Computer Engineering

Electronic and Communication Systems

Perceptron

Project Report

TEAM MEMBERS:
Olgerti Xhanej

Academic Year: 2020/2021

Contents

1 Introduction **2**

1.1 Problem Description . . . 2

1.2 Applications . . . 3

1.3 Possible Architectures . . . 4

2 Architecture **5**

2.1 Multiplication Circuit Architecture . . . 5

2.2 Adder Circuit Architecture . . . 8

2.3 Activation Function Circuit Architecture . . . 11

3 VHDL CODE **12**

3.1 Modules List . . . 12

3.2 Perceptron . . . 12

3.3 Parallel Multiplier . . . 13

3.4 Unsigned Parallel Multiplier . . . 15

3.5 Tree Adder . . . 17

3.5.1 Ripple Carry Adder Pipelined . . . 18

3.6 LUT . . . 20

3.6.1 Lut generation code . . . 21

4 Test Plan **22**

5 XILINX VIVADO Report **23**

6 Conclusion **24**

1 — Introduction

1.1 Problem Description

The main goal of the activity described in this report is the following: realizing a network implementing a **perceptron** with a **sigmoid activation function**.
Before describing the whole design and implementation process a very little introduction about the architecture must be done.

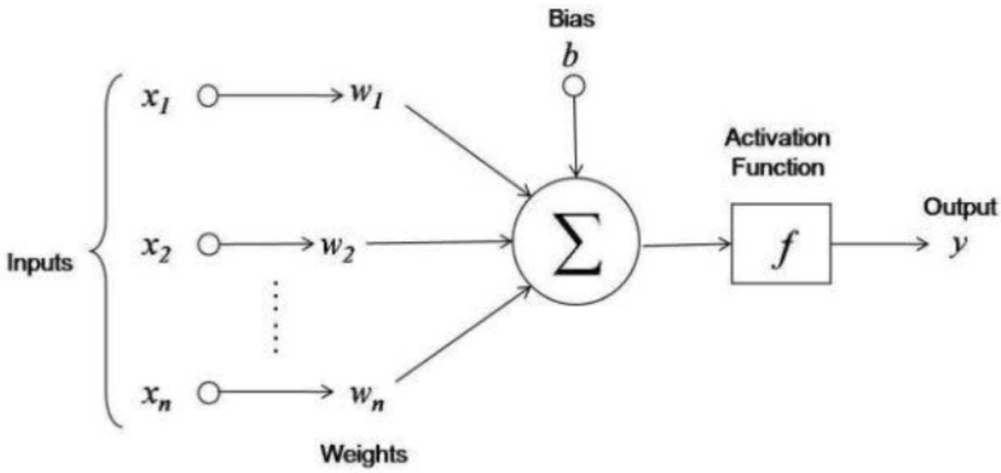


Figure 1: Perceptron Architecture

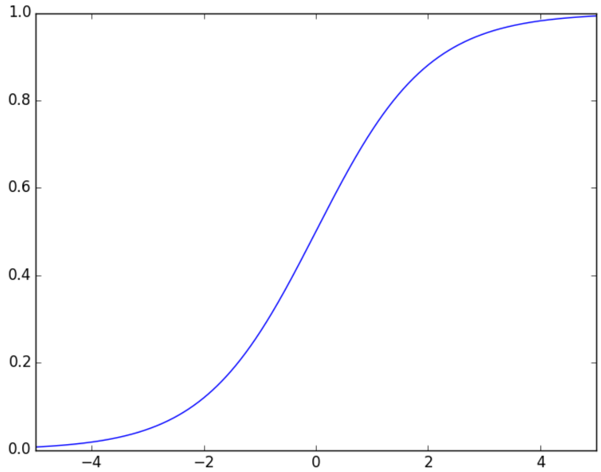
A **Perceptron** is a *binary classifier that maps his inputs to a specific output y = f(z), where f() is the **activation function** of the perceptron.* The inputs are real numbers and the input z of the activation function is obtained as:

$$z = b + \sum_{i=0}^{N_L-1} w_i x_i \tag{1}$$

Every input x_i, every weight w_i and the bias b are real numbers in the range of $[-1, 1]$.
The **activation function**, in our case, will be a **sigmoid function**, described as follows:

$$y = \frac{1}{1 + e^{-z}} \tag{2}$$

Figure 2: Sigmoid Function Plot

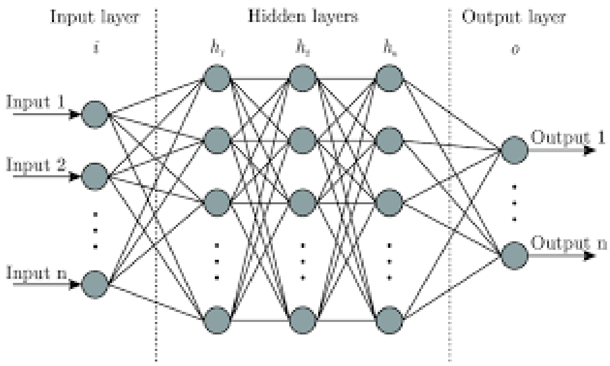


Where z is the result of the equation (1.1).

1.2 Applications

A single perceptron is the building block of *artificial neural networks*, in which different layers of perceptrons are connected. The output of the neural network is a real number and could be use to classify *complex objects*: patterns, human faces, handwritings, medical diagnosis, e-mail spams.

Figure 3: Neural network example



In the image above there is a simple schema of a neural network, in which the circles represent the perceptrons.

1.3 Possible Architectures

The main architecture will be made up by three main logical parts, from an higher-lever point of view:

- **Multiplication Circuit**: implementation of the multiplication operation between each input x_i and each weight w_i.
- **Adder Circuit**: implementation of the addition between the results of the former phase and the bias b.
- **Activation Function Circuit**: implementation of the computation of the sigmoid function.

In the next chapter the architecture will be documented with more precision. Different project choices could be made for each logical part of the architecture:

- **Multiplication Circuit**: could be implemented through a **ROM-based solution** in which every possible result is stored and the two inputs represent the addresses for getting the result. This solution is good only with a very low number of bits, which is not our case: in fact the the ROM will be composed by $2^{(n_{w_i}+n_{b_i})}$ memory cells. In order to implement the multiplication circuit will be implemented through a **Paraller Multiplier**.
- **Adder Circuit**: different choice could be made to implement the adder circuit. Starting from the simplest to the more complex solution we can exploit the **Serial Adder**, the **Parallel Adder** or the **Parallel Adder with Pipeline** . The first one needs less logic but requires n clock cycles for computing an n bits result. The second solution improves the first one by computing one result in **one clock cycle**, on the other hand it could add some problems due to long logic chains between two register. The third solution is the best from the perspective of the number of clock cycles required and the **critical path**, in fact by adding some registers in between the computation of the bits will reduce the logic chains.
- **Activation Function Circuit**: As seen during the laboratory class, this part will be implemented by exploiting a Look-Up-Table. In order to do so, could be necessary a **truncation** of the result of the former computation in order to limit the size of the LUT. With 12 bits are necessary $2^{12} = 4096$ entries, which could be even reduced by performing some optimization by exploiting the sigmoid function symmetry.

2 — Architecture

In this chapter will be discussed deeply the architecture of the three main parts of the perceptron. The general structure could be summarized by the following schema:

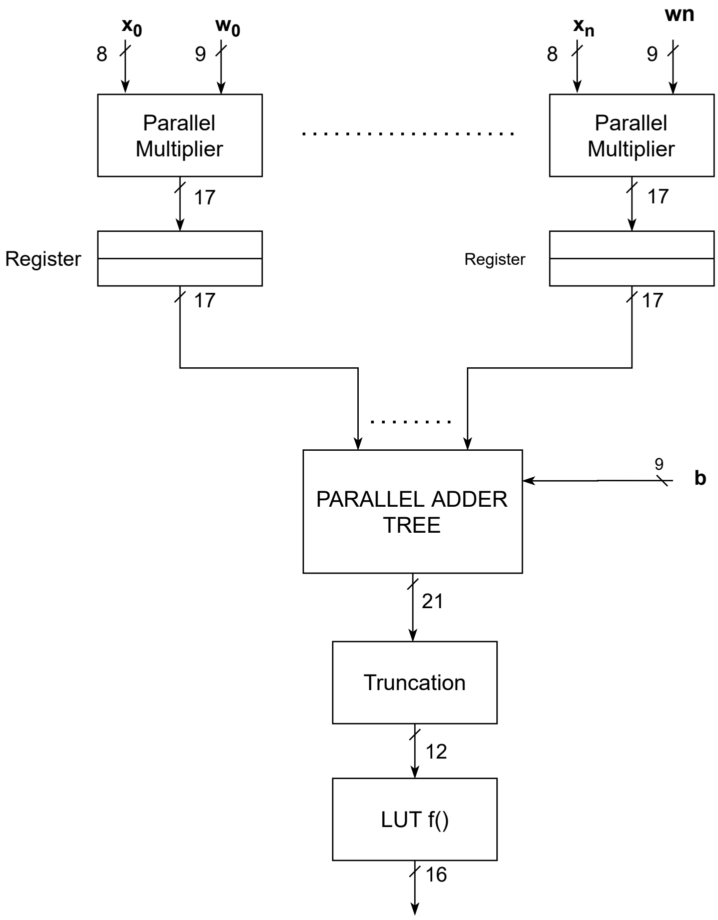


Figure 4: General Schema

2.1 Multiplication Circuit Architecture

The Multiplication Circuit, as said before, will be implemented through a Parallel Multiplier. The inputs b_i and w_i are composed respectively by $b_x = 8$ bits and $b_w = 9$ bits. In order to compute the multiplication in the correct way, the inputs need to be translated in the **unsigned form** and then is possible to perform the multiplication with the parallel multiplier. In the following image is presented the general schema of the Parallel Multiplier:

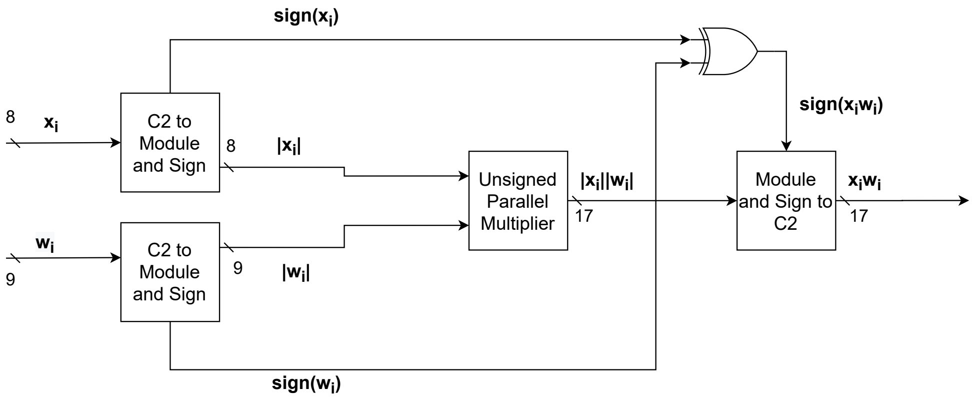


Figure 5: Parallel Multiplier Architecture

Notice that the sign of the result will be computed by a simple XOR operation between the inputs signs. The Unsigned Parallel Multiplier architecture is the following:

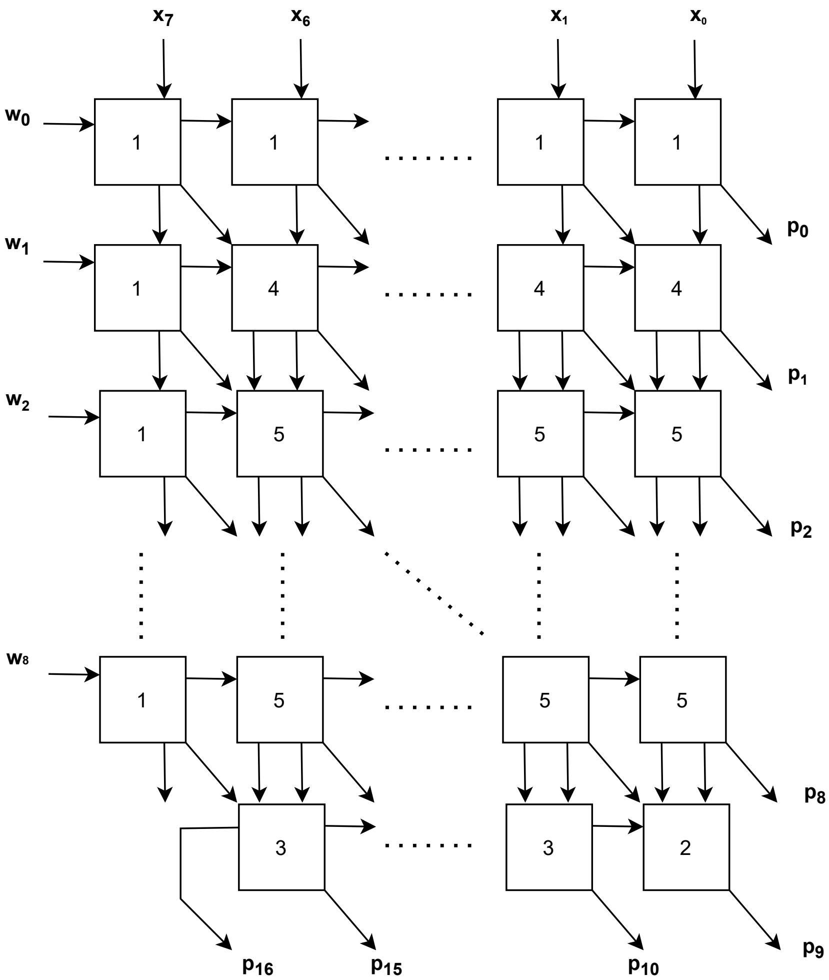


Figure 6: Unsigned Parallel Multiplier Architecture

Each logic block is translated with a related logic block:

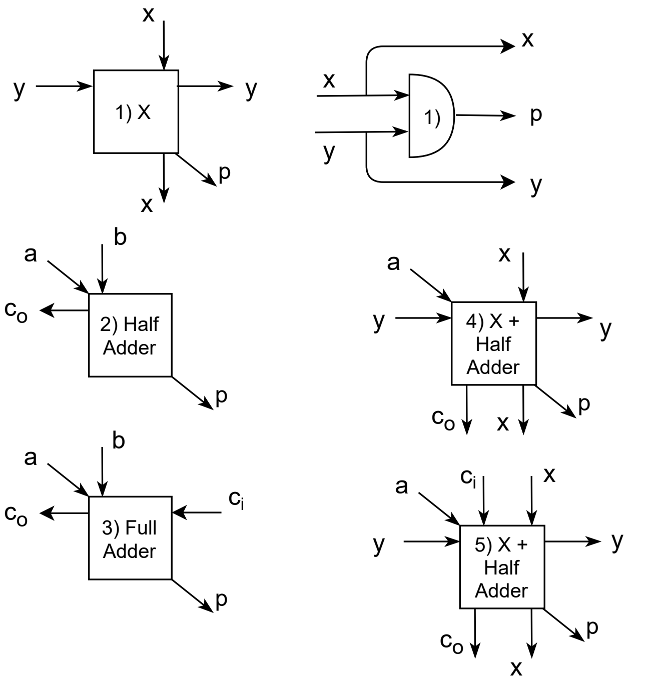


Figure 7: Unsigned Parallel Multiplier Architecture

2.2 Adder Circuit Architecture

In order to compute the equation (1.1) different sums need to be computed. The building block of this part will be the **Parallel Adder with Pipeline**: as said before, by adding some registers in between the Carry chains, the critical path impact can be reduced. Furthermore, by exploiting the parallel architecture, a single sum can be computed in a single clock cycle. In the next figure will be presented the Parallel Adder:

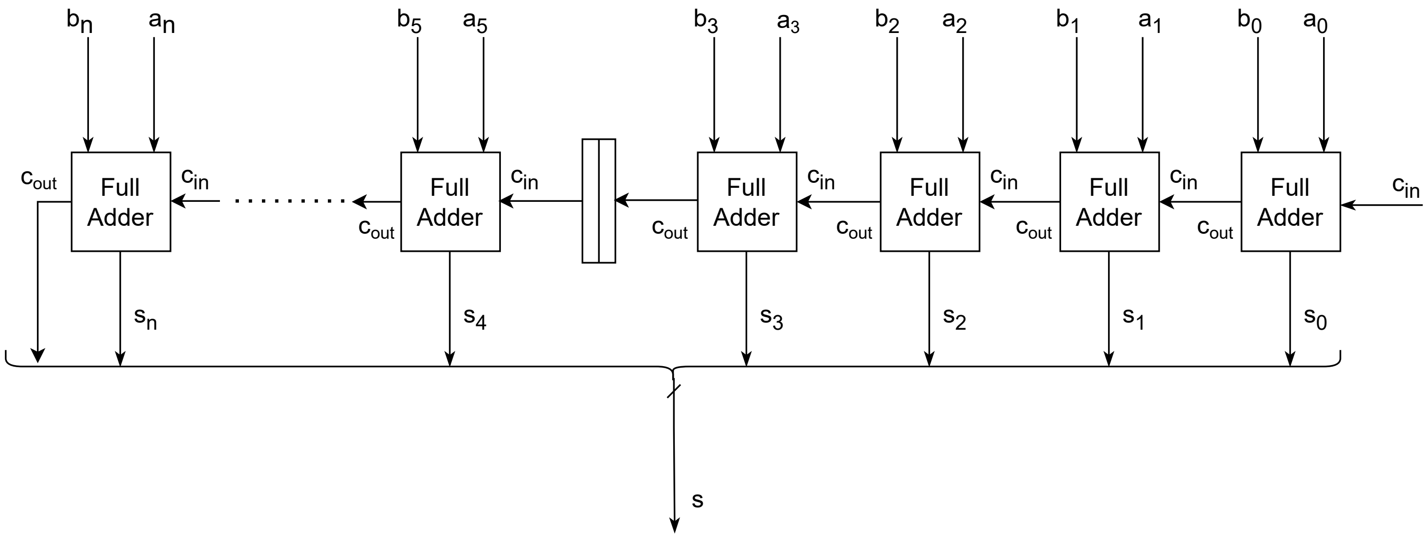


Figure 8: Parallel Adder Architecture

To implement the whole sum of 11 terms, in order to decrease the number of cycles needed to compute the whole sum and to reduce the number of bits needed, a tree approach has been chosen. The schema of the tree parallel adder is the following:

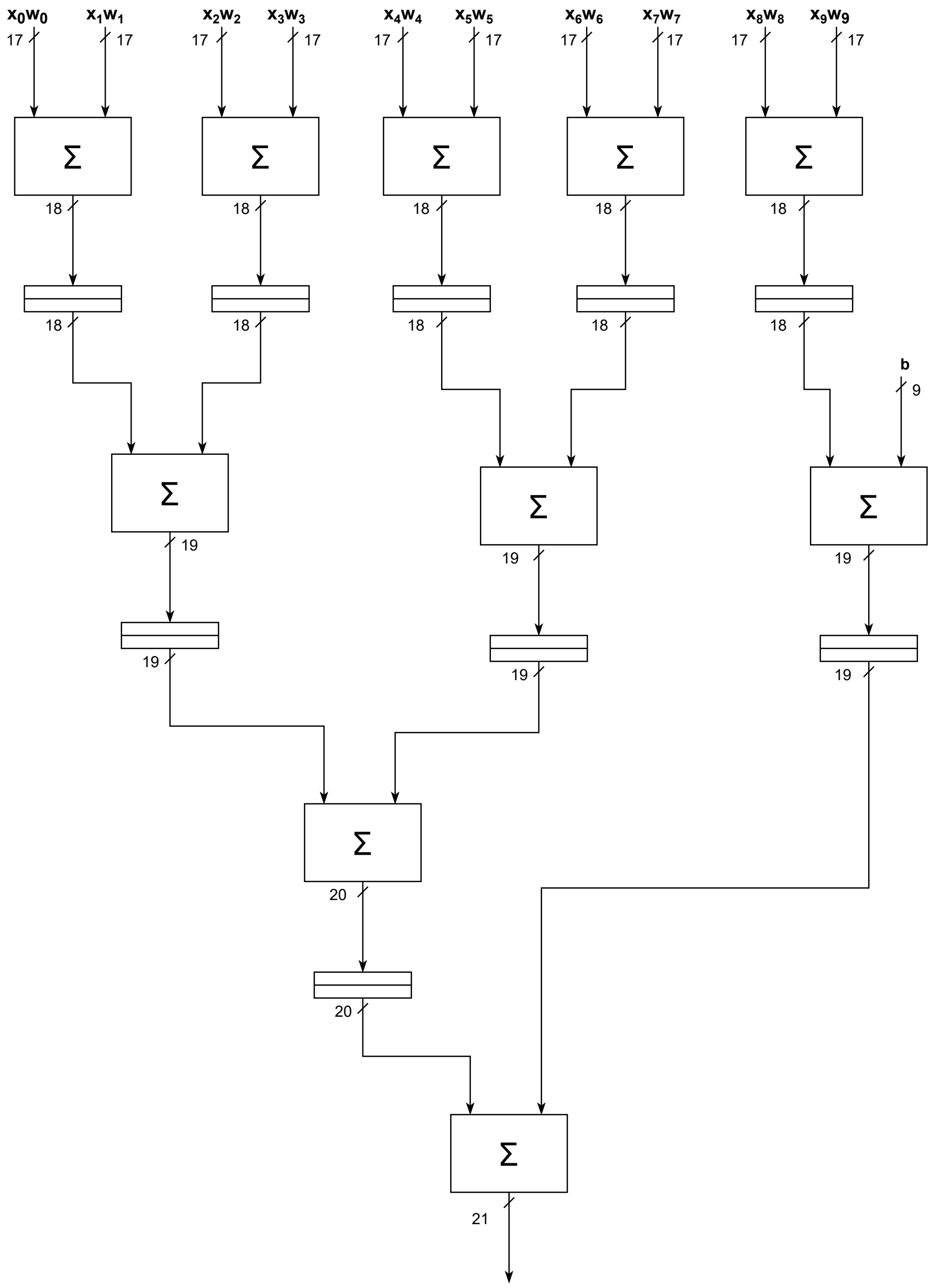


Figure 9: Parallel Multiplier Architecture

Some register has been put in between the sum to limit the critical path impact on the performances and clock period limit.

2.3 Activation Function Circuit Architecture

At the end of the computation of the latter phase the output is composed by 21 bits. The computation of the sigmoid function will be done through a **Look-Up-Table**, which will need 2^{21} =**2097152 entries** of different outputs with 16 bits. In order to reduce the size of the Look-Up-Table a truncation is needed: from 21 bits to 12 bits. In this case the Look-Up Table will be composed by 2^{12} =**4096 entries**, but, by exploiting the **odd symmetry** of the sigmoid, only 4096/2 = **2048 entries** are needed.

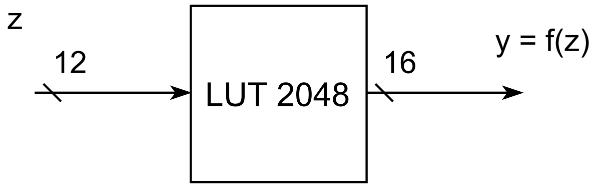


Figure 10: Look-Up Table Architecture

3 — VHDL CODE

In this chapter will be presented the main modules that compose the architecture of the **Perceptron with sigmoid activation function**.

3.1 Modules List

As presented in the last chapter, I have followed a similar approach for creating the architecture. The following modules were created:

- Perceptron
  - Parallel_Multiplier
    - Unsigned Parallel Multiplier
      - Full Adder
      - Half Adder
  - Tree_Adder
    - Ripple_Carry_Adder_Pipelined
      - DFF
      - Full Adder
  - Sigmoid_Lut_2048

A **bottom-up strategy** was followed in order to build up the architecture: starting from some modules that will made up the architecture, after finishing each of them some testbenches were written in order to test each building block of the **Perceptron** (See next chapter for details).

3.2 Perceptron

The main hardware description of the architecture. This module will connect all the other modules in order to create the correct architecture. In order to not show too much lines of code only the entity definition of this module will be shown.

```
entity Perceptron is
port(

  -- x_1 to x_10 inputs of the perceptron with 8 bits
  x_1: in std_logic_vector(7 downto 0);
```

```
    x_2: in std_logic_vector(7 downto 0);
    x_3: in std_logic_vector(7 downto 0);
    x_4: in std_logic_vector(7 downto 0);
    x_5: in std_logic_vector(7 downto 0);
    x_6: in std_logic_vector(7 downto 0);
    x_7: in std_logic_vector(7 downto 0);
    x_8: in std_logic_vector(7 downto 0);
    x_9: in std_logic_vector(7 downto 0);
    x_10: in std_logic_vector(7 downto 0);

    -- w_1 to w_10 inputs of the perceptron with 9 bits
    w_1: in std_logic_vector(8 downto 0);
    w_2: in std_logic_vector(8 downto 0);
    w_3: in std_logic_vector(8 downto 0);
    w_4: in std_logic_vector(8 downto 0);
    w_5: in std_logic_vector(8 downto 0);
    w_6: in std_logic_vector(8 downto 0);
    w_7: in std_logic_vector(8 downto 0);
    w_8: in std_logic_vector(8 downto 0);
    w_9: in std_logic_vector(8 downto 0);
    w_10: in std_logic_vector(8 downto 0);

    -- b input of the perceptron with 9 bits
    b: in std_logic_vector(8 downto 0);

    clk: in std_logic;
    rst: in std_logic;

    -- output of the perceptron 16 bits
    f_z: out std_logic_vector(15 downto 0)
  );
end Perceptron;
```

In the rest of this modules are instantiated and linked the various sub-modules that made up the **Perceptron** module.

3.3 Parallel Multiplier

This module has the duty to convert the inputs, which are signed with a 2's complement representation, link their unsigned representation with the **Unsigned Parallel Multiplier** module and then reconvert the product in the signed representation. The general architecture is shown in Figure 5.

```
entity Parallel_Multiplier is
  generic (
    Nbit_a : positive;
    Nbit_b: positive
  );
```

```
    port(
      a_p_signed: in std_logic_vector(Nbit_a - 1 downto 0);
      b_p_signed: in std_logic_vector(Nbit_b - 1 downto 0);
      -- The product will need Nbit_a + Nbit_b bits
      p_signed: out std_logic_vector(Nbit_a + Nbit_b - 1 downto 0)
    );
  end entity Parallel_Multiplier;

  architecture rtl of Parallel_Multiplier is

    -- Building blocks of the Parallel Multiplier
    component Unsigned_Parallel_Multiplier
      generic(
        Nbit_a : positive;
        Nbit_b : positive
      );
      port(
        a_p: in  std_logic_vector(Nbit_a - 1 downto 0);
        b_p : in  std_logic_vector(Nbit_b - 1 downto 0);
        p   : out std_logic_vector(Nbit_a + Nbit_b - 1 downto 0)
      );
    end component Unsigned_Parallel_Multiplier;


    -- Unsigned component (will work for the unsigned parallel multiplier
    signal p_unsigned: std_logic_vector(Nbit_a + Nbit_b - 1 downto 0);
    signal a_p_unsigned: std_logic_vector(Nbit_a - 1 downto 0);
    signal b_p_unsigned: std_logic_vector(Nbit_b - 1 downto 0);

    -- will carry the sign bit for the signed rapresentation of the inputs
    signal a_sign: std_logic;
    signal b_sign: std_logic;

  begin

    -- Compute the unsigned representation from the signed one
    a_p_unsigned <= std_logic_vector(abs(signed(a_p_signed)));
    b_p_unsigned <= std_logic_vector(abs(signed(b_p_signed)));

```

```
    -- 2's complement rapresentation, the result sign uis
   computed through the xor op. between a and b
    p_signed <= std_logic_vector(unsigned(not(p_unsigned)) +
   1) when (((a_sign xor b_sign) = '1')) else p_unsigned;

    -- Getting of the sign from a and b (the MSB of the C2
   representation)
    a_sign <= a_p_signed(Nbit_a - 1);
    b_sign <= b_p_signed(Nbit_b - 1);

    unsigned_parallel_mul: Unsigned_Parallel_Multiplier
      generic map(
        Nbit_a => Nbit_a,
        Nbit_b => Nbit_b
      )
      port map(
        a_p =>  a_p_unsigned,
        b_p =>  b_p_unsigned,
        p   => p_unsigned
      );

  end architecture rtl;
```

3.4 Unsigned Parallel Multiplier

This module will effectively compute a multiplication. Through the replication of the architecture shown in Figure 6 and 7 this module will return the *unsigned product* of two *unsigned operands*. In order to implement the architecture in the simplest way a **Structured approach** has been followed in the description. Some lines of code were just skipped in order to show only a general schema, for further details there is also the source code.

```
entity Unsigned_Parallel_Multiplier is
  generic (
    Nbit_a : positive;
    Nbit_b: positive
  );
  port(
    -- Unsigned representation of inputs
    a_p: in std_logic_vector(Nbit_a - 1 downto 0);
    b_p: in std_logic_vector(Nbit_b - 1 downto 0);

    -- p = a_p * b_p
    p: out std_logic_vector(Nbit_a + Nbit_b - 1 downto 0)
  );
end entity Unsigned_Parallel_Multiplier;

```

```
architecture rtl of Unsigned_Parallel_Multiplier is
  -- Building blocks of the Unsigned Parallel Multiplier
  component FULL_ADDER is
    ...
  end component;

  component HALF_ADDER is
    ...
  end component;

  -- Will hold the carry signals among the whole architecture
  signal carry_signal: std_logic_vector((Nbit_a - 1)*(Nbit_b - 1) - 1 downto 0);
  signal last_carry_signal: std_logic_vector((Nbit_b - 1) downto 0);

  -- Will hold the sum result of the FA and HA among the whole architecture
  signal sum_signal: std_logic_vector((Nbit_a - 1)*(Nbit_b - 2) - 1 downto 0);

  -- will hold the precomputed values for the inputs a and b of the various Half Adder and Full Adder
  signal a_multiplier: std_logic_vector(Nbit_a + Nbit_b - 2 downto 0);
  signal b_multiplier: std_logic_vector((Nbit_a - 1)*(Nbit_b - 1) - 1 downto 0);

begin

  -- First bit of the result
  p(0) <= (a_p(0) and b_p(0));


  -- Computation of the various inputs of each HA and FA
  d_process: process(a_p, b_p)
  begin

  for j in 1 to Nbit_b loop
  a_multiplier(j - 1) <= (a_p(0) and b_p(Nbit_b - j));
  end loop;

  ...
  end process d_process;


  -- Architecture will follow schema of the Parallel Multiplier
```

```
    -- Row index i
    GEN_a: for i in 1 to Nbit_a generate
      -- Column index j
      GEN_b: for j in 1 to Nbit_b - 1 generate
        FIRST_ROW: if i=1 generate
        -- In the first Row only HA
        LEFT: if j < Nbit_b -1 generate
        ROW1_LEFT: HALF_ADDER
        port map
        (
        a    => a_multiplier(j - 1),
        b    => b_multiplier(j - 1),
        s    => sum_signal(j - 1),
        cout => carry_signal(j - 1)
        );
        end generate LEFT;
        RIGHT: if j = Nbit_b - 1 generate
        ROW1_RIGHT: HALF_ADDER
        port map
        (
        a    => a_multiplier(j - 1),
        b    => b_multiplier(j - 1),
        s    => p(1), -- Result bit
        cout => carry_signal(j - 1)
        );
        end generate RIGHT;
        end generate FIRST_ROW;


        INTERNAL_ROW: if i > 1 and i < Nbit_a generate
          ...
        end generate INTERNAL_ROW;


        LAST_ROW: if i = Nbit_a generate
          ...
        end generate LAST_ROW;
      end generate GEN_b;
    end generate GEN_a;
  end architecture rtl;
```

3.5 Tree Adder

This module will take up the ten multiplication results and the bias and will sum up every term by making the computation shown at the equation (1). Even in this case will not be shown the architectural code due to the fact that consist only in linking some submodules in the proper way in order to replicate the architecture shown in Figure (9).

```
entity Tree_Adder is
  port(
    -- Inputs: result of the multiplication of xi*wi
    in_1: in std_logic_vector(16 downto 0);
    in_2: in std_logic_vector(16 downto 0);
    in_3: in std_logic_vector(16 downto 0);
    in_4: in std_logic_vector(16 downto 0);
    in_5: in std_logic_vector(16 downto 0);
    in_6: in std_logic_vector(16 downto 0);
    in_7: in std_logic_vector(16 downto 0);
    in_8: in std_logic_vector(16 downto 0);
    in_9: in std_logic_vector(16 downto 0);
    in_10: in std_logic_vector(16 downto 0);

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

    -- Output
    z: out std_logic_vector(20 downto 0)
  );
end Tree_Adder;
```

3.5.1 Ripple Carry Adder Pipelined

This module will be the main building block of the **Tree Adder** module. In order to reduce the logic chain some registers were added by exploiting the **DFF** module as seen in the Lab lectures.

```
entity Ripple_Carry_Adder_Pipelined is
generic (Nbit: positive);
  port(
    -- Inputs
    a_r: in std_logic_vector(Nbit-2 downto 0);
    b_r: in std_logic_vector(Nbit-2 downto 0);
    cin_r: in std_logic;
    cout_r: out std_logic;

    -- Will store the result of a_r+b_r
    s_r: out std_logic_vector(Nbit-1 downto 0);
    clk: in std_logic;
    rst: in std_logic
  );
end Ripple_Carry_Adder_Pipelined;

architecture rtl of Ripple_Carry_Adder_Pipelined is
```

```vhdl
  -- Building blocks of the Ripple Carry Adder Pipelined
  component FULL_ADDER
    ...
  end component FULL_ADDER;

  -- Need of a register to obtain the pipelined version
  component DFF
  ...
  component DFF;

  -- Will propagate the carry signal among the whole architecture
  signal carry_signal: std_logic_vector(Nbit-1 downto 1);

  -- Will store the outputs signal of the registers
  signal dff_signal: std_logic_vector(Nbit-1 downto 0) := (others => '0');

begin
  -- Implemented in a structured way in a similar fashion as seen in the Lab lessions
  GEN: for i in 1 to Nbit generate
    FIRST: if i=1 generate
    -- First FA
      FFI: FULL_ADDER port map (a_r(0), b_r(0), cin_r, s_r(0), carry_signal(1));
    end generate FIRST;
    INTERNAL: if i > 1 and i < Nbit generate
      -- Need of Register detection
      PIPE: if (i mod 3 = 0) generate
        DFF_I: DFF
          port map(
          d_dff      => carry_signal(i-1),
          clk_dff    => clk,
          resetn_dff => rst,
          q_dff      => dff_signal(i-1)
        );
        FFI: FULL_ADDER port map (a_r(i-1), b_r(i-1), dff_signal(i-1), s_r(i-1), carry_signal(i));
      end generate PIPE;

      -- No need of a register
      NOT_PIPE: if (i mod 3 /= 0) generate
        FFI: FULL_ADDER port map (a_r(i-1), b_r(i-1), carry_signal(i-1), s_r(i-1), carry_signal(i));
      end generate NOT_PIPE;
    end generate INTERNAL;

```

```
    -- Implicit extension (the inputs have Nbit-2 bits, the output has Nbit-1 bits and there
    -- are Nbit-1 FA so the last bit is replicated in order to make the extension in the
    -- correct way in C2 representation)

    LAST: if i=Nbit generate
    FFI: FULL_ADDER port map (a_r(Nbit-2), b_r(Nbit-2), carry_signal(Nbit-1), s_r(Nbit-1), cout_r);
    end generate LAST;
  end generate GEN;
end rtl;
```

3.6 LUT

This module will store every possible output of the sigmoid function in the input range of $[-11; +11]$ with 12 bits of precision. The input will be treated as an address signal to obtain the correct output value in the same fashion as shown in the Laboratory lectures.

```
entity sigmoid_lut_2048 is
  port (
    address : in  std_logic_vector(10 downto 0);
    dds_out : out std_logic_vector(15 downto 0)
  );
end sigmoid_lut_2048;


-- Output between [-11; +11], rapresented with fixed point
-- [Reach the LSB method]
-- LSB(in) = (11)/(2^11 - 1) = 0.0053737176355642403517342452369 3
-- LSB(out) = (1)/(2^15 - 1) = 3.0518509475997192297128208258309e-5
-- inputs -> [-11, +11]
-- outputs -> [0, 1]
-- Q(f(x)) = round(f(x)/LSB(out))*LSB(out)
--
-- What to store in the lut? round(f(x)/LSB(out)) for x in [0; 2047]*LSB(in)

architecture rtl of sigmoid_lut_2048 is
  type LUT_t is array (natural range 0 to 2047) of integer;
  constant LUT: LUT_t := (
    0 => 16384,
    1 => 16428,
    ...
    2046 => 32766,
```

```
    2047 => 32766
  );

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

3.6.1 Lut generation code

The whole Lut was not compiled "by hand" obviously. The look-up table outputs were generated through the following python script by exploiting the computation concerning the LSB with 12 bits and 16 bits resolution made before.

```
import math

#Calculate lsb of x (16 bits) and f(x) (12 bits)
lsb_out = (1)/(2**15 - 1)
lsb_in = (11)/(2**11 - 1)
result = ""

for x in range(0, 2048):
  f_x = (1)/(1 + math.exp(-(x*lsb_in)))
  lut = round(f_x/lsb_out)

  #Generate lut entries for every x
  result += str(x) + " => " + str(lut) + ",\n"

print(result)

```

4 — Test Plan

5 — XILINX VIVADO Report

6 — Conclusion