UNIVERSITÀ DI PISA

Computer Engineering

Electronic Systems

Linear Interpolator Documentation

Project Report

Matteo Abaterusso

Academic Year: 2021/2022

Contents

1 Introduction

A **linear interpolator** is a digital circuit that implements the following function:

$$y(nT + uT) = (y_{n+1} - y_n)u + y_n \qquad u = \frac{k}{L}, k \in \{0, 1, ..., L-1\}$$

In general, given **two consecutive input signals** sampled every T, the interpolator generates L **output signals**, with a period of $\frac{T}{L}$ (for this reason L is defined **factor of interpolation**). If the two input signals represent two points on a cartesian plane, the output signals lie on the line that connects the two inputs.

1.1 Design Specifications

The interpolator shall have the following characteristics:

- **Input** and **output** are represented by 16 bit, so they can assume any integer values between 0 and 65535.
- The input signals have a **sampling period** of T, so the circuit will receive every T a new signal.
- The factor of interpolation is fixed to $L = 4$. So the **period** with which **output signals** have to be produced is $\frac{T}{4}$.

Moreover the structure of the circuit's **I/O ports** have to be the following:

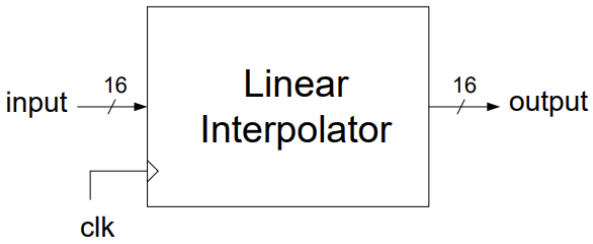

Figure 1: Circuit's I/O Ports

1.2 Useful Symbols

The following **symbol** will be used to identify the signals:

- y_{n+1}, is the last received signal.
- y_n, is the second last received signal.
- OUT_i, is one of the four output signals.

2 Architecture Description

The first problem to solve is the **complexity** of the **operations**. In fact, the linear interpolation requires **multiplications** and **divisions**, whose circuits aren't trivial. But in this case the factor interpolation is fixed to 4, and is a **power of two**. So is possible for example to exploit the **shift operation** in base two, to do the division by powers of two in a zero complexity way.

All the **simplifications** made in the calculation of all outputs are described below:

$$OUT_i = (y_{n+1} - y_n)\frac{i}{4} + y_n = y_{n+1}\frac{i}{4} - y_n\frac{i}{4} + y_n$$

$$OUT_0 = y_{n+1}\frac{0}{4} - y_n\frac{0}{4} + y_n = y_n$$

$$OUT_1 = y_{n+1}\frac{1}{4} - y_n\frac{1}{4} + y_n = y_{n+1}\frac{1}{4} + y_n\frac{3}{4} = y_{n+1}\frac{1}{4} + y_n\frac{2}{4} + y_n\frac{1}{4} = y_{n+1}\frac{1}{4} + y_n\frac{1}{2} + y_n\frac{1}{4}$$

$$OUT_2 = y_{n+1}\frac{2}{4} - y_n\frac{2}{4} + y_n = y_{n+1}\frac{2}{4} + y_n\frac{2}{4} = y_{n+1}\frac{1}{2} + y_n\frac{1}{2}$$

$$OUT_2 = y_{n+1}\frac{3}{4} - y_n\frac{3}{4} + y_n = y_{n+1}\frac{3}{4} + y_n\frac{1}{4} = y_{n+1}\frac{1}{4} + y_{n+1}\frac{2}{4} + y_n\frac{1}{4} = y_{n+1}\frac{1}{4} + y_{n+1}\frac{1}{2} + y_n\frac{1}{4}$$

So in conclusion the equations that have to be implemented are:

$$OUT_0 = y_n$$

$$OUT_1 = \frac{y_{n+1}}{4} + \frac{y_n}{2} + \frac{y_n}{4}$$

$$OUT_2 = \frac{y_{n+1}}{2} + \frac{y_n}{2}$$

$$OUT_2 = \frac{y_{n+1}}{4} + \frac{y_{n+1}}{2} + \frac{y_n}{4}$$

It's clear from the equations above that:

- There are **no subtractions**, but only summations, so it is possible to use **unsigned signals** that can be summed by a Ripple Carry Adder.
- There are **no multiplications**, but only divisions by powers of two. The signals are unsigned, so they can be implemented by **bit a bit right shifts**.

Using integer signals and divisions that ignores reminder, the results will have some **error** compared to the original formula. But this is an acceptable error, considering the huge benefits obtained in terms of circuit complexity.

2.1 Block Diagram

An high level representation of the the circuit is the following:

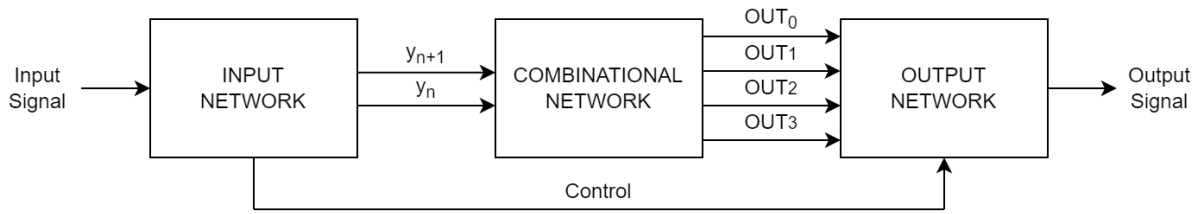

Figure 2: Block Diagram

Every block will be analized, starting from the combinational network that performs the calculation of the output signals.

2.2 Combinational Network

The combinational network is at the heart of the circuit. Given two signals in input, it provides **all four interpolated signals**:

Figure 3: Combinational Network I/O Ports

The calculations is divided into two parts:

1. It compute the **four signals needed** throw bit a bit shifts ($\frac{y_{n+1}}{4}$, $\frac{y_{n+1}}{2}$, $\frac{y_n}{4}$, $\frac{y_n}{2}$).
2. It combine the signals obtained with a set of Ripple Carry Adders, to compute the **final results**.

The second point is implemented in this way:

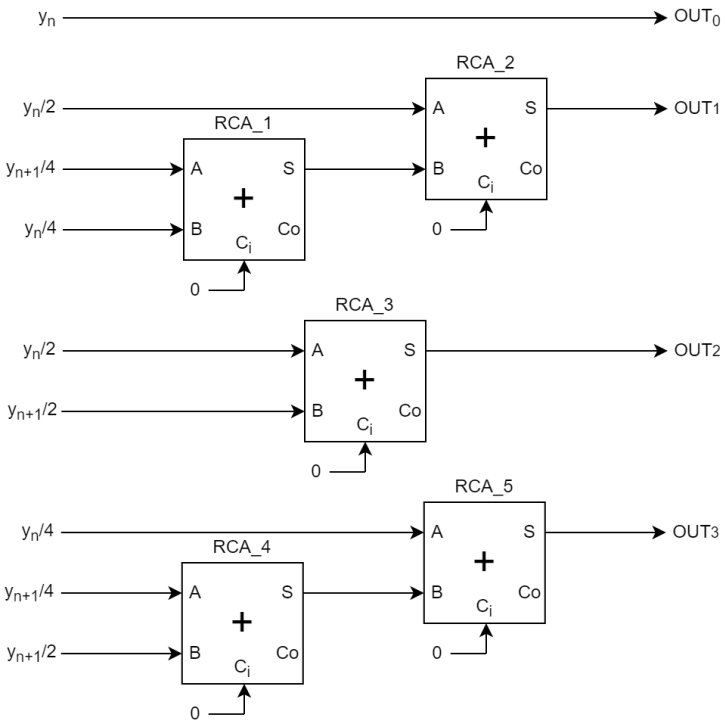

Figure 4: Output Signals Calculation

So **five Ripple Carry Adders** are needed.

2.2.1 The Synchronization Problem

Before moving to the study of input and output blocks, it is necessary to consider that the frequency with which the output signals have to be produced is **four time greater** the input signals arrival frequency.

In order to evaluate the **timing requirements** of the circuit, registers after input and before output are needed (to have always a **register-logic-register structure**). So it's clear that the output register has to be updated more frequently than the one in input.

The possible solutions to this problem are two:

- Implement **two different clock domains**, one with a period of T and one of $\frac{T}{4}$.
- Implement only the fastest clock (the one with a period of $\frac{T}{4}$), and exploit the **input register enabler port** to memorize a new signal only once every four clock cycles.

It's clear that the first solution offers the best performance, but at very high cost in terms of complexity. So the second solution has been chosen, that as before in the calculation of the output signals, is the **best trade-off** between performance and design complexity.

Now will be described how this solution has been implemented in the input network.

2.3 Input Network

The main purpose of this network is to **receive** and **memorize** every period T the **new signal**, and to **maintain it stable** for the combinational network. Moreover even the **second last input** must also be kept stable, because it is required for the interpolation as well.

To do that **two registers**, composed by 16 **D Flip Flop**, have been used. The first register (named *REG_NI*) take in input the **receiving signal**. The output of this register provides the signal y_{n+1} to the combinational network. Also it goes in input to a second register (named *REG_OI*). This second register provides to the combinational network the signal y_n. In fact, when the clock arrives *REG_NI* memorizes the new signal from the outside, and *REG_OI* memorizes the previous value contained in *REG_NI*.

The network described above looks like this:

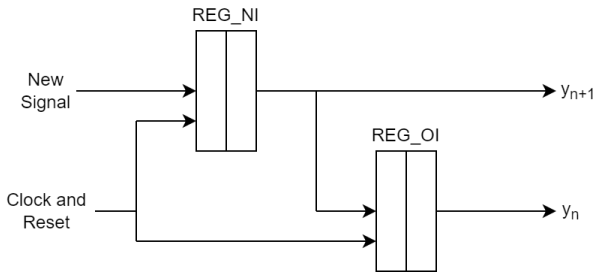

Figure 5: Input Network Registers

In order to implement the **synchronization problem solution** (section 2.2.1) is necessary to set the registers enabler port to one every four clock cycle. To do that a **2 bit counter** has been implemented, the count from 0 to 3, increasing its value of one **every clock cycle**. Assuming that a new signal arrives when the output of counter is 0, the enabler has to be one in the previous cycle, so when the counter output is 3 ($|11|_2$ in base 2). Therefore, the counter output goes in input to an **AND port**, whose output becomes the **enabler line** for the two input registers.

The resulting network is as follows:

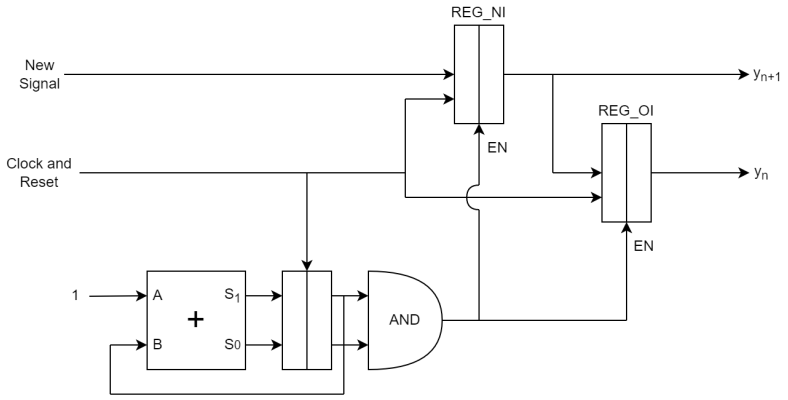

Figure 6: Total Input Network

So **one Ripple Carry Adder** and **three Registers** are needed.

2.4 Output Network

The last network of the circuit has to **output one at a time** the four signals calculated by the combinatorial network.

To do that a **multiplexer** has been used. It take in input the four signals resulting from the combinational network and provides them one at time. The output of multiplexer is controlled by the **counter** of the **input network**, whose output is exactly a line that change from 0 to 3 every clock cycle.

The output of the multiplexer goes in input to a **register** (named REG_OUT), whose output is the **output** of the **entire circuit**.

The network described above looks like this:

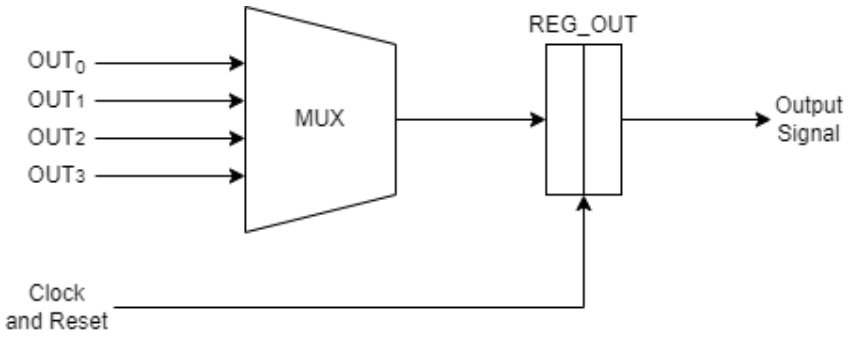

Figure 7: Output Network

For this last network **one multiplexer** and **one register** are needed.

3 VHDL Code Analysis

The VHDL analysis will be done through a **bottom-up approach**. In the first step the basic components will be analyzed, and through them the Linear Interpolator will be obtained.

3.1 Ripple Carry Adder

The Ripple Carry Adder is based on the **Full Adder**, whose implements the sum between two bits, taking into account a possible carry. The Full Adder is defined as follows:

```
----------------------------------------------------------
-- Entity
----------------------------------------------------------

entity FullAdder is
    port (
        a    : in  std_logic;
        b    : in  std_logic;
        Cin  : in  std_logic;
        S    : out std_logic;
        Cout : out std_logic
    );
end FullAdder;

----------------------------------------------------------
-- Architecture
----------------------------------------------------------

architecture rtl of fullAdder is
begin

    S    <= a xor b xor Cin;
    Cout <= (a and b) or (a and Cin) or (b and Cin);

end rtl;
```

Figure 8: Full adder

So the **Ripple Carry Adder** consists of a cascade of Full Adder:

```
----------------------------------------------------------
-- Entity
----------------------------------------------------------

entity RippleCarryAdder is
    generic (NBit : positive := 16);
    port (
        A    : in  std_logic_vector(NBit - 1 downto 0);
        B    : in  std_logic_vector(NBit - 1 downto 0);
        Cin  : in  std_logic;
        S    : out std_logic_vector(NBit - 1 downto 0);
        Cout : out std_logic
    );
end RippleCarryAdder;
```

Figure 9: Ripple Carry Adder Entity

```
----------------------------------------------------------
-- Architecture
----------------------------------------------------------

architecture rtl of RippleCarryAdder is

    -- FullAdder
    component FullAdder
        port (
            a    : in  std_logic;
            b    : in  std_logic;
            Cin  : in  std_logic;
            S    : out std_logic;
            Cout : out std_logic
        );
    end component;

    signal c_s : std_logic_vector(NBit - 1 downto 1);
```

Figure 10: Ripple Carry Adder Component and Signals

```
begin
    -- generate NBit istances of FullAdder
    GEN: for i in 1 to NBit generate
        -- first FullAdder
        FIRST_FA: if i = 1 generate
            FA1: FullAdder port map(
                a       => A(0),
                b       => B(0),
                Cin     => Cin,
                S       => S(0),
                Cout    => c_s(1)
            );
            end generate FIRST_FA;

        -- FullAdders from 2 to NBit - 1
        INTERNAL_FA: if i > 1 and i < NBit generate
            FAI: FullAdder port map(
                a       => A(i-1),
                b       => B(i-1),
                Cin     => c_s(i-1),
                S       => S(i-1),
                Cout    => c_s(i)
            );
            end generate INTERNAL_FA;

        -- last FullAdder
        LAST_FA: if i = NBit generate
            FAN: FullAdder port map(
                a       => A(i-1),
                b       => B(i-1),
                Cin     => c_s(i-1),
                S       => S(i-1),
                Cout    => cout
            );
            end generate LAST_FA;
        end generate GEN;
end rtl;
```

Figure 11: Ripple Carry Adder Internal Architecture

3.2 D Flip Flop

A **D Flip Flop** involves the use of a low active asynchronous reset. When the clock arrives it stores the new input data only if the enabler is set to one.

```
------------------------------------------------------------
-- Entity
------------------------------------------------------------

entity DFF_N is
    generic (NBit : positive := 16);
    port (
        clk     : in  std_logic;
        a_rst_n : in  std_logic;
        en      : in  std_logic;
        D       : in  std_logic_vector(NBit - 1 downto 0);
        Q       : out std_logic_vector(NBit - 1 downto 0)
    );
end DFF_N;

------------------------------------------------------------
-- Architecture
------------------------------------------------------------

architecture rtl of DFF_N is

begin
    -- flip-flop sequential logic, asyncronous reset
    DDF_N_PROC: process(clk, a_rst_n)
        begin
            if(a_rst_n = '0') then
                Q <= (others => '0');
            elsif(rising_edge(clk)) then
                if(en = '1') then
                    Q <= D;
                end if;
            end if;
        end process;
end rtl;
```

Figure 12: D Flip Flop

3.3 Counter

The **Counter** take in input clock and reset, and at every clock cycle it have to increment by one the value that provides in output.

```
------------------------------------------------------------
-- Entity
------------------------------------------------------------

entity counter is
    generic (NBit : positive := 2);
    port (
        clk     : in  std_logic;
        a_rst_n : in  std_logic;
        dout    : out std_logic_vector(NBit - 1 downto 0)
    );
end counter;
```

Figure 13: Counter Entity

Internally it is composed by a register (D Flip Flop) to save and keep stable the output value. The increment operation is implemented through a Ripple Carry Adder.

```
-------------------------------------------------------
-- Architecture
-------------------------------------------------------

architecture rtl of counter is
    -- signals
    signal rca_s    : std_logic_vector(NBit - 1 downto 0);
    signal dout_s   : std_logic_vector(NBit - 1 downto 0);

    -- constants
    constant zero   : std_logic := '0';
    constant one    : std_logic := '1';
    constant one_v  : std_logic_vector(1 downto 0) := "01";

    -- DFF for counting
    component DFF_N
        generic (NBit : positive := 16);
        port (
            clk     : in  std_logic;
            a_rst_n : in  std_logic;
            en      : in  std_logic;
            D       : in  std_logic_vector(NBit - 1 downto 0);
            Q       : out std_logic_vector(NBit - 1 downto 0)
        );
    end component;

    -- RCA for the increment
    component RippleCarryAdder
        generic (NBit : positive := 16);
        port (
            A    : in  std_logic_vector(NBit - 1 downto 0);
            B    : in  std_logic_vector(NBit - 1 downto 0);
            Cin  : in  std_logic;
            S    : out std_logic_vector(NBit - 1 downto 0);
            Cout : out std_logic
        );
    end component;
```

Figure 14: Counter Components and Signals

The register's output returns in feedback to the Ripple Carry Adder the sums it to one.

```
begin
    -- rca_s <= dout_s + one_v;
    RCA: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            a       => dout_s,
            b       => one_v,
            Cin     => zero,
            s       => rca_s
        );

    -- flip flop
    REG: DFF_N
        generic map (NBit => NBit)
        port map(
            clk     => clk,
            a_rst_n => a_rst_n,
            en      => one,
            D       => rca_s,
            Q       => dout_s
        );

    -- continuos assignment for the output
    dout  <= dout_s;
end rtl;
```

Figure 15: Counter Internal Architecture

3.4 Combinational Network

The **Combinational Network** follows the structure described in the section 2.2.

```
------------------------------------------------------------
-- Entity
------------------------------------------------------------

entity CombInterpolation is
    generic (NBit : positive := 16);
    port (
        ni      : in  std_logic_vector(NBit - 1 downto 0);
        oi      : in  std_logic_vector(NBit - 1 downto 0);
        out0    : out std_logic_vector(NBit - 1 downto 0);
        out1    : out std_logic_vector(NBit - 1 downto 0);
        out2    : out std_logic_vector(NBit - 1 downto 0);
        out3    : out std_logic_vector(NBit - 1 downto 0)
    );
end CombInterpolation;
```

Figure 16: Combinational Network Entity

As regards the signals, the first four in figure 17 need to implement the divisions by power of two, so they are the signals used by the Ripple Carry Adders to compute the resulting signals.

```
------------------------------------------------------------
-- Architecture
------------------------------------------------------------

architecture rtl of CombInterpolation is

    -- constant
    constant zero   : std_logic := '0';

    -- signals
    signal ni_d2_s      : std_logic_vector(NBit - 1 downto 0);
    signal ni_d4_s      : std_logic_vector(NBit - 1 downto 0);
    signal oi_d2_s      : std_logic_vector(NBit - 1 downto 0);
    signal oi_d4_s      : std_logic_vector(NBit - 1 downto 0);
    signal rca_1_out_s  : std_logic_vector(NBit - 1 downto 0);
    signal rca_4_out_s  : std_logic_vector(NBit - 1 downto 0);

    -- RCA for the sums
    component RippleCarryAdder
        generic (NBit : positive := 16);
        port (
            A    : in  std_logic_vector(NBit - 1 downto 0);
            B    : in  std_logic_vector(NBit - 1 downto 0);
            Cin  : in  std_logic;
            S    : out std_logic_vector(NBit - 1 downto 0);
            Cout : out std_logic
        );
    end component;
```

Figure 17: Combinational Network Components and Signals

The next part implements the calculations of the signals described above ($\frac{y_{n+1}}{4}$, $\frac{y_{n+1}}{2}$, $\frac{y_n}{4}$, $\frac{y_n}{2}$). In fact, a single bit shift to right is carried out to implement the division by two, ad of two bit to implement the division by four.

```
begin

    -----------------------
    -- signal assignment --
    -----------------------

    -- ni_d2_s = ni/2 (ni right shift of 1)
    ni_d2_s(NBit - 1) <= '0';
    ni_d2_s(NBit - 2 downto 0) <= ni(NBit - 1 downto 1);

    -- ni_d4_s = ni/4 (ni right shift of 2)
    ni_d4_s(NBit - 1 downto NBit - 2) <= (others => '0');
    ni_d4_s(NBit - 3 downto 0) <= ni(NBit - 1 downto 2);

    -- oi_d2_s = oi/2 (oi right shift of 1)
    oi_d2_s(NBit - 1) <= '0';
    oi_d2_s(NBit - 2 downto 0) <= oi(NBit - 1 downto 1);

    -- oi_d4_s = oi/4 (oi right shift of 2)
    oi_d4_s(NBit - 1 downto NBit - 2) <= (others => '0');
    oi_d4_s(NBit - 3 downto 0) <= oi(NBit - 1 downto 2);
```

Figure 18: Combinational Network Shift bit a bit

The second part of architecture, instead, implements the network described by the figure 4, that is a composition of five Ripple Carry Adders, which sums the four signals obtained by the shifts.

```
    ----------
    -- OUT0 --      out0 = oi
    ----------

    out0 <= oi;

    ----------
    -- OUT1 --      out1 = ni_d4_s + oi_d4_s + oi_d2_s
    ----------

    RCA_1: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            A       => ni_d4_s,
            B       => oi_d4_s,
            Cin     => zero,
            S       => rca_1_out_s
        );

    RCA_2: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            a       => oi_d2_s,
            b       => rca_1_out_s,
            Cin     => zero,
            s       => out1
        );

    ----------
    -- OUT2 --      out2 = ni_d2_s + oi_d2_s
    ----------

    RCA_3: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            a       => ni_d2_s,
            b       => oi_d2_s,
            Cin     => zero,
            s       => out2
        );

    ----------
    -- OUT3 --      out3 = ni_d4_s + ni_d2_s + oi_d4_s
    ----------

    RCA_4: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            a       => ni_d4_s,
            b       => ni_d2_s,
            Cin     => zero,
            s       => rca_4_out_s
        );

    RCA_5: RippleCarryAdder
        generic map (NBit => NBit)
        port map(
            a       => oi_d4_s,
            b       => rca_4_out_s,
            Cin     => zero,
            s       => out3
        );

end rtl;
```

Figure 19: Combinational Network Output

3.5 Linear Interpolator

As said before, the **Linear Interpolator** takes a signals in input and provides in output another signals.

```
------------------------------------------------------------
-- Entity
------------------------------------------------------------

entity LinearInterpolator is
    generic (NBit : positive := 16);
    port (
        clk         : in  std_logic;
        a_rst_n     : in  std_logic;
        signal_in   : in  std_logic_vector(NBit - 1 downto 0);
        signal_out  : out std_logic_vector(NBit - 1 downto 0)
    );
end LinearInterpolator;
```

Figure 20: Linear Interpolator Entity

The Linear Interpolator contains all the previous circuits, so Ripple Carry Adders, D Flip Flops and the Combinational Network.

```
------------------------------------------------------------
-- Architecture
------------------------------------------------------------

architecture rtl of LinearInterpolator is
    -- signals
    signal ctr_out_s    : std_logic_vector(1 downto 0);
    signal reg_ni_s     : std_logic_vector(NBit - 1 downto 0);
    signal reg_oi_s     : std_logic_vector(NBit - 1 downto 0);
    signal and_out_s    : std_logic;
    signal ci_out_0_s   : std_logic_vector(NBit - 1 downto 0);
    signal ci_out_1_s   : std_logic_vector(NBit - 1 downto 0);
    signal ci_out_2_s   : std_logic_vector(NBit - 1 downto 0);
    signal ci_out_3_s   : std_logic_vector(NBit - 1 downto 0);
    signal mux_to_reg_s : std_logic_vector(NBit - 1 downto 0);

    -- constant
    constant one    : std_logic := '1';

    component DFF_N
        generic (NBit : positive := 16);
        port (
            clk     : in  std_logic;
            a_rst_n : in  std_logic;
            en      : in  std_logic;
            D       : in  std_logic_vector(NBit - 1 downto 0);
            Q       : out std_logic_vector(NBit - 1 downto 0)
        );
        end component;

    component counter
        generic (NBit : positive := 2);
        port (
            clk     : in  std_logic;
            a_rst_n : in  std_logic;
            dout    : out std_logic_vector(NBit - 1 downto 0)
        );
        end component;

    component CombInterpolation
        generic (NBit : positive := 16);
        port (
            ni      : in  std_logic_vector(NBit - 1 downto 0);
            oi      : in  std_logic_vector(NBit - 1 downto 0);
            out0    : out std_logic_vector(NBit - 1 downto 0);
            out1    : out std_logic_vector(NBit - 1 downto 0);
            out2    : out std_logic_vector(NBit - 1 downto 0);
            out3    : out std_logic_vector(NBit - 1 downto 0)
        );
    end component;
```

Figure 21: Linear Interpolator Entity

REG_OI and *REG_NI* (that are two D Flip Flop), implements the Input Network. In addition to them there is the Counter, whose output go in input to an AND port, implemented by the signal and_out _s. This single bit signal is used as enabler by *REG_OI* and *REG_NI*.

The output signals of this two registers are the inputs for the Combinational Network that provides in output the four computed signals. These obtained signals go in input to a Multiplexer, whose output is controlled by the output of the Counter.

The combination between Multiplexer and the register that memorizes its output (*REG_OUT*) represent the Output Network.

```
begin

    and_out_s <= ctr_out_s(0) and ctr_out_s(1);

    -- new input
    REG_NI: DFF_N
        generic map (NBit => NBit)
        port map(
            clk     => clk,
            a_rst_n => a_rst_n,
            en      => and_out_s,
            D       => signal_in,
            Q       => reg_ni_s
        );

    -- old input
    REG_OI: DFF_N
        generic map (NBit => NBit)
        port map(
            clk     => clk,
            a_rst_n => a_rst_n,
            en      => and_out_s,
            D       => reg_ni_s,
            Q       => reg_oi_s
        );

    -- counter to control DFFs' activation
    COUNTER_1: counter
        generic map (NBit => 2)
        port map(
            clk     => clk,
            a_rst_n => a_rst_n,
            dout    => ctr_out_s
        );

    -- combinatorial interpolator for new input and old input
    CI: CombInterpolation
        generic map (NBit => Nbit)
        port map(
            ni      => reg_ni_s,
            oi      => reg_oi_s,
            out0    => ci_out_0_s,
            out1    => ci_out_1_s,
            out2    => ci_out_2_s,
            out3    => ci_out_3_s
        );

    -- multiplexer for the output
    MUX: process(ctr_out_s, ci_out_0_s, ci_out_1_s, ci_out_2_s, ci_out_3_s)
    begin
        case ctr_out_s is
            when "00" => mux_to_reg_s <= ci_out_0_s;
            when "01" => mux_to_reg_s <= ci_out_1_s;
            when "10" => mux_to_reg_s <= ci_out_2_s;
            when "11" => mux_to_reg_s <= ci_out_3_s;
            when others => mux_to_reg_s <= (others => '0');
        end case;
    end process;

    -- DFF for the output
    REG_OUT: DFF_N
    generic map (NBit => NBit)
    port map(
        clk     => clk,
        a_rst_n => a_rst_n,
        en      => one,
        D       => mux_to_reg_s,
        Q       => signal_out
    );

end rtl;
```

Figure 22: Linear Interpolator Entity

4 Test Plan

The following steps have been made to test the proper functioning of every part of the Linear Interpolator:

1. **Ripple Carry Adder**, **D Flip Flop** and **Counter** have been tested independently, using separate test bench.
2. The **Combinational Network** has been tested through comparison with specific values previously decided.
3. The same values are been used to test the total **Linear Interpolator**.
4. Two **scripts** have been made to test the total circuit with **signals dynamically generated**.

4.1 Basic Circuits Test

The **Ripple Carry Adder** has been tested by choosing specific values for the input, and checking that the values of outputs represent the right sum and carry out.

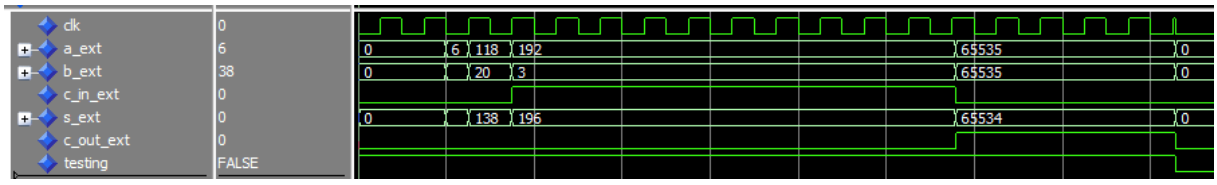

Figure 23: Ripple Carry Adder Test

For the **D Flip Flop** has been tested the ability to save and keep stable the values received in input. Moreover, has been tested the proper functioning of enabler port.

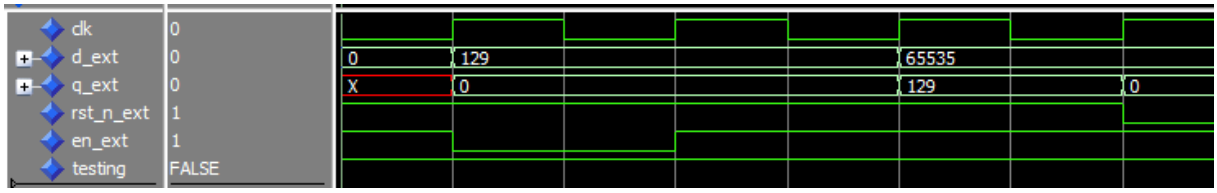

Figure 24: D Flip Flop Test

For the **counter**, the output was tested as the various clock cycles passed:

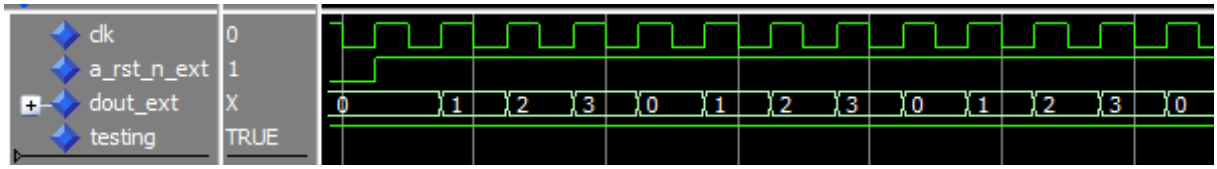

Figure 25: CounterTest

4.2 Combinational Network Test

In order to test the proper functioning of **Combinational Network**, an Excel file has been made, shown in the following figure:

i	Yi	X1	X2	X3	X4
0	73				
1	88	73,00	76,75	80,50	84,25
2	46	88,00	77,50	67,00	56,50
3	89	46,00	56,75	67,50	78,25
4	93	89,00	90,00	91,00	92,00
5	30	93,00	77,25	61,50	45,75
6	67	30,00	39,25	48,50	57,75
7	55	67,00	64,00	61,00	58,00
8	82	55,00	61,75	68,50	75,25
9	41	82,00	71,75	61,50	51,25
10	41	41,00	41,00	41,00	41,00
11	65535	41,00	16414,50	32788,00	49161,50
12	65535	65535,00	65535,00	65535,00	65535,00

L	4			
k	0	1	2	3
u	0	0,25	0,5	0,75

VERSION WITH SIMPLIFICATIONS AND TRUNCATIONS					
i	Yi	X1	X2	X3	X4
0	73				
1	88	73	76	80	84
2	46	88	77	67	56
3	89	46	56	67	77
4	93	89	89	90	91
5	30	93	76	61	45
6	67	30	38	48	56
7	55	67	62	60	56
8	82	55	60	68	74
9	41	82	71	61	50
10	41	41	40	40	40
11	65535	41	16413	32787	49160
12	65535	65535	65533	65534	65533

HEX
0049
0058
002E
0059
005D
001E
0043
0037
0052
0029
0029
FFFF
FFFF

Figure 26: Test Plan for Combinational Network

In this file a series of values have been dynamically generated, and they are used to compute the linear interpolation with and without simplification (shift for division, etc...).

After that, some of this values are used to create the stimuli for the Combinational Network, in a test bench specifically created. The correctness of the results has been verified through a visual analysis. The ModelSim simulation obtained is the following:

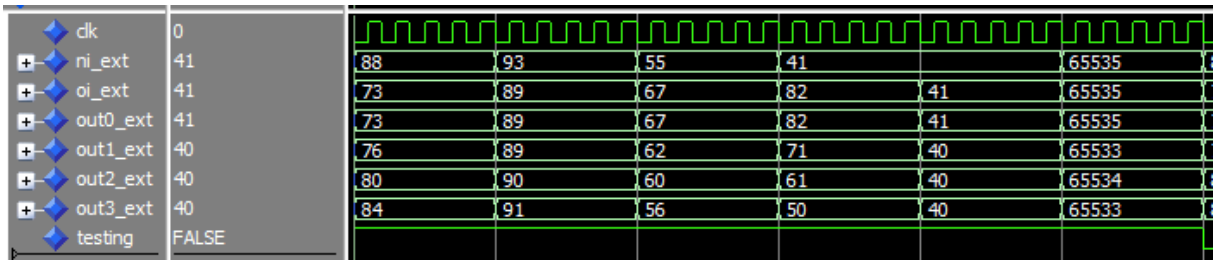

Figure 27: Combinational Network Test

Moreover, in the Excel file there are both approximated and effective interpolated signals, so it is possible to compute the mean error, as shown below:

Y	Y simpl	Residuals		Mean Error
73,00	73	0,00		0,77
88,00	88	0,00		
46,00	46	0,00		
89,00	89	0,00		
93,00	93	0,00		
30,00	30	0,00		
67,00	67	0,00		
55,00	55	0,00		
82,00	82	0,00		
41,00	41	0,00		
41,00	41	0,00		
65535,00	65535	0,00		
76,75	76	0,75		
77,50	77	0,50		
56,75	56	0,75		

Figure 28: Mean Error caused by the simplifications

It's clear that in this case the mean error is below one; This confirms the hypothesis that the error caused by the approximation is acceptable.

4.3 Linear Interpolator Test

As said before, the values obtained in the previous Excel file (Figure 26), has been used even in the **Linear Interpolator** test. The following figure represents the stimuli used in the test bench:

```
stimulus : process
    begin
    wait for 150 ns;
    a_rst_n_ext <= '1';
    wait for 250 ns;     -- 73
    sig_ext     <= x"0049";
    wait for 400 ns;     -- 88
    sig_ext     <= x"0058";
    wait for 400 ns;     -- 46
    sig_ext     <= x"002E";
    wait for 400 ns;     -- 89
    sig_ext     <= x"0059";
    wait for 400 ns;     -- 93
    sig_ext     <= x"005D";
    wait for 400 ns;     -- 30
    sig_ext     <= x"001E";
    wait for 400 ns;     -- 67
    sig_ext     <= x"0043";
    wait for 400 ns;     -- 55
    sig_ext     <= x"0037";
    wait for 400 ns;     -- 82
    sig_ext     <= x"0052";
    wait for 400 ns;     -- 41
    sig_ext     <= x"0029";
    wait for 400 ns;     -- 41
    sig_ext     <= x"0029";
    wait for 400 ns;     -- 65535
    sig_ext     <= x"FFFF";
    wait for 400 ns;     -- 65535
    sig_ext     <= x"FFFF";
    wait for 500 ns;
    testing  <= false;
end process;
```

Figure 29: Stimuli for Linear Interpolator

Even in this case, the correctness of the results have been tested by a visual analysis, through the comparison with the Excel results.

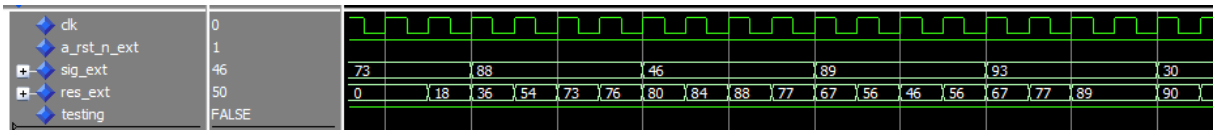

Figure 30: Linear Interpolator First Test

In this test is possible to see that when a new signals arrives, the first of the four output signals is provided after two clock cycles. This delay is caused by the input registers and the output register that are between the input and the output of the network.

4.4 Linear Interpolator Dynamic Test

As last step, a python script named *TB_Generation.py* has been created, that produces a test bench with a number of **dynamically generated stimuli**. This script takes in input as optional parameters the number of stimuli that have to generate, and the seed used for the generation (in order to have repeatable experiments).

```
PS C:\Users\matte\Desktop\Progetto\script> python .\TB_Generation.py --seed 0 --number 16
- Seed for generation: 0
- Will be generated 16 signals

[+] Testbench correctly generated
[+] Results files correctly generated
PS C:\Users\matte\Desktop\Progetto\script>
```

Figure 31: Script for Test bench Generation

The script generates three different files:

- *LinearInterpolator_tb.vhd*, is generated the test bench.
- *ExpectedResults.txt*, contains the list of the results that the circuit have to produce. It is used by a second script described below.
- *TabulatedResults.txt*, contains the same results of previous file, but in a tabulated form, useful for an eventual visual analysis.

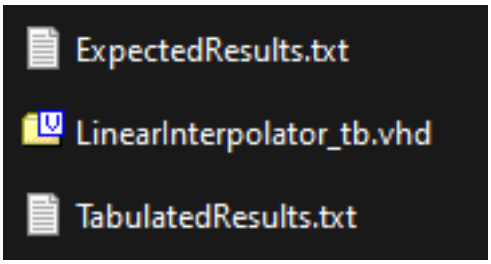

Figure 32: Generated Files

Below there is an example of stimuli dinamically generated:

```
stimulus : process
    begin
    wait for 150 ns;
    a_rst_n_ext <= '1';
    wait for 250 ns;
    sig_ext     <= x"8843";
    wait for 400 ns;
    sig_ext     <= x"af4c";
    wait for 400 ns;
    sig_ext     <= x"e100";
    wait for 400 ns;
    sig_ext     <= x"67a7";
    wait for 400 ns;
    sig_ext     <= x"1127";
    wait for 400 ns;
    sig_ext     <= x"bc11";
    wait for 400 ns;
    sig_ext     <= x"f716";
    wait for 400 ns;
    sig_ext     <= x"629f";
    wait for 400 ns;
    sig_ext     <= x"e865";
    wait for 400 ns;
    sig_ext     <= x"641a";
    wait for 400 ns;
    sig_ext     <= x"4567";
    wait for 400 ns;
    sig_ext     <= x"513a";
    wait for 400 ns;
    sig_ext     <= x"b81c";
    wait for 400 ns;
    sig_ext     <= x"fdaf";
    wait for 400 ns;
    sig_ext     <= x"d482";
    wait for 400 ns;
    sig_ext     <= x"b172";
    wait for 400 ns;
    wait for 500 ns;
    testing  <= false;
end process;
```

Figure 33: Generated Stimuli

Instead, below there is an example of *TabulatedResults.txt*:

SIG	OUT0	OUT1	OUT2	OUT3
34883	0	8720	17441	26161
44876	34883	37380	39879	42377
57600	44876	48057	51238	54419
26535	57600	49833	42067	34300
4391	26535	20997	15462	9925
48145	4391	15328	26267	37205
63254	48145	51921	55699	59476
25247	63254	53751	44250	34747
59493	25247	33807	42369	50930
25626	59493	51025	42559	34092
17767	25626	23660	21696	19730
20794	17767	18522	19280	20036
47132	20794	27378	33963	40547
64943	47132	51584	56037	60489
54402	64943	62306	59672	57036

Figure 34: Expected Results in tabular form

The purpose of this system is to be able to test the circuit automatically and with a very large number of input signals.

For this purpose it is necessary to export from ModelSim the results obtained as output signals in a list.lst file, and run a second script (*Results_validator.py*) that compares the file exported from ModelSim with the results computed by the first script.

```
PS C:\Users\matte\Desktop\Progetto\script> python .\Results_Validator.py
[+] OK
[+] OK
[+] OK
[+] OK
[+] OK
[+] OK
[+] OK
...
[+] OK
[+] OK
[+] OK
[+] OK
[+] OK

0 errors found
PS C:\Users\matte\Desktop\Progetto\script>
```

Figure 35: Script for Results Validation

If the script finds errors, these will be notified as shown in the figure below:

```
[+] OK
[+] OK
[+] OK
[-] 57036 57035

1 errors found
PS C:\Users\matte\Desktop\Progetto\script>
```

Figure 36: Example of wrong result

5 Vivado Report

In this last section an **automatic synthesis** will be done, through the tool **Xilinx Vivado**. The target working device is the FPGA *xc7z010clg400-1* and the three steps that will be analyzed are the following:

1. **Elaborated Design** Analysis.
2. **Synthesis** and Report analysis.
3. **Implementation** and Report analysis.

5.1 Elaborated Design Analysis

In this first part will be analyzed the **Elaborated Design**, that is the Vivado representation of the Linear Interpolator at the **Register Transfer Level**. It should be equal to the structure defined in the Architecture Description Section.

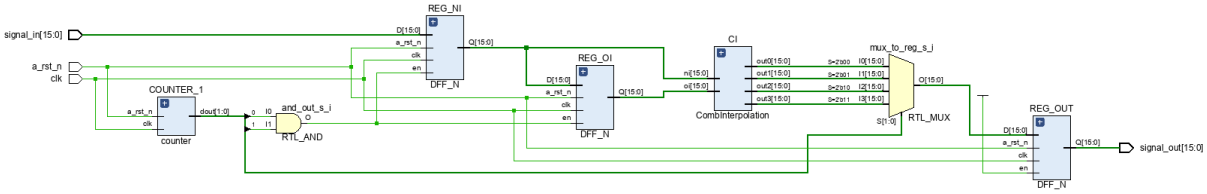

Figure 37: Elaborated Design

From the figure above is clear that the circuit respects the structure chosen in the planning stage, so it's possible to start the synthesis.

5.2 Synthesis Analysis

In this phase the Elaborated Design previously generated will be **translated** in circuits that the FPGA **can implement**. This will be done respecting all given **constraints**. In this case the only constraint is the clock, that must have a period of $8ns$.

The synthesis result is the following:

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Constraints:	constrs
Incremental synthesis:	None

Figure 38: Synthesis Result

There are no errors or warnings, so it's possible to study the reports.

5.2.1 Timing Report and Critical Path

The **timings** obtained are as follows:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,986 ns	Worst Hold Slack (WHS):	0,273 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:	66	Total Number of Endpoints:	66	Total Number of Endpoints:	50

All user specified timing constraints are met.

Figure 39: Timing Report

First of all, there are **no negative slacks**, so the clock with a period of $8ns$ is clearly enough for the produced design. Moreover, the **worst negative slack** is $0.986ns$, so the clock can be at least faster of this value.

Analyzing the **synthesized design**, the **critical path** is the following:

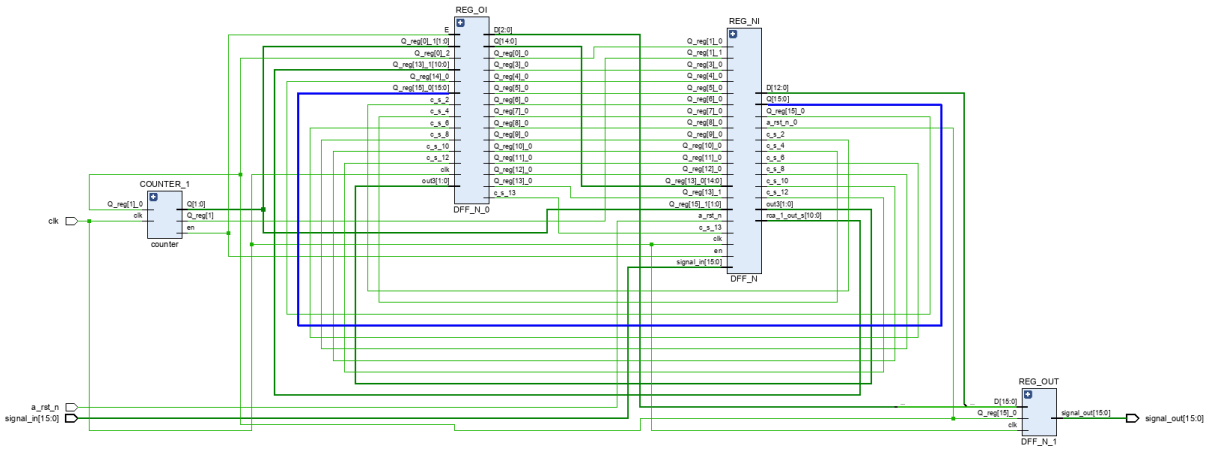

Figure 40: Critical Path for Setup time

5.2.2 Utilization Analysis

The **utilization report** obtained are as follows:

Utilization — **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	104	17600	0.59
FF	50	35200	0.14

Figure 41: Utilization Report

The utilization for **Look Up Tables** and **Flip Flops** are lower then the 1%, so the FPGA has all resources needed to implement this circuit.

5.2.3 Power Consumption Analysis

The last report is about the **power consumption**. This is a very rough estimation, but allows to have a general idea of the circuit's power requirements.

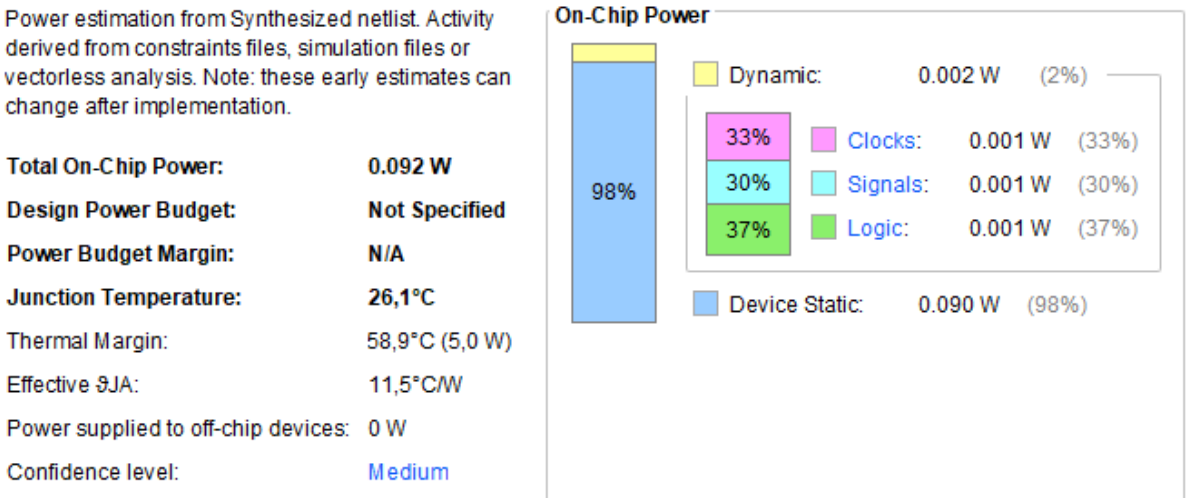

Figure 42: Power Report

The **general consumption** is lower than $100mW$. Moreover, it's clear that the percentage of static power is much greater than the dynamic power. So it's possible to say that the **switching activity** in this circuit is relatively low.

5.3 Implementation analysis

In this phase the **place and route** will be performed by Vivado, with some useful **optimizations**. In general this phase is preceded by the **I/O Planning**, in which the I/O Physical Ports of the FPGA are associated with the I/O Ports of the Elaborated Design. But the FPGA has not enough ports to implement an input and an output of 16 bit. So the implementation will be performed in *Out of Context Mode*, that allows to do implementation without I/O Planning.

The implementation result is the following:

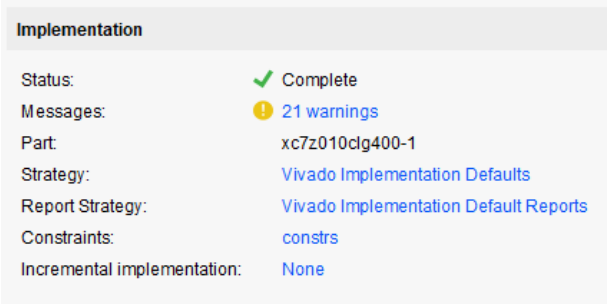

Figure 43: Implementation Result

There are no errors, but there are warnings. They will be analyzed later.

5.3.1 Timing Report and Critical Path

The **timings** obtained are as follows:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,567 ns	Worst Hold Slack (WHS):	0,130 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:	66	Total Number of Endpoints:	66	Total Number of Endpoints:	50

All user specified timing constraints are met.

Figure 44: Timing Report

Even in this case there are **no negative slacks**, so a clock period of $8ns$ is fast enough. But in this case the **worst negative slack is worse** than before. This is due to the fact that the implementation allows to to obtain a more accurate estimate.

The **critical path** that cause this slack is the following:

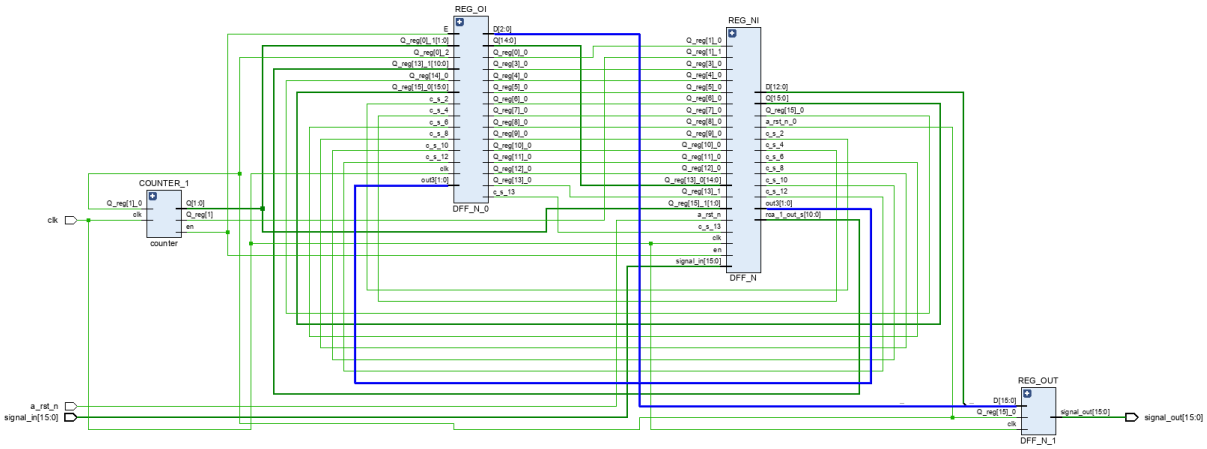

Figure 45: Critical Path for Setup time

5.3.2 Utilization Analysis

The **utilization report** obtained are as follows:

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	110	17600	0.63
FF	50	35200	0.14

Figure 46: Utilization Report

Compared to the synthesis, the number of **Look Up Table** has increased to 110, while the **Flip Flops** are the same.

5.3.3 Power Consumption Analysis

As regards power consumption, the results are the following:

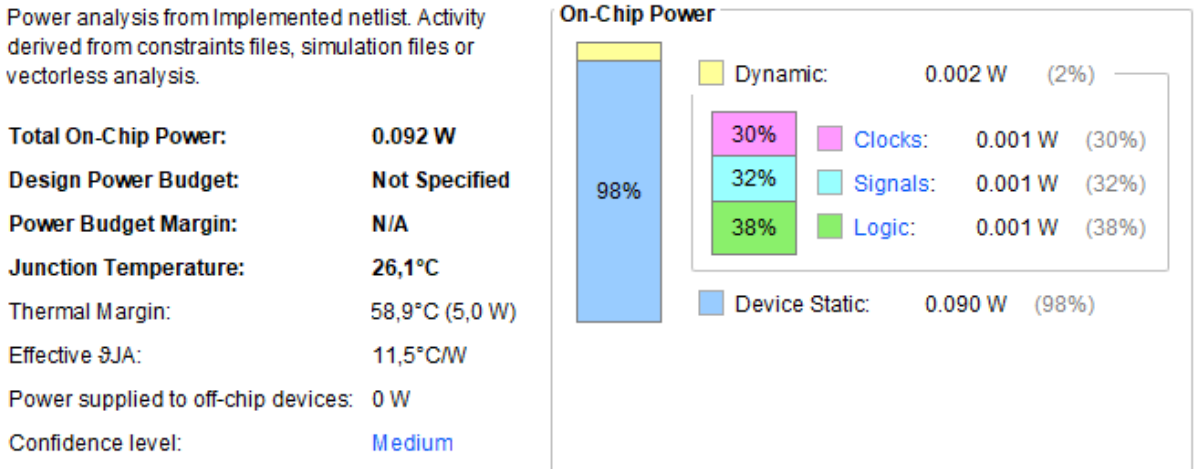

Figure 47: Power Report

Compared to the synthesis, there are no changes.

5.3.4 Warnings Analysis

The Implementation's Warnings are the following:

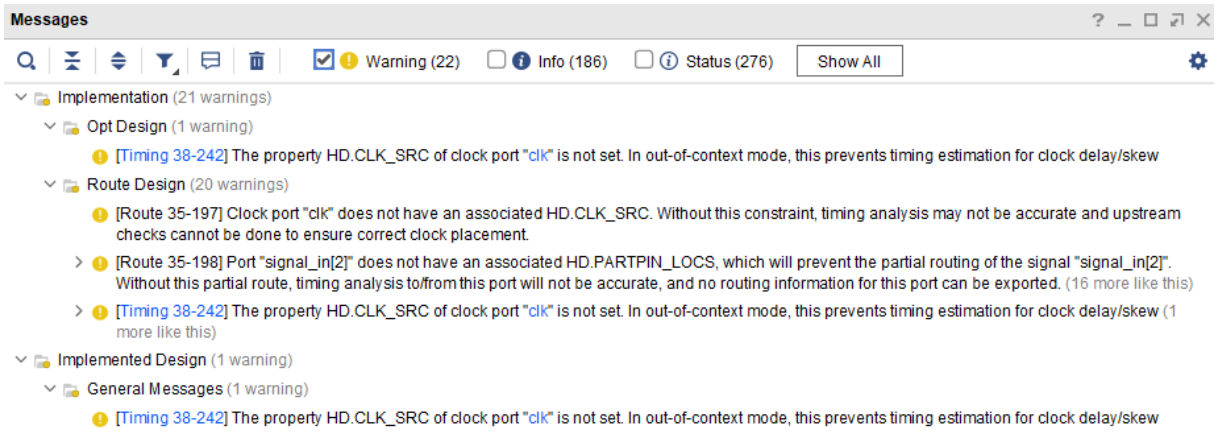

Figure 48: Implementation Warnings

All of them are Vivado internal warnings, that can be ignored.

6 Conclusions

The **Linear Interpolator** is a circuit that, received two consecutive signals, provides in output the L (factor of interpolation) signals that represent the linear interpolation between them.

Considering a circuit with a fixed L, there are many possible implementations, each of them allowing to get a different **trade-off** between **performance**, **complexity** and **precision**.

- If an **high level of precision** is needed, the interpolation formula should not have any oversimplifications, so is necessary to implement multiplication and division modules (high complexity).
- If an **high speed** or **high power efficiency** is needed, a multiple clock domain should be implemented, obtaining an higher level of complexity.
- If a **low complexity** is needed, the proposed implementation should be chosen. In fact, it allows to implements the entire computation using unsigned signals and simple Ripple Carry Adders.