University of Pisa

Master in Computer Engineering
Project for Electronic Systems

Perceptron

Author:
Jacopo Carlon
587412

Academic year
2022/2023

Contents

1 Introduction

1.1 Specification

It is requested to design a perceptron which takes $N_{IN} = 10$ inputs x_n represented with $b_x = 8$ bits. The network will carry out products between x_n and generic coefficients w_n, adding to the result a bias b . w_n and b are represented with $b_w = 9$ bits.
x_n, w_n and b shall be considered in range [-1, 1] using fixed point arithmetic.

This means that the inputs will have 2 bits for the integer part and 6 (for x_n) or 7 (for w_n and b) bits for the fractionary part.

The output y of the system shall be represented with $b_y = 16$ bits, truncating the least significant bits.

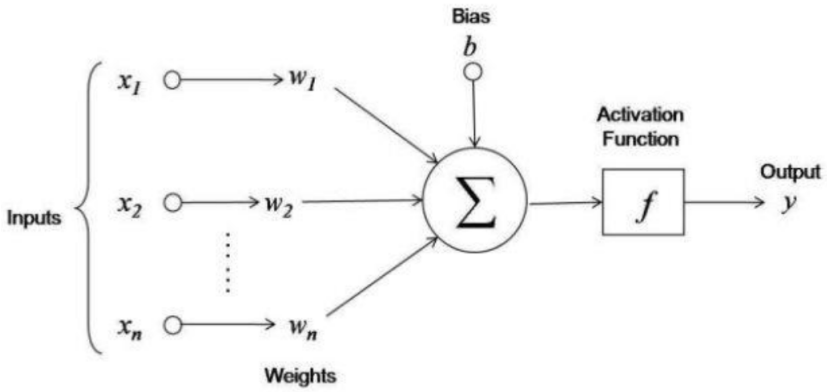


Figure 1: Perceptron

The activation function of the perceptron shall be the one represented in the next figure:

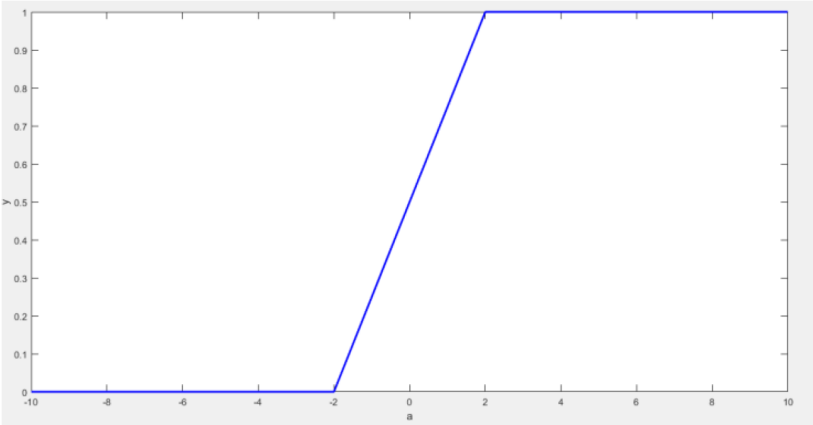


Figure 2: Activation Function

In the design used for Vivado implementation there will be registers catching all the inputs at $rising\_edge(clk)$, the operations will be evaluated in a combinatory manner, and the output will be assigned to the $output - register$ at the following $rising\_edge(clk)$.

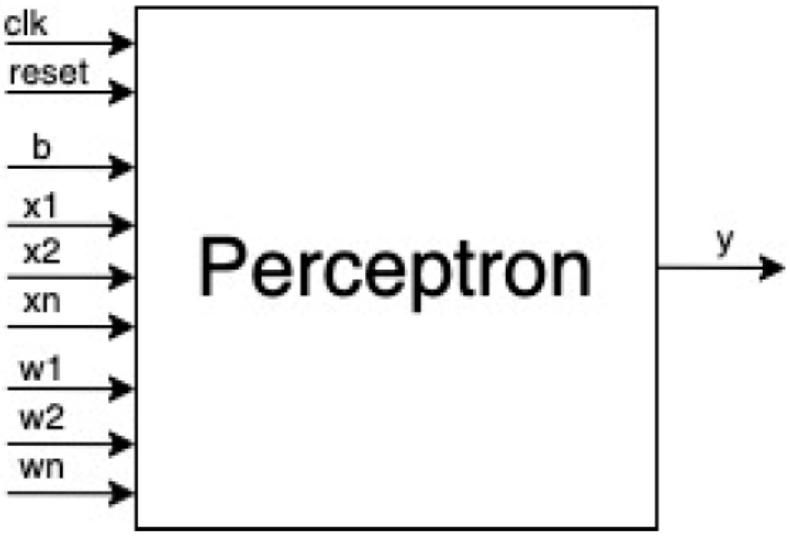


Figure 3: Perceptron Interface

1.2 Circuit Employments

A Perceptron, in Deep Learning, is a neural network link that contains computations to track features and use Artificial Intelligence in the input data. These artificial neurons are obtained using simple logic gates with binary outputs.
The perceptron is equivalent to an artificial neuron, which invokes the mathematical function and has node, input, weights, and output equivalent to the cell nucleus, dendrites, synapse, and axon, respectively, compared to a biological neuron.
In 1957 Frank Rosenblatt proposed a Perceptron learning rule based on the original MCP neuron.
A Perceptron is an algorithm for supervised learning of binary classifiers.
This algorithm enables neurons to learn and process elements in the training set one at a time.

This perceptron (and a combination of) can be used as an accelerator to improve neural network performances (e.g. during training phase).

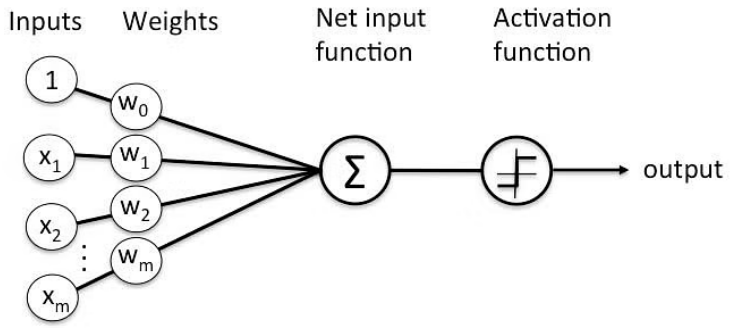


Figure 4: Perceptron Idea

2 Architecture

This implementation of the perceptron is obtained by having one main block process, used to assign default values to all the registers at the *asynchronous reset*, and to assign input values to the input-registers at $rising\_edge(clk)$. This main block also receives from the combinatory logic the elaborated y value in 16 bits, and assigns it to the output register at *rising_ edge(clk)* .

Other than the main logic, there is a big combinatory part used to implement the product of x_j, w_j for the j inputs, adding them together and adding the *bias b*.
From this prod-sum , the final output is generated by further combinatory logic, which implements the activation-function.

2.1 Sum-Product

Since we have inputs x_i in 8 bits (2 int & 6 fract) and weights w_i (2 int & 7 fract), and since they are to be considered *signed* (they range in [-1, 1]), each product result should be *signed* in 17 bits (4 int & 13 fract).
We also need to add -once- the bias (2 int & 7 fract), which would need to be expanded -with sign- to 18 bits at the least. The overall result "sp_res"of these 11 sums would be in 4 more bits to a total of 21 bits (8 int & 13 fract).

However, this size is not necessary! We know that all inputs are in [-1,1], therefore all products will be in [-1,1] and the sum will be in [-11, +11]. Since we do not want to lose accuracy, and since we need to keep in mind that these are *signed* values, we can represent sp_res in just 18 bits, 5 for the integer part and 13 for the fractionary part.

This is because we know that the [-11, +11] range in binary will be like [10101 , 01011]

In order to implement the operations so far described, we expand-with-sign all products to 18 bits (5 int & 13 fract) and sum them without any fear of overflowings.

2.2 Activation Function

The function is to be read as follows : if sp_res is less than -2, return 0;

- if sp_res is less than -2, return 0;
- if sp_res is greater than 2, return 1;
- else : map the values from [-2,2] to y [0,1] on a straight line.

It must be noted that, since the output will be in 16 bits, it will have 3 of integer part and 13 bits of fractionary part (we are required to truncate -during the division-). The output will therefore be in one of three forms : 00-0-.00...00, 00-1-.00...00 or 00-0-.XX...XX, which means the output will always be positive !

To implement the comparison we simply do a comparison with sign.
To implement the tilted-line part, however, we need some more considerations.

We could implement the function $y = x/2 + 1/2$, or the function $\mathbf{y = (x+2)/4}$. Considering that we enter the -else- block only if sp_res is in [-2,2], in order to get rid of the heavy *signed* property, I will firstly add 2, (represented in 5 int & 13 fract). By doing so I'll obtain positive number "sp_plus_two" (5 int & 13 fract) which will be in range [0, 4] and therefore will be in the form **00XXX.fract_13**.

The following operation of division by 4 could easily be implemented by a double shift to the right (padding the left with 0 since we have positive numbers is not a problem).
This would give us a value in the form $SS00X.XXfract\_11$ (5 int $ 13 fract) with S==0, and we would need to later reduce it to **00X.XXfract_11** (3 int $ 13 fract)for the output in 16 bits.

However, we can optimize this even more!
The final result can easily be achieved by simply extracting from sp_plus_two only 16 bits starting from the most significant bit (the leftmost one).

[Note that this elegant solution is so easily implementable because we know that the value is positive after adding 2!
This is the true reason for the above choice of function to implement.]

2.3 Other optimizations

After the first synthesis I noted that Vivado was implementing the sum of partial-products as a chain sum (cascading sum) which inevitably was causing delays in the generation of the final value.

In order to avoid such problems, I have manually rewritten the sum operations in the **structure of a balanced binary tree**.
This approach needed more signals (for the partial sum-prods), but reduced the sum portion of the combinatory part from 10 to **only 4 logic levels**, therefore allowing to decrease the minimum clock period (==increase the maximum clock frequency).

This also uses exactly the same number of adders, so no costs were added to this re-organization!

2.4 Timing and inputs

Since the time theoretically required to generate the output from given inputs is only due to the combinatory logic part, the timing is formalized as follows: - @$rising\_edge(clk)[t]$ inputs$[j]$ are saved, current output is stabilized and kept - output$[j]$ is elaborated - @$rising\_edge(clk)[t+1]$ inputs$[j+1]$ are saved, output$[j]$ is stabilized and kept

This means that, as long as all inputs are "as expected" (in range [-1,1] in 2-bits-int & 6|7-bits-fract), the Perceptron will be **sensible to inputs only at rising_edge(clk)[t]** , and the output of these inputs is **assured to be correct (exactly and only) at the following rising_edge(clk)[t+1]** .

Should inputs not be in the correct range, then the behaviour is N.D.:

Depending on the value (and sign!!!) of sp_res, the output could exchange 0 for 1, or mistakenly fall in the "else".

3 VHDL code

In this section it is shown and commented the VHDL code used to design the Perceptron. The test bench (TB) source is reported here with a brief description, however its logic is analysed in the section dedicated to the verification approach.

3.1 Entity List

Due to the extreme simple nature of the component, only a single VHDL entity was implemented:

- Perceptron

3.2 Perceptor : code

The following code shows a main process sensible to (asynchronous) reset and clock. The main process has a number of registers corresponding to the 21 inputs , and a register for the output y. The architecture also has a considerable combinational logic part, which works as described above, in order to calculate all the products and sums, and in order to implement the activation function of the Perceptron.

```
1
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4  use IEEE.numeric_std.all;
5  use IEEE.std_logic_unsigned.all;
6
7  entity Perceptron is
8      port(
9          clk           : in std_logic;
10         resetn        : in std_logic; -- active low
11
12         -- input values xi 8 bit, fpv, in [-1, 1]
13         x0:      in  std_logic_vector(8 - 1 downto 0);
14         x1:      in  std_logic_vector(8 - 1 downto 0);
15         x2:      in  std_logic_vector(8 - 1 downto 0);
16         x3:      in  std_logic_vector(8 - 1 downto 0);
17         x4:      in  std_logic_vector(8 - 1 downto 0);
18         x5:      in  std_logic_vector(8 - 1 downto 0);
19         x6:      in  std_logic_vector(8 - 1 downto 0);
20         x7:      in  std_logic_vector(8 - 1 downto 0);
21         x8:      in  std_logic_vector(8 - 1 downto 0);
22         x9:      in  std_logic_vector(8 - 1 downto 0);
```

```
        -- input weights wi 9 bit, fpv, [-1, 1]
        w0:     in  std_logic_vector(9 - 1 downto 0);
        w1:     in  std_logic_vector(9 - 1 downto 0);
        w2:     in  std_logic_vector(9 - 1 downto 0);
        w3:     in  std_logic_vector(9 - 1 downto 0);
        w4:     in  std_logic_vector(9 - 1 downto 0);
        w5:     in  std_logic_vector(9 - 1 downto 0);
        w6:     in  std_logic_vector(9 - 1 downto 0);
        w7:     in  std_logic_vector(9 - 1 downto 0);
        w8:     in  std_logic_vector(9 - 1 downto 0);
        w9:     in  std_logic_vector(9 - 1 downto 0);
        -- input bias b, 9 bit, fpv, [-1, 1]
        b:      in  std_logic_vector(9 - 1 downto 0);
        -- output y, 16 bit, fpv, [-1, 1]
        y :     out std_logic_vector(16 - 1 downto 0)
        --  ;
    );
end Perceptron;

--  /**/ /**/ /**/
architecture J_perceptron of Perceptron is
    -- "registers" input
        signal reg_x0       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x1       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x2       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x3       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x4       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x5       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x6       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x7       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x8       :   std_logic_vector(8 - 1 downto 0);
        signal reg_x9       :   std_logic_vector(8 - 1 downto 0);
        signal reg_w0       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w1       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w2       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w3       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w4       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w5       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w6       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w7       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w8       :   std_logic_vector(9 - 1 downto 0);
        signal reg_w9       :   std_logic_vector(9 - 1 downto 0);
        signal reg_bias     :   std_logic_vector(9 - 1 downto 0);
    -- "register"  output
        -- s   reg out :
        signal reg_y        :   std_logic_vector(16 - 1 downto 0);
```

```
69
70      -- sum-prod of 2int + 6|7 fract -> 17 = 4 int 13 fract in [-1,1]
71      -- sum 11 of these -> 21 = 8 int 13 fract in [-11,11]
72      -- ->>> 5 int 13 fract suffice to avoid OF, -> all in 18 bits 5
    int 13 fract
73          signal esp_bias     :   std_logic_vector(18 - 1 downto 0);
74          signal prod_0       :   std_logic_vector(18 - 1 downto 0);
75          signal prod_1       :   std_logic_vector(18 - 1 downto 0);
76          signal prod_2       :   std_logic_vector(18 - 1 downto 0);
77          signal prod_3       :   std_logic_vector(18 - 1 downto 0);
78          signal prod_4       :   std_logic_vector(18 - 1 downto 0);
79          signal prod_5       :   std_logic_vector(18 - 1 downto 0);
80          signal prod_6       :   std_logic_vector(18 - 1 downto 0);
81          signal prod_7       :   std_logic_vector(18 - 1 downto 0);
82          signal prod_8       :   std_logic_vector(18 - 1 downto 0);
83          signal prod_9       :   std_logic_vector(18 - 1 downto 0);
84
85          signal sum_0to1     :   std_logic_vector(18 - 1 downto 0);
    -- group 2
86          signal sum_2to3     :   std_logic_vector(18 - 1 downto 0);
    -- group 2
87          signal sum_4to5     :   std_logic_vector(18 - 1 downto 0);
    -- group 2
88          signal sum_6to7     :   std_logic_vector(18 - 1 downto 0);
    -- group 2
89          signal sum_8to9     :   std_logic_vector(18 - 1 downto 0);
    -- group 2
90          signal sum_0to3     :   std_logic_vector(18 - 1 downto 0);
    -- group 4
91          signal sum_4to7     :   std_logic_vector(18 - 1 downto 0);
    -- group 4
92          signal sum_8to9_bs  :   std_logic_vector(18 - 1 downto 0);
    -- group 3
93          signal sum_0to7     :   std_logic_vector(18 - 1 downto 0);
    -- group 8
94
95          -- sum_product_result   -- [ "-11" , "+11" ], 5 int 13 fract
96          signal sp_res:          std_logic_vector(18 - 1 downto 0);
97          signal sp_plus_two:     std_logic_vector(18 - 1 downto 0);
98          signal cut_sp_plus_two: std_logic_vector(16 - 1 downto 0);
99
100         -- filter output _y_    -- [ "-1" , "+1" ], 5 int 13 fract
101         signal value_to_out:    std_logic_vector(16 - 1 downto 0);
102         signal af_out:          std_logic_vector(16 - 1 downto 0);
103
104     -- end signals
```

```
105
106     -- begin constants
107         --
      "0123456789ABCDEF01"2345678"
108         -- 18 bits : 5 int 13 fract :
109         constant DUE_POS_18bit   : std_logic_vector(18 - 1 downto 0)
    := "000100000000000000" ;
110         constant DUE_NEG_18bit   : std_logic_vector(18 - 1 downto 0)
    := "111100000000000000" ;
111         constant UN_MEZ_18bit    : std_logic_vector(18 - 1 downto 0)
    := "000001000000000000" ;
112         -- 16 bits : 3 int 13 fract
113         constant ZERO_16b        : std_logic_vector(16 - 1 downto 0)
    := ( others => '0')      ; --   0
114         constant PUNO_16b        : std_logic_vector(16 - 1 downto 0)
    := ("0010000000000000") ; --  +1
115         --
       "0123456789ABCDEF"012345678"
116     -- end constants
117
118     -- architecture description [begin, end)
119     -- architecture begin
120     begin --J_perceptron
121         mainBlock: process  (    clk,
122                                  resetn
123                             )
124         begin
125             -- handle @resetn==0
126             if ( resetn = '0' ) then
127                 -- all inputs are considered 0
128                 reg_x0      <= (others => '0') ;
129                 reg_x1      <= (others => '0') ;
130                 reg_x2      <= (others => '0') ;
131                 reg_x3      <= (others => '0') ;
132                 reg_x4      <= (others => '0') ;
133                 reg_x5      <= (others => '0') ;
134                 reg_x6      <= (others => '0') ;
135                 reg_x7      <= (others => '0') ;
136                 reg_x8      <= (others => '0') ;
137                 reg_x9      <= (others => '0') ;
138                 reg_w0      <= (others => '0') ;
139                 reg_w1      <= (others => '0') ;
140                 reg_w2      <= (others => '0') ;
141                 reg_w3      <= (others => '0') ;
142                 reg_w4      <= (others => '0') ;
143                 reg_w5      <= (others => '0') ;
```

```
144                 reg_w6      <= (others => '0') ;
145                 reg_w7      <= (others => '0') ;
146                 reg_w8      <= (others => '0') ;
147                 reg_w9      <= (others => '0') ;
148                 -- bias
149                 reg_bias    <= (others => '0') ;
150                 -- s   reg uscita :
151                 -- output default 0
152                 reg_y       <= (others => '0') ;
153             elsif ( rising_edge(clk) ) then
154                 -- acquire input (8 or 9 bit)
155                 reg_x0      <=  ( x0 )      ;
156                 reg_x1      <=  ( x1 )      ;
157                 reg_x2      <=  ( x2 )      ;
158                 reg_x3      <=  ( x3 )      ;
159                 reg_x4      <=  ( x4 )      ;
160                 reg_x5      <=  ( x5 )      ;
161                 reg_x6      <=  ( x6 )      ;
162                 reg_x7      <=  ( x7 )      ;
163                 reg_x8      <=  ( x8 )      ;
164                 reg_x9      <=  ( x9 )      ;
165                 reg_w0      <=  ( w0 )      ;
166                 reg_w1      <=  ( w1 )      ;
167                 reg_w2      <=  ( w2 )      ;
168                 reg_w3      <=  ( w3 )      ;
169                 reg_w4      <=  ( w4 )      ;
170                 reg_w5      <=  ( w5 )      ;
171                 reg_w6      <=  ( w6 )      ;
172                 reg_w7      <=  ( w7 )      ;
173                 reg_w8      <=  ( w8 )      ;
174                 reg_w9      <=  ( w9 )      ;
175                 reg_bias    <=  ( b  )      ;
176                 -- s   reg uscita :
177                 reg_y       <=  af_out      ;
178             end if;
179         end process mainBlock;
180
181         -- s   reg uscita :
182         y <= reg_y ;
183
184         prod_0      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x0)) * to_integer(signed(reg_w0)) ) , 18 ) ) ;
185         prod_1      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x1)) * to_integer(signed(reg_w1)) ) , 18 ) ) ;
186         prod_2      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x2)) * to_integer(signed(reg_w2)) ) , 18 ) ) ;
```

```
187         prod_3      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x3)) * to_integer(signed(reg_w3)) ) , 18 ) ) ;
188         prod_4      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x4)) * to_integer(signed(reg_w4)) ) , 18 ) ) ;
189         prod_5      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x5)) * to_integer(signed(reg_w5)) ) , 18 ) ) ;
190         prod_6      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x6)) * to_integer(signed(reg_w6)) ) , 18 ) ) ;
191         prod_7      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x7)) * to_integer(signed(reg_w7)) ) , 18 ) ) ;
192         prod_8      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x8)) * to_integer(signed(reg_w8)) ) , 18 ) ) ;
193         prod_9      <= std_logic_vector( to_signed ( ( to_integer(
    signed(reg_x9)) * to_integer(signed(reg_w9)) ) , 18 ) ) ;
194
195         -- reg_bias is signed 9 bit, signed expand to 18 bit
196         esp_bias    <= ( reg_bias(9-1) & reg_bias(9-1) & reg_bias
    (9-1) & reg_bias(9-1) & reg_bias(9-1) & reg_bias(9-1) & reg_bias
    (9-1) & reg_bias(9-1) & reg_bias(9-1) & reg_bias );
197
198         -- from considerations above, 5 int 13 fract enough to not
    worry for OF,
199         -- sum in BALANCED BINARY TREE
200         sum_0to1        <= ( prod_0 + prod_1 ) ;     -- group 2
201         sum_2to3        <= ( prod_2 + prod_3 ) ;     -- group 2
202         sum_4to5        <= ( prod_4 + prod_5 ) ;     -- group 2
203         sum_6to7        <= ( prod_6 + prod_7 ) ;     -- group 2
204         sum_8to9        <= ( prod_8 + prod_9 ) ;     -- group 2
205
206         sum_0to3        <= ( sum_0to1 + sum_2to3 ) ; -- group 4
207         sum_4to7        <= ( sum_4to5 + sum_6to7 ) ; -- group 4
208         sum_8to9_bs     <= ( sum_8to9 + esp_bias ) ; -- group 3
209
210         sum_0to7        <= ( sum_0to3 + sum_4to7 ) ; -- group 8
211
212         sp_res  <= ( sum_0to7 + sum_8to9_bs ) ;
213         --  non balanced sum (sequential):
214         --  sp_res <= ( prod_0 + prod_1 + prod_2 + prod_3 + prod_4 +
     prod_5 + prod_6 + prod_7 + prod_8 + prod_9 + esp_bias );
215
216         -- !!! OUT_DECISOR !!!
217         -- let sp_res = x
218         -- _ if x > 2, 1
219         -- _ if x <2, 0
220         -- _ else : [ (x+2)/4 ]  ==  [ (x/2) + 1/2 ]
221         -- from given, note : x+2 GE0 && x+2 AE0
```

```
222         -- sp_plus_two :         00XXX.fract_13,
223         -- after shift :        SS00X.XXfract_11
224         -- reduce to 16 bit :   00X.XXfract_11
225         -- i will just cut after addition and all is good
226 
227         sp_plus_two <= sp_res + DUE_POS_18bit;
228         cut_sp_plus_two <= ( sp_plus_two((18-1) downto 2 ) );
229 
230         -- no reg uscita :
231         --  y <=    PUNO_16b when ( signed(sp_res) > signed(
    DUE_POS_18bit) ) else
232         --          ZERO_16b when ( signed(sp_res) < signed(
    DUE_NEG_18bit) ) else
233         --          cut_sp_plus_two ;
234 
235         -- s   reg uscita :
236         af_out <=   PUNO_16b when ( signed(sp_res) > signed(
    DUE_POS_18bit) ) else
237                     ZERO_16b when ( signed(sp_res) < signed(
    DUE_NEG_18bit) ) else
238                     cut_sp_plus_two ;
239 
240     -- (end architecture J_perceptron)
241     end J_perceptron;
```

3.3 Test Bench : code

The test bench (TB) structure will be better described in the next section; however, it is important to notice that it has been organized in order to behave like ordinary procedural code. The TB initializes an instance of the Perceptron, then monitors its output to verify that it behaves consistently with the specifications, adjourning the "fail" signal if any problem occurs.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  use IEEE.std_logic_unsigned.all;
5 
6  entity Perceptron_tb is
7  end Perceptron_tb ;
8 
9  architecture tb of Perceptron_tb is
10     constant CLK_PERIOD : time := 400 ns;
11     constant THREE_QUARTERS_PERIOD : time := 300 ns;
12 
```

```
13     constant ZERO_15b     : std_logic_vector(15 - 1 downto 0)   := ( others => '0') ;-- 0
14     constant ZERO_6b      : std_logic_vector(6 - 1 downto 0)    := ( others => '0') ;-- 0    -- per 7f
15     constant ZERO_7b      : std_logic_vector(7 - 1 downto 0)    := ( others => '0') ;-- 0    -- per 6f
16     constant UNO_15b      : std_logic_vector(15 - 1 downto 0)   := ( others => '1') ;-- 1
17     constant UNO_6b       : std_logic_vector(6 - 1 downto 0)    := ( others => '1') ;-- 1    -- per 7f
18     constant UNO_7b       : std_logic_vector(7 - 1 downto 0)    := ( others => '1') ;-- 1    -- per 6f
19
20     constant PUNO_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_6b & "010000000" ) ; -- +1
21     constant PHLF_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_6b & "001000000" ) ; -- +0.5
22     constant PQRT_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_6b & "000100000" ) ; -- +0.25
23     constant NQRT_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_6b  & "111100000" ) ; -- -0.25
24     constant NHLF_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_6b  & "111000000" ) ; -- -0.5
25     constant NUNO_30b_7f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_6b  & "110000000" ) ; -- -1
26
27     constant PUNO_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_7b & "01000000" ) ; -- +1
28     constant PHLF_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_7b & "00100000" ) ; -- +0.5
29     constant PQRT_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( ZERO_15b & ZERO_7b & "00010000" ) ; -- +0.25
30     constant NQRT_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_7b  & "11110000" ) ; -- -0.25
31     constant NHLF_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_7b  & "11100000" ) ; -- -0.5
32     constant NUNO_30b_6f  : std_logic_vector(30 - 1 downto 0)   := ( UNO_15b  & UNO_7b  & "11000000" ) ; -- -1
33
34     constant ZERO_30b     : std_logic_vector(30 - 1 downto 0)   := ( others => '0' ) ;-- 0
35
36     -- y ha 13 fract e 3 interi, ed    in [0, 1]
37     -- il +1 sarebbe 001_000_000_000_000_0
38     constant ZERO_16b     : std_logic_vector(16 - 1 downto 0)   := ( others => '0') ;-- 0
```

```
39      constant PUNO_16b     : std_logic_vector(16 - 1 downto 0)   := (
    "0010000000000000") ; -- +1
40      constant PHLF_16b     : std_logic_vector(16 - 1 downto 0)   := (
    "0001000000000000") ;-- 0
41
42
43      component Perceptron
44          port(
45              clk     : in std_logic;
46              resetn  : in std_logic; -- active low
47              -- input xi a 8 bit, in virgola fissa, tra -1 e 1
48              x0      : in  std_logic_vector(8 - 1 downto 0);
49              x1      : in  std_logic_vector(8 - 1 downto 0);
50              x2      : in  std_logic_vector(8 - 1 downto 0);
51              x3      : in  std_logic_vector(8 - 1 downto 0);
52              x4      : in  std_logic_vector(8 - 1 downto 0);
53              x5      : in  std_logic_vector(8 - 1 downto 0);
54              x6      : in  std_logic_vector(8 - 1 downto 0);
55              x7      : in  std_logic_vector(8 - 1 downto 0);
56              x8      : in  std_logic_vector(8 - 1 downto 0);
57              x9      : in  std_logic_vector(8 - 1 downto 0);
58              -- input weights wi a 9 bit, in virgola fissa, tra -1 e
    1
59              w0      : in  std_logic_vector(9 - 1 downto 0);
60              w1      : in  std_logic_vector(9 - 1 downto 0);
61              w2      : in  std_logic_vector(9 - 1 downto 0);
62              w3      : in  std_logic_vector(9 - 1 downto 0);
63              w4      : in  std_logic_vector(9 - 1 downto 0);
64              w5      : in  std_logic_vector(9 - 1 downto 0);
65              w6      : in  std_logic_vector(9 - 1 downto 0);
66              w7      : in  std_logic_vector(9 - 1 downto 0);
67              w8      : in  std_logic_vector(9 - 1 downto 0);
68              w9      : in  std_logic_vector(9 - 1 downto 0);
69              -- input bias a 9 bit, in virgola fissa, tra -1 e 1
70              b       : in  std_logic_vector(9 - 1 downto 0);
71              -- output [-1, 1]
72              y       : out std_logic_vector(16 - 1 downto 0)
73          );
74      end component;
75
76      -- control signals
77      -- timings signals
78      signal clk_tb   : std_logic     := '0';
79      signal resetn_tb: std_logic     := '0';
80      -- test estimators
81      signal success  : std_logic     := '0';
```

```
82      signal fail     : std_logic     := '0';
83      signal pass_input: std_logic    := '0';
84      signal testing  : boolean := true;
85      -- mi preparo tutti i registri a 0
86      -- input vals
87      signal tb_reg_x0    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
88      signal tb_reg_x1    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
89      signal tb_reg_x2    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
90      signal tb_reg_x3    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
91      signal tb_reg_x4    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
92      signal tb_reg_x5    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
93      signal tb_reg_x6    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
94      signal tb_reg_x7    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
95      signal tb_reg_x8    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
96      signal tb_reg_x9    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
97      -- input weights
98      signal tb_reg_w0    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
99      signal tb_reg_w1    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
100     signal tb_reg_w2    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
101     signal tb_reg_w3    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
102     signal tb_reg_w4    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
103     signal tb_reg_w5    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
104     signal tb_reg_w6    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
105     signal tb_reg_w7    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
106     signal tb_reg_w8    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
107     signal tb_reg_w9    :   std_logic_vector(30 - 1 downto 0)   := (
        others => '0' ) ;
```

```
108     -- input system bias
109     signal tb_reg_bias  :   std_logic_vector(30 - 1 downto 0)   := (
    others => '0' ) ;
110     -- output filtered result
111     signal tbs_y_rcv    :   std_logic_vector(16 - 1 downto 0)   := (
    others => '0' ) ;
112     -- end signals
113 
114     -- begin testing
115     begin
116         clk_tb <= not clk_tb after CLK_PERIOD / 2 when testing else
    '0';
117         -- attach device to test (device under test)
118         dut : Perceptron
119         port map(
120             clk         =>  clk_tb,
121             resetn      =>  resetn_tb,
122             x0          =>  tb_reg_x0( 8-1 downto 0 ) ,
123             x1          =>  tb_reg_x1( 8-1 downto 0 ) ,
124             x2          =>  tb_reg_x2( 8-1 downto 0 ) ,
125             x3          =>  tb_reg_x3( 8-1 downto 0 ) ,
126             x4          =>  tb_reg_x4( 8-1 downto 0 ) ,
127             x5          =>  tb_reg_x5( 8-1 downto 0 ) ,
128             x6          =>  tb_reg_x6( 8-1 downto 0 ) ,
129             x7          =>  tb_reg_x7( 8-1 downto 0 ) ,
130             x8          =>  tb_reg_x8( 8-1 downto 0 ) ,
131             x9          =>  tb_reg_x9( 8-1 downto 0 ) ,
132             w0          =>  tb_reg_w0( 9-1 downto 0 ) ,
133             w1          =>  tb_reg_w1( 9-1 downto 0 ) ,
134             w2          =>  tb_reg_w2( 9-1 downto 0 ) ,
135             w3          =>  tb_reg_w3( 9-1 downto 0 ) ,
136             w4          =>  tb_reg_w4( 9-1 downto 0 ) ,
137             w5          =>  tb_reg_w5( 9-1 downto 0 ) ,
138             w6          =>  tb_reg_w6( 9-1 downto 0 ) ,
139             w7          =>  tb_reg_w7( 9-1 downto 0 ) ,
140             w8          =>  tb_reg_w8( 9-1 downto 0 ) ,
141             w9          =>  tb_reg_w9( 9-1 downto 0 ) ,
142             b           =>  tb_reg_bias( 9-1 downto 0 ) ,
143             y           =>  tbs_y_rcv
144         );
145 
146         -- testing "program"
147         TB_PROC: process -- (clk_tb)
148         begin
149             if testing then
150                 wait until rising_edge(clk_tb);
```

```
151                 resetn_tb <= '0' ;
152                 wait until rising_edge(clk_tb);
153                 -- end reset (resent begins at 0 in tb, now i put it
    at 1)
154                 resetn_tb <= '1' ;
155             -- give initial data: weights not 0, x all 0
156                 wait until rising_edge(clk_tb);
157                 -- x ha 6 bit fract e 2 interi, metto tutto a 0 _ il
    reg    a 30 bit
158                 tb_reg_x0 <= ZERO_30b ;
159                 tb_reg_x1 <= ZERO_30b ;
160                 tb_reg_x2 <= ZERO_30b ;
161                 tb_reg_x3 <= ZERO_30b ;
162                 tb_reg_x4 <= ZERO_30b ;
163                 tb_reg_x5 <= ZERO_30b ;
164                 tb_reg_x6 <= ZERO_30b ;
165                 tb_reg_x7 <= ZERO_30b ;
166                 tb_reg_x8 <= ZERO_30b ;
167                 tb_reg_x9 <= ZERO_30b ;
168                 -- w e bias hanno 7 bit fract e 2 interi _ il reg
    a 30 bit
169                 tb_reg_w0 <=  PUNO_30b_7f ;   -- +1
170                 tb_reg_w1 <=  PUNO_30b_7f ;   -- +1
171                 tb_reg_w2 <=  PUNO_30b_7f ;   -- +1
172                 tb_reg_w3 <=  PHLF_30b_7f ;   -- +0.5
173                 tb_reg_w4 <=  PQRT_30b_7f ;   -- +0.25
174                 tb_reg_w5 <=  NUNO_30b_7f ;   -- -1
175                 tb_reg_w6 <=  NUNO_30b_7f ;   -- -1
176                 tb_reg_w7 <=  NUNO_30b_7f ;   -- -1
177                 tb_reg_w8 <=  NHLF_30b_7f ;   -- -0.5
178                 tb_reg_w9 <=  NQRT_30b_7f ;   -- -0.25
179                 -- bias come sopra
180                 tb_reg_bias <= ZERO_30b ;     -- +0
181                 pass_input <= '1';
182             --  tutti gli input sono 0, output dovrebbe essere 1/2;
183                 wait until rising_edge(clk_tb);
184                 pass_input <= '0';
185                 wait until rising_edge(clk_tb);
186                 wait for 15 ns;
187                 if tbs_y_rcv /= PHLF_16b then
188                     fail<= '1' ;
189                 end if;
190         -- daccapino
191             -- test x values:
192             -- pos -> 1
193                 wait until rising_edge(clk_tb);
```

```
194                 pass_input <= '1';
195                 wait for THREE_QUARTERS_PERIOD ;
196                 -- cambio solo un paio di registri intanto
197                 tb_reg_x0 <= PUNO_30b_6f    ; -- * +1
198                 tb_reg_x1 <= PUNO_30b_6f    ; -- * +1
199                 tb_reg_x2 <= PUNO_30b_6f    ; -- * +1
200                 tb_reg_x5 <= PUNO_30b_6f    ; -- * -1
201                 tb_reg_x6 <= ZERO_30b       ; -- * -1
202                 tb_reg_x7 <= ZERO_30b       ; -- * -1
203                 -- adesso y dovrebbe essere 1+1+1-1-0-0 = 2 quindi 1
204                 wait until rising_edge(clk_tb);
205                 pass_input <= '0';
206                 wait until rising_edge(clk_tb);
207                 wait for 15 ns;
208                 if tbs_y_rcv /= PUNO_16b then
209                     fail<= '1' ;
210                 end if;
211             -- neg -> 0
212                 wait until rising_edge(clk_tb);
213                 pass_input <= '1';
214                 wait for THREE_QUARTERS_PERIOD ;
215                 -- cambio solo un paio di registri intanto
216                 tb_reg_x0 <= NUNO_30b_6f    ; -- * +1
217                 tb_reg_x1 <= NUNO_30b_6f    ; -- * +1
218                 tb_reg_x2 <= NUNO_30b_6f    ; -- * +1
219                 tb_reg_x5 <= NUNO_30b_6f    ; -- * -1
220                 tb_reg_x6 <= ZERO_30b       ; -- * -1
221                 tb_reg_x7 <= ZERO_30b       ; -- * -1
222                 -- adesso y dovrebbe essere -1 -1 -1 +1 +0 +0 = -2
    quindi 0
223                 wait until rising_edge(clk_tb);
224                 pass_input <= '0';
225                 wait until rising_edge(clk_tb);
226                 wait for 15 ns;
227                 if tbs_y_rcv /= ZERO_16b then
228                     fail<= '1' ;
229                 end if;
230             -- zero -> 0.5
231                 wait until rising_edge(clk_tb);
232                 pass_input <= '1';
233                 wait for THREE_QUARTERS_PERIOD ;
234                 -- cambio solo un paio di registri intanto
235                 tb_reg_x0 <= PUNO_30b_6f    ; -- * +1
236                 tb_reg_x1 <= PUNO_30b_6f    ; -- * +1
237                 tb_reg_x2 <= PUNO_30b_6f    ; -- * +1
238                 tb_reg_x5 <= PUNO_30b_6f    ; -- * -1
```

```
239                 tb_reg_x6 <= PUNO_30b_6f    ; -- * -1
240                 tb_reg_x7 <= PUNO_30b_6f    ; -- * -1
241                 -- adesso y dovrebbe essere -1 -1 -1 +1 +0 +0 = 0
    quindi 1/2
242                 wait until rising_edge(clk_tb);
243                 pass_input <= '0';
244                 wait until rising_edge(clk_tb);
245                 wait for 15 ns;
246                 if tbs_y_rcv /= PHLF_16b then
247                     fail<= '1' ;
248                 end if;
249             -- daccapone
250                 -- test x values:
251             -- pos -> 1
252                 wait until rising_edge(clk_tb);
253                 pass_input <= '1';
254                 wait for THREE_QUARTERS_PERIOD ;
255                 -- cambio solo un paio di registri intanto
256                 tb_reg_x0 <= PUNO_30b_6f    ; -- * +1
257                 tb_reg_x1 <= PUNO_30b_6f    ; -- * +1
258                 tb_reg_x2 <= PUNO_30b_6f    ; -- * +1
259                 tb_reg_x5 <= PUNO_30b_6f    ; -- * -1
260                 tb_reg_x6 <= ZERO_30b       ; -- * -1
261                 tb_reg_x7 <= ZERO_30b       ; -- * -1
262                 -- adesso y dovrebbe essere 1+1+1-1-0-0 = 2 quindi 1
263                 wait until rising_edge(clk_tb);
264                 pass_input <= '0';
265                 wait until rising_edge(clk_tb);
266                 wait for 15 ns;
267                 if tbs_y_rcv /= PUNO_16b then
268                     fail<= '1' ;
269                 end if;
270             -- neg -> 0
271                 wait until rising_edge(clk_tb);
272                 pass_input <= '1';
273                 wait for THREE_QUARTERS_PERIOD ;
274                 -- cambio solo un paio di registri intanto
275                 tb_reg_x0 <= NUNO_30b_6f    ; -- * +1
276                 tb_reg_x1 <= NUNO_30b_6f    ; -- * +1
277                 tb_reg_x2 <= NUNO_30b_6f    ; -- * +1
278                 tb_reg_x5 <= NUNO_30b_6f    ; -- * -1
279                 tb_reg_x6 <= ZERO_30b       ; -- * -1
280                 tb_reg_x7 <= ZERO_30b       ; -- * -1
281                 -- adesso y dovrebbe essere -1 -1 -1 +1 +0 +0 = -2
    quindi 0
282                 wait until rising_edge(clk_tb);
```

```
283                 pass_input <= '0';
284                 wait until rising_edge(clk_tb);
285                 wait for 15 ns;
286                 if tbs_y_rcv /= ZERO_16b then
287                     fail<= '1' ;
288                 end if;
289             -- zero -> 0.5
290                 wait until rising_edge(clk_tb);
291                 pass_input <= '1';
292                 wait for THREE_QUARTERS_PERIOD ;
293                 -- cambio solo un paio di registri intanto
294                 tb_reg_x0 <= PUNO_30b_6f    ; -- * +1
295                 tb_reg_x1 <= PUNO_30b_6f    ; -- * +1
296                 tb_reg_x2 <= PUNO_30b_6f    ; -- * +1
297                 tb_reg_x5 <= PUNO_30b_6f    ; -- * -1
298                 tb_reg_x6 <= PUNO_30b_6f    ; -- * -1
299                 tb_reg_x7 <= PUNO_30b_6f    ; -- * -1
300                 -- adesso y dovrebbe essere -1 -1 -1 +1 +0 +0 = 0
    quindi 1/2
301                 wait until rising_edge(clk_tb);
302                 pass_input <= '0';
303                 wait until rising_edge(clk_tb);
304                 wait for 15 ns;
305                 if tbs_y_rcv /= PHLF_16b then
306                     fail<= '1' ;
307                 end if;
308                 -- other tests omitted for brevity of print
309                 -- ...
310                 wait until rising_edge(clk_tb);
311                 pass_input <= '1';
312                 wait until rising_edge(clk_tb);
313                 pass_input <= '0';
314                 wait until rising_edge(clk_tb);
315                 pass_input <= '1';
316                 wait until rising_edge(clk_tb);
317                 pass_input <= '0';
318                 testing <= false;
319             end if; -- (di testing == 1)
320
321         end process;
322
323     end architecture;
324 -- fine
```

4 Verification and Testing

The above test-bench was prepared to verify the design and the synthesis of the circuit. It can be interpreted as an ordinary procedural script that sets the input parameter of the Perceptron, and then ensures that the device under test (DUT) behaves as specifications require.

The inputs are all assigned at reset so that x_i are all zero, while the weights w_i are assigned and, in the above tests, never changed.

Due to the lenghty writing of the inputs, during the tests only part of them are changed (time by time).

Some testing signals have been added: fail (defaults 0, 1 if any test fails), pass_input (to explicit the beginning of the iteration, falls as stabile inputs are read by Perceptron), testing (to run the clock only during tests).

The iterations after reset are as follows:

1. @$rising\_edge(clk)$[i] set pass_input=1 ,
2. after some time (delay in input generation) assign inputs,
3. @$rising\_edge(clk)$[i+1] set pass_input=0 , (inputs are received in Perceptron)
4. wait random propagation delay (not needed, also works without this)
5. verify output correctness (all before $rising\_edge(clk)$[i+2]

Since the Perceptron receives the inputs at $rising\_edge(clk)$[i+1] and is expected to (and does) give correct output at $rising\_edge(clk)$[i+2], the elaboration time is within one-clock-period.

It needs to be stated that nothing stops us from using step 4 & 5 to give new inputs, since, as stated, the whole Perceptron works withing a 1-clock-period.

In the above testbench it was not done just to ease readibility of the ModelSim output.

The obtained results are coherent with the specifications, so the testing phase has been successfully passed.

4.1 Test-Bench output example

The following image is an example of the TB output obtained with the application "Intel FPGA Starter Edition". As evidenced, the test was successfull (note that "fail" remains 0 all the time).

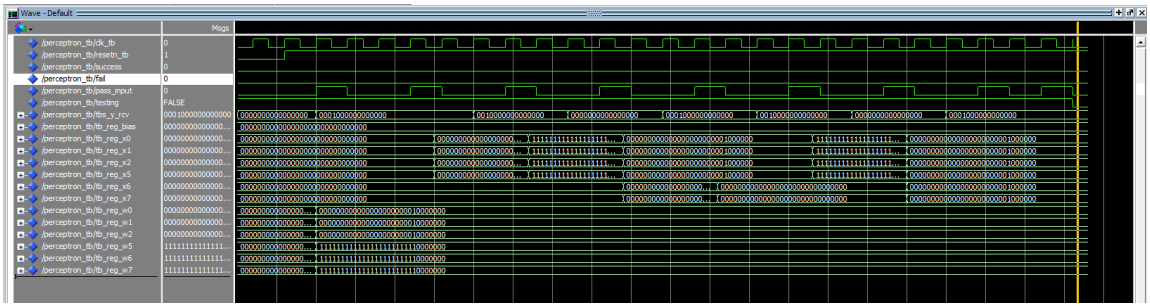


Figure 5: Perceptron and TB signals showed by Intel FPGA Starter Edition.

4.2 Expanding the Test-Bench

Obviously, the testing can be expanded at will, as it was during the writing phase.

The current testing aims to verify the expected output given some "easy to generate" inputs, and, considering the mathematical generality of the algorithm I have implemented, no surprises are to be expected should the inputs vary in the defined input space of [-1,+1].

The above testbench already covers reset, value greater than 2, value less than -2, and value in [-2,2], so all expected outputs are tested at the least twice (every time successfully).

5 Synthesis and Implementation

After the verification phase, the circuit has been synthesized and implemented using the Vivado Tool for the Zybo Zync-7010 board and the produced results have been studied. VIVADO has been instructed to set the clock speed to 62.5MHz (corresponding to 16ns clock period), accordingly with the board's characteristics.

5.1 Vivado design flow

The following steps of the design flow have been performed using Xilinx Vivado software, which allows to perform RTL Elaboration, Synthesis and Implementation on FPGA and also allows to set constraints and get power or timing reports. To ensure no signal path is omitted during the analysis from the software, all system routes have to be associated with a register-logic-register path, therefore all combinatory logic has to be wrapped between a pair of registers.

This was already achieved while designing the Perceptron : every input and output passes through registers (except clock and resetn).

5.2 RTL

VIVADO produced a logic network made of:

1. 220 cells (i.e. multiplexers, registers, AND gates, etc...);
2. 197 IO ports: (10*8 + 11*9 input bits, 16 output bits, clock and resetn);
3. 778 nets: used to interconnect all other components.

5.3 RTL Elaboration

The RTL Elaboration generated results consistent with the expected structure of the system :

1. registers for the inputs and output;
2. after the MULTipliers, ADDers are disposed in a balanced binary tree;
3. two multiplexers are used to implement the out_decisor.

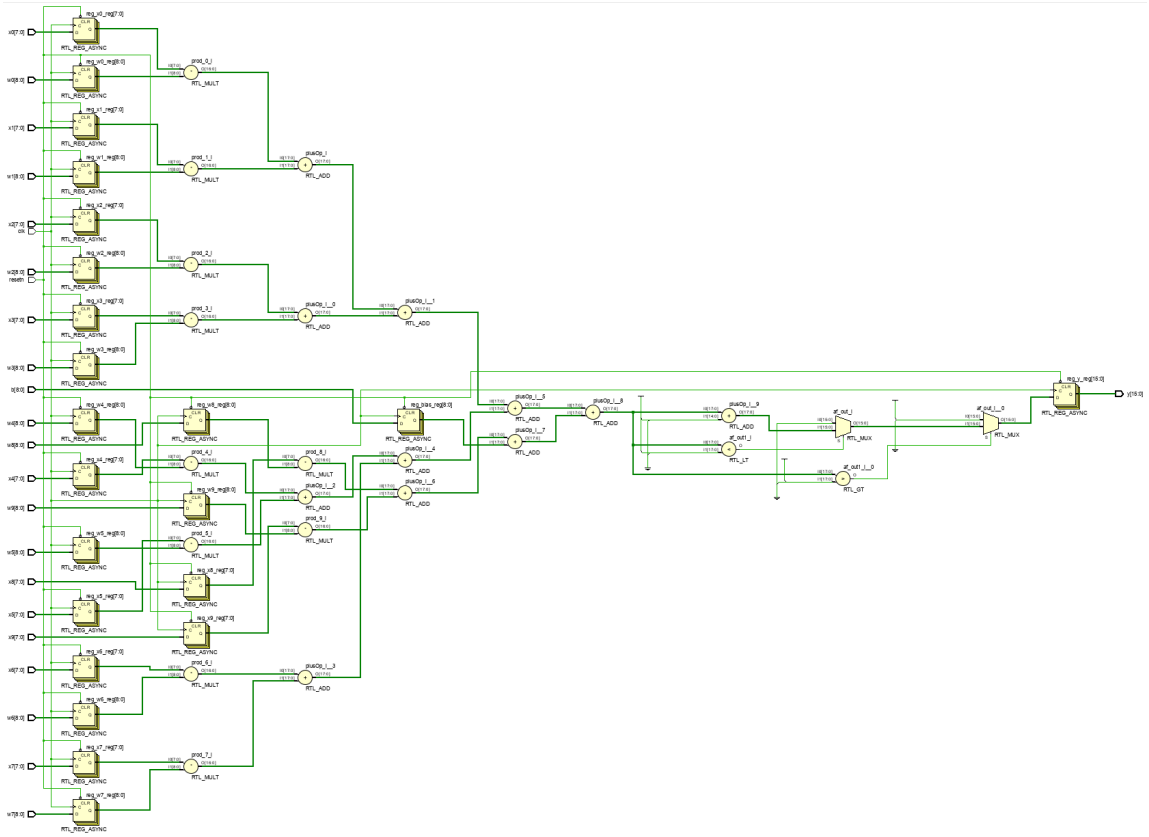


Figure 6: Elaborated RTL Design

5.4 Synthesis and Implementation

The Synthesis runs successfully, but it generates a warning, due to the very high number of inputs, which is unavoidable despite enabling or disabling hierarchy in synthesis.

The Timing Constraints are verified against a **62.5MHz clock (16 ns period)**.

The Implementation yields successful results, although the synthesis is run in *out of context (OOC)* mode, since no mapping of I/O pins is required.

This raises a few extra warnings due to the difficulty in calculating latency on input signals for the board, but accurate timing and power reports can still be obtained.

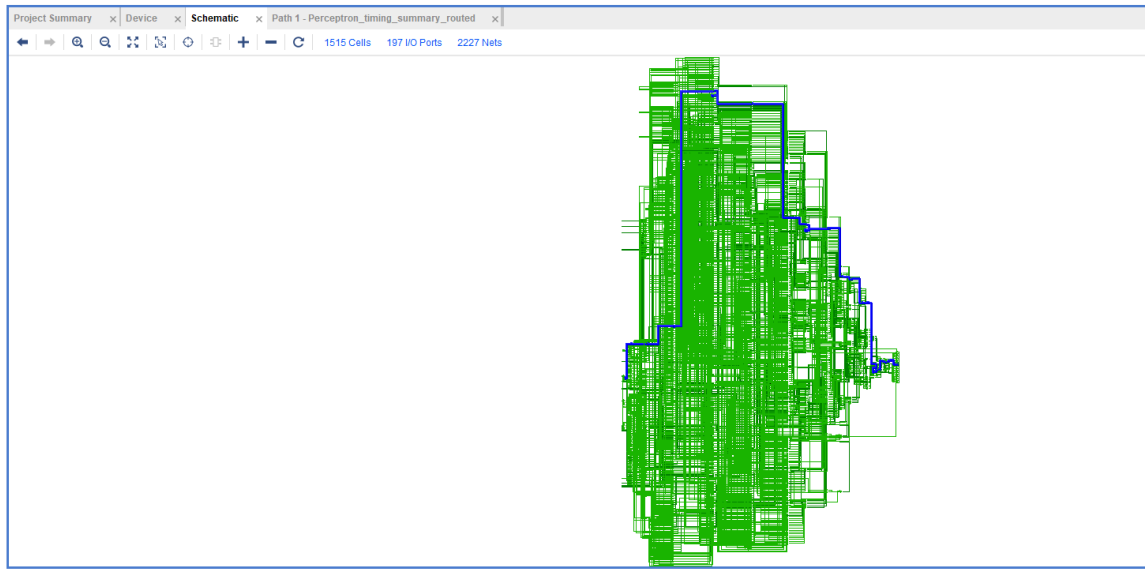
5.5 Trubleshooting and warnings

No errors were encountered during synthesis and implementation, but a few warnings are found:

- [Netlist 29-101] Netlist 'Perceptron' is not ideal for floorplanning, since the cellview 'Perceptron' contains a large number of primitives. Please consider enabling hierarchy in synthesis if you want to do floorplanning. : This is due to the huge number of inputs of this design, and appears to be unavoidable (even when following documentation, and forum suggestions);
- [Timing 38-242] The property HD.CLK_SRC of clock port "clk" is not set. In out-of-context mode, this prevents timing estimation for clock delay/skew & [Route 35-197] Clock port "clk" does not have an associated HD.CLK_SRC. Without this constraint, timing analysis may not be accurate and upstream checks cannot be done to ensure correct clock placement. : another common warning, it is related to the impossibility to perfectly collect latency data in OOC implementation mode. However, it is possible to easily fix this by setting the requested property to the clock port. To do so, one can simply access the Vivado command line and issue the command: set_property HD.CLK_SRC BUFGCTRL_X0Y0 [get_ports pclk_in] which connects a global clock buffer to the clock port of the system.
- [Route 35-198] Port [...] does not have an associated HD.PARTPIN_LOCS, which will prevent the partial routing of the signal [...]. Without this partial route, timing analysis to/from this port will not be accurate, and no routing information for this port can be exported. : this is a warning associated with OOC mode as well and it is similar to the one encountered for clock port, due to the inability to connect external ports to I/O pins. However the warning is not critical and global buffers are only available for clock signals, therefore it has been ignored.

6 Vivado Results

6.1 Critical Path



Power | DRC | Methodology | Timing

Intra-Clock Paths - perc_clk_62-500_MHz_reg - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 2	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 3	0.228	20	19	reg_x4_reg[1]/C	reg_y_reg[4]/D	15.719	7.202	8.517	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 4	0.243	20	19	reg_x4_reg[1]/C	reg_y_reg[5]/D	15.748	7.231	8.517	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 5	0.292	20	19	reg_x4_reg[1]/C	reg_y_reg[10]/D	15.653	7.202	8.451	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 6	0.296	20	19	reg_x4_reg[1]/C	reg_y_reg[12]/D	15.651	7.202	8.449	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 7	0.312	20	19	reg_x4_reg[1]/C	reg_y_reg[11]/D	15.679	7.228	8.451	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 8	0.314	20	19	reg_x4_reg[1]/C	reg_y_reg[14]/D	15.677	7.228	8.449	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 9	0.337	20	19	reg_x4_reg[1]/C	reg_y_reg[8]/D	15.611	7.013	8.598	16.0	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035

Figure 7: Most Critical Path: Schematic & Timing

The most Critical Paths in both the synthesis and implementation go from the input, through the multipliers and adders, to the multiplexers and output. Since basically all inputs make the same theoric paths, there is nothing too noteworthy in this behaviour.

Methodology | Timing

Intra-Clock Paths - perc_clk_62-500_MHz_reg - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
Path 1	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 2	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 3	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 4	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 5	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 6	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 7	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 8	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 9	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 10	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 11	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 12	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 13	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 14	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 15	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 16	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 17	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 18	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 19	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 20	0.106	20	19	reg_x4_reg[1]/C	reg_y_reg[6]/D	15.842	7.202	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 21	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 22	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 23	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 24	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 25	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 26	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 27	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 28	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 29	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035
Path 30	0.124	20	19	reg_x4_reg[1]/C	reg_y_reg[7]/D	15.867	7.227	8.640	16.000	perc_clk_62-500_MHz_reg	perc_clk_62-500_MHz_reg		0.035

Figure 8: Timing of the most critical paths

6.2 Timing Report

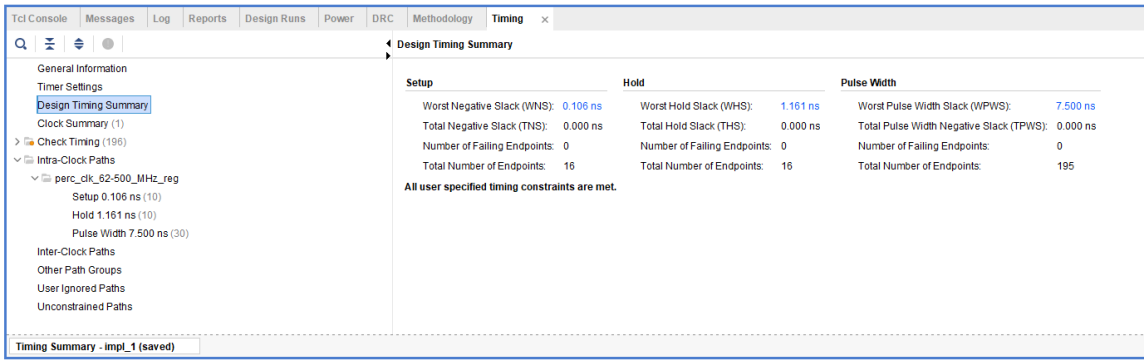


Figure 9: Timing Report

The Timing Report shows quite close margin on the Worst Negative Slack (WNS), and indicates that the **minimum clock-period 16 ns**, corresponding to a **maximum clock-frequency 62.500 MHz** , is extremely close to the best usable.

This frequency, having very small timing tolerance (0.106 ns), might cause some problems in case of clock-jitter.

6.3 Utilization Report

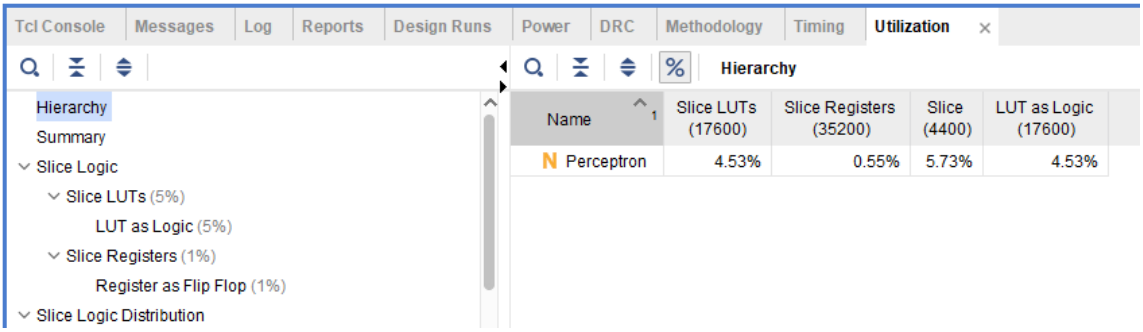


Figure 10: Utilization Report

The Utilization Report indicates a FPGA utilization of 5.73% .

6.4 Power Report

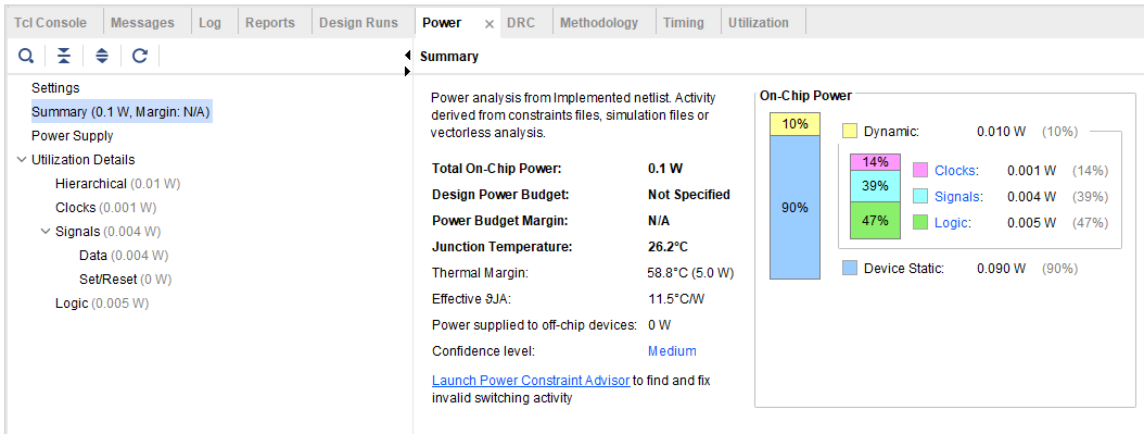


Figure 11: Power Report

The Power Report calculates 0.1W of static power consumption with medium confidence level, mostly related to Logic (47%) and Signals (39%), and an average chip temperature slightly over room temperature (26.2° C) and a wide thermal margin.

Unfortunately, dynamic power consumption is not easy to extract without measuring the system behaviour while it is actively functioning.

7 Final Considerations

The studied circuit is conceptually extremely simple, and definitely clear and concise in what it has to do. The precise -and renowned- application in the field of AI and Deep Learning makes it a very interesting component to study, optimize, build and use.

It can be integrated, with proper timing, into more complex systems of neuron multipliers.

7.1 VIVADO results analysis

Considering the utilization report, it would seem that the circuit is not well suited to be hosted by itself in the considered FPGA (most of the available resources are not needed), but it also indicates that it is designed to perform using quite a simple logic.

More combined or complex implementations will need to deal with the very thin time-margin, but could be optimized (in case of sequential blocks) to reduce the number of registers in separation between "first-input" and "last-output", obviously after proper time analysis of hazards (alee) and worst-paths.