UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

LOW-LATENCY PERCEPTRON DESIGN AND IMPLEMENTATION IN FPGA

SANTIAGO ROMERO POMAR

2018

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Título:	Low-latency perceptron design and implementation in FPGA
Autor:	D. Santiago Romero Pomar
Tutor:	D. Pablo Ituero Herrero
Ponente:	D. ………………
Departamento:	Departamento de Ingeniería Electrónica

MIEMBROS DEL TRIBUNAL

Presidente:	D. ……………
Vocal:	D. …………..
Secretario:	D. …………..
Suplente:	D. ……………..

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: ………

Madrid, a de de 20…

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

LOW-LATENCY PERCEPTRON DESIGN AND IMPLEMENTATION IN FPGA

SANTIAGO ROMERO POMAR

2018

ACKNOWLEDGEMENTS

En primer lugar, agradecer a Don Pablo Ituero Herrero que me diera la oportunidad de realizar un trabajo de fin de grado sobre un tema tan actual y a la vez con tanto futuro como es el de la inteligencia artificial y más concretamente las redes neuronales.

Agradecer a mis compañeros y amigos de la ETSIT por los piques, las cervezas, el mus, el tráfico de apuntes, el compañerismo, y su inestimable colaboración.

Agradecer a mis amigos de siempre por su apoyo, por todas las horas pasadas en la biblioteca y por todo lo vivido.

Y, por último, y más importante, agradecer a toda mi familia, especialmente a mi madre y Ana por hacer todo más fácil, animarme y aguantarme cuando las cosas no salen, y enseñarme que el esfuerzo nunca cae en saco roto y que se recoge lo que se siembra.

Sin vosotros no hubiera sido posible llegar hasta aquí o por lo menos hubiera sido mucho más complicado, ¡MUCHAS GRACIAS A TODOS!

RESUMEN

Debido al incremento exponencial del número de aparatos electrónicos conectados a internet, la cantidad de datos que los mismos producen ha aumentado en la misma medida. Para hacer frente a esto, se ha generalizado el uso de algoritmos de aprendizaje automático, conocidos como *Machine Learning*, siendo el más popular el conocido como redes neuronales. Estos algoritmos necesitan, para su implementación, gran cantidad de recursos para realizar cálculos, por lo que tradicionalmente se han dado en circuitos integrados específicos para una aplicación concreta. Sin embargo, recientemente se está produciendo un auge en las implementaciones en arrays de puertas programables, más conocidos como FPGAs, ya que permiten mayor paralelismo en la implementación de los algoritmos.

Este proyecto consiste en el diseño e implementación de una neurona y una red neuronal con un objetivo principal claro y conciso: la baja latencia. Por este motivo, la plataforma por la que se ha decantado en vista a la implementación es una FPGA, concretamente se usará la placa Nexys 4 DDR de Digilent basada en una FPGA de Xilinx. Para el diseño se ha seguido una estrategia: búsqueda y comparación de diferentes topologías de todos los módulos que forman la neurona (memoria, multiplicador, sumador, función de activación) hasta encontrar la que introduzca menor latencia. También se ha establecido en el sistema una notación numérica que favorezca la baja latencia y que minimice las pérdidas de precisión en la computación de datos. Una vez encontradas dichas topologías, el entorno de Vivado se utiliza para la síntesis, implementación y verificación de esas entidades en VHDL. Una vez diseñadas todas las partes, se ponen en común y se añade un controlador necesario tanto para aprovechar el paralelismo que ofrece este hardware como para el correcto funcionamiento del sistema. Cuando ya se ha completado el diseño de la neurona, se forma una red replicando varias de ellas. Se han implementado en MATLAB pruebas funcionales para simular el funcionamiento tanto de la neurona como de la red, así como para establecer una comparación del tiempo de ejecución con respecto a la implementación en VHDL. Estas implementaciones son más de 400.000 veces más rápidas que las pruebas funcionales en MATLAB. Para concluir, se ha probado la implementación de esta red en la FPGA anterior.

SUMMARY

Due to the exponential increase of electronic devices that are connected to the Internet, the amount of data that they produce have grown to the same extent. In order to face the processing of these data, the use of some automatic learning algorithms, also known as Machine Learning, has become widespread. The most popular is the one known as neural networks. These algorithms need a great deal of resources to compute all their operations, and because of that, they have been traditionally implemented in application specific integrated circuits. However, recently there have been a boom in implementations in field programmable gate arrays, also known as FPGAs. These allow greater parallelism in the implementation of the algorithms.

This project consists of the design and implementation of a neuron and a neural network with a clear main objective: low latency. For this reason, the platform that has been chosen regarding to the implementation is an FPGA, specifically the Nexys 4 DDR board from Digilent based on a Xilinx FPGA. The strategy followed when in the design phase was the following: search and comparison of the different topologies of all the modules that make up a neuron (memory, multiplier, adder, activation function) until the one that introduces the lowest latency was found. A numerical notation that boosts low latency and minimizes precision loss in data computation has also been implemented. Once these topologies have been found, Vivado framework was used for synthesizing, implementing and verifying these entities in VHDL. When designed all these parts, they were put together. A controller, necessary for both exploiting the parallelism provided by the piece of hardware used in this project and the correct performance of the system, was implemented. When the design of the neuron was finished, a network was formed by replicating some of these neurons. Functional tests were implemented in MATLAB in order to mimic the behaviour and establish an execution time comparison with the VHDL implementation for both the neuron and the network. These implementations are more than 400,000 times faster than MATLAB functional tests. Finally, the implementation of this network was tested in the FPGA.

PALABRAS CLAVE

Neurona, baja latencia, aprendizaje automático, inteligencia artificial, redes neuronales, VHDL, Vivado, FPGA.

KEYWORDS

Neuron, low latency, machine learning, artificial intelligence, neural networks, VHDL, Vivado, FPGA.

GLOSSARY

AI	Artificial Intelligence
AM	Array Multiplier
ANN	Artificial Neural Network
AN	Artificial Neuron
ASIC	Application Specific Integrated Circuit
CSA	Carry Select Adder
DM	DADDA Multiplier
DSP	Digital Signal Processor
FA	Full Adder
FPGA	Field Programmable Gate Array
HA	Half Adder
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Multiply-Accumulate
ML	Machine Learning
MSB	Most Significant Bit
MUX	Multiplexer
RALUT	Range Addressable LUT
RCA	Ripple Carry Adder
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit

CONTENTS

TABLE OF CHARTS

TABLE OF FIGURES

TABLE OF VIVADO IMPLEMENTATION FIGURES

1. INTRODUCTION AND OBJECTIVES

1.1. NEURAL NETWORKS

According to what John McCarthy said in the 1950s, Artificial Intelligence (AI) is "*the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do*" and moreover Neural Networks are a wide part of AI.

Also in the 1950s, specifically in 1959, Arthur Samuel defined Machine Learning (ML), which is a category included in AI, as "*the field of study that gives computers the ability to learn without being explicitly programmed*". This means that a single program can be used for many different ML applications as it will learn how to solve these problems in contrast to purpose-built algorithms, which only perform in a single scenario. It is achieved by the use of training processes instead of customizing the program with a hit-or-miss approach.

1.1.1. SUPERVISED VS. UNSUPERVISED LEARNING

Machine learning algorithms can be divided in two main categories according whether their learning is supervised or unsupervised.

In the former method, all the training examples are labelled with the correct class, so that the algorithm learns by comparing its outputs with the right given answers. These comparisons lead the algorithm to make changes in the weights to adapt them to the right result.

In the latter method, training examples are not labelled so the algorithm divides the training examples into clusters and finds some structure in the data [1].

1.1.2. REGRESSION VS. CLASSIFICATION

There are two kinds of ML problems, which are regression and classification problems. While in the former the output is a continue value, in the latter the output is a discrete value. Each one is solved using different types of algorithms.

When having a regression problem, the best algorithm that can be used is linear (or polynomial) regression. But when there is a classification problem, a logistic regression algorithm will perform well. These mentioned algorithms work well when the problems that have to be solved present a small number of *features*, but not as well when the number is higher.

When classifying images, the training examples are images that the algorithm has to process. In these cases, every single pixel of the image is a feature of the algorithm, so the number of features is huge. Because of this, regression algorithms cannot be used to process images and more powerful ones are needed.

The most used algorithm for classification problems is one that works better than logistic regression when the number of features is large, it is known as Artificial Neural Networks (ANN). This is a ML, brain inspired algorithm [2].

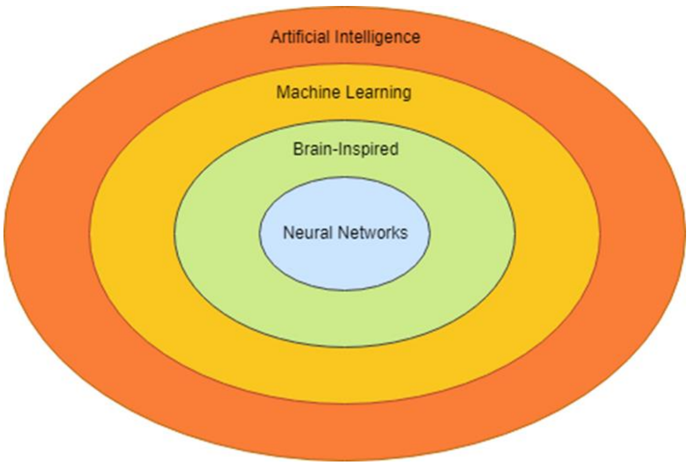


Figure 1. Neural Networks in the context of Artificial Intelligence

Neural Networks' name is due to the brain components, which make up the best-known "machine" used to learn and solve problems. The way this organ is believed to operate served as inspiration to design a ML algorithm. The neuron is the main component of the brain, so it is in these algorithms. Because of that, an analogy between both can be drawn. Neurons are connected with each other with some entering elements called *dendrites* and some leaving ones called *axons*. Information signals, which enter to the neuron via the dendrites, perform some computation and generate a signal on the axons, are the *activations*. Connexions between axons and dendrites are referred to as *synapses*. These can make changes in the information entering to the neuron, scaling the signal (x_i) by some factor, which is the *weight* (w_i).

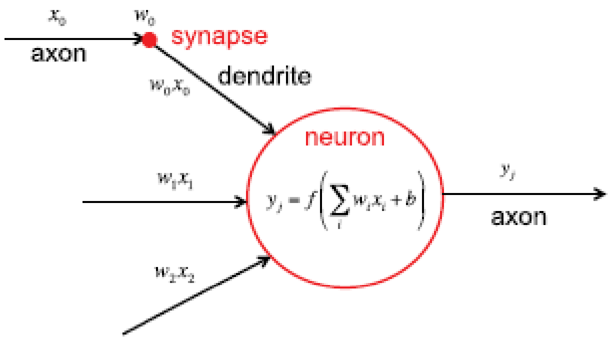


Figure 2. Connections to a neuron in the brain

The way the brain is supposed to learn is by making changes to the weights associated to synapses: different weights lead to different responses to the same input, keeping the same organization [1].

An Artificial Neural Network implements an algorithm that can mimic the brain. Each artificial neuron (AN) or perceptron performs a scaled sum of its inputs. The scaling is determined by the parameters, also known as weights. After that, the result of this weighted sum goes through a non-linear function. This function is also referred to as the activation function, because the neuron generates a non-zero output only if the value of this sum is greater than an established threshold.

TRAINING VS INFERENCE

The learning process involves the way to determine the value of the weights in the network. This is called *training*. Once the program is trained, it can perform its tasks using the correct weights. This is referred to as *inference*. In this project, learning process is supposed to be finished, so the final values of weights are known and all the computations belong to inference processes.

Not only ANNs imitate the brain behaviour, but also its topologies. These networks are distributed into different layers and always will be an input and output layer. Optionally, there can be one or more hidden layers between the input and output ones. All these layers are compound by a number of neurons.

There are as many neurons in the input layer as the feature size of the algorithm. The size of the output layer is the same as the number of categories that there are in the (multiclass) classification algorithm.

There are some different topologies that can be implemented: feed-forward vs recurrent, fully-connected vs sparsely-connected, etc. The most common type is feed-forward, fully connected topology, which means that every output of the neurons of a layer are connected to all the neurons of the next layer.

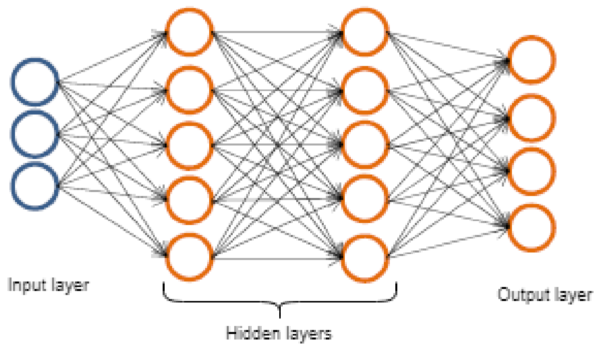


Figure 3. Example of a feed-forward, fully-connected Artificial Neural Network

Each layer has an extra neuron that has no input but one output that has value ‘1’. It is called the bias neuron and it is denoted with a subscript 0. Apart from this, it works as any other neuron, because it has a weight associated to it.

Each neuron computes the following: $a_j^{(l+1)} = g\left(z_j^{(l+1)}\right) = g\left(\Theta_{j0}^{(l)} a_0^{(l)} + \Theta_{j1}^{(l)} a_1^{(l)} + \cdots + \Theta_{jn}^{(l)} a_n^{(l)}\right)$, where l represents the number of layer, j represents the number of neuron, $\Theta_{ji}^{(l)}$ represents the weight of the i input of the j neuron of the l layer, $a_j^{(l)}$ represents the activation of the j neuron of the l layer and g represents the activation function.

The previous computation is called Forward Propagation and it is used in inference process and in training process along with Back Propagation. The former flows from the left to the right and outputs the result of the weighted sum through the activation function. The latter flows from the right to the left and uses the results of Forward Propagation to minimize errors and adjust the weights of the network [2].

But as I mentioned before, in this project we suppose that Back Propagation and all the computations necessary to adjust and calculate the weights of the neuron have been previously carried out.

There is a wide range of activation functions that can be implemented in the neuron. These should be non-linear functions. They can be grouped in two types: unipolar and bipolar. Among those in the first category, the most important ones are the unit step, the unipolar sigmoid function and the Rectified Linear Unit (ReLU). In the second group, the most important ones are the hyperbolic tangent and the bipolar sigmoid function.

When choosing the activation function, it is also important its derivative, because it is needed in Back Propagation. Sigmoid function is one of the most used activation functions because it varies between 0 and 1, as well as probabilities which are the outputs of the neurons. ReLU is also very used because when the input is negative, there is no need of activation. The latter would be very useful in huge ANNs because of being computationally cheaper [1] and [3].

In all of these functions, the decision boundary that work as the threshold in the classification can be changed depending on the necessities of the system.

1.1.3. IMPLEMENTATION

Tons of data as well as lots of power are needed to train an ANN. These networks require a great deal of resources in order to process all the information.

The main reason that is stated in favour of developing custom integrated circuits or ASIC in order to implement ANN applications is that general-purpose processors do not exploit to the maximum the parallelism that the ANNs can afford. But this statement is not entirely correct, because the goal is achieving high performance using attributes of the model, not just parallelism.

It is evident that FPGAs cannot match ASICs in terms of performance. But they can achieve better cost/performance ratios than both ASICs and conventional general-purpose processors. What is more, the capacity of reconfiguration and programmability lead FPGAs to expand to a wide range of applications, that is to say, a wide range of different types of ANNs. FPGAs also allow the user to design customized architectures in which a certain degree of parallelism can be achieved and to use blocks that had been previously designed, such as Digital Signal Processor (DSP) slices or Block RAMs (BRAMs), which can reduce the area implementation and the latency.

These facts make FPGAs an interesting alternative, so they are contributing to an important increase of their use. And these attributes are also the reason why we are going to use one of them in this project [3] and [4].

1.1.4. PARALLELISM

As parallelism was mentioned before, it is important to examine the different types and levels which ANN architectures can be parallelized at:

- **Training parallelism**: this means that different training sessions of the same ANN can be run at the same time.
- **Layer parallelism**: in a multilayer ANN, the computation of different layers can coexist.
- **Node parallelism**: this means that every single AN is computed at the same time. It can be achieved in FPGAs because they consist of a huge number of cells onto which ANs are mapped.
- **Weight parallelism**: in the computation of an AN, the partial products of the multiply and accumulate (MAC) operation can be processed at the same time. More than one sum can also be calculated at the same time.

These different levels can be implemented depending on the architecture, the application and the constraints of a particular ANN. A good practice of parallelism can be achieving a trade-off between these different levels.

Node parallelism is the most important level because if it is exploited, the higher levels are exploited too. Although full parallelism cannot be achieved, the ANN model allows the algorithm to reach a great degree of parallelism [3].

1.1.5. APPLICATIONS

ANNs can be used in a wide range of different applications:

- **Image and video**: more than 70% of the Internet traffic are images and videos, so all these data feed ANNs to extract all the meaningful information. Among many other applied examples of this area, the most used tasks are image classification, detection and recognition.
- **Speech and language (sound)**: the accuracy of speech recognition has been significantly improved. In addition, tasks such as machine translation, natural language processing and audio generation can be performed too.
- **Medical**: medical is arguably the area where machine learning has made the biggest impact. It helps in very important labours such as recognising patterns of genetic diseases and medical imaging.
- **Robotics**: robotics has grown up tied to machine learning as both try to mimic human abilities. Most of the applications are based on controlling a number of engines in order to achieve a particular motion. But the most important application of this area is autonomous navigation.

- **General data classification**: when some data of multiple origins has to be analysed in order to predict some values or classify in different categories, ANNs are the best choice.
- **Control**: ANNs appeared to be a perfect solution in some applications that require real-time performances, such as some video games.
- **Testing**: when evaluating an ANN, some simple testing is performed, such as implementing logic gates (AND, OR, NOR, XNOR).

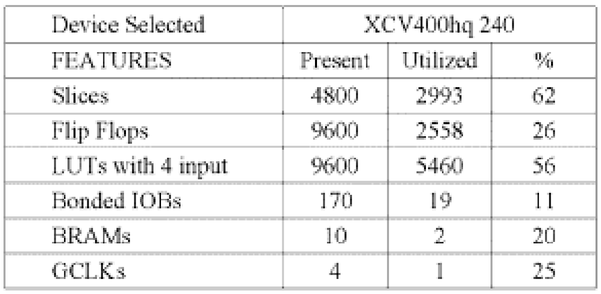
There are also some areas where ML and ANN can make an impact in the future [1] and [4].

1.2. STATE OF THE ART

Nowadays, when implementing feedforward artificial neural networks in FPGAs, there are some possible approaches.

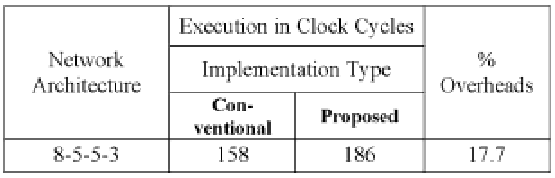
One of them is using Layer Multiplexing for Effective Resource Utilization which is presented in [5]. Due to the great number of multipliers in an ANN the size of the network that can be implemented using a single FPGA is limited, what make ANN applications not viable commercially. It is solved by cutting down on resource requirement without affecting to the speed, so that a larger ANN can be implemented in a single chip at a lower cost using a method of layer multiplexing. It consists of, instead of programming a complete network, only the single largest layer is implemented. The same layer works as different layers with the help of a control block that ensures correct behaviour by assigning the correspondent inputs, weights and activation function of the layer that is currently being worked out.

These ANNs have been designed, synthesized, implemented using Xilinx FPGA XCV400hq240 using Verilog language in the Xilinx6.1i framework and simulated with ModelSim XE II 5.7c. The result of this process was tested in the FPGA mentioned before using mechatronics test equipment-model MXUK-SMD-001. The highest speed that this implementation can achieve is 73 MHz. The area utilization and timing results of the implementation for an ANN with 8-5-5-3 ANs in each layer are shown below in two different tables, respectively.



Device Selected	XCV400hq 240		
FEATURES	Present	Utilized	%
Slices	4800	2993	62
Flip Flops	9600	2558	26
LUTs with 4 input	9600	5460	56
Bonded IOBs	170	19	11
BRAMs	10	2	20
GCLKs	4	1	25

Table 1. Synthesis result of an 8-5-5-3 ANN



Network Architecture	Execution in Clock Cycles		% Overheads
	Implementation Type		
	Con-ventional	**Proposed**	
8-5-5-3	158	186	17.7

Table 2. Speed for 8-5-5-3 conventional and proposed implementation

In the 7th chapter of [3], it is discussed another possible approach. It is the implementation of Neocognitrons, which were proposed by Kunihiko Fukushima in the middle 1980s. They are massively parallel, feed-forward neural networks that are organized in a particular way. There are two different kinds of neuron layers: simple (Simple-cell, S-cell) and complex (Complex-cell, C-cell). These networks are organized in sequence of S-cell and C-cell layers. The former is composed by some S-cell planes, and each one consists of a matrix of neurons that process the feature extraction applying convolution to the previous C-cell layer. The latter is composed by some C-cell planes, which process an average operation to the previous S-cell layer. They also reduce the number of neuron cells at the following layers until the last one, where only a unique neuron represents each one of the recognized objects.

In the 8th and the 9th chapters of [3] another possible approach is presented. It is the use of Self-Organizing Maps (SOM), which are a type of ANN that are trained with an unsupervised learning algorithm. The behaviour of this architecture consists of a sequence of inputs that enter to a network that self-organizes to quantize the input space as optimally as possible. This can also adopt a massively parallel computing structure. In order to be implemented, a number of FPGAs work in parallel in order to increase the size of the maps and improve the performance of the system. Each FPGA contains its own controller and a matrix of processing elements. Most calculations are performed in parallel distributed by all processing elements.

These ANNs have been described in VHDL and synthesized using the Synopsys FPGA compiler and the Xilinx Alliance tools for placing and routing. They have been implemented in a series of different FPGAs of the Xilinx Virtex series, as it is shown in the following table.

Device	N_{PE}	l_{max}	f_{FPGA} [MHz]	$P_{C,r}$ [MCPS]	$P_{C,l}$ [MCUPS]	Utilization of Slices
XCV1000-6	64	256	50	2825	325	48
XCV812E-8	70	2048	65	4463	495	68
XCV2000E-8	160	512	65	9717	1097	77
XCV3200E-8	208	512	65	12632	1426	59
XC2V6000-6	288	1024	105	29158	3253	78
XC2V10000-6	384	1024	105	38877	4338	58

Table 3. Comparison of performance values of different FPGAs of the Xilinx Virtex series

Where N_{PE} is the number of processing elements, l_{max} is the maximum input vector dimension, f_{FPGA} is the maximum work frequency of the FPGA, $P_{C,r}$ is the performance during recall and $P_{C,l}$ is the performance during learning. The last two can be calculated by $P_{C,r} = \frac{l \cdot N_N}{T_r}$; $P_{C,l} = \frac{l \cdot N_N}{T_l}$; where l is the input vector dimension, N_N is the number of ANs and N_{NA} is the number of ANs that are updated during learning. T_r is the time required to find the best matching element for an input vector and T_l is the time required for one learning step.

In the 10th chapter of [3], another pair of implementations are explained, they are Multi-Layer Perceptron (MLP) and eXtended Multi-Layer Perceptron (XMPL). While MLP is well-known example of feed-forward ANN that is compound of at least three layers (input, hidden and output), XMLP is a modified version. The latter is an MLP-like feed-forward ANN with two-dimensional layers and configurable connections pathways. In this work there are introduced two implementation versions, a parallel and a sequential design of both MLP and XMLP. The designs of both the MLP and the XMLP have been carried out using VHDL in the FPGA Advantage 5.3 tool, from Mentor Graphics and have been placed and routed onto a Virtex-E 2000 FPGA, using the synthesis tool Xilinx Foundation 3.5i. The synthesis results are presented in the following tables:

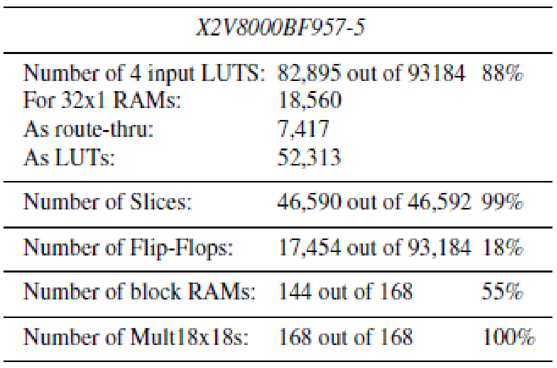
MLP Design	♯ Slices	% Slices	♯EMBs RAM	%EMBs RAM	Clock (ns)	♯ Cycles	EvaluationTime (ms)
Serial	379	1.5	19	11	49.024	5630	276.005
Parallel	1614	8.5	26	16	53.142	258	13.710

Table 4. Synthesis result of MLP design

MLP Design	♯ Slices	% Slices	♯EMBs RAM	%EMBs RAM	Clock (ns)	♯ Cycles	EvaluationTime (ms)
Serial	267	2	11	6	37.780	2478	93.618
Parallel	1747	9	49	30	53.752	152	8.170

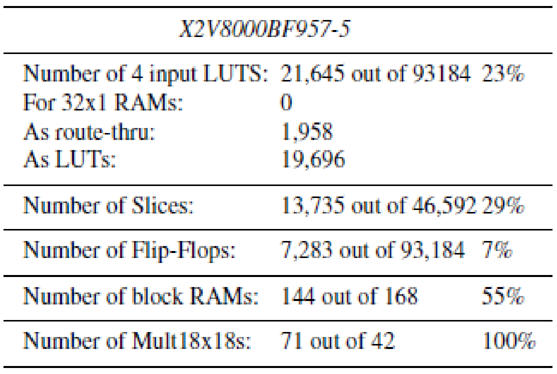
Table 5. Synthesis result of XMLP design

In the 11th chapter of [3], another interesting approach is explained, it is the Non-Linear Predictor. It is implemented using an MLP, which has been explained recently, with the backpropagation learning algorithm. The design flow is based on VHDL and it is achievable on a single FPGA, obtaining a throughput of 50 MHz in XC2V8000-BF957-5 of Xilinx.



X2V8000BF957-5		
Number of 4 input LUTS:	82,895 out of 93184	88%
For 32x1 RAMs:	18,560	
As route-thru:	7,417	
As LUTs:	52,313	
Number of Slices:	46,590 out of 46,592	99%
Number of Flip-Flops:	17,454 out of 93,184	18%
Number of block RAMs:	144 out of 168	55%
Number of Mult18x18s:	168 out of 168	100%

Table 6. Synthesis result of MLP (42-128-50-1) with PWL15



X2V8000BF957-5		
Number of 4 input LUTS:	21,645 out of 93184	23%
For 32x1 RAMs:	0	
As route-thru:	1,958	
As LUTs:	19,696	
Number of Slices:	13,735 out of 46,592	29%
Number of Flip-Flops:	7,283 out of 93,184	7%
Number of block RAMs:	144 out of 168	55%
Number of Mult18x18s:	71 out of 42	100%

Table 7. Synthesis result of MLP (42-128-50-1) with PWL15 without training

PWL15 is the piecewise linear (PWL) approximation of the sigmoid function composed of 15 segments.

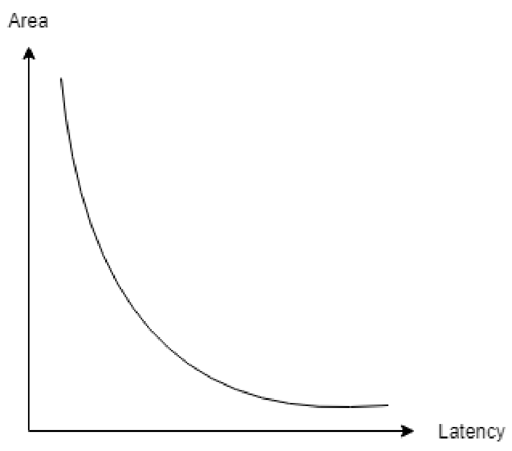


Figure 4. Area vs latency graph

While the maximum achievable frequency of these implementations is slightly higher than the achieved in this project, the number of cycles needed in order to complete the computation is much lower in this work than in the previous implementations. These facts mean that the total computation time is much bigger in the previous architectures. It is caused because while the previous topologies exploit multiplexing, self-organizing or multi-dimensional algorithms; the one developed in this project exploits low-latency and parallelism. On the other hand, these previous topologies perform a better trade-off between area utilization and latency, but as the main objective of this project is achieving the lowest possible latency, the developed topology performs better with these constraints and requirements.

The proposed design would be located at the leftmost point of the curve shown in the previous figure, while the other designs would be located in some centred position of the curve because of the mentioned trade-off that they achieve.

1.3. OBJECTIVES

The main goal of this project is **to define a specific architecture of a perceptron that will be used to build low-latency ANNs on FPGAs**. This architecture must meet the following constraints:

- The hardware target is Xilinx's FPGA part number XC7A100T-1CSG324C of the Nexys 4 DDR board.
- VHDL will be the hardware description language used in the project.
- The architecture will be tested with a logic gate example.
- The architecture must be **as fast as possible**.

A second objective of the project is to make the perceptron as configurable as possible. This means that the user of the ANN will be able to customize some aspects of the architecture. This is useful in order to be able to reuse the same implementation to a wide range of applications.

In this project I achieved an implementation of a AN in an FPGA that gets a speed-up greater than 10^5 when compared to the functional test computed with a general-purpose machine, such as a laptop. Similar results can be obtained in the case of the network implemented with these ANs. Timing results demonstrate that the main purpose of the project was accomplished: a low-latency perceptron that computes much faster than a mathematical processing software, such as MATLAB.

There were also some important academic objectives and achievements. In first place, I acquired some knowledge about different algorithms of ML, which was important when putting in context. I also developed my programming skills, not only in VHDL or MATLAB, but also in every aspect involved in the programming process: looking for documentation, debugging, verifying, testing, etc. What is more, I came across some features of Vivado that I did not know and that I had never used but that turned out to be very useful, so that now I am more fluent when working with that programming environment. In last place, I learned how to apply a specific strategy to a design in order to achieve results that meet specific conditions, what I believe is the most important aspect in this project.

1.4. DESCRIPTION OF THE DEVELOPMENT OF THE PROJECT

I will now specify how the project was developed, trying to summarize how I spent each hour dedicated to it.

The first step of the project development consisted on completing the first five weeks of Coursera’s “Machine Learning” course [2], which took me about 40 hours, where I learnt the basics about Machine Learning, Linear and Logistic Regression and ANNs. I also implemented some examples of ANNs in Octave language when doing the course assignments.

Once I learnt some basic notions about ANNs, I started reading some scientific and technical articles in order to go deeper into the subject [6], [3], [1] and [4]. I also searched some information about the implementation of ANNs in FPGAs [7] and [8] which gave me a better global perspective. This research lasted for approximately 30 hours.

Then, in order to compose the lowest-latency Artificial Neuron, I started to look for the lowest-latency components of the perceptron reviewing some books [9] and articles that I found on the Internet [10] and [11]. Some of them are comparatives between a variety of components in terms of latency, power consumption and area.

The first components that I acquired knowledge about were those which make up a MAC unit: multipliers and adders. Not only I read these comparisons, but I also implemented these components in VHDL language using Vivado framework in order to corroborate those analysis. As it happens in every software development process, in first place there is an early stage where the requirements of the algorithm are specified. After that, there is a design phase where it is covered how the program is going to be created. Then, the implementation stage, where the code is developed and synthesized in Vivado, and then the results of the timing and utilization analysis are presented. At the end, the product of the previous phases is verified using the simulator that is integrated in the framework.

Once I finished these implementations, I could understand better how these adders and multipliers work. I could also make my own comparations between them in order to help me to choose which arithmetic element was the best choice for each group. The whole process of research, implementing and veifying for both types of entities took me at least 58 hours.

When the MAC unit was ready, it was necessary to provide meaning to the signals that are involved in the system. As this work is focused in achieving the faster possible perceptron, the best approach is fixed-point quantification because of the simplicity of its arithmetic. As in [2] there were some MATLAB scripts that analyse the accuracy of an ANN, I modify them in order to help me to choose the best precision distribution of the digital word that keep the precision of the algorithm. The changes that I applied in these scripts consisted of converting all the numbers that intervened into numbers with the same fixed-point quantification, trying all the possible combinations of the integer and decimal bits that kept the most significant bit (MSB) for the sign bit. This quantification was achieved by the use of MATLAB's "Construct fixed-point numeric object", which transforms an object number into an object that represent a number with a specific precision. The result of this transformation must be converted again to the same previous object number type without changing the value in order to be able to operate with them. The distribution that achieved the highest accuracy was 1-4-3: MSB for the sign bit, the next four bits for the integer part and the remaining three for the decimal part. I achieve this in approximately 10 hours.

When the quantification structure of binary words was ready it could be applied in the entities when necessary.

The remaining components of the AN were the sigmoid function and a memory to store the weights. The latter was implemented using the IP catalog integrated in Vivado software, which provides an easy way to design in great detail a wide range of components [12]. The final design was a single-port Read Only Memory (ROM) that has a 16-word depth of 8-bit words. The memory was initialised a specific type of file. This component was ready after 8 hours of work.

I searched in the Internet different ways to implement the sigmoid function and I found some articles [13] and [14] that were interesting because there was a comparison between different approaches to the sigmoid function in terms of latency, area utilization and precision error. Finally, I implemented the sigmoid function using a Look-Up Table (LUT), which is the simplest and one of the fastest ways of dealing with this non-linear function. This task took me about 10 hours.

Only a controller was missing in order to get the full AN. As one of the objectives of this project is making the AN as configurable as possible, the most likely way to achieve it is making the controller configurable too using constants that represent parameters of the AN as much as possible. I designed this entity to perform as many operations of each type (multiplications and additions) as inputs has the neuron. This is possible because there is a counter signal that checks that the number of operations is correspondent to the number of inputs of the neuron. I also added some input and output control signals in order to make easier the composition of a network, such as start and finish signals. The controller was ready after I dedicated 14 hours to it.

After every part of the AN was ready, I have to put everything together and add the part between the output of the MAC and the input of the LUT where the value of the signal is truncated and rounded because its width changes to 8 bits. I achieve this in approximately 15 hours.

When I finished the AN and the results of the verifying were appropriate, I started to set up a simple ANN in order to demonstrate that my neuron implementation can perform in a network. This took me about 13 hours.

The approximate amount of time dedicated to this project is 198 hours. These tasks and their breakdown according to the dedicated hours is summarized in a Gantt chart, which is attached in Annex C.

The structure of this document is the following. Chapter 2 is a description of the methods and materials that were used in this work. In chapter 3, the attributes of the different entities that take place in the neuron are presented in order to choose the fastest of each type as well as their design and implementation. In chapter 4 the experimental results of the work are introduced and discussed. Chapter 5 is an exposition of the conclusions of the project and some future lines that can be followed.

2. METHODS AND MATERIALS

In this chapter I am going to detail step by step the strategies that I adopted when developing this project. I am also going to describe the framework used in this work, as well as the hardware that intervened in it.

2.1. WORK METHODOLOGY

As I mentioned before in the introduction, after acquiring the necessary knowledge, I started looking for the components with the lowest possible latency. In first place, I begun with the parts of the MAC: multipliers and adders, and then I continued with the sigmoid function and the ROM.

Once I designed the entities in VHDL, I had to synthesize them in order to be able to program a FPGA with those algorithms. There is a synthesizer built-in in Vivado framework that not only checks if there are any inconsistencies in the code but also optimizes the implementation with different strategies. As there was not any of them that optimizes latency, I chose the default strategy. In addition to that, the synthesizer provides different analysis of the attributes of the entity, such as power consumption, area utilization, noise, methodology and timing reports. It is also possible to see a schematic of the implemented module.

This last review is the most important because it shows if the design meets the timing requirements, which are crucial in this project, also depending on which frequency the entity works at. The worst negative slacks and total negative slacks of setup time, hold time and pulse width are presented. Also, it is possible to see the number of failing endpoints over the total. Another interesting feature is that it is possible to check the critical path or paths of the module that limit the maximum work frequency and to what extent.

In the utilization report it is possible to check the amount of FPGA resources used and unused, such as DSPs, flip-flops (FFs), BRAMs, LUTs and input and output pins (IOs), among others.

In some entities of the project the generic DSP element embedded in FPGA is used. The advantages of using it are that they reduce latency and area utilization. These entities are the top one of the AN, which also contains the connections and controller of the system; and the DADDA multiplier. These slices are used in these particular entities because the DSP48E1 slice contains a 48-bit accumulator and a 25 × 18 two's-complement multiplier, among other elements. In these cases, the following lines must be implemented in the entity header:

```
attribute use_dsp48 : string;   -- use DSP Slices when synthesizing
attribute use_dsp48 of Artificial_Neuron : entity is "yes";
attribute use_dsp48 : string;   -- use DSP Slices when synthesizing
attribute use_dsp48 of dadda_multi : entity is "yes";
```

I also introduced some input and output delays in the implementation to make the design more realistic:

```
set_input_delay -clock sys_clk_pin -max 0.5 [all_inputs];
set_input_delay -clock sys_clk_pin -min 0.1 [all_inputs];
set_output_delay -clock sys_clk_pin 0.5 [all_outputs];
```

The first two lines mean that in every input signal of the top module of the system will be delayed for a period bounded between 0.1 and 0.5 ns. However, the clock signal cannot be delayed, so the synthesizer automatically does not apply this order to that signal. The last line means that every output of signal of the top module of the system will be delayed for a maximum period of 0.5 ns.

I verified the different entities by the use of a VHDL test bench per each. These tests contemplate a wide range of possibilities of the input signals in order to check that the unit under test operates as it should. The test modules were computed using the simulator included in Vivado, so it is possible to check the behaviour of the middle and output signals depending on the stimuli, the different combinations of input signals.

When I found an architecture that seemed to be the lowest-latency one, I repeat the same process with another topology that is not as fast as the previous in order to make a proper comparison between them and contrast the timing results.

When I had to choose the quantification of the signals in the system I used some scripts and functions from [2] that measure the accuracy of an ANN. Those were previously modified to adapt to the precision constraints of the system. Once all the possibilities were contemplated, the quantification structure which both had the best accuracy and adapted the best to a particular network that was going to be implemented was the chosen one.

I also used some self-made MATLAB scripts when implementing the sigmoid function in order to see the picture of the function with the adjusted precision. With this plot, I could check if the applied approach was correct. In addition, this script writes out all the possible input values and their respective output values in binary base applying the VHDL methodology. This automatization made it easier to implement the entity in VHDL, because all that I needed to do was copying and pasting the result of the MATLAB script on the *.vhd* file.

When putting all the modules together I had to implement a controller that coordinates all the system. This is the most important part of the AN because it can optimize the timing distribution and achieve some level of parallelism. The process that was explained at the beginning of this section was applied again. When the AN was working properly, it was not very difficult to form a network of neurons with the previous architecture. Once again, the process of designing, implementing, verifying and analysing timing results was needed.

In addition to the verifying phase, I programmed some MATLAB scripts that emulated the behaviour of both the AN and ANN that had previously been implemented. Those worked as functional tests in order to verify if both VHDL and MATLAB got the same results. As I had these scripts running, I added some piece of code to measure execution times of the scripts in order to be able to compare with VHDL simulations

The last stage in this project was the FPGA implementation, when the ANN algorithm that had previously been implemented is loaded into the board. Moreover, some interactive features were added in order to make the results of the algorithm more visual, such as the use of switches, push buttons, displays and leds that can be found on the FPGA board.

2.2. DEVELOPMENT PLATFORM NEXYS 4 DDR

Nexys 4 DDR board is a development platform based on Artix-7 FPGA from Xilinx. This, in addition to the huge number of memories, ports peripherals and I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

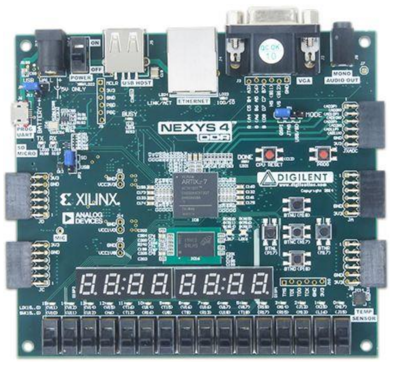


Figure 5. Nexys 4 DDR board

This board is optimized for Xilinx's Vivado Design Suite. Vivado lets its users synthesize their HDL designs, perform timing and utilization analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. It is a design environment for FPGA products from Xilinx.

3. HARDWARE SELECTION, DESIGN AND IMPLEMENTATION ISSUES

In this chapter I am going to detail the process that was described in the work methodology for each component that take place in this project. Several stages can be distinguished: search, implementation, verifying and analysing; word-length calculation; connection of the components; and implementation of the system controller.

3.1. MODULES THAT COMPOSE AN ARTIFICIAL NEURON

As I learnt in [7] and [8], an AN is a composition of some modules that perform the arithmetical operations that are necessary to get the right result. In first place, there is a multiply and accumulate operation, that can be broken down into a series of multiplications and additions. Then the result goes through a non-linear function, in this case a unipolar sigmoid function. This diagram of an AN was found in [3], while the MAC unit one was in [11].

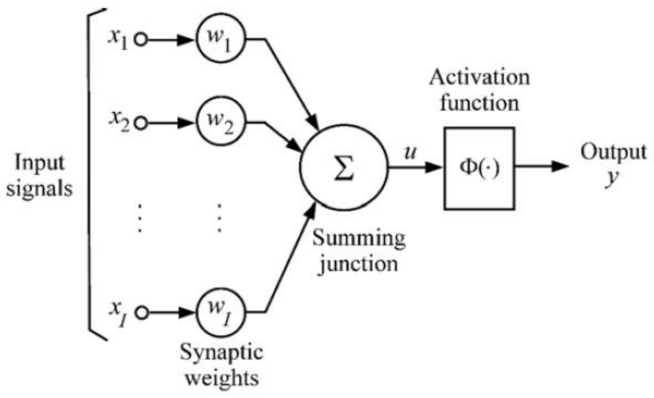


Figure 6. Behaviour of artificial neuron

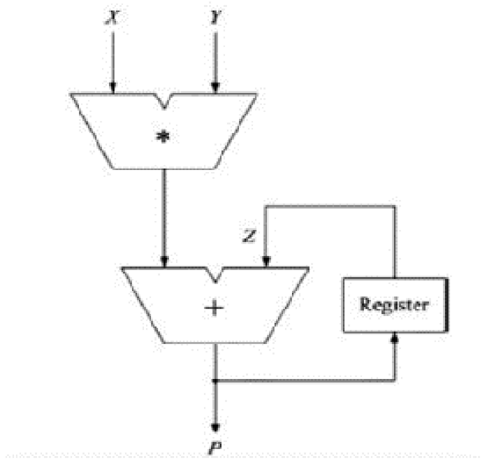


Figure 7. Design of the multiply and accumulate unit

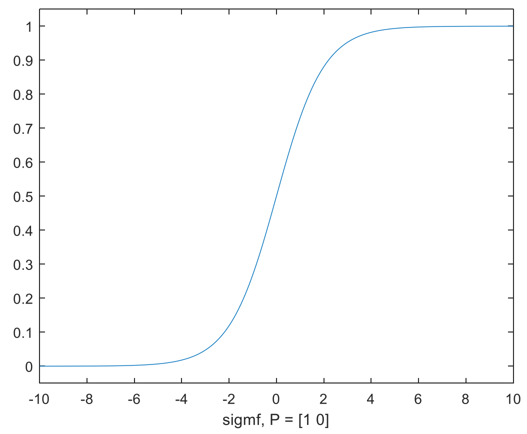


Figure 8. Sigmoid function representation

The following is a block diagram of the components of the AN, where X denotates a variable bit width of the signal.

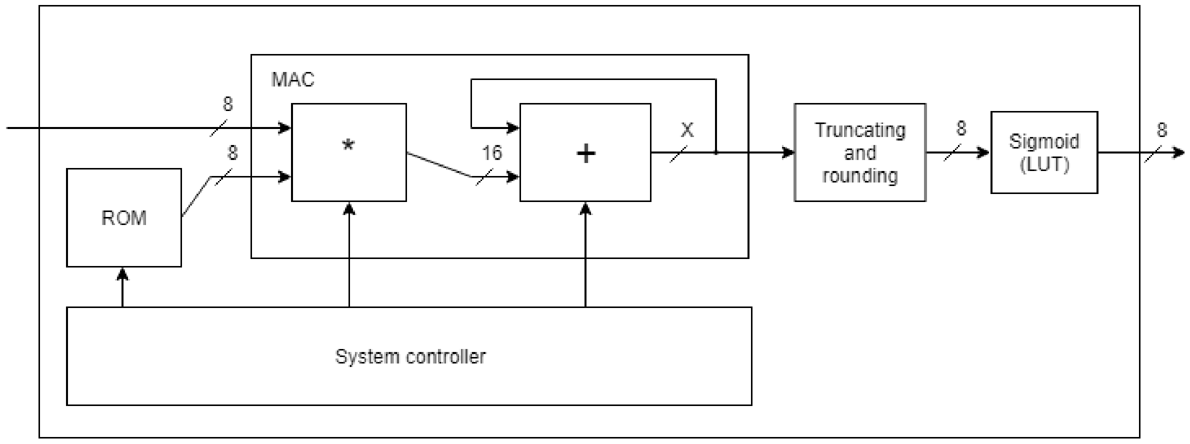


Figure 9. Block diagram of an artificial neuron

The following figure represents the different stages that are passed through in the designing process of the artificial neuron entities:

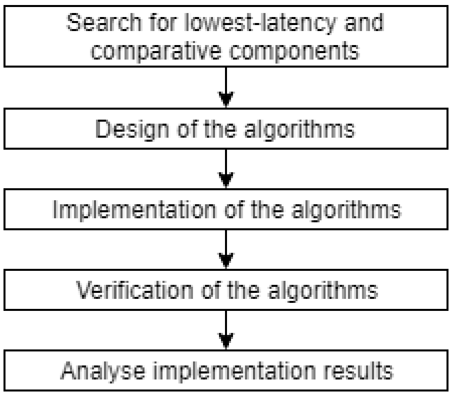
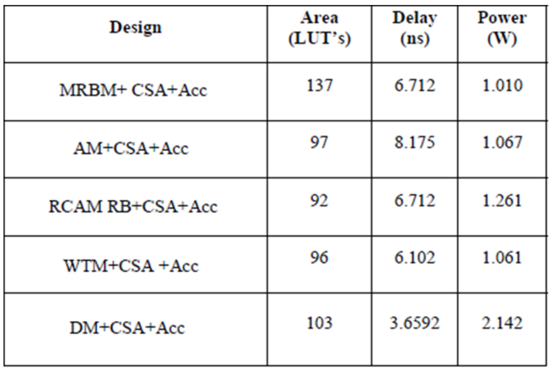

Search for lowest-latency and comparative components
Design of the algorithms
Implementation of the algorithms
Verification of the algorithms
Analyse implementation results

Figure 10. Design flow of the artificial neuron components

3.1.1. MULTIPLIER

SEARCH OF LOWEST LATENCY COMPONENT

I sought for multiplier topologies with the lowest latency in [9] and [11], and I found that the DADDA Multiplier (DM) was the faster one. It has the lowest delay compared to the rest of multiplier topologies. In order to compare delays between multipliers, I chose the one with the greatest, which was the Array Multiplier (AM). The following tables and figures were found in [11].



Design	Area (LUT's)	Delay (ns)	Power (W)
MRBM+ CSA+Acc	137	6.712	1.010
AM+CSA+Acc	97	8.175	1.067
RCAM RB+CSA+Acc	92	6.712	1.261
WTM+CSA +Acc	96	6.102	1.061
DM+CSA+Acc	103	3.6592	2.142

Table 8. Comparison of performance metrics of MAC unit models

The results presented in this table represent some MAC designs implemented using Verilog hardware description language (HDL), simulated and synthesized using Xilinx ISE 13.2 for Virtex-6 40nm technology. A Carry Save Adder and an accumulator (Acc) were used in all the MAC topologies. The different multiplier topologies that are shown in the table are Modified Radix-2 Booth Multiplier (MRBM), Array Multiplier (AM), Ripple Carry Array Multiplier with Row Bypassing Technique (RCAM RB), Wallace Tree Multiplier (WTM) and DADDA Multiplier (DM).

DESIGN OF THE ALGORITHM

The DM has two 8-bit inputs and a 16-bits output. The first input corresponds to the inputs of the AN and the second to the correspondent weight associated to that input. As there may be negative numbers, inputs with the MSB with value '1', two modules (one for each input) that perform two's complement if necessary have been included before computing the multiplication. Another one performs two's complement if necessary, at the end of the computation. It is important to emphasize that due to the non-symmetry of the two's complement, the minimum number that can be represented, which must be negative, will be transformed in two's complement to the maximum number that can be represented, which entails a certain degree of error, specifically a bit of error.

The DADDA topology consists of a multiplication (AND gates) between both inputs. Each bit of the first operand is multiplied by each bit of the second producing several partial products that are grouped as it is done when multiplying decimal numbers manually. The distribution can be seen as a table, compound of rows and columns. The number of rows must be reduced until there are left at most two of them. Then, the bits of each column are added. Reductions are controlled by a maximum-number of rows sequence d_j. This sequence starts with $d_1 = 2$ and continue by $d_{j+1} = floor\,(1.5 * d_j)$. The initial value of j is chosen as the largest value of d_j that is smaller than the shortest width between both inputs. If the number of rows of a column is lesser or equals to d_j, the column does not require any action. If it is equals to $d_j + 1$, the top two elements are added (XOR gates) placing the result at the bottom of the column and the carry at the top of the next column, then the next column is examined. Any other result needs to add (XOR gates) the top three elements placing the result at the bottom of the column and the carry at the top of the next column, then the next column is examined.

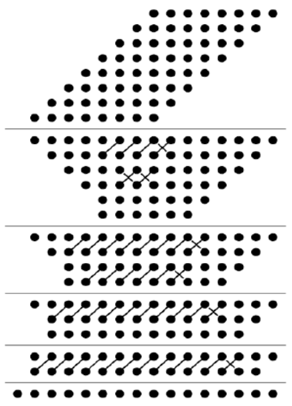


Figure 11. Behaviour of DADDA multiplier

This algorithm is shown in the diagram for an 8-bit DM. I found the code of this multiplier on the Internet [15] and I made some changes on it.

I attempted to achieve a configurable design for this multiplier, however the algorithm depends on the inputs' length and the code should contemplate as many cases as possible inputs' lengths and this would be infinite. This fact also made it impossible to make the length of the inputs, weights and output signals configurable. Because of this, we set them to an 8-bit length.

COMPARISON WITH ANOTHER COMPONENT

In order to make a proper comparison between both multipliers, the AM was also implemented. It consists of a succession of Half Adders and Full Adders, as it is shown in the diagram for an 8-bit AM.

It performs the weighted partial products (AND gates) and then they are added using FAs and HAs connected forming daisy chains.

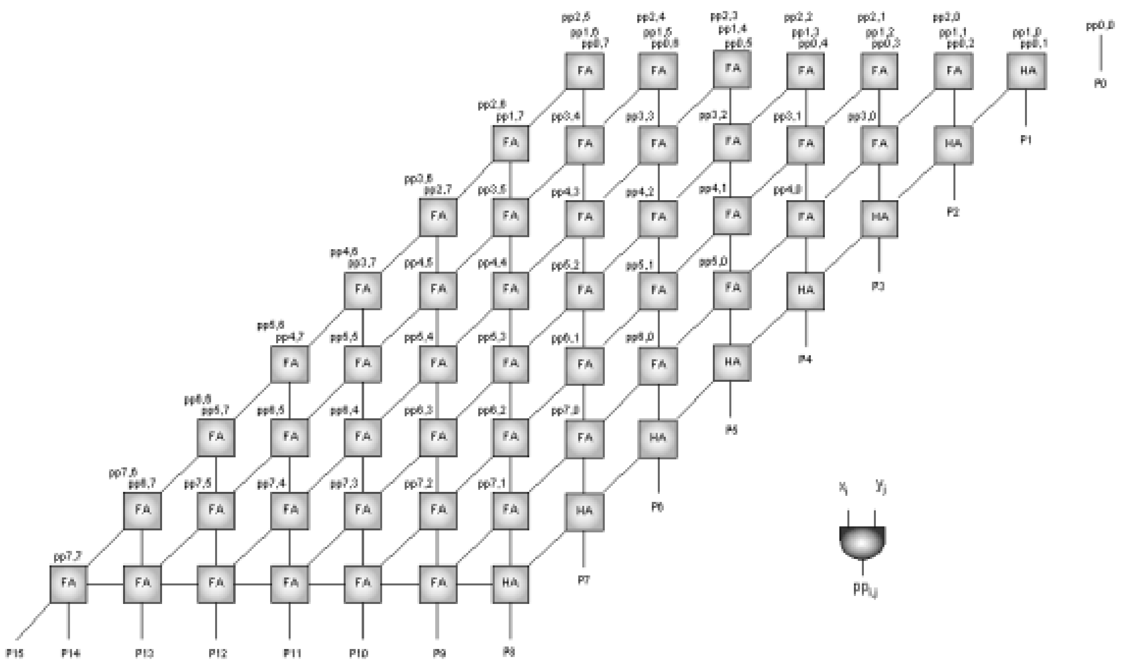
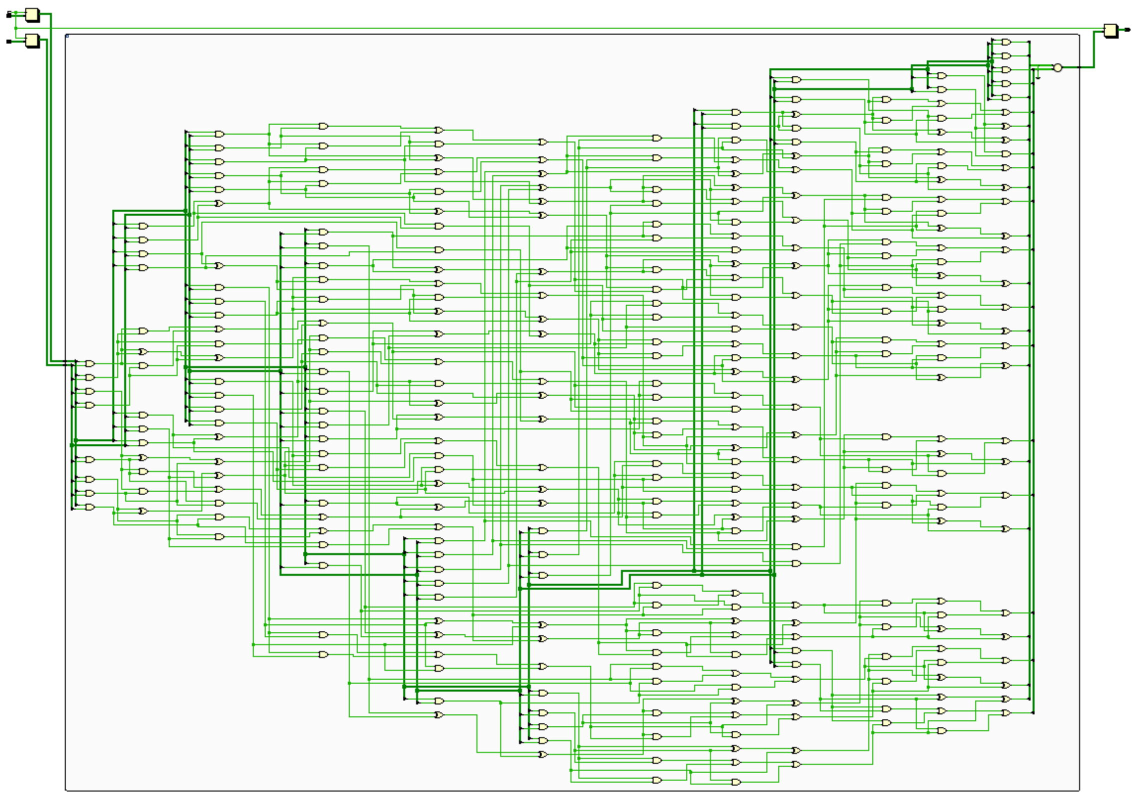


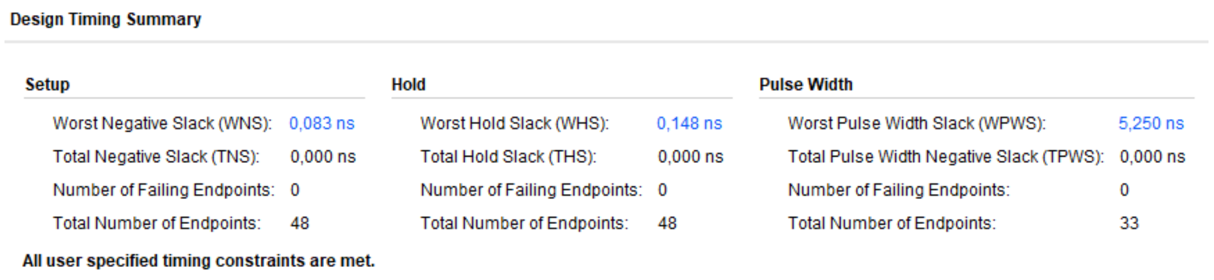
Figure 12. Schematic of Array Multiplier

IMPLEMENTATION OF THE COMPONENTS

In order to check whether the approach of the previous table was correct or not, the two modules were implemented, and the results showed that the former can work in a higher frequency (86.96 MHz ≡ 11.5 ns) than the latter (85.11 MHz ≡ 11.75 ns). The detailed pictures of the RTL schematics can be found on Annex D.



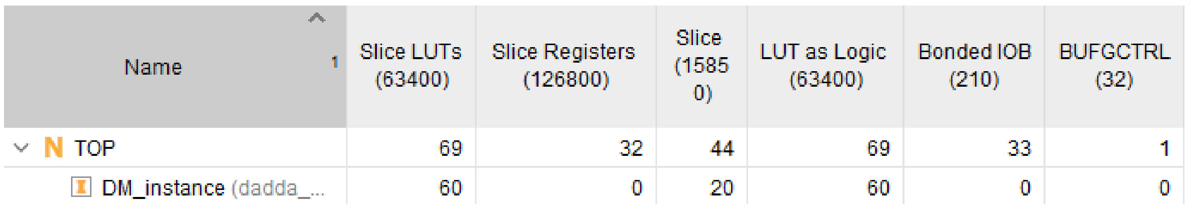
Vivado Implementation 1. DADDA multiplier: RTL schematic


Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	0,083 ns	Worst Hold Slack (WHS):	0,148 ns	Worst Pulse Width Slack (WPWS):	5,250 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	48	Total Number of Endpoints:	48	Total Number of Endpoints:	33

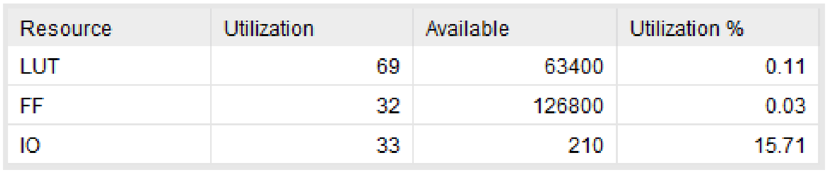
All user specified timing constraints are met.

Vivado Implementation 2. DADDA multiplier: design timing summary

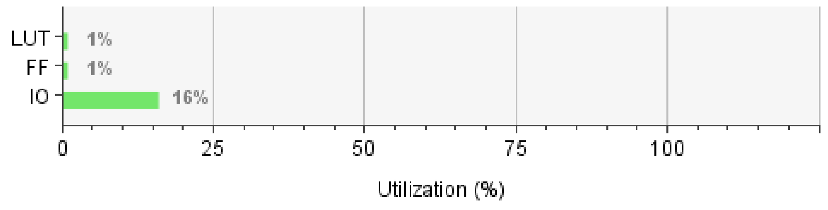


Name 1	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	Bonded IOB (210)	BUFGCTRL (32)
N TOP	69	32	44	69	33	1
DM_instance (dadda_...	60	0	20	60	0	0

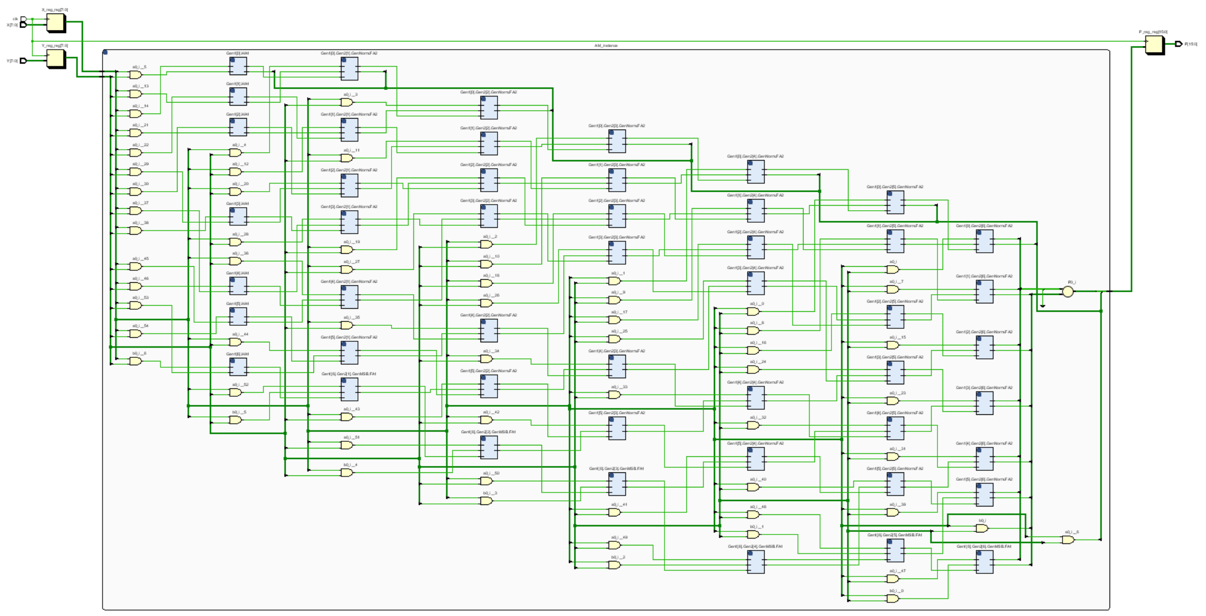
Vivado Implementation 3. DADDA multiplier: utilization hierarchy



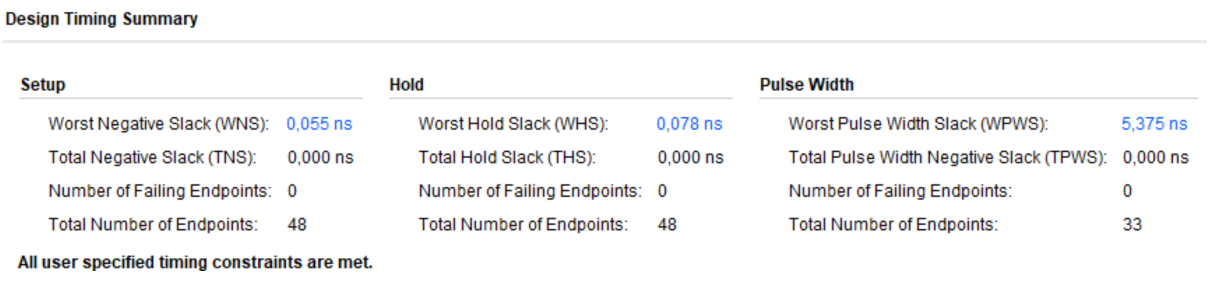
Resource	Utilization	Available	Utilization %
LUT	69	63400	0.11
FF	32	126800	0.03
IO	33	210	15.71



Vivado Implementation 4. DADDA multiplier: utilization summary



Vivado Implementation 5. Array multiplier: RTL schematic

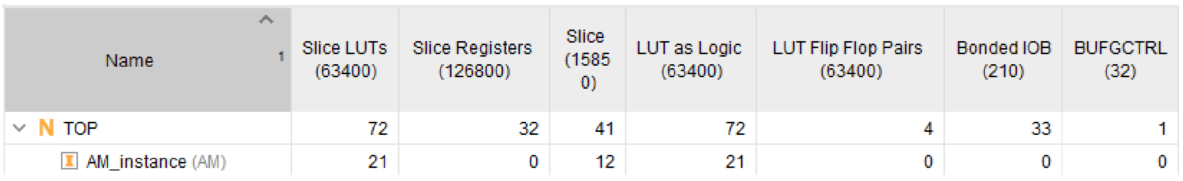


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,055 ns	Worst Hold Slack (WHS):	0,078 ns	Worst Pulse Width Slack (WPWS):	5,375 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	48	Total Number of Endpoints:	48	Total Number of Endpoints:	33

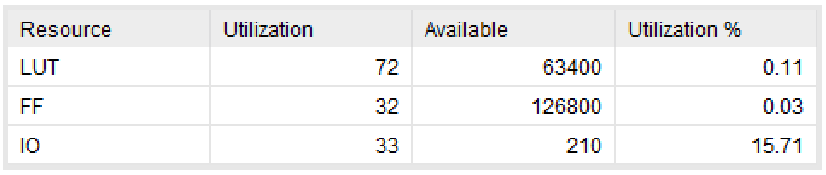
All user specified timing constraints are met.

Vivado Implementation 6. Array multiplier: design timing summary

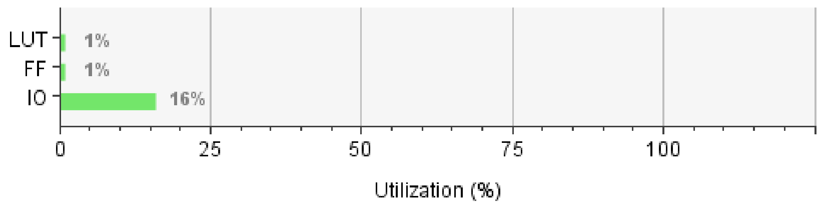


Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Bonded IOB (210)	BUFGCTRL (32)
TOP	72	32	41	72	4	33	1
AM_instance (AM)	21	0	12	21	0	0	0

Vivado Implementation 7. Array multiplier: utilization hierarchy



Resource	Utilization	Available	Utilization %
LUT	72	63400	0.11
FF	32	126800	0.03
IO	33	210	15.71



Vivado Implementation 8. Array multiplier: utilization summary

ANALYSIS OF IMPLEMENTATION RESULTS

As a result of these implementations, even though the maximum frequency at which the DM can work correctly is higher than that of the AM, its temporal slacks are greater, which means that the timing is even better than what was thought at first.

In the DM, critical paths for setup time are those from the output register to the output ports, while for hold time they are the ones from the input ports to the input registers of both inputs.

In the AM, the critical paths for both setup time and hold time are the same ones than the DM.

Furthermore, the utilization of the DM is slightly smaller than AM, which shows another advantage of using the former multiplier.

3.1.2. ADDER

SEARCH OF LOWEST LATENCY COMPONENT

I sought for adder topologies with the lowest latency in [9] and [10], and I found that the Carry Select Adder was the faster one. It has the lowest delay among the rest of adders. In order to compare delays between adders, I chose the one with the greatest, which was the Ripple Carry Adder. The following tables and figures were found in [10].

Adder topology	Gate count			Power dissipation (mW)	Area μm^2	Delay (ns)
	nMOS	pMOS	Total			
RCA	144	144	288	0.206	2214	4.208
CSaA	288	288	576	1.082	5904	2.924
CLA	136	136	272	0.312	2160	3.1
CIA	171	171	342	0.261	2793	2.880
CSkA	194	194	388	0.603	3486	3.022
CByA	186	186	372	0.459	3116	3.01
CSelA	300	300	600	1.109	6201	2.75

Table 9. Area, delay and power dissipation of adders

The results presented in this table represent some designs implemented using Verilog HDL, simulated and synthesized at 0.12µm 6metal layer CMOS technology using Microwind tool. The different multiplier topologies that are shown in the table are Ripple Carry Adder (RCA), Carry Save Adder (CSaA), Carry Look-Ahead Adder (CLA), Carry Increment adder (CIA), Carry Skip Adder (CSkA), Carry Bypass Adder (CByA) and Carry Select Adder (CSelA), this last one from now on will be named as CSA.

DESIGN OF THE ALGORITHM

The CSA is compound of a pair of daisy chains made of RCAs. One performs the addition with the carry-in bit with value '1' and the other does the same with the carry-in bit with value '0'. The carry-out bit of the RCA is the carry-in bit of the next one. Then, all the outputs of the RCAs and the last carry-out bits are multiplexed in pairs grouped by weights with the carry-in bit as the control signal. Those RCAs are designed with 4-bit inputs and outputs because this length should be a divisor of the neuron's inputs length and a 4-bit RCA is faster than an 8-bit one. An 8-bit CSA is shown in the diagram. The final width of the RCA was linked to an integer constant that was defined in the CSA package, *nbits_RCA*.

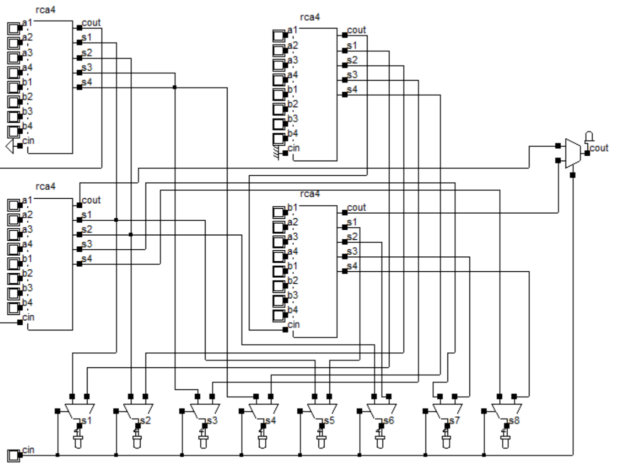


Figure 13. Schematic of Carry Select Adder

The point of this CSA is that it is part of a MAC unit, so the accumulation is achieved feeding one of its inputs with its output. The other input is connected to the multiplier output, so it is a 16-bit one. This width was associated to an integer constant that was defined in the CSA package, *nbits*.

What is more, this module must perform as many additions as inputs has the neuron. In order to achieve this, this entity has been designed in a configurable way. Because of that, the importance of using configurable design is capital and the use of variables in the design process makes it easier to understand the code. The number of inputs of the neuron is an integer signal that is inherited from the top entity of the AN, *ninputs*.

In order to avoid overflowing problems and make sure the input signal fills all the RCA input ports, the width of the output and the feedback input is set to be as long as the multiplier output width plus the number of inputs of the neuron plus a remaining factor that makes the value multiple of 4. This factor has been implemented in VHDL and linked to the integer constant *remaining* using the rem operator in the following operation *nbits_RCA - (ninputs rem nbits_RCA)*. The width of the feedback input and the output is equals to $nbits + ninputs + remaining$.

The number of RCAs that are used in the CSA can be calculated by the use of the following formula: $2 * \left\lceil \frac{nbits+ninputs}{nbits\_RCA} \right\rceil$. This value is assigned to the integer constant *nRCA*. In VHDL the way of implementing ceiling function is adding one unit to the result.

The number of MUXs that are involved in the CSA is instantiated as *nMUX* integer constant and it can be estimated using the next formula: $nbits + ninputs + remaining + 1$. This unit that is added at the end is necessary because the carry-out bit is multiplexed too.

In order to avoid sign problems, the most significant bits of the non-feedback input whose indexes are greater than 15 are filled with zeros when the number is positive or with ones if it is negative.

As this component is designed in a configurable way depending on the number of inputs of the neuron, the number of CSAs, MUXs and signals that connects these components are variable. In order to achieve this behaviour, a generate statement is used for RCAs and nested generate statement is used for MUXs in the middle stage of the implementation. The signals used in the connections come from arrays that are declared in the package, arrays of *nbits_RCA*–bit vectors for the sum signals and arrays of bits for the carry signals. There are as many carry signals as RCAs are in the entity and the number of sum signals is equals to is equal to twice the width of the output signal.

The following piece of VHDL code corresponds to the middle stage of the RCA behaviour implemented in a configurable way:

```vhdl
-- Middle stages follow the same pattern
gen_stage:
    for I in 1 to nRCA/2 - 2 generate    -- Neither the first nor the last
stages
     RCA_X_1: RCA_4b     -- carry-in bit is 1
      port map(a => addend_a_fin((nbits_RCA*I+(nbits_RCA-1)) downto
(nbits_RCA*I)),
               b => addend_b_fin((nbits_RCA*I+(nbits_RCA-1)) downto
(nbits_RCA*I)),
               cin => carry_1(I-1),
               s => sum_1(I)((nbits_RCA-1) downto 0),
               cout => carry_1(I));
     RCA_X_0: RCA_4b     -- carry-in bit is 0
      port map(a => addend_a_fin((nbits_RCA*I+(nbits_RCA-1)) downto
(nbits_RCA*I)),
               b => addend_b_fin((nbits_RCA*I+(nbits_RCA-1)) downto
(nbits_RCA*I)),
               cin => carry_0(I-1),
               s => sum_0(I)((nbits_RCA-1) downto 0),
               cout => carry_0(I));
     gen_stage_mux_med:
      for J in 0 to (nbits_RCA-1) generate
        MUX_X: mux_2to1  -- MUX to select the final value of the sum
         port map(a => sum_0(I)(J),
                  b => sum_1(I)(J),
                  ctrl => cin_input,
                  s => sum_output(nbits_RCA*I+J));
      end generate gen_stage_mux_med;
    end generate gen_stage;
```

`sum_1` and `sum_0` are the arrays of vectors that link the sum output of the RCAs with the correspondent MUXs, when carry-in bit is ‘1’ and ‘0’, respectively. `carry_1` and `carry_0` are the arrays of vectors that link the carry signal from the output of an RCA to the input of the correspondent one, when carry-in bit is ‘1’ and ‘0’, respectively. When implementing for loops, handling the variables that represent the indexes of the different signal vectors was crucial to get the right performance.

COMPARISON WITH ANOTHER COMPONENT

The RCA that is used for the comparation has the same width as the CSA in the inputs and output, so it must be compound of as many FAs as bits have the inputs connected forming a daisy chain. It is much slower than an RCA. An 8-bit RCA and a FA are shown in the diagrams.

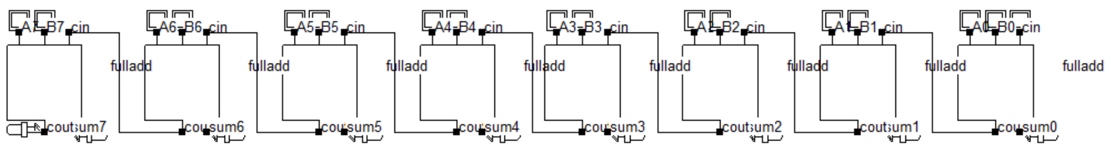


Figure 14. Design of Ripple Carry Adder

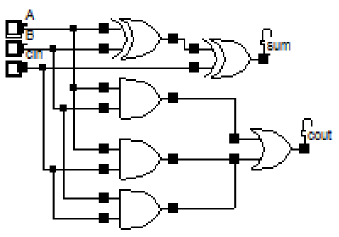
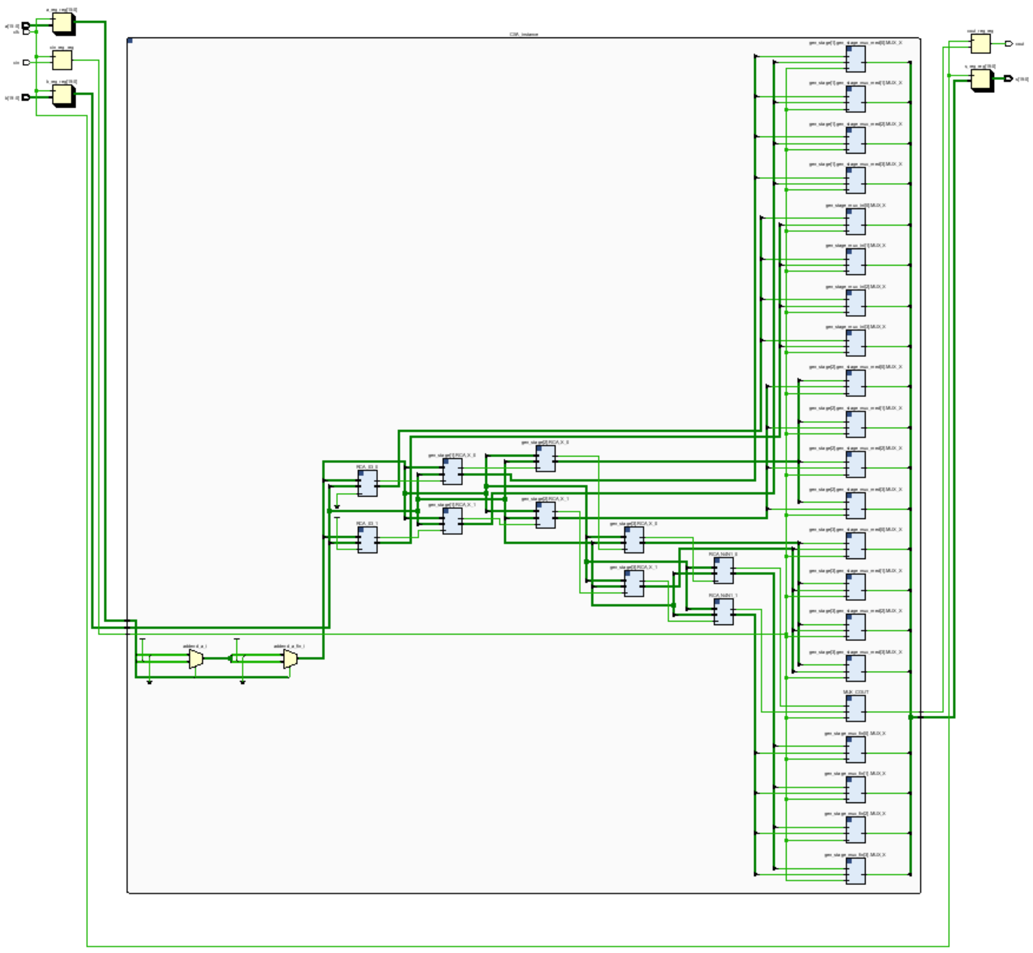


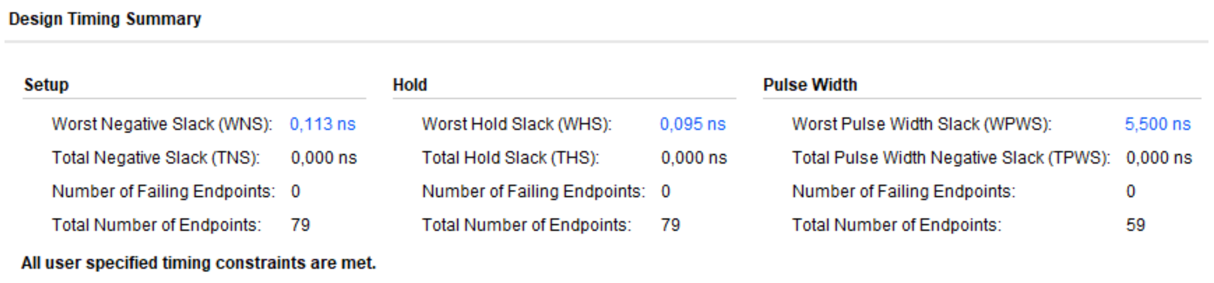
Figure 15. Design of Full Adder

IMPLEMENTATION OF THE COMPONENTS

In order to check whether the approach of the previous table was correct or not, the two modules were implemented, and the results showed that the former can work in a higher frequency (85.11 MHz ≡ 11.75 ns) than the latter (83.33 MHz ≡ 12 ns). The detailed pictures of the RTL schematics can be found on Annex D.



Vivado Implementation 9. Carry Select Adder: RTL schematic

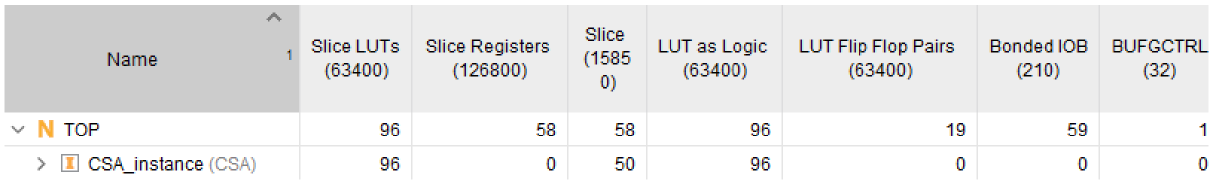


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,113 ns	Worst Hold Slack (WHS):	0,095 ns	Worst Pulse Width Slack (WPWS):	5,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	79	Total Number of Endpoints:	79	Total Number of Endpoints:	59

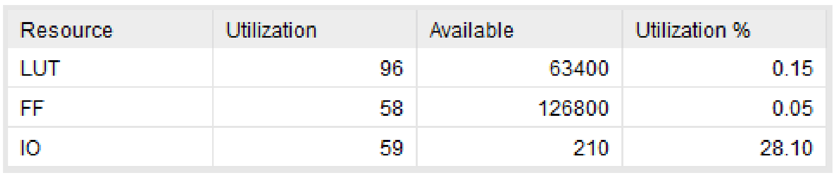
All user specified timing constraints are met.

Vivado Implementation 10. Carry Select Adder: design timing summary

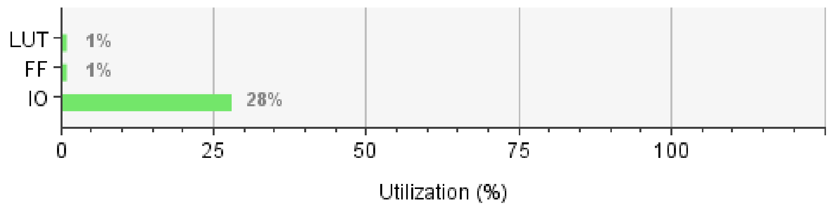


Name 1	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Bonded IOB (210)	BUFGCTRL (32)
N TOP	96	58	58	96	19	59	1
CSA_instance (CSA)	96	0	50	96	0	0	0

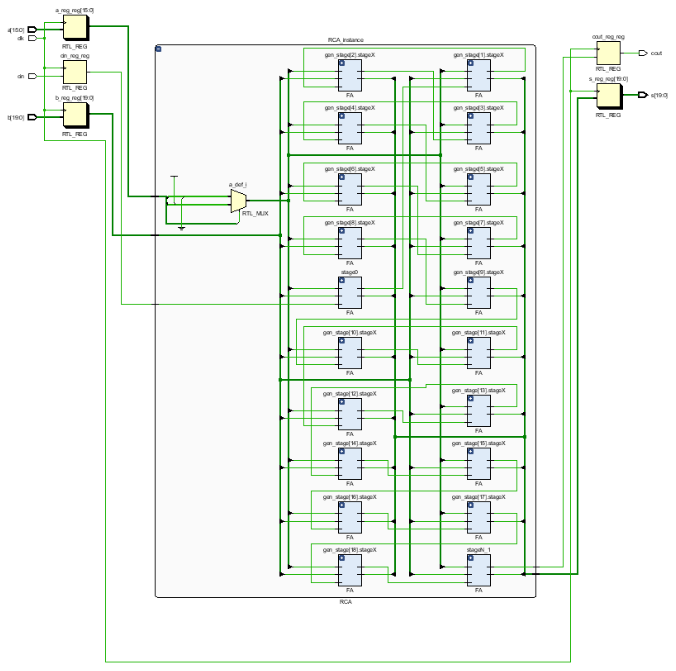
Vivado Implementation 11. Carry Select Adder: utilization hierarchy



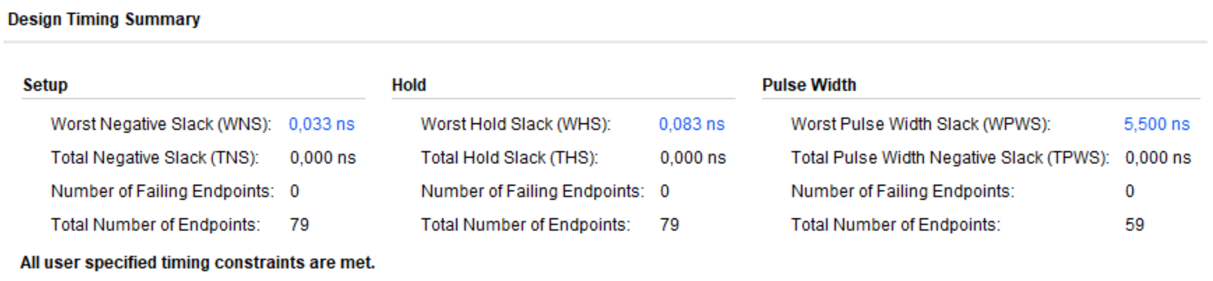
Resource	Utilization	Available	Utilization %
LUT	96	63400	0.15
FF	58	126800	0.05
IO	59	210	28.10



Vivado Implementation 12. Carry Select Adder: utilization summary



Vivado Implementation 13. Ripple Carry Adder: RTL schematic



Design Timing Summary

Setup

Worst Negative Slack (WNS):	0,033 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	79

Hold

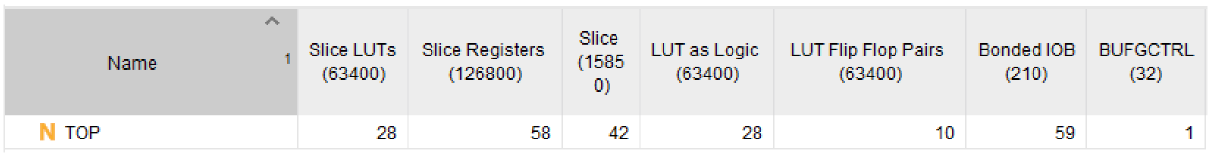
Worst Hold Slack (WHS):	0,083 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	79

Pulse Width

Worst Pulse Width Slack (WPWS):	5,500 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	59

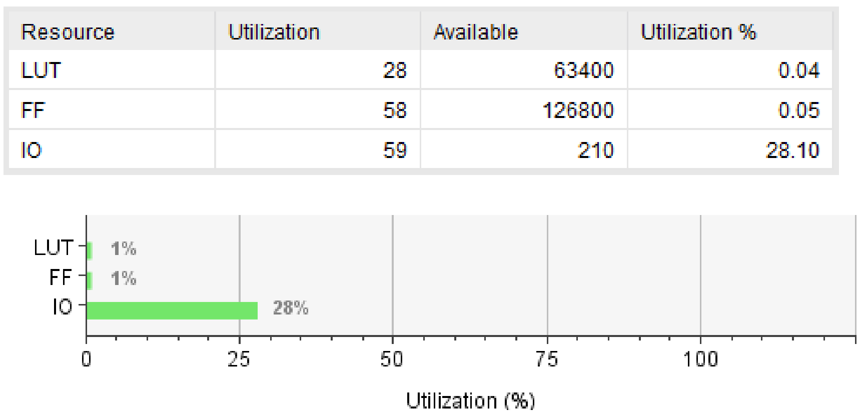
All user specified timing constraints are met.

Vivado Implementation 14. Ripple Carry Adder: design timing summary



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Bonded IOB (210)	BUFGCTRL (32)
N TOP	28	58	42	28	10	59	1

Vivado Implementation 15. Ripple Carry Adder: utilization hierarchy



Resource	Utilization	Available	Utilization %
LUT	28	63400	0.04
FF	58	126800	0.05
IO	59	210	28.10

Vivado Implementation 16. Ripple Carry Adder: utilization summary

ANALYSIS OF IMPLEMENTATION RESULTS

As a result of these implementations, even though the maximum frequency at which the CSA can work correctly is higher than that of the RCA, its temporal slacks are greater, which means that the timing is even better than what was thought at first.

In the CSA, critical paths for setup time are those from the output register to the output ports for the sum, while for hold time they are the ones from the input ports to the input registers for both inputs.

In the RCA, the critical paths for setup time are those from the output register to the output ports for both the sum and the carry-out bit, while for hold time they are the same ones than the CSA plus the carry-in bit.

Although the utilization of the CSA is a bit higher than RCA, it is not a decisive factor since the aim of this project is to implement a low-latency oriented design.

3.1.3. SIGMOID FUNCTION

In first place, the unipolar sigmoid function is a mathematical function whose most important values are concentrated around x = 0, since as it moves away from this point it presents horizontal asymptotes for y = 0 and y = 1. Its formula is: $f(x) = 1/(1 + e^{-x})$. It is represented in Figure 8.

SEARCH OF LOWEST LATENCY COMPONENT

I was looking for the fastest way to implement the sigmoid function in [13] and [14] and I found that the best way was using non-volatile, read-only memories. Those memories can be Look-Up Tables (LUTs) or Range Addressable LUTs (RALUTs). The former is a kind of ROM where every output corresponds to a unique address, whereas the latter is another kind of ROM where every output corresponds to a range of addresses in order to save up area utilization using smaller memories. The table is from [13] and the figure from [14].

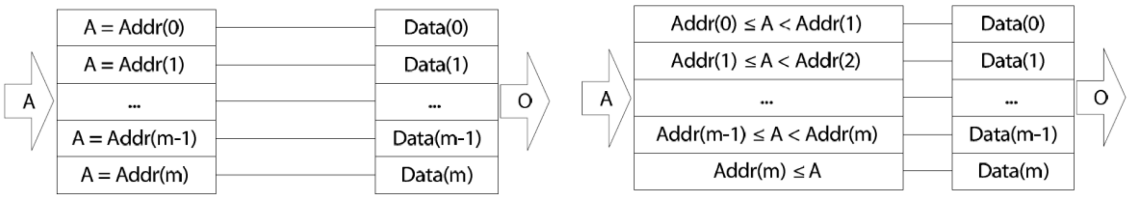
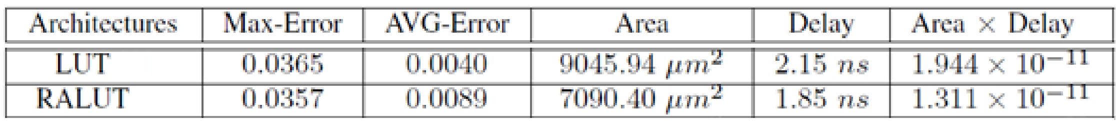


Figure 16. LUT and RALUT addressing schemes



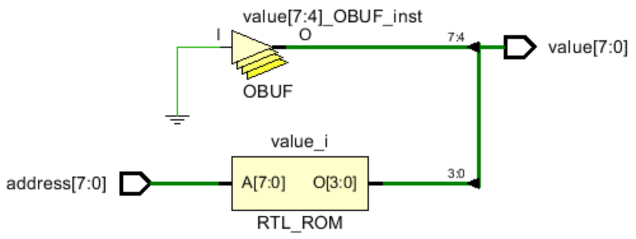
Architectures	Max-Error	AVG-Error	Area	Delay	Area × Delay
LUT	0.0365	0.0040	9045.94 μm^2	2.15 ns	1.944×10^{-11}
RALUT	0.0357	0.0089	7090.40 μm^2	1.85 ns	1.311×10^{-11}

Table 10. Complexity comparison of different implementations

The results presented in this table represent, for maximum error equals to 0.04, some designs implemented using HDL, synthesized using Synopsys' Design Compiler to library gates at a TSMC 0.18μm CMOS.

DESIGN OF THE ALGORITHM

As the neuron's input and output are 8-bit signals, $2^8 = 256$ points of the unipolar sigmoid function centred on x = 0 have to be represented. As it is far from being a constant function in its central section, which is the most important one, I decided to use a LUT instead a RALUT because it will not get any significant improve in terms of latency and memory size. It also has the advantage that all the values of both input and output of the sigmoid function can be easily written out in binary base to a plain text file by a simple MATLAB script. These values are written using the methodology of VHDL language so that it is only necessary to copy it and paste it into the code. The input values of this function are bounded between the minimum and maximum values that can be represented and with a step between each of them equals to $\frac{\max - min}{2^8 - 1}$, using MATLAB's "sigmoidal membership function", `sigmf(inputValues,[scaleFactor biasFactor])`, being the last two parameters 1 and 0, respectively. This script also plots both the sigmoid function and the sigmoid function with the precision and quantification used in the AN using MATLAB's "Construct fixed-point numeric object", `fi(value, signedness, wordLength, fractionLength)`, as it is presented in Figures 18 and 19. This component was implemented as a ROM in VHDL, using a case structure.



Vivado Implementation 17. LUT (sigmoid function): RTL schematic

3.1.4. ROM

SEARCH OF LOWEST LATENCY COMPONENT

When designing a memory in order to store the weights of the artificial neuron, the first attempt was based on a BRAM, because the FPGA got a great deal of these blocks and its implementation would be optimum. However, as the learning process is supposed to be finished, the values of weights that are stored are definitive, so there is no need for a writing port in the memory. Due to this fact, the type of memory that is selected is ROM, as there is no significant difference between using either in terms of latency.

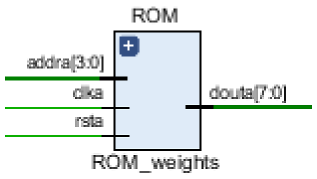
DESIGN OF THE ALGORITHM

This ROM has been implemented using the IP Catalog that can be found in Vivado because it lets the user to configure a wide range of memory characteristics. The final design consists in a Single Port ROM with a native type interface and the strategy used in the implementation is to minimize area utilization because it is the most useful option among those available. The width of the words is 8 bits and the size of the memory is 16 memory locations, which means that the AN number of inputs cannot be greater than 15, taking into account the bias neuron of the previous layer. It also has an asynchronous reset port that sets the output signal to '0' when active. The memory is initialized using a *.coe* file where the values are introduced in binary base and the remaining memory locations are filled with '0' value. The safety circuit is not implemented as it increases by a clock the read latency.

The way that the *.coe* files are programmed is the following:

```
memory_initialization_radix=2;
memory_initialization_vector=00000001 00000010 00000100;
```

The first line indicates the numerical base on which the values of the memory locations will be introduced. The second line represents the values of the memory locations from 0 up to the maximum, in the numerical base indicated in the previous line.



Vivado Implementation 18. ROM: RTL schematic

3.1.5. CONTROLLER

SEARCH OF LOWEST LATENCY COMPONENT

At first, one controller was made for the MAC (DM and CSA) and another for the global system inputs, but the combination of both made it more difficult to manage the performance of the global system and making it configurable.

The inputs and the weights were registered before entering to the MAC unit in the latter controller. Then, inputs and weights were registered again in the former one. The results of the multiplications and the additions were also registered. This repetition of the register operation added latency to the procedure, more specifically one clock signal cycle.

Then, this approach of having two controllers was put away and only the controller of the global system remains because it is the best for the system in terms of latency. And it also makes it easier to control the configurability of the controller depending on the number of inputs of the neuron. The most remarkable change that I had to make was instead of implementing a MAC unit entity (components and controller) inside the AN entity, I implemented the components of the MAC unit inside the AN entity and then coordinate the whole system with a single controller.

DESIGN OF THE ALGORITHM

This controller is based on a counter signal. It checks that the number of inputs read by the AN is equal to the number of inputs of the neuron. It also takes care that the number of operations of each type is equal to the number of entries of the neuron. The count starts when there is a rising edge of the clock signal and the start signal is read with value ‘1’. When the operations end, the count keeps its value and only in the first clock cycle after finishing the finish signal comes out with value '1'.

The behaviour of the controller is as follows: when the start input value is ‘1’ and coexists with a rising edge of the clock signal, the computation of the input data begins. The first two rising edges of the clock signal are a period when the registered signal that addresses the memory location takes the right value and get the value of the corresponding weight. In order to coordinate both multiplier inputs, two extra registers were needed between the neuron input and the multiplier input. In the third rising edge of the clock signal the first pair of multiplier inputs are registered. In the fourth, the second pair of inputs and the first multiplier output. In the fifth, the third pair of inputs, the second multiplier output and the first adder output. And so on. The number of register operations per pair of inputs is three: input register, multiplication register and addition register. Because of that, the number of rising edges of the clock signal between the start and the finish of the computation is:

$$2\ (getting\ 1^{st}\ weight) + 3\ \ (\#\ \text{register operations per pair of inputs}) + (\#\ input\ neurons - 1)$$

But the total time of operation can be measured in terms of clock signal cycles:

$$\#\ rising\ edges\ clock\ signal - 1$$

This can easily be measured in terms of units of measurement of time of the International System:

$$\#\ clock\ signal\ cycles * \ clock\ period\ (ns/cycle)$$

The detailed picture of the RTL schematics can be found on Annex D. The behaviour of the controller can be summed up in the detailed ASMD diagram that can be found in Annex E and the following pseudocode:

```
if (reset = 1)
  started = 0
  input = 0
  weight = 0
  p = 0
  s = 0
  count = 0
  address = 0
  finished = 0
else
  if (start = 1)
    started = 1
    input = inputs(0)
    weight = weights(0)
    p = 0
    s = 0
    count = 0
    address = 0
    finished = 0
  elsif ((started = 1) AND (count >= 0) AND (count < n_inputs))
    started = 1
    input = inputs(address)
    weight = weights(address)
    p = product
    s = sum
    count = count + 1
    address = address + 1
    finished = 0
  elsif ((started = 1) AND (count >= n_inputs) AND (count < (n_inputs +
n_registers)))
    started = 1
    input = 0
    weight = 0
    p = product
    s = sum
    count = count + 1
    address = address
    finished = 0
  elsif ((started = 1) AND (count = (n_inputs + n_registers)))
    started = 0
    input = 0
    weight = 0
    p = 0
    s = 0
    count = count + 1
    address = address
    finished = 1
  elsif (count > (n_inputs + n_registers))
    started = 0
    input = 0
    weight = 0
    p = 0
    s = 0
    count = count
    address = address
    finished = 0
```

3.2.FULL NEURON

As I mentioned before, an AN is a composition of some modules that perform some arithmetical operations and then the result goes through a non-linear function. But there are other elements that play an important role such as the memory where the weights are stored and the system controller.

3.2.1. CONNECTIONS

Figure 9 helps to give a global vision on this aspect. In first place, the neuron only has two kinds of input signals: control signals and input data signal. The former group is divided in three 1-bit signals: the clock signal, which governs the system; the start signal, which triggers the whole computation of the neuron; and the reset signal, which clears the processes and moves the system to an idle state. The latter is a single 8-bit signal, which takes the values of the information that has to be processed.

The other signals that are multiplied at the beginning are the weights, which are stored in a ROM and their values are specific to each neuron and the function they perform. The output of the multiplier is connected to one of the adder inputs. And the output of the adder is feeding its other input. It is also connected to LUT input, but before entering to it, the signal has to be processed. The following figure was found in [1].

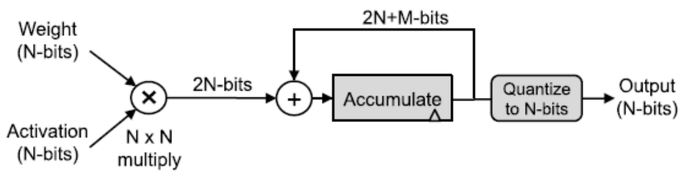


Figure 17. Reducing the precision of MAC

Both LUT's input and output are 8-bit signals, but the output of the adder has a variable width greater than 16 bits. Here is where the values of signals lose some precision, because a process of truncation and rounding is applied to the output signal of the adder. The less significant bits are those correspondents to the decimal part of the value. The MSB represent the sign. The remaining bits are the integer part. According to the precision and quantification used in the system, the MSB, the most significant bits of the decimal part and the less significant bits of the integer part are the ones that are selected in the truncation process.

If the signal represents a positive value and there is at least one bit with '1' value between those correspondents to the integer ones that are more significant than the selected ones, the signal saturates at the maximum representable value. Analogously, the signal also saturates for the minimum representable value when a negative value is represented and any of the aforementioned bits has '0' value. If none of the previous cases is given, rounding is applied.

For both positive and negative values, if the less significant bit of the selected ones has '0' value and the most significant of the non-selected decimal bits has '1' value, the former gets '1' value. Both truncation conditions are checked by the use of reduced OR and AND functions for positive and negative values, respectively. This method is the best in terms of latency. The detailed picture of the RTL schematics can be found on Annex D.

The neuron only has two output signals: the finish signal, which shows that the whole computation of the neuron has finished; and the output data, which is the output of the LUT.

3.2.2. PRECISION/QUANTIFICATION

When choosing which was the best way to quantify 8-bit signals, I used a MATLAB script that I had to implement when completing [2] assignments. This script belongs to the fourth week of the course and it calculates the accuracy of an ANN. I modified it in order to apply the precision change for all the values involved in the computation using MATLAB's "Construct fixed-point numeric object". I tried all the possible combinations that reserved the most significant bit to the sign, and the best accuracy that I obtained was assigning the MSB to the sign, only one bit to the integer part and the remaining six bits to the decimal part, because the values had been previously regularized, and they were bounded between -1.5 and 1.5.

Bits integer part	Bits decimal part	Training set accuracy (%)
1	6	97.740000
2	5	97.660000
3	4	97.460000
4	3	97.020000
5	2	95.920000
6	1	90.820000

Table 11. Training set accuracy depending on the precision

All the numbers that can be represented with that precision are bounded in the interval [-2, 1.984375], with a step between each of them of ±0.015625. The representation of the sigmoid function was the one that is shown in the next figure.

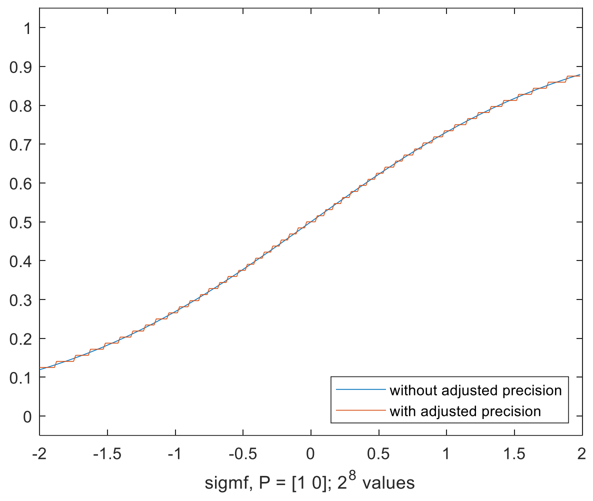


Figure 18. MATLAB representation of the unipolar sigmoid function with the first precision

But when I tried to implement a MATLAB simulation of a full ANN as it is showed in the fourth week in [2] with this precision, the weights must be scaled by a factor of 1/15 and the loss of precision caused was so great that the results of the computation were greatly affected.

Then I tried again all the combinations that I mentioned before in order to get the correct results in the ANN simulation and I chose the one that had the best results in terms of accuracy on the first script that I mentioned. That precision was assigning the MSB to the sign, four bits to the integer part and the remaining three bits to the decimal part. The weights scale factor is eight times higher, so it is more similar to the example. All the numbers that can be represented with that precision are bounded in the interval [-16, 15.875], with a step between each of them of ±0.125. The representation of the sigmoid function was the one that is shown in the figure.

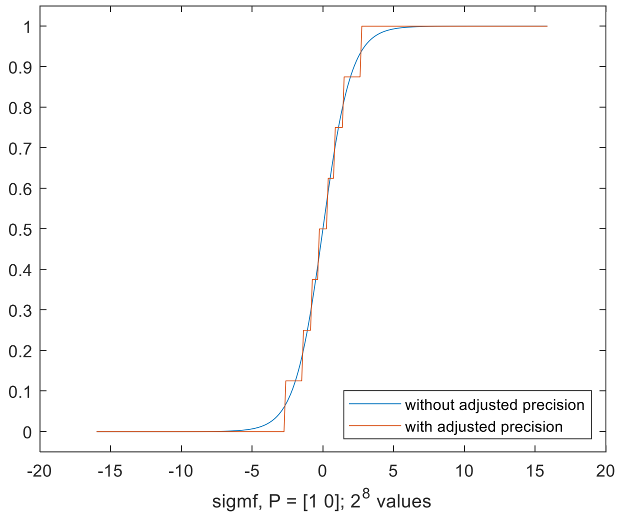
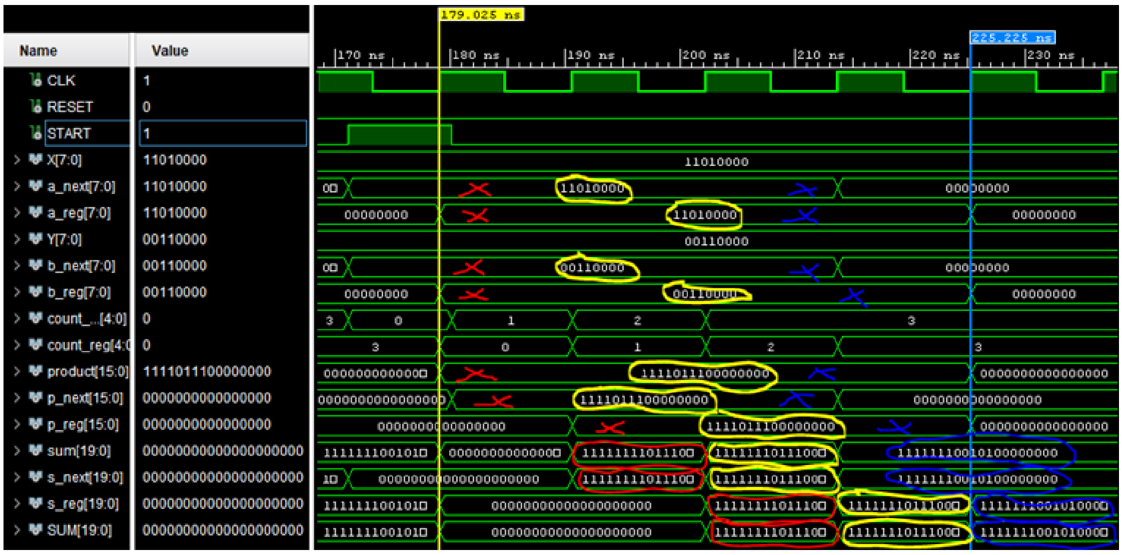


Figure 19. MATLAB representation of the unipolar sigmoid function with the final precision

3.2.3. CHRONOGRAMS

The signal propagation dataflow of the MAC unit controller in a 3-inputs AN that was mentioned before is the following:

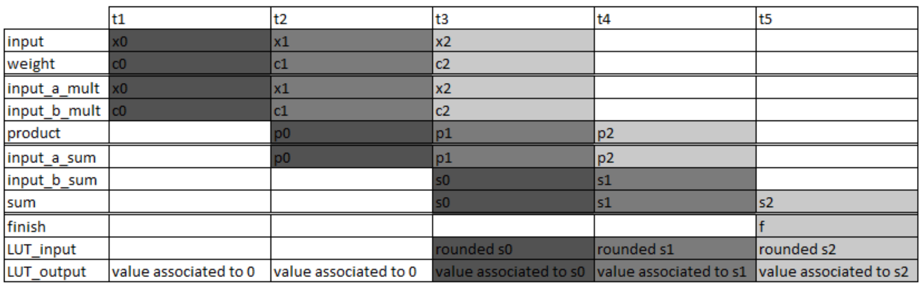


Name	Value
CLK	1
RESET	0
START	1
X[7:0]	11010000
a_next[7:0]	11010000
a_reg[7:0]	11010000
Y[7:0]	00110000
b_next[7:0]	00110000
b_reg[7:0]	00110000
count_...[4:0]	0
count_reg[4:0	0
product[15:0]	1111011100000000
p_next[15:0]	0000000000000000
p_reg[15:0]	0000000000000000
sum[19:0]	00000000000000000000
s_next[19:0]	00000000000000000000
s_reg[19:0]	00000000000000000000
SUM[19:0]	00000000000000000000

Figure 20. Vivado Simulation Dataflow of the first version of the MAC unit

After the start signal is read with value '1' at a clock signal rising edge, both MAC inputs are registered. Then the multiplication is processed, and the product is registered at the next rising edge of the clock signal. At the next one, the result of the addition is registered. But this MAC controller was discarded.

The target signal propagation diagram in a 3-inputs AN is the following:

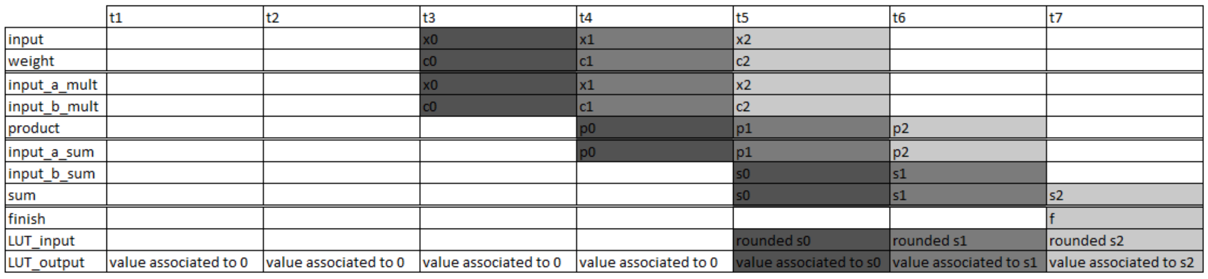


	t1	t2	t3	t4	t5
input	x0	x1	x2		
weight	c0	c1	c2		
input_a_mult	x0	x1	x2		
input_b_mult	c0	c1	c2		
product		p0	p1	p2	
input_a_sum		p0	p1	p2	
input_b_sum			s0	s1	
sum			s0	s1	s2
finish					f
LUT_input			rounded s0	rounded s1	rounded s2
LUT_output	value associated to 0	value associated to 0	value associated to s0	value associated to s1	value associated to s2

Figure 21. Target propagation diagram of the AN

The ideal behaviour is the following: after the start signal is read with value ‘1’ at a clock signal rising edge, both the input data and the weights are registered, which are the multiplier inputs. The multiplication product is registered at the next rising edge of the clock signal. At the next one, the sum is registered. The finish signal takes value ‘1’ at the clock cycle when the last sum is registered. At the same time the value of the sum is processed in order to enter to the LUT and its output goes out at the same clock cycle.

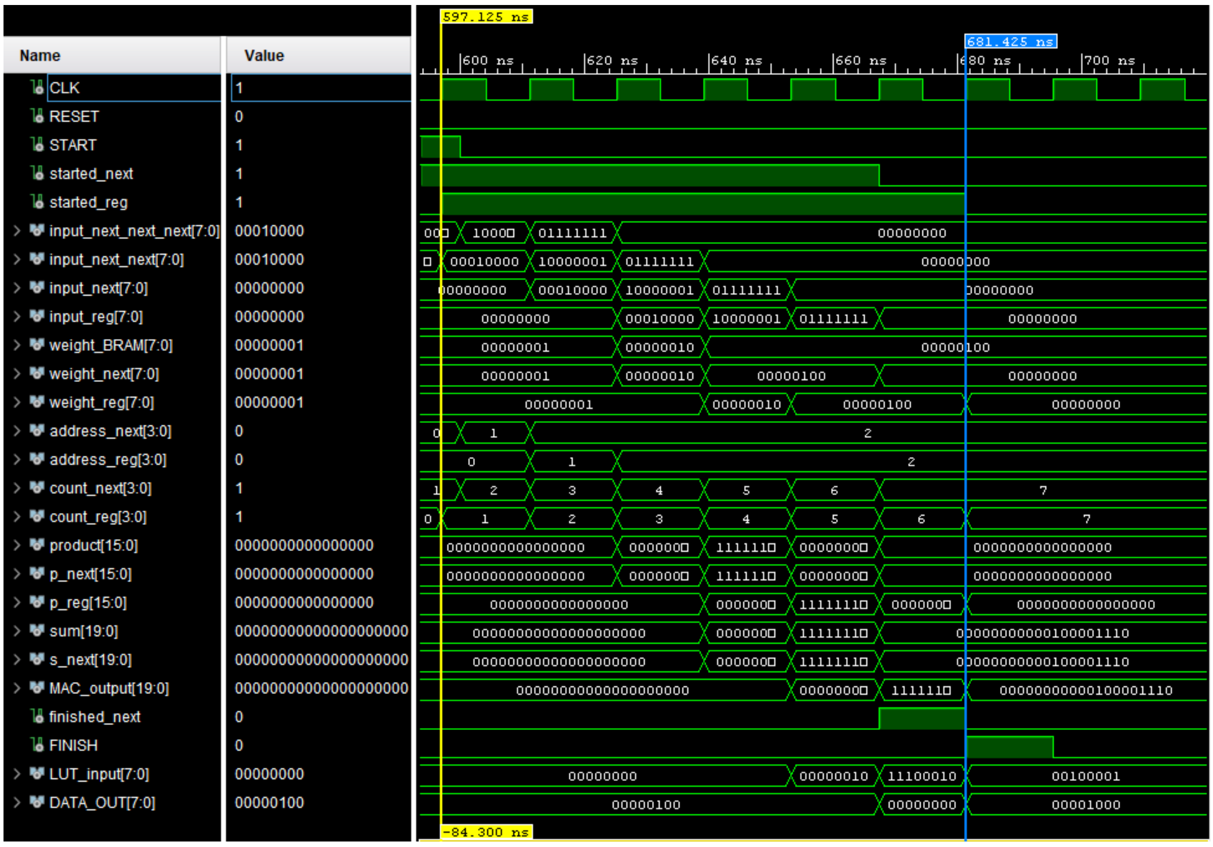
But as I mentioned before, the first two rising edges of the clock signal are a period when the registered signal that addresses the memory location takes the right value and the registered signal that goes into the multiplier gets the value of the corresponding weight. Because of that, the actual signal propagation diagram in a 3-inputs AN is the following:



	t1	t2	t3	t4	t5	t6	t7
input			x0	x1	x2		
weight			c0	c1	c2		
input_a_mult			x0	x1	x2		
input_b_mult			c0	c1	c2		
product				p0	p1	p2	
input_a_sum				p0	p1	p2	
input_b_sum					s0	s1	
sum					s0	s1	s2
finish							f
LUT_input					rounded s0	rounded s1	rounded s2
LUT_output	value associated to 0	value associated to 0	value associated to 0	value associated to 0	value associated to s0	value associated to s1	value associated to s2

Figure 22. Actual propagation diagram of the AN

The signal propagation dataflow of the global system controller in a 3-inputs AN that that evinces the previous diagrams is the following:



Name	Value
CLK	1
RESET	0
START	1
started_next	1
started_reg	1
input_next_next_next[7:0]	00010000
input_next_next[7:0]	00010000
input_next[7:0]	00000000
input_reg[7:0]	00000000
weight_BRAM[7:0]	00000001
weight_next[7:0]	00000001
weight_reg[7:0]	00000001
address_next[3:0]	0
address_reg[3:0]	0
count_next[3:0]	1
count_reg[3:0]	1
product[15:0]	0000000000000000
p_next[15:0]	0000000000000000
p_reg[15:0]	0000000000000000
sum[19:0]	00000000000000000000
s_next[19:0]	00000000000000000000
MAC_output[19:0]	00000000000000000000
finished_next	0
FINISH	0
LUT_input[7:0]	00000000
DATA_OUT[7:0]	00000100

Figure 23. Vivado Simulation Dataflow of the full version of the AN

4. RESULTS

In this chapter, I present the results of the implementation of the Artificial Neuron. I also describe a particular example of a 3-layer Artificial Neural Network, which can be made up from three different and simpler networks, and the steps needed for its implementation. I also present the results of high-level simulations of both the neuron and the network and the testing of the latter in Nexys 4 DDR FPGA.

4.1. NEURON

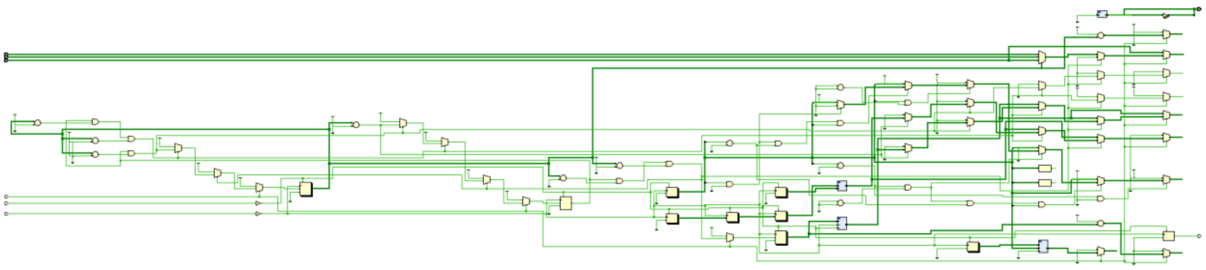
The AN is composed of a hierarchical set of entities and their respective packages. The top entity (*Artificial_Neuron.vhd*) and its package (*artificial_neuron_utils.vhd*) contain the behaviour of the global controller of the system, as well as the number precision notation and the rounding and truncating stage. Inside this module multiple entities and their packages are instantiated:

- ROM where the AN weights correspondent to the inputs are stored, it's implemented using Vivado's IP Catalog (*ROM_weights.vhd*)
- DADDA multiplier (DM) (*dadda_multi.vhd* and *dadda_utils.vhd*)
    - 8-bit two's complement converter (*Twos_complement_input.vhd*)
    - 16-bit two's complement converter (*Twos_complement_output.vhd*)
- Carry Select Adder (CSA) (*CSA.vhd* and *csa_utils.vhd*)
    - Ripple Carry Adder (RCA) (*RCA_4b.vhd* and *rca_utils.vhd*)
        - Full Adder (FA) (*FA.vhd*)
    - Two-input multiplexer (*mux_2to1.vhd*)
- Sigmoid function implemented in a Look-Up Table (LUT) (*LUT.vhd*)

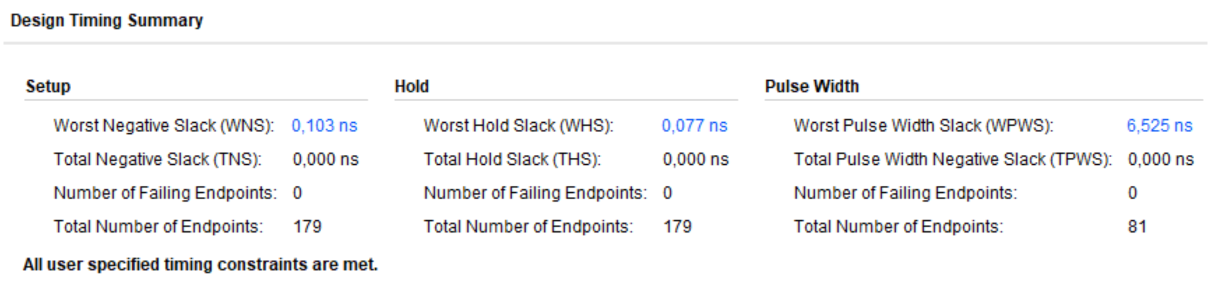
The full performance of the AN is shown in Figure 23.

4.1.1. IMPLEMENTATION

The AN was implemented, and the results showed that the maximum frequency that allows this design to work properly is lower (71.17 MHz ≡ 14.05 ns) than the maximum one of the previous entities that were implemented, such as the multipliers or the adders. Despite of that fact, the frequency is high, and this proves that the main goal of the project, a low-latency neuron, has been achieved. The detailed pictures of the RTL schematics can be found on Annex D.



Vivado Implementation 19. Artificial Neuron: RTL schematic

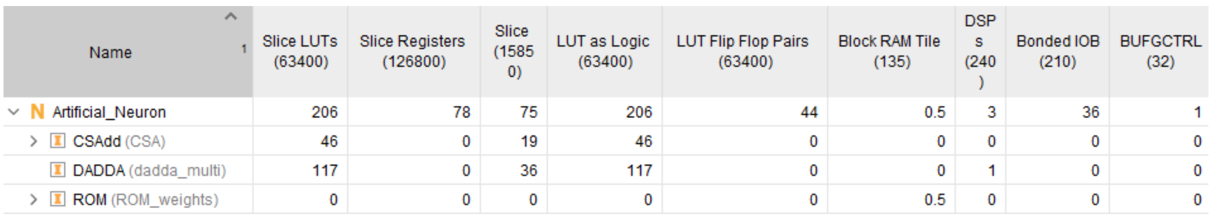


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,103 ns	Worst Hold Slack (WHS):	0,077 ns	Worst Pulse Width Slack (WPWS):	6,525 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	179	Total Number of Endpoints:	179	Total Number of Endpoints:	81

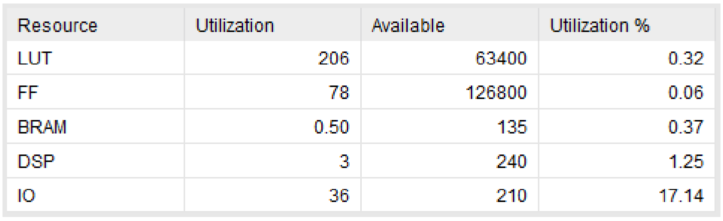
All user specified timing constraints are met.

Vivado Implementation 20. Artificial Neuron: design timing summary

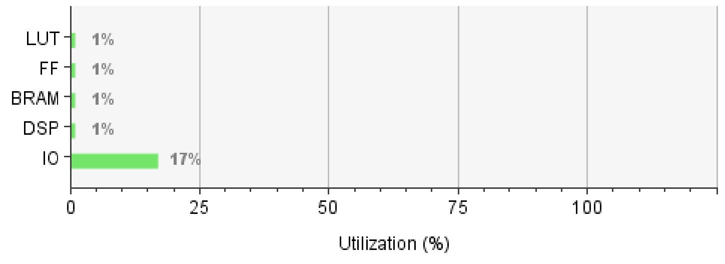


Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
Artificial_Neuron	206	78	75	206	44	0.5	3	36	1
CSAdd (CSA)	46	0	19	46	0	0	0	0	0
DADDA (dadda_multi)	117	0	36	117	0	0	1	0	0
ROM (ROM_weights)	0	0	0	0	0	0.5	0	0	0

Vivado Implementation 21. Artificial Neuron: utilization hierarchy



Resource	Utilization	Available	Utilization %
LUT	206	63400	0.32
FF	78	126800	0.06
BRAM	0.50	135	0.37
DSP	3	240	1.25
IO	36	210	17.14



Vivado Implementation 22. Artificial Neuron: utilization summary

4.1.2. ANALYSIS OF IMPLEMENTATION RESULTS

As a result of these implementations, it is important to highlight that if DSP slices had not been used the area utilization would be much larger and the maximum frequency at which the AN can work would be much lower.

Critical paths for setup time are those from the output register to the output ports of both the data and the finish signals, while for hold time they are the ones from the input ports to the input registers of the data signals.

4.2. NEURAL NETWORK

Once the AN is working properly, it is easy to shape an ANN, because the finish signals of the previous layer are the start signals of the next layer. And it is only needed a global start signal to initiate all the procedure, which connects directly with the second layer, as the first coincides with the input one. Also, the finish signal of the output layer connects with the global finish signal.

A bias neuron with value equal to ‘1’ has to be included in each layer. At the output, if the value is less than 0.5, it is rounded to 0; on the other hand, if it is greater than or equal to 0.5, it is rounded to 1.

Each AN has its own weight values associated to its entries, so as many ANs as there are in the ANN have to be implemented. All the ANs have the same structure, only the entity names and initialization files of the ROMs change. Almost any ANN can be implemented, since each neuron has a generic port with its number of inputs and its behaviour depends on that number.

A particular ANN is implemented. It was explained in [2] and computes a XNOR function. It has two 1-bit input signals (x1 and x2) and one 1-bit output signal. It can be implemented by concatenating other three binary functions:

[x1 AND x2], [(NOT x1) AND (NOT x2) = x1 NOR x2], [x1 OR x2].

The following are some possible implementations of these simple neural networks:

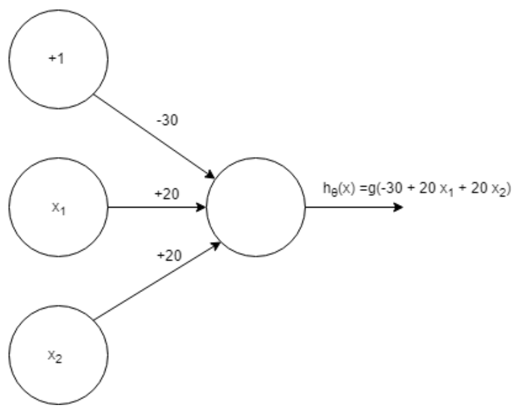


Figure 24. AND function ANN

x_1	x_2	x_1 AND x_2
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Table 12. AND function truth table

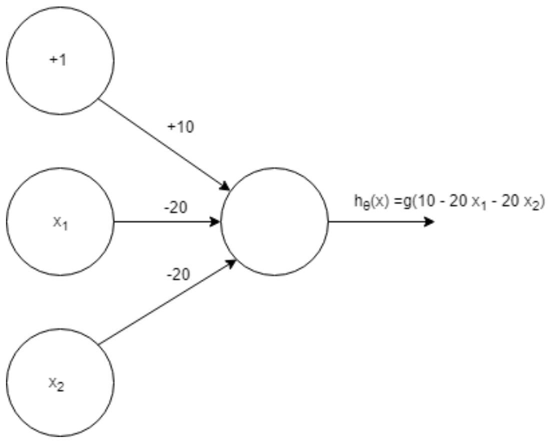


Figure 25. NOR function ANN

x_1	x_2	NOT(x_1) AND NOT(x_2) = x_1 NOR x_2
0	0	$g(10) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(-30) \approx 1$

Table 13. NOR function truth table

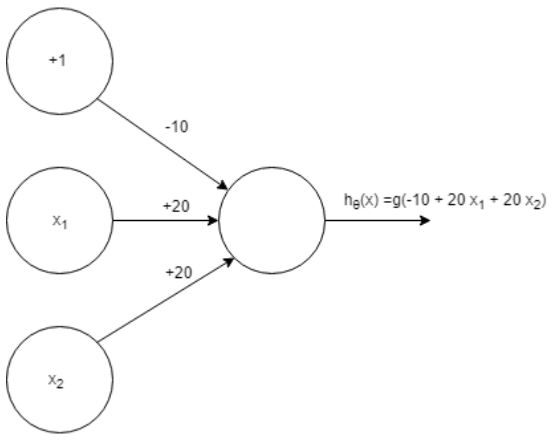


Figure 26. OR function ANN

x_1	x_2	x_1 OR x_2
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 0$
1	0	$g(10) \approx 0$
1	1	$g(30) \approx 1$

Table 14. OR function truth table

The concatenation of the three previous ANNs (functions) produces a new ANN (XNOR function).

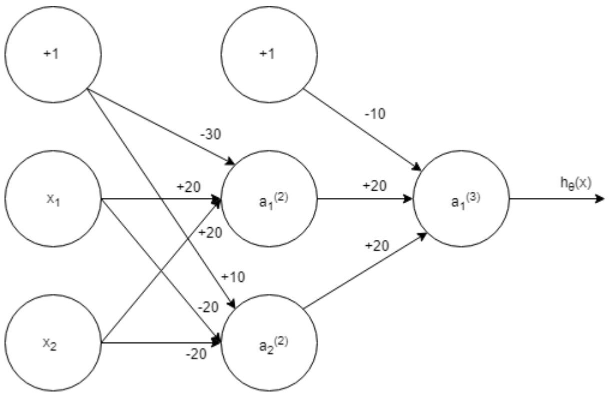


Figure 27. XNOR function ANN

x_1	x_2	$a_1^{(2)}$ (AND)	$a_2^{(2)}$ (NOR)	$a_1^{(3)} = h_\theta(x)$ (OR)
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Table 15. XNOR function truth table

In order to be able to represent these weights with the smallest possible scaling factor and to obtain the correct result, the precision had to be changed, as it was mentioned before. The final scale factor was $8/15 = 0.5\hat{3}$. The scaled values are: $-30 \rightarrow -16 \rightarrow -16$; $|20| \rightarrow |32/3| = |10.\hat{6}| \rightarrow 10.5$ and $|10| \rightarrow |16/3| = |5.\hat{3}| \rightarrow 5.25$.

Regarding to the previous information about neuron behaviour and considering the inputs as the input layer, the number of rising edges of the clock signal between the start and the finish of the computation is:

$$(\#\, rising\ edges\ clock\ signal/neuron) * (\#\, layers - 1)$$

But the total time of operation can be measured in terms of clock signal cycles:

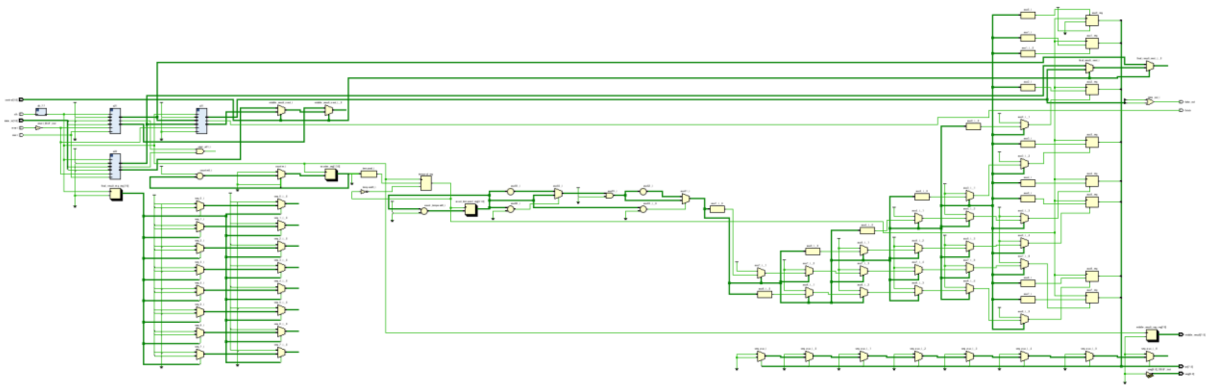
$$\#\, rising\ edges\ clock\ signal - 1$$

This can easily be measured in terms of units of measurement of time of the International System:

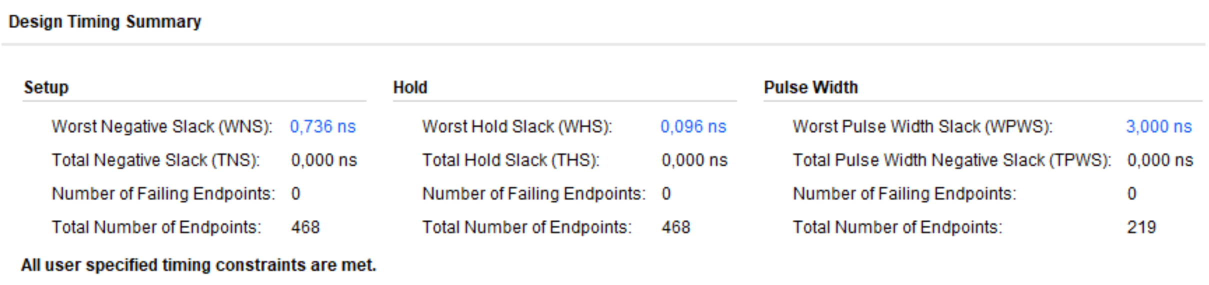
$$\#\, clock\ signal\ cycles * \ clock\ period\ (ns/cycle)$$

4.2.1. IMPLEMENTATION

The ANN was implemented, and the results showed that the maximum frequency that allows this design to work properly is much lower (50 MHz ≡ 20 ns) than the maximum one of the AN. Despite of this, the design needs only a few clock cycles to complete its computation, so the result of the implementation meets the requirements of the project. The detailed pictures of the RTL schematics can be found on Annex D.



Vivado Implementation 23. Artificial Neural Network: RTL schematic

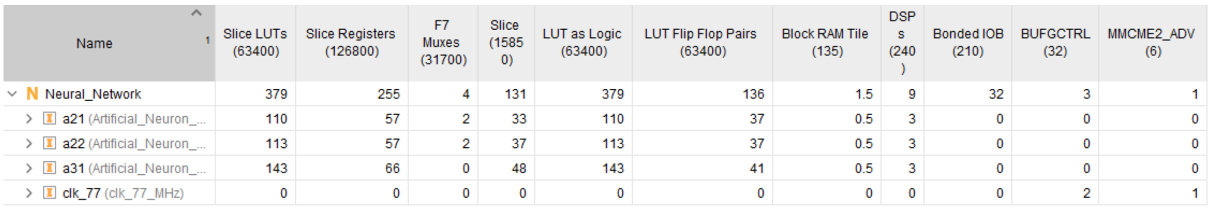


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,736 ns	Worst Hold Slack (WHS):	0,096 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	468	Total Number of Endpoints:	468	Total Number of Endpoints:	219

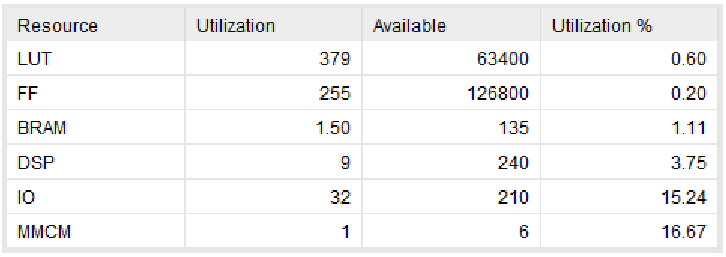
All user specified timing constraints are met.

Vivado Implementation 24. Artificial Neural Network: design timing summary

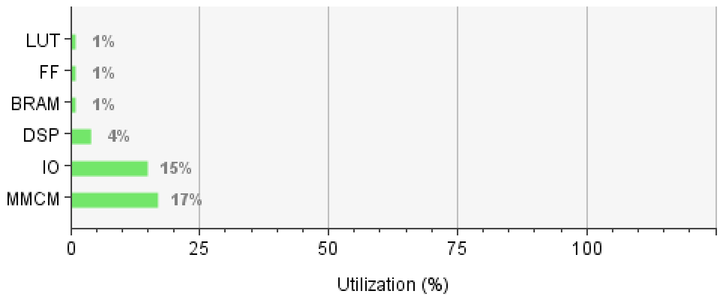


Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)	LUT Flip Flop Pairs (63400)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
Neural_Network	379	255	4	131	379	136	1.5	9	32	3	1
a21 (Artificial_Neuron_...	110	57	2	33	110	37	0.5	3	0	0	0
a22 (Artificial_Neuron_...	113	57	2	37	113	37	0.5	3	0	0	0
a31 (Artificial_Neuron_...	143	66	0	48	143	41	0.5	3	0	0	0
clk_77 (clk_77_MHz)	0	0	0	0	0	0	0	0	0	2	1

Vivado Implementation 25. Artificial Neural Network: utilization hierarchy



Resource	Utilization	Available	Utilization %
LUT	379	63400	0.60
FF	255	126800	0.20
BRAM	1.50	135	1.11
DSP	9	240	3.75
IO	32	210	15.24
MMCM	1	6	16.67



Vivado Implementation 26. Artificial Neural Network: utilization summary

4.2.2. ANALYSIS OF IMPLEMENTATION RESULTS

As a result of these implementations, it is important to highlight that if DSP slices had not been used the area utilization would be much larger and the maximum frequency at which the ANN can work would be much lower.

Critical paths for setup time are those from the output registers to the output ports of both the data and finish signals, while for hold time they are the ones from the registers of the address signal of two of the neurons to the input port of its corresponding ROM.

4.3. HIGH-LEVEL SIMULATIONS

ANN and AN test benches (*Artificial_Neuron_tb.vhd* and *Neural_Network_tb.vhd*) were implemented and simulated using Vivado.

The AN simulation dataflow is shown in Figure 23. In this simulation the input data values were in decimal base `[2 -15.875 15.875]` (`[00010000 10000001 01111111]` in binary base) and their respective weights were `[0.125 0.25 0.5]` (`[00000001, 00000010,` `00000100]` in binary base). The result of the operative was 4.21875, which is equivalent to 4.125 in the precision used in the system (`00100001` in binary base). The final result of the computation is `1.0` after the sigmoid function.

In that figure, it can be noticed how the signals flow between registers. First in the inputs, from *data_in* to input registers and the weights from the ROM to weight register. Then, the product and the sum registers. It can also be perceived how the value is rounded and truncated at the MAC output and sent to the LUT with the sigmoid function values, before getting the result of the computation. It is important to highlight that how both external (reset, start and finish) and internal (started, address and count) control signals work can be recognized too.

The following figure presents the signal dataflow of the previous ANN, where both the input and output data signals of each AN can be distinguished and also their start and finish signals. It is important to notice how the finish and *data_out* signals work, because they are the output signals of a layer, but also the input signals of the next one.

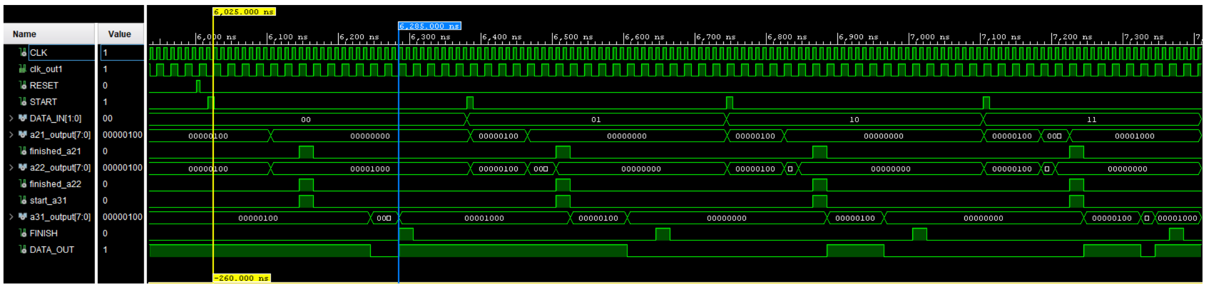


Figure 28. Vivado Simulation Dataflow of the ANN

Functional tests were also implemented in MATLAB for both the computation of a 3-input AN and the ANN, using the same values utilized in the previous simulations. The conditions in which the execution time has been measured are the following: in an Asus S510UA BR409T PC with 8 GB of installed RAM and Intel® Core™ i5 8250U processor, which is up to 3.4 GHz; running Microsoft Windows 10 Home as its operating system.

The average elapsed time in the execution of the AN script was 0.042338 seconds, while the FPGA implementation only needs 84.3 nanoseconds using the fastest possible clock signal. This fact shows that the implementation in VHDL is more than 500,000 (502,230.1305) times faster than the implementation in MATLAB.

On the other hand, the average elapsed time in the execution of the ANN script was 0.072501 seconds, while the FPGA implementation only needs 168.35 ns using the fastest possible clock signal. This also reveals that the implementation in VHDL is more than 400,000 (430,656.3707) times faster than the implementation in MATLAB.

These timing results demonstrate that the main aim of the project was achieved: a low-latency perceptron that computes much faster than a mathematical processing software, such as MATLAB.

4.4. TEST IN FPGA

The previous ANN is implemented in the FPGA for its test. As it has a 2-bit signal of input data, two switches that the FPGA features are used to perform that function. The ANN also has two 1-bit control signals (reset and start), that provides functionality to two different buttons. The clock signal that the system uses is the internal clock of the FPGA, which has a 100 MHz frequency. This clock signal is transformed at the input of the system into another one of a lower frequency (50 MHz) so that the ANN timing constraints are met. Both 1-bit signal of output data and 1-bit control finish signal come out as different LEDs on the FPGA, as well as the 8-bit signal that goes into the LUT.

In order to show more appealing evidence of the behaviour, the displays are used to show the different outputs of the different ANs. There is only one display active at the same time, but they have a refresh rate of 500 Hz so that the human eye cannot perceive their blinking. In order to activate the displays, it is necessary to control the 8 anodes and the 7 segments of which they are integrated. These former 8- and 7-bit signals are active-low. There are another two different switches that perform the task of controlling the output and the LUT input signal of which AN appears on the displays and in the leds, respectively.

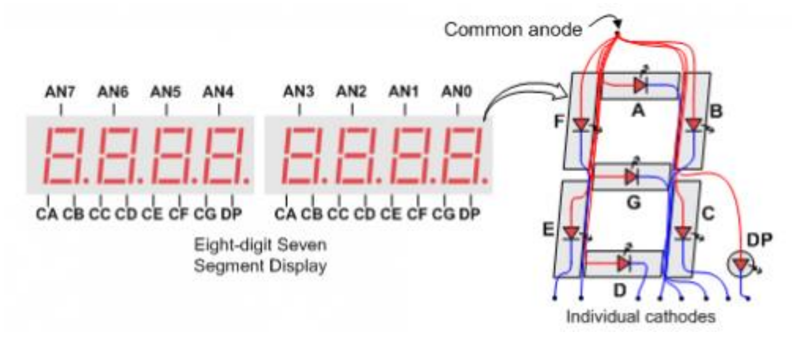


Figure 29. Seven segment displays: common anode circuit node

All the code used in this project is uploaded in my GitHub: https://github.com/sromerop/ArtificialNeuron

5. CONCLUSIONS AND FUTURE LINES

In this chapter, I present the conclusions of this project, my personal evaluation and future lines that can be followed in order to continue or improve the work I have done.

5.1. CONCLUSIONS

Nowadays, the implementation of artificial neural networks is crucial in order to process tons of data that are obtained from the millions of electronic devices of all around the world. This area is in continuous development because it is important to improve speed and performance to meet the constraints and requirements that are established. Because of that FPGAs are the most complete way to implement these algorithms, as they present a great trade-off between power dissipation, area utilization and parallelism.

These facts made an FPGA the best choice to implement the artificial neuron and the network made up with them. The architecture that has been developed and presented in this project has a great level of parallelism that makes the implementation very efficient. It has also been designed in a configurable way that allows the user to create lots of different neural networks for a wide range of purposes. But the most important aspect of the architecture that has been developed is the strategy that has been followed, all the parts of the neuron, and therefore of the network, are those that introduce the lowest possible latency. All of these allow the design of the AN to work at a fairly high frequency (71.17 MHz ≡ 14.05 ns) and to need only few clock cycles (depending of the number of inputs of the neuron) to finish its computation. Analogously, the ANN also operates at a relatively high frequency, but less than that of an isolated neuron (50.00 MHz ≡ 20.00 ns) and only needs few cycles (depending on the number of layers of the network and the number of inputs of the neurons that form these layers) to finish its computation. All these timing achievements are proved by the implementation and run of simulations and functional tests. These also show that a speed-up of hundred thousand times is achieved by the VHDL implementation in regard to the MATLAB one.

To sum up, I achieved all the proposed goals of this project. Although, this work can be used to improve the architecture that I proposed or to develop a particular application. Because of that, some lines for future approach are identified.

5.2. PERSONAL EVALUATION

During the documentation of this project, I have been able to learn about some different machine learning problems and algorithms, but I have focused my interests in artificial neural networks. So, I learned the methodologies that are required in order to achieve better results in this area.

I followed a strict low-latency strategy in order to select the parts of the neuron with their fastest possible topologies. I designed a specific architecture regarding to the particular components that had to be implemented in it and coordinated by a controller that exploits node parallelism, and then mapped it into a specific FPGA board. In the design process, there were some problems specially when designing configurability using generic parameters because it was tough to work with so many nested for loops and some arrays of vectors.

To conclude, after this project, I acquired deep knowledge about machine learning and neural networks in particular. And also, has made me improve my skills in VHDL and in programming in general and also discover some features and applications included in the Vivado framework.

5.3. FUTURE LINES

In the proposed design, node parallelism is achieved, but there is another level of parallelism that can also be reached: weight parallelism. As only latency is being considered, one possible future approach can be implementing more than one entity of each type in order to reduce much more the latency.

Implementing as many multipliers as inputs has the neuron would be a superb way to reduce latency greatly, because all the multiplications could be computed at a time in the same clock cycle. Then, all the partial products of the multipliers could be grouped in pairs in order to be added or be added altogether. After that only remains to pass the sum through the sigmoid function. The disadvantage of this design is that it would greatly increment the area utilization of the FPGA, and it would not be possible to implement as many ANs in the same FPGA as using the design developed in this project. But as in this project latency is the only attribute that matters, it could achieve huge improvements. It would be possible because this new topology would cut down the number of clock cycles required to finish the computation of ANs, and therefore of the ANNs.

Another possible approach is programming the ANs in order to be able to process the training algorithm when they form an ANN. The previous stage to backpropagation algorithm is the forward propagation stage, which is the algorithm that has been implemented in this project. But then, the difference between the output and the expected output, the error is propagated from the output layer to the first hidden layer in order to adjust the values of the corresponding weights of the neurons. It would need some major changes in the global controller as well as replacing ROMs for RAMs, implementing subtractors and the derivative of the activation function in every neuron. This modification would make the system more independent, but I believe it would be slightly slower.

6. BIBLIOGRAPHY

[1] V. Sze, Y.-H. Chen, T.-J. Yang and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," in *Proceedings of the IEEE*, vol. 105, 2017, pp. 2295-2329.

[2] A. Ng, "Coursera: Machine Learning," [Online]. Available: https://www.coursera.org/learn/machine-learning.

[3] A. R. Omondi and J. C. Rajapakse, "FPGA Neurocomputers," in *FPGA Implementations of Neural Networks*, New York, NY, USA, Springer, 2006.

[4] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," in *arXiv preprint arXiv:1705.06963*, 2017.

[5] S. Himavathi, D. Anitha and A. Muthuramalingam, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," in *IEEE Transactions on Neural Networks*, vol. 18, 2007, pp. 880-888.

[6] Google, "Google Scholar," [Online]. Available: https://scholar.google.es/.

[7] A. Gomperts, A. Ukil and F. Zurfluh, "Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications," in *IEEE Transactions on Industrial Informatics*, vol. 7, 2011, pp. 78-89.

[8] N. Izeboudjen, A. Farah, S. Titri and H. Boumeridja, "Digital implementation of artificial neural networks: from VHDL description to FPGA implementation," in *International Work-conference on Artificial Neural Networks*, Berlin, 1999, pp. 139-148.

[9] A. P. Chandrakasan, W. J. Bowhill and F. Fox, "High-Speed VLSI Arithmetic Units: Adders and Multipliers," in *Design of High-Performance Microprocessor Circuits*, Wiley-IEEE press, 2000.

[10] U. Ramadass, V. Vijayan, M. Mohanapriya and S. Paul, "Area, delay and power comparison of adder topologies," in *International Journal of VLSI Design & Communication Systems*, vol. 3, 2012, pp. 153-168.

[11] M. S. Kumar, D. A. Kumar and P. Samundiswary, "Design and Performance Analysis of Multiply-Accumulate (MAC) Unit," in *Circuit, Power and Computing Technologies (ICCPCT)*, 2014, pp. 1084-1089.

[12] Xilinx, *Block Memory Generator v8.4: LogiCORE IP Product Guide, Vivado Design Suite*, 2017.

[13] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu and M. Ahmadi, "Efficient Hardware Implementation of the Hyperbolic Tangent Sigmoid Function," in *IEEE International Symposium on Circuits and Systems*, 2009, pp. 2117-2120.

[14] K. Leboeuf, A. H. Namin, R. Muscedere, H. Wu and M. Ahmadi, "High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function," in *Third 2008 International Conference on Convergence and Hybrid Information Technology*, 2008, pp. 1070-1073.

[15] F. Oliveira Carvalho, "GitHub: An 8x8 Dadda Multiplier implemented in VHDL," 14 October 2011. [Online]. Available: https://github.com/philix/dadda_mult.

ANNEX A: ETHICAL, ECONOMIC, SOCIAL AND ENVIRONMENTAL ASPECTS

A.1 ETHICAL IMPACT

As in almost every machine learning application, there are many ethical dilemmas in their use. Machine learning and specifically artificial neural network applications could be used to perform some jobs that are currently done by people, such as a taxi driver being replaced by a self-driving car.

The ethical issue behind this fact is that, as the machines where these algorithms will be implemented will not have morality nor feelings, depending on the different tasks that they carry out some problems could appear. For example, there is some risk when they are working with sensitive information.

A.2 ECONOMIC IMPACT

As mentioned before, one of the main objectives of this project is to implement a low-latency artificial neuron as generic and reusable as possible, so that many artificial neural networks can be implemented with a faster design.

Having done so, this project could be very useful for a wide range of applications and the user only has to supply some parameters depending on the networks that has to be created.

These aspects can save the user an important amount of time and money, because the network would be already designed and implemented.

Even though artificial neural networks are widely used, they are still an unusual topic. This means that only few people with very specific training and knowledge are able to implement a network with very specific requirements and constraints, so a specialist must be hired in order to complete the task. Knowing that the average salary of an engineering specialist is around 40€/hour, this means that if the engineer needs a month to design, implement and test the artificial neural network, working 5 hours per day, with an average of 20 working days per month, the user would spend around 4,000 € in paying the engineer.

A.3 SOCIAL IMPACT

This project can cause an important social impact because the neural networks applications are implemented in day-to-day tools. As it is said in the introduction, artificial intelligence is getting more and more important in some life aspects.

The area where the impact is more distinguishable is medical applications. Pattern detection can help to prevent or allow early detection of some serious illnesses, such as cancer or hereditary genetic diseases.

As this project is oriented to low-latency, it could have an enormous impact on timing, allowing being used in real time data processing.

Some other areas where this project could cause some high social impact could be image and video recognition, self-driving cars, search & voice-activated assistants, speech and language recognition and translation, etc.

A.4 ENVIRONMENTAL IMPACT

As the artificial neurons and artificial neural networks are implemented in FPGAs and not in ASICs, they have some advantages. They can be used in many different applications or reprogrammed in order to make some changes in the software, such as to improve its power efficiency.

Because of these facts, this project will make a smaller impact in the environment and also will consume less, helping to reduce the global carbon footprint.

ANNEX B: ECONOMIC BUDGET

After a necessary learning period, the daily amount of time dedicated to this project is 4 hours. The average number of days worked per month is 24 and the project took four months to be finished. This gives a total number of hours of 384.

In order to test and implement the AN and the ANN, a FPGA board Nexys 4 DDR was bought, with a price of 230 €.

During the development stage, the software was created using Vivado and MATLAB. While both frameworks usually need a fee for their full use, the cost was zero because of some agreements that the companies that distribute the software made with Universidad Politécnica de Madrid.

After all the computations taking into account all these factors and the computer that was used, the total budget is 8,872.12 €.

	Hours	Price/hour	TOTAL
LABOUR COST (direct cost)	384	15.00 €	**5,760.00 €**

MATERIAL COST (direct cost)

	Acquisition cost	Use in months	Amortization in years	TOTAL
Personal computer (software included)	1,500.00 €	6	5	150.00 €
Nexys 4 DDR	230.00 €	6	5	23.00 €
Licenses	- €	6	5	- €

TOTAL	**173.00 €**

OVERHEAD COSTS (indirect cost)	15%	over DC	**889.95 €**
INDUSTRIAL BENEFIT	6%	over DC+IC	**409.38 €**

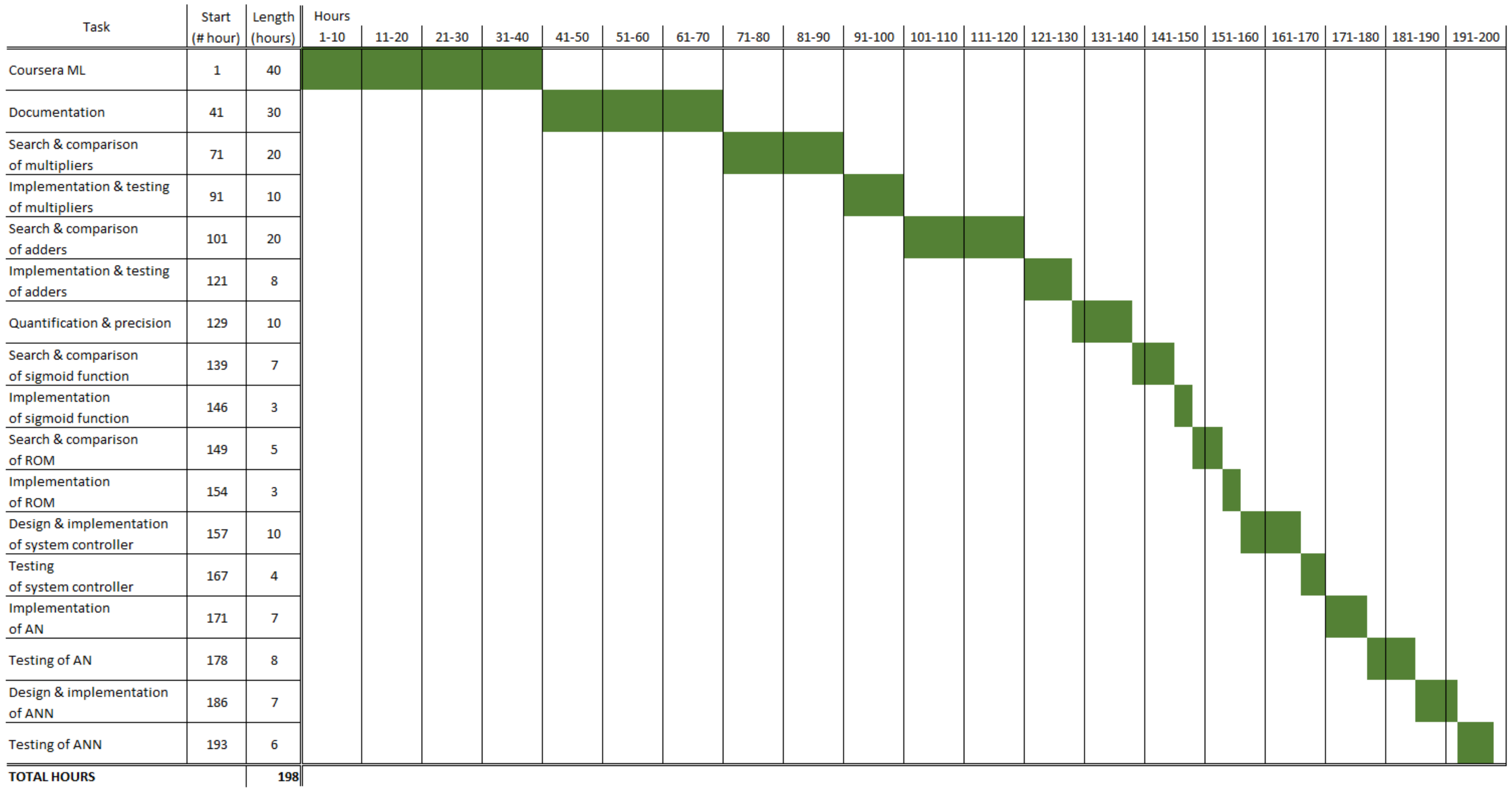
EXPENDABLE MATERIALS

Printing	**100.00 €**

SUBTOTAL BUDGET		**7,332.33 €**
VAT ENFORCEABLE	21%	**1,539.79 €**

TOTAL BUDGET	**8,872.12 €**

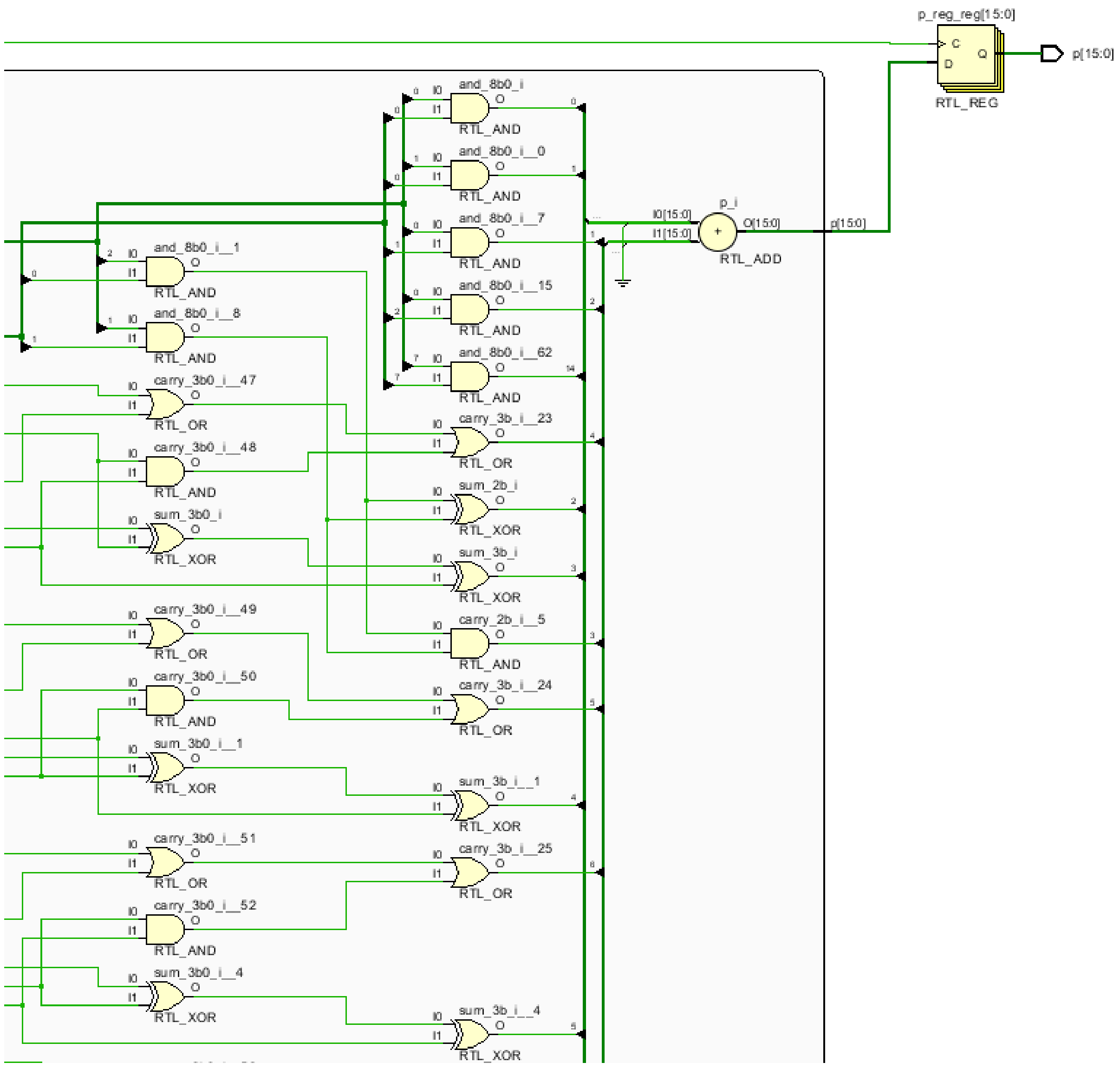
ANNEX C: GANTT CHART



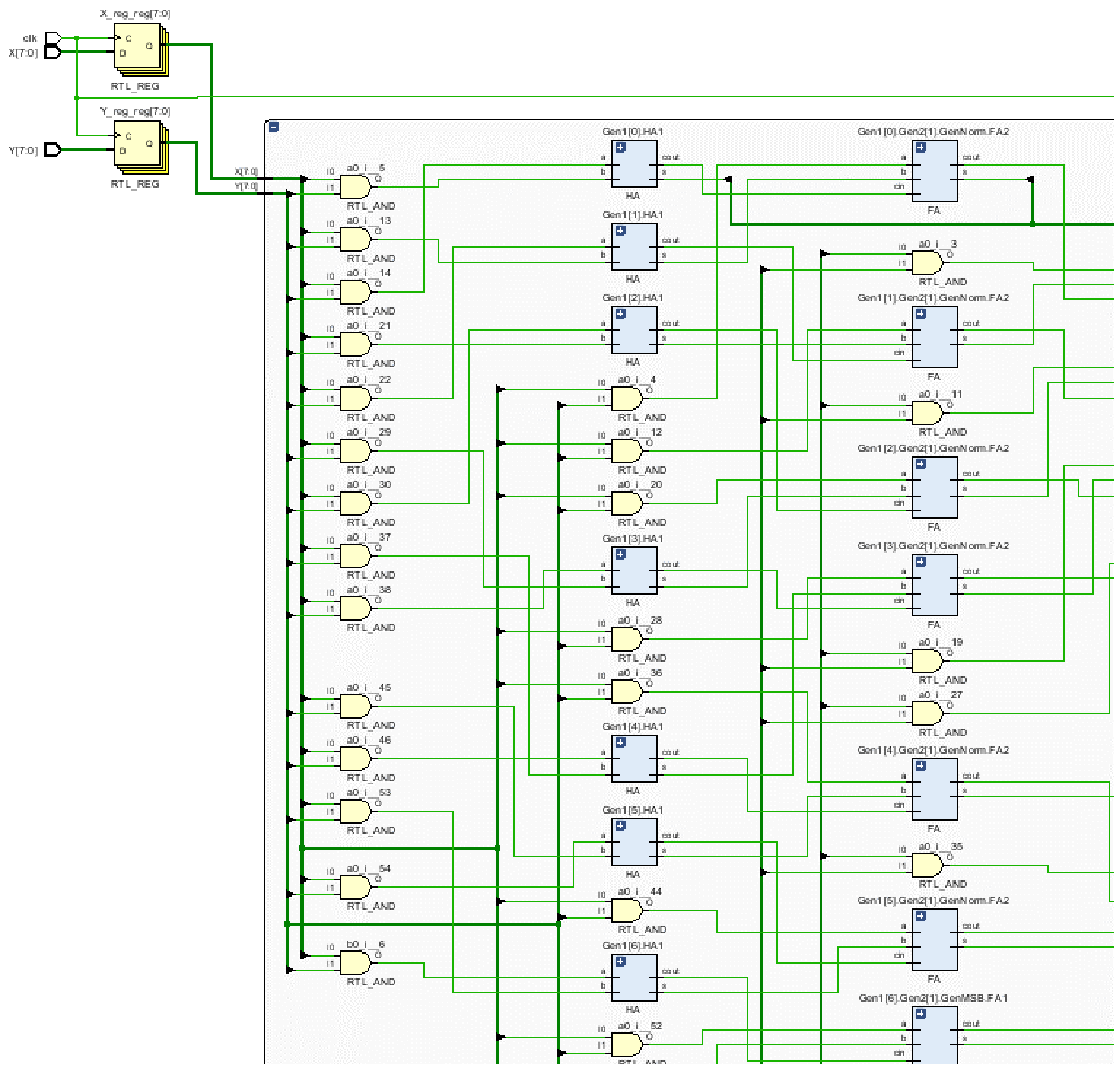
Task	Start	Length	Hours																			
	(# hour)	(hours)	1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	101-110	111-120	121-130	131-140	141-150	151-160	161-170	171-180	181-190	191-200
Coursera ML	1	40	🟩	🟩	🟩	🟩																
Documentation	41	30					🟩	🟩	🟩													
Search & comparison of multipliers	71	20								🟩	🟩											
Implementation & testing of multipliers	91	10										🟩										
Search & comparison of adders	101	20											🟩	🟩								
Implementation & testing of adders	121	8													🟩							
Quantification & precision	129	10													🟩	🟩						
Search & comparison of sigmoid function	139	7														🟩	🟩					
Implementation of sigmoid function	146	3															🟩					
Search & comparison of ROM	149	5															🟩	🟩				
Implementation of ROM	154	3																🟩				
Design & implementation of system controller	157	10																🟩	🟩			
Testing of system controller	167	4																	🟩			
Implementation of AN	171	7																		🟩		
Testing of AN	178	8																		🟩	🟩	
Design & implementation of ANN	186	7																			🟩	🟩
Testing of ANN	193	6																				🟩
TOTAL HOURS		**198**																				

Figure 30. Gantt chart: tasks and dedication

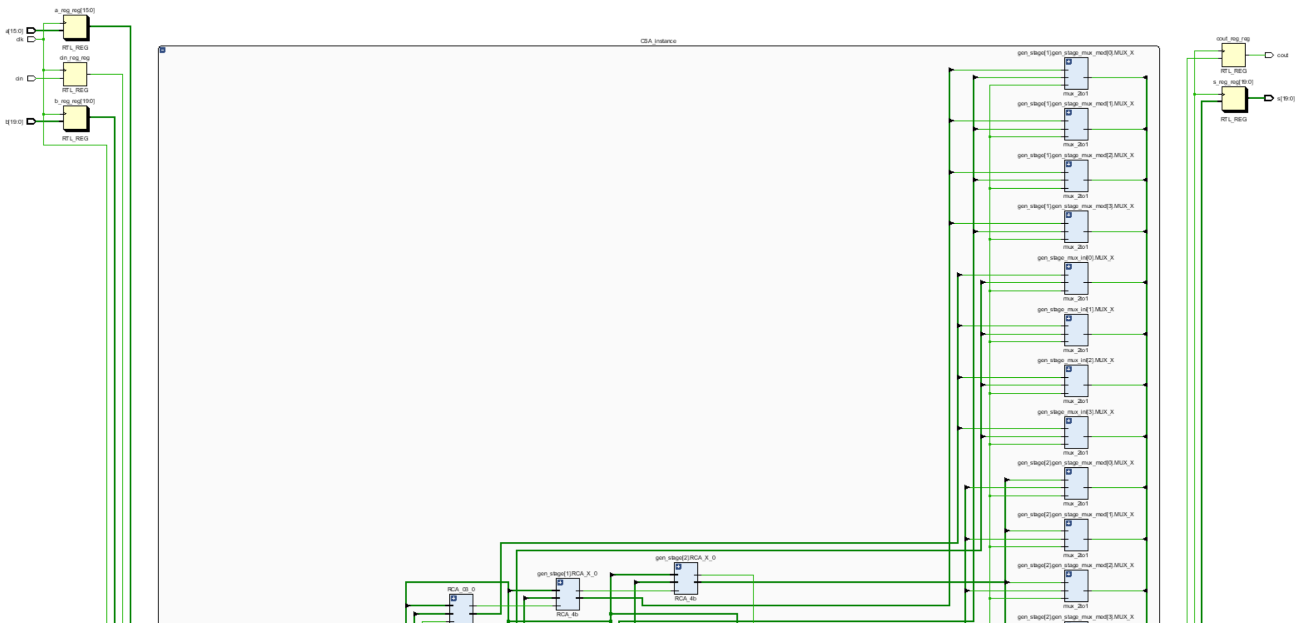
ANNEX D: DETAIL OF RTL SCHEMATICS



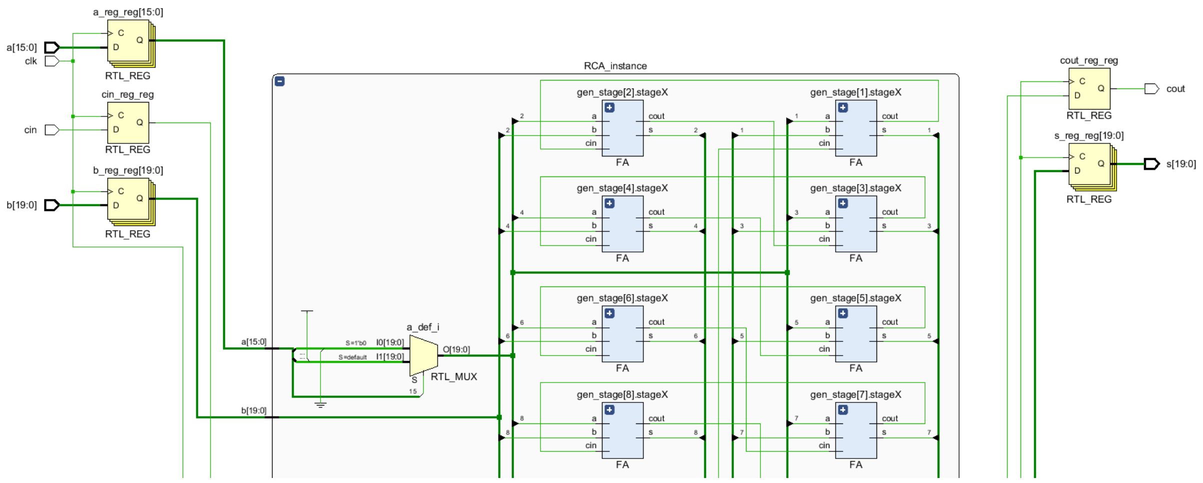
Vivado Implementation 27. DADDA multiplier: detail of the RTL schematic



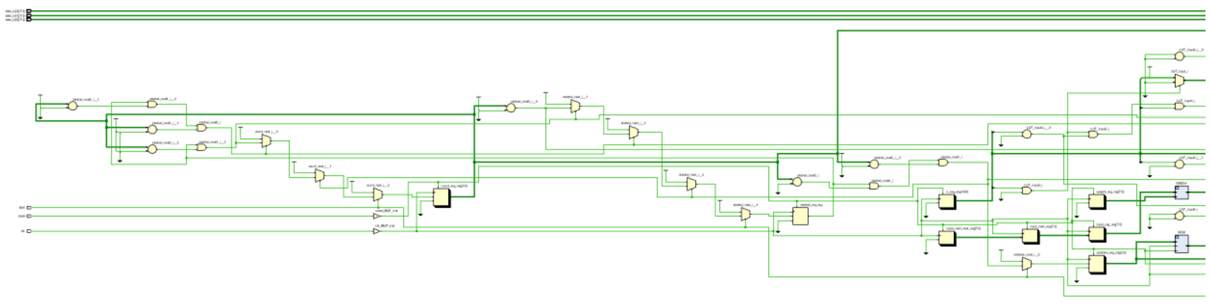
Vivado Implementation 28. Array multiplier: detail of the RTL schematic



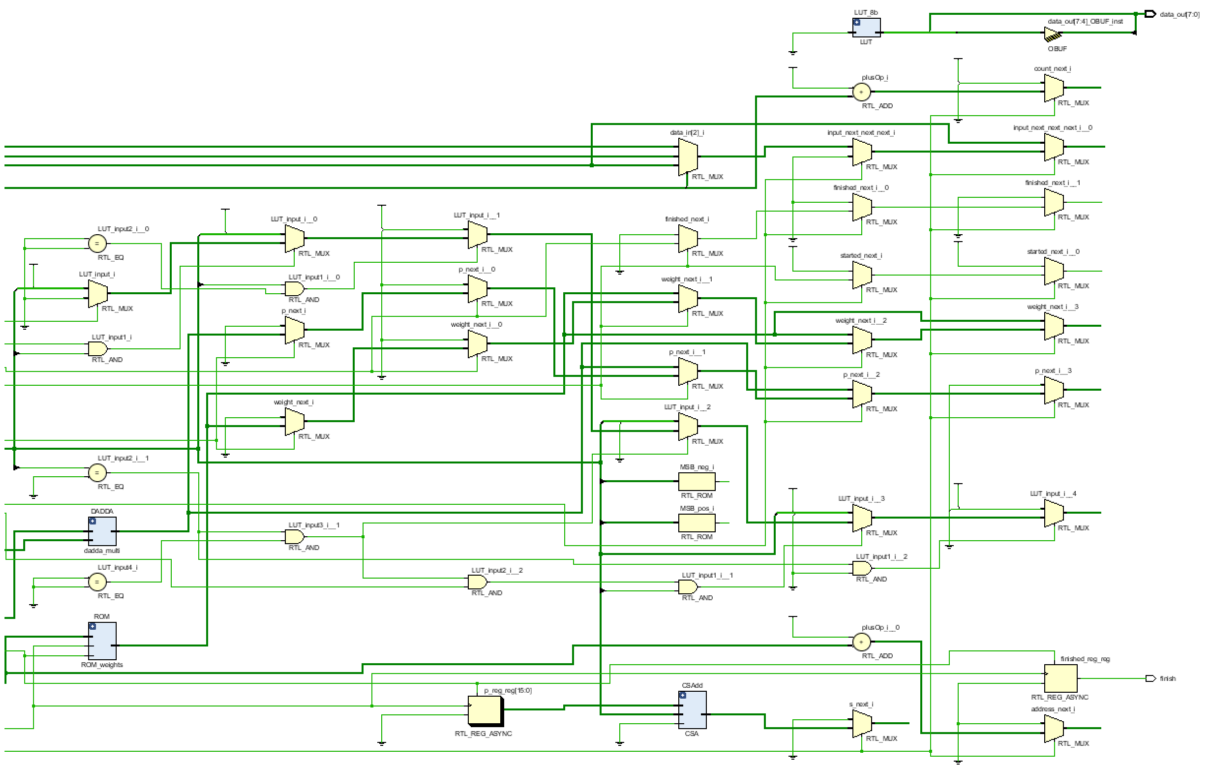
Vivado Implementation 29. Carry Select Adder: detail of the RTL schematic



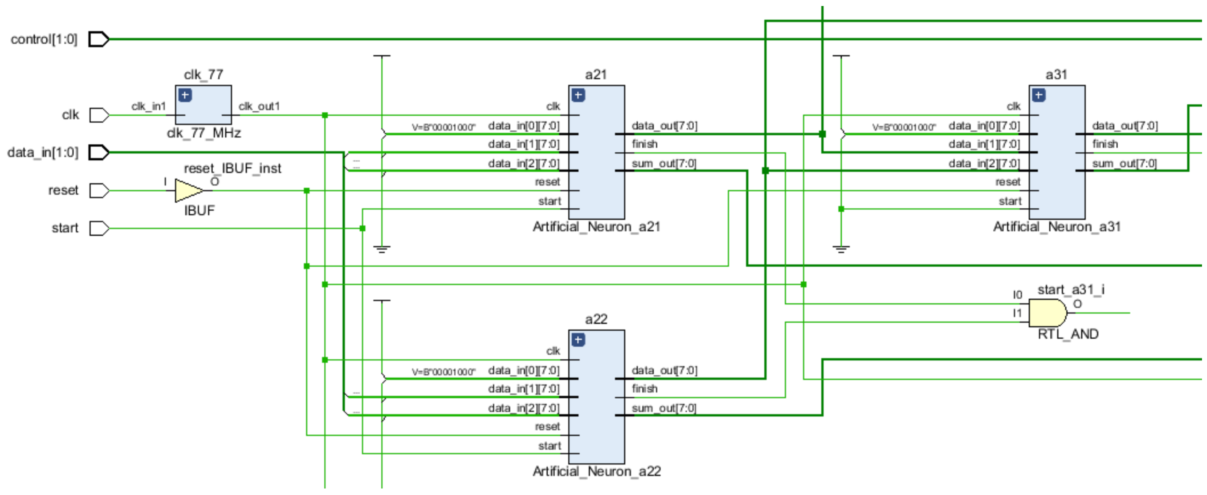
Vivado Implementation 30. Ripple Carry Adder: detail of the RTL schematic



Vivado Implementation 31. Artificial Neuron: detail of the RTL schematic of the system controller



Vivado Implementation 32. Artificial Neuron: detail of the RTL schematic of the truncating and rounding stage



Vivado Implementation 33. Artificial Neural Network: detail of the RTL schematic of the connections between Artificial Neurons

ANNEX E: DETAIL OF ASMD DIAGRAM OF SYSTEM CONTROLLER

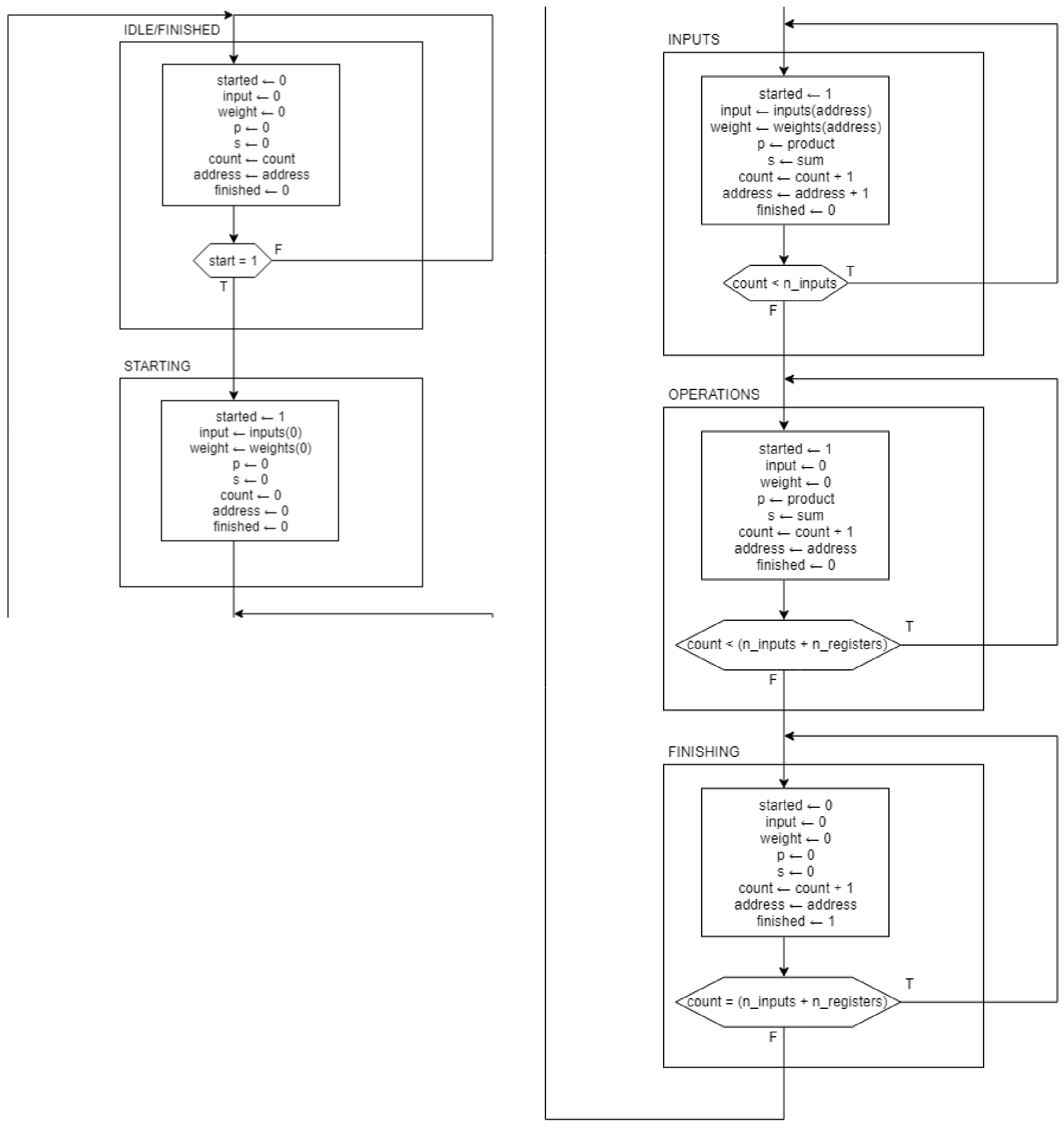


Figure 31. Detail of ASMD diagram