UNIVERSITÀ DI PISA

Computer Engineering

Electronics and communication systems

Perceptron Project Report

Team members:

Francesco Martoccia

Salvatore Lombardi

ACADEMIC YEAR 2021/2022

Contents

1 - Introduction

1.1 Algorithm Description

The perceptron is an algorithm for learning a binary classifier called a threshold function: a function that maps its input x (a real-valued vector) to an output value $f(x)$ (a single binary value):

$$f(x) = \begin{cases} 1 \ \textit{if} \ w \cdot x + b > 0, \\ 0 \ \textit{otherwise} \end{cases} \tag{1.1}$$

In the formula, w is a vector of real valued weights, $w \cdot x$ is the dot product $\sum_{i=1}^{n} w_i x_i$, n is the number of inputs to the perceptron and b is the *bias*, that shifts the decision boundary away from the origin and does not depend on any input value.

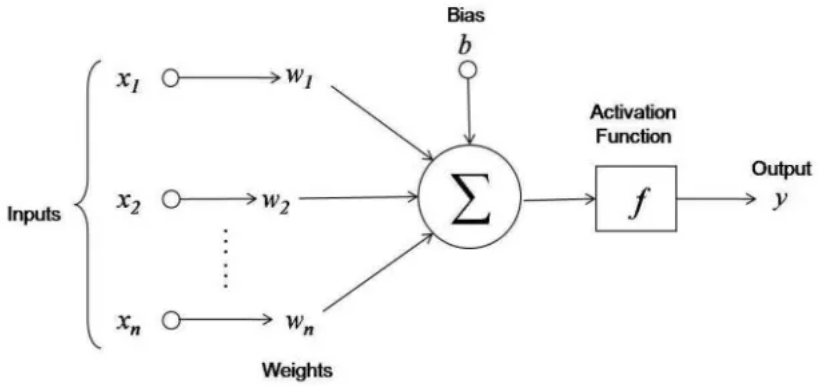

Figure 1.1: Perceptron structure

The architecture to implement consists of a Perceptron which takes 10 vectors x_n represented using 8 bits, 10 vectors w_n represented with 9 bits and the *bias* also with 9 bits. All the inputs have to be considered in the range [-1, 1] with the standard 2's complement notation. The activation function to implement is a sigmoid in this case. The output of the Perceptron is represented on 16 bits and its value is in the range [0, 1].

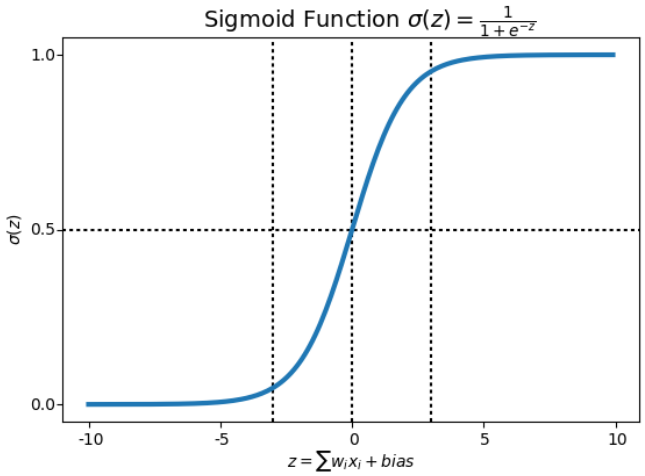

Figure 1.2: Sigmoid function

1.2 Possible Architectures

To realize an implementation of the algorithm described previously, there are 3 main components needed:

- A **multiplier** to do the products of the x vectors with the corresponding w vectors. This can be implemented in different ways, we chose to exploit a **parallel multiplier**.
  Another option was to realize the multiplication by using a ROM-based solution, but in our case the number of bits of the operands was too high to adopt this technique. In particular, the number of words that we would need in this case is 2^{17} (since 17 is the sum of the size of the operands in bits), but this kind of approach is best suited for cases in which the number of bits is very low, to have less memory usage and latency.

- An **adder** to sum the results of the previously calculated products and the *bias*. In this case different techniques could be used to implement this circuit, we opted for a parallel adder, in particular a **Ripple Carry Adder**.
  Other possible options were the Serial Adder, that requires a number of clock cycles equal to the number of bits of the result to compute the sum, the Carry-Look-Ahead Adder and the Ripple Carry Adder with pipeline. The last 2 are faster solutions but require extra complexity in the architecture with respect to the Ripple Carry Adder.

- A circuit that implements the **Activation Function** of the Perceptron. We decided to use a **Look-up table**, as suggested in the project assignment, and we exploited the symmetry of the sigmoid function to optimize the size of the LUT.
  Another approach was to use dedicated architectures, but this kind of solution is much more complex to realize.

2 - Architecture description

2.1 General Architecture

As introduced before, to implement the algorithm described, the main components of the architecture we realized are a **multiplier circuit**, an **adder circuit** and a **Look-Up Table** for the sigmoid activation function. For the adder circuit we have chosen to adopt an adder-tree configuration, that will be described in detail later. In designing the architecture we followed the rule to have always paths register-logic-register, so that during the synthesis phase Vivado could evaluate all paths and make accurate estimates about the timing and the delays of each of them.

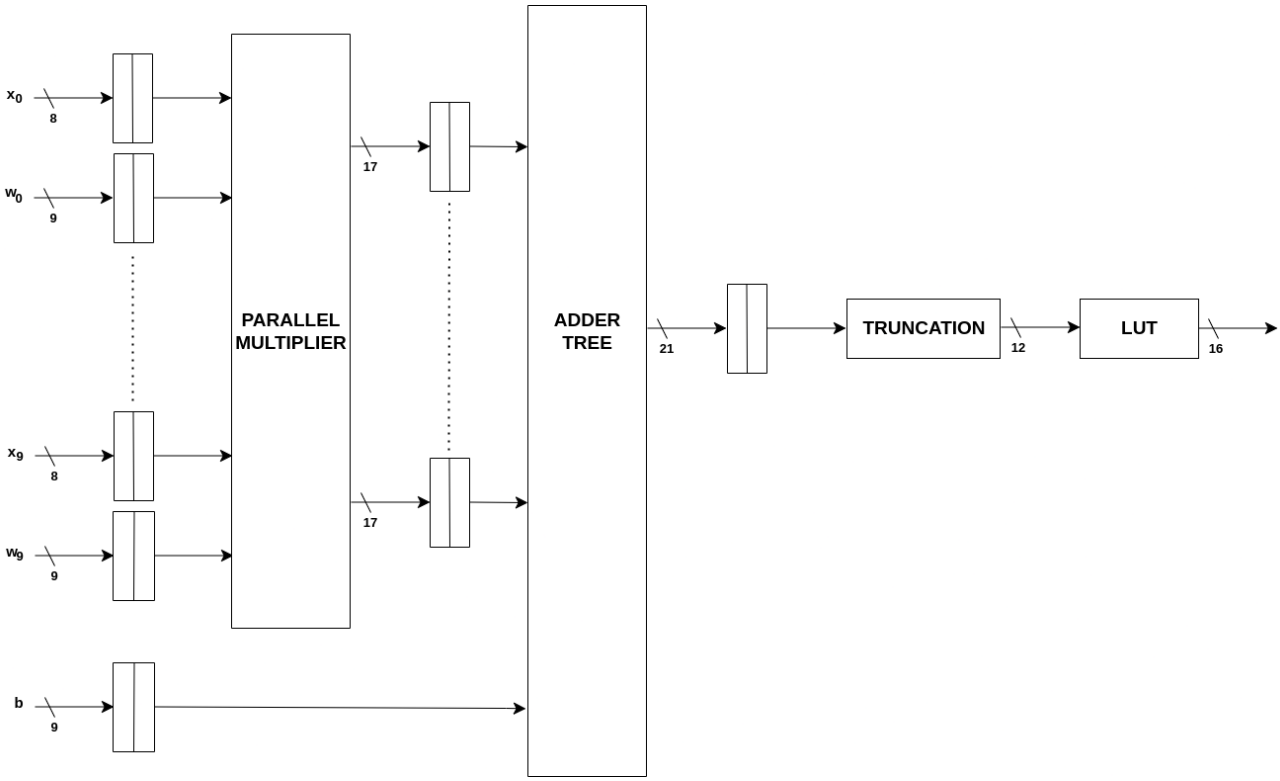

Figure 2.1: Perceptron architecture

2.2 Multiplier Circuit

To compute the 10 products between the x and w vectors (with 8 and 9 bits respectively) we implemented a **Parallel Multiplier**. This block takes all the 20 input vectors (except the *bias*) and produces in output the 10 results of the multiplication. In our case the products to compute are between real numbers, represented with the standard 2's complement representation, so we needed additional logic to handle the sign. For this reason, we created a block called **Signed Parallel Multiplier** that handles the sign and allows to perform the multiplication without sign through the **Unsigned Parallel Multiplier** block.

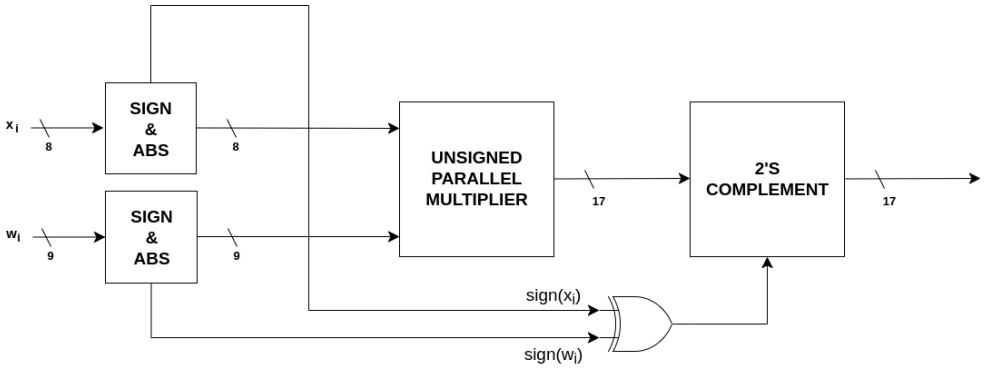

Figure 2.2: Signed Parallel Multiplier

As described by the figure, the **Signed Parallel Multiplier** takes the 2 signs of the original vectors to multiply and performs the module of the 2 signals, giving them in input to the **Unsigned Parallel Multiplier** inside the block. At the end of the computation, if the XOR of the signs was 1, the output is complemented and summed with 1, according to the 2's complement notation. The **Unsigned Parallel Multiplier** as the name suggests, allows to compute an unsigned multiplication. The architecture is shown in the following figure:

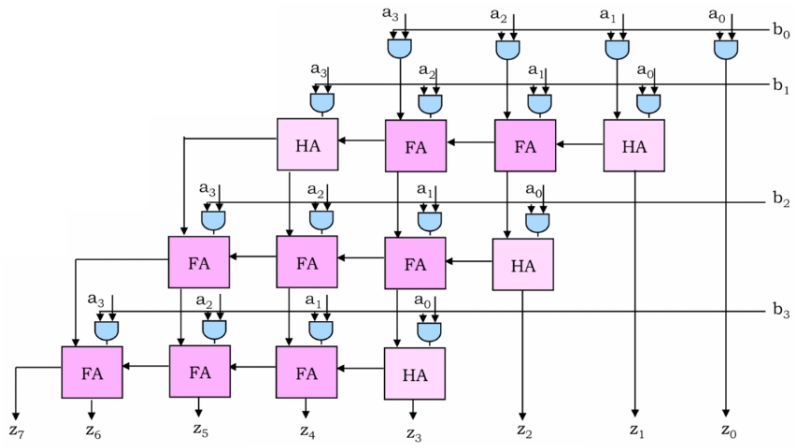

Figure 2.3: Unsigned Parallel Multiplier

As we can see in figure 2.3, we used a combination of **AND ports**, **Half-Adders** and **Full-Adders**, trying to exploit the simpler components whenever possible, to avoid the addition of extra complexity to the architecture without any benefit.

2.3 Adder Circuit

To execute the sums between the 10 products and the bias we need an Adder circuit. We opted for a parallel architecture in this case, in particular an **Adder Tree** configuration.

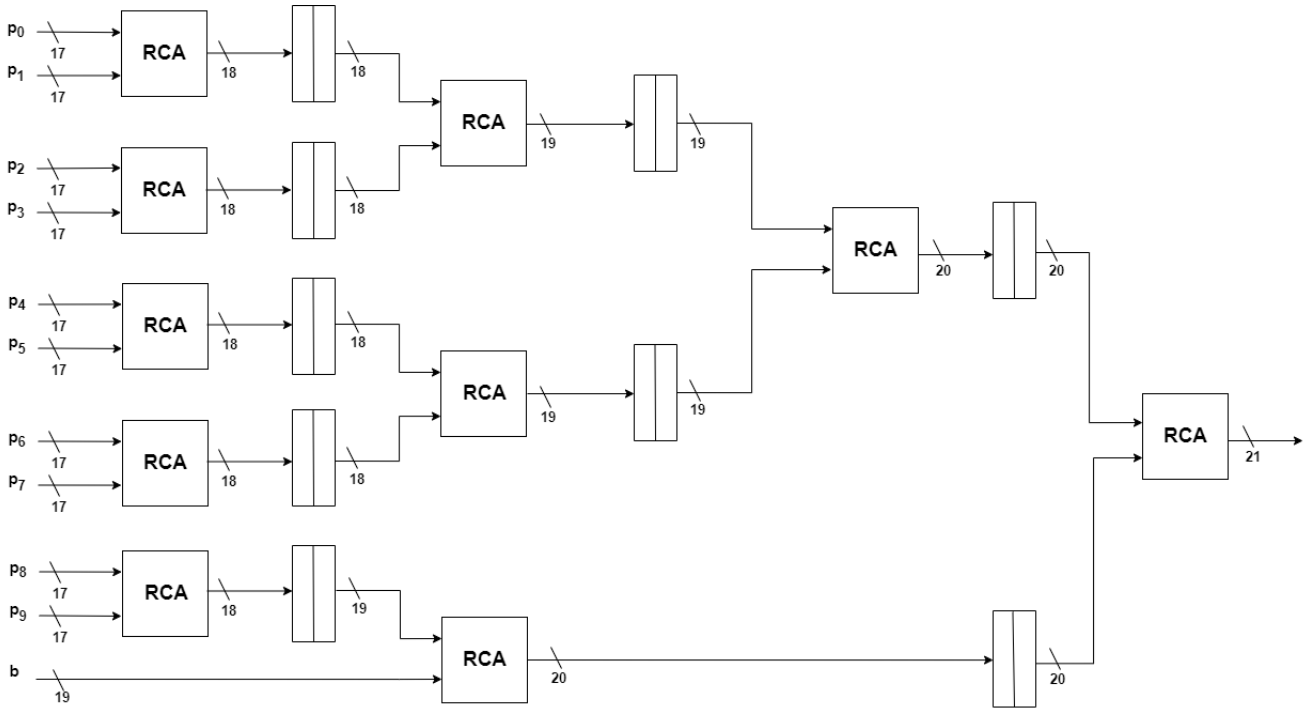

Figure 2.4: Adder Tree

Through the **Adder Tree** shown in figure 2.4 we computed the sums exploiting a **Ripple Carry Adder** for each sum we had to perform. The addition is realized in a faster way with respect to a standard serial configuration, because we don't have to wait for the carry, but the sums can be executed in parallel, reducing the number of clock cycles needed to compute the result. We inserted a register in each path between 2 adders, to reduce the impact of the critical path.

2.4 Look-Up Table

The output of the Adder Circuit is the overall sum, that is represented on 21 bits. To realize a **Look-Up Table** with inputs of that size, we would need 2^{21} entries (2097152), with each one mapping a 16 bits value in output. A LUT of this size would be too large, and the additional space required could lead to more latency to

get the value corresponding to each entry. For this reason, we decided to perform a **truncation** from 21 to 13 bits. The LUT obtained in this way has a size of 2^{13} entries, but since it was possible to exploit the symmetry of the sigmoid function, we stored only half of the LUT, so 4096 entries were sufficient.

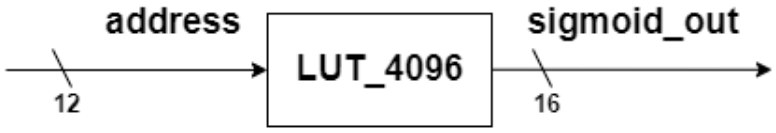

Figure 2.5: Look-Up Table

3 - VHDL code

The VHDL code will be illustrated with a Top-Down approach, starting with the higher level block, called **Perceptron**, that contains the whole architecture and then analyzing one by one its most important components.

3.1 Perceptron

The **Perceptron** module contains all the components needed to implement the architecture, takes the inputs and returns the final output of the circuit.

```
entity Perceptron is
    port(
        -- Input values for x
        x_0: in std_logic_vector(7 downto 0);
        ...
        x_9: in std_logic_vector(7 downto 0);

        -- Input values of w
        w_0: in std_logic_vector(8 downto 0);
        ...
        w_9: in std_logic_vector(8 downto 0);

        clk: in std_logic;
        rst: in std_logic;
        b: in std_logic_vector(8 downto 0); -- Bias
        -- Output of the Perceptron
        s_out: out std_logic_vector(15 downto 0)
    );
end Perceptron;
```

The entire description of the Perceptron module and the definition of its components it's not completely shown here, because we would need too much lines of code. The components used in the Perceptron architecture are: **Parallel_Multiplier**, **Adder_Tree**, **LUT_4096** and **DFF_N**. The latter has been used to add the registers at the beginning, that will contain the inputs, and the registers in each

intermediate stage between a logic block and another. The other 3 components will be explained in detail later.

```
lut_process: process(s_r3_out, s_lut_out)
    begin
        if(s_r3_out(20) = '0') then
            s_lut_in <= s_r3_out;
        else
        -- The 2's complement is needed if the input
        -- of the LUT is negative
            s_lut_in <= std_logic_vector(unsigned(not(s_r3_out)) + 1);
        end if;

        if (s_r3_out(20) = '0') then
            s_out <= std_logic_vector(unsigned(s_lut_out));
        else
        -- If the input of the LUT is negative, we get the output
        -- of the Perceptron from the output
        -- of the LUT exploiting symmetry
            s_out <= std_logic_vector(32767 - unsigned(s_lut_out));
        end if;
    end process;
```

To represent the real numbers in the architecture, we adopted the 2's complement representation as said before, and represented all values with the std_logic_vector type. To handle the sign in the case of the LUT, the above process was needed. In case the input of the LUT is negative, it takes the 2's complement of the number, to always have a positive number as address, since we only store half of the entries. We have to also compute the correct result for a negative value, exploiting symmetry.

3.2 Parallel Multiplier

This module takes all the x and w vectors in input and produces in output the 10 products that will be later used in the Adder Tree for the sum. To realize the products, 10 components of type **Signed_Parallel_Multiplier** were needed.

```
entity Parallel_Multiplier is
    port(
        -- Values of the vectors x to multiply
        x_p_0: in std_logic_vector(7 downto 0);
        ...
        x_p_9: in std_logic_vector(7 downto 0);

        -- Values of the vectors w to multiply
```

```
        w_p_0: in std_logic_vector(8 downto 0);
        ...
        w_p_9: in std_logic_vector(8 downto 0);

        -- Values of the products
        p_0 : out std_logic_vector(16 downto 0);
        ...
        p_9 : out std_logic_vector(16 downto 0)
    );
end entity Parallel_Multiplier;

architecture beh of Parallel_Multiplier is

begin
    SPM_0: Signed_Parallel_Multiplier
        generic map(
            Nbit_x => 8,
            Nbit_w => 9
        )
        port map(
            x_p_sign => x_p_0,
            w_p_sign => w_p_0,
            p_sign   => p_0
        );
...

    SPM_9: Signed_Parallel_Multiplier
        generic map(
            Nbit_x => 8,
            Nbit_w => 9
        )
        port map(
            x_p_sign => x_p_9,
            w_p_sign => w_p_9,
            p_sign   => p_9
        );

end architecture beh;
```

3.3 Signed Parallel Multiplier

This component implements the signed multiplication, taking the input values from the **Parallel_Multiplier** module and converting them in order to execute the

product with the **Unsigned_Parallel_Multiplier** block.

```
entity Signed_Parallel_Multiplier is
    generic (
        Nbit_x: positive; -- Number of bits of the vector x_p
        Nbit_w: positive  -- Number of bits of the vector w_p
    );
    port(
        x_p_sign: in std_logic_vector(Nbit_x - 1 downto 0);
        w_p_sign: in std_logic_vector(Nbit_w - 1 downto 0);
        p_sign  : out std_logic_vector(Nbit_x + Nbit_w - 1 downto 0)
    );
end entity Signed_Parallel_Multiplier;

architecture beh of Signed_Parallel_Multiplier is
...
begin
    -- Value of the MSB used for the sign representation
    -- of the x_p_sign vector
    x_sign <= x_p_sign(Nbit_x - 1);
    -- Value of the MSB used for the sign representation
    -- of the w_p_sign vector
    w_sign <= w_p_sign(Nbit_w - 1);
    -- Absolute value of the signal to pass to the
    -- Unsigned_Parallel_Multiplier
    x_p_unsign <= std_logic_vector(abs(signed(x_p_sign)));
    -- Absolute value of the signal to pass to the
    -- Unsigned_Parallel_Multiplier
    w_p_unsign <= std_logic_vector(abs(signed(w_p_sign)));
    p_sign <= std_logic_vector(unsigned(not(p_unsign)) + 1) when
    ((x_sign xor w_sign) = '1') else p_unsign;

    UPM: Unsigned_Parallel_Multiplier
        generic map(
            Nbit_x => Nbit_x,
            Nbit_w => Nbit_w
        )
        port map(
            x_p => x_p_unsign,
            w_p => w_p_unsign,
            p   => p_unsign
        );
end architecture beh;
```

As we can see in the code, this module initially stores the signs of the 2 input vectors x and w, then takes the absolute value of the numbers and stores the result in a std_logic_vector. If the XOR operation between the signs bits is 1, the 2's complement of the result of the multiplication produced by the **Unsigned_Parallel_Multiplier** is performed. If the XOR is 0 the result remains unchanged.

3.4 Unsigned Parallel Multiplier

This module performs the unsigned multiplication, taking the inputs from the **Signed_Parallel_Multiplier** block. In the VHDL code above we defined the architecture of the component, following the structure described earlier in figure 2.3. The components needed in this case were the **FULL_ADDER** and the **HALF_ADDER**, both defined in the architecture. To perform the partial products we defined a process that initializes a vector with all the intermediate results. To do all the sums needed between these partial products, we generated all the required adders, dividing them in 3 main cases: FIRST_ROW, INTERNAL_ROW and LAST_ROW since they have different characteristics in terms of port mapping (that is not shown here for simplicity).

```
entity Unsigned_Parallel_Multiplier is
    generic (
        Nbit_x: positive; -- Number of bits of the vector x_p
        Nbit_w: positive   -- Number of bits of the vector w_p
    );
    port(
        x_p: in std_logic_vector(Nbit_x - 1 downto 0);
        w_p: in std_logic_vector(Nbit_w - 1 downto 0);
         -- Product vector
        p  : out std_logic_vector(Nbit_x + Nbit_w - 1 downto 0)
    );
end entity Unsigned_Parallel_Multiplier;

architecture beh of Unsigned_Parallel_Multiplier is
    component FULL_ADDER is
    port
    (
        a    : IN std_logic ;
        b    : IN std_logic ;
        cin  : IN std_logic ;
        s    : OUT std_logic ;
        cout : OUT std_logic
    );
```

```
    end component;

    component HALF_ADDER is
    port
    (
        a    : IN std_logic ;
        b    : IN std_logic ;
        s    : OUT std_logic ;
        cout : OUT std_logic
    );
    end component;

begin
    -- Process that computes the values of the partial products
    -- and initializes the values of the product vector
    p_process : process(x_p,w_p)
    begin
        for i in 0 to Nbit_w-1 loop
            for j in 0 to Nbit_x-1 loop
                product((i*Nbit_x) + j) <= x_p(j) and w_p(i);
            end loop;
        end loop;
    end process p_process;

    p(0) <= product(0); -- Setting of the first bit of the result

    -- Generation of the Unsigned Parallel Multiplier's components
    GEN_row: for i in 1 to Nbit_w-1 generate
        GEN_column: for j in 0 to Nbit_x-1 generate
        FIRST_ROW:      if i = 1 generate
            RIGHT_HA:       if j = 0 generate
                                FIRST_ROW: HALF_ADDER
                                port map
                                (
                                    ...
                                );
            end generate RIGHT_HA;
            CENTRAL_FA:     if j > 0 and j < Nbit_x-1 generate
                                FIRST_ROW: FULL_ADDER
                                port map
                                (
                                    ...
                                );
```

```
    end generate CENTRAL_FA;
    LEFT_HA:        if j = Nbit_x-1 generate
                        FIRST_ROW: HALF_ADDER
                        port map
                        (
                            ...
                        );
    end generate LEFT_HA;
end generate FIRST_ROW;

INTERNAL_ROW:    if i > 1 and i < Nbit_w-1 generate
    RIGHT_HA:        if j = 0 generate
                        INTERNAL_ROW: HALF_ADDER
                        port map
                        (
                            ...
                        );
    end generate RIGHT_HA;
    CENTRAL_FA:     if j > 0 and j < Nbit_x-1 generate
                        INTERNAL_ROW: FULL_ADDER
                        port map
                        (
                            ...
                        );
    end generate CENTRAL_FA;
    LEFT_FA:        if j = Nbit_x-1 generate
                        INTERNAL_ROW: FULL_ADDER
                        port map
                        (
                            ...
                        );
    end generate LEFT_FA;
end generate INTERNAL_ROW;

LAST_ROW:        if i = Nbit_w-1 generate
    RIGHT_HA:       if j = 0 generate
                    LAST_ROW: HALF_ADDER
                    port map
                    (
                        ...
                    );
    end generate RIGHT_HA;
    CENTRAL_FA:     if j > 0 and j < Nbit_x-1 generate
```

```
                        LAST_ROW: FULL_ADDER
                        port map
                        (
                            ...
                            );
            end generate CENTRAL_FA;
            LEFT_FA:        if j = Nbit_x-1 generate
                        LAST_ROW: FULL_ADDER
                        port map
                        (
                            ...
                        );
            end generate LEFT_FA;
        end generate LAST_ROW;
        end generate GEN_column;
    end generate GEN_row;
end architecture beh;
```

3.5 Adder Tree

This component performs all the sums of the products computed by the **Parallel_Multiplier** plus the *bias*, following the configuration described in figure 2.4. The components exploited in this architecture are the **DFF_N** and the **Ripple_Carry_Adder**, the latter executes the addition while the former is placed between each layer of adders in the tree. As we can see in the code, for each adder there is a register that takes the output value and passes it to the adder in the following layer. Since we had a *bias* represented on 9 bits, but the full adder in that level takes 2 inputs on 19 bits, it was necessary to do a left shift in order to align the point in the new representation and a sign extension.

```
entity Adder_Tree is
    port(
        -- Values of the products that have to be summed
        l1_s1_in1 : in std_logic_vector(16 downto 0);
        ...
        l1_s5_in2 : in std_logic_vector(16 downto 0);

        clk: in std_logic;
        rst: in std_logic;
        b: in std_logic_vector(8 downto 0); -- Bias

        sum: out std_logic_vector(20 downto 0) -- Sum vector
```

```
    );
end entity Adder_Tree;

architecture beh of Adder_Tree is

    component DFF_N
        generic( N : integer);
        port(
            clk     : in std_logic;
            a_rst_n : in std_logic;
            d       : in std_logic_vector(N - 1 downto 0);
            q       : out std_logic_vector(N - 1 downto 0)
        );
    end component DFF_N;

    component Ripple_carry_adder
        generic (Nbit : positive);
        port (
          a    : in std_logic_vector(Nbit - 2 downto 0);
          b    : in std_logic_vector(Nbit - 2 downto 0);
          cin  : in std_logic;
          s    : out std_logic_vector (Nbit - 1 downto 0);
          cout : out std_logic
        );
    end component Ripple_carry_adder;

begin
    -- First layer of adders
    l1_s1: Ripple_carry_adder
        generic map(
            Nbit => 18
        )
        port map(
            ...
        );
    l1_r1: DFF_N
        generic map(
            N => 18
        )
        port map(
            ...
        );
        ...
```

```
-- Second layer of adders
l2_s1: Ripple_carry_adder
    generic map(
        Nbit => 19
    )
    port map(
        ...
    );
l2_r1: DFF_N
    generic map(
        N => 19
    )
    port map(
        ...
    );
    ...

-- Third layer of adders
l3_s1: Ripple_carry_adder
    generic map(
        Nbit => 20
    )
    port map(
        ...
    );
l3_r1: DFF_N
    generic map(
        N => 20
    )
    port map(
        ...
    );

-- Fourth layer of adders
l4_s1: Ripple_carry_adder
    generic map(
        Nbit => 21
    )
    port map(
        ...
    );
```

```
end architecture beh;
```

3.6 Look-Up Table

As introduced before, we decided to exploit a **LUT** to implement the sigmoid function and get the correct output value, giving in input the result of the sum computed in the previous steps. As we can see in the code, the module takes in input an address on 12 bits, since we have 4096 entries, and produces an output on 16 bits. The VHDL code of the **LUT** was generated with a Python script (Gen_LUT.py). To find the right values to insert in the table, a quantization method was needed, to select the precision and calculate the LSB. In this case the choice was to adopt the **reach the LSB** method using the following formula:

$$LSB = \frac{max(x)}{2^{N-1} - 1} \tag{3.1}$$

```
entity LUT_4096 is
    port (
        address : in  std_logic_vector(11 downto 0);
        sigmoid_out : out std_logic_vector(15 downto 0)
    );
end LUT_4096;

architecture beh of LUT_4096 is
    type LUT_t is array (natural range 0 to 4095) of integer;
    constant LUT: LUT_t := (
    0 => 16384,
    ...
    4095 => 32767
    );

begin
    sigmoid_out <= std_logic_vector(TO_SIGNED(LUT(TO_INTEGER
    (unsigned(address))),16));
end beh;
```

4 - Test Plan

To evaluate the correctness of the architecture, a set of testbenches was developed, in order to check the functionality of each module. The results obtained with the TBs were also compared with the ones produced by different Python scripts, that we created to make it easier to do the comparison.

4.1 Unsigned Parallel Multiplier

This module performs the unsigned multiplication. Through the analysis of the results of the testbench, we proved that this operation produces the correct result.

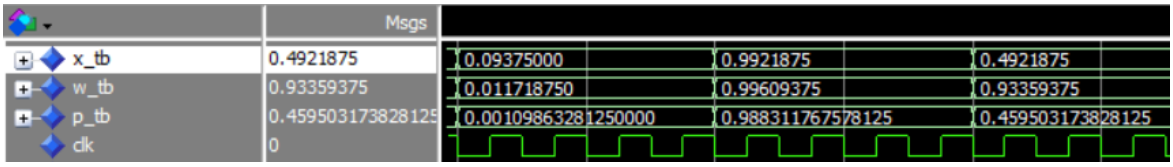

Figure 4.1: Unsigned Parallel Multiplier TB

```
0.09375
0.01171875

product: 0.0010986328125


0.9921875
0.99609375

product: 0.988311767578125


0.4921875
0.93359375

product: 0.459503173828125
```

Figure 4.2: Results computed by the Gen_Results_Multipliers.py script

4.2 Signed Parallel Multiplier

This module is responsible for the signed multiplication, that is handled by passing unsigned values to the **Unsigned_Parallel_Multiplier**. With this tests we

checked also the correctness of products with signed values and they are computed correctly.

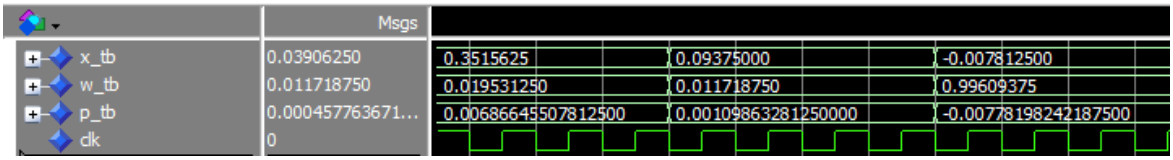

Figure 4.3: Signed Parallel Multiplier TB

```
0.3515625
0.01953125

product: 0.006866455078125


0.09375
0.01171875

product: 0.0010986328125


-0.0078125
0.99609375

product: -0.007781982421875
```

Figure 4.4: Results computed by the Gen_Results_Multipliers.py script

4.3 Parallel Multiplier

This component has the task to execute all the 10 products between the x and w input vectors. To do so 10 **Signed_Parallel_Multiplier** are used. Since this module takes many inputs and it's harder to test it manually, we wrote a Python script to automatically generate the testbench. The script also produces the correct results for each input configuration in a .txt file, to make it easier to do the comparison.

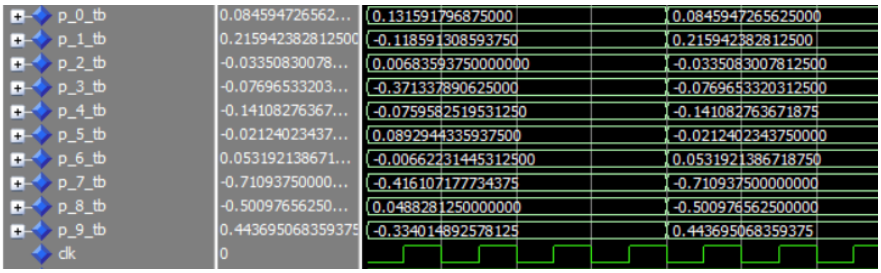

Figure 4.5: Parallel Multiplier TB

```
Test 1:
-0.21875 * -0.6015625 = 0.131591796875
-0.453125 * 0.26171875 = -0.11859130859375
0.0078125 * 0.875 = 0.0068359375
-0.40625 * 0.9140625 = -0.371337890625
-0.1484375 * 0.51171875 = -0.075958251953125
0.171875 * 0.51953125 = 0.08929443359375
0.0546875 * -0.12109375 = -0.006622314453125
-0.7890625 * 0.52734375 = -0.416107177734375
0.5 * 0.09765625 = 0.048828125
-0.4296875 * 0.77734375 = -0.334014892578125


Test 2:
-0.1640625 * -0.515625 = 0.0845947265625
-0.453125 * -0.4765625 = 0.2159423828125
-0.953125 * 0.03515625 = -0.03350830078125
0.1015625 * -0.7578125 = -0.07696533203125
0.5234375 * -0.26953125 = -0.141082763671875
-0.2265625 * 0.09375 = -0.021240234375
-0.1640625 * -0.32421875 = 0.053192138671875
-0.8125 * 0.875 = -0.7109375
-0.75 * 0.66796875 = -0.5009765625
-0.5234375 * -0.84765625 = 0.443695068359375
```

Figure 4.6: Results computed by the TB_Parallel_Multiplier_gen.py script

4.4 Adder Tree

This component performs the sum between the products executed by the **Parallel Multiplier** and the *bias*. Also in this case the testbench was generated with the help of a Python script, in figure 4.8 an example of comparison by means of a generated .txt file is shown.

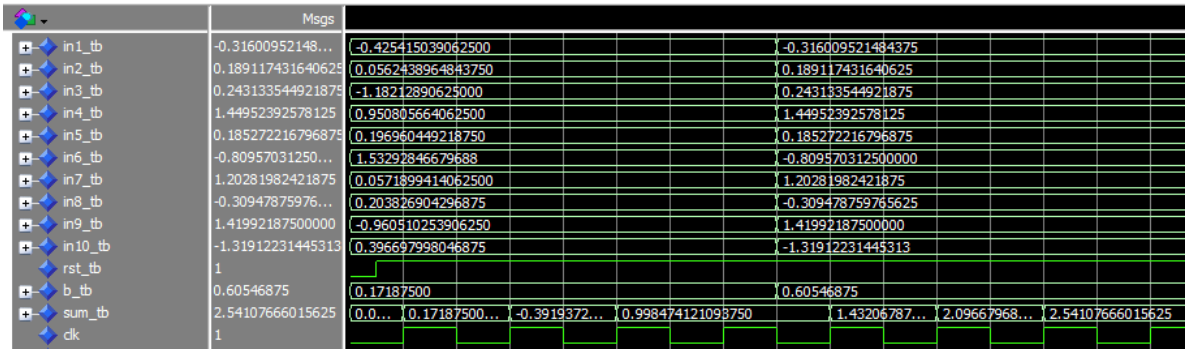

Figure 4.7: Adder Tree TB

```
11100100110001100       value 1: -0.4254150390625
00000011100110011       value 2: 0.056243896484375
10110100010110000       value 3: -1.18212890625
00111100110110100       value 4: 0.9508056640625
00001100100110110       value 5: 0.19696044921875
01100010000110111       value 6: 1.532928466796875
00000011101010010       value 7: 0.05718994140625
00001101000010111       value 8: 0.203826904296875
11000010100001110       value 9: -0.96051025390625
00011001011000111       value 10: 0.396697998046875
000101100               bias: 0.171875

sum 1: 0.99847412109375
```

Figure 4.8: Results computed by the TB_Adder_Tree_gen.py script

4.5 Look-Up Table

To test the behaviour of the **LUT**, we created a testbench, in which we just check if the mapping between the address and the output is the correct one, as defined in the VHDL code.

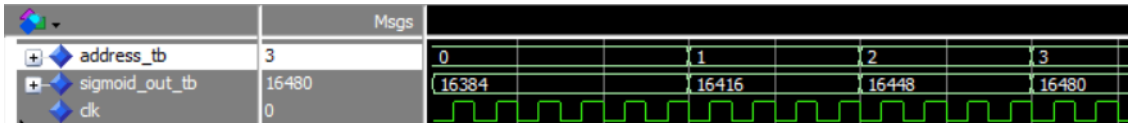

Figure 4.9: Look-Up Table TB

4.6 Perceptron (Static)

To test the functionality of the overall architecture, both static and dynamic versions of testbenches were developed. In the static ones, we evaluated the behaviour of the Perceptron to check if the sigmoid function was reproduced correctly. For this reason, 3 different static testbenches were created, each of one to test a different part of the function.

In figure 4.10, the results produced by the first static testbench of the **Perceptron** are shown. This test addresses the behaviour of the function around 0, that is almost linear as expected. In this testbench the input vectors are all zeros, except for the *bias*, that is tested in all its range incrementally. In the figure, we can see that there is a first phase in which the output grows linearly (starting from 0 and then assuming bigger values), that corresponds to the positive bias inputs. Then, at a certain point, there is a decrease in the output that happens when the first negative value is passed to the **Perceptron**. From that moment on, all possible negative values of the bias are tested, giving results that increase almost linearly also in this case.

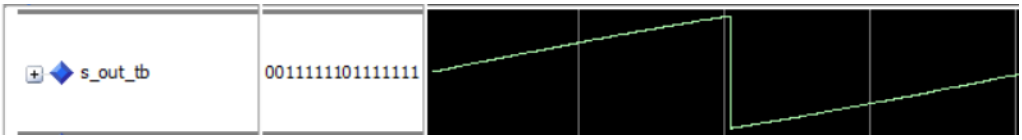

Figure 4.10: Perceptron TB 1

In the second static testbench developed for the **Perceptron** we assessed the behaviour of the function for high input values. In this case all input vectors were set to 1 and the bias was changed incrementally. As expected, the conduct is almost constant, as in the case of the proper sigmoid function, that can be observed in figure 1.2.

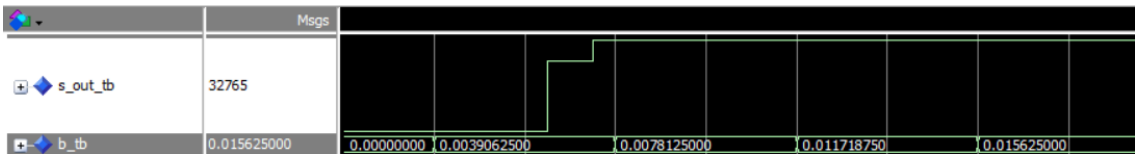

Figure 4.11: Perceptron TB 2

In the third testbench it was tested the opposite case, the one with minimum values. To have -1 as result of each product, the values of the x vectors were all set to 1 and the ones of the vector w to -1. Also in this case the *bias* values are changed incrementally.

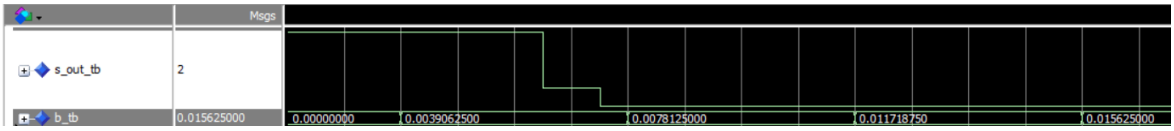

Figure 4.12: Perceptron TB 3

4.7 Perceptron (Dynamic)

The purpose of this testbench is twofold, it has to:

- evaluate the correctness of the Perceptron implementation across the whole range of possible values that can be given in input to the circuit;
- allow the comparison of the results with the ones produced by a Python script.

To do so we created a testbench called **Perceptron_Results_tb** and a Python script called Gen_Results.py. The latter produces a .txt file from which we can check the results obtained by the testbench, we can see for example tests 2 and 3 in figure 4.13 and 4.14. The values computed by the script are:

- **Sum**: sum between x and w input vectors and the bias (corresponds to the signal s_r3_out);

- **Address of the LUT**: address that the LUT should have in input (corresponds to address);
- **Sigmoid output**: computed output of the sigmoid function (corresponds to the first s_out_tb);
- **Quantized sigmoid output**: output of the sigmoid divided by the LSB_out value (corresponds to the second s_out_tb).

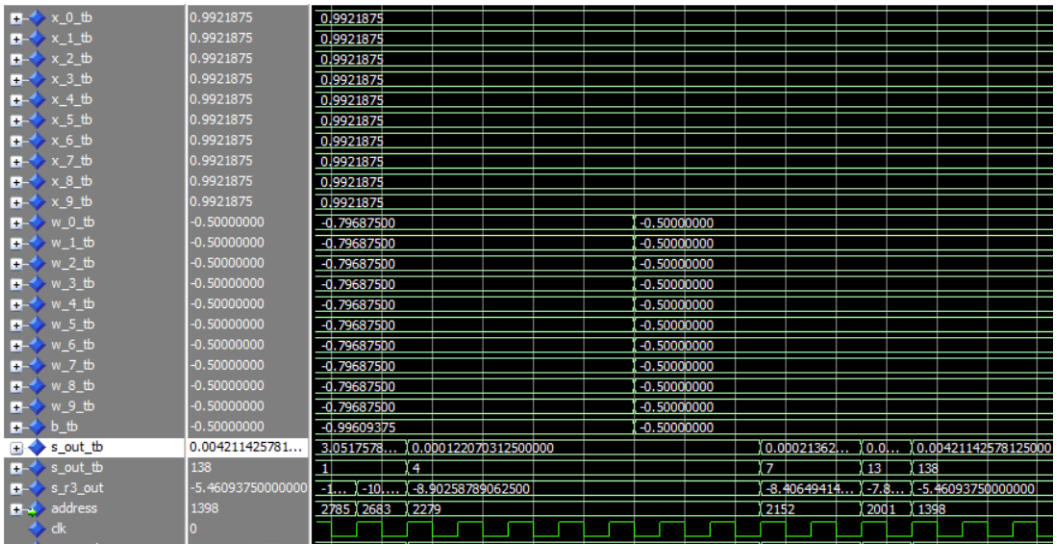

Figure 4.13: Perceptron Dynamic TB

```
------------------------------------------------------------------------------------------------------------

TEST 2:

x:[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]     w:[-0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8]     b:-1

Sum:   -9.0

Address of the LUT:   2303

Sigmoid output:   0.00012339457598623172

Quantized sigmoid output:   4

------------------------------------------------------------------------------------------------------------

TEST 3:

x:[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]     w:[-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5]     b:-0.5

Sum:   -5.5

Address of the LUT:   1408

Sigmoid output:   0.004070137715896128

Quantized sigmoid output:   133

------------------------------------------------------------------------------------------------------------
```

Figure 4.14: Results created by the Gen_Results.py script

Comparing the results, it can be observed that the values are almost equal, of course there is an error due to the approximation. In fact we can't represent precisely the

input and output values, since we are using a limited number of bits, using a fixed point representation. In the results computed by Python there is more accuracy, because values are represented with a higher number of bits. Nevertheless, for our purposes, we can say that the circuit has the expected behaviour and the error is acceptable.
The output of the Perceptron module was also visualized graphically, to compare the obtained behaviour with the one of the sigmoid function, as shown in figure 4.15. The architecture behaves as expected given the input provided, that start with the smallest possible values and then increase up to the highest possible ones. The step in the middle corresponds to the test number 5, in which the sum is equal to 0 and the output of the sigmoid function has to be 0.5.

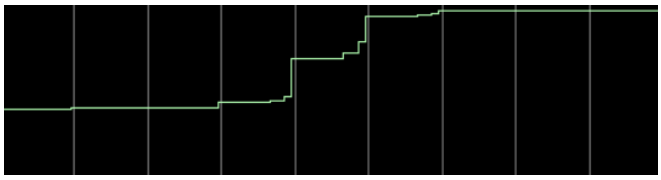

Figure 4.15: Output produced by the testbench

5 - Vivado Report

Finally, the Vivado tool was used, in order to realize and analyze the following phases:

- Elaborated design analysis
- Synthesis analysis
- Implementation analysis

As working device the xc7z010clg400-1 FPGA was selected.

5.1 Elaborated Design

Before doing the synthesis, Vivado can generate a **Register Transfer Level (RTL)** description of the architecture, that we compared with the one we planned to realize, described previously in figure 2.1. We can conclude that the design was reproduced successfully.

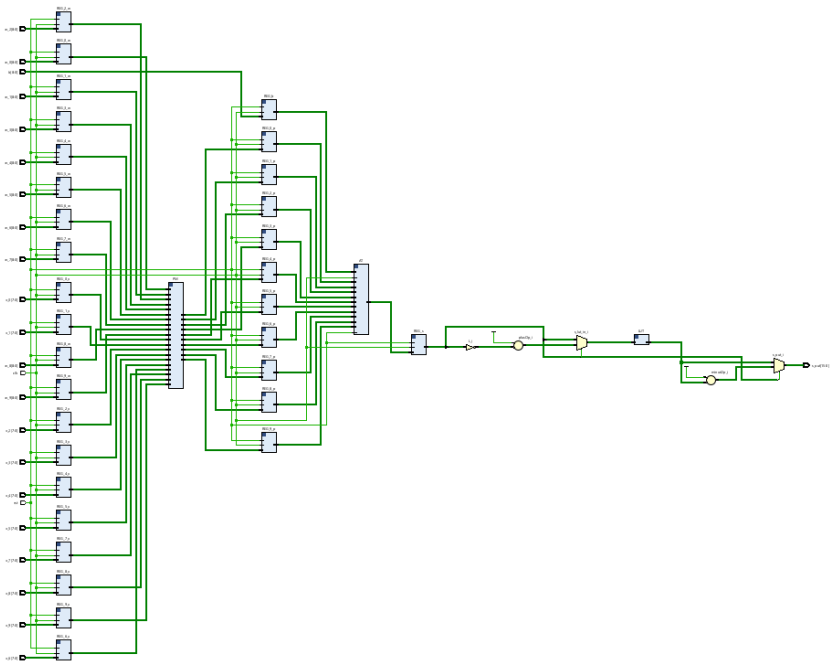

Figure 5.1: RTL schematic

5.2 Synthesis

During the synthesis phase, the modules described with VHDL are mapped to circuits that the **FPGA** is able to implement, exploiting the components in the **Netlist**. Vivado tries to do that respecting the given constraints, in our case the only one was the **clock period**, that, after many tests, we set to a final value of 14.27 ns. As we can see in figure 5.2, the synthesis phase completed correctly and no errors or warnings were produced by the tool.

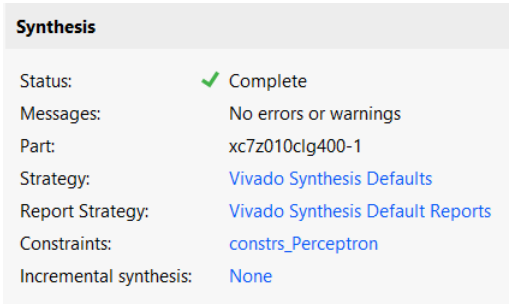

Figure 5.2: Synthesis results

5.2.1 Timing Report and Critical Path

The timing report is the following:

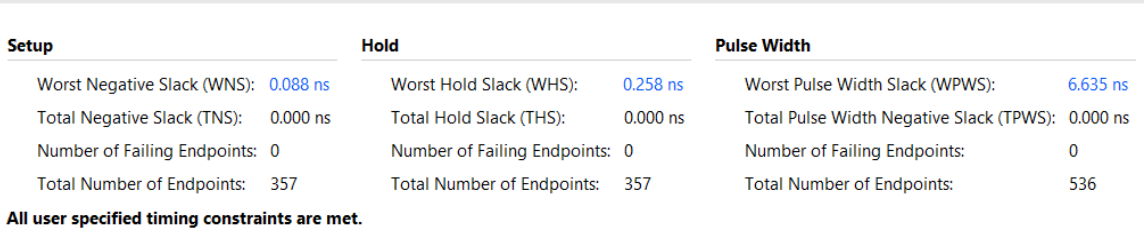

Figure 5.3: Timing Report

With the specified constraint there was **no negative slack**, so the clock period is sufficiently high in this case. We tested different values for the **constraint**, concluding that the **maximum clock frequency** that we can obtain is around **70.07 MHz**. The maximum clock frequency can also be obtained through the following formula:

$$f_{max} = \frac{1}{T_{clk} - WSN} \approx 70 \text{ MHz} \tag{5.1}$$

The most **critical path** in the architecture is the one shown in figure 5.4. This kind of path is present for 10 times inside the circuit, always with a delay of 14.160 ns. It starts from one of the registers that store an x input vector, and ends in one of the register that store a product vector from the **Parallel Multiplier**. So the

critical path is due to the multiplication operation, that is the most complex one implemented in the circuit.

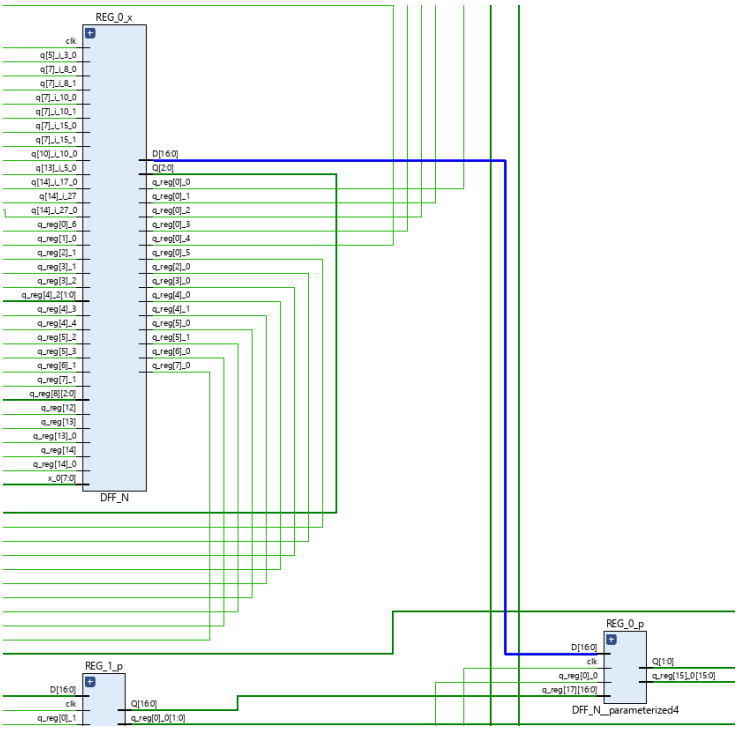

Figure 5.4: Critical path

5.2.2 Utilization

The Utilization Report produced after the Synthesis phase is the following one:

Summary

Resource	Utilization	Available	Utilization %
LUT	2207	17600	12.54
FF	536	35200	1.52

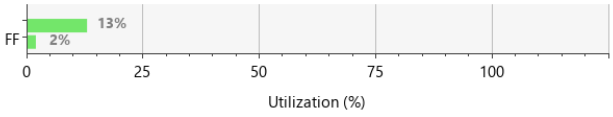

Figure 5.5: Utilization

As we can see, the resources needed to implement our circuit are far below the total number of components available, so it can be said that there are enough **LUTs** and **Flip Flops** to realize the architecture.

5.2.3 Power Consumption

A first estimate of the Power Consumption was produced after the Synthesis, that can be shown in figure 5.6.

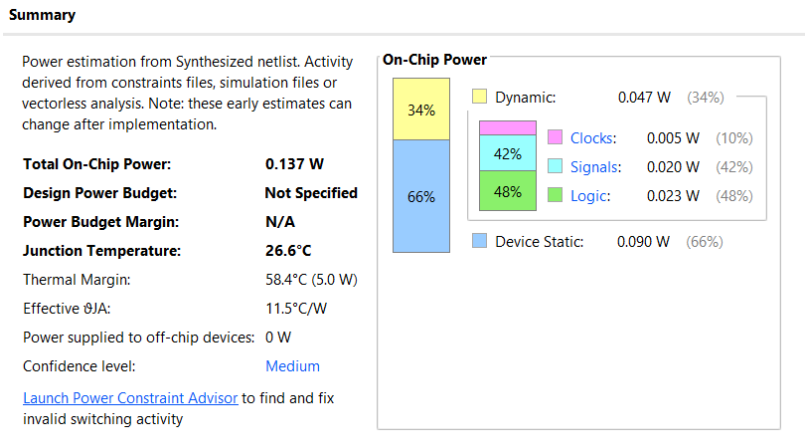

Figure 5.6: Power consumption

The power required was 137 mW (66% of which consists in **Static Power** and 34% in **Dynamic Power**).

5.3 Implementation

During the implementation phase the **Place and Route** is performed, that, as the name suggests, decides were to place the components in the FPGA and defines all the wires needed for the interconnections. Before that, there is the **I/O Planning**, that consists in mapping the ports on the FPGA with the ones in the Elaborated design. Since the number of **I/O ports** that we would need to perform the mapping in this way is too high, in order to bypass this problem, the implementation was executed in *out_of_context* mode. The results of the implementation are the following:

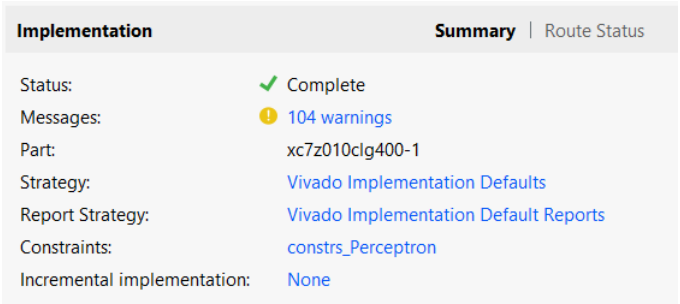

Figure 5.7: Implementation results

In this case some warnings appeared, they will be explained later in detail.

5.3.1 Timing Report and Critical Path

The timing report is the following:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.150 ns	Worst Hold Slack (WHS):	0.144 ns	Worst Pulse Width Slack (WPWS):	6.635 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:	357	Total Number of Endpoints:	357	Total Number of Endpoints:	536

All user specified timing constraints are met.

Figure 5.8: Timing report

In this case, despite the fact that in the implementation also the delays of the interconnections are taken into account, the **Worst Negative Slack** increased. This could be due to the fact that in this phase also different optimizations are executed, that can lead to better performance. During the implementation, the critical paths are analyzed in more details, with more information available. In this case, the most critical path for the architecture was the one shown in figure 5.9. Also in this case this path goes from the inputs to the output registers of the **Parallel Multiplier**.

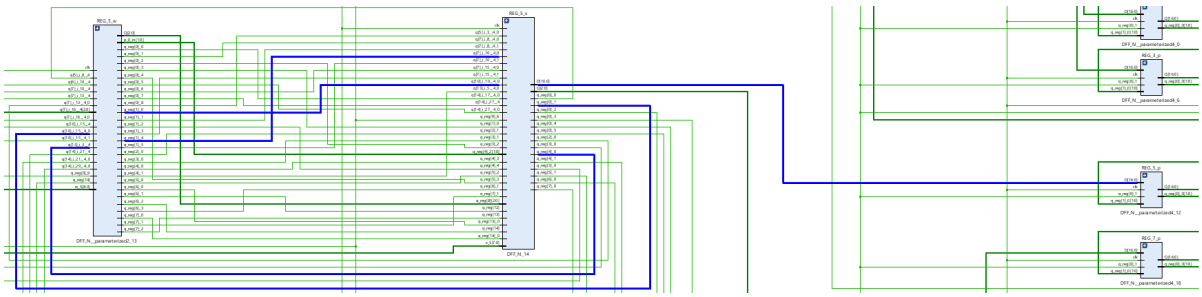

Figure 5.9: Critical path

5.3.2 Utilization

The results from the **Utilization Report** are the following:

Summary

Resource	Utilization	Available	Utilization %
LUT	2214	17600	12.58
FF	536	35200	1.52

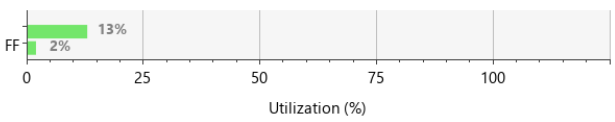

Figure 5.10: Utilization

As we can see, the value of FFs required is the same, while there was a slight increase in the number of LUTs needed that went from 2207 to 2214 with an increase in the **utilization percentage** of 0.04%.

5.3.3 Power Consumption

Regarding the **Power Consumption** of the circuit, there was a slight increase in the **Total On-Chip Power**, that went from 137 mW in the synthesis to 138 mW after the implementation. There was also a small change in the distribution of this power. In particular, the dynamic power contribution increased by 1% with respect to the previous value, while the static power had a corresponding decrease of 1%.

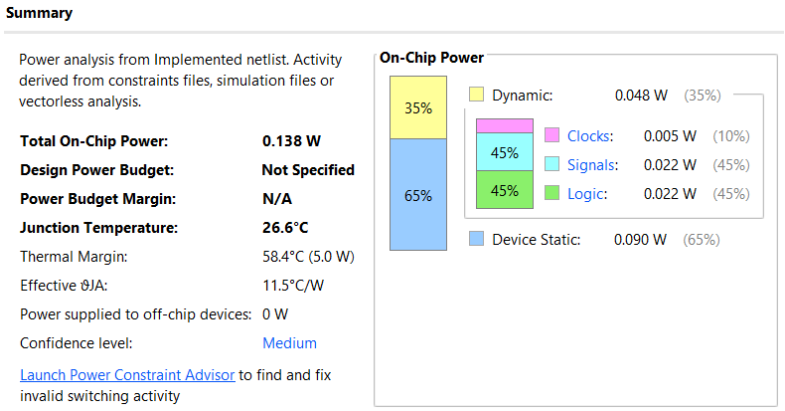

Figure 5.11: Power Consumption

5.3.4 Warnings

The warnings produced after the implementation are the following:

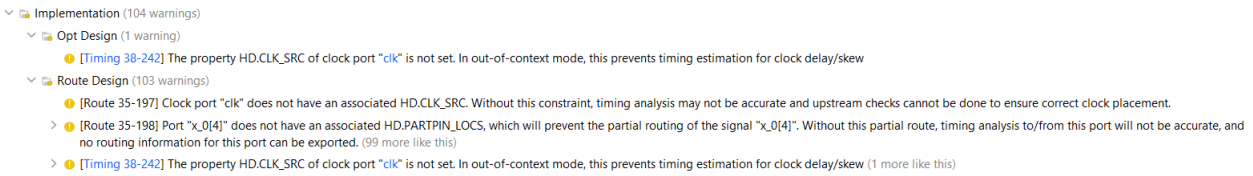

Figure 5.12: Warning messages

As we can see, in this case there were some warnings produced, related to the **clock** and the other **ports**. The warnings say that this issue doesn't allow the tool to perform accurate timing estimations. Despite that, under the supervision of Eng. Nannipieri we concluded that these warnings can all be attributable to the fact that we ran the **Synthesis** in *out_of_context* mode, so can be ignored.

6 - Conclusions

The **Perceptron** to implement in our case, corresponds to a circuit that takes 10 vectors x (on 8 bits), 10 vectors w (on 9 bits) and a *bias* vector b (on 9 bits) as inputs, and gives as output a result on 16 bits. To design the architecture we tried to strike a balance between performances and complexity, but different trade-offs and implementations can be realized, depending on the needs that the circuit has to satisfy.
If the main goal is to **maximize the Performance** of the architecture, one possible improvement in the design could be to substitute the Ripple Carry Adders exploited in our approach with a **pipelined version**, reducing the critical path and leading to a speed-up of the circuit.
On the other hand, if the goal is to **reduce the complexity** of the design, one possible solution could be to use a **sequential approach** instead of an Adder Tree, to sum the products and the bias. This would lead to a simpler architecture, but with significant downsides in terms of performance.