UNIVERSITÀ DI PISA

Electronics and Communication Systems

CORDIC for VHDL Arctangent implementation

Mione Francesco, Lossi Leonardo

Anno Accademico 2018-2019

Contents

1. Introduction

In this project we have designed a digital circuit that achieves the *arctangent* computation of the ratio between two integer numbers using `CORDIC (Coordinate Rotation Digital Computer) algorithm`. The top level scheme of the circuit that we had to implement is shown in Figure 1.1

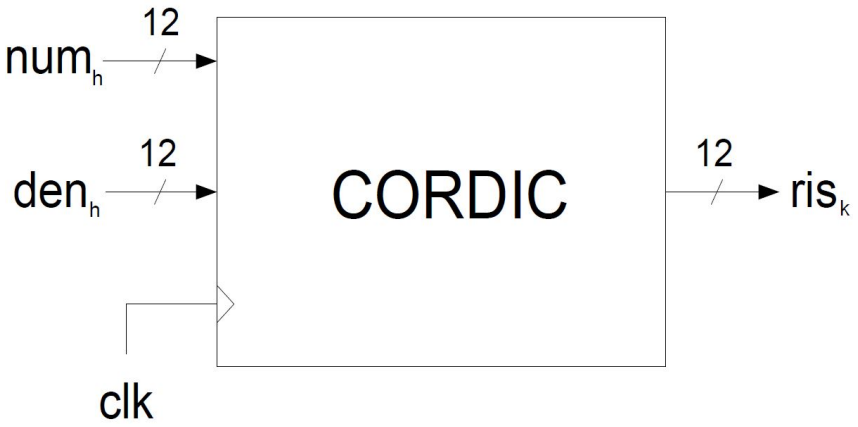


Figure 1.1: top-level scheme of Cordic Arctan circuit.

2. CORDIC algorithm

CORDIC algorithm, also known in the literature as Volder algorithm or Digit-by-Digit method, is an algorithm that allows to compute efficiently trigonometric and hyperbolic functions without hardware multipliers, using only adders, shifters, inverters and multiplexers. From the point of view of the number of logic gates we are not so far from the one that is necessary to implement an hardware multiplier, this difference becomes significant with the increasing of the precision of the result: in fact while the number of adders needed to implement the CORDIC algorithm increases linearly with the number of bits, the number of logic gates to implement an hardware multiplier increases quadratically with the number of bits. With the increasing of the number of logical gates obviously we have a larger area occupancy, which is not always allowed, and a higher power consumption. It is necessary to underline that, when it is possible, the use of multipliers allows to run faster algorithm. These are the main reasons that make possible to discriminate between the two solutions.
In this project we want to estimate the arctangent value of the ratio between two integer numbers, named num_h and den_h, i.e. mathematically:

$$ris_k = arctan\left(\frac{num_h}{den_h}\right)$$

The algorithm is based on complex numbers theory: supposing to map on the Gauss plan the two numbers num_h and den_h respectively as the real and the imaginary part of a complex number we will have:

$$z = a + jb$$

$$where \begin{cases} a = den_h \\ b = num_h \end{cases}$$

If we represent this number in the Polar Complex plan in the Modulus and argument form, under specific conditions the value of the angle we would like to compute is the argument of the complex number, i.e. the angle of vector z with the positive real axis of the Gauss plane. In particular:

$$iff\ ((a > 0)\ and\ (b \neq 0)) \implies \alpha = arctan\left(\frac{b}{a}\right)$$

In the cases not treated in the above condition, the application of the entire algorithm is neither applicable nor convenient, in fact we have:

- $if(a < 0)$ the algorithm does not work, but the result we would like to obtain is the same we reach using -den_h and -num_h instead of the direct integer inputs, before applying the algorithm we must compute the opposites of both the inputs.

- $if(a = 0)$ the complex number lies on the imaginary axis, the choice of the right result is driven by the sign of b (the num_h input) as follows:
  - $if(b < 0) \implies \alpha = -90°$
  - $if(b > 0) \implies \alpha = +90°$
- If *(b = 0)* the complex number lies on the real axis, and we can directly compute the result as:

$$if(b = 0) \implies \alpha = 0°$$

From the complex numbers theory we know that we can also represent the complex number in the so called trigonometric form (in the following formula we name as ρ the modulus and as φ the argument of the complex number z):

$$z = \rho(cos(\varphi) + jsin(\varphi))$$

Using Euler?s formula this can be written as:

$$z = \rho e^{j\varphi}$$

The property of complex numbers on which the algorithm is based is the folowing: given two complex numbers z_1 and z_2 if we compute the product between them we obtain

$$z' = z_1 * z_2 = (\rho_1 * \rho_2)e^{j(\varphi_1+\varphi_2)}$$

So the resulting complex number z' is such that its modulus is the result of the multiplication of the modulus of the two complex factors and its argument is the sum of the arguments of the two factors. Consequently we can conclude that multiplying a complex number z_1 (i.e. a point on the Gauss plan, identified by a vector) by another one z_2 with modulus ρ and argument φ we obtain an amplification of the modulus by a factor of ρ and a rotation of the vector z of an angle φ (the positive direction for the rotation of the angle is the counterclockwise one).

In practice CORDIC algorithm iteratively rotates the complex number by multiplying it by complex number whose argument becomes smaller iteration by iteration; in the canonical form the algorithm stops when the rotated vector lies on the real axis, in the specification of this project was suggested to iterate at least 8 times so the result produced by the circuit could be not accurate, i.e. the rotated vector after 8 iteration could not lie on the real axis.

The result of the algorithm is given by the summation of the arguments of the factor that have been chosen iteration by iteration (so they are both positive and negative contributes) until the rotated vector is arrived on the x-axis.

In Figure 2.1 we can see a graphical representation of the behavior of the algorithm, in particular, the rotation by both positive and negative angles of the input vector.

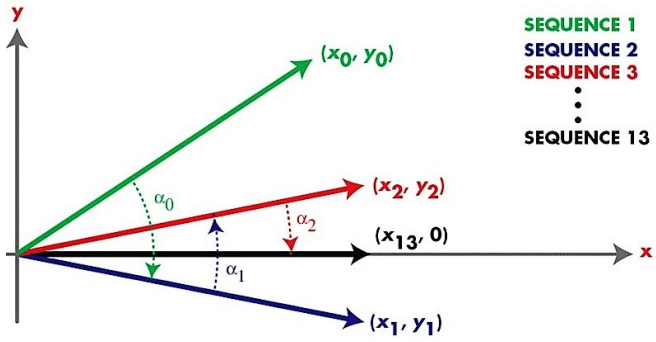


Figure 2.1: *CORDIC* functioning representation.

To implement CORDIC algorithm in a "smart" way we have to deal with simple complex numbers with the property of having argument always smaller then the one used in the previous iteration; to reach this goal to rotate the initial complex number (given by den_h e num_h inputs) we used complex numbers obtained as follows:

$$z_{2i} = (1 + jk_i) \quad where \begin{cases} k_i = 2^{-i} \\ i = \ number\ of\ the\ current\ iteration \end{cases}$$

This form for factors has been chosen because computing the product of them with a generic complex number $z = a + jb$, we obtain:

- *first case:*
  in case ($b_i < 0$) *we must compute the product :* $(a_i + jb_i) * (1 + jk_i) = (a_i - b_i k_i) + j(b_i + a_i k_i)$.
  So, the values to give in input at the following step are:

$$\begin{cases} a_{i+1} = (a_i - b_i k_i) \\ b_{i+1} = (b_i + a_i k_i) \end{cases}$$

- *second case:*
  in case ($b_i > 0$) *we must compute the product :* $(a_i + jb_i) * (1 - jk_i) = (a_i + b_i k_i) + j(b_i - a_i k_i)$.
  So, the values to give in input at the following step are:

$$\begin{cases} a_{i+1} = (a_i + b_i k_i) \\ b_{i+1} = (b_i - a_i k_i) \end{cases}$$

Since $k = 2^{-i}$ and since numbers are represented by bit strings (number in base 2) the two products $b_i k_i$ and $a_i k_i$ can be computed simply right shifting respectively b_i and a_i of i positions.
Moreover using this form for the factors we can also reach the desired property of smaller arguments iteration by iteration, in fact we have:

- *1st iteration:* $i = 0 \rightarrow z_{2,0} = (1 \pm j2^{-0}) = (1 \pm j)$
  - $if(b < 0) \rightarrow z_{2,0} = (1 + j) = 1,414213562\ \exp(j\frac{\pi}{4}) \implies rotation\ of\ 45°$
  - $if(b > 0) \rightarrow z_{2,0} = (1 - j) = 1,414213562\ \exp(-j\frac{\pi}{4}) \implies rotation\ of\ -45°$

- 2^{nd} *iteration:* $i = 1 \rightarrow z_{2,1} = (1 \pm j2^{-1}) = (1 \pm 0,5j)$
  - $if(b < 0) \rightarrow z_{2,1} = (1+0,5j) = 1,118033989\ \exp(j26,56505118) \implies rotation\ of\ 26,56505118°$
  - $if(b > 0) \rightarrow z_{2,1} = (1-0,5j) = 1,118033989\ \exp(-j26,56505118) \implies rotation\ of\ -26,56505118°$
- 3^{rd} *iteration:* $i = 2 \rightarrow z_{2,2} = (1 \pm j2^{-2}) = (1 \pm 0,25j)$
  - $if(b < 0) \rightarrow z_{2,2} = (1+0,25j) = 1,030776406\ \exp(j14,03624347) \implies rotation\ of\ 14,03624347°$
  - $if(b > 0) \rightarrow z_{2,2} = (1-0,25j) = 1,030776406\ \exp(-j14,03624347) \implies rotation\ of\ -14,03624347°$
- 4^{th} *iteration:* $i = 3 \rightarrow z_{2,3} = (1 \pm j2^{-3}) = (1 \pm 0,125j)$
  - $if(b < 0) \rightarrow z_{2,3} = (1+0,125j) = 1,007782219\ \exp(j7,125016349) \implies rotation\ of\ 7,125016349°$
  - $if(b > 0) \rightarrow z_{2,3} = (1-0,125j) = 1,007782219\ \exp(-j7,125016349) \implies rotation\ of\ -7,125016349°$
- 5^{th} *iteration:* $i = 4 \rightarrow z_{2,4} = (1 \pm j2^{-4}) = (1 \pm 0,0625j)$
  - $if(b < 0) \rightarrow z_{2,4} = (1+0,0625j) = 1,001951221\ \exp(j3,576334375) \implies rotation\ of\ 3,576334375°$
  - $if(b > 0) \rightarrow z_{2,4} = (1-0,0625j) = 1,001951221\ \exp(-j3,576334375) \implies rotation\ of\ -3,576334375°$
- 6^{th} *iteration:* $i = 5 \rightarrow z_{2,5} = (1 \pm j2^{-5}) = (1 \pm 0,03125j)$
  - $if(b < 0) \rightarrow z_{2,5} = (1+0,03125j) = 1,000488162\ \exp(j1,789910608) \implies rotation\ of\ 3,1,789910608°$
  - $if(b > 0) \rightarrow z_{2,5} = (1-0,03125j) = 1,000488162\ \exp(-j1,789910608) \implies rotation\ of\ -1,789910608°$
- 7^{th} *iteration:* $i = 6 \rightarrow z_{2,6} = (1 \pm j2^{-6}) = (1 \pm 0,015625j)$
  - $if(b < 0) \rightarrow z_{2,6} = (1+0,015625j) = 1,000122063\ \exp(j0,89517371) \implies rotation\ of\ 0,89517371°$
  - $if(b > 0) \rightarrow z_{2,6} = (1-0,015625j) = 1,000122063\ \exp(-j0,89517371) \implies rotation\ of\ -0,89517371°$
- 8^{th} *iteration:* $i = 7 \rightarrow z_{2,7} = (1 \pm j2^{-7}) = (1 \pm 0,0078125j)$
  - $if(b < 0) \rightarrow z_{2,7} = (1+0,0078125j) = 1,000030517\ \exp(j0,44761417) \implies rotation\ of\ 3,1,789910608°$
  - $if(b > 0) \rightarrow z_{2,7} = (1-0,0078125j) = 1,000030517\ \exp(-j0,44761417) \implies rotation\ of\ -0,44761417°$

Obviously since we are using a fixed value for rotations at each iteration we can have a visible drowback: it can be possible to get, at a given iteration *'i'*, a rotated vector very close to the semipositive axis of the Gauss plan but with $Im \neq 0$ (stop condition not verified), in this case it is not guaranteed that the following steps will return a more accurate angle then the one at the iteration *'i'* we have mentioned.

2.1 Possible HW implementation for CORDIC algorithm

The first and immediate idea we got when we start thinking on the implementation of this algorithm was the most obvious one: to design a feed-forward combinatorial sequence of steps. Soon we realized that with this kind of architecture we would have pricipally 2 drowbacks: the first was that the paths along the circuit would be too much long and so we could not drive our circuit at satisfying clock frequency, the second was that the principal advantage in using CORDIC (the smaller number of logic gates w.r.t. multipliers) was not exploited.

To overcome the first of the two drawbacks we thought to implement a sort of pipeline, inserting pipeline registers between consequent steps; we realized that it was only a partial solution of our problems because it solved only the clock frequency problem, but it did it increasing the latency of the circuit and maintaining the second drowback.

The third and final solution lies on the fact that, as we underlined in the previous chapter, CORDIC algorithm is an iterative algorithm and during its iterations the same operations are repeated every time on the data produced at the previous step. So we thought to roll the previous architecture adding a feedback with a register in our circuit in order to use a smaller number of logic gates by means of the reuse of them. The conclusion was that using this third architecture we could save lot of area occupancy and we could drive our circuit with a sufficiently high clock frequency, so we decided to use this one to implement as we will see in the following chapter.

3. Architecture implementation

3.1 Block diagram with a top-down approach

As we have seen in the first chapter (in Figure 3.1) the top level scheme of the circuit we would like to design is the following:

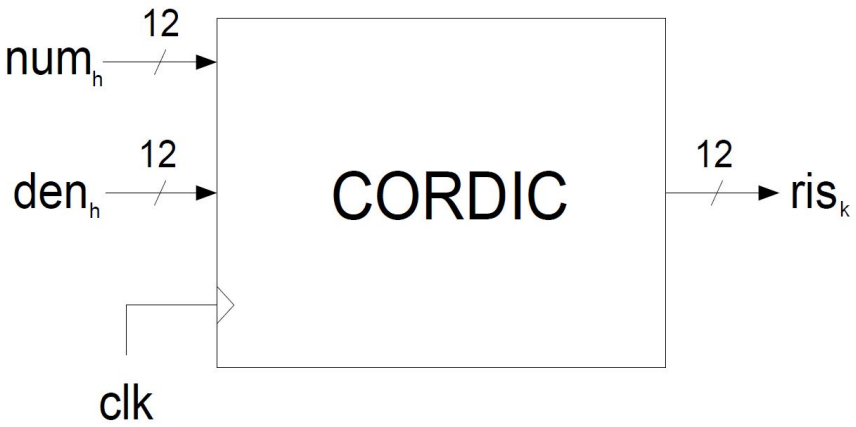


Figure 3.1: top-level scheme of Cordic Arctan circuit.

Where:

- num_h is an integer number in one's complement representation;
- den_h is an integer number in one's complement representation;
- ris_k is a real number in fixed point representation, the integer part of the representation is a one's complement representation and the fractional one has a fixed number of digits.

The numbers of bits of the links were suggested in the project specifications, so the most significant bits of num_h and den_h representations represent the signs of the respective represented numbers('0' for positive integers, '1' for negative ones); regarding ris_k we assumed a integer part of the fixed point representation made by 8 bits, where the most significant one represents the sign, and a fractional part composed by the remaining 4 bits.

3.1.1 Block-diagram of CORDIC arctangent circuit

Now we can focus on the level-below block diagram, this is shown in the following figure (Figure 3.2).

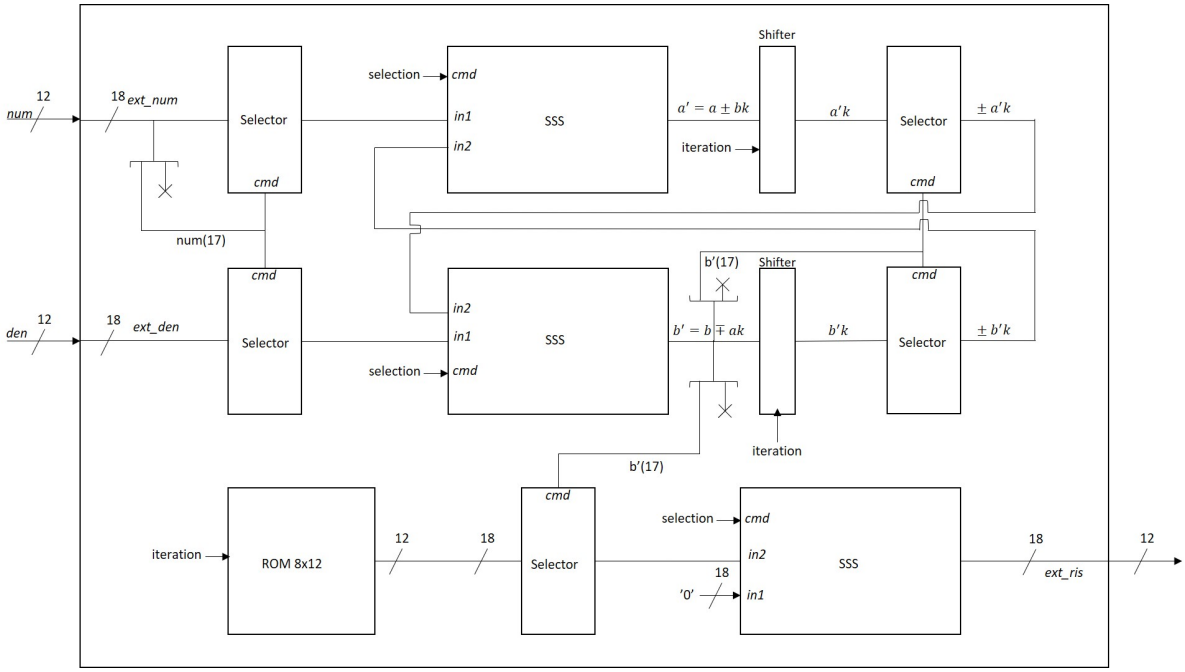


Figure 3.2: block-diagram of Cordic arctangent circuit.

As we can see in the above figure we can identify two main parts of the circuit. The first one is the high part of the circuit, it is composed by the components directly connected to the inputs num_h and den_h and rolled to produce the feedback for the reuse of components we have mentioned in Section 2.1. This part of the circuit computes at each step of the algorithm the product $(a_i \pm b_i) * (1 \mp j2^{-i})$ and allows to check whether the algorithm has converged or not (this last check is done by simply seeing if $b' = 0$). The second main part of the circuit is the one directly connected to the output ris_k so it is the part in charge to produce the result summing the values of arctangents of the factors $(1 \pm j2^{-i})$ coming from the ROM component at each iteration until the stop condition is verified.
It has to be stressed that both the inputs and the output of the circuit are extended while travelling in the circuit, the reasons are 2:

1. the first one is that computations rely on multiplications by negative powers of two, this is done by right shifting the representations and it is easy to understand that sooner or later there will be an *underflow condition* with loss of information. To overcome this problem we could add a certain number of less significant bits but since from specifications we are designing a circuit that should for a maximum of 8 iterations,to completely avoid the problem we should add 21 bits to the representation and this is not feasible. After some attemps we realised that adding 4 less significant bits we could reach an error below one degree, so we decided we could be satisfied with this level of accuracy.

2. the second reason is that, as we can see form the introduction to CORDIC algorithm in Chapter 2, the real part of the complex number z_{i+1} resulting from the multiplication $z_{i+1} = (a_i \pm b_i) * (1 \mp j2^{-i})$ has an increasing real part and a imaginary part that is converging to 0. Due to this fact the real part of this complex number can lead to a *overflow condition*; considering the worst ideal case in which both num_h and den_h assume the maximum representable value in one's complement on 12 bit, i.e. they both assume value 2047, and the imaginary part stays constant to that maximum value along all the iterations, we will have: in the first iteration the value of the real part will double, in the following ones at each step the addend will be one half that the addend at the previous step. At this point is easy to understand that, even in this ideal case which gives us an *upper bound* of the real part, this one will never reach its *quadruple* and so we can simply add 2 most significant bits to the representation to completely avoid this problem.

This is why we have chosen to extend num_h, den_h and ris_k within the circuit on 18-bit representations. From the Figure 3.2 we can also see that there are some missing *control* parts of the circuit like:

- the logic to manage the base cases $num_h = 0$ or $den_h = 0$ in which the result can be immediately produced without any computation;

- the logic to manage internal signals like *selection* and *iteration*, whose function will be explained in the next section;

- the logic to drive correctly the *in2* input of the *SSS* components: as we will see from the description of the *SSS* components in the following section after the *multiplexer* there is an *adder* before the *register* (the *dff*), it is necessary to keep the *in2* input with value '0' when $selection = 0$ to make sure that the computations will begin with the correct value.

3.1.2 Components structure

In this subsection we will see how the components of Figure 3.2 are implemented starting from basic components as *inverters, multiplexers, adders and D-positive edge-triggered flip-flops.*

3.1.2.1 Selector

As we can see from Figure 3.3 the *selector* component simply negates the input in case the command variable *cmd* is set, otherwise it lets pass the input without any alterations. In our implementation we exploit this component first to select the proper sign for the inputs num_h and den_h in the case in wich num_h (i.e. a) is negative, second to select the proper second addend for the two higher *SSS* components during the iterations and third to select the proper sign for the value of the argument of the second factor which goes in input as *in2* of the lower *SSS* component as a contribute in the accumulation that will produce the result.

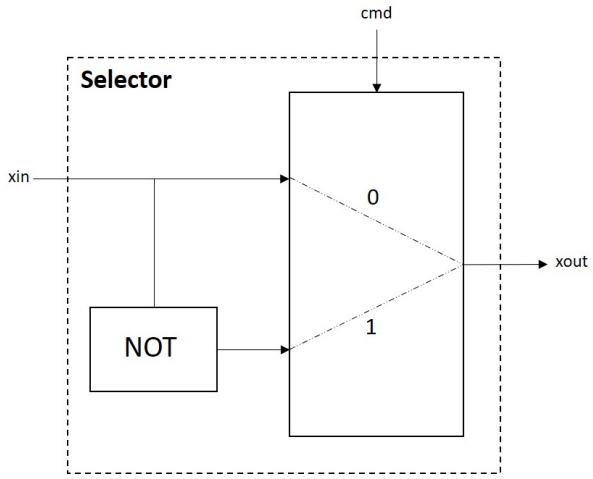


Figure 3.3: *selector*: internal design of the component.

3.1.2.2 SSS (Select-Sum-Save) Component

As we can see from Figure 3.4 the *SSS* component starts with a multiplexer, driven by the input *cmd*, which allows to select the proper first addend for the *adder* component. In fact, we have discriminate between the first iteration, in which the multiplexer must select the input *in1* as first addend, and the subsequent ones, in which we must take as first addend the feedback coming from the *dff* component. In the block diagram of Figure 3.2 we can see that we have used 3 istances of this components: the two higher istances to compute and save the results of $a'_{i+1} = a_i \pm bk_i = a_i \pm b2^{-i}$ and $b'_{i+1} = b_i \mp ak_i = b_i \mp a2^{-i}$, the lower istance has a fixed value $in1 = 0$ and computes the accumulation of the arguments of the factors by which the original vector is rotated.

As we will see in the VHDL description the *adder* is a *one's complement adder* in which the eventual *carry out* is sent in feedback as the *carry in* for the following step. This configuration as we will see leads to a correct behaviour in terms of results produced by the circuit but also produces some warnings during the synthetisis phase in the Vivado Tool.

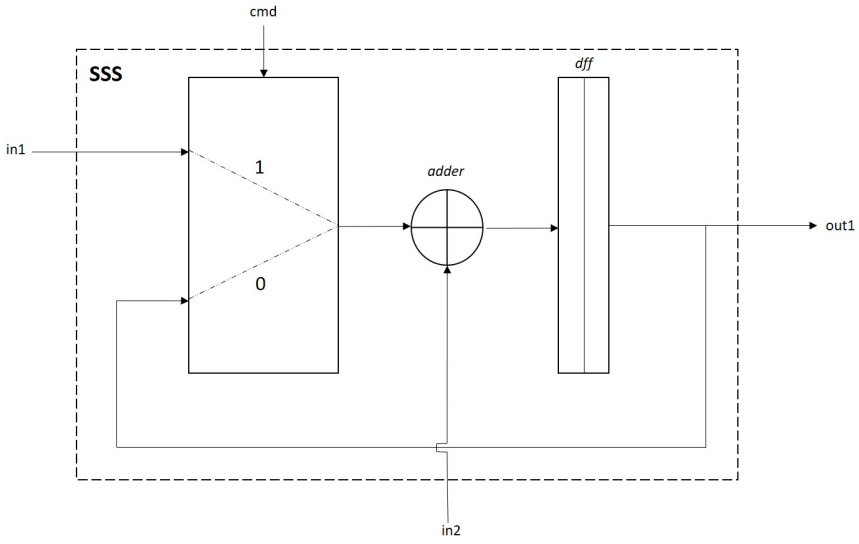


Figure 3.4: *SSS*: internal design of the component.

3.1.2.3 Shifter

The *Shifter* component is simply a *barrel shifter circuit* that is a digital circuit to execute bit-shift operations by a number of location given in input by the *cmd* input, usually implemented by a sequence of feed-forward multiplexers. Since from specifications we designed a circuit which executes a maximum of 8 iteration, in our project we had to deal with shift-operations by a maximum of 7 bits, so the *cmd* inputs of our *Shifter* components are 3-bit words, as we can see from Figure 3.5. Since we are using the one's complement representation the choice of the value of the most significant bits we have to concatenate is driven by the most significant bit of te input *din*.

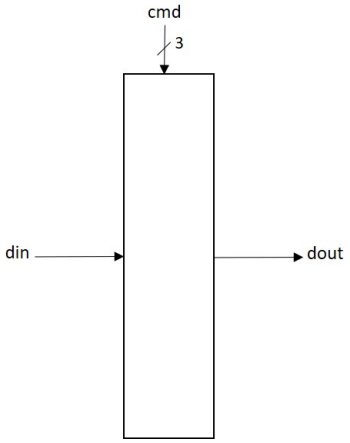


Figure 3.5: *Shifter*: internal design of the component.

3.1.2.4 ROM

The *ROM* component is necessary to maintain and to send to the lower *SSS* component through a *Selector* all the values of the arctangents of the factors $1 + jk_i = 1 + j2^{-i}$. To represents these real numbers, we decided to use the fixed point representation with six bits for both the integer and the fractional parts. Since from specificationa the circuit executes a maximum of 8 iteration, we created a ROM with 8 entries, so we had to deal with addresses composed by natural numbers in the range from 0 to 7 and the corresponding *address* input of our *ROM* component is a 3-bit word.

4. VHDL Code

4.1 Component VHDL description

4.1.1 Selector

```
1  library IEEE;
   use IEEE.std_logic_1164.all;
3
   entity selector is
5    generic (N: integer := 18);

7    port(
       xin: in std_logic_vector (N-1 downto 0);
9      cmd: in std_logic;
       xout: out std_logic_vector(N-1 downto 0)
11   );
   end selector;
13
   architecture bhv of selector is
15   begin
       selector_proc: process(cmd, xin)
17     begin
         if(cmd='1') then
19         xout <= not xin;
         else
21         xout <= xin;
         end if;
23     end process selector_proc;
   end bhv;
```

selector.vhd

4.1.2 SSS component

4.1.2.1 D-positive edge-triggered flip-flop (*dff*)

```
library ieee;
use ieee.std_logic_1164.all;


entity dff is
  generic( N : integer := 18);

  port(
    d : in std_logic_vector(N - 1 downto 0); -- Input of the register
    q : out std_logic_vector(N - 1 downto 0); -- Output of the register
          clk : in std_logic;
          rst : in std_logic
  );
end dff;


architecture bhv of dff is

  begin
    dff_proc : process(clk, rst)
      begin
        if(rst = '0') then
          q <= (N - 1 downto 0 => '0');
        elsif(clk = '1' and clk'event) then
          q <= d;
        end if;
    end process;
end bhv;
```

dff.vhd

4.1.2.2 Adder (*onecompl_adder*)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity onecompl_adder is
  generic(N: integer := 18);

  port(
    a : in std_logic_vector(N-1 downto 0);
    b : in std_logic_vector(N-1 downto 0);
    c_in : in std_logic;
    res : out std_logic_vector(N-1 downto 0);
    c_out : out std_logic
  );

end onecompl_adder;

architecture bhv of onecompl_adder is
  signal internal_c: std_logic;
  begin
    sum: process(a, b, c_in)

      begin
        internal_c<= c_in; -- first carry

        for i in 0 to N-1 loop
          res(i)<= a(i) xor b(i) xor internal_c;
          internal_c <= (a(i) and b(i)) or (a(i) and internal_c) or (b(i)
        and internal_c);
        end loop;

        c_out <= internal_c;

    end process sum;
end bhv;
```

onecompl_adder.vhd

4.1.2.3 SSS (Select-Sum-Save)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity sss is
  generic(N : integer := 18);
  port(
    in1: in std_logic_vector(N-1 downto 0);
    in2: in std_logic_vector(N-1 downto 0);
    out1: out std_logic_vector(N-1 downto 0);
    cmd: in std_logic;
    rst: in std_logic;
    clk: in std_logic
  );
end sss;

architecture structural of sss is
-- ******************************************************************
-- COMPONENTS
-- ******************************************************************
  component dff is
    generic( N : integer := 18);
    port(
      d : in std_logic_vector(N - 1 downto 0); -- Input of the register
      q : out std_logic_vector(N - 1 downto 0); -- Output of the register
            clk : in std_logic;
            rst : in std_logic
    );
  end component dff;

  component onecompl_adder is
    generic ( N : integer := 18);
    port (
      a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      c_in : in std_logic;
      res : out std_logic_vector(N-1 downto 0);
      c_out : out std_logic
    );
  end component onecompl_adder;

-- ******************************************************************
-- Internal signals
-- ******************************************************************
  signal din_selected : std_logic_vector(N-1 downto 0);
  signal dff_out : std_logic_vector(N-1 downto 0);
```

```
  signal add_result : std_logic_vector(N-1 downto 0);
49  signal previous_carry : std_logic;

51  begin
    DFF1 : dff
53    port map(add_result, dff_out, clk, rst);


55
    ONECOMPLEADDER : onecompl_adder
57    port map(din_selected, in2, previous_carry, add_result, previous_carry);


59
    MULTIPLEXER : process(in1, cmd, dff_out)
61    begin
        if(cmd ='1') then
63        din_selected <= in1;
        else
65        din_selected <= dff_out;
        end if;
67    end process MULTIPLEXER;

69  out1 <= dff_out;

71 end structural;
```

sss.vhd

4.1.3 Shifter

```
1  library IEEE;
   use IEEE.std_logic_1164.all;
3
   entity shifter is
5    generic(N : integer := 18);
     port(
7      din: in std_logic_vector(N-1 downto 0);
       dout: out std_logic_vector(N-1 downto 0);
9      cmd: in std_logic_vector (2 downto 0)
     );
11 end shifter;

13 architecture bhv of shifter is

15   signal worker1, worker2:  std_logic_vector(N-1 downto 0); -- internal signals,
       feedforward connection of the 3 multiplexers

17 -- ********************************************************************
   -- According to the value of the 3 bits of the cmd word we right shift a different number
        of locations
19 --  -> cmd(0) = '1' => right-shift of 1 location
   --  -> cmd(1) = '1' => right-shift of 2 locations
21 --  -> cmd(2) = '1' => right-shift of 4 locations
   -- ********************************************************************
23   begin

25     shift_by_one:process(din, cmd(0))
       begin
27       if(cmd(0)='1') then
           worker1(N-1) <= din(N-1);
29         worker1(N-2 downto 0) <= din(N-1 downto 1);
         else
31         worker1(N-1 downto 0) <= din(N-1 downto 0);
         end if;
33     end process;

35     shift_by_two:process(worker1, cmd(1))
       begin
37       if(cmd(1)='1') then
           worker2(N-1) <= worker1(N-1);
39         worker2(N-2) <= worker1(N-1);
           worker2(N-3 downto 0) <= worker1(N-1 downto 2);
41       else
           worker2(N-1 downto 0) <= worker1(N-1 downto 0);
43       end if;
       end process;
45
```

```
    shift_by_four:process(worker2, cmd(2))
    begin
      if(cmd(2)='1') then
        dout(N-1) <= worker2(N-1);
        dout(N-2) <= worker2(N-1);
        dout(N-3) <= worker2(N-1);
        dout(N-4) <= worker2(N-1);
        dout(N-5 downto 0) <= worker2(N-1 downto 4);
      else
        dout(N-1 downto 0) <= worker2(N-1 downto 0);
      end if;
    end process;
end bhv;
```

shifter.vhd

4.1.3.1 ROM

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rom is
  port(
    addr: in std_logic_vector(2 downto 0); -- rom address, 8 locations = 3 bit
    data: out std_logic_vector(11 downto 0) -- output datum
  );
end rom;

architecture bhv of rom is
  type int_array is array (natural range 0 to 7) of std_logic_vector(11 downto 0);
  constant rom : int_array := (

--    binary value          natural value   fixed point value
    0=>"101101000000",  --  2880            45°
    1=>"011010100100",  --  1700            26,5625°
    2=>"001110000010",  --  898             14,03125°
    3=>"000111001000",  --  456             7,125°
    4=>"000011100101",  --  229             3,578125°
    5=>"000001110011",  --  115             1,796875°
    6=>"000000111001",  --  57              0,890625°
    7=>"000000011101"   --  29              0,453125°
  );
  begin
    data <= rom(conv_integer(addr));

end bhv;
```

rom.vhd

4.2 CORDIC Arctan VHDL description

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity cordic_arctan is
  port(
    den : in std_logic_vector(11 downto 0);
    num : in std_logic_vector(11 downto 0);
    ris : out std_logic_vector(11 downto 0);
    rst: in std_logic;
    clk: in std_logic
  );
end cordic_arctan;
architecture structural of cordic_arctan is
-- ******************************************************************
-- COMPONENTS
-- ******************************************************************
  component selector is
    generic (N: integer := 18);
    port(
      xin: in std_logic_VECTOR (N-1 downto 0);
      cmd: in std_logic;
      xout: out std_logic_VECTOR(N-1 downto 0)
    );
  end component;

  component sss is
    generic(N : integer := 18);
    port(
      in1: in std_logic_vector(N-1 downto 0);
      in2: in std_logic_vector(N-1 downto 0);
      out1: out std_logic_vector(N-1 downto 0);
      cmd: in std_logic;
      rst: in std_logic;
      clk: in std_logic
    );
  end component;

  component shifter is
    generic(N : integer := 18);
    port(
      din: in std_logic_vector(N-1 downto 0);
      dout: out std_logic_vector(N-1 downto 0);
      cmd: in std_logic_vector (2 downto 0)
    );
  end component;
```

```
49    component rom is
        port(
51        addr: in std_logic_vector(2 downto 0);
          data: out std_logic_vector(11 downto 0)
53      );
      end component;
55
  -- ******************************************************************
57  -- Internal Signals
  -- ******************************************************************
59
      -- higher part of circuit "worker"
61    signal ext_num : std_logic_vector(17 downto 0);
      signal sel1_sss1 : std_logic_vector(17 downto 0);
63    signal sss1_shifter1 : std_logic_vector(17 downto 0);
      signal shifter1_sel3: std_logic_vector(17 downto 0);
65    signal sel3_sss2 : std_logic_vector(17 downto 0);

67    -- lower part of circuit "worker"
      signal ext_den : std_logic_vector(17 downto 0);
69    signal sel2_sss2 : std_logic_vector(17 downto 0);
      signal sss2_shifter2 : std_logic_vector(17 downto 0);
71    signal shifter2_sel4: std_logic_vector(17 downto 0);
      signal sel4_sss1 : std_logic_vector(17 downto 0);
73
      -- control signals for selectors
75    signal sign_num : std_logic;
      signal sign_b : std_logic;
77    signal not_sign_b : std_logic;

79    -- signals of circuit which produces result
      signal rom_sel5 : std_logic_vector(17 downto 0);
81    signal sel5_sss3 : std_logic_vector(17 downto 0);
      signal sss3_out : std_logic_vector(17 downto 0); -- ext_ris
83
      -- signal to "clean" the din2 of sss components
85    signal real_sel4_sss1 : std_logic_vector(17 downto 0);
      signal real_sel3_sss2 : std_logic_vector(17 downto 0);
87    signal real_sel5_sss3 : std_logic_vector(17 downto 0);

89    -- internal signals
      signal iteration : std_logic_vector(2 downto 0); -- signal to count iterations
91    signal start_computation : std_logic;
      signal selection : std_logic;
93    signal end_computation : std_logic;


95
      begin
```

```
97  -- ********************************************************************
    -- Mapping of components : 1. SELECTORS
99  -- ********************************************************************
      -- selectors to select the proper input (case a < 0)
101   SEL1 : selector
        port map(ext_num, sign_num, sel1_sss1);
103
      SEL2 : selector
105     port map(ext_den, sign_num, sel2_sss2);

107   -- selectors to select the proper din2 (ak or bk) for sss components of "worker"
      SEL3 : selector
109     port map(shifter1_sel3, not_sign_b, sel3_sss2);

111   SEL4 : selector
        port map(shifter2_sel4, sign_b, sel4_sss1);
113
      -- selector to select the proper angle (positive or negative)
115   SEL5 : selector
        port map(rom_sel5, sign_b, sel5_sss3);
117 -- ********************************************************************
    -- 2. SSS COMPONENTS
119 -- ********************************************************************
      SSS1 : sss
121     port map(sel1_sss1, real_sel4_sss1, sss1_shifter1, selection, rst, clk);

123   SSS2 : sss
        port map(sel2_sss2, real_sel3_sss2, sss2_shifter2, selection, rst, clk);
125
      SSS3 : sss
127     port map(B"0000000000000000", real_sel5_sss3, sss3_out, selection, rst, clk);
    -- ********************************************************************
129 -- 3. SHIFTERS
    -- ********************************************************************
131   SHIFTER1 : shifter
        port map(sss1_shifter1, shifter1_sel3, iteration);
133
      SHIFTER2 : shifter
135     port map(sss2_shifter2, shifter2_sel4, iteration);
    -- ********************************************************************
137 -- 4. ROM
    -- ********************************************************************
139   ROM1 : rom
        port map(iteration, rom_sel5(11 downto 0));
141
    -- ********************************************************************
143 -- MULTIPLEXING PROCESSES: see Subsection 3.1.1.
    -- ********************************************************************
145
```

```
  MULTIPLEXER1 : process(selection, sel4_sss1)
147   begin
      if(selection = '1') then
149     real_sel4_sss1 <= (others => '0');
      else
151     real_sel4_sss1 <= sel4_sss1;
      end if;
153 end process MULTIPLEXER1;

155 MULTIPLEXER2 : process(selection, sel3_sss2)
    begin
157   if(selection = '1') then
        real_sel3_sss2 <= (others => '0');
159   else
        real_sel3_sss2 <= sel3_sss2;
161   end if;
  end process MULTIPLEXER2;
163
  MULTIPLEXER3 : process(selection, sel5_sss3)
165   begin
      if(selection = '1') then
167     real_sel5_sss3 <= (others => '0');
      else
169     real_sel5_sss3 <= sel5_sss3;
      end if;
171 end process MULTIPLEXER3;

173 -- ***********************************************************************
  -- CORDIC ALGORITHM FOR ARCTAN ESTIMATION
175 -- ***********************************************************************
  CORDIC_ARCTAN_PROCESS : process(rst, clk)
177   variable execute : std_logic := '0';
    variable counter : integer range 0 to 15;
179 variable first_step : std_logic;
    begin
181   if(rst ='0') then
        counter := 0;
183     execute := '0';
        first_step := '1';
185     start_computation <= '0';
        selection <= '1';
187   elsif(clk = '1' and clk'event) then
        if(execute = '0' and start_computation <= '0') then
189       --  base cases handling
          if(den = B"000000000000" or den = B"111111111111") then
191         execute := '0';
            ris <= B"000000000000";
193       elsif(num = B"000000000000" or num = B"111111111111") then
            execute := '0';
```

```
195               if(den(11) = '0') then
                    ris <= B"010110100000";
197               else
                    ris <= B"101001011111";
199               end if;
                end else
201           --  not one of the base cases
                execute := '1';
203             selection <= '1';
                counter := 0;
205             start_computation <= '1';
                iteration <= std_logic_vector(to_unsigned(counter,3));
207           end if;

209         else
            --  execution of the algorithm (not base case)
211           selection <= '0'; -- enabling of feedbacks of sss components
              if(counter < 8) then -- execution not ended
213             if(sss2_shifter2 = B"000000000000000000" or sss2_shifter2 =
                  B"111111111111111111") then
                -- algorithm has converged, adjustment of the result
215               if(sss3_out(17)='0') then
                    ris(10 downto 0) <= sss3_out(12 downto 2);
217               else
                    ris(10 downto 0) <= not sss3_out(12 downto 2);
219               end if;
                  ris(11) <= sss3_out(17);
221               execute := '0'; -- start_computation is 1
                else
223             --  algorithm has not converged yet
                  if(first_step = '1') then
225                 first_step := '0';
                  else
227                 counter := counter+1;
                    iteration <=std_logic_vector(to_unsigned(counter,3));
229               end if;
                end if;
231           else -- counter = 8, end of execution
                execute := '0';
233             counter := 0;
                if(den(11) = '0') then
235               ris(10 downto 0) <= sss3_out(12 downto 2);
                else
237               ris(10 downto 0) <= not sss3_out(12 downto 2);
                end if;
239           end if;
            end if;
241       end if;
  end process CORDIC_ARCTAN_PROCESS;
```

```
243 -- ******************************************************************
-- MANAGEMENT OF INTERNAL SIGNALS
245 -- ******************************************************************
  sign_num <= num(11);
247  sign_b <= sss2_shifter2(17);
  not_sign_b <= not sign_b;
249
-- ******************************************************************
251 -- EXTENSIONS OF INPUTS AND ROM DATUM
-- ******************************************************************
253  ext_num(15 downto 4) <= num;
  ext_num(17) <= num(11);
255  ext_num(16) <= num(11);
  ext_num(3) <= num(11);
257  ext_num(2) <= num(11);
  ext_num(1) <= num(11);
259  ext_num(0) <= num(11);

261  ext_den(15 downto 4) <= den;
  ext_den(17) <= den(11);
263  ext_den(16) <= den(11);
  ext_den(3) <= den(11);
265  ext_den(2) <= den(11);
  ext_den(1) <= den(11);
267  ext_den(0) <= den(11);

269  rom_sel5(17 downto 12) <= B"000000";

271 end structural;
```

cordic_arctan.vhd

5. Testplan

In order to see if the combinatorial logic works correctly, we have used a VHDL test-bench file to verify the behavior in all the possible cases like:

- base cases ($num_h = 0$ or $den_h = 0$);
- convergence of the algorithm at different iteration (also in case of $num_h < 0$);
- non-convergence of the algorithm with the result returned after the 8 iterations.

5.1 VHDL TestBench for verification

```
1  library IEEE;
   use IEEE.std_logic_1164.all;
3
   entity cordic_arctan_tb is
5  end entity cordic_arctan_tb;

7  architecture structural of cordic_arctan_tb is -- Testbench architecture declaration

9  -- ******************************************************************
   -- Testbench constants
11 -- ******************************************************************
     constant T_CLK   : time := 10 ns; -- Clock period
13   constant T_RESET : time := 25 ns; -- Period before the reset deassertion

15 -- ******************************************************************
   -- Testbench signals
17 -- ******************************************************************
     signal num_tb : std_logic_vector(11 downto 0);
19   signal den_tb : std_logic_vector(11 downto 0);
     signal ris_tb : std_logic_vector(11 downto 0);
21   signal rst_tb  : std_logic := '0'; -- reset signal
     signal clk_tb : std_logic := '0'; -- clock signal, intialized to '0'
23   signal end_sim : std_logic := '1'; -- signal to use to stop the simulation

25 -- ******************************************************************
   -- Component to test (DUT) declaration
27 -- ******************************************************************
```

```
  component cordic_arctan is
    port(
      num : in std_logic_vector(11 downto 0);
      den : in std_logic_vector(11 downto 0);
      ris : out std_logic_vector(11 downto 0);
      rst : in std_logic;
      clk : in std_logic
    );
  end component;

  begin
    clk_tb <= (not(clk_tb) and end_sim) after T_CLK / 2;
    rst_tb <= '1' after T_RESET;

    test_cordic_arctan: cordic_arctan
      port map(
        num => num_tb,
        den => den_tb,
        ris => ris_tb,
        rst => rst_tb,
        clk => clk_tb
      );

    d_process: process(clk_tb, rst_tb)
      variable t : integer := 0;
    begin
        if(rst_tb = '0') then
          t := 0;
        -- ******************************************************
        -- Here we can put the desired test inputs
          num_tb <= "000000000011";
          den_tb <= "111111111110";
        -- ******************************************************
      elsif(rising_edge(clk_tb)) then
          case(t) is
          when 16 => end_sim <= '0';
                when others => null;
          end case;
          t := t + 1;
      end if;
     end process;
end structural;


```

cordic_arctan_tb.vhd

5.2 Results

5.2.1 Base Cases Test results

In this first test phase, we tested the base cases and we reached the results shown in Table 5.1.

Base Cases	Desired Result	Returned Result	Error
$den_h = 0$ & $num_h = 0$	0	0	0
$den_h = 0$ & $num_h \neq 0$	0	0	0
$den_h > 0$ & $num_h = 0$	90°	90°	0
$den_h < 0$ & $num_h = 0$	-90°	-90°	0

Table 5.1: Base Cases Table results

5.2.2 Convergence Test results

In this second test phase we tested the convergence cases.
The total number of convergence test cases (equal to the number of the possible accumulation values) is

$$\sum_{i=1}^{8} 2^i = 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 = 510$$

To be a more accurate it is slightly smaller beacause there are cases such that the accumulation of the angles are greater than 90° or smaller than $-90°$; these angles are determined by complex numbers with negative real part, as we have seen these numbers are treated in the same way, i.e. applying the simmetry with respect to the origin of the Gauss plane.
Omitting the first and the second iterations in which we could test respectively two and four cases, we decided to test only some convergence test cases until the 5^{th} iteration and we obtained the results shown in Table 5.2. We decided to take the values for those test cases from the first quadrant because as we can see in the table below from the first three iterations, complex numbers with opposite imaginary part simply result in opposites value with the same error in absolute value.

Complex Number	**Target Result [°]**	**Returned Result [°]**	**Error [°]**
Convergence Cases at 1^{st} iteration			
$1 + j$	45	45	0
$1 - j$	-45	-45	0
Convergence Cases at 2^{nd} iteration			
$5 + 15j$	71,56505118	71,5625	0,00255
$5 - 15j$	-71,56505118	-71,5625	0,00255
$15 + 5i$	18,43494882	18,4375	0,00255118
$15 - 5i$	-18,43494882	-18,4375	0,00255118
Convergence Cases at 3^{rd} iteration			
$1 + 13j$	85,60129465	85,5625	0,03879465
$1 - 13j$	-85,60129465	-85,5625	0,03879465
$7 + 11j$	57,52880771	57,5	0,02880771
$7 - 11j$	-57,52880771	-57,5	0,02880771
$11 + 7j$	32,47119229	32,4375	0,03369229
$11 - 7j$	-32,47119229	-32,4375	0,03369229
$13 + j$	4,398705355	4,375	0,023705355
$13 - j$	-4,398705355	-4,375	0,023705355
Convergence Cases at 4^{th} iteration			
$21 + 103j$	78.4762783	78,4375	0,0387783
$45 + 95j$	64,65382406	64,625	0,02882406
$95 + 45j$	25,34617594	25,3125	0,03367594
$105 + 5j$	2,726310994	3,0625	0,336189006
$81 + 67j$	39.59620864	39,5625	0,03370864
$103 + 21j$	11.5237217	11,5	0,0237217
Convergence Cases at 5^{th} iteration			
$25 + 1685j$	89,14997662	89,125	0,2497662
$233 + 1669j$	82,05261267	82	0,05261267

Table 5.2: Convergence Cases Table results

We have stopped to this base cases because starting from the 4^{th} iteration some theoretical base cases started to not converge because of the errors due to the truncations of the shifted value and it was not easy to find complex numbers that would converge in the last iterations.

5.2.3 Non-Convergence Test results

At this point we can see some cases in which we are sure the algorithm do not converge, so it will execute all the 8 iterations provided by the design.

Complex Number	**Target Result** [°]	**Returned Result** [°]	**Error** [°]
$2047 + j$	0,027990119	0,375	0,347009881
$2 + j$	26,56505118	25,75	0,81505118
$1 + 2j$	63,43494882	64,1875	0,75255118
$5 + 16j$	72,64597536	72,625	0,02097536
$1 + 2047j$	89,97200988	89,5625	0,40950988
$1 - 2047j$	-89,97200988	-89,5625	0,40950988
$5 - 16j$	-72,64597536	-72,625	0,02097536
$1 - 2j$	-63,43494882	-64,1875	0,75255118
$2 - j$	-26,56505118	-25,75	0,81505118
$2047 - j$	-0,027990119	-0,375	0,347009881

Table 5.3: Non-Convergence Cases Table results

As we can see from the results reported in Table 5.3, in this case the error committed by the algorithm is slightly greater with respect to the the converge cases reported in Table 5.2, but we could expect it at the beginning of our test phase.
We can notice by all the collected results that our algorithm cannot commit results with an error greater than the summation of the values of the bit of the fractional part of the result, i.e.
$0,5 + 0,25 + 0,125 + 0,0625 = 0,9375$.

6. Synthesis

After the development of the architecture we synthesized it on the Zybo (ZYnq BOard)[1] using Xilinx VIVADO Design Suite.
For the synthesis, we have chosen a clock speed constraint of 20ns, thus a frequency of 50MHz.
We then run it and it successfully completed with some warnings:

- **[Constraints 18-5210]No constraint will be written out.**
  According to the Xilinx Forum[2] this warning can be safely ignored.
- **ZPS7 The PS7 cell must be used in this Zynq design in order to enable correct default configuration.**
  Even in this case, from the Xilinx Forum[3], we can ignore it since it refers to the ARM Processor environment which is out of our scopes.
- **35 out of 35 logical ports use I/O standard (IOSTANDARD) value 'DEFAULT', instead of a user assigned specific value [. . .].**
  This is due to we did not configure I/O pins of the boards and since we will not test the mini router with the board, also this warning can be safely ignored.
- **[Synth 8-295] found timing loop.[cordic_arctan.vhd:8](2 more like this).**
  These warnings say that the tool has found a bunch of combinatorial functions that form a loop (i.e. the output of some gate feeds back to the input cone of that gate). After some attempts we have ascertained that these warnings are caused by the feedback of the *c_out* output in the *c_in* input of the *onecompl_adder* inside the *SSS* components, so these loops are necessary for the correct production of the one's complement addition and moreover does not affect the final result of the computation.

[1] XC7Z020-Data-Sheet.pdf
[2] https://forums.xilinx.com/t5/Synthesis/Meaning-of-synthesis-warning-Constraints-18-5210-No-constraint/m-p/881007
[3] https://forums.xilinx.com/t5/Welcome-Join/How-to-instantiate-the-PS7-block/m-p/333953

6.0.1 Timing Report

The timing report of the synthesis is shown in Figure 6.1 and it shows that the clock time constraint previously chosen is met.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,905 ns	Worst Hold Slack (WHS):	0,150 ns	Worst Pulse Width Slack (WPWS):	9,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	97	Total Number of Endpoints:	97	Total Number of Endpoints:	78

All user specified timing constraints are met.

Figure 6.1: Timing Report

As we could expect, the worst critical path is one of the two paths that starts from the output of the *dff register* of a *SSS* component and ends at the input of the *dff register* of the other *SSS* component as we can see in Figure 6.2

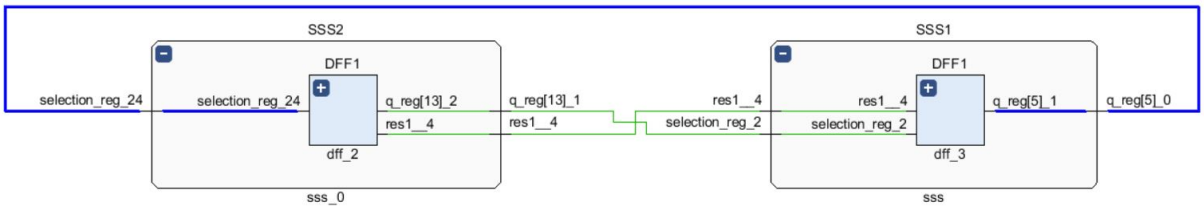


Figure 6.2: Critical path

6.0.2 Max Operating Frequency

We know that the clock speed must be

$$t_{clk} \geq t_{su} + t_{p-logic} + t_{c-q}$$

where:

- t_{su} is the time during which the input shall be stable before the clock edge;
- t_{c-q} is the time after the clock edge necessary for the stabilization on the output;
- $t_{p-logic}$ is the time necessary for the combinatorial logic to produce the output given the input

All this three parameters are fixed.
In our case, we also have the slack time t_{slack} that represent the amount of time that the data is stable, before the clock edge, in addition to the setup time t_{su}:

$$t_{clk} = t_{su} + t_{p-logic} + t_{c-q} + t_{slack}$$

To minimize the clock period we have to be in the limit case where the t_{slack} is equals to 0:

$$t_{clk_{min}} = t_{su} + t_{p-logic} + t_{c-q} + 0$$

.

The last equation is inserted in the t_{clock} equation :

$$t_{clk} = t_{clk_{min}} + t_{slack}$$

Since we know both $t_{clk} = 20\text{ns}$ and $t_{slack} = 7,905\text{ns}$ we can easily compute

$$t_{clk_{min}} = t_{clock} - t_{slack} = 20 - 7,905 = 12,095\text{ns}$$

And finally

$$f_{max} = \frac{1}{t_{clk_{min}}} = 82.678792889624 \text{ MHz}$$

Note that the values reported after the synthesis are *estimated* values and they do not take into account placement or routing information. More accurate values will be reported after the implementation phase.

7. Zybo Implementation

We have also run the implementation in order to see where and which components of the board would be used if we had loaded the `cordic_arctan` circuit on the board.
As we previously said, the timing report during the synthesis is an estimation, in fact, after the implementation, as we can see from Figure 7.1, the slack it has been increased to $7,865$ ns.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,865 ns	Worst Hold Slack (WHS):	0,200 ns	Worst Pulse Width Slack (WPWS):	9,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	97	Total Number of Endpoints:	97	Total Number of Endpoints:	78

All user specified timing constraints are met.

Figure 7.1: Timing Report

After the implementation the max operating frequency is:

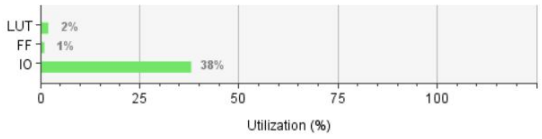
$$f_{max} = 82.406262875979 MHz \text{MHz}$$

7.1 Resources Utilization

Looking at the resources utilization report we can see the results in the following Figure.

Resource	Utilization	Available	Utilization %
LUT	268	17600	1.52
FF	77	35200	0.22
IO	38	100	38.00

(a) Used resources



(b) Used resources of the total available on the board

Figure 7.2: Resources Utilization

Figure 7.2a shows the amount of resources used on the board to implement the system, Figure 7.2b represents the percentage of the used resources with respect to the total available on the Zybo: as we can see, except I/O pins, the values are low. This is legitimate, since this project is very small with respect to the complexity that the board is able to perform, the LUTs and the Flip Flops required are a small percentage of the total.
The story is slightly different with respect to the I/O pins, this is due to the number of signals involved in the `cordic_arctan` circuit, i.e. the two inputs num_h and den_h and the output res_k, that all require

12 bits.

The real implementation of the system on the board is shown in Figure 7.3

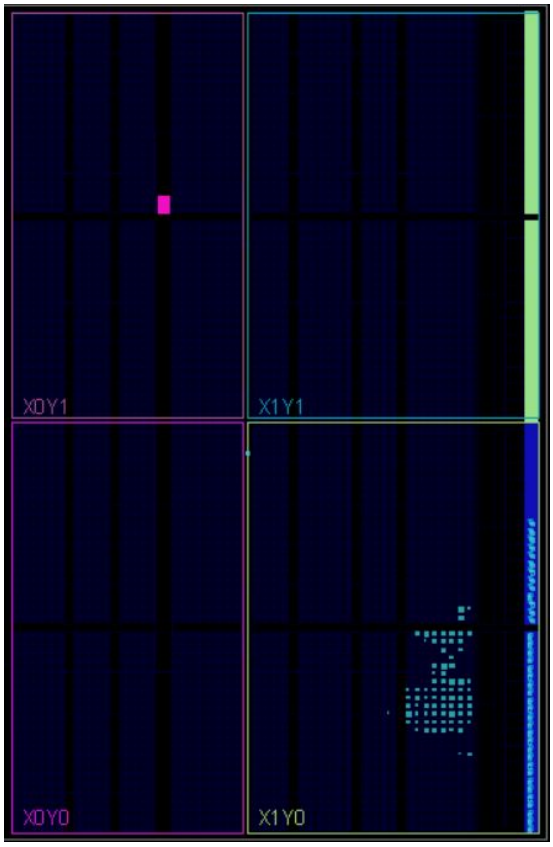


Figure 7.3: *cordic_arctan* device implementation

7.2 Power Utilization

As we can see in Figure 7.4, nearly the total of the dissipated power is static (95%). The remaining 5% is attributable to the clock and as we said for the resources utilization, the I/O.

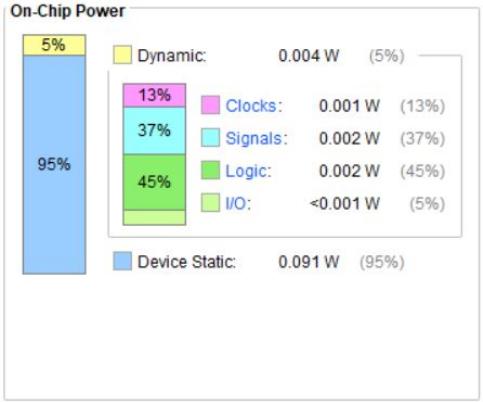


Figure 7.4: Power Utilization

8. Conclusion

CORDIC uses simple shift-add operations for several computing tasks such as the calculation of trigonometric, hyperbolic and logarithmic functions, real and complex multiplications, division, square-root calculation, solution of linear systems, eigenvalue estimation, singular value decomposition, QR factorization and many others. As a consequence, CORDIC has been used for applications in diverse areas such as signal and image processing, communication systems, robotics and 3D graphics apart from general scientific and technical computation[1].
Moreover methods such as power series or table lookups usually need multiplications to be performed. If a hardware multiplier is not available, a CORDIC is generally faster, but if a multiplier can be used, other methods may be faster.
CORDICs can also be implemented in many ways, including a single-stage iterative method, which requires very few gates when compared to multiplier circuits. Also, CORDICs can compute many functions with precisely the same hardware, so they are ideal for applications with an emphasis on reduction of cost (e.g. by reducing gate counts in FPGAs) over speed. An example of this priority is in pocket calculators, where CORDICs are very frequently used[2].
In our case we must underline that we could have 3 types of error that lead to unaccurate results:

1. errors caused by the finite number of iterations implemented;
2. errors caused by truncation of the shifted numbers;
3. errors caused by the truncation of the values stored in the ROM.

[1] Source: Wikipedia
[2] Source: Wikibooks