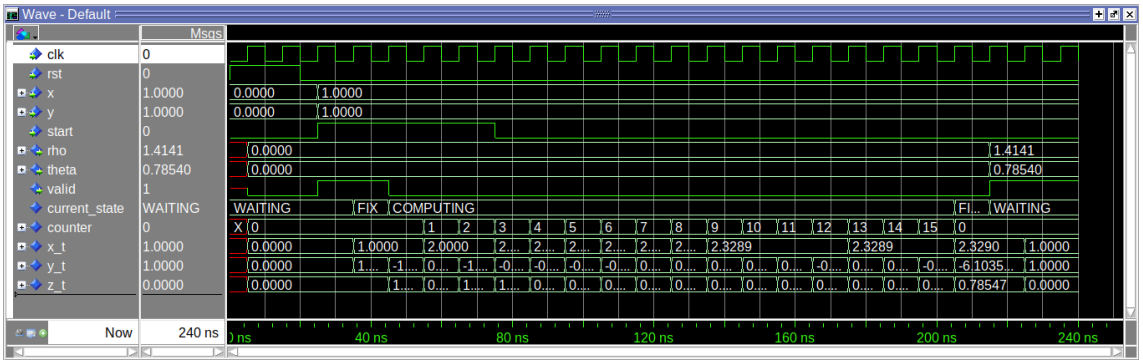

Figure 4.5: ...

5 Synthesis and Implementation

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)

5.3 RTL Elaboration

[**Trenitalia:** Or maybe just say that it's all loops even though it looks linear] The RTL Analysis generated the following Elaborated Design consistent with the expected structure of the system and the block diagram:

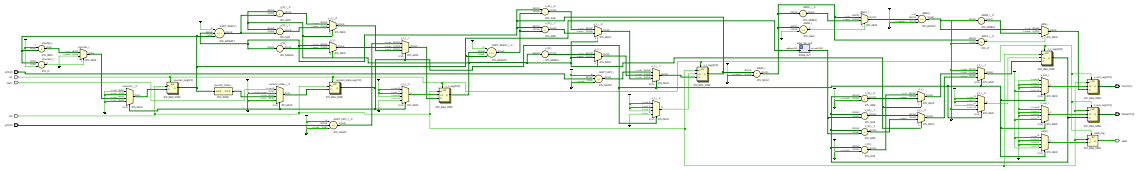

Figure 5.1: Elaborated RTL design.

5.4 Synthesis and Implementation

The Synthesis runs successfully without any errors, but a warning is reported regarding I/O constraints. The Timing Constraints are verified against a 20 ns clock period (50 MHz frequency), meeting the desired specifications. The Implementation also completes successfully; however, a warning (UCIO #1) is raised, indicating that all 68 logical ports lack user-assigned specific location constraints (LOC). This warning highlights potential risks such as I/O contention, board power or connectivity issues, performance degradation, signal integrity problems, or, in extreme cases, damage to the device or connected components. However, the assignment of specific pin locations was outside the scope of this project and was therefore not addressed.

6 Vivado results

6.1 Critical Path

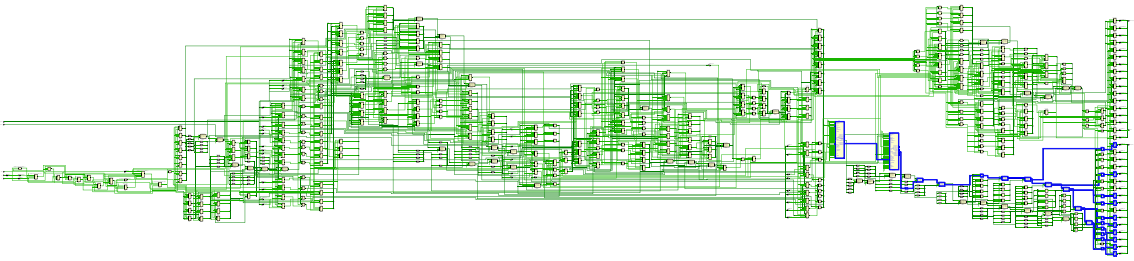

Figure 6.1: Figure showing the elaborated RTL design with the main critical paths for set-up-time-violation found in synthesis step highlighted in blue.

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

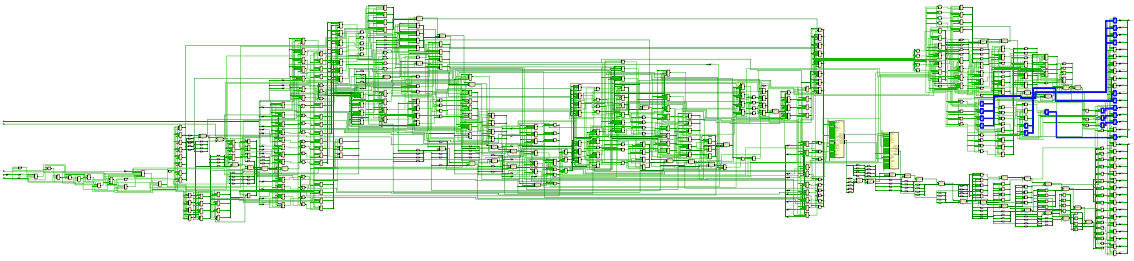

Figure 6.2: Figure showing the elaborated RTL design with the main critical paths for hold-time-violation found in synthesis step highlighted in blue.

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

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

[**Trenitalia:** need to comment a bit these results, for example the paths for setup-time-violation correspond exactly between synthesis and implementation, whereas

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

hold-time-violation are different, these means that place and route had a significant impact in determining these values.]

[**Trenitalia:** The DRC actually gives us 6 warnings, it tells us to pipeline the multiplier block in order to enhance performances and power consumption. But we adopted a CORDIC approach with loopbacks, that converges in 16 steps, 17 in the worst case when we have the fix step, inserting a register in between would cause higher delays in order to get the result.]

6.2 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 6.5: Resource utilization for the CORDIC design (only non-zero values shown)

6.3 Power Report

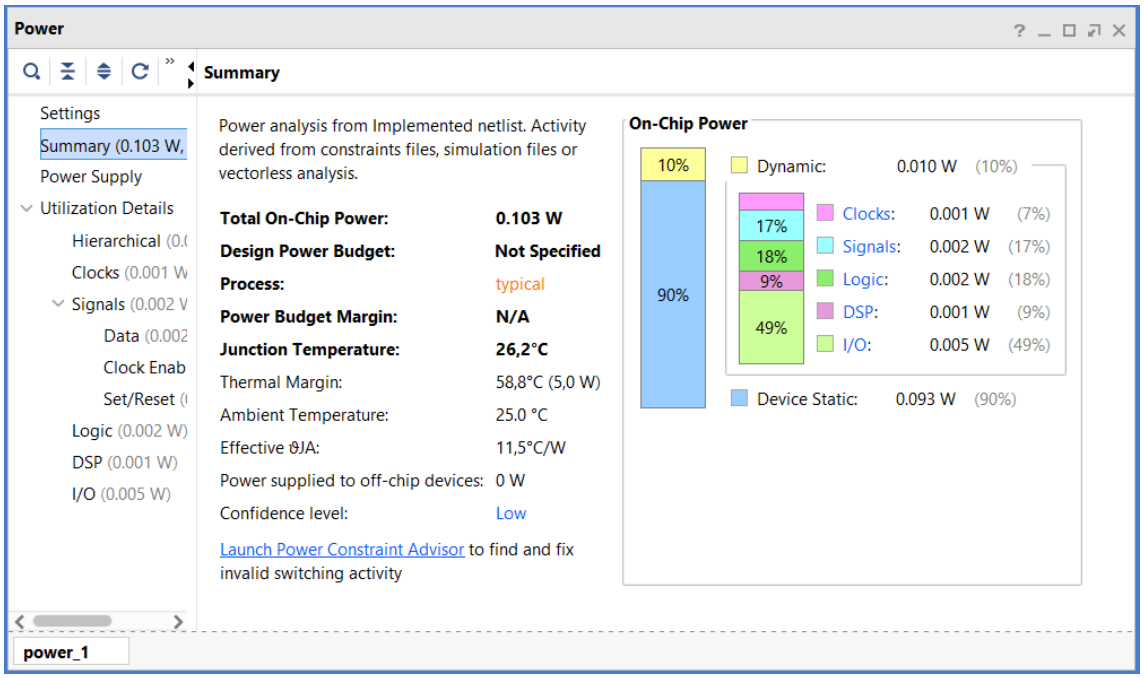

Figure 6.3: Vivado Power Report after implementation.

7 Final considerations