MSc in Computer Engineering

Electronics and Communication Systems

UNIVERSITÀ DI PISA

VHDL Perceptron Project

December 2019

Lorenzo Susini

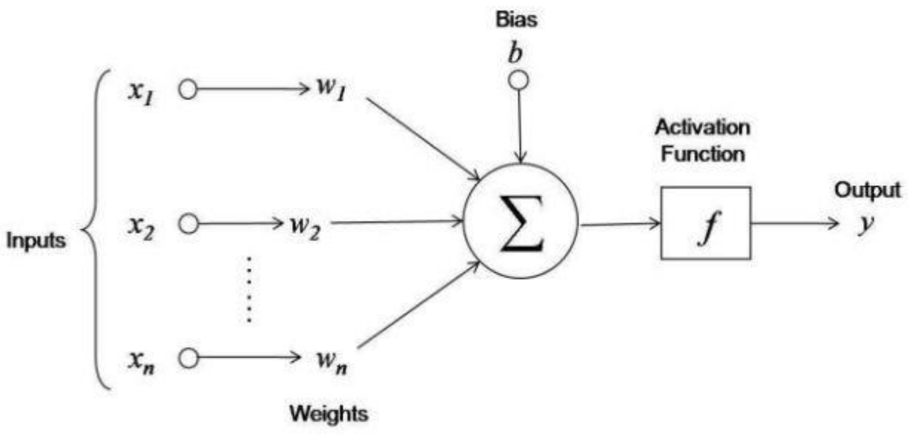
Table of Contents

Requirements

The aim of this project is to develop a simple perceptron in VHDL code and implement it on FPGA. This neuron takes as input $N_{in} = 10$ inputs x and weights w, represented on $b_x = 8$ and $b_w = 9$ bits, respectively. The perceptron also takes as input a bias b, which is also represented on b_w bits. The output of the network must be on $b_{out} = 16$ bits. Each input of the network must be considered in the range $[-1, 1]$. The overall behavior of the perceptron is summarized as

$$y = f(\sum_{i=1}^{N_{in}}(x_i w_i) + bias)$$

The activation function f is defined as $f(a) = \begin{cases} 0 & if\ a < -2 \\ 1 & if\ a > 2 \\ \frac{1}{4}x + \frac{1}{2} & otherwise \end{cases}$



Matlab Model

First, in order to implement the perceptron, I had to define how to represent real number in my solution. I decided to use the Fixed-Point Representation because in this way I can use integers arithmetic in VHDL and because in general it is faster than other representations. Then, I built a Matlab model that helped me to understand if the math in the Fixed-Point notation was correct. I tested the math using the integers corresponding to the real numbers and then checked if the results were correct using the real number representation of Matlab. In order to do so, I decided to test the overall perceptron's logic by splitting it into two parts: the Sum Of Products part (SOP) and the Activation Function part (AF).

In the *testSOP.m* file I tried to understand how many bits were needed for the products and the sums and how $LSBs$ must be modified according to these operations. The most important things to notice are:

- $LSB_{product} = LSB_x \times LSB_w$
- The number of bits of the sum of 10 elements (excluding the bias for now) on $b_{product}$ bits is $b_{sum} = b_{product} + ceil(log_2(10))$
- In order to sum the bias, it must be aligned changing its LSB (adding zeros at the tail) and extended to fit on b_{sum} bits (replicating the MSB at the head as many times as needed)

Then, I wrote down a for loop of $trials$ iteration in which I randomly chose inputs and computed the output of the *SOP*, both with the Fixed-Point integers and the Matlab representation. Storing each result of all the iterations in an array, let me be able to compute and plot the error of the computed *SOP* w.r.t the expected *SOP*. I also computed the *MSE* and it turned out that it is on the order of magnitude of 10^{-5}.

In the *testAF.m* file I tested the Activation Function Logic. This function can be divided in three parts: one linear part and two constant parts. For this reason, there's no need for a LUT in the VHDL code. Using comparators will be enough to figure out how the function must behave. In order to do so, I found the integers corresponding to 2 and -2, that will be directly used in my VHDL code. I built a vector containing real values in the range $[-3, -3]$ with a step of 0.1 and then I gave those values as input to the Activation Function. As before, I compared the output of the Fixed-Point arithmetic with the output of Matlab, then checked that they were equal.

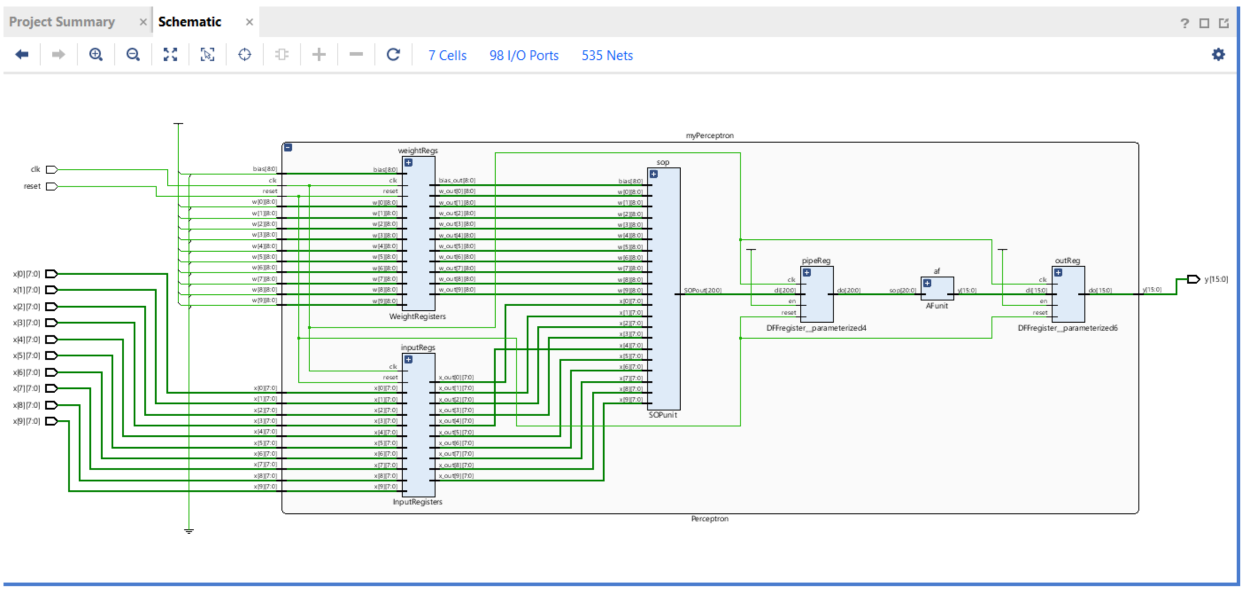
These were the tests to understand if the chosen representation worked. After, I had to develop the perceptron in VHDL. Of course, other kind of tests will be required.

VHDL code

The structure of the VHDL source code is the following:

- Perceptron, the file containing all the building blocks needed. The only objective of this file is to connect the building blocks in the proper way.
- PerceptronWrapper, the file that will be used in the Vivado tool. In this file I decided to make all the weights and the bias constant because otherwise the number of input and output bits of the network is 197 and the board on which I'm going to put my design has only 100 I/O pins. I also decided to put the weights as constant among the other inputs because when an ANN is implemented in HW, most of the times it has to solve a specific problem, so it is already trained in SW and the weights are known.
- SOPunit, Sum of Products Unit
- AFunit, Activation Function Unit
- InputRegisters, a collection of registers containing inputs
- WeightRegisters, a collection of registers containing weights and bias
- DFFregister, the register building block
- Utils, a file containing constants and definitions of useful types to improve readability of code and to write it down in a quicker way. In particular, I defined the types *inputs* and *weights*, that are array of std_loic_vector.

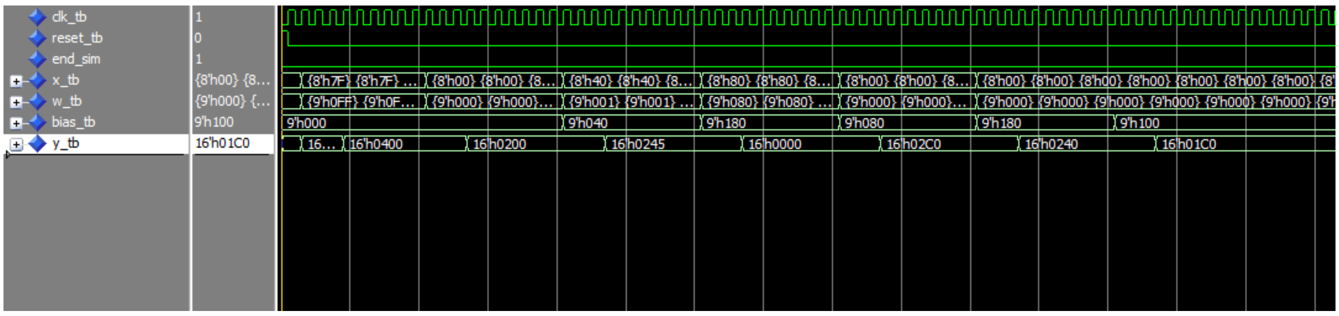
The overall schematic of the network can be seen in the image below.



Notice that the core of the logic is represented by the *SOPunit* and the *AFunit.* I decided to let the clock be at 50MHz (when synthetizing in Vivado) and to use a pipeline register between these two main arithmetic units in order to be sure that no setup violations occurred. Of course, this will increase the power consumption and the latency (measured as number of clock cycles needed to get the relevant output), but will also increase the throughput. Since I don't have this kind of constraints, I decided to put the pipeline register to be sure that I'll meet the timing constraints when using Vivado.

Testbenches and ModelSim

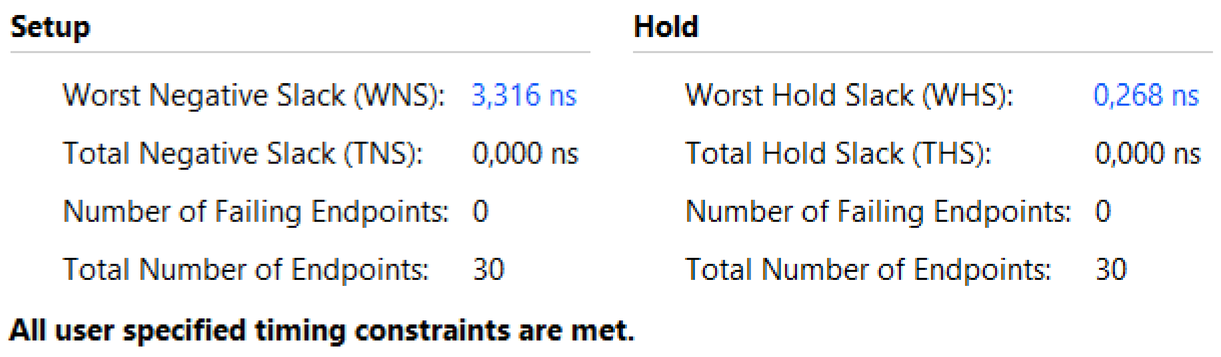
Once the description of the perceptron was done, I also implemented testbenches in order to understand if everything worked. I simulated these tests in ModelSim. First, I tested the *SOPunit* and the *AFunit* separately, using the files *SOPunitTB* and *AFunitTB.* Then, I've created a testbench for the overall perceptron in the *PerceptronTB* file. This file represents the main unit of testing of the VHDL code. The aim of the latter file was to test the overall network and to see if the perceptron was able to put the output in the desired state. I've stimulated the network in such a way that all the three parts of the activation function were evaluated. Notice that to find out the real number corresponding to the output of the network you have to use $LSB_{out} = LSB_{sum} \times 2^5$. This is because the activation function works on $b_{sum} = 21$ bits but then, once the output is computed, it has to be truncated to fit $b_{out} = 16$ bits. The truncation forces the *LSB* to change in that way, since 5 bits are always truncated. Down below you can see a screen of the perceptron testbench in ModelSim.


clk_tb 1
reset_tb 0
end_sim 1
x_tb {8'h00} {8...
w_tb {9'h000} {...
bias_tb 9'h100
y_tb 16'h01C0
{8'h7F} {8'h7F} ... {8'h00} {8'h00} {8... {8'h40} {8'h40} {8... {8'h80} {8'h80} {8... {8'h00} {8'h00} {8... {8'h00} {8'h00} {8'h00} {8'h00} {8'h00} {8'h00} {8'h00}
{9'h0FF} {9'h0F... {9'h000} {9'h000}... {9'h001} {9'h001} ... {9'h080} {9'h080} ... {9'h000} {9'h000}... {9'h000} {9'h000} {9'h000} {9'h000} {9'h000} {9'h000}
9'h000 9'h040 9'h180 9'h080 9'h180 9'h100
16... 16'h0400 16'h0200 16'h0245 16'h0000 16'h02C0 16'h0240 16'h01C0

After, I also decided to carry out a quick check even on the wrapper *PerceptronWrapper* using the file *PerceptronWrapperTB*. Everything went fine, so I was ready to synthetize and implement the design using the Vivado tool.

Vivado

In order to use Vivado, I've created a new project called *Perceptron* in the Vivado folder. Then, I loaded my files and checked if the schematic was correct. I have chosen the same board used during lectures (xc7z010clg400-1). Then, I ran the synthesis using $T_{clk} = 20\,ns$, so $f_{clk} = 50\,Mhz$. The result of the timing report was that the timing constraints were met and even with wide margin (which is good since the synthesis do not consider routing), so I could proceed with the implementation phase. The first time I ran the implementation, I got an error and so it failed. The error was related to the fact that I didn't put yet the weights and the bias as constants, so the number of I/O pins needed were greater than number of those which are available in the board, as already mentioned in this document. So, I fixed the perceptron wrapper putting all the weights and the bias as constant to 0.5 and the implementation went with neither errors nor warnings. The result of the timing report after the implementation phase can be seen in the image below.


Setup

Worst Negative Slack (WNS):	3,316 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	30

Hold

Worst Hold Slack (WHS):	0,268 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	30

All user specified timing constraints are met.

As you can see, the setup slack is good. Of course, this is also due to the pipeline register, which split the propagation delay of the logic in two parts. This means that the clock frequency could be increased defining T'_{clk} in this way:

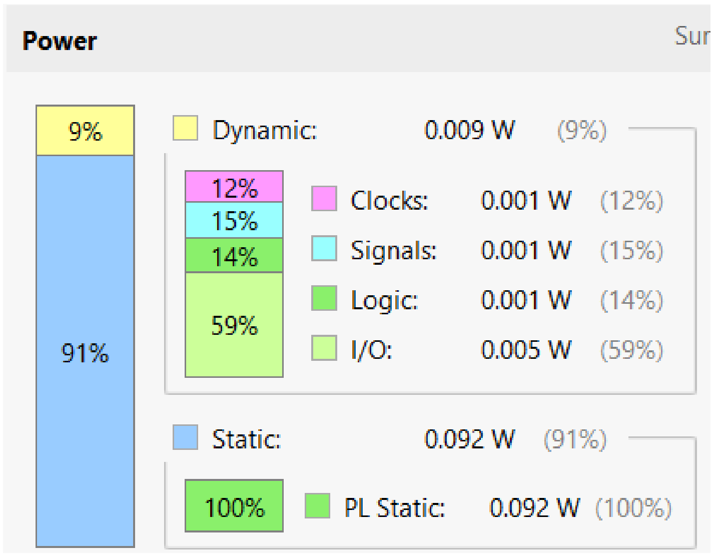
$$T_{clk} - T_{slack} = T_{cq} + T_{prop} + T_{setup}$$

$$T'_{clk} = T_{clk} - T_{slack}$$

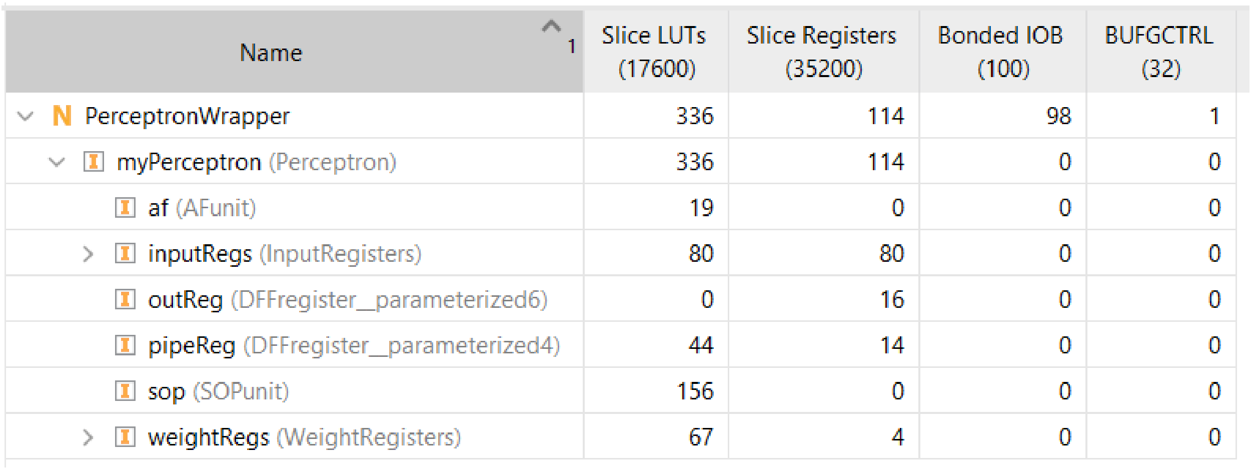
$$f_{max} = \frac{1}{T'_{clk}} \cong 58\ MHz$$

To further increase the clock frequency, other consideration can be done. For example, an idea is to reduce T_{prop} of the logic. In fact, the logic in my design is not optimized, since the arithmetic operations, such as sum and products, are those that are defined in the *ieee.numeric_std* library package. To improve the maths, one can define new adders and multipliers, implementing specific algorithm in order to speed up the logic and achieving higher clock frequencies.

Down below, you can also see the power consumption report and the utilization of the resources on the FPGA.

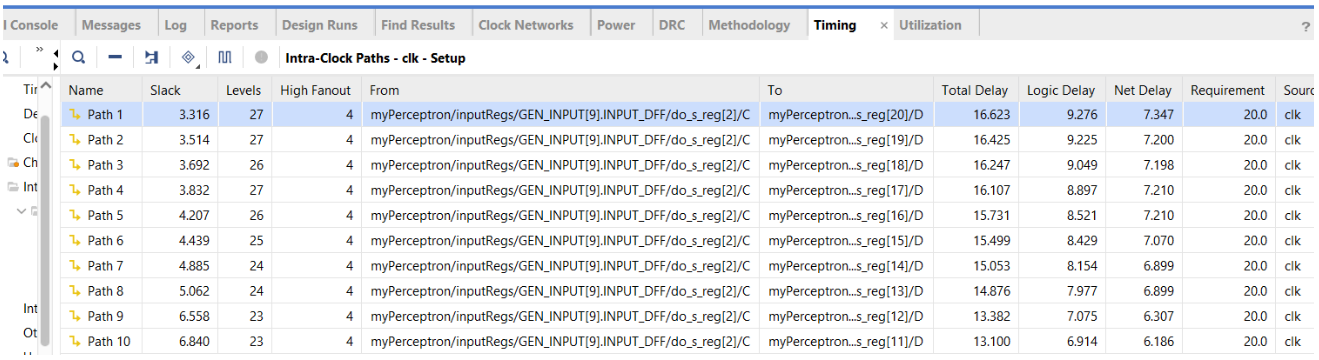


Dynamic:	0.009 W	(9%)
Clocks:	0.001 W	(12%)
Signals:	0.001 W	(15%)
Logic:	0.001 W	(14%)
I/O:	0.005 W	(59%)
Static:	0.092 W	(91%)
PL Static:	0.092 W	(100%)



Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (100)	BUFGCTRL (32)
PerceptronWrapper	336	114	98	1
myPerceptron (Perceptron)	336	114	0	0
af (AFunit)	19	0	0	0
inputRegs (InputRegisters)	80	80	0	0
outReg (DFFregister__parameterized6)	0	16	0	0
pipeReg (DFFregister__parameterized4)	44	14	0	0
sop (SOPunit)	156	0	0	0
weightRegs (WeightRegisters)	67	4	0	0

Here you can also see the worst ten critical path of the design, computed by Vivado.



Intra-Clock Paths - clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Sourc
Path 1	3.316	27	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[20]/D	16.623	9.276	7.347	20.0	clk
Path 2	3.514	27	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[19]/D	16.425	9.225	7.200	20.0	clk
Path 3	3.692	26	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[18]/D	16.247	9.049	7.198	20.0	clk
Path 4	3.832	27	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[17]/D	16.107	8.897	7.210	20.0	clk
Path 5	4.207	26	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[16]/D	15.731	8.521	7.210	20.0	clk
Path 6	4.439	25	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[15]/D	15.499	8.429	7.070	20.0	clk
Path 7	4.885	24	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[14]/D	15.053	8.154	6.899	20.0	clk
Path 8	5.062	24	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[13]/D	14.876	7.977	6.899	20.0	clk
Path 9	6.558	23	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[12]/D	13.382	7.075	6.307	20.0	clk
Path 10	6.840	23	4	myPerceptron/inputRegs/GEN_INPUT[9].INPUT_DFF/do_s_reg[2]/C	myPerceptron...s_reg[11]/D	13.100	6.914	6.186	20.0	clk

Possible applications

Artificial intelligence applications often require working on huge amount of data and at the fastest speed possible. Many big companies are using FPGA to improve throughput and latency of Deep Neural Networks' inferencing. For instance, Microsoft and Intel has developed a project called *Brainwave project*. It is a deep learning platform for real-time AI inference based on cloud. The core of the computing unit is a Neural Processing Unit (NPU), based on FPGAs. As they state in their website (https://www.microsoft.com/en-us/research/project/project-brainwave/), "Project Brainwave is transforming computing by augmenting CPUs with an interconnected and configurable compute layer composed of programmable silicon". It can be used for computer vision and natural language processing applications, for instance. New generations of FPGAs outperform GPU, that were also used for HW acceleration. This happens when compact data types are used (instead of 32-bit floating point data used by GPU). This data types will have the precision needed for the specific application. Moreover, the architecture of the networks for inferencing are rapidly changing, so a flexible and reprogrammable device (such as FPGAs) is very useful.