University of Pisa
MSc in Computer Engineering
Elettronics Systems

Linear Interpolator implementation in VHDL language

Leonardo Fontanelli

Academic Year 2020/2021

Contents

1 Introduction

1.1 Description of the assignment

The purpose of this project is to design a digital circuit that, having as an input a sampled signal, with a sampling period equal to T, gives as an output an interpolated version of the input signal.
Defining L as the *interpolation factor*, for every sample, the system will introduce $L - 1$ points.

$$y(nT + uT) = (y_{n+1} - y_n) * u + y_n$$

$$u = \frac{k}{L}, k \in \{0, 1, \dots, L - 1\}$$

For this project, we can consider $L = 4$ and to work with 16 bit representation. Hence, with the assumptions we have made, it is possible to achieve an interpolation, giving 1 point at a time, the system will give as output 3 points that approximate the curve between the 2 points.

1.2 Example of the algorithm

As an example to clarify the algorithm, we can compute the interpolation points of $y_n = 80$ and $y_{n+1} = 247$

$$k = 0, u = \frac{0}{4} \text{ then } y = (247 - 80) * \frac{0}{4} + 80 = 80$$

$$k = 1, u = \frac{1}{4} \text{ then } y = (247 - 80) * \frac{1}{4} + 80 = 121$$

$$k = 2, u = \frac{2}{4} \text{ then } y = (247 - 80) * \frac{2}{4} + 80 = 163$$

$$k = 3, u = \frac{3}{4} \text{ then } y = (247 - 80) * \frac{3}{4} + 80 = 205$$

As we can see from this example, the results are approximated when there are rests in the calculation. So we always obtain a positive integer as a result.
With the example, we can see that starting from 2 points $(80, 247)$, we've obtained other 4 points $80, 121, 163, 205$ that represents the interpolated points to connect the input ones.

1.3 Simplifications of the algorithm

Giving a first look to the assignment and the proposed example, it is needed a simplification in order to design in a simpler way the system. The best way in order achieve this is to have only division by 2 and 4 in the formula. Hence, the algorithm has been simplified:

$$output = (y_{n+1} - y_n) * u + y_n$$

$$output = uy_{n+1} - uy_n + y_n$$

$$output = \frac{k}{L}y_{n+1} - \frac{k}{L}y_n + y_n$$

With $k = 1, 2, 3$ the equation gives the following results:

$$k = 0, output = y_n$$

$$k = 1, output = \frac{y_{n+1}}{4} - \frac{y_n}{4} + y_n$$

$$k = 2, output = \frac{y_{n+1}}{2} - \frac{y_n}{2} + y_n$$

$$k = 3, output = \frac{3}{4}y_{n+1} - \frac{3}{4}y_n + y_n = y_{n+1} - \frac{y_{n+1}}{4} + \frac{y_n}{4}$$

It is necessary to specify, that the simplifications have been made with the purpose of working with 16 bit numbers and L = 4.

2 Architecture Description

The architecture of the Linear Interpolator is the following:

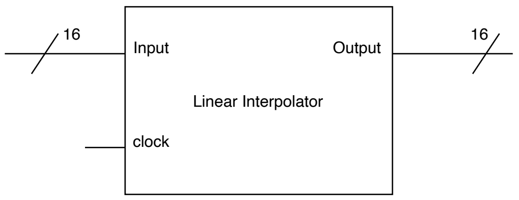


Figure 1: Architecture of the Linear Interpolator

- *Input:* is a 16 bit entry that takes the binary representation of the number, coming from the sampled signal.
- *Output:* is a 16 bit exit of the net that gives the interpolated points for the sampled signal.
- *Clock:* represent the input clock signal.

This net has an input the sampled signal represented on 16 bits, and gives as output the interpolated signal. Hence, given two consecutive sampled points, the system should return as output three points that connects the inputs calculated with the formula saw in the previous section.
For further details of the architecture, will be given in the next section with the VHDL implementation.

3 VHDL implementation

In this section will be analyzed the VHDL implementation in details, by showing all the code for every .vhd file generated.

3.1 Linear Interpolator

The main component of the net is the Linear Interpolator, it is composed by some other internal nets.

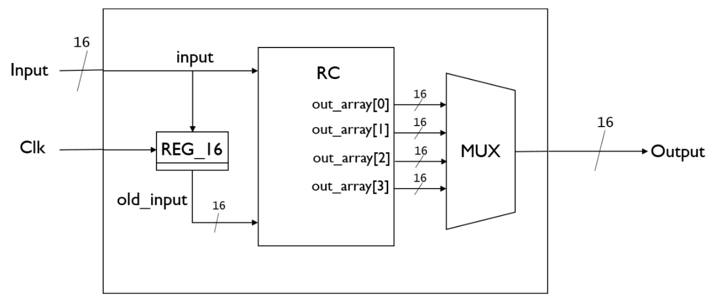


Figure 2: Detailed Architecture of the Linear Interpolator

- *REG_16*: it's a 16 bits register, which it's useful for storing the previous input of the net and give as output the interpolated points.
- *RC*: it's the net that calculates the outputs of the linear interpolator.
- *MUX*: this is a multiplexer that select the correct output, and it's piloted by the clock.

```vhdl
-----------------------------
--   Linear Interpolator   --
-----------------------------
library IEEE;
use ieee.numeric_std.all;
use IEEE.std_logic_1164.all;

entity LI is
  generic (n : positive := 16);
  port(
    input : in std_ulogic_vector(n-1 downto 0);
    output  : out std_ulogic_vector(n-1 downto 0);
    clk   : in std_ulogic
  );
end LI;

architecture beh of LI is

    -----------------------------
    --  Component Declaration  --
    -----------------------------

    -- This register is the one that saves the previous input
  component REG_16 is
    port (
      d_reg : in std_ulogic_vector(n-1 downto 0);
      rst   : in std_ulogic;
      clk   : in std_ulogic;
      q_reg : out std_ulogic_vector(n-1 downto 0)
    );
  end component REG_16;

    -- Multiplexer that select the correct output
  component MUX is
    port(
      sel : in  std_ulogic_vector(1 downto 0);
          x1  : in  std_ulogic_vector(n-1 downto 0);
          x2  : in  std_ulogic_vector(n-1 downto 0);
          x3  : in  std_ulogic_vector(n-1 downto 0);
          x4  : in  std_ulogic_vector(n-1 downto 0);
          y   : out std_ulogic_vector(n-1 downto 0)
    );
    end component MUX;

    -- Combinatorial net that calculates the outputs.
    -- The inputs are the current point and the previous one.
  component RC is
  port(
    input : in std_ulogic_vector(n-1 downto 0);
    old_in  : in std_ulogic_vector(n-1 downto 0);
    out0  : out std_ulogic_vector(n-1 downto 0);
    out1  : out std_ulogic_vector(n-1 downto 0);
    out2  : out std_ulogic_vector(n-1 downto 0);
    out3  : out std_ulogic_vector(n-1 downto 0)
  );
  end component RC;

    -- Declaration of the array of string type
  type array_of_string is array (0 to 3) of std_ulogic_vector(15 downto 0);

    -----------------------------
    --         Signals         --
    -----------------------------

  signal old_input  :   std_ulogic_vector(15 downto 0); -- to store the previous input value
  signal sel_s    : std_ulogic_vector(1 downto 0) := "11"; -- multiplexer selector
  signal rst      : std_ulogic := '0';
  -- Output Signal - Save the 4 outputs
  signal out_array  : array_of_string;

  begin
    -- register to update old_input
    reg: REG_16 port map(input, rst, clk, old_input);
```

```
    -- Calculates the output
    res: RC port map(input, old_input, out_array(0), out_array(1),
    out_array(2), out_array(3));

    COUNTER_P : process(clk)  -- process to count clocks
    begin
      if(rising_edge(clk)) then
         case sel_s is
              when "00" => sel_s <= "01";
              when "01" => sel_s <= "10";
              when "10" => sel_s <= "11";
              when "11" => sel_s <= "00";
          when others => sel_s <= "00";
        end case;
      end if;
    end process;

    -- multiplexer to select correct output
    choose: MUX port map(sel_s, out_array(0), out_array(1),
    out_array(2), out_array(3), output);

end beh;
```

3.2 16 bits Register

As already said, this register is the one that stores the input and it is composed by 16 Flip Flop D, one for every bit.

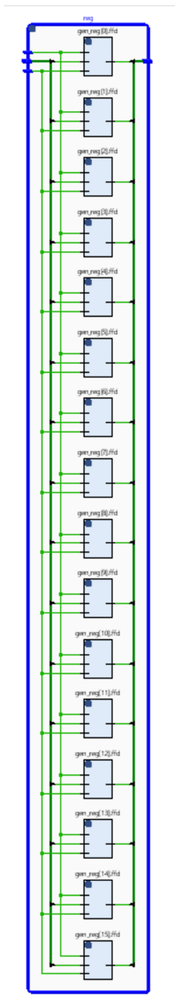


Figure 3: 16 bits register architecture

```
--16 bit register
library IEEE;
use IEEE.std_logic_1164.all;

-----------------------------
--   Entity declaration    --
-----------------------------

entity REG_16 is
  generic (n : positive := 16);
  port (
    d_reg : in std_ulogic_vector(n-1 downto 0);
    rst   : in std_ulogic;
    clk   : in std_ulogic;
```

```vhdl
    q_reg : out std_ulogic_vector(n-1 downto 0)
  );
end REG_16;

-----------------------------
-- Architecture definition --
-----------------------------

architecture str of REG_16 is

  component FF_D
    port (
      d : in std_ulogic;
      clk : in std_ulogic;
      rst : in std_ulogic;
      q : out std_ulogic
    );
  end component;

  signal q_sig  : std_ulogic_vector(n-1 downto 0);
  signal rst_sig  : std_ulogic;
  begin
    gen_reg:
      for i in 0 to n-1 generate
          ffd : FF_D port map(d_reg(i), clk, rst, q_sig(i));
      end generate gen_reg;
    q_reg <= q_sig;
end str;
```

3.3 Combinatorial Net

The Combinatorial Net is the one that computes the output of the Linear Interpolator, by using the formula computed before:

$$k = 0, output = y_n$$

$$k = 1, output = \frac{y_{n+1}}{4} - \frac{y_n}{4} + y_n$$

$$k = 2, output = \frac{y_{n+1}}{2} - \frac{y_n}{2} + y_n$$

$$k = 3, output = \frac{3}{4}y_{n+1} - \frac{3}{4}y_n + y_n = y_{n+1} - \frac{y_{n+1}}{4} + \frac{y_n}{4}$$

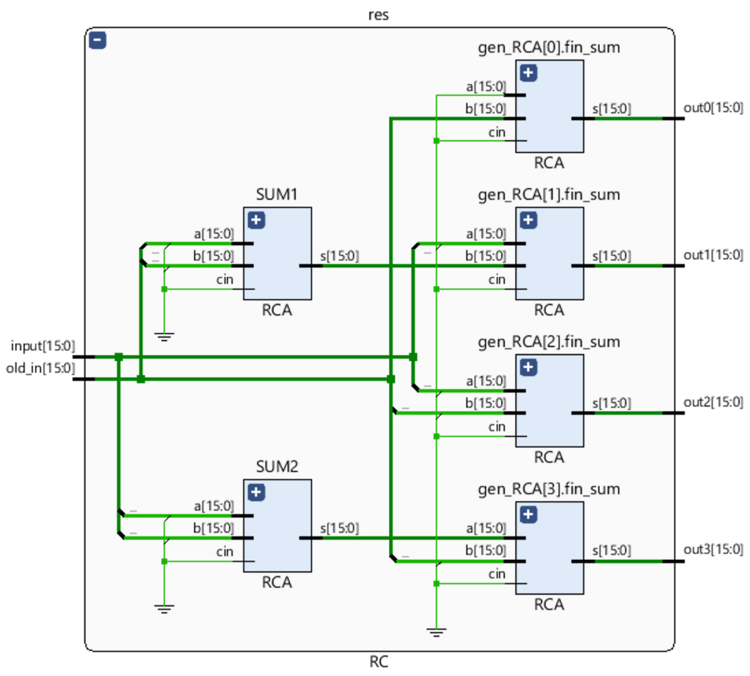


Figure 4: Combinatorial Net architecture

The two input signals (input and old_in, the latter one coming from the 16 bits register) are shifted in order to obtain other 4 signals that are: *i1sh2, i1sh4, i2sh2, i2sh4* that represent the shift of the bits (2 and 4) in order to compute the division.
Then two arrays *yn1* and *yn2* are initialized with the values that must be summed. And the summed will be provided by four Ripple Carry Adders that gives the final output of the net. The code of this component is the following:

```vhdl
-- Intermediate combinatorial net to calculate the four output
    values
library IEEE;
use ieee.numeric_std.all;
use IEEE.std_logic_1164.all;

entity RC is
  generic (n : positive := 16);
  port(
    input : in std_ulogic_vector(n-1 downto 0);
    old_in  : in std_ulogic_vector(n-1 downto 0);
    out0   : out std_ulogic_vector(n-1 downto 0);
    out1   : out std_ulogic_vector(n-1 downto 0);
    out2   : out std_ulogic_vector(n-1 downto 0);
    out3   : out std_ulogic_vector(n-1 downto 0)
  );
end RC;

```

```vhdl
architecture rtl of RC is

  -----------------------------
    --  Component Declaration  --
    -----------------------------

  component RCA is         -- net to sum signals
    port (
      a : in std_ulogic_vector(n-1 downto 0);
      b : in std_ulogic_vector(n-1 downto 0);
      cin : in std_ulogic;
      s : out std_ulogic_vector(n-1 downto 0);
      cout: out std_ulogic
    );
  end component RCA;

  -- Declaration of the array of string type
  type array_of_string is array (0 to 3) of std_ulogic_vector(15 downto 0);

  signal out_array  : array_of_string;  -- to save the outputs of RCAs
  signal yn1      : array_of_string;
  signal yn2      : array_of_string;

  -- signals to store shifted input
  signal i1sh2    : std_ulogic_vector(15 downto 0);   -- first input shifted by one position (divided by two)
  signal i2sh2    : std_ulogic_vector(15 downto 0);   -- second input shifted by one position

  signal i1sh4    : std_ulogic_vector(15 downto 0);   -- first input shifted by two positions (divided by four)
  signal i2sh4    :   std_ulogic_vector(15 downto 0);   -- second input shifted by two positions

  -- signals for the rests
  signal s_cout   : std_ulogic_vector(3 downto 0);
  signal s_cin, s_cout1, s_cout2  :   std_ulogic := '0';

  begin

    main: process(input)
    begin
      -- in1 and in2 shifted (to divide per two)
      i1sh2(15) <= '0'; i1sh2(14 downto 0) <= input(15 downto 1);
      i2sh2(15) <= '0'; i2sh2(14 downto 0) <= old_in(15 downto 1);

      -- in1 and in2 double shifted (to divide per four)

      i1sh4(15) <= '0'; i1sh4(14) <= '0'; i1sh4(13 downto 0) <= input(15 downto 2);
      i2sh4(15) <= '0'; i2sh4(14) <= '0'; i2sh4(13 downto 0) <= old_in(15 downto 2);

      yn2(0) <= old_in; -- yn2(0) must take the 'actual' old_in
    end process;

    -- u = 0
    yn1(0) <= "0000000000000000";

    -- u = 1/4
    yn1(1) <= i1sh4;
    SUM1 : RCA port map(i2sh4, i2sh2, s_cin, yn2(1), s_cout1);

    -- u = 2/4
    yn1(2) <= i1sh2;
    yn2(2) <= i2sh2;

    -- u = 3/4
    yn2(3) <= i2sh4;
    SUM2 : RCA port map(i1sh4, i1sh2, s_cin, yn1(3), s_cout2);

    -- 4 Ripple Carry Adders to generate the four output signals
    gen_RCA:
    for i in 0 to 3 generate
      fin_sum : RCA port map(yn1(i), yn2(i), s_cin, out_array(i), s_cout(i));
    end generate gen_RCA;

    out0 <= out_array(0);
    out1 <= out_array(1);
    out2 <= out_array(2);
    out3 <= out_array(3);

end rtl;
```

3.4 Multiplexer 4to1

The multiplexer is used as selector for the output, it is guided by the 2 bits input *sel*, that selects the correct output.

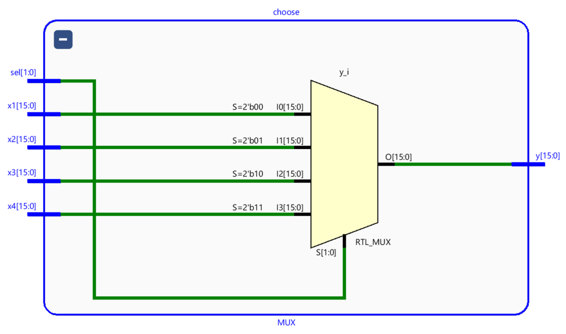


Figure 5: Multiplexer architecture

```
-- Multiplexer 4to1
library IEEE;
```

```vhdl
use IEEE.STD_LOGIC_1164.ALL;

entity MUX is
  generic (n : positive := 16);
    port (
    sel : in  std_ulogic_vector(1 downto 0);
        x1  : in  std_ulogic_vector(n-1 downto 0);
        x2  : in  std_ulogic_vector(n-1 downto 0);
        x3  : in  std_ulogic_vector(n-1 downto 0);
        x4  : in  std_ulogic_vector(n-1 downto 0);
        y   : out std_ulogic_vector(n-1 downto 0)
  );

end MUX;

architecture beh of MUX is
begin
  main: process (sel) is
  begin
    case sel is
      when "00" => y <= x1;
      when "01" => y <= x2;
      when "10" => y <= x3;
      when "11" => y <= x4;
      when others => y <= "0000000000000000";
    end case;
  end process;
end beh;
```

3.5 Flip Flop D

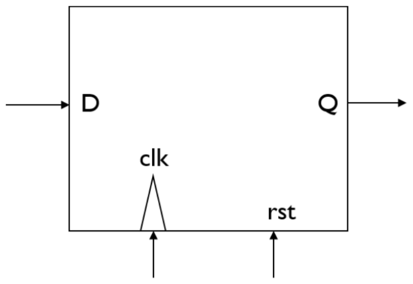


Figure 6: Flip Flop architecture

```vhdl
-- Flip Flop D
library IEEE;
use IEEE.std_logic_1164.all;

entity FF_D is
  port (
    d : in std_ulogic;
    clk :   in std_ulogic;
    rst : in std_ulogic;
    q :   out std_ulogic
  );
end FF_D;

architecture beh of FF_D is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if (rst = '1') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end beh;
```

3.6 Ripple Carry Adder

Using a bit a bit xor, the Ripple Carry Adder gives as output the summation of the inputs *a* and *b*.

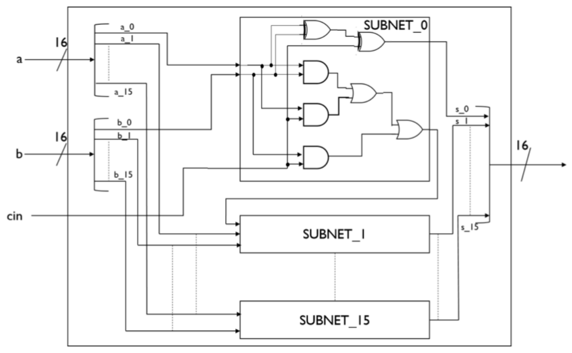


Figure 7: Ripple Carry Adder architecture

```
-- Ripple Carry Adder: it adds a and b std_ulogic_vectors of 16
    bits giving the sum s
library IEEE;
use IEEE.std_logic_1164.all;

entity RCA is
  generic (n : positive := 16);
  port (
    a : in std_ulogic_vector(n-1 downto 0);
    b : in std_ulogic_vector(n-1 downto 0);
```

```vhdl
    cin : in std_ulogic;
    s : out std_ulogic_vector(n-1 downto 0);
    cout: out std_ulogic
  );
end RCA;

architecture beh of RCA is

begin
  combinational: process(a,b,cin)
  variable c: std_ulogic;
  begin
    c := cin;
    for i in 0 to n-1 loop
      s(i)<= a(i) xor b(i) xor c;
      c   := (a(i) and b(i)) or (a(i) and c) or (b(i) and c);
    end loop;
    cout <= c;
  end process combinational;
end beh;
```

4 Testplan and Testbench

In order to verify the correctness of the architecuture, it has been developed 2 Python scripts and a VHDL testbench in order to compare the results and verify the results.

4.1 Python Scripts

The Python script are the *"linearInterpolator.py"* and *"inputGenerator.py"*. The first one is the implementation of the Linear Interpolator in Python and computes the points of interpolation. It takes as input a file named *"input_points.txt"*, computes the interpolation points and then plots them and saves in a file named *"output.txt"* the results. In the plot we can recognize the input points with a blue point and the output (interpolated points) with a orange "x". The following is the Pyhton code of the script:

```python
import matplotlib.pyplot as plt
import numpy as np

input_points = np.loadtxt("input_points.txt", delimiter=',', dtype=
    np.int)
x_coords = []
output = []
y_coords = []

L = 4  # interpolation factor
T = 1  # period time
j = 0
i = 0

for i in range(0, len(input_points) - 1):
    x_coords.append(T * (i - 1))
    k = 0  # first point
    output.append(input_points[i])
    y_coords.append(((i - 1) * T + (k / L) * T))  # calculate the
    coordinates of the interpolated points
    print("Element ", j, ": ", input_points[i])
    j += 1
    k = 1  # second point
    element = np.math.floor(input_points[i + 1] / 4) + np.math.
    floor(input_points[i] / 2) + np.math.floor(input_points[i] / 4)
    output.append(element)
    y_coords.append(((i - 1) * T + (k / L) * T))  # calculate the
    coordinates of the interpolated points
    print("Element ", j, ": ", element)
    j += 1
    k = 2  # third point
    element = np.math.floor(input_points[i+1] / 2) + np.math.floor(
    input_points[i] / 2)
    output.append(element)
    y_coords.append(((i - 1) * T + (k / L) * T))  # calculate the
    coordinates of the interpolated points
    print("Element ", j, ": ", element)
    j += 1
    k = 3  # fourth point
    element = np.math.floor(input_points[i] / 4) + np.math.floor(
    input_points[i+1] / 4) + np.math.floor(input_points[i+1] / 2)
    output.append(element)
    y_coords.append(((i - 1) * T + (k / L) * T))  # calculate the
    coordinates of the interpolated points
    print("Element ", j, ": ", element)
    j += 1
x_coords.append(T*i)  # calculate the coordinates of the input
    points

plt.plot(x_coords, input_points, marker='o')
plt.plot(y_coords, output, marker='x')
plt.show()

outputFile = open("output.txt", 'w')
for i in range(0, len(output)):
    outputFile.write("Point:" + str(i+1) + '{:>10}  {:>10}'.format(
    str(output[i]), str('{0:016b}'.format(output[i]))) + "\n")
outputFile.close()
```

The second Python script reads the *"input_points.txt"* and generates a file "input.txt" that will be copied in the Testbench code. The output of this code is the binary representation of the numbers in the *"input_points.txt"*.

```
import numpy as np

input_points = np.loadtxt("input_points.txt", delimiter=',', dtype=
    np.int)
inputFile = open("input.txt", 'w')
inputFile.write("(")
for i in range(len(input_points)):
    element = '"' + '{0:016b}'.format(input_points[i]) + '"'
    if i != len(input_points) - 1:
        element += ", "
    inputFile.write(element)
inputFile.write(');')
inputFile.close()
```

4.2 Testbench

The Testbench code is the following, it has in the *s_input* signal the input generated by the Python script presented before. This signal is the input of the Linear Interpolator, and they are presented to the net every 4 clock cycle.

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

entity LI_tb is
end LI_tb;

------------------------------
-- Architecture definition --
------------------------------
```

```

architecture test of LI_tb is

  component LI is
    generic(n : positive := 16);
    port (
      input : in std_ulogic_vector(n-1 downto 0);

      output  : out std_ulogic_vector(n-1 downto 0);
      clk    : in std_ulogic
    );
  end component;

-----------------------------
--          Signals          --
-----------------------------
  constant clk_p  :  time     := 8 ns;  -- clock period

-- INPUT SIGNALS
  type array_of_input is array (0 to 15) of std_ulogic_vector(15
    downto 0);
  signal s_input  : array_of_input := ("0000000000000111", "
    0000000000010011", "0000100000011010", "0000000010100001", "
    0000000010001100", "0000000000100000", "0000000000101001", "
    0000000001010111", "0000001001101011", "0000001100100100", "
    0000000011110001", "0000000000111111", "0000000001001010", "
    0000000001001110", "0000010101111111", "0000010101110101");
  signal s_in   : std_ulogic_vector(15 downto 0) := "
    0000000000000111";
  signal s_clk  :   std_ulogic := '0';
  signal i    : integer := 0;

-- OUTPUT SIGNAL
  signal s_out  : std_ulogic_vector(15 downto 0);

  begin

    iDUT : LI port map(s_in, s_out, s_clk);
    s_clk <= not s_clk after clk_p/2;
    i <= i+1 after 4*clk_p;             --input must change every 4
     clock signals
    s_in <= s_input(i);

end test;
```

In order to have a correct behaviour of the net is necessary to change the input every 4 clock periods.

4.3 Simulation

In order to simulate and compare the Python output with the VHDL output, the input number will be the same.
Hence the selected input will be:

$$7, 19, 2074, 161, 140, 32, 41, 87, 619, 804, 241, 63, 74, 78, 1407, 1397$$

inserted in the *"input_points.txt"* file.
The following is a piece of simulation, performed with the Vivado software:

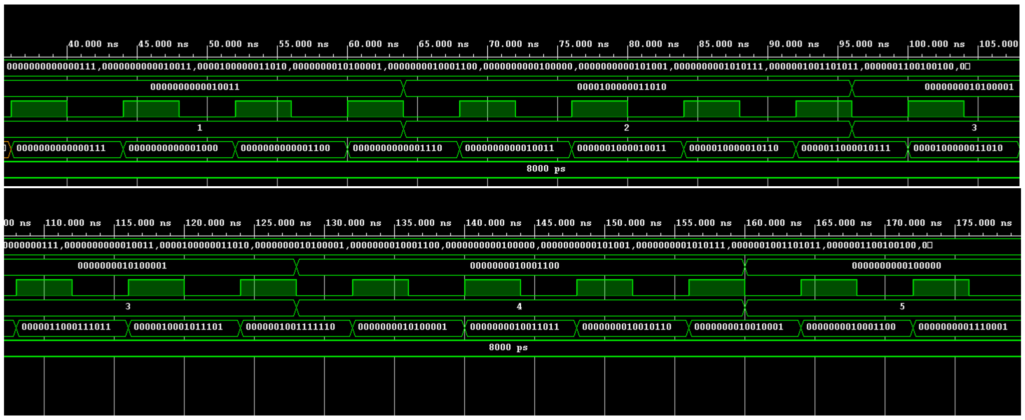
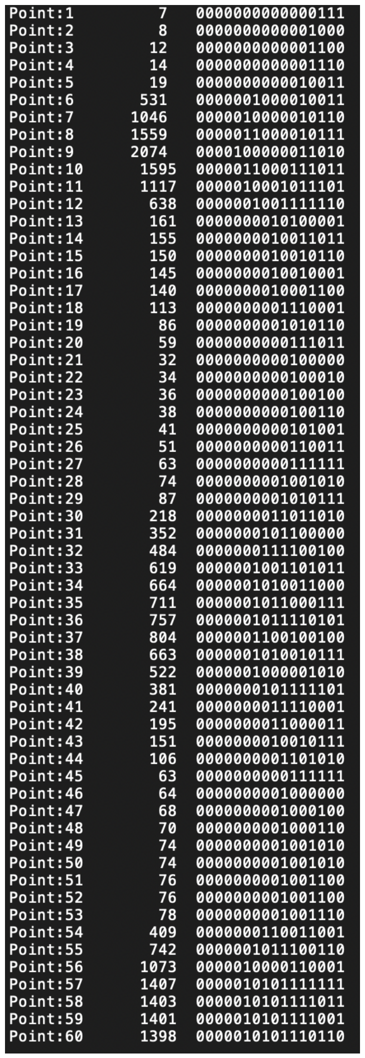


Figure 8: The simulation of the net

The output is the penultimate signal in the images.
Then it has been performed the execution of the Python script, the output is the following:



```
Point:1         7  0000000000000111
Point:2         8  0000000000001000
Point:3        12  0000000000001100
Point:4        14  0000000000001110
Point:5        19  0000000000010011
Point:6       531  0000001000010011
Point:7      1046  0000010000010110
Point:8      1559  0000011000010111
Point:9      2074  0000100000011010
Point:10      1595  0000011000111011
Point:11      1117  0000010001011101
Point:12       638  0000001001111110
Point:13       161  0000000010100001
Point:14       155  0000000010011011
Point:15       150  0000000010010110
Point:16       145  0000000010010001
Point:17       140  0000000010001100
Point:18       113  0000000001110001
Point:19        86  0000000001010110
Point:20        59  0000000000111011
Point:21        32  0000000000100000
Point:22        34  0000000000100010
Point:23        36  0000000000100100
Point:24        38  0000000000100110
Point:25        41  0000000000101001
Point:26        51  0000000000110011
Point:27        63  0000000000111111
Point:28        74  0000000001001010
Point:29        87  0000000001010111
Point:30       218  0000000011011010
Point:31       352  0000000101100000
Point:32       484  0000000111100100
Point:33       619  0000001001101011
Point:34       664  0000001010011000
Point:35       711  0000001011000111
Point:36       757  0000001011110101
Point:37       804  0000001100100100
Point:38       663  0000001010010111
Point:39       522  0000001000001010
Point:40       381  0000000101111101
Point:41       241  0000000011110001
Point:42       195  0000000011000011
Point:43       151  0000000010010111
Point:44       106  0000000001101010
Point:45        63  0000000000111111
Point:46        64  0000000001000000
Point:47        68  0000000001000100
Point:48        70  0000000001000110
Point:49        74  0000000001001010
Point:50        74  0000000001001010
Point:51        76  0000000001001100
Point:52        76  0000000001001100
Point:53        78  0000000001001110
Point:54       409  0000000110011001
Point:55       742  0000001011100110
Point:56      1073  0000010000110001
Point:57      1407  0000010101111111
Point:58      1403  0000010101111011
Point:59      1401  0000010101111001
Point:60      1398  0000010101110110
```

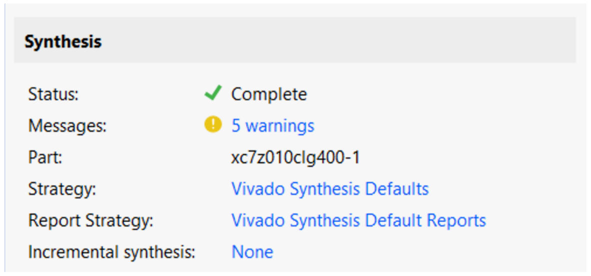
Figure 9: The Python Output

By comparing the binary output of the Vivado simulation and the Python script, we can clearly see the points are the same. Hence we can conclude that the results of the net are verified.

5 Vivado Logic Synthesis and Implementation

5.1 Synthesis

At first, it has been performed the synthesis with the Vivado simulator tool. The first simulation has been performed without constraints in order to check the VHDL consistency: it concluded correctly, with some warning errors.

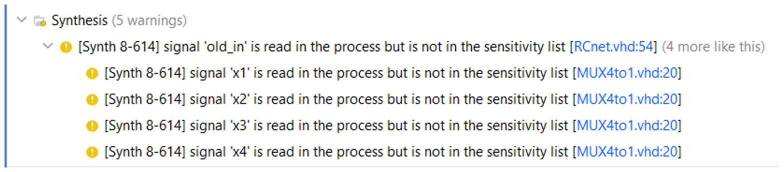


Synthesis

Status:	✓ Complete
Messages:	5 warnings
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Figure 10: The result of the simulation

The warnings that came out from the synthesis were the following:



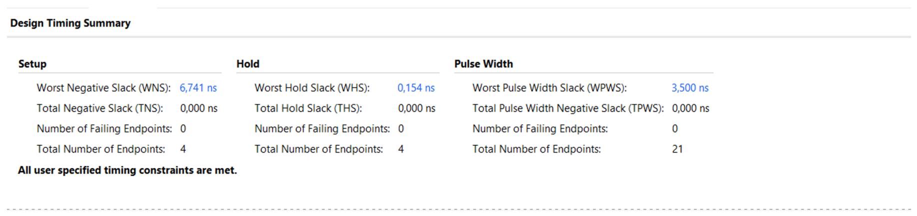
```
Synthesis (5 warnings)
  [Synth 8-614] signal 'old_in' is read in the process but is not in the sensitivity list [RCnet.vhd:54] (4 more like this)
    [Synth 8-614] signal 'x1' is read in the process but is not in the sensitivity list [MUX4to1.vhd:20]
    [Synth 8-614] signal 'x2' is read in the process but is not in the sensitivity list [MUX4to1.vhd:20]
    [Synth 8-614] signal 'x3' is read in the process but is not in the sensitivity list [MUX4to1.vhd:20]
    [Synth 8-614] signal 'x4' is read in the process but is not in the sensitivity list [MUX4to1.vhd:20]
```

Figure 11: Warnings from the synthesis

Those warnings indicates that some signal were declared in a process without being inserted in the sensitivity list. Given that, in the sensitivity list must be declared anything that the process needs to know about changes of, we can discard the warning given that we are not interested in the changes of the signals, the process doesn't need to be sensible to those signals.

5.1.1 Timing

For the definitive synthesis it has been added a clock constraint of $8ns$.
The results for the timing of the net are the following:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6,741 ns	Worst Hold Slack (WHS):	0,154 ns	Worst Pulse Width Slack (WPWS):	3,500 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:	4	Total Number of Endpoints:	4	Total Number of Endpoints:	21

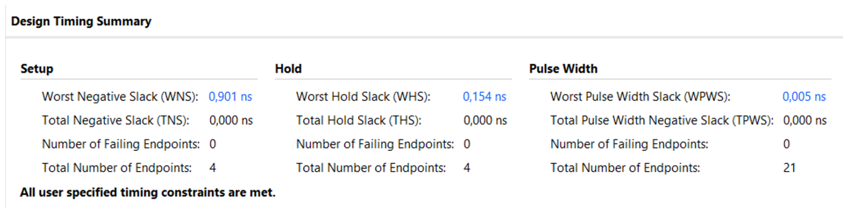
All user specified timing constraints are met.

Figure 12: The timing results for the synthesis

Where:

- *Worst Negative Slack - WNS*: is the critical path, corresponds to the worst slack of all the timing paths for max delay analysis.
- *Total Negative Slack - TNS*: is the sum of negative slack.
- *Worst Hold Slack - WHS*: corresponds to the worst slack of all the timing paths for min delay analysis.
- *Total Hold Slack - THS*: is the sum of negative hold slack paths.
- *Worst Pulse Width Slack - WPWS*: corresponds to the worst pulse width slack for (Min low pulse width, min high pulse width, min period, max period).
- *Total Worst Pulse Width - TWPW*: is the sum of all WPWS violations.

We can notice that all the slack are positive, hence the clock period has no need to be increased and can be decreased instead.
It resulted that the minimum clock for which the slack are all grater than 0 is $2.16ms$. The resulting timing summary is the following:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,901 ns	Worst Hold Slack (WHS):	0,154 ns	Worst Pulse Width Slack (WPWS):	0,005 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:	4	Total Number of Endpoints:	4	Total Number of Endpoints:	21

All user specified timing constraints are met.

Figure 13: The timing results for the synthesis with the new clock

5.1.2 Critical path

The critical path resulted to be the following:

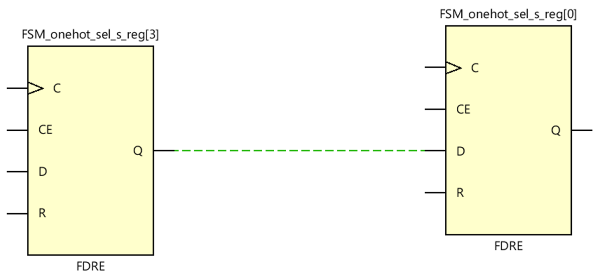
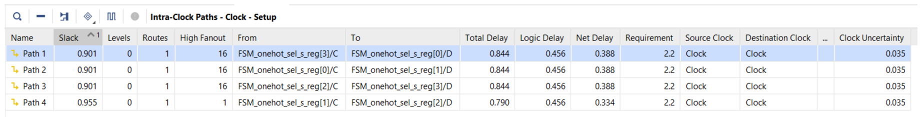


Figure 14: Critical Path

In general, the delay for the path are the following:

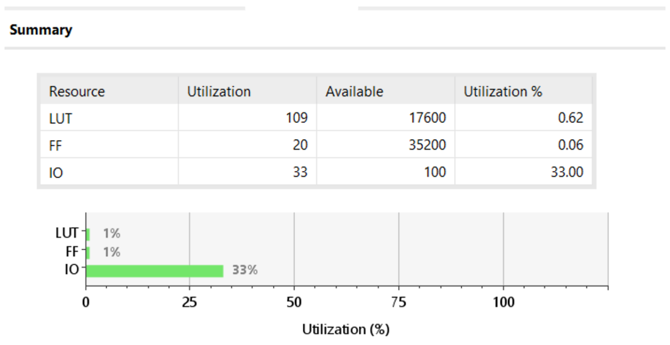

Intra-Clock Paths - Clock - Setup

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	...	Clock Uncertainty
Path 1	0.901	0	1	16	FSM_onehot_sel_s_reg[3]/C	FSM_onehot_sel_s_reg[0]/D	0.844	0.456	0.388	2.2	Clock	Clock		0.035
Path 2	0.901	0	1	16	FSM_onehot_sel_s_reg[0]/C	FSM_onehot_sel_s_reg[1]/D	0.844	0.456	0.388	2.2	Clock	Clock		0.035
Path 3	0.901	0	1	16	FSM_onehot_sel_s_reg[2]/C	FSM_onehot_sel_s_reg[3]/D	0.844	0.456	0.388	2.2	Clock	Clock		0.035
Path 4	0.955	0	1	1	FSM_onehot_sel_s_reg[1]/C	FSM_onehot_sel_s_reg[2]/D	0.790	0.456	0.334	2.2	Clock	Clock		0.035

Figure 15: Delay for every path of the net

5.1.3 Utilization

The utilization of the net resulted to be the following:


Summary

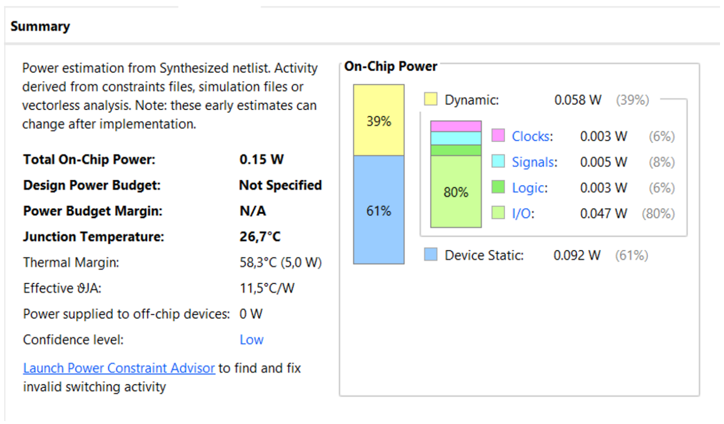
Resource	Utilization	Available	Utilization %
LUT	109	17600	0.62
FF	20	35200	0.06
IO	33	100	33.00

Figure 16: Synthesis Utilization

In this scheme, we can see that 33/100 of the utilization are used for the I/O since there is 16 bit for input, 16 bits for output plus the clock bit.

5.1.4 Power Consumption

Regarding the power consumption, it has been obtained the following result:


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:	**0.15 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,7°C**
Thermal Margin:	58,3°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

On-Chip Power

Dynamic:	0.058 W	(39%)	
	Clocks:	0.003 W	(6%)
	Signals:	0.005 W	(8%)
	Logic:	0.003 W	(6%)
	I/O:	0.047 W	(80%)
Device Static:	0.092 W	(61%)	

Figure 17: Synthesis Power Consumption

We can notice that the total power consumption of the net is $0.15W$, with 39% related to Dynamic power and 61% to Static power.

5.1.5 Maximum Operating Frequency

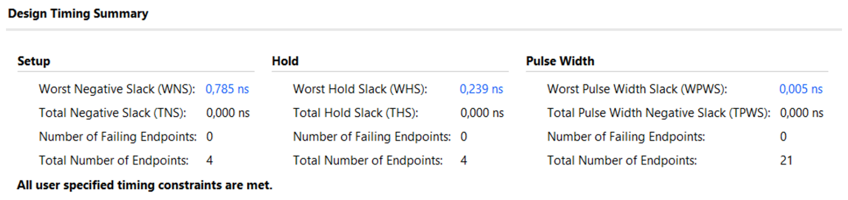
With the clock constraint equal to $2.16ms$, it can be computed the maximum operating frequency which is equal to $f = \frac{1}{T_{clock}} = \frac{1}{2.16ms} = 462.95Hz$.

5.2 Implementation

Still with the Vivado software it has been performed the implementation step. The computation perfomed well and gave the same warnings explained in the previous section.

5.2.1 Timing

Also with the implementation step it has been reported the timing of the net, which is the following:


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,785 ns	Worst Hold Slack (WHS):	0,239 ns	Worst Pulse Width Slack (WPWS):	0,005 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:	4	Total Number of Endpoints:	4	Total Number of Endpoints:	21

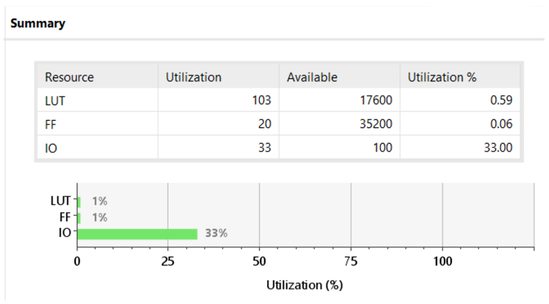
All user specified timing constraints are met.

Figure 18: The timing results for the implementation

We can notice that the slack are still positive, hence we can keep the same clock as before.

5.2.2 Utilization

The utilization resulted similar to the one of the synthesis, except for the LUT, which is decreased to 103, with 0.59% of utilization.



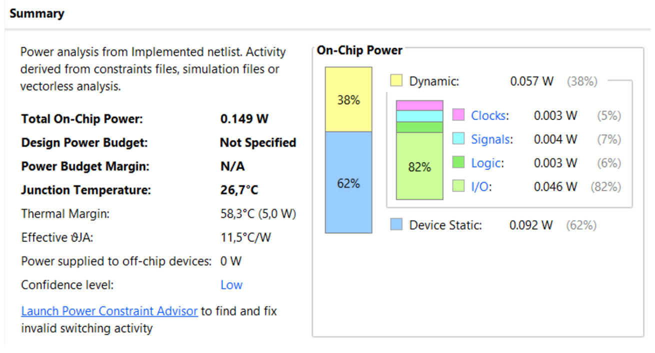
Summary

Resource	Utilization	Available	Utilization %
LUT	103	17600	0.59
FF	20	35200	0.06
IO	33	100	33.00

Figure 19: The utilization results for the implementation

5.2.3 Power Consumption

The power summary resulted slightly different from the synthesis:



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.149 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,7°C**
Thermal Margin:	58,3°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

On-Chip Power

🟨	Dynamic:	0.057 W	(38%)
🟪	Clocks:	0.003 W	(5%)
🟦	Signals:	0.004 W	(7%)
🟩	Logic:	0.003 W	(6%)
🟩	I/O:	0.046 W	(82%)
🟦	Device Static:	0.092 W	(62%)

Figure 20: The power consumption results for the implementation

In this case, the *Total On-Chip Power* resulted $0.149W$ with 38% for the Dynamic Power and 62% for the Static Power.

5.2.4 Maximum Operating Frequency

Since no problem has been found in the timing, we can conclude that the maximum operating frequency is the same than the synthesis and it is $462,95Hz$

6 Conclusions

The designed net is able to generate interpolation points giving 2 input (one actual and one old) with the following formula: $y(nT+uT) = (y_{n+1} - y_n) * u + y_n u = \frac{k}{L}, k \in \{0, 1, \ldots, L-1\}$ All the design assignment has been successfully satisfied. For the architecture description it has been used the VHDL language. The correctness of the circuit has been tested through simulations using test-benches and Python script. It has been done the logic synthesis in order to find the maximum clock frequency and the critical path. Also the implementation of the net has been done with the Vivado software.