UNIVERSITÀ DI PISA

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 **6**
- 2.1 Multiplication Circuit Architecture . . . 6
- 2.2 Adder Circuit Architecture . . . 9
- 2.3 Activation Function Circuit Architecture . . . 12

3 VHDL CODE **13**
- 3.1 Modules List . . . 13
- 3.2 Perceptron . . . 13
- 3.3 Parallel Multiplier . . . 15
- 3.4 Unsigned Parallel Multiplier . . . 16
- 3.5 Tree Adder . . . 19
  - 3.5.1 Ripple Carry Adder Pipelined . . . 19
- 3.6 LUT . . . 21
  - 3.6.1 Lut generation code . . . 22

4 Test Plan **23**
- 4.1 System Estimation Test . . . 23
  - 4.1.1 Estimation Test 1 . . . 23
  - 4.1.2 Estimation Test 2 . . . 24
  - 4.1.3 Estimation Test 3 . . . 25
- 4.2 System Aimed Test . . . 25

5 XILINX VIVADO Report **30**

6 Conclusion **31**

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.

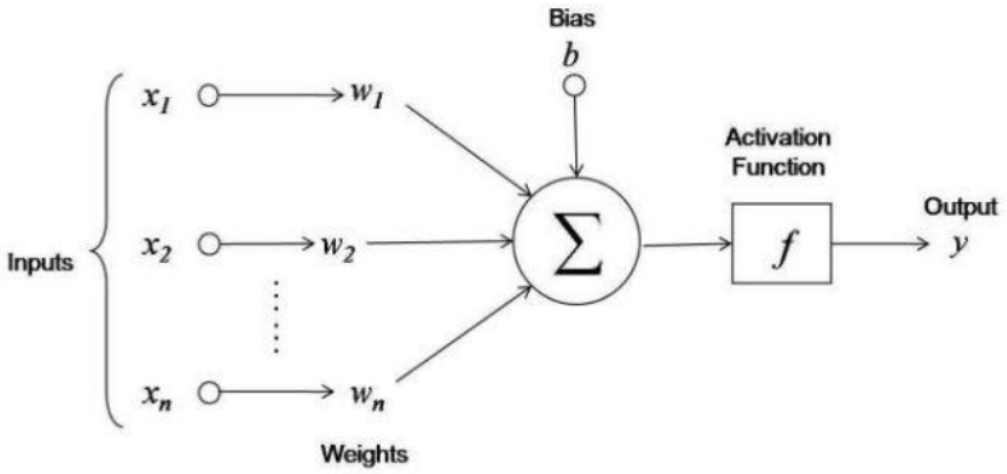


Figure 1: Perceptron Architecture

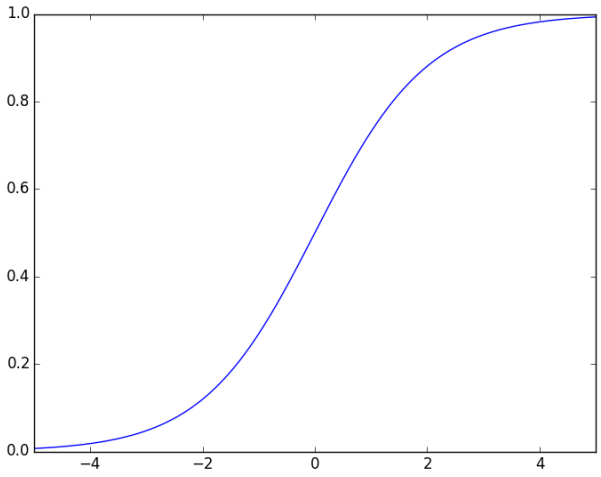
A **Perceptron** is a *binary classifier that maps his inputs to a specific output y = f(z), where f() is the* ***activation function*** *of the perceptron.* The inputs are real numbers and the input z of the activation function is obtained as:

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

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

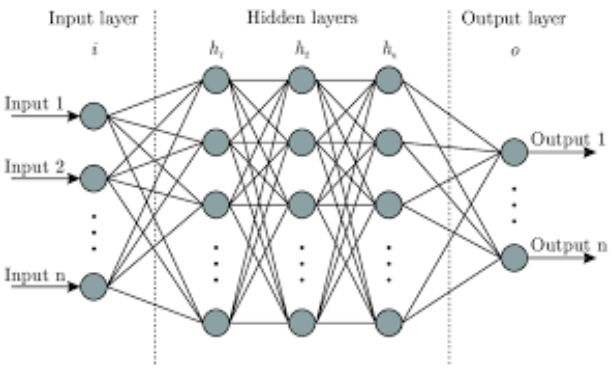


Where z is the result of the equation (1.1).

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 and so on and so forth.

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 (n_{w_i} represent the number of bits of w_i and the same for n_{b_i} and b_i). In order to implement the multiplication circuit will be implemented through a **Paraller Multiplier**, with some additional logic to handle the signed inputs.
- **Adder Circuit**: different choices 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 (and increasing the number of clock cycles though).
- **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. For further details see next Chapter.

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:

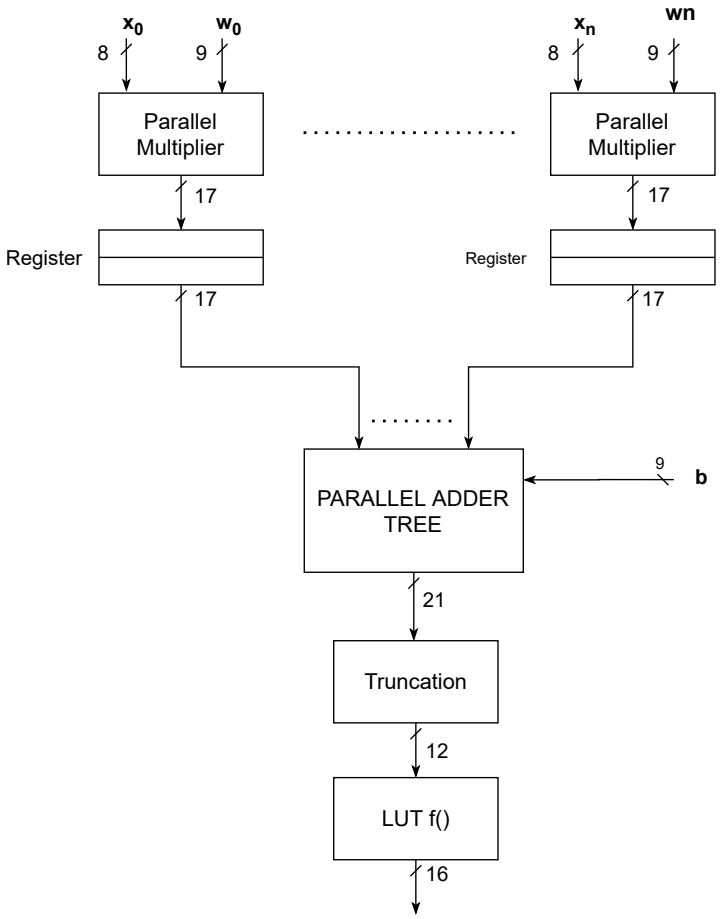


Figure 4: General Schema

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:

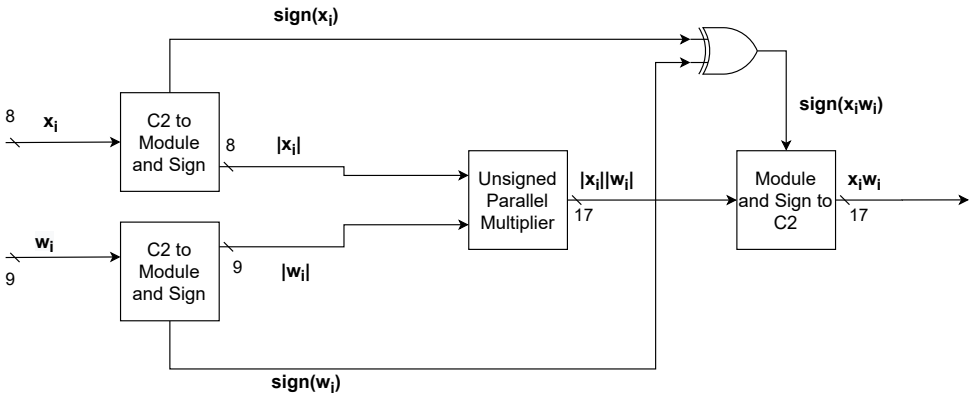


Figure 5: Parallel Multiplier Architecture

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:

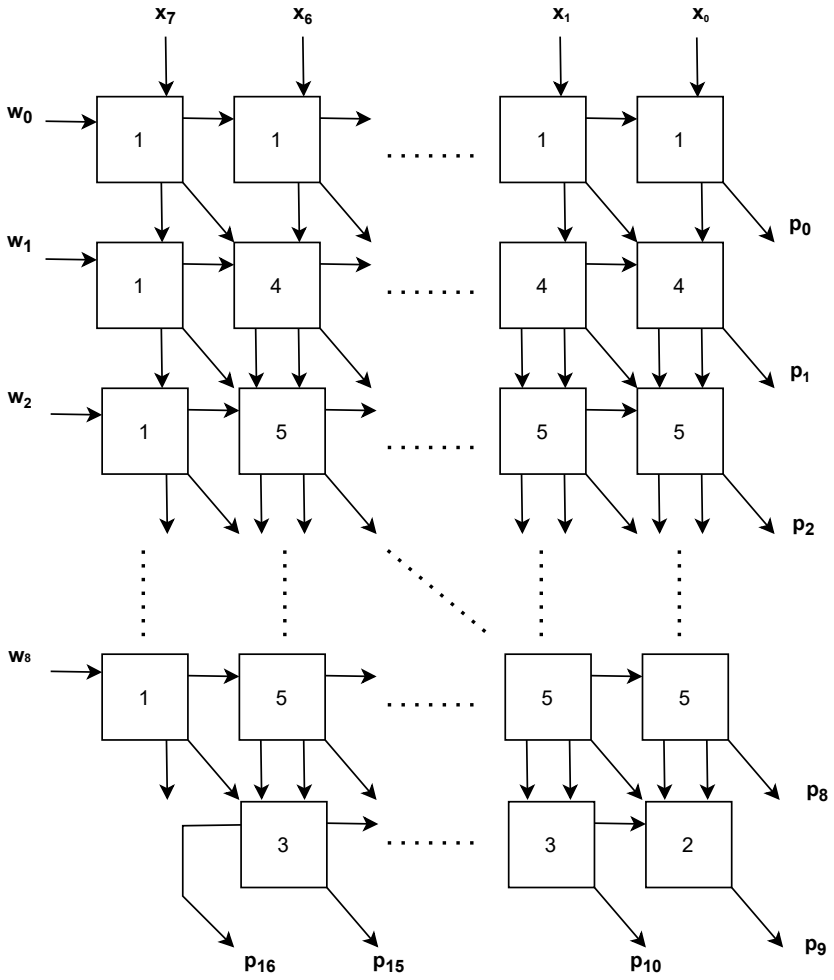


Figure 6: Unsigned Parallel Multiplier Architecture

Each logic block is translated with a related logic block:

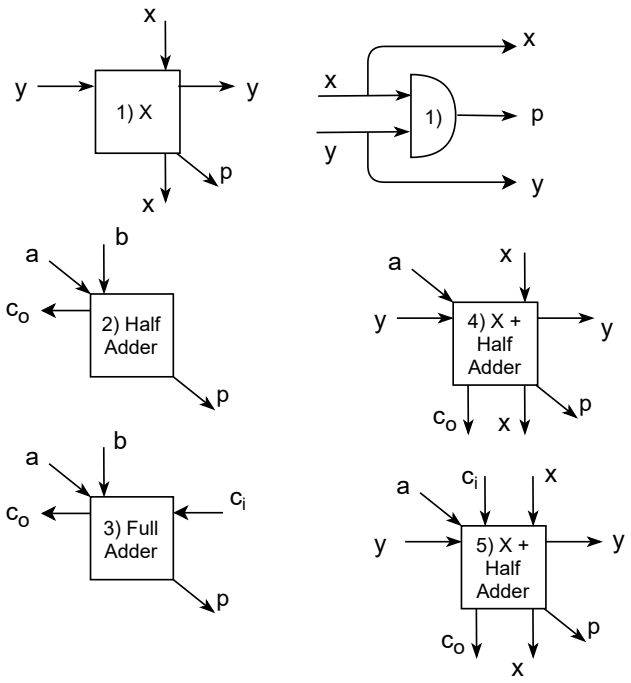


Figure 7: Unsigned Parallel Multiplier Architecture

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:

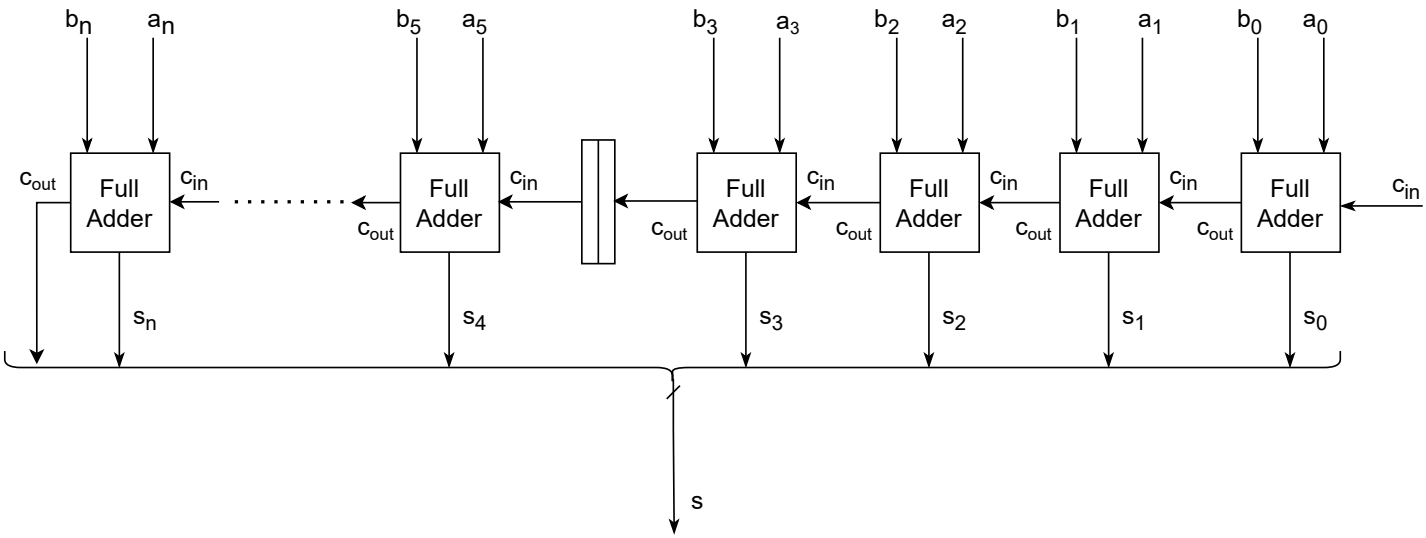


Figure 8: Parallel Adder Architecture

In order to obtain an output, after an input drive, there is a need to wait $\left\lfloor \frac{N}{N_{pipeline}} \right\rfloor$ clock cycles, where N represent the number of bits of a or b and $N_{pipeline}$ represent the maximum number of consecutive FA without a register in between.
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:

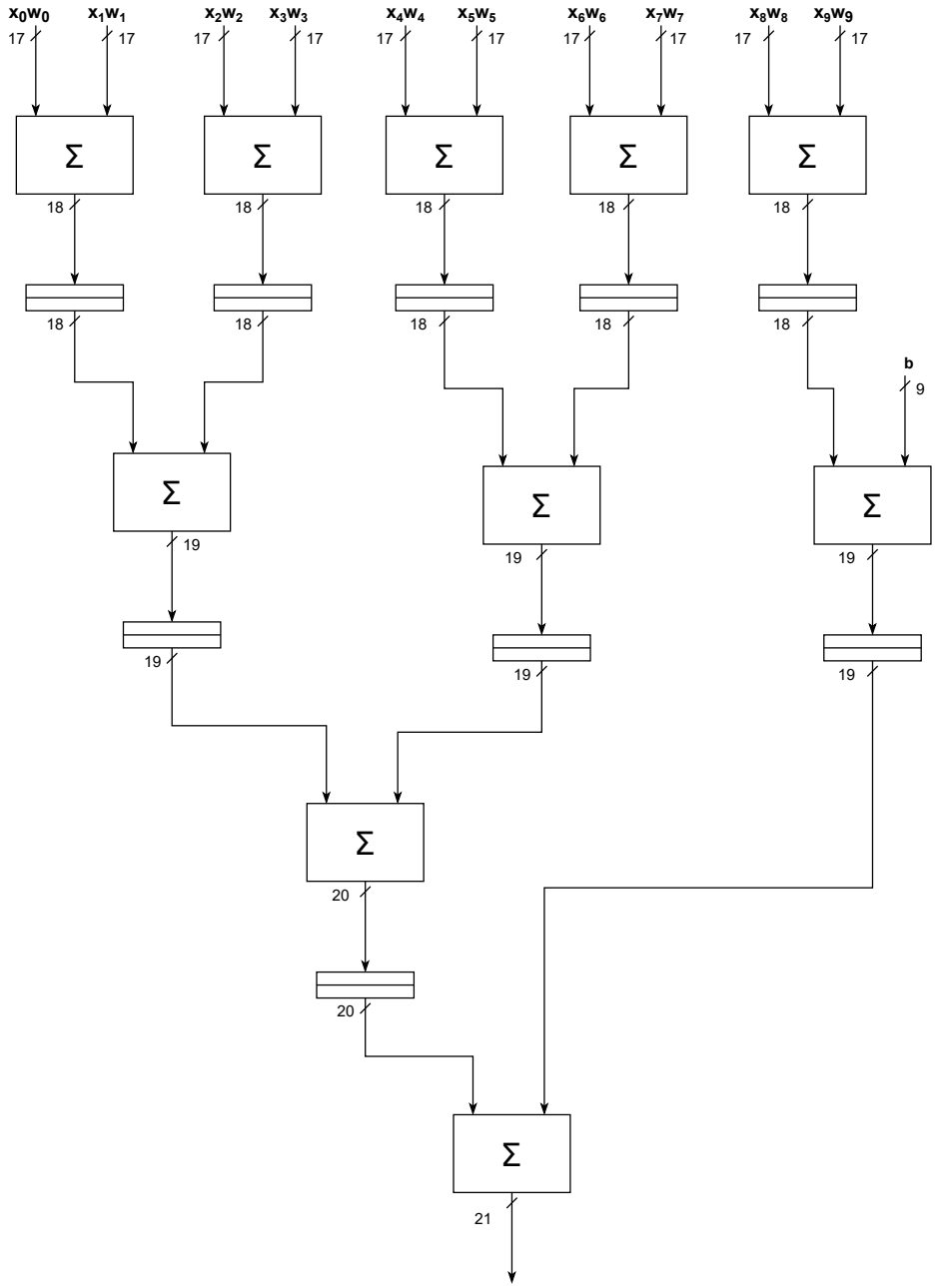


Figure 9: Parallel Multiplier Architecture

Note some extension or left shifts (i.e. for b) were not represented.
Some register has been put in between the sum to limit the critical path

impact on the performances and clock period limit. To obtain a good output after an input drive there is a need to sum to 3 (the maximum number of consecutive register in the previous architecture) each Ripple Carry Adder contribution in terms of number of clock as seen before.

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.

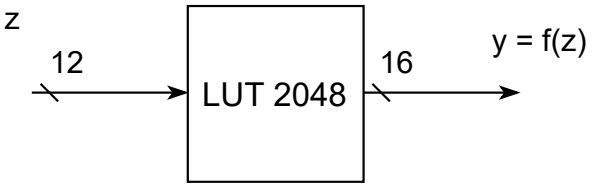


Figure 10: Look-Up Table Architecture

All things considered, by performing the calculation showed before, there is a need of 26 **clock cycles** to obtain a correct output after driving an input.

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.

```
1  entity Perceptron is
2  port(
3
4    -- x_1 to x_10 inputs of the perceptron with 8 bits
5    x_1: 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_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;
architecture rtl of Perceptron is
  ...
begin
  ...
  d_process: process(clk, rst)
  begin
    -- If z, the candidate input of the sigmoid function, is
   negative,
    -- then is passed his complement.
    if(z_in(20) = '1') then
      z_in_lut <= std_logic_vector(unsigned(not(z_in)) + 1);
    else
      z_in_lut <= z_in;
    end if;



    -- On the output side if the candidate input was negative
    -- the output is complemented with the highest possible
    -- number in the lut in order to mirror it
    if (z_in(20) = '1') then
      f_z <= std_logic_vector(32766 - unsigned(f_z_todo));
    else
      f_z <= std_logic_vector(unsigned(f_z_todo));
    end if;
  end process d_process;

end rtl;
```

In the rest of this modules are instantiated and linked the various submodules that made up the **Perceptron** module. At the bottom of the previous

code snippet is shown how the optimization of the LUT is made.

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.

```
1  entity Parallel_Multiplier is
2    generic (
3      Nbit_a : positive;
4      Nbit_b: positive
5    );
6    port(
7      a_p_signed: in std_logic_vector(Nbit_a - 1 downto 0);
8      b_p_signed: in std_logic_vector(Nbit_b - 1 downto 0);
9      -- The product will need Nbit_a + Nbit_b bits
10     p_signed: out std_logic_vector(Nbit_a + Nbit_b - 1
   downto 0)
11   );
12 end entity Parallel_Multiplier;
13
14 architecture rtl of Parallel_Multiplier is
15
16   -- Building blocks of the Parallel Multiplier
17   component Unsigned_Parallel_Multiplier
18     generic(
19       Nbit_a : positive;
20       Nbit_b : positive
21     );
22     port(
23       a_p: in  std_logic_vector(Nbit_a - 1 downto 0);
24       b_p : in  std_logic_vector(Nbit_b - 1 downto 0);
25       p   : out std_logic_vector(Nbit_a + Nbit_b - 1 downto
   0)
26     );
27   end component Unsigned_Parallel_Multiplier;
28
29
30   -- Unsigned component (will work for the unsigned
   parallel multiplier
31   signal p_unsigned: std_logic_vector(Nbit_a + Nbit_b - 1
   downto 0);
32   signal a_p_unsigned: std_logic_vector(Nbit_a - 1 downto
   0);
33   signal b_p_unsigned: std_logic_vector(Nbit_b - 1 downto
   0);
```

```
34
35      -- will carry the sign bit for the signed rapresentation
        of the inputs
36      signal a_sign: std_logic;
37      signal b_sign: std_logic;
38
39  begin
40
41      -- Compute the unsigned representation from the signed
        one
42      a_p_unsigned <= std_logic_vector(abs(signed(a_p_signed)))
        ;
43      b_p_unsigned <= std_logic_vector(abs(signed(b_p_signed)))
        ;
44
45      -- 2's complement rapresentation, the result sign uis
        computed through the xor op. between a and b
46      p_signed <= std_logic_vector(unsigned(not(p_unsigned)) +
        1) when (((a_sign xor b_sign) = '1')) else p_unsigned;
47
48      -- Getting of the sign from a and b (the MSB of the C2
        representation)
49      a_sign <= a_p_signed(Nbit_a - 1);
50      b_sign <= b_p_signed(Nbit_b - 1);
51
52      unsigned_parallel_mul: Unsigned_Parallel_Multiplier
53        generic map(
54          Nbit_a => Nbit_a,
55          Nbit_b => Nbit_b
56        )
57        port map(
58          a_p =>  a_p_unsigned,
59          b_p =>  b_p_unsigned,
60          p   => p_unsigned
61        );
62
63  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));

```

```
42
43      -- Computation of the various inputs of each HA and FA
44      d_process: process(a_p, b_p)
45      begin
46
47      for j in 1 to Nbit_b loop
48      a_multiplier(j - 1) <= (a_p(0) and b_p(Nbit_b - j));
49      end loop;
50
51      ...
52      end process d_process;
53
54
55      -- Architecture will follow schema of the Parallel
    Multiplier
56      -- Row index i
57      GEN_a: for i in 1 to Nbit_a generate
58        -- Column index j
59        GEN_b: for j in 1 to Nbit_b - 1 generate
60          FIRST_ROW: if i=1 generate
61            -- In the first Row only HA
62            LEFT: if j < Nbit_b -1 generate
63              ROW1_LEFT: HALF_ADDER
64                port map
65                (
66                  a    => a_multiplier(j - 1),
67                  b    => b_multiplier(j - 1),
68                  s    => sum_signal(j - 1),
69                  cout => carry_signal(j - 1)
70                );
71              end generate LEFT;
72            RIGHT: if j = Nbit_b - 1 generate
73              ROW1_RIGHT: HALF_ADDER
74                port map
75                (
76                  a    => a_multiplier(j - 1),
77                  b    => b_multiplier(j - 1),
78                  s    => p(1), -- Result bit
79                  cout => carry_signal(j - 1)
80                );
81            end generate RIGHT;
82          end generate FIRST_ROW;
83
84
85          INTERNAL_ROW: if i > 1 and i < Nbit_a generate
86            ...
87          end generate INTERNAL_ROW;
88
89
```

```
        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_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;

```

```
10      -- Will store the result of a_r+b_r
11      s_r: out std_logic_vector(Nbit-1 downto 0);
12      clk: in std_logic;
13      rst: in std_logic
14    );
15  end Ripple_Carry_Adder_Pipelined;
16
17  architecture rtl of Ripple_Carry_Adder_Pipelined is
18    -- Building blocks of the Ripple Carry Adder Pipelined
19    component FULL_ADDER
20      ...
21    end component FULL_ADDER;
22
23    -- Need of a register to obtain the pipelined version
24    component DFF
25    ...
26    component DFF;
27
28    -- Will propagate the carry signal among the whole
       architecture
29    signal carry_signal: std_logic_vector(Nbit-1 downto 1);
30
31    -- Will store the outputs signal of the registers
32    signal dff_signal: std_logic_vector(Nbit-1 downto 0) := (
       others => '0');
33
34  begin
35    -- Implemented in a structured way in a similar fashion as
       seen in the Lab lessions
36    GEN: for i in 1 to Nbit generate
37      FIRST: if i=1 generate
38      -- First FA
39        FFI: FULL_ADDER port map (a_r(0), b_r(0), cin_r, s_r(0)
       , carry_signal(1));
40      end generate FIRST;
41      INTERNAL: if i > 1 and i < Nbit generate
42        -- Need of Register detection
43        PIPE: if (i mod 3 = 0) generate
44          DFF_I: DFF
45            port map(
46            d_dff      => carry_signal(i-1),
47            clk_dff    => clk,
48            resetn_dff => rst,
49            q_dff      => dff_signal(i-1)
50          );
51          FFI: FULL_ADDER port map (a_r(i-1), b_r(i-1),
       dff_signal(i-1), s_r(i-1), carry_signal(i));
52        end generate PIPE;
53
```

```
54        -- No need of a register
55        NOT_PIPE: if (i mod 3 /= 0) generate
56          FFI: FULL_ADDER port map (a_r(i-1), b_r(i-1),
    carry_signal(i-1), s_r(i-1), carry_signal(i));
57        end generate NOT_PIPE;
58      end generate INTERNAL;
59
60      -- Implicit extension (the inputs have Nbit-2 bits, the
    output has Nbit-1 bits and there
61      -- are Nbit-1 FA so the last bit is replicated in order
    to make the extension in the
62      -- correct way in C2 representation)
63
64      LAST: if i=Nbit generate
65      FFI: FULL_ADDER port map (a_r(Nbit-2), b_r(Nbit-2),
    carry_signal(Nbit-1), s_r(Nbit-1), cout_r);
66      end generate LAST;
67    end generate GEN;
68  end rtl;
```

3.6 LUT

This module will store every possible output of the sigmoid function in the unsigned input range of $[0; +31]$ with 12 bits of precision, (even if the working range is $[-11; +11]$, but with the addition of some logic to make the lut optimization an unsigned representation was used), and output range of $[0; 1]$ with 16 bits of precision. In order to do so a quantization is needed: is unthinkable to define precisely a rational number with a finite number of bits. To determine the quantized quantity of the input and the output there is a need to calculate the weight of the LSB in that range. Through the **Reach the LSB** method seen during the Lab lectures, the LSB, with N bits of precision, can be calculated as:

$$LSB = \frac{\max x}{2^{N-1} - 1} \quad or \quad \frac{|\min x|}{2^{N-1}} \tag{3}$$

So, in out case:

$$LSB(in) = \frac{31}{2^{11} - 1} = 0.015144113 \tag{4}$$

$$LSB(out) = \frac{1}{2^{15} - 1} = 3.051850947e - 5 \tag{5}$$

The Look-Up table will store the values $round(f(x)/LSB/out)$ for x in $[0; 2047]$.
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;

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 = (31)/(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

In order to verify the correctness of the system the following tests were made:

1. **Unit Tests**: following the **bottom-up** strategy, each sub-module (Parallel Multiplier, Ripple Carry Adder Pipelined...), after completing the implementation, has a dedicated testbench in order to check the correctness of the single sub-module in isolation. Considering the fact that this test are **trivial** (just checking if the sum or the product of some numbers is correct) this will not be showed in this documentation.

2. **System Estimation Test**: in this phase, some testbenches were written with particular inputs. The aim of this test is only to check if the result will resemble the sigmoid curve by varying the inputs in time in an increasing way.

3. **System Aimed Test**: after checking that the system is *likely* correct by the latter test, through a python script will be made a test with different inputs in the range considered and check, with an additional testbench, if the outputs are equal.

4.1 System Estimation Test

Even before checking the correctness of the output of the system by a given input, three "estimation tests" were made. The aim of these tests is just to obtain a sigmoid curve by setting each x_i and w_i and varying the bias b in the range of $[-1; +1]$.

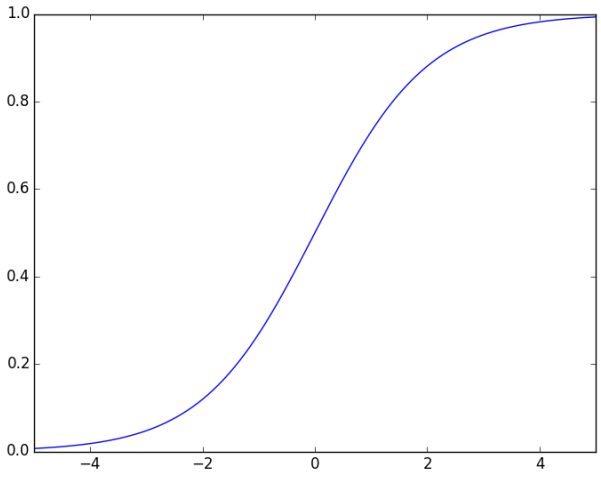
4.1.1 Estimation Test 1

In this case we have the following inputs:

$$x_1 \dots x_{10} = 0, w_1 \dots w_{10} = 0 \tag{6}$$

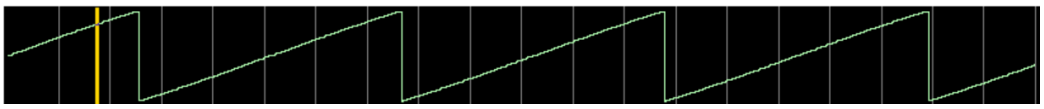
The bias b will vary in the whole range of $[-1, +1]$. Just to remind the sigmoid function curve, we expect to obtain a curve with an odd symmetry and a "linear" behaviour, due to the fact that we are considering the values near zero of the curve:

Figure 11: Sigmoid Function Plot



The output of the system is the following:

Figure 12: Output of the System 1



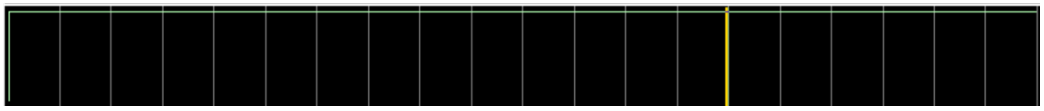
There are different replication of the system due to the fact that the bias b will "turn back" when he will get to his maximum. We can state that, by comparing the two figures, the **first estimation test is passed**.

4.1.2 Estimation Test 2

$$x_1 \dots x_{10} \approx 1, w_1 \dots w_{10} \approx 1 \tag{7}$$

The bias b will vary in the whole range of $[-1, +1]$. We expect to obtain a likely flat curve with some "high values" due to the fact that the summation of the ten product is 10 and the sigmoid at that value tends to 1.

Figure 13: Output of the System 2



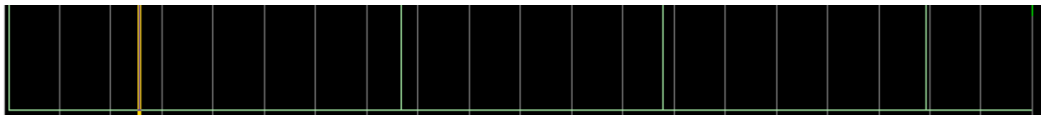
The output of the system is similar to a flat curve. We can state that, by comparing the system output with what we expected, the **second estimation test is passed**.

4.1.3 Estimation Test 3

$$x_1 \dots x_{10} \approx -1, w_1 \dots w_{10} \approx 1 \tag{8}$$

The bias b will vary in the whole range of $[-1, +1]$. We expect to obtain a likely flat curve with some "low values" due to the fact that the summation of the ten product is -10.

Figure 14: Output of the System 3



The curve is flat with low values but we can notice some *noise*. This is due to the fact that to obtain a proper output some clock cycles are needed: this will lead to obtain intermediate results that are not good. But we can state that, by comparing the system output with what we expected, the **third estimation test is passed**.

4.2 System Aimed Test

At this point will be carried out some tests with the same inputs using the **Perceptron** architecture realized at this point and a *python script* which will simulate the desired behaviour of the **Perceptron**. The latter is described through the following script:

```
import math

def get_outputs(i, x, w, b):
  print(f"################### TEST #{i} ###################")
  print(f"X: {x}")
  print(f"W: {w}")
  print(f"b: {b}")

  sum = summation(x, w, b)
  print(f"Sum result:\t\t\t\t {sum}")

  f_z = sigmoid_output(sum)
  print(f"Sigmoid output:\t\t\t {f_z}")

  #the sum with 12 bits
  sum_in_circuit = round(sum/lsb_in)
print(f"Sum value quantized:\t {sum_in_circuit}")

f_z_in_circuit = round(f_z/lsb_out)
```

```
  print(f"Output value quantized:\t {f_z_in_circuit}")
  def summation(x, w, b):
  sum = 0
  for i in range(0, 10):
  sum += x[i]*w[i]
  sum += b
  return sum

def sigmoid_output(s):
  res = (1)/(1 + math.exp(-s))
  return res

lsb_out = (1)/(2**15 - 1)
lsb_in = (32)/(2**11 - 1)
#Test #1
x = [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
w = [1,1,1,1,1,1,1,1,1,1]
b = 0
get_outputs(1, x, w, b)
...
#Other tests
...
```

By running the python script the following output has been displayed in the console:

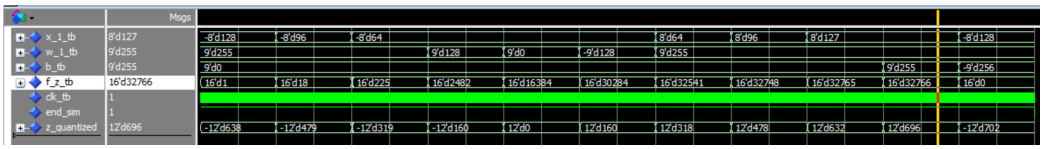
```
#################### TEST #1 ####################
X: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:         -10
Sigmoid output:       4.5397868702434395e-05
Sum value quantized:    -640
Output value quantized:  1
#################### TEST #2 ####################
X: [-0.75, -0.75, -0.75, -0.75, -0.75, -0.75, -0.75, -0.75,
    -0.75, -0.75]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:         -7.5
Sigmoid output:       0.0005527786369235996
Sum value quantized:    -480
Output value quantized:  18
#################### TEST #3 ####################
X: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:         -5.0
Sigmoid output:       0.0066928509242848554
```

```
Sum value quantized:   -320
Output value quantized:  219
################### TEST #4 ###################
X: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5]
W: [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
b: 0
Sum result:        -2.5
Sigmoid output:      0.07585818002124355
Sum value quantized:   -160
Output value quantized:  2486
################### TEST #5 ###################
X: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5]
W: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
b: 0
Sum result:        0.0
Sigmoid output:      0.5
Sum value quantized:   0
Output value quantized:  16384
################### TEST #6 ###################
X: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5]
W: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5,
    -0.5]
b: 0
Sum result:        2.5
Sigmoid output:      0.9241418199787566
Sum value quantized:   160
Output value quantized:  30281
################### TEST #7 ###################
X: [0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:        5.0
Sigmoid output:      0.9933071490757153
Sum value quantized:   320
Output value quantized:  32548
################### TEST #8 ###################
X: [0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75,
    0.75]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:        7.5
Sigmoid output:      0.9994472213630764
Sum value quantized:   480
Output value quantized:  32749
################### TEST #9 ###################
X: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 0
Sum result:         10
Sigmoid output:       0.9999546021312976
Sum value quantized:    640
Output value quantized:  32766
#################### TEST #10 ####################
X: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: 1
Sum result:         11
Sigmoid output:       0.999983298578152
Sum value quantized:    704
Output value quantized:  32766
#################### TEST #11 ####################
X: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
W: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
b: -1
Sum result:         -11
Sigmoid output:       1.670142184809518e-05
Sum value quantized:    -704
Output value quantized:  1
```

By running a new testbench with **likely** the same inputs the following results were displayed in **Modelsim**:

Figure 15: Output of the System 3



A comparison with the outputs can be seen in the following table:

Test	**Python Script**		**Modelsim**	
	$round\left(\frac{z}{LSB}\right)$	$round\left(\frac{f(z)}{LSB}\right)$	$round\left(\frac{z}{LSB}\right)$	$round\left(\frac{f(z)}{LSB}\right)$
Test #1	-640	1	-638 (+2)	1 (=)
Test #2	-480	18	-479 (+1)	18 (=)
Test #3	-320	219	-319 (+1)	225 (+5)
Test #4	-160	2486	-160 (=)	2482 (-4)
Test #5	0	16384	0 (=)	16384 (=)
Test #6	160	30281	160 (=)	30284 (+3)
Test #7	320	32548	318 (-2)	32541 (-7)
Test #8	480	32749	478 (-2)	32748 (-1)
Test #9	640	32766	632 (-8)	32765 (-1)
Test #10	704	32766	696 (-8)	32766 (=)
Test #11	-704	1	-702 (+2)	0 (-1)

As we can see in the latter table the outputs are **likely** the same, with some few differences that can be ignored. These differences can be easily explained: **Python's float** number will use **64 bits** instead of 12 or 16 as in our case. This difference will change the outputs, in fact, in our case, the number +1 ("01111111" in base of x_i with 8 bits), for example, can't be represented precisely with a finite number of bits: so, the higher number of bits are available, the higher precision will be granted. All things considered, we can state **the system has passed the System Aimed Test** and, for our purpose, **can be considered verified.**

5 — XILINX VIVADO Report

6 — Conclusion