Digital System Design

Transformation from cartesian coordinates to polar coordinates using CORDIC

Authors:
Alessio Sanfratello
Andrea Beconcini
Francesco Mola

Supervisor:
Prof. Luca Fanucci

Academic Year 2014/2015

Contents

Chapter 1

Introduction

1.1 The CORDIC algorithm

The goal of this project is to design an integrated digital circuit which implements a converter from cartesian coordinates to polar ones, using the CORDIC algorithm.

CORDIC is an acronym for *COordinate Rotation DIgital Computer* and it was first described by Jack E. Volder in 1959.

CORDIC has two mode of operation: *rotation* and *vector*. The former mode takes the coordinates of an input vector plus an angle of rotation and returns the new coordinates after the rotation has been applied.

The vector mode can convert an input vector from cartesian to polar coordinates and its result depends on multiple iterations of the CORDIC rotation mode. Basically, vector mode rotates the input vector until its y coordinate become 0, so the modulus of the given vector is exactly equal to the value of the x coordinate of the rotated vector and its angle is equal to the opposite of the total rotation angle.

The rotation angle performed at i-th iteration α is:

$$\alpha_i = \arctan\left(\frac{1}{2^i}\right) \tag{1.1}$$

The reason for choosing such angles is that the rotated coordinates x_{i+1} y_{i+1} after i rotation become:

$$x_{i+1} = x_i - d_i y_i 2^{-i} \tag{1.2}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i} \tag{1.3}$$

Where d_i is equal to $+1$ if $y_i < 0$ and -1 otherwise.

The rotated coordinates can be computed using just sums and shift operation, as we can see in the previous equations.

The rotation performed at each stage i is equal to α_i and the total rotation is given by the sum of the previous contribution such that the total rotation performed at iteration $i + 1$ is

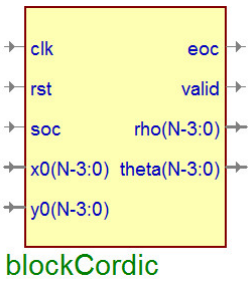
$$z_{i+1} = z_i + d_i \arctan\left(\frac{1}{2^i}\right) \tag{1.4}$$

The results of the arctan() function can be stored in a ROM since this algorithm has to apply this function to a limited set of value. Such set depends on the number of iteration we are interested in.

Note that the CORDIC algorithm do not produce a correct output if input is equal to the null vector. In fact such input generates a null vector at every iteration while the total rotation angle keeps increasing. Such a behavior produces a wrong output.

The aim of this work is to produce a component which implements the CORDIC vector mode.

1.2 Specification of requirements



Our network has five inputs and four outputs. *x0* and *y0* are the Cartesian coordinates which have to be transformed in polar coordinates.

After entering the values of these inputs the user has to put *soc* to the high value (*1*) to let the network starting the conversion. To obtain the result the user has to put again the input *soc* to the low value (*0*); when the output *eoc* goes to the high value the result can be read as modulus and angle respectively in the outputs *rho* and *theta*, the values obtained are given in fixed point with a notation Q8.24 (8 bits for the integer part, 24 bits for the decimal part).

The value of the angle θ is always correct and is given in radians, instead the value of the modulus ρ is correct if and only if the output *valid* is on the high value, otherwise it means that the value can't be represented on two's complement on the given number of bits.

The input *rst* is low triggered.

Chapter 2

Architecture

2.1 Input and output notation

Our circuit deals with 32-bit inputs and outputs in Q8.24 fixed point notation. This means our network uses 8 bits for the integer part and 24 bits for the decimal part of its input and output data.

This notation let us represent value in the interval $[-128; 127.9999999]$ and it gives us enough precision to represent 2^{-24} radians.

2.2 ROM and addresses generator

In order to compute the rotation angle at each iteration, we need the value of α_i for each possible iteration i.

As we said in section 1.1, we use a 64x32bit ROM to store the arctan() values.

The ROM is filled such that the first 25 location contain angles α like:

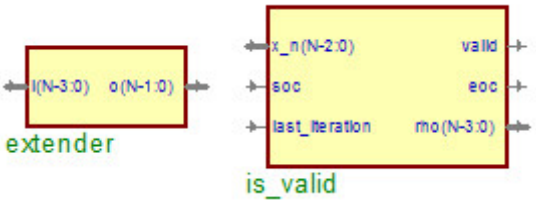
$$\alpha_i = \arctan\left(-\frac{1}{2^i}\right) \text{ with } i \in [0; 24] \tag{2.1}$$

The location from address 32 to 56 are filled with angles α like:

$$\alpha_i = \arctan\left(\frac{1}{2^i}\right) \text{ with } i \in [0; 24] \tag{2.2}$$

We can not go any further storing the arctan() values because we do not have enough precision to represent them. We choose to fill the remaining locations with zeros. Doing this, we do not modify the value of the output angle when the conversion is complete.

2.3 Extender and is_valid components



As we discussed before, our data representation puts some limits on the range of signals we can manage. In particular there are some possible input values whose polar transformation can not be represented with Q8.24 notation.

In order to be sure whether an output is congruent with the input or not, we implement a valid mechanism. Such mechanism takes advantages of 2 main components: *Extender* and *Is_valid*. The cordic algorithm returns the modulus multiplied by a factor of $A_n\sqrt{2}$ where:

$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}} \tag{2.3}$$

so, A_n is constant with the number of iteration we perform. The only thing we have to do to provide the modulus is to divide it by A_n.

When an input is provided to our network, we extend it with 2 bits more. Such extension takes into account the multiplication factor A_n and we have no representation problem.

Once we have computed the result, we use the *Is_valid* to reset the validity bit if the output is not representable.

The *Is_valid* is also in charge of managing the correct sequence of control bits to drive our network. It reports the fact that the computation is over by setting the *eoc* bit. Anyway the handshake protocol requires the *soc* bit to be reset before setting *eoc*.

2.4 Counter

Since our circuit performs multiple iterations to reach the expected output values, we need a counter to stop the computation.

We can stop the algorithm at the 25th iteration since our data format does not support precision provided with longer computation, in other words, the rotation angles after the 25th iteration are too small to be represented with Q8.24.

2.5 Soc, Eoc, Rst

We provide the user some bits to control the working status of our cordic circuit. In particular, we provide the following:

- Soc (Start Of Conversion): This tells the circuit to take datas from the input bus (x_0 and y_0) and start the conversion.

- Eoc (End Of Conversion): this is an output bit set by the cordic circuit when the conversion is over and the user can read the result on the output bus (rho and theta);
- Rst (ReSeT): this input bit is low triggered and it must be set to 0 when we want to ensure cordic circuit to be in a consistent state, so it must be set to 0 every time the user want to perform the first conversion.

In order to drive the circuit correctly, the user must firstly set *rst* to 0. Then, when the *rst* is set to 1 again, *soc* must assume 1 to notify the circuit that data on x_0 and y_0 are valid and the conversion can start.

2.6 Starter

Our circuit includes a specific submodule, called *Starter*, in order to implement the correct starting process. The *Starter* module checks the value of *soc* and *rst*, and it provides a reset signal to all internal registers. In particular it triggers a reset when:

- *rst* goes to 0;
- *soc* goes to 1 (in this case it is kept to 0 for just 10 ns).

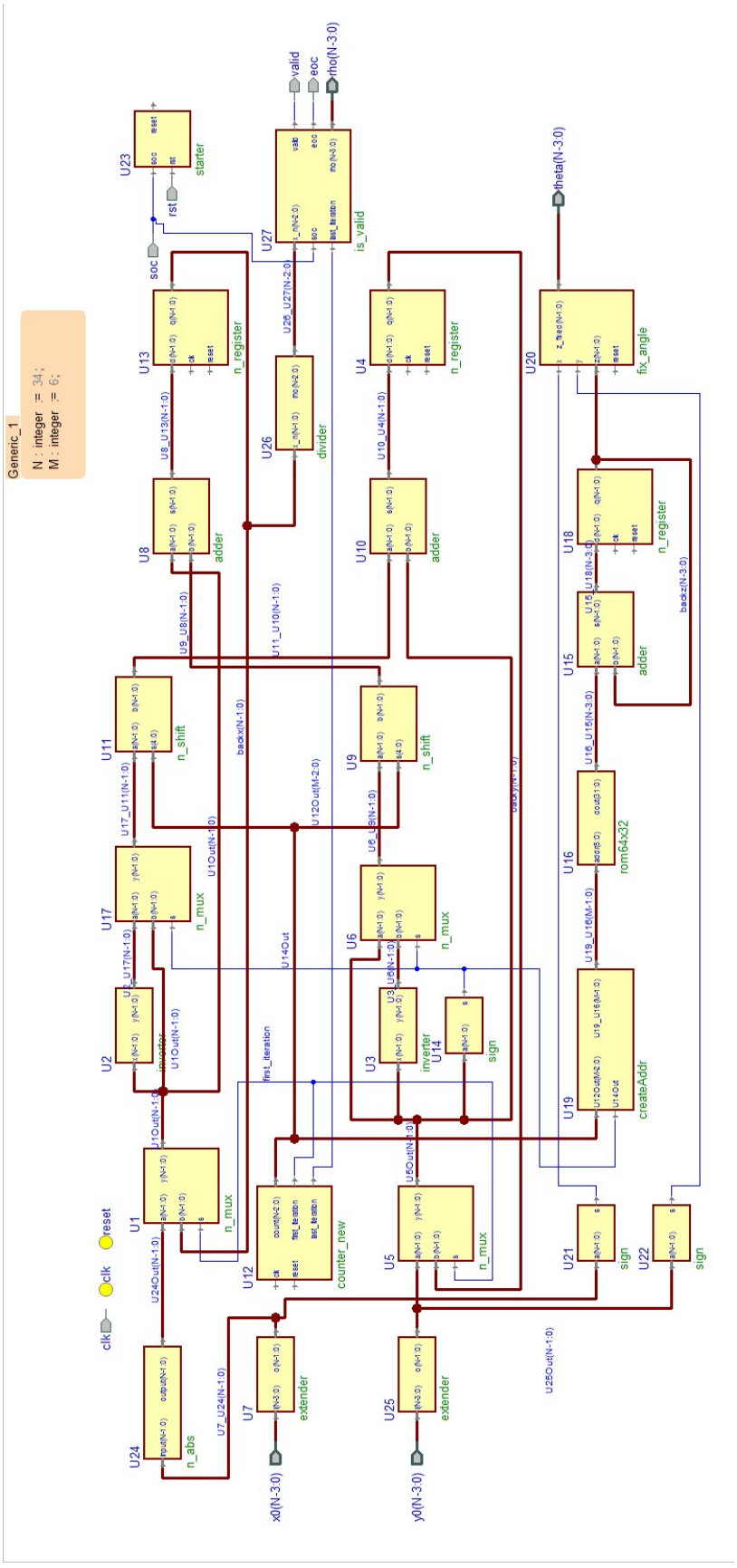


Figure 2.1: General Overview

Chapter 3

Testing

3.1 Design of testbench

We designed two different kind of tests to check the correctness of both outputs rho and theta respectively. The former ensure the modulus and validity bit while the latter validates the output angle.

To perform those we used a VHDL testbench file which can take an external input file containing input and expected values and compare cordic results with the expected ones.

The VHDL testbench and code for generating its input files can be found at section A.1

Listing 3.1: A snippet from the testbench

```
begin
        file_open(file_TEST, "test_cordic.txt", read_mode);

        rst <= '0';
        wait for 20 ns;
        rst <= '1';
        wait for 20 ns;

        while not endfile(file_TEST) loop
                readline(file_TEST, v_ILINE);
                hread(v_ILINE, v_x0); --reading first input
                hread(v_ILINE, v_y0); --read second input
                hread(v_ILINE, v_theta); --reading expected theta
                hread(v_ILINE, v_rho); --reading expected rho
                read(v_ILINE, v_valid); --reading valid bit

                x0 <= v_x0;
                y0 <= v_y0;

                wait for 10 ns;
                soc <= '1' after 0 ns, '0' after 10 ns;
                wait for 20 ns;
```

```
                while eoc = '0' loop
                        wait for 50 ns;
                end loop;

                write(v_OLINE, "rho ");
                hwrite(v_OLINE, rho);
                write(v_OLINE, " ");
                hwrite(v_OLINE, v_rho);
                write(v_OLINE, " theta");
                write(v_OLINE, " ");
                hwrite(v_OLINE, theta);
                write(v_OLINE, " ");
                hwrite(v_OLINE, v_theta);
                write(v_OLINE, " valid bit ");
                write(v_OLINE, valid);
                writeline(OUTPUT, v_OLINE);

                h_rho := rho(N-1 downto 16);
                h_theta := theta(N-1 downto 16);

                assert valid = v_valid
                        report "Error on valid bit"
                        severity ERROR;

                if v_valid = '1' then
                        assert h_rho = v_rho(N-1 downto 16)
                                report "Error on rho"
                                severity ERROR;

                        assert h_theta = v_theta(N-1 downto 16)
                                report "Error on theta"
                                severity ERROR;
                end if;
        end loop;

        file_close(file_TEST);
        write(v_OLINE,"Fine del test");
        writeline(OUTPUT,v_OLINE);
        wait;
end process;
```

3.1.1 Testing modulus and validity bit

In this test we feed our circuit with coordinates of points which belongs to the function:

$$y = x \tag{3.1}$$

We start from point $P = (-127, -127)$ and move toward the first quadrant with

a step equal to 1 until we reach point $P = (127, 127)$.

Then we concentrate our tests near the point which produce a *rho* overflow. Such overflow occurs around $P = (90.5, 90.5)$, so we concentrate our test around that point with a step equal to 0.01 and 0.001.

3.1.2 Testing angle

After having tested the modulus output (*rho*), we check the output angle correctness. We start from a point whose polar coordinates are $\rho = 127$ and $\theta = 0$, then we move on a circumference with an angular step equal to $\frac{\pi}{24}$.

3.2 Waveform Examples

In the following figures you can see some examples of how signals evolve in time for a couple of different inputs.

In particular in figure 3.1 we pass $[1, 1]$ as input and after 25 iterations we obtain congruent result.

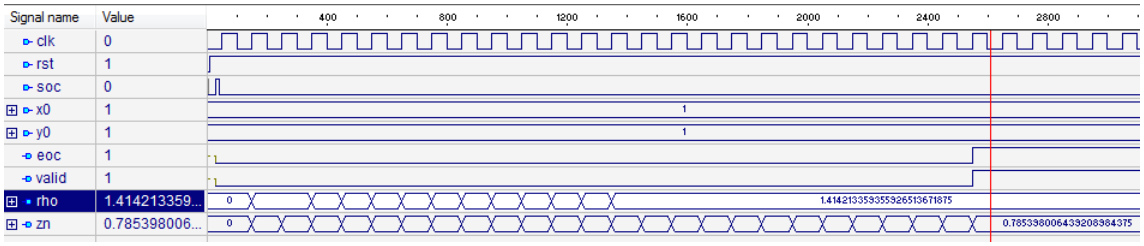


Figure 3.1: Waveform example of valid output

If instead we use $[120, 126]$ as input, we obtain an invalid result, therefore valid flag has been set to 0.

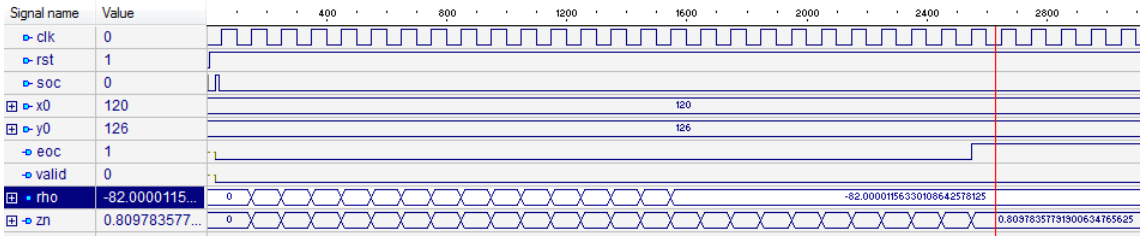


Figure 3.2: Waveform example of invalid output

Chapter 4

Synthesis using Xilinx ISE Tool

At the end we decide to synthesize our project using the Xilinx ISE Tool. We decide to set a balanced design goal which automatically sets the right process properties to achieve this optimization. After that we start our synthesis. After fixing some warnings not shown by the simulation tool, we get its result shown in the Device Utilization Summary (see figure 4.1).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of 4 input LUTs	90	9,312	1%
Number of occupied Slices	46	4,656	1%
Number of Slices containing only related logic	46	46	100%
Number of Slices containing unrelated logic	0	46	0%
Total Number of 4 input LUTs	90	9,312	1%
Number of bonded IOBs	66	232	28%
Number of MULT18X18SIOs	4	20	20%
Average Fanout of Non-Clock Nets	1.07		

Figure 4.1: Utilization Summary provided by Xilinx tool

In figure 4.2 we can see the timing report with the calculation of the maximum path delay which gives us the relative time constraint that we have to respect in order to obtain the maximum possible frequency to drive our network.

```
=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 7.306ns (frequency: 136.874MHz)
  Total number of paths / destination ports: 3825344 / 88
-------------------------------------------------------------------------
```

Figure 4.2: Timing report provided by Xilinx tool

Appendix A

Source code

A.1 Generation of test data using C++

Listing A.1: Test-data generation

```
#include <stdio.h>
#include <math.h>
#include <stdint.h>

#define TWO_TO_31 0x80000000
#define MUL 16777216
#define N_ENTRIES 32
#define DIV 24

struct Polar {
    int32_t rho;
    int32_t theta;
    bool valid;
};

int32_t rom[2*N_ENTRIES];

Polar cordic(int32_t x0, int32_t y0) {
    int64_t x = 0, y = 0;
    int32_t z = 0;

    // Extension to "34" bits (using int64_t)
    x = x0; y = y0;

    // Moving to first or fourth quadrant
    x = (x < 0) ? -x : x;

    for (int i=0; i<=25; i++) {
        int64_t x_old = x, y_old = y;
        int32_t z_old = z;

        // Computing d
```

```
        int d = (y < 0) ? 1 : -1;

        // Computing (inverter + mux + n_shift + adder)
        x = x_old - ((d * y_old) >> i);
        y = y_old + ((d * x_old) >> i);

        // reading ROM entry
        int entry = (d > 0) ? i + N_ENTRIES : i;

        // Computing z using ROM's value
        z = z_old + rom[entry];
    }

    Polar result = {0,0,false}; // Preparing result

    // Computing rho
    const int32_t A_n_inverted = 0x009B74ED;
    int64_t temp_rho = ((x >> 2) * A_n_inverted) >> 22;
    result.valid = (temp_rho >= TWO_TO_31) ? false : true;
    result.rho = (int32_t) temp_rho;

    // Computing theta
    const int32_t PI = 0x03243F6B;
    if (x0 < 0) {
        if (y0 >= 0) {
            result.theta = z + PI;
        } else {
            result.theta = z - PI;
        }
    } else {
        result.theta = -z;
    }

    return result;
}

void gen_rom() {
    // Computing negative values for atan(2^-i)
    for (int i = 0; i < N_ENTRIES; i++) {
        double a = (-1)*atan(pow(0.5, (double) i));
        rom[i] = (i < 25) ? (int32_t) (round(a*MUL)) : 0;
    }

    // Computing positive values for atan(2^-i)
    for (int i = 0; i < N_ENTRIES; i++) {
        double a = atan(pow(0.5, (double) i));
        rom[i+N_ENTRIES] = (int32_t) (round(a*MUL));
    }
}
```

```
int main() {
    gen_rom(); // Filling ROM entries

    // Bisector test [-127,127]
    for (int k=-127; k<=127; k++) {
        Polar p = cordic(k<<24,k<<24);
        printf("%08X %08X %08X %08X %d\n", k<<24, k<<24, p.theta, p.rho, p.
            valid);
    }

    // Bisector test [90,91]
    for (int k=0; k<=100; k++) {
        double x = 90.0 + k/100.0;
        double y = 90.0 + k/100.0;
        int32_t x_int = (int32_t) (round(x*MUL));
        int32_t y_int = (int32_t) (round(y*MUL));
        Polar p = cordic(x_int,y_int);
        printf("%08X %08X %08X %08X %d\n", x_int, y_int, p.theta, p.rho, p.
            valid);
    }

    // Bisector test [90.49,90.52]
    for (int k=0; k<=30; k++) {
        double x = 90.49 + k/1000.0;
        double y = 90.49 + k/1000.0;
        int32_t x_int = (int32_t) (round(x*MUL));
        int32_t y_int = (int32_t) (round(y*MUL));
        Polar p = cordic(x_int,y_int);
        printf("%08X %08X %08X %08X %d\n", x_int, y_int, p.theta, p.rho, p.
            valid);
    }

    // Angles test
    const double r = 127;
    for (int k=0; k<2*DIV; k++) {
        double a = k * M_PI / DIV;
        int32_t x_int = round(r*cos(a)*MUL);
        int32_t y_int = round(r*sin(a)*MUL);
        Polar p = cordic(x_int,y_int);
        printf("%08X %08X %08X %08X %d \n", x_int, y_int, p.theta, p.rho, p
            .valid);
    }

    return 0;
}
```