University of Pisa
MSc in Computer Engineering
Electronics Systems

VHDL project: Approximation of complex number modulus

Sara Lotano
Matricola 517978

Academic Year 2020/2021

Contents

1 Introduction

The aim of this project is to compute the approximation of the modulus of a complex number and the corresponding Mean Square Error (MSE).
Given a complex number Z = P + iQ the modulus is denoted by $|Z|$ and is defined as $|Z| = \sqrt{P^2 + Q^2}$.

1.1 Possible algorithms

There are several algorithms that can be used to compute the modulus of a complex number.

1.1.1 Mathematical approach

Given a complex number z = x + iy, the algorithm is the following:

1. Find the real and imaginary parts:
   - Real part = x
   - Imaginary part = y
2. Find the square of x and y separately:
   - Square of Real part = x^2
   - Square of Imaginary part = y^2
3. Find the sum of the computed squares:
   - Sum = $x^2 + y^2$
4. Find the square root of the computed sum:
   - $|z| = \sqrt{x^2 + y^2}$

The result represents the modulus of the given complex number.
This approach cannot be implemented within digital circuits due to the use of square and square-root operations.

1.1.2 Alpha max plus beta min algorithm

The alpha max plus beta min algorithm is an approximation of the square root of the sum of two squares, for this reason it can be used to compute the modulus of a complex number.
The goal is to find the hypotenuse of a right triangle given the two side lengths, without performing the square and square-root operations.
Given a complex number z = x + iy, the approximation is expressed as:

$$|z| = \alpha Max + \beta Min$$

where Max is the maximum absolute value of x and y, and Min is the minimum absolute value of x and y.
Some values of parameters α and β allow the implementation of the multiplication operation into a simple shift of binary digits that is particularly well suited to be used in high-speed digital circuitry.
For the closest approximation, the optimum values are $\alpha \simeq 0.960$ and $\beta \simeq 0.398$.

1.2 Possible applications

Complex numbers have many applications in a variety of sciences and related areas such as signal processing, control theory, electronics, quantum mechanics and many others.

- In control theory, systems are often transformed from the time domain to the frequency domain using the Laplace transform. The system's poles and zeros are then analyzed in the complex plane.
- In signal analysis and other fields for the description of periodically varying signals the real functions, representing actual physical quantities, are often expressed in terms of sines and cosines. For a sine wave of a given frequency, the absolute value $|z|$ of the corresponding z is the amplitude and the argument arg(z) the phase.
- In electrical engineer, complex numbers are used to analyze quantities as voltage, current and resistance because they are able to express the two dimensions of frequency and phase shift at one time. For example, the Fourier transform is used to analyze varying voltages and currents. The treatment of resistors, capacitors, and inductors can then be unified by introducing imaginary, frequency-dependent resistances for the latter two and combining all three in a single complex number called the impedance. This approach is called phasor calculus.
- In digital signal and image processing, they can be used in the digital versions of Fourier analysis (and wavelet analysis) to transmit, compress, restore, and otherwise process digital audio signals, still images, and video signals.

2 Architecture

The signals P and Q, representing respectively the real and imaginary parts of the complex number, are represented in 2's complement on N bits.
The representation interval is symmetric, for this reason the values are included within the range $[-(2^{N-1}-1), 2^{N-1}-1]$.
Thereby, the signals $|P|$ and $|Q|$ are represented in the interval $\left[0, 2^{N-1}-1\right]$.

2.0.1 Implemented algorithm

The approximation algorithm used in this project is the following:

$$\sqrt{P^2+Q^2} = max\left\{|P|,|Q|\right\} + \frac{1}{2} min\left\{|P|,|Q|\right\} - \frac{1}{16}\left[max\left\{|P|,|Q|\right\} + min\left\{|P|,|Q|\right\}\right]$$

The algorithm avoids performing the square and square-root operations. It uses simple operations such as comparison, multiplication and addition.

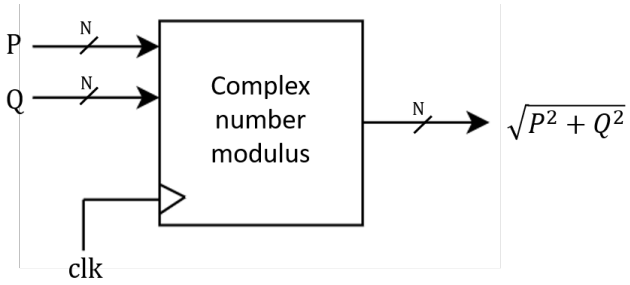

Figure 1: Inputs and output of the implemented block

Circuit characteristics:

- The **inputs** of the circuit are the real part P and the imaginary part Q of a complex number.
- The **output** represents the modulus of the complex number given as input.
- The circuit simulation is performed for $\mathbf{N = 10}$. The range of values is $[-511, 511]$.

Even if the module is a pure combinatorial circuit, it is possible to use a clock signal within the testbenches in order to keep trace of the execution time of the simulation and to synchronize the generation of inputs.

3 VHDL Description

The final module is composed by different submodules each of which has a specific functionality. The submodules are: absolute value, max and min, division by 2, division by 16, adder and subtractor.
In order to guarantee the correct implementation, each module has its own testbench. A complete testbench should test all values within the above specified range, but only limit or interesting cases will be considered.

3.1 Abs

Since the input values can be either positive or negative, this module checks the value of the most significant bit: if it is equal to 0, the number is positive, if it is equal to 1, the number is negative.
To compute the absolute value of a negative number, the input value is complemented and then is incremented by 1.

```
entity Abs_module is
    generic(Nbit: integer);
    port (
        input : in std_logic_vector (Nbit - 1 downto 0);
        output : out std_logic_vector (Nbit - 1 downto 0)
    );
end Abs_module;

architecture behavioral of Abs_module is
begin
    process(input)
        begin
            if (input(Nbit - 1) = '0') then
                output <= input;
            else
                output <= std_logic_vector(unsigned(not(input)) + 1);
            end if;
    end process;
end behavioral;
```

3.1.1 Abs Testbench

Clock cycle	Input value		Output value	
	Decimal	**Binary**	**Decimal**	**Binary**
1	0	0000000000	0	0000000000
2	+1	0000000001	+1	0000000001
3	-1	1111111111	+1	0000000001
4	+2	0000000010	+2	0000000010
5	-2	1111111110	+2	0000000010
6	+510	0111111110	+510	0111111110
7	-510	1000000010	+510	0111111110
8	+511	0111111111	+511	0111111111
9	-511	1000000001	+511	0111111111

Table 1: Tested values for Abs module

3.2 Max-Min

This module computes the maximum and minimum value of the two input values.

```
entity MaxMin is
    generic(Nbit: positive);
    port (
        input1 : in std_logic_vector (Nbit - 1 downto 0);
        input2 : in std_logic_vector (Nbit - 1 downto 0);
        max : out std_logic_vector (Nbit - 1 downto 0);
        min : out std_logic_vector (Nbit - 1 downto 0)
    );
end MaxMin;

architecture behavioral of MaxMin is
begin
    max<=input1 when(input1>input2) else input2;
    min<=input1 when(input1<=input2) else input2;
end behavioral;
```

3.2.1 Max-Min Testbench

Clock cycle	Input1 value		Input2 value		Max value	Min value
	Decimal	Binary	Decimal	Binary	Decimal	Decimal
1	0	000000000	1	000000001	1	0
2	511	111111111	2	000000010	2	1
3	511	111111111	510	111111110	511	510
4	0	000000000	511	111111111	511	0
5	511	111111111	511	111111111	511	511

Table 2: Tested values for Max-Min module

3.3 Division by 2

The division by 2 can be implemented as a shift to the right of one position, the most significant bit is filled with a zero.
In this case x is an unsigned number and the operation corresponds to $\lfloor \frac{x}{2} \rfloor$.

```
entity DivBy2 is
    generic (Nbit : positive);
    port (
        input : in std_logic_vector (Nbit - 1 downto 0);
        output : out std_logic_vector (Nbit - 1 downto 0)
    );
end DivBy2;

architecture behavioral of DivBy2 is
begin
    output <= std_logic_vector(shift_right(unsigned(input) , 1));
end behavioral;
```

3.3.1 Division by 2 Testbench

Clock cycle	Input value		Output value	
	Decimal	Binary	Decimal	Binary
1	1	000000001	0	000000000
2	2	000000010	1	0000000001
3	3	000000011	1	0000000001
4	4	000000100	2	0000000010
5	5	000000101	2	0000000010
6	511	111111111	255	011111111
7	510	111111110	255	011111111
8	509	111111101	254	011111110
9	508	111111100	254	011111110

Table 3: Tested values for Division by 2 module

3.4 Division by 16

The division can be implemented as a shift to the right of four position. The four most significant bits are filled with zeros.
In this case x is an unsigned number and the operation corresponds to $\left\lfloor \frac{x}{2^4} \right\rfloor$.

```
entity DivBy16 is
    generic (Nbit : positive);
    port (
        input : in std_logic_vector (Nbit - 1 downto 0);
        output : out std_logic_vector (Nbit - 1 downto 0)
    );
end DivBy16;

architecture behavioral of DivBy16 is
begin
    output <= std_logic_vector(shift_right(unsigned(input) , 4));
end behavioral;
```

3.4.1 Division by 16 Testbench

Clock cycle	Input value		Output value	
	Decimal	Binary	Decimal	Binary
1	1	0000000001	0	0000000000
2	16	0000010000	1	0000000001
3	17	0000010001	1	0000000001
4	32	0000100000	2	0000000010
5	48	0000110000	3	0000000011
6	64	0001000000	4	0000000100
7	80	0001010000	5	0000000101
8	510	0111111110	31	0000011111
9	511	0111111111	31	0000011111

Table 4: Tested values for Division by 16 module

3.5 Adder

The circuit that corresponds to an Adder has been implemented starting from a 1-bit Full Adder. The latter sums the bits a and b of the addends and the carry in c_i, the output is composed by the bit o which represents the sum and the possible carry c_o.
The carry out bit c_o should be interpreted in the following way:

- $c_o = 0$ means that the result of the sum can be represented on the same number of bits on which the operands are represented.
- $c_o = 1$ means that the result of the sum cannot be represented on the same number N of bits on which the operands are represented, but on N+1 bits. In this case the most significant bit will be 1.

In order to sum addends that are represented on many bits, the Full Adder N bits has been implemented as a cascade of several 1-bit Full Adder connected by the internal signal c_{int} that connects the carry out of a previous adder with the carry in of the next adder.

3.5.1 Full Adder 1 bit

```
1  entity FullAdder_1bit is
2      port(
3          a   : in std_logic;
4          b   : in std_logic;
5          c_i : in std_logic;
6          o   : out std_logic;
7          c_o : out std_logic
8      );
9  end FullAdder_1bit;
10
11 architecture data_flow of FullAdder_1bit is
12 begin
13   o   <= a xor b xor c_i;
14   c_o <= (a and b) or (b and c_i) or (c_i and a);
15 end data_flow;
```

3.5.2 Full Adder N bits

```
1  entity FullAdder_Nbit is
2      generic(Nbit : positive);
3      port(
4          a   : in std_logic_vector(Nbit - 1 downto 0);
5          b   : in  std_logic_vector(Nbit - 1 downto 0);
6          c_i : in std_logic;
```

```
7           o   : out  std_logic_vector(Nbit - 1 downto 0);
8           c_o : out std_logic
9           );
10  end FullAdder_Nbit;
11
12  architecture struct of FullAdder_Nbit is
13      component FullAdder_1bit is
14          port(
15               a   : in std_logic;
16               b   : in std_logic;
17               c_i : in std_logic;
18               o   : out std_logic;
19               c_o : out std_logic
20          );
21      end component;
22
23      signal c_int : std_logic_vector(Nbit downto 0);
24
25      begin
26      n_full_adder_gen : for i in 0 to Nbit - 1  generate
27      i_full_adder : FullAdder_1bit
28          port map(
29              a    =>  a(i),
30              b    =>  b(i),
31              c_i  =>  c_int(i),
32              o    =>  o(i),
33              c_o  =>  c_int(i + 1)
34          );
35      end generate;
36
37          c_int(0) <= c_i;          -- Input carry mapping
38          c_o <= c_int(Nbit);       -- Output carry mapping
39
40  end struct;
```

3.5.3 Adder Testbench

Clock cycle	Input1 value		Input2 value		Carry out	Sum value
	Decimal	Binary	Decimal	Binary	Binary	Binary
1	0	000000000	1	000000001	0	000000001
2	0	000000000	511	111111111	0	111111111
3	1	000000001	511	111111111	1	000000000
4	510	111111110	511	111111111	1	111111101
5	511	111111111	511	111111111	1	111111110

Table 5: Tested values for Adder module

3.6 Subtractor

The implementation of the Subtractor circuit follows the same approach of the Adder. It has been implemented starting from a 1-bit Full Subtractor which subtracts the subtrahend b and the borrow in b_{in} to the minuend a and outputs the difference o and the borrow out b_{out}.
The borrow out bit should be interpreted in the following way:

- $b_{out} = 0$ means that the minuend is greater than the subtrahend and therefore the result is a positive number.
- $b_{out} = 1$ means that the minuend is less than the subtrahend and therefore the result is a negative number.

The N-bit Full Subtractor circuit considers multiple 1-bit Full Subtractor in cascade to allow the subtraction operation between operands that are represented on more than one bit.

3.6.1 Full Subtractor 1 bit

```
entity FullSubtractor_1bit is
    port(
        a   : in std_logic;
        b   : in std_logic;
        b_i : in std_logic;
        o   : out std_logic;
        b_o : out std_logic
    );
end FullSubtractor_1bit;

architecture data_flow of FullSubtractor_1bit is
begin
  o   <= a xor b xor b_i;
  b_o <= ((not a and b) or (b and b_i) or (not a and b_i));
end data_flow;
```

3.6.2 Full Subtractor N bits

```
entity FullSubtractor_Nbit is
    generic(Nbit : positive);
    port(
        a   : in std_logic_vector(Nbit - 1 downto 0);
        b   : in  std_logic_vector(Nbit - 1 downto 0);
        b_i : in std_logic;
        o   : out  std_logic_vector(Nbit - 1 downto 0);
        b_o : out std_logic
```

```
        );
end FullSubtractor_Nbit;

architecture struct of FullSubtractor_Nbit is
    component FullSubtractor_1bit is
        port(
             a   : in std_logic;
             b   : in std_logic;
             b_i : in std_logic;
             o   : out std_logic;
             b_o : out std_logic
        );
        end component;

        signal b_int : std_logic_vector(Nbit downto 0);

    begin
        n_full_adder_gen : for i in 0 to Nbit - 1  generate
        i_full_adder : FullSubtractor_1bit
            port map(
                a    =>  a(i),
                b    =>  b(i),
                b_i  =>  b_int(i),
                o    =>  o(i),
                b_o  =>  b_int(i + 1)
            );
        end generate;

    b_int(0) <= b_i;     -- Input borrow mapping
    b_o <= b_int(Nbit); -- Output borrow mapping
end struct;
```

3.6.3 Subtractor Testbench

Clock cycle	Input1 value		Input2 value		Borrow	Difference
	Decimal	Binary	Decimal	Binary	Binary	Binary
1	0	0000000000	1	0000000001	1	1111111111
2	511	0111111111	1	0000000001	0	0111111110
3	1	0000000001	511	0111111111	1	1000000010
4	510	0111111110	511	0111111111	1	1111111111
5	511	0111111111	510	0111111110	0	0000000001
6	511	0111111111	511	0111111111	0	0000000000

Table 6: Tested values for Subtractor module

3.7 Complex number module

The circuit that implements the approximation of the modulus of a complex number, consists of the interconnection of all the previous submodules.
The submodules are connected in the following way:

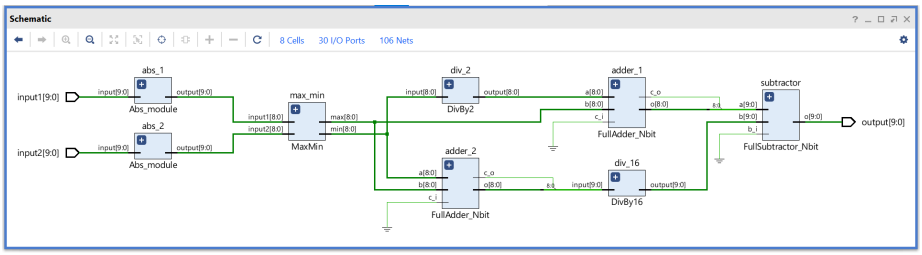

Figure 2: RTL design provided by Vivado

The corresponding VHDL code is:

```vhdl
entity ComplexModulus is
        generic(Nbit: positive);
        port (
        input1 : in std_logic_vector (Nbit - 1 downto 0);
        input2 : in std_logic_vector (Nbit - 1 downto 0);
        output : out std_logic_vector (Nbit - 1 downto 0)
    );
end ComplexModulus;

architecture behavioral of ComplexModulus is
        signal abs1 : std_logic_vector (9 downto 0);
        signal abs2 : std_logic_vector (9 downto 0);
        signal max : std_logic_vector (8 downto 0);
        signal min : std_logic_vector (8 downto 0);
        signal div2 : std_logic_vector (8 downto 0);
        signal sum1 : std_logic_vector (8 downto 0);
        signal c1: std_logic;
        signal sum2 : std_logic_vector (8 downto 0);
        signal c2: std_logic;
        signal div16 : std_logic_vector (9 downto 0);

        -- components declaration (omitted)

begin
    abs_1 : Abs_module
        generic map(Nbit => 10)
        port map(
```

```
28              input => input1,
29              output => abs1
30          );
31
32      abs_2 : Abs_module
33          generic map(Nbit => 10)
34          port map(
35              input => input2,
36              output => abs2
37          );
38
39      max_min: MaxMin
40          generic map(Nbit => 9)
41          port map(
42              input1 => abs1(8 downto 0),
43              input2 => abs2(8 downto 0),
44              max => max,
45              min => min
46          );
47
48      div_2: DivBy2
49          generic map(Nbit => 9)
50          port map(
51              input => min,
52              output => div2
53          );
54
55      adder_1 : FullAdder_Nbit
56          generic map(Nbit => 9)
57          port map(
58              a => div2,
59              b => max,
60              c_i => '0',
61              o => sum1,
62              c_o => c1
63          );
64
65      adder_2 : FullAdder_Nbit
66          generic map(Nbit => 9)
67          port map(
68              a => min,
69              b => max,
70              c_i => '0',
71              o => sum2,
72              c_o => c2
73          );
74
75      div_16: DivBy16
```

```
76          generic map(Nbit => 10)
77          port map(
78              input(9) => c2,
79              input(8 downto 0) => sum2,
80              output => div16
81          );
82
83      subtractor: FullSubtractor_Nbit
84          generic map(Nbit => 10)
85          port map(
86              a(9) => c1,
87              a(8 downto 0) => sum1,
88              b => div16,
89              b_i => '0',
90              o => output,
91              b_o => open
92          );
93
94  end behavioral;
```

In the FullSubtractor_Nbit component, the borrow out bit b_o is left *open* because the subtrahend is always smaller than the minuend (due to the division by 16) and therefore the result of the subtraction will never be a negative number.

4 Test plan

A complete testbench would involve testing all possible values within the range $[-511, 511]$, but consider and evaluate all these cases becomes impossible. For this reason, only the most interesting cases have been considered.
The VHDL testbench code for the Complex Number module is the following:

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

entity ComplexModulus_tb is
end ComplexModulus_tb;

architecture behavioral of ComplexModulus_tb is

    ------------------------------------------------------------------------------------
    -- Testbench constants
    ------------------------------------------------------------------------------------
    -- Clock period definitions
        constant T_CLK : time := 10 ns;
        constant T_RESET : time := 5 ns;
        constant N : integer := 10;

    ------------------------------------------------------------------------------------
    -- Testbench signals
    ------------------------------------------------------------------------------------
    --Inputs
        signal clk : std_logic := '0';
        signal rst : std_logic := '0';
        signal end_sim : std_logic := '1';
        signal p : std_logic_vector(N - 1 downto 0);
        signal q : std_logic_vector(N - 1 downto 0);

        --Output
        signal output : std_logic_vector(N - 1 downto 0);

    ------------------------------------------------------------------------------------
    -- Component to test (DUT) declaration
    ------------------------------------------------------------------------------------
    component ComplexModulus is
        generic (Nbit : positive);
        port(
            input1 : in std_logic_vector (Nbit - 1 downto 0);
            input2 : in std_logic_vector (Nbit - 1 downto 0);
            output : out std_logic_vector (Nbit - 1 downto 0)
        );
```

```
41      end component;
42
43      begin
44          -- Clock process definitions
45          clk <= (not(clk) and end_sim) after T_CLK / 2;
46          rst <= '1' after T_RESET;
47
48          -- Instantiate the Device Under Test (DUT)
49          dut: ComplexModulus
50              generic map(Nbit => N)
51              port map (
52                  input1 => p,
53                  input2 => q,
54                  output => output
55                  );
56
57          d_process: process(clk, rst)
58              variable t : integer := 0;
59              begin
60                  if(rst = '0') then
61                      p <= "0000000000";
62                      q <= "0000000000";
63                      t := 0;
64                  elsif(rising_edge(clk)) then
65                      case(t) is
66                          when  1  => p <= "0000000000"; q <= "0000000000"; -- p= 0   , q= 0
67                          when  2  => p <= "0000000000"; q <= "1111111111"; -- p= 0   , q=-1
68                          when  3  => p <= "1111111111"; q <= "0000000000"; -- p=-1   , q= 0
69                          when  4  => p <= "0000000000"; q <= "1000000001"; -- p= 0   , q=-511
70                          when  5  => p <= "0111111111"; q <= "0111111111"; -- p= 511 , q= 511
71                          when  6  => p <= "0111111111"; q <= "1000000001"; -- p= 511 , q=-511
72                          when 10 => end_sim <= '0';
73                          when others => null;
74                      end case;
75                  t := t + 1;
76                  end if;
77              end process;
78  end behavioral;
```

Clock cycle	Input P	Input Q	Output value	Expected value
1	0	0	0	0
2	0	-1	1	1
3	-1	0	1	1
4	0	-511	480	511
5	511	511	703	722.67
6	511	-511	703	722.67

Table 7: Tested values for Complex Number module

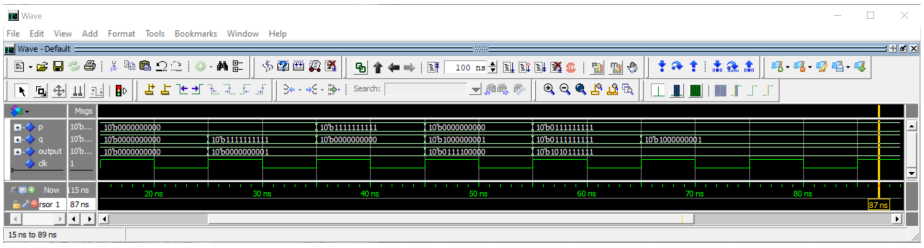

Figure 3: Modelsim execution results

4.1 Mean Squared Error

The mean squared error (MSE) represents the average of the squares of the errors, i.e. the average squared difference between the actual value Y_i and the estimated value $\hat{Y}_i$.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

The estimated values are the ones computed with the approximation and the actual values are those computed using square and square-root operations.
In order to compute the MSE as precise as possible, all the possible values were tested with the following **Matlab** script:

```
Nbit = 10;
max_value = 2^(Nbit-1)-1;

%The matrix exact_values contains the modulus of the complex number
%computed by using square and square-root operations.
exact_values = zeros(max_value);

%The matrix approximated_values contains the modulus of the complex number
%computed by using the approximated algorithm implemented in the digital circuit.
approximated_values = zeros(max_value);

```

```
%p represents the real part
%q represents the imaginary part

for p = 1:max_value
    for q = 1:max_value
        exact_values(p,q) = sqrt(p^2 + q^2);
        approximated_values(p,q) = max(p,q) + floor((1/2)*min(p,q)) -
                                   floor((1/16)*(max(p,q) + min(p,q)));
    end
end

%Mean Squared Error
MSE = immse(exact_values,approximated_values);
```

The script was run for values of *Nbit* between [3, 10], in this way it has been possible to examine the variation of MSE w.r.t the number of bits used for the inputs.

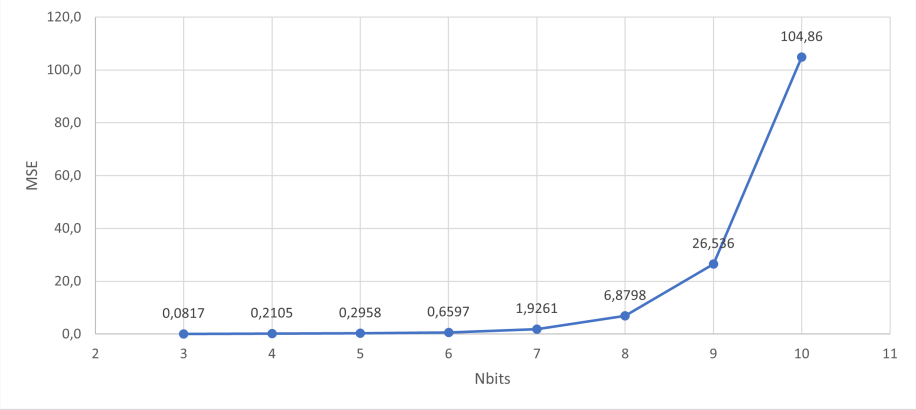

Figure 4: Mean Squared Error

As can be seen from the graph, the MSE increases as the number of bits increases and this growth is mainly due to the division by 16 used in the approximation algorithm.

5 Synthesis and Implementation

Xilinx Vivado 2018.3 software was used to synthesize and implement the circuit. In particular, it was considered the ZyBo-Zynq 7000 board, which uses the **xc7z010clg400-1** as its FPGA.

5.1 Synthesis

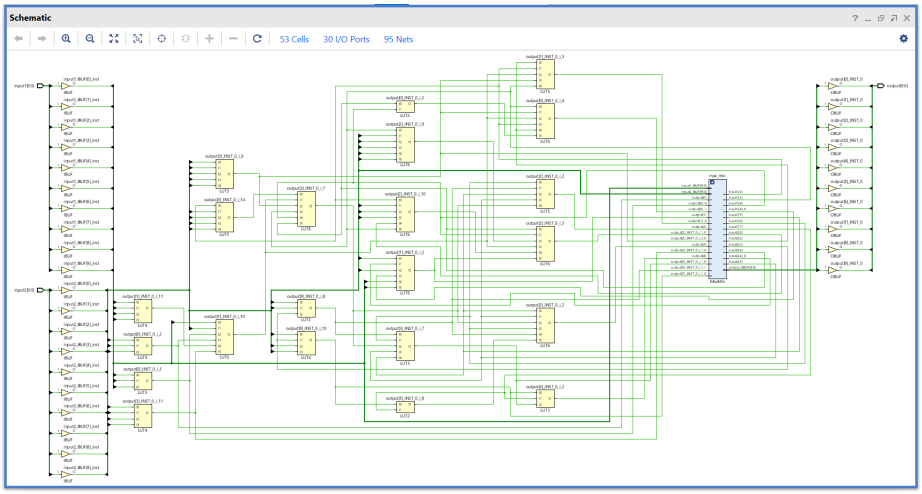

Figure 5: Synthesis design provided by Vivado

5.1.1 Warnings

During the synthesis process Vivado has reported only the following warning:

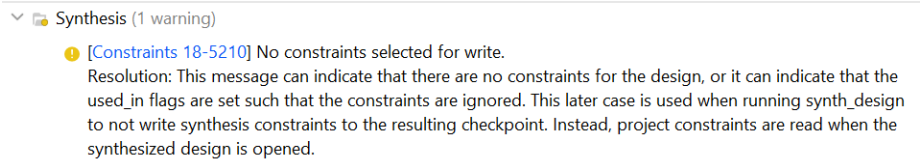

Figure 6: Synthesis warning message

As said during laboratory lectures, this warning can be ignored.
Indeed, this warning message is no longer triggered in the upcoming release of Vivado (2020.1).

5.1.2 Timing constraints

Timing constraints are not available because the circuit is a fully combinatorial network and does not need the clock to function properly.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	NA	Worst Hold Slack (WHS):	NA	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	NA	Total Hold Slack (THS):	NA	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	NA	Number of Failing Endpoints:	NA	Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA	Total Number of Endpoints:	NA	Total Number of Endpoints:	NA

All user specified timing constraints are met.

Figure 7: Timing constraints not available

For this reason it is not possible to evaluate the maximum operating frequency of the circuit and the critical paths.

5.1.3 Power Analysis

Total on-chip power is the power consumed internally within the FPGA, in this case is equal to 105 mW.

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.105 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,2°C**
Thermal Margin:	58,8°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

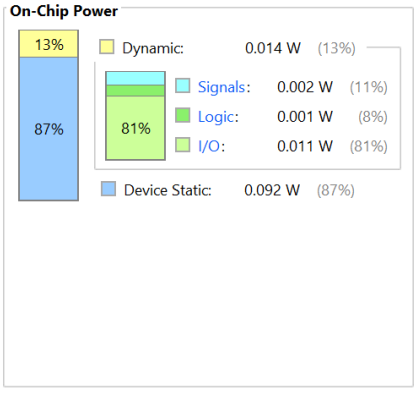

Figure 8: Power summary provided by Vivado

Device static power is the power required for the FPGA to operate normally and it corresponds to the 87% of the total power. This means that a consistent part of the energy is used to keep the device on. The remaining 13% is the Dynamic Power that is mostly required by the I/O operations, as expected.

5.1.4 Utilization

The resources used by the circuit are the following:

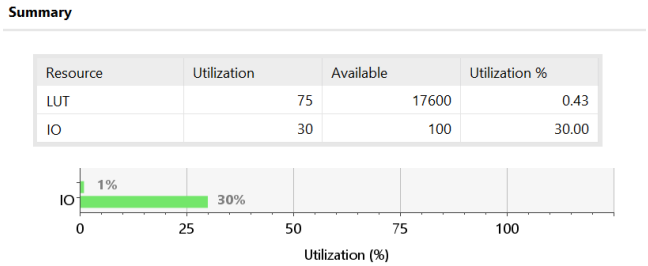

Figure 9: Utilization summary provided by Vivado

As can be seen from the figure, most of the used resources are related to I/O, this comes from the fact that the circuit has two 10-bit inputs and one 10-bit output.

5.2 Implementation

The following figure shows the area of the FPGA occupied by the implemented circuit:

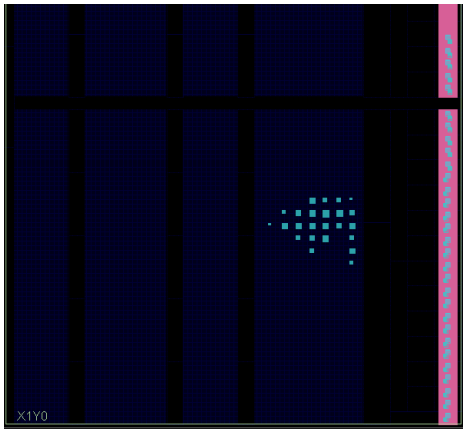

Figure 10: Implementation design provided by Vivado on Zybo board