

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique



Electronics Department

Laboratoire des Dispositifs de Communication et de Conversion Photovoltaïque

Graduation dissertation for the diploma of

Electronics Engineer

Theme

**Implementation of Artificial Neural on an FPGA board
Application on Induction Motor speed control**

Presented by

- **SAADI Khalid**
- **OUADRIA Anes Abderrahim**

Publicly presented on June 19th, 2017

Jury members:

SADOUN Rabah	MCA	ENP	President
GUELLAL Ammar	PHD	ENP	Mentor
LARBES Cherif	Professor	ENP	Mentor
HADDADI Mourad	Professor	ENP	Examiner

ENP 2017

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique



Electronics Department

Laboratoire des Dispositifs de Communication et de Conversion Photovoltaïque

Graduation dissertation for the diploma of

Electronics Engineer

Theme

**Implementation of Artificial Neural on an FPGA board
Application on Induction Motor speed control**

Presented by

- **SAADI Khalid**
- **OUADRIA Anes Abderrahim**

Publicly presented on June 19th, 2017

Jury members:

SADOUN Rabah	MCA	ENP	President
GUELLAL Ammar	PHD	ENP	Mentor
LARBES Cherif	Professor	ENP	Mentor
HADDADI Mourad	Professor	ENP	Examiner

ENP 2017

ACKNOWLEDGEMENTS

We would like to thank our mentor, Amar Guellal for guiding and supporting us over the graduate period. He has set an example of excellence as a researcher, advisor and instructor.

We would like also to thank our professor Cherif LARRES for all of his guidance through this process; his discussion, ideas, and feedback have been absolutely invaluable.

We would like to thank our fellow graduate students who contributed to this research. I am very grateful to all of you.

Finally, we would especially like to thank our amazing families for the love, support, and constant encouragement we have got over the years. We undoubtedly could not have done this without them.

ملخص: في العديد من التطبيقات الهندسية، هناك حاجة ماسة إلى أنظمة موازية التوزيع. الشبكات العصبية الاصطناعية تمثل أنظمة موزعة للغاية، و هي مستوحاة مباشرة من الدماغ البشري. و بالتالي فهي مناسبة لهذه التطبيقات. غير أن تنفيذ هذه الشبكات أظهر تحديا كبيرا. وهو يتطلب التوازي والمرونة. من بين جميع أنواع الدارات، FPGA أثبتت أنها الأكثر ملائمة لذلك. ولكن تصميم بنية شبكات الأجهزة العصبية الاصطناعية ليتم تنفيذها على FPGA لديها بعض المشاكل التي يتعين التعامل معها، مثل كيفية تحقيق التوازن بين الدقة الرقمية، التوازي و نقص الموارد. من خلال العمل المنجز في إطار هذه الأطروحة اقترحنا تطبيق C++ يولد الوصف الأمثل لشبكات الأجهزة العصبية الاصطناعية. بدأنا أولا بعرض عموميات على الشبكات العصبية الاصطناعية، و ذلك في الفصل الأول. و خصصنا الفصل الثاني لتقديم لمحة عامة عن بنية FPGA. في الفصل الثالث عرضنا بإيجاز المسائل الحسابية في تصميم الشبكات العصبية الاصطناعية. بعد ذلك، عرضنا كل من البنية الخاصة بعصبون واحد و البنية الخاصة بالشبكة العصبية الاصطناعية كاملة. انتهينا من هذا الفصل من خلال عرض طريقة برمجة تطبيقنا ومبدأ عمله. تمت المحاكاة بواسطة ModelSim حيث أجريت وأظهرت حسن سير عمل الشبكات العصبية الناتجة عن التطبيق. في الفصل الأخير استخدمنا تطبيقنا لتوليد ستة شبكات عصبية اصطناعية، تم استخدامها في تقنية جديدة خاصة بالتحكم في سرعة المحرك اللامتزامن. هذه التقنية تسمى ANN SHE. أجريت المحاكاة وكانت النتائج جيدة وكما هو متوقع.

كلمات مفاتيح : الشبكات العصبية الاصطناعية, FPGA, تطبيق C++, المحرك اللامتزامن.

Résumé : Dans de nombreuses applications d'ingénierie, les systèmes parallèles qui satisfont la contrainte de temps-réel sont fortement nécessaires. Les réseaux de neurones artificiels représentent des systèmes à distribution parallèles, ils ont été directement inspirés du cerveau humain. Ils sont donc utilisés dans de nombreuses applications de ce type. Cependant, l'implémentation de ces réseaux s'est avérée très difficile. Cela nécessite un parallélisme et une flexibilité. Parmi tous les types de circuits, les circuits FPGA se sont avérés les plus pratiques pour cela. Mais la conception de l'architecture RNA à implémenter sur les FPGA a quelques défis, comme la façon d'équilibrer entre la précision numérique (requis pour la précision), le parallélisme et les limites des ressources. Dans ce travail, nous avons proposé une application basée sur C++ qui génère des descriptions de RNA optimisées. On a commencé par introduire des généralités sur les RNA au chapitre un. On a consacré le deuxième chapitre à présenter un aperçu de l'architecture de l'FPGA. Dans le chapitre trois, nous avons brièvement présenté les problèmes d'arithmétique dans les implémentations de RNA. Après cela, on a présenté les architectures d'un seul neurone et celle du RNA entier. On a terminé ce chapitre en représentant la façon dont nous avons codé notre application et le principe de son fonctionnement. Des simulations en ModelSim ont été effectuées et elles ont montré le bon fonctionnement des RNA générés par l'application. Dans le dernier chapitre, on a utilisé notre application pour générer 6 RNAs qui ont été utilisées dans une nouvelle technique de contrôle de moteur asynchrone appelée ANN SHE. Des simulations ont également été effectuées et les résultats étaient bons et comme prévu.

Mot clés : Réseaux de neurones artificiels (RNA), FPGA, application C++, moteur asynchrone.

Abstract: In many engineering applications, parallel distributed systems that satisfy the real-time constraint are strongly needed. Artificial neural networks (ANNs) represent highly parallel distributed systems that were directly inspired from the human brain. Thus they are appropriate for such applications. However the implementation of these networks proved to be quite challenging. It requires parallelism and flexibility. Among all types of circuits, FPGAs have proved to be the most convenient for that. But designing ANNs architectures to be implemented on FPGAs have some issues to be dealt with, like how to balance between numeric precision (required for accuracy), parallelism, and resources limitations. In this work we have proposed a C++ based application that generates optimized ANNs descriptions. We started first by introducing generalities on ANNs in chapter one. We devoted the second chapter to present an overview on the FPGA's architecture. In chapter three we briefly presented the arithmetic issues in ANNs implementations. After that, we presented both architectures of a single neuron, and the whole ANN. we ended this chapter by presenting the way we coded our application and the principle of its functioning. ModelSim simulations were performed and they showed the good functioning of neural networks generated by the application. In the last chapter we have used our application to generate 6 ANNs that were used in a new induction motor control technique called ANN SHE. Simulations were performed as well and results were good and as expected.

Key words: Artificial Neural Network (ANN), FPGA, C++ based application, induction motor.

CONTENTS

Acknowledgements	
Contents	
Table List	
Figure List	
Introduction	9
CHAPTER 1. Artificial neural networks (ANNs)	12
1.1 Introduction	12
1.2 Historical Perspective on Neural Nets	12
1.3 Biological inspiration	14
1.3.1 Structure	14
1.3.2 Functioning of a neuron	15
1.4 Artificial neural networks	17
1.4.1 Mathematical model of artificial neuron	18
1.4.2 Architectures of neural networks	21
1.4.3 Training Neural Networks	22
1.5 The properties of neural networks	25
1.6 Areas of application of neural networks	26
1.7 Conclusion	26
CHAPTER 2. FPGA Architecture	28
2.1 Introduction	28
2.2 The classification of digital circuits	28
2.2.1 Circuits with programmable architecture	29
2.3 The FPGA circuit	30
2.3.1 History	30
2.3.2 Application	30
2.3.3 FPGA Architecture	31
2.4 The circuit board Nexys 2	35
2.4.1 Spartan 3E architecture	36
2.5 VHDL	37
2.5.1 Brief on VHDL	37

2.5.2	Utility of VHDL	37
2.5.3	VHDL structure	38
2.5.4	The modes used in VHDL	39
2.6	Conclusion	39
CHAPTER 3.FPGA implementation of neural networks		41
3.1	Introduction	41
3.2	Arithmetic in ANN digital implementation	41
3.3	The Activation Function implementation	44
3.3.1	Implementation of Tangent Sigmoid and Log sigmoid	45
3.4	The architecture overview of the ANN generated by the C ⁺⁺ application	53
3.4.1	Concept of layer multiplexing	54
3.4.2	Single neuron architecture	54
3.4.3	The Global ANN architecture	56
3.5	C++ application	57
3.6	Conclusion	60
CHAPTER 4.ANN SHE PWM technique		62
4.1	Introduction	62
4.2	SHE PWM	62
4.2.1	Introduction	62
4.2.2	Principle of operation	63
4.2.3	The switching angles	64
4.3	Implementation of ANN SHE PWM	65
4.3.1	Architecture of the ANN SHE	66
4.3.2	Implementation of ANN SHE	70
4.4	Simulation and Results	75
4.5	Conclusion	76
General Conclusion		77
Bibliography		79

TABLE LIST

Table 1-1 : Analogy between real and artificial neuron

Table 3-1: LUT for the hyperbolic tangent activation Function

Table 3-2 : LUT for $f(x) = \text{sig2}(x) - 0.5$

Table 4-1: ANN SHE characteristic

Table 4-2 : An example of switching angles ($m=7$) generated by a Matlab program

Table 4-3 : weights and biases calculated in the training phase

Table 4-4 : The exact values for switching angles for $m=0.595$ and $m=0.615$

Table 4-5 : A comparison between the exact and ANN SHE switching angles

Table 4-6: The intervals for the variation of i_m

FIGURE LIST

- Figure 1-1 A Neuron Cell Anatomy
Figure 1-2: The Synapse
Figure 1-3: Action potential in a Neuron
Figure 1-4 : Non linear model of a neuron
Figure 1-5: Types of Activation Functions
Figure 1-6: Feed-Forward neural networks
Figure 1-7: Recurrent Network
Figure 1-8: Types of training
Figure 1-9 : Supervised learning scheme
Figure 1-10: Unsupervised learning scheme
Figure 1-11: Reinforcement learning scheme
- Figure 2-1: Classification of digital circuits
Figure 2-2: Overview of FPGA architecture
Figure 2-3 : Basic Logic Element (BLE)
Figure 2-4 : A configurable Logic Block (CLB) having four BLEs
Figure 2-5 : Static Memory cell
Figure 2-6 : Nexys 2 board FPGA
Figure 2-7 : Spartan E3 family architecture
Figure 2-8 : Basic structure of VHDL description
- Figure 3-1: Sum of products 1
Figure 3-2: Sum of Products 2
Figure 3-3: Serial sum of products
Figure 3-4: Hyperbolic Tangent Sigmoid Activation Function
Figure 3-5: Saturation region of tangent hyperbolic
Figure 3-6 : Hyperbolic Tangent LUT block diagram
Figure 3-7: Range decoder simulation
Figure 3-8: Hyperbolic Tangent simulation
Figure 3-9: Sigmoid activation function
Figure 3-10 Log sigmoid2 simulation
Figure 3-11: Sigmoid2 Implementation diagram
Figure 3-12 : Layer multiplexed ANN
Figure 3-13: A Single neuron block diagram
Figure 3-14 : ANN Architecture
Figure 3-15: Flow state of 2 state machines
Figure 3-16 : ANN 1_1_15
Figure 3-17 : ANN 1_1_5

Figure 4-1 : The Inverters output normalized Voltage

Figure 4-2 : ANN SHE Architecture

Figure 4-3: A ModelSim simulation to generate the angles ($i_m = 0.595$ & 0.615)

Figure 4-4 : Interval selector structure

Figure 4-5 : PWM generator algorithm

Figure 4-6: The three-phase PWM signals ($i_m = 40\%$)

Figure 4-7: The three-phase PWM signals ($i_m = 59.5\%$)

Figure 4-8 : Frequency spectrum of PWM signal for $i_m = 40\%$

INTRODUCTION

The man in his attempts to understand how the human brain works, developed what is so called artificial neural networks (ANNs). These artificial networks consist of many processing units connected together in a parallel distribution, to form a network that can behave like the brain in doing a particular task. But the brain is still a very complex system that scientists have not fully understand yet, even with today's highly developed technologies. Thus modeling it stays a farfetched dream to reach for now. Instead researchers tried to model its most important elementary unit called the neuron. Many works have been done this way, and the results were impressive; the actual models of ANNs made them capable of learning from experience, and have flexibility that allows them to adapt their structure for a particular application. These features are what made them very successful. Now neural networks are being used everywhere, in patter recognition, voice recognition, data mining, machine learning, intelligence control ... etc.

However the implementation of these neural nets proved to be quite challenging from many sides, and it still be an open research field. An implementation by software provides an excellent architecture flexibility and testability, however running a parallel structure like ANNs in sequential general purpose hardware does not really fit for real time applications. Parallel reconfigurable circuits have appeared, the most successful ones are ASICs (Application-specific integrated circuit) and FPGAs (Field Programmable Gate Arrays), and these two specifically were very promising for the ANNs implementations.

The FPGAs were quickly optimized; they are now very quick circuits that have bigger integration density and better technical support. This had a direct impact in making them the most suited for neural networks, since they can be used for reconfigurable computing and offer software design flexibility with performance speeds closer to ASICs.

Even though, using FPGAs as technology for ANNs implementation was the best solution, there still many challenges that will face a designer of ANNs when using an FPGA, like the necessity to balance between area (FPGA resources) and numeric precision needed for accuracy and speed of convergence.

To program actual FPGAs, one could use Xilinx ISE (an Integrated Synthesis Environment); it offers the possibility to code in high level languages like the VHDL (VHSIC Hardware Description Language).

In our work we have proposed a C++ based application whose function is to generate readymade feed forward ANNs descriptions, in form of VHDL files. Naturally these generated ANNs descriptions are specific, and chosen by the user, who should specify the

anatomy of his wanted ANN (number of inputs, outputs , layers , neurons per layer) as well as its weights and biases (parameters generated by the off line training).

In chapter one, we have introduced generalities on ANNs, like their types, training, and their mathematical model, that is used later in their implementation. The second chapter presents the FPGA circuit architecture and its features that made it the best choice for ANNs hardware implementation. The third chapter starts by introducing the implementations issues. Then we presented the optimized methods used in implementing the different blocks of the artificial neuron, and the global ANN. After, we explained briefly the way we coded the C++ application and how it works in general. To test it we presented some simulations. Then in chapter four we have used 6 ANNs generated by our application, in a motor control method called ANN SHE PWM. Finally, we concluded by a general conclusion.

CHAPTER 1

CHAPTER 1. ARTIFICIAL NEURAL NETWORKS (ANNS)

1.1 Introduction

Today's conventional digital computers are getting extremely fast; they can perform a lot of instructions and highly complex operations in just few clock cycles, it is way quicker than the human in this. However faster is not enough in solving problems, there are many tasks in which the computer loses against the human brain, the latter is a highly *complex, non linear, and parallel computer* (information processing-system). It works in a totally different way, it has the capability to adapt its structural constituents called neurons, so as to perform certain computations (e.g., patten recognition, perception, and motor control) many times faster than the fastest digital computer in existence today. For instance, given two pictures, a preschool child can easily tell the difference between a cat and a dog. Yet, this same simple task is extremely difficult for today's computers.

The artificial neural network is a machine designed to model the way our brain performs a particular task in solving a given problem, it can be defined as following:

A neural network is a massively parallel distributed processor made up of simple processing units whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the inter-unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns.[1]

This chapter begins with a small historical overview on neural networks, and their development through years, then comes a brief part in which we exposed the biological neuron's anatomy, the origin of the actual model of artificial neurons. After that the mathematical model of a single neuron is presented as well as some topologies of ANNs. To arrive to the most important part, that is the learning characteristic of ANNs, and its different methods. Finally, we presented a non exhaustive list of actual areas of applications for neural networks.

1.2 Historical Perspective on Neural Nets

Neural networks have been in use with computers since 1949 when D.Hebb, an American physiologist, published his book entitled "the organization of behavior", in which he exposed some of his ideas on learning for the very first time , the Hebb rule that he proposed was one

of the learning rules on which rests most of today's connectionist algorithms[2]. Through the years, many neural network architectures have been presented.

In 1957, F. Rosenblatt developed the model of the perceptron, one of the earliest neural networks, which was an attempt to understand human memory, learning and cognitive processes. In 1960, Rosenblatt demonstrated the Mark I perceptron. The Mark I was the first machine that could "learn" to identify optical patterns. Based on the Hebb rule, the perceptron was then considered to be the first machine that could "learn" from experience. Unfortunately it was unable to learn to recognize inputs that were not "linearly separable" [3]. This would prove to be a huge obstacle that would take some time to overcome.

A new neural model was developed by B. Widrow and T. Hoff in 1960, called the Adaline network (Adaptative Linear Element). In its structure it resembles to the perceptron, but the learning rule was different. They proposed the minimization of the quadratic output errors as a learning algorithm. The Adaline network is considered as the basic model of multilayer networks [2].

In 1969, the theoretical limitations of the perceptron were demonstrated by M. Minsky and S. Papert. These limitations concerned the impossibility of dealing with nonlinear problems using this model. The impact of their results has frustrated most researchers in this field, especially compute scientists. This stagnation lasted almost 20 years. During this period, researchers and investors turned to the approach of artificial intelligence, which seemed to be more promising.[1]

This discipline was brought to life again, in 1982 thanks to J. J. Hopfield, an eminent physicist, who was able to detect the similarity between networks proposed by McCulloch and Pitts, with an elementary system with magnetic moment or spin, and then he studied the storage and restoration of information "associative memories". This led to one of Hopfield's major contributions when he had the idea of using an energy function to maintain stability of neural networks, with such a function, states tend to change to a local minimum. This work interested physicists because of the isomorphism of the Hopfield model with the Ising model, also called spin glasses. It is important to note that this model did not remove the limits of the perceptron and its variants. In spite of this, the perceptron and the reasons for its failure proved to be quickly forgotten [1],[2].

Then, Boltzmann's machine was proposed in 1983 by Hinton and his team, it was the firstly model that removed the perceptron limitations satisfactorily. This model used what is so called hidden cells whose role is to compute intermediate variables to perform non linear separable functions. Unfortunately, the convergence of the algorithm was extremely long, it had a defect due to its probabilistic nature, and it was corrected by the gradient backpropagation algorithm proposed in 1986 by three researchers Rumelhart, Hinton and William.

Finally, in 1989 Moody and Darken exploited some results of the multi-variable interpolation to propose what is known as the radial basis function network[2].

Recently, the new discoveries in neurobiology and the explosive interest of parallel processing, in addition to the development of semiconductor technology, have given great impetus to the field of neural networks.

1.3 Biological inspiration

1.3.1 Structure

To create a machine capable of “human-like thought”, researchers have used the best available model available “the human brain”. However, this one is far too complex to be modeled. Rather, they studied the individual cells that make it up. At the most basic level the brain is composed of neuron cells. They are the basic building blocks of the human brain; there are about 100 milliards of them in it. Artificial neural networks are an attempt to simulate these cells’ behavior.

A stereotypical neuron cell is show in Figure 1-1. It consists of:

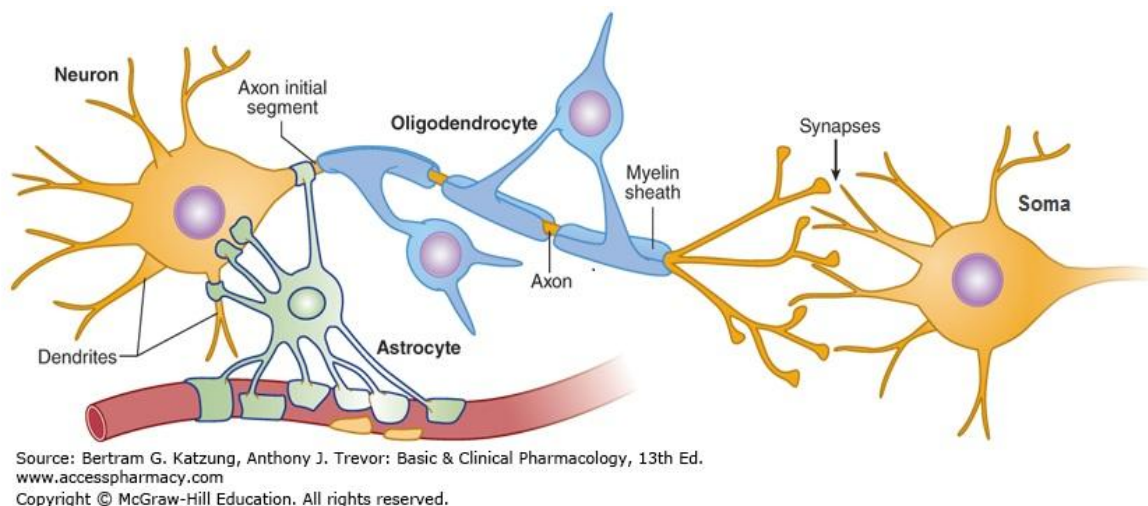


Figure 1-1 A Neuron Cell Anatomy [4]

Cell body or “*soma*” contains the usual sub-cellular components to be found in most cells throughout the body (nucleus, mitochondria, Golgi body, etc.) but these are not shown in the diagram. Instead this diagram was made to focus on what differentiates neurons from other cells allowing the neuron to function as a signal processing device. This ability stems largely from the properties of the neuron’s surface covering or membrane, which supports a wide variety of electrochemical processes. Morphologically the main difference lies in the set of fibers that emanate from the cell body. One of these fibers is called the axon.

The axon is responsible for transmitting signals to other neurons and may therefore be considered the neuron output. All other fibers are called dendrites.

The dendrites carry signals from other neurons to the cell body, thereby acting as neural inputs. Each neuron has only one axon but can have many dendrites. The latter often appear to

have a highly branched structure and so we talk of dendritic arbors. The axon may, however, branch into a set of collaterals allowing contact to be made with many other neurons. With respect to a particular neuron, other neurons that supply input are said to be afferent, while the given neuron's axonal output, regarded as a projection to other cells, is referred to as an efferent. Afferent axons are said to innervate a particular neuron and make contact with dendrites at the junctions called *synapses* see Figure 1-2

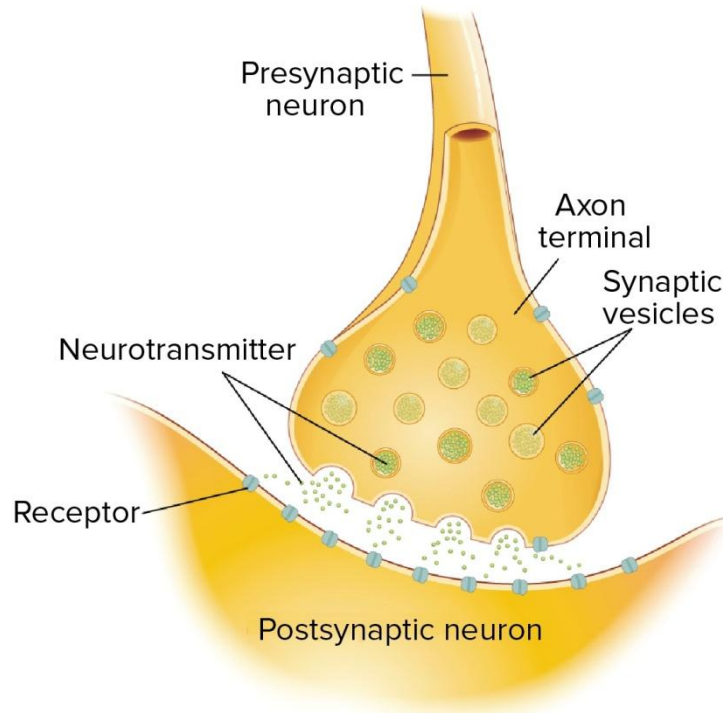


Figure 1-2: The Synapse

Here, the extremity of the axon, or axon terminal, comes into close proximity with a small part of the dendritic surface—the postsynaptic membrane. There is a gap, the synaptic cleft, between the presynaptic axon terminal membrane and its postsynaptic counterpart, which is of the order of 20 nanometers (2×10^{-8} m) wide. Only a few synapses are shown in Figure 1-1, but in reality they are located over all dendrites and also, possibly, the cell body.

Finally the two other cells “Astrocyte” and “Oligodendrocytes” are the Glial cells (Figure 1-1). Their main role is to assure protection for the neuron cells.

1.3.2 Functioning of a neuron

At the simplest level, neurons produce pulses, called “Action Potentials,” and they do this when stimulated by other neurons (or, if they are sensory neurons, by outside influences, which they pick up through their modified dendrites).

When a neuron is at rest, before it becomes stimulated, it is said to be polarized. This means that, although the neuron is not receiving any electrical signal from other neurons, it is charged up and ready to produce a pulse. This is due to the fact that its membrane at equilibrium, works to maintain an electrical imbalance of negatively and positively charged ions, this causes a potential difference across the membrane with the inside polarized by approximately 70mV, with respect to the outside.

Each neuron has associated with it a level of stimulus, above which a nerve pulse or action potential will be generated. Only when it receives enough stimulation, from one or more sources, will it initiate a pulse. The mechanism by which the pulses travel and the neuron maintains its general electrical activity is rather complex, it was first worked out by Hodgkin & Huxley (1952) [5], and it works through an exchange of ions in the fluid that surrounds the cell, rather than by the flow of electrons as anyone would get in a wire. This means that signals travel very slowly - at a couple of hundred meters per second. The pulse, which the neuron generates and travels down the axon, is shown in Figure 1-3

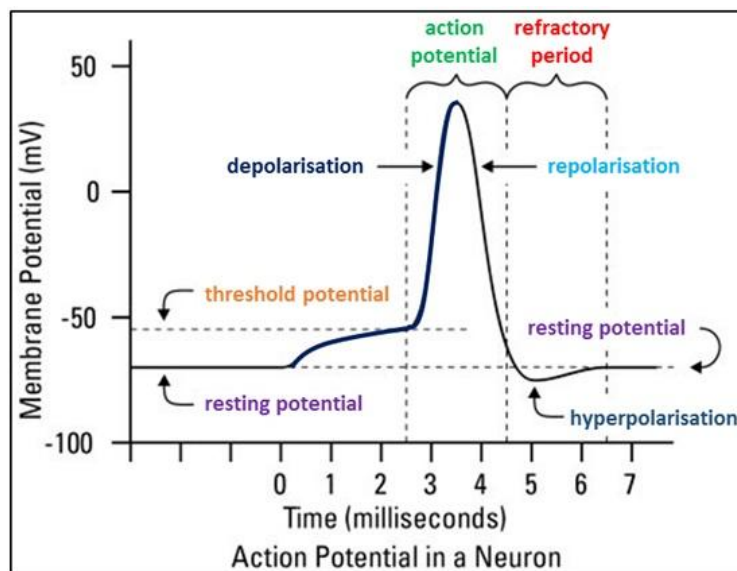


Figure 1-3: Action potential in a Neuron

Because these pulses are only a couple of milliseconds wide, they often appear as spikes if viewed on an oscilloscope screen. So, if one neuron is receiving lots of stimulation from another (receiving lots of pulses through its dendrites) then it will itself produce a strong output - that is more pulses per second.

Once a signal or an action potential reaches the axon terminal, these contain a chemical substance “nanotransmitters” held within a large number of small vesicles (literally “little spheres”) (Figure 1-2). On receipt of an action potential the vesicles migrate to the presynaptic membrane and release their nanotransmitters across the synaptic cleft, the transmitter then binds with receptor sites at the postsynaptic membrane. This initiates an electrochemical process that changes the polarization state of the membrane local to the

synapse. This postsynaptic potential (PSP) can serve either to depolarize the membrane from its negative resting state towards 0 volts, or to hyperpolarize the membrane to an even greater negative potential. The PSP spreads out from the synapse, travels along its associated dendrite towards the cell body and eventually reaches the axon hillock—the initial segment of the axon (Figure 1-1) where it joins the soma. Concurrent with this are thousands of other synaptic events distributed over the neuron. These result in a plethora of PSPs, which are continually arriving at the axon hillock where they are summed together to produce a resultant membrane potential. The integrated PSP at the axon hillock will affect its membrane potential and, if this exceeds a certain *threshold* (typically about -55mV) (Figure 1-3), an action potential is generated, which then propagates down the axon, along any collaterals, eventually reaching axon terminals resulting in a shower of synaptic events at neighbouring neurons “downstream” of our original cell.

In 1949 Donald Hebb postulated one way for the network to learn. If a synapse is used more, it gets strengthened – releases more Neurotransmitter. This causes that particular path through the network to get stronger, while others, not used, get weaker. One might say that each connection has a *weight* associated with it – larger weights produce more stimulation and smaller weights produce less. These were the first steps to understanding the learning mechanism of the network [6].

To summarize:

- Signals are transmitted between neurons by action potentials, which have a stereotypical profile and display an “*all or nothing*” character; there is no such thing as half an action potential.
- When an action potential impinges on a neuronal input (synapse) the effect is a PSP, which is variable or graded and depends on the physicochemical properties of the synapse.
- The PSPs may be excitatory or inhibitory.
- A synapse can be strengthened when used more, when not used, it gets weaker. It *adapts*, the connection is then said to be *weighted*
- The PSPs are summed together at the axon hillock with the result expressed as its membrane potential.
- If this potential exceeds a *threshold* an action potential is initiated that proceeds along the axon.

1.4 Artificial neural networks

An Artificial neural network is an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the inter unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns [5]. They possess several fundamental characteristics:

- They are composed of two or more layers. Typically, these include an input layer, whose processing units encode the initial representation of the situation, one or more hidden layers,

The units combine the information from the input units, and an output layer, Whose units produce the system's response to the situation.

- Simple artificial neurons are connected to other neurons in different layers (and sometimes within the same layer). The weight of connections changes when the system acquires more experience (training), these weights are crucial for determining the treatment performed.
- As in the brain, a given processing unit activates when the stimulus level received from all other units to which it is connected exceeds a certain threshold. The level of stimulus received from each unit is determined, on the one hand, by the degree of activation of that unit and, on the other hand, by the weight of the connection between the sending and the receiving unit.
- The activity of most processing units occurs in parallel (simultaneously).
- Knowledge is represented by the weight of connections within all units of the system.
- Learning occurs when the system that receives inputs, elaborates a response, observes the difference between the response provided and the correct response and adjusts the weight of the connections between the processing units to produce a better response. Adjustments include strengthening some connections and weakening others.
- The generalization of knowledge of the system is based on the similarity of new situations to those already encountered by the system.

1.4.1 Mathematical model of artificial neuron

A neuron is an information-processing unit that is fundamental to the operation of a neural network. The Table 1-1 resumes the analogy between a real and an artificial neuron. The diagram of Figure 1-4 shows the model of a neuron, which forms the basis for designing an artificial neural network. Three basic elements of the neuronal modal can be identified:

- **A set of synapses** or connecting links, each of which is characterized by weight or strength of its own. Specifically a signal X_k at the input of synapse k connected to neuron j is multiplied by the synaptic weight W_{jk} . The first subscript refers to the neuron in question and the second subscript refers to the input end synapse to which the weight refers. Unlike real neurons, the synaptic weights can have negative values.
- **An adder** for summing the input signals, weighted by the respective synapses of the neuron; till here the operations described constitute a linear combiner.
- **An activation function** for limiting the amplitude of the output of a neuron. The activation function is also referred to as a squashing function in that it squashes (limits) the permissible amplitude range of the output signal to some finite value. Typically, the normalized amplitude range of the output of a neuron is written as the closed interval $[0; 1]$ or alternatively $[-1; 1]$.

The model also in the Figure 1-4 includes an externally applied bias, denoted θ_j It has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively.

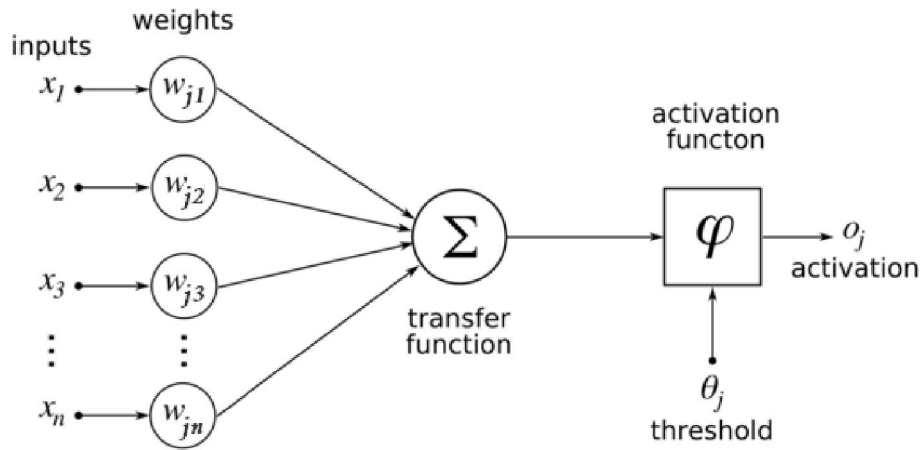


Figure 1-4 : Non linear model of a neuron

In mathematical terms, we may describe a neuron j by writing the following pair of equations:

$$S_j = \sum_{k=1}^{k=n} W_{jk} \times X_k \tag{1-1}$$

$$o_j = \varphi(S_j + \theta_j) \tag{1-2}$$

where X_1, X_2, \dots, X_n are the inputs, and φ is the activation function .

Table 1-1 : Analogy between real and artificial neuron

Real neuron	Artificial neuron
Cell body (Soma)	Activation Functions
Axons	Output signals
Synapses	Synaptic weights
Dendrites	Input signals

1.4.1.1 Types of Activation Functions:

Here we identified three basic Activation functions:

• **Threshold Function:** also is referred to as a Heaviside function, it is described the following (Figure 1-5):

$$\varphi(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{if } s < 0 \end{cases} \tag{1-3}$$

This model of neurons based on this Activation Function is referred to in the literature as the McCulloch-Pitts model, in recognition of the pioneering work done by McCulloch and Pitts (1943).

- **Piecewise-Linear Function:** For the one described in Figure 1-5 we have

$$\varphi(s) = \begin{cases} 1, & s \geq \frac{1}{2} \\ s + \frac{1}{2}, & \frac{1}{2} > s > -\frac{1}{2} \\ 0, & s \leq -\frac{1}{2} \end{cases} \quad (1-4)$$

where the amplification factor inside the linear region of operation is assumed to be unity. This form of an activation function may be viewed as an approximation to a non-linear amplifier.

This function can have two special forms:

- A linear combiner, if the linear region of operation is maintained without running into saturation (Figure 1-5).
 - A threshold function, if the amplification factor of the linear region is made infinitely large.
- **Sigmoid Function:** It is by far the most common form of activation function used in the construction of artificial neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the logistic function (Figure 1-5), defined by

$$\varphi(s) = \frac{1}{1 + \exp(-as)} \quad (1-5)$$

where a is the slope parameter of the sigmoid function. By varying the parameter a , sigmoid functions of different slopes can be obtained. In fact, if the slope parameter approaches infinity, the sigmoid function becomes simply a threshold function. In contrast with the threshold function the sigmoid function assumes a continuous range of values from 0 to 1 and is differentiable (Differentiability is an important feature of neural network theory).

The activation functions defined in equations (1-3), (1-4), and (1-5) range from 0 to 1. Other activation functions are antisymmetric, and range from -1 to 1 (see Figure 1-5)

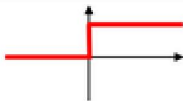
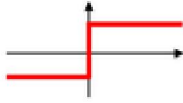
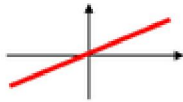
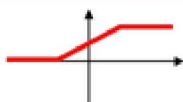


Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

Figure 1-5: Types of Activation Functions

Another important model of artificial neurons is the stochastic neuron, described as follows:

$$\phi(s) = \begin{cases} 1 & \text{with probability } P(s) \\ 0 & \text{with probability } 1 - P(s) \end{cases}$$

Where the probability is chosen to be:

$$P(s) = \frac{1}{1 + \exp\left(-\frac{s}{T}\right)} \tag{1-6}$$

This model has the same activation function of the McCulloch-Pitts model with a probabilistic interpretation. That is to say that the neuron is permitted to stay in only one of two states 0 or 1. The decision for the neuron to fire (i.e. to change the state from 0 to 1) is probabilistic.

1.4.2 Architectures of neural networks

From an architectural view, neural networks can be sorted into two big categories:

- **Feed-forward** networks, where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple layers of units, but no feedback connections are present, that is, connections extending from outputs of units to inputs of units in the same layer or previous layers. In this category, we can distinguish single-layer networks

(e.g. perceptron) and multilayer networks with an input layer, an output layer and one or more hidden layers (Figure 1-6).

- **Recurrent networks** that do contain feedback connections. In this category, we can distinguish competitive networks, the Kohonen network, the Hopfield networks (Figure 1-7) and the ART models "Theory of Artificial Resonance".

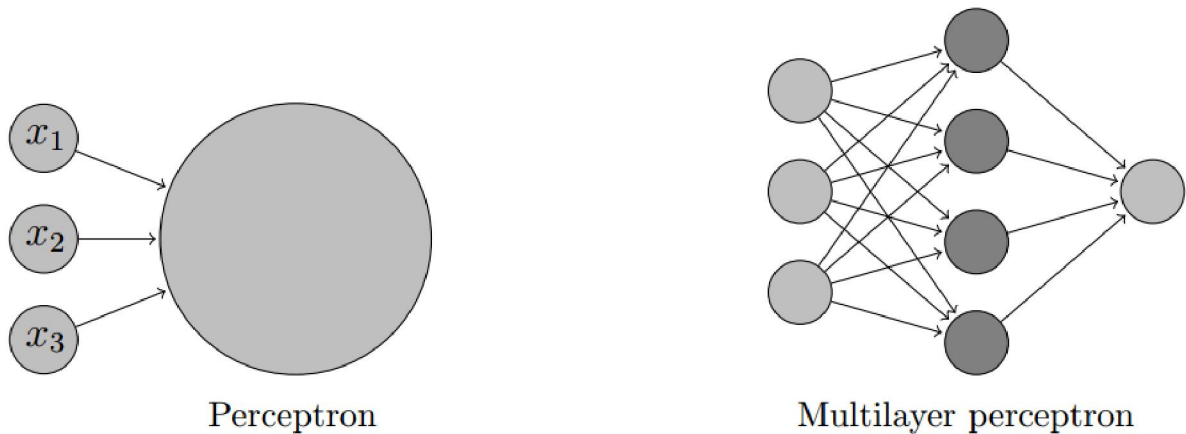


Figure 1-6: Feed-Forward neural networks [7]

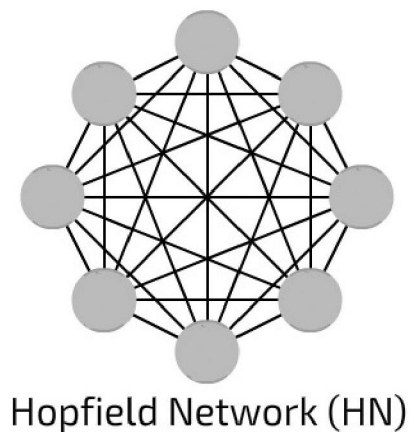


Figure 1-7: Recurrent Network

1.4.3 Training Neural Networks [3], [8]

In a neural network, individual neurons are interconnected through their synapses. These connections allow the neurons to signal each other as information is processed. Not all connections are equal. Each connection is assigned a connection weight. If a weight is zero then there is not a connection. These weights are what determine the output of the neural network; therefore, it can be said that these weights form the memory of the neural network. Thus training the networks means to configure it (its weights) such that the application of a set of inputs produces the desired set of outputs.

In general training algorithms begin by assigning random values to the weights. Then, the validity of the neural network is examined. Next, the weights are adjusted based on how well the neural network performed and the validity of the results. This process is repeated until the validation error is within an acceptable limit. There are many ways of training. One way is to set the weights explicitly, using a priori knowledge. Another way is to ‘train’ the neural network by feeding it teaching patterns and letting it change its weights according to some learning rule.

1.4.3.1 Types of trainings:

There are mainly two categories of training (see Figure 1-8):

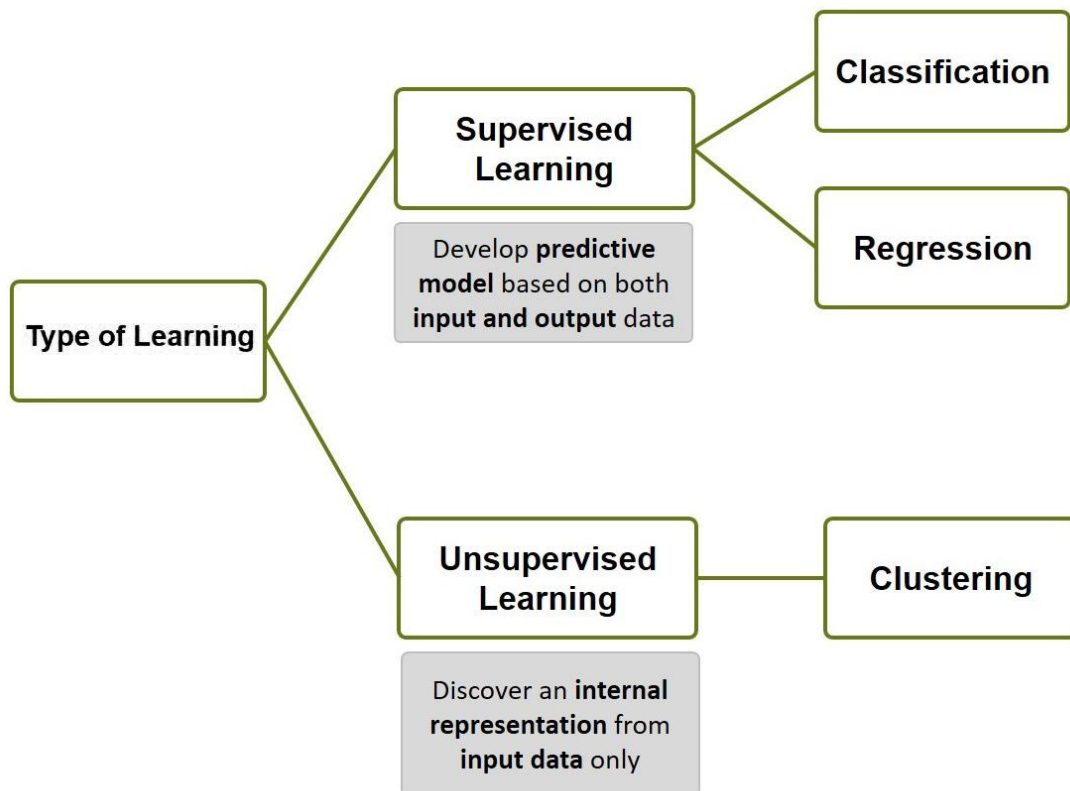


Figure 1-8: Types of training

- **Supervised training** is when the network is trained by providing it a set of inputs along with the anticipated outputs from each of these samples. Supervised training is the most common form of neural network training. As supervised training proceeds, the neural network is taken through a number of iterations, or epochs, until the output of the neural network matches the anticipated output, with a reasonably small rate of error. Each epoch is one pass through the training samples (see Figure 1-9).

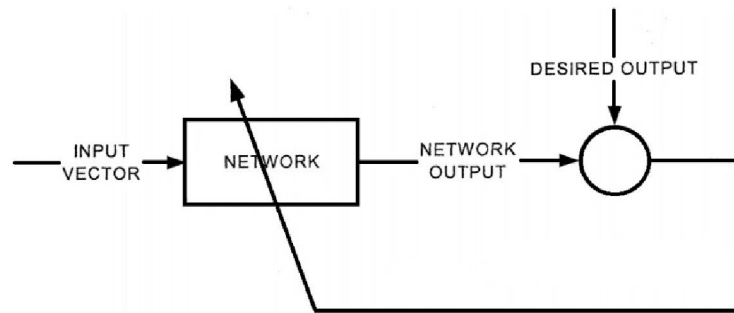


Figure 1-9 : Supervised learning scheme [9]

• **Unsupervised training** is similar to supervised training, except that no anticipated outputs are provided. Unsupervised training usually occurs when the neural network is being used to classify inputs into several groups. The training involves many epochs, just as in supervised training. As the training progresses, the classification groups are “discovered” by the neural network (see Figure 1-10).

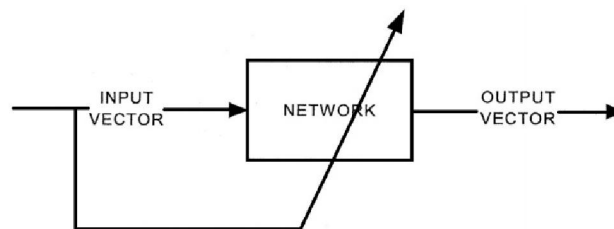


Figure 1-10: Unsupervised learning scheme [9]

There are several hybrid methods that combine aspects of both supervised and unsupervised training. One such method is called *reinforcement training*. In this method, a neural network is provided with sample data that does not contain anticipated outputs, as is done with unsupervised training. However, for each output, the neural network is told whether the output was right or wrong given the input (see Figure 1-11).

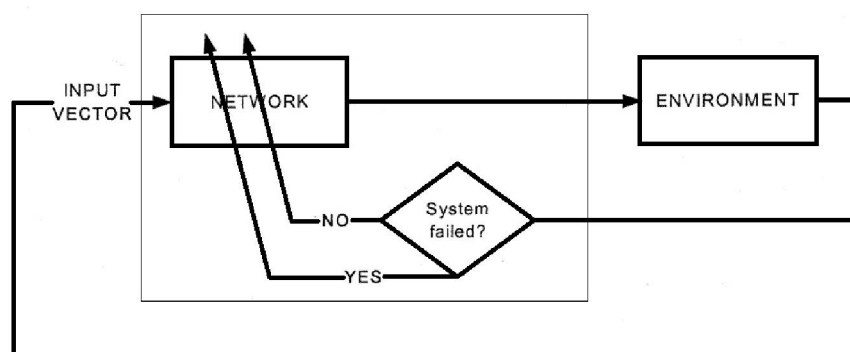


Figure 1-11: Reinforcement learning scheme [9]

It is very important to understand how to properly train a neural network. Once the neural network is trained, it must be validated to see if it is ready for use.

1.4.3.2 Backpropagation algorithm in multilayer perceptron networks (MLP):

An MLP network is designed to perform a desired task defined by a learning database. Each element of this database is called a learning example and it is in the form of a pair (X, Y^*) where X is an input value of the network and Y^* is the corresponding output target value. The network architecture, the structure of its connections, as well as the activation functions, can be set according to the task to be performed by the network.

The aim of learning is therefore to determine the values W^* of the matrix W of the weights of the network connections so that the output Y is close to the target value Y^* .

The algorithm of gradient backpropagation is widely known and most used in applications of MLP neural networks. This supervised learning technique uses a gradient descent procedure, working on the quadratic error between the actual output of the network and the target output. It calculates the partial derivatives of the output error with respect to all network weights and then applies a gradient procedure to minimize the error. At each iteration, an example of learning (X, Y^*) is retrieved and the weights are updated. This iteration is carried out in two phases:

1. Forward Propagation :

At each iteration, an element of the training set is introduced through the input layer. The evaluation of the network outputs takes place layer by layer, from the input of the network to its output.

2. Back propagation :

This step is similar to the previous one. However, the calculation is done in the opposite direction.

At the output of the network, the performance criterion is formed as a function of the actual output of the system and its target value. Then, the gradient of this performance is evaluated with respect to the different weights, starting with the exit layer and going up to the input layer.

1.5 The properties of neural networks

The main interest in neural networks is justified in the following properties:

- **Learning capacity:**

Learning ability refers to the ability of the neural network to learn to solve problems from examples in a similar way to humans or animals

- **The generalization capacity:**

The ability to generalize translates into the ability of a system to learn and retrieve from a set of examples rules that solve a given problem not learned.

- **The parallelism:**

This notion is at the basis of the architecture of neural networks considered as a set of elementary entities that work simultaneously. Parallelism allows higher computational speed but requires thinking and posing problems differently.

1.6 Areas of application of neural networks

Being at the intersection of different domains (computer science, electronics, cognitive science, neurobiology and even philosophy), the study of neural networks is a promising avenue of Artificial Intelligence, which has applications in many areas:

- **Defense:** Weapons management, target tracking, radars: processing, compression, noise suppression, signal / image identification, etc.
- **Industry:** quality control, process control, fault diagnosis, correlations between data provided by different sensors, handwritten signature or writing analysis, speech synthesis, automated vehicle guidance system, etc.
- **Entertainment:** Animation, special effects.
- **Finance:** Forecasting and modeling of the market (currencies ...), forecasting of economic indicators, selection of investments, credit allocation, forecasting of prices, etc.
- **Telecommunications and data processing:** signal analysis, noise cancellation, recognition of shapes (noises, images and lyrics), data compression, etc.
- **Medical:** analysis of EEG signals, ECG, prostheses, cancer analysis, etc.
- **Environment:** risk assessment, chemical analysis, forecasting and weather modeling, resource management, etc.

1.7 Conclusion

To conclude with, the artificial neural networks characteristics inspired from the human brain (parallelism, nonlinearity, and learning) allowed them to perform effectively in many tasks, where a conventional digital computer may have had a hard time. They are being used more and more in many fields because of their robustness and plasticity of architecture. However despite the fact that research in neural networks is an open field, the question is whether it will last long like that, knowing that neural networks have some big challenges like:

- 1- The actual model is too simple compared to the complexity of the human brain, which means that it stills far from being able to behave like a real brain.
- 2- There must be a technology that allows the implementation of complex neural networks models.

This leads us to other questions, like: what is the actual technology used in implementing actual neural networks models? And what are the techniques used in these implementations?

CHAPTER 2

CHAPTER 2. FPGA ARCHITECTURE

2.1 Introduction

Over the last years, in order to face the hardware implementation issues, the modern electronics is increasingly turning to digital, which has many advantages over the analog. Although the growing development of the electronics nowadays, there is still some architecture designs which are challenging. Some of these architectures such as ANNs have presented so many implementation difficulties.

Since the architecture of ANN requires the parallelism, there was many attempts to build that architecture on ASIC boards which have some parallel processing units [10]. However, many limitations, related to reconfigurability and to the size of the network, have appeared. Circuits such as FPGAs have been showed up with their flexibility in design like software's but with performance speed closer to ASIC. These circuits have represented a natural fit, offering a parallelism which helps for the implementation of the ANN.

In this chapter, we discuss all the types of the available digital circuits then we mainly introduce the architecture of the FPGA circuits which are considered as the best choice among all the circuits for ANNs. We began with a classification of the digital circuits then a brief history about the FPGA circuits and then we detailed their architecture and presented the board Nexys 2 development card that we used to develop our project. Finally, we presented the VHDL language we used to design our application.

2.2 The classification of digital circuits

There are three main types of digital circuits. First, standard logic circuits that include combinatorial circuits and flip-flops. Then, the programmable circuits include microprocessors, microcontrollers and DSPs (Digital Signal Processor). In this category there is an arithmetic and logic unit that executes a program located in a program memory using synchronization clock. Finally, circuits with a programmable architecture such as ASICs (Application Specific Integrated Circuit), PLDs (Programmable Logic Device) and FPGAs (Field Programmable Gate Arrays) form the third category [11]. In this category, we design a circuit that corresponds to our own needs, the use of a clock is not compulsory, and we can easily implement applications that require parallelism (Figure 1-11).

2.2.1 Circuits with programmable architecture [10]

2.2.1.1 ASIC (Application Specific Integrated Circuit)

By definition, ASIC circuits include all circuits whose function can be customized in one way or another for a specific application, as opposed to standard circuits whose function is defined and perfectly described in the catalog of components. The use of an ASIC leads to many advantages, mainly due to the reduction of the size of the systems such as the reduction of the number of components on the printed circuit, the consumption and the space requirement. The ASIC concept ensures maximum optimization of the circuit to be realized. Finally, we have an integrated circuit that really corresponds to our own needs which gives the designer a confidentiality and industrial protection. The major disadvantage of ASICs lies in the fact that the passage to the founder is obligatory, which entails high costs and a high development time of the circuit. As a result, ASICs are generally more suitable for mass production of designs already verified and not for prototypes.

2.2.1.2 PLD (Programmable Logic Device)

PLDs are chips that can be programmed to behave like an arbitrary design. A PLD can be programmed for an implementation as simple as a combinatorial logic operation as it could be for much larger designs. It typically includes AND gate array connected to an array of OR gates [12].

2.2.1.3 FPGA (Field Programmable Gate Arrays)

FPGAs are completely reconfigurable components which allow them to be reprogrammed at will, in order to accelerate some calculations. They consist of a matrix of programmable logic blocks surrounded by programmable input/output blocks. They are all connected by a programmable interconnection network. The FPGAs are not optimized for a specific application, therefore they consume more power than ASICs. On the other hand, they are much simpler to be programmed and reprogrammed, which shortens the design cycles and allows following the evolution of the application for which they were designed. The advantage of this type of circuit is its great flexibility, which makes it possible to reuse them at will in different algorithms in a very short time. The FPGAs are more suitable for prototypes and for limited mass productions that are not of the quality of ASICs. This technology also permits the implementation of a large number of applications and offers a low-cost hardware installation solution for small companies for whom the cost of developing a specific integrated circuit involves too much investment. The major disadvantage of FPGA circuits is that they are not very secure in terms of confidentiality, since it is enough to analyze the contents of the associated ROM to go back to the imagined schematics [13].

In the proposed algorithm, the parallelism is required, for the implementation of neural networks; we have chosen a Xilinx Nexys 2FPGA circuit to implement these networks.

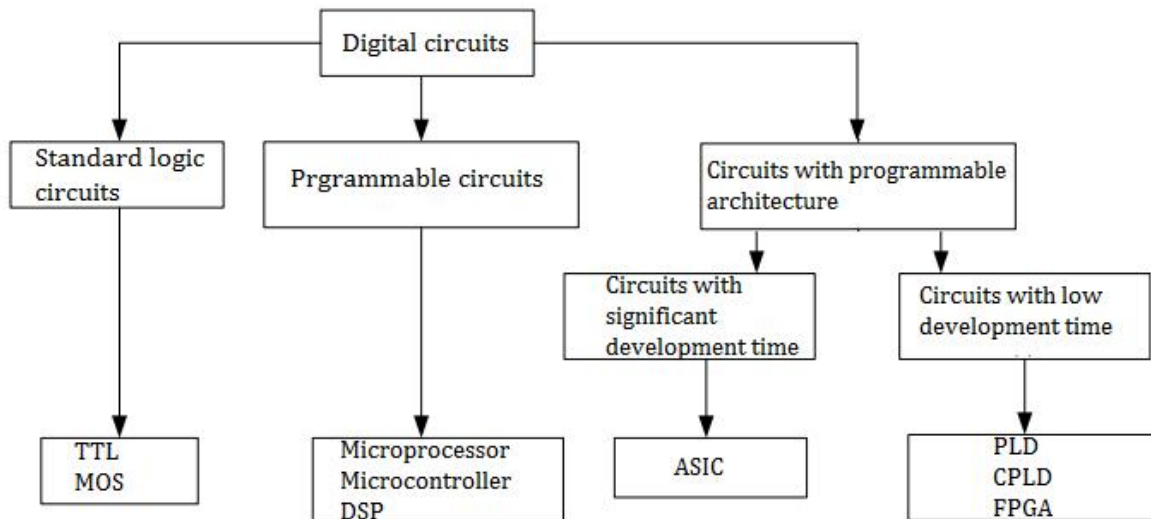


Figure 2-1: Classification of digital circuits

2.3 The FPGA circuit

2.3.1 History

The principle of the programmable logic dates back to the early 1960s, the concept has been proposed by Estrin. But it was not until the 1980s that the first material implementation was introduced into the market. The advent of this type of circuit was first made through simple Programmable Array Logic (PAL) circuits which are programmed as non-volatile ROM-type memories and are used to implement simple combinatorial functions such as address decoders or bus controllers.

With microelectronics evolutions, various families of programmable circuits have emerged: the Complex Logic Programmable Devices (CPLDs), then the Field Programmable Gate Arrays (FPGAs) introduced by Xilinx in 1985. With the emergence of increasingly efficient circuits that can be programmed at will, the industrialization and marketing of this type of circuit has taken a place on a large scale. At present there are a dozen manufacturers, the market is being clearly dominated by Xilinx, Altera (reprogrammable circuits) and Actel (non-reprogrammable circuits [14]).

2.3.2 Application [15]

Taking advantage of the ever increasing density of the chips, the FPGAs are used for several applications such as telecommunications, image and signal processing. More recently, other application fields are in growing demand, such as medical equipment, robotics, automotive, and space and aircraft embedded control system. Finally, industrial electrical control systems are also of great interest because the ever-increasing level of expected performance while at the same time reducing the cost of the control systems.

2.3.3 FPGA Architecture

The architecture of an FPGA is divided into:

- Processing resources including memories, logic and registers. They are grouped into logical blocks of different types (CLB,IOB).
- The programmable routing resources that connect the logic blocks together.

The programming of a reconfigurable circuit therefore consists in specifying the functionality of each logic block and in organizing the interconnection network in order to perform the requested function. Some FPGAs also incorporate RAMs, Multipliers, and even processor cores [16].

A generalized example of an FPGA is shown in Figure 2-2 where CLBs (Configurable Logic Block) are arranged in a two dimensional grid and are interconnected by a programmable routing resources. The CLBs are surrounded by IOB blocks which are arranged at the periphery of the grid and they are also connected to the routing interconnection [17],[18].

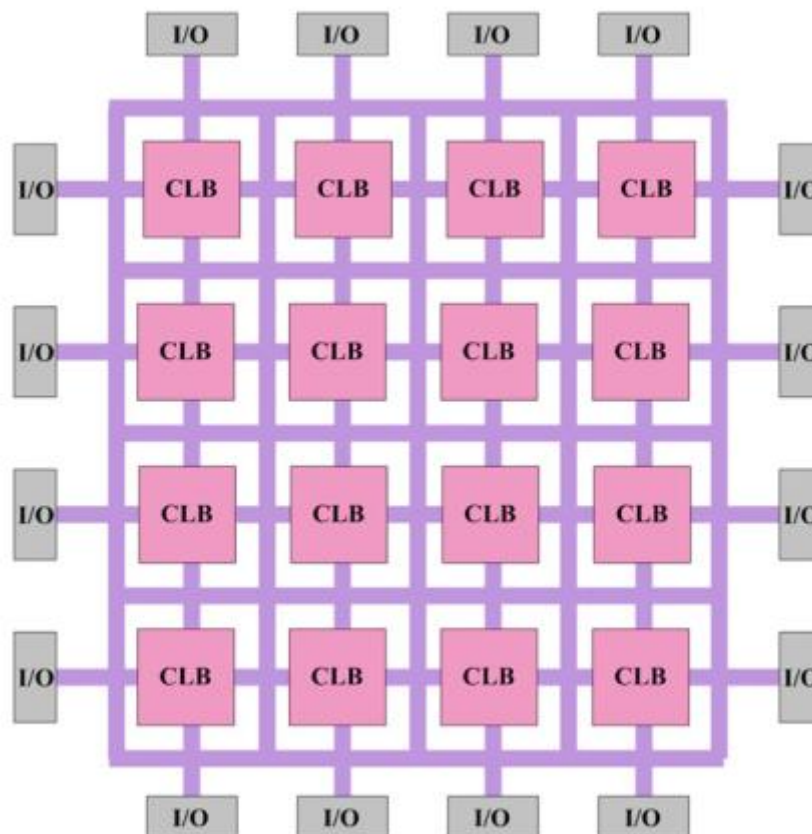


Figure 2-2: Overview of FPGA architecture

2.3.3.1 Configurable Logic Block (CLB):

The Configurable Logic Block is a basic component of an FPGA; it provides the basic logic and storage functionality for a target application design. In order to provide the appropriate basic logic and storage capability, the basic logic can be either a transistor or an entire processor. In between these two extremes exists a spectrum of basic logic blocks. Some of them include logic blocks that are made of NAND gates, an interconnection of multiplexors, lookup table (LUT) ...etc.

The choice of the logic block depends on performance, power consumption and the amount of the programmable interconnect ...etc.

A CLB can comprise of a single BLE (Basic Logic Element), or a group of interconnected BLEs. A simple BLE consist of a LUT, and a flip-flop. A LUT with k inputs contains 2^k configurations bits and it can implement any K -input Boolean function. The Figure 2-3 shows an example for $k=4$ [16].

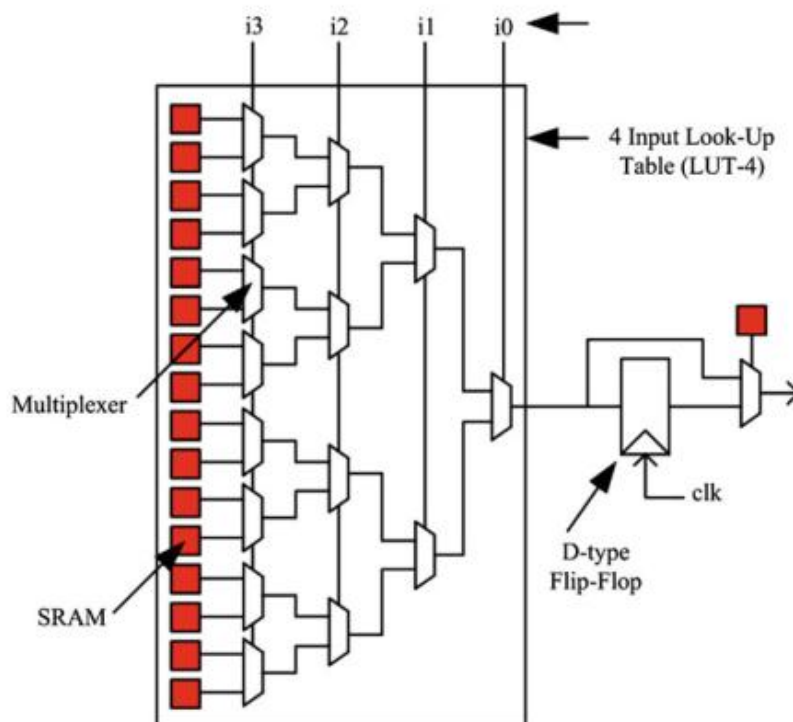


Figure 2-3 : Basic Logic Element (BLE)

The LUT in that example uses 16 SRAM (SRAM are explained in next sections) bits to implement any 4 inputs Boolean function.

A CLB can contain a cluster of BLEs connected through a local routing network. Figure 2-4 shows a cluster of 4 BLEs of 4 inputs, each BLE contains a LUT and a Flip-Flop. The BLE output is accessible to other BLEs from the same group through a local routing network. The numbers of output pins of a cluster are equal to the number of BLEs in the cluster. However the numbers of input pins in a cluster can be less than or equal to the sum of pins required by all the BLE in the cluster [16].

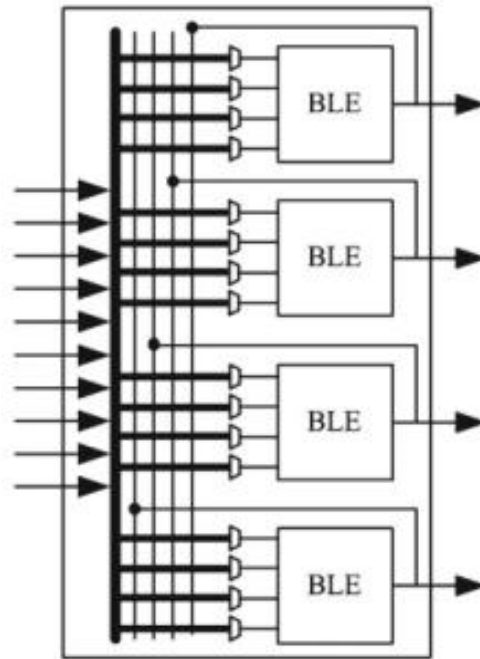


Figure 2-4 : A configurable Logic Block (CLB) having four BLEs

2.3.3.2 Input Output Block (IOB):

The IOBs allow the interface between the FPGA component pins and the internal logic developed inside the component. They are present the entire periphery of the FPGA circuit. Each IOB block command a component pin and can be defined as input, output, bidirectional signal or unused (high impedance) [11].

2.3.3.3 Routing interconnections:

As we discussed earlier, the computing functionality is provided by its programmable logic blocks which are connected to each other through a programming routing network. This programmable routing network provides routing connection among logic blocks and IOB blocks to implement any user-defined circuit. The routing interconnect of an FPGA consists of wires and programmable switches that form the required connection. These programmable switches are configured using the programmable technology.

To assure a variety of the reprogrammable circuits, the FPGA routing interconnect must be very flexible. Although the routing requirements vary from a circuit to another, certain common characteristics of these circuits can be used optimally design the routing interconnect of FPGA architecture. For example most of designs exhibit locality, hence requiring plenty of short wires. But at the same time there are some distant connections, which lead to the use of long wires [16].

The arrangement of the routing resources plays an important role in the overall efficiency of the FPGA Architecture. This arrangement here considered as global routing architecture

whereas the microscopic details regarding the switches topology of different switch blocks is considered as detailed routing architecture.

2.3.3.4 Programming technology:

There are numerous programming technologies that have been used for reconfigurable architectures. Each of these technologies has different characteristics that, in turn, have a significant impact on the programmable architecture. Some of the well-known technologies include static, flash and anti-fuse memory.

In this section we discuss only the SRAM based Programming technology since it is the most used technology by the commercial vendors.

SRAM based programming technology

Static memory cells are the basic cells used for SRAM-based FPGAs. Most commercial vendors use static memory (SRAM) based programming technology in their devices. These devices use static memory cells that are distributed throughout the FPGA to ensure configurability. An example of this memory cell is shown in the Figure 2-5. In an SRAM-based FPGA, SRAM cells are mainly used for the following purposes:

- To program the routing interconnect of FPGAs which are generally steered by small multiplexors.
- To program Configurable Logic Blocks (CLBs) used to implement logic functions.

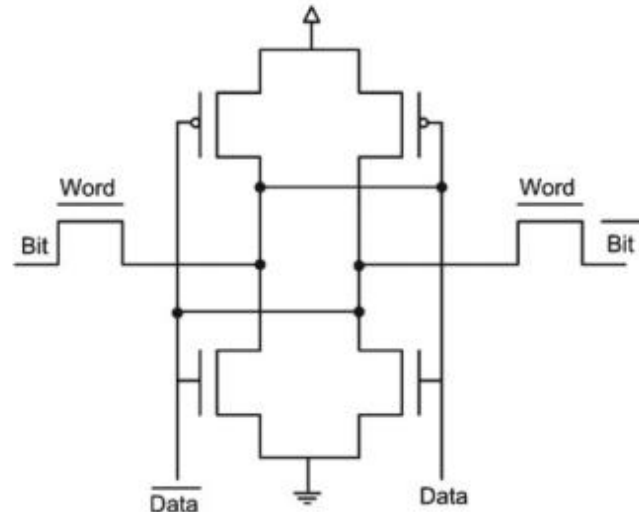


Figure 2-5 : Static Memory cell

SRAM-based programming technology has become the dominant approach for FPGAs because of its re-programmability and the use of standard CMOS process technology and therefore leading to increased integration, higher speed and lower dynamic power consumption of new process with smaller geometry. There is however a number of drawbacks associated with SRAM-based programming technology. For example an SRAM cell requires 6 transistors which make the use of this technology costly in terms of area compared to other

programming technologies. Further SRAM cells are volatile in nature and external devices are required to permanently store the configuration data. These external devices add to the cost and area overhead of SRAM-based FPGAs.

2.4 The circuit board Nexys 2[19]

The Nexys2 circuit board is a complete, ready-to-use circuit development platform based on a Xilinx Spartan 3E FPGA. It has been used to implement the proposed algorithm by providing a complete application development solution on Xilinx's Spartan 3E family. It uses the circuit "FPGA XC3S500E-FGG320" which belongs to the Spartan 3E family of Xilinx. The high integration density of the logic gates and the large number of inputs / outputs available for the user allow the implementation of complete systems on the FPGA board. This board offers a design environment that is very suitable for varied application prototyping, including those of general purpose digital systems and embedded systems. It provides a complete application development solution and is also ideal for video control, video processing and signal processing applications in general. Figure 2-6 shows the board components.

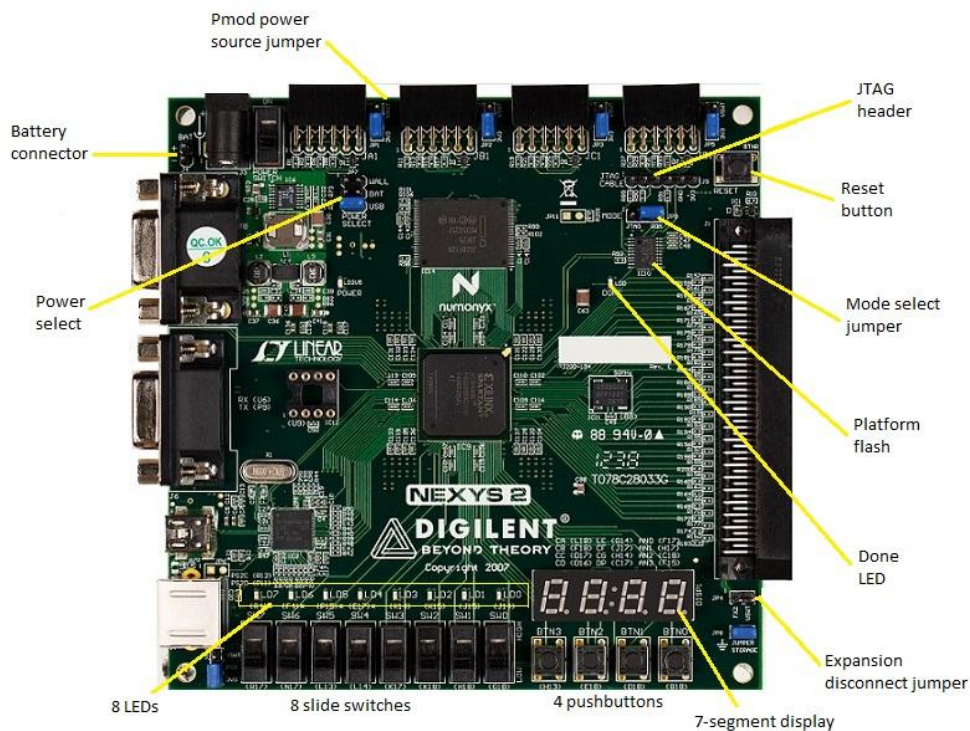


Figure 2-6 : Nexys 2 board FPGA

Its onboard high-speed USB2 port, 16Mbytes of RAM and ROM, and several I/O devices and ports make it an ideal platform for digital systems of all kinds, including embedded processor systems based on Xilinx's MicroBlaze. The USB2 port provides board power and a programming interface, so the Nexys2 board can be used with a notebook computer to create a truly portable design station.

2.4.1 Spartan 3E architecture

The Spartan-3E family architecture consists of five fundamental programmable functional elements [20] :

Configurable Logic Blocks (CLBs): contain flexible Look-Up Tables (LUTs) that implement logic plus storage elements used as flip-flops or latches. CLBs perform a wide variety of logical functions as well as store data.

Input/output Blocks (IOBs): control the flow of data between the I/O pins and the internal logic of the device. Each IOB supports bidirectional data flow plus 3-state operation. Supports a variety of signal standards, including four high-performance differential standards. Double Data-Rate (DDR) registers are included.

Block RAM: provides data storage in the form of 18-Kbit dual-port blocks.

Multiplier Blocks: accept two 18-bit binary numbers as inputs and calculate the product.

Digital Clock Manager (DCM): Blocks provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase-shifting clock signals.

These elements are organized as shown in Figure 2-. A ring of IOBs surrounds a regular array of CLBs. Each device has two columns of block RAM. Each RAM column consists of several 18-Kbit RAM blocks. Each block RAM is associated with a dedicated multiplier. The DCMs are positioned in the center with two at the top and two at the bottom of the device.

The Spartan-3E family features a rich network of traces that interconnect all five functional elements, transmitting signals among them. Each functional element has an associated switch matrix that permits multiple connections to the routing.

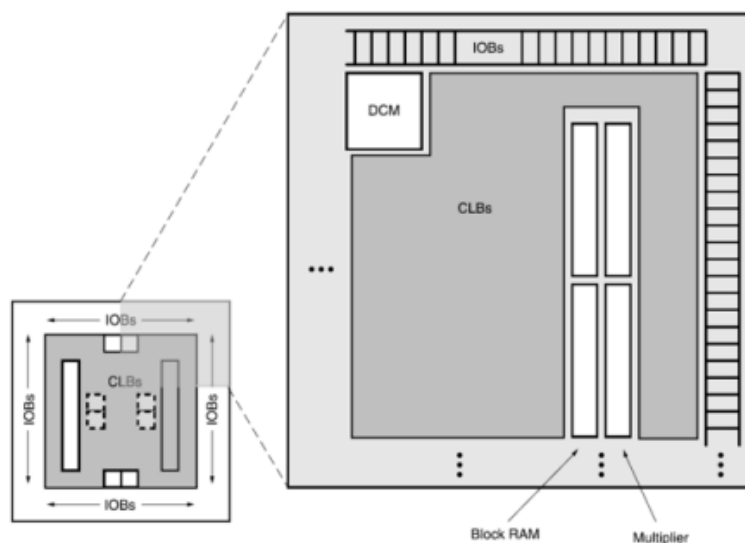


Figure 2-7 : Spartan E3 family architecture

2.5 VHDL

2.5.1 Brief on VHDL

The VHDL is a hardware description language. It describes the behavior of an electronic circuit, from which the physically circuit can then be implemented. The VHDL stands for VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuits.

An initiative funded by the US Department of Defense in the 1980s that led to the creation of VHDL. Its objective was to describe the complex circuits, in order to establish a common language with its suppliers.

Its first version was the VHDL 87, later an upgrade to the so-called VHDL 93. The VHDL was the first hardware description language standardized by the Institute of Electrical and Electronics Engineers IEEE, thanks to the IEEE 1076. An additional standard, the IEEE 1164, was then added to introduce multi-valued logic systems.

2.5.2 Utility of VHDL

The VHDL is a language of specification, de simulation and also design. Unlike other languages (CUP, ABEL) that were primarily design languages; the VHDL is primarily a specification language. The standardization first took place for specification and simulation (1987) and then for synthesis (1993).

Specification:

It is in this field that the standard is currently the most established. It's quite possible to describe a circuit by a standard VHDL code so that it is readable everywhere. This ability to describe circuits in a universal language is also very practical to avoid language problems.

Simulation:

The VHDL is also a simulation language. To do so, the notion of time, in different forms, has been introduced. Modules, intended only for simulation, can thus be created and used to validate a logical or temporal operation of the VHDL code. The ability of simulating with VHDL programs should considerably facilitate the writing of tests on a prototype which are much more expensive and whose errors are more difficult to find.

Conception:

The VHDL also allows circuit design. The two main immediate applications of VHDL are in the field of programmable logic circuits, including CPLD and FPGA, and in the field of ASIC circuits. The VHDL is intended for circuit synthesis as well as circuit simulation. However, since this language is designed primarily for specification and then for simulation, as a result some language variants are not yet usable for design.

2.5.3 VHDL structure

As illustrated in **Figure 2-8** the typical structure of a VHDL description is composed of at least three fundamental parts:

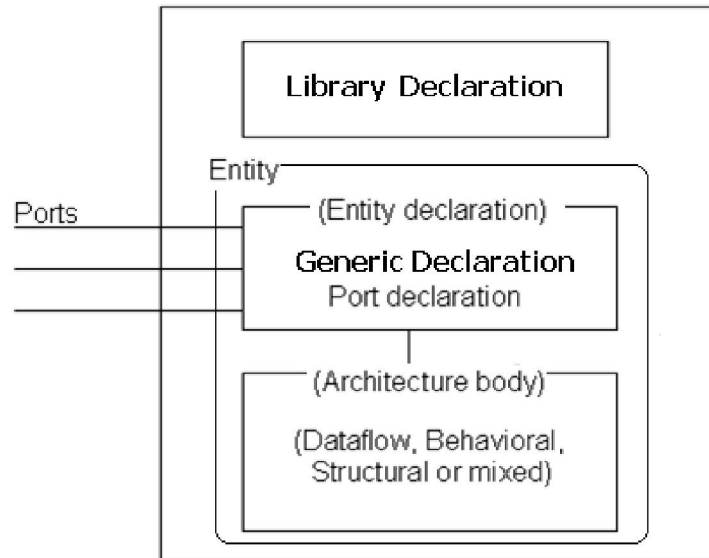


Figure 2-8 : Basic structure of VHDL description

Libraries declaration:

Any description used in the VHDL code must be defined in a library. The main libraries are standardized by IEEE. They contain the definitions of the types of electronics signals, functions and subprograms used to perform arithmetic and logic operations, and so on. The “use” directive is used to select which libraries to use.

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

The entity:

It represents an external view of the description. The declaration of the entity makes it possible to define the name of the VHDL description as well as the inputs and outputs used, the instruction that defines them is “port”.

```
entity entity_name is
port(
    -- All the ports are declared here
);
end entity;
```


The architecture:

Contains the appropriate VHDL code, which describes how the circuit should behave to perform the expected operation. It represents the internal structure of the description.

```
Architecture bhv of entity_name is
  -- Declaration of internal signals
begin
  -- Concurrent instructions
  process(--sensitivity list)
  begin
    -- Sequential instructions
  end process;
end architecture;
```

2.5.4 The modes used in VHDL

The VHDL uses two modes of operation: combinatorial mode (concurrent mode) and sequential mode. Each of these modes is used in specific cases.

Combinatorial mode:

In combinatorial mode, all instructions in a VHDL description are evaluated and affect the output signals at the same time (in parallel), so the order in which the instructions are written is irrelevant. Indeed the description generates electronic structures, it is the great difference between a VHDL description and a classical computer language. So with VHDL you have to try to think of the structure that will be generated by the synthesizer to write a good description.

Sequential mode:

The sequential mode uses “process” in which time is an essential variable. A process is a part of the description of a circuit in which the instructions are executed sequentially, that is to say one after the other. It can perform signal operations using the standard instructions of structured programming as in microprocessor systems.

2.6 Conclusion

To be able to implement a ANN in an FPGA circuit, one should have a minimum knowledge on them. thus in this chapter, FPGA circuits have been described and they proved to be a good choice for implementation of architectures such as the ANN's. We have presented the architecture of FPGA circuits, specifically Xilinx Spartan E3. According to the architecture study, FPGAs are reconfigurable components, they are generally constituted by a programmable logic block matrix (CLB) surrounded by programmable Input/ Output blocks (IOB). These two blocks can be connected by a network of programmable interconnections. In addition, the Xilinx Spartan E3 contains two other blocks, which are the Multiplier block and the Digital Clock Manager (DCM) block. Before that we had introduced briefly the different type of digital circuits. Finally, the VHDL description language, which is a language of a specification, simulation and design, has also been presented at the end of this chapter.

CHAPTER 3

CHAPTER 3. FPGA IMPLEMENTATION OF NEURAL NETWORKS

3.1 Introduction

ANNs are becoming more popular these years; this is due to the growing interest in their applications. However implementing them in the traditional way by software running in a general-purpose processor couldn't really meet the real-time requirements in many cases, especially in intelligence control. In contrast to it the hardware implementation allows neural networks to take full advantage of their inherent parallelism and run orders of magnitude faster than software.

The hardware implementation of neural networks can be realized using either analog or digital hardware; still the latter is the most appropriate and popular as it has many qualities among which we state higher accuracy, better repeatability and testability, lower noise sensitivity, and higher flexibility and compatibility with other types of processors.

Furthermore as it is stated in the previous chapter, the FPGA showed up to be the most suitable for ANN implementation as it preserves the parallelism, flexibility and reconfigurability in the neuron's architecture.

However there are some challenges and certain tradeoffs that must be dealt with in order to implement Neural Networks on FPGAs. Generally it requires large resource because of nonlinear activation functions and several synaptic weights (multipliers) present in the network. It is a major problem where there should be a balance between precision and speed, and the cost of more FPGA resources (logic areas) associated with increased precision.

This chapter proposes a C⁺⁺ based application that generates descriptions of Feed Forward Layered ANNs in form of VHDL files, to be implemented in an FPGA, the ANNs descriptions are based on a simple architecture that allows implementing large neural networks with minimum recourses which is layer multiplexing. The chapter starts by introducing some arithmetic issues in implementing neural networks (data representation, products computation, sum of products, activation function). After that a general architecture of ANN generated by our application and its different main blocks are presented. Then it is concluded by a brief overview on the essential of our work that is the C⁺⁺ based application.

3.2 Arithmetic in ANN digital implementation

There are several aspects to take into account when designing ANNs circuits; these include data representation, products and sum of products computations, and activation functions

implementations. The most important are inner-products and non-linear activation function, because they are area-greedy. Indeed the latter is the most complex, and naturally the one that consumes FPGA resources the most. It has an entire separate section devoted just to it. Given the ease with which arithmetic operations can be implemented, the activation function stays the biggest limiting factor in performance.

3.2.1.1 Data representation:

A neural network operates with real numbers; it can be represented in many ways. However the problem is how to balance between numeric precision, which is important for the accuracy and convergence and FPGA resources.

The simple precision floating point representation is ideal since it offers the greatest amount of precision (i.e. minimal quantization error); however using it in ANN implementation on FPGAs is not feasible since it consumes huge amounts of resources which are very limited in this case.

Instead of Floating point, fixed point representation can be used. Even though this means less precision, its benefits compromise its inconveniences. It is more area-efficient than floating point, and much simpler in arithmetic operations. It comes naturally to use two's complement representation for negative numbers. It was mentioned in [10] that many studies established 16 bits for weight and 8 bits for activation-function as good enough. In the coming proposed architecture, 15 bits are used for weights and 9 bits for the activation-function input.

3.2.1.2 Inner products:

Multiplayer has been identified as the most area-intensive arithmetic operator used in FPGA-based ANNs, there many forms of multipliers:

Bit-serial multipliers: In this one the calculation is done by a bit at a time, whereas fully parallel multipliers calculate all bits simultaneously. Thus the first one can scale to a signal representation of any range-precision, however, this means that the bit-serial multiplication time grow quadratically, with the length of the signal representation. This means that it is not effective in real time applications of ANN.

Other ways have been tried, like imposing the values of synaptic weights into powers of two values, so that the products get simplified to numbers of shifts. Sadly this type of design practice reduces drastically the ANN performance.

One last method to be used is direct full parallel-bit multiplier; FPGAs have limited numbers of these multipliers in different dimensions. In Spartan 3E, there are only 20 multipliers of dimension (18×18) . In the architecture proposed in next sections, only 6 multipliers have been used in parallel, this allows the ANN VHD description generated by the proposed C++ application, to have in the same time good parallelism and to be implemented in low resource FPGAs too.

3.2.1.3 Sum of products

The number of adder is not what lacks in today’s FPGAs, and the sum of products can be carried in many ways. In ANN the sum of products is used to cumulate the elementary products of the neuron inputs with their associated synaptic weights, $\sum_{k=1}^n W_{ik} X_k$. Its implementation depends on the number of elementary products to be accumulated. If n is small enough then a direct implementation like the one in Figure 3-1 or Figure 3-2 can be used.

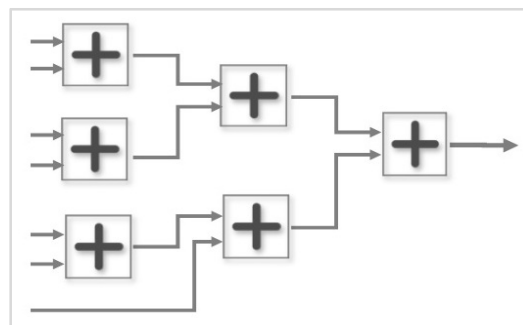


Figure 3-1: Sum of products 1

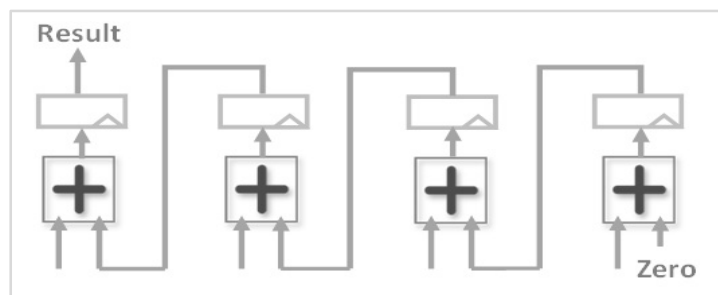


Figure 3-2: Sum of Products 2

As it is stated previously in the proposed ANN implementation only 6 products can be computed at the same time, one multiplier per neuron at a time, it receives the neuron’s associated inputs and multiply them by their appropriate weight in a sequential way, this method causes more latency in computation, but in the other hand it is more area-efficient and preserves the parallelism characteristic between neurons. So the elementary products for each neuron can be summed using just one adder with a feedback loop since they come out of the multipliers sequentially, see the Figure 3-3.

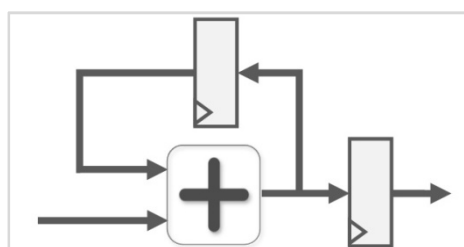


Figure 3-3: Serial sum of products

3.3 The Activation Function implementation

The activation function implementation is one of the most important arithmetic-design issues when implementing ANNs on FPGA boards. There exist many types of activation functions by now, see chapter one. The most interesting ones for our application are Tangent sigmoid, Log sigmoid and linear activation function. For the implementation of the first two functions which are nonlinear, many works have been proposed [10],[21], [22],[23] and more. Mainly there are the next five implementation approaches: piecewise linear approximation (PWL), piecewise nonlinear approximation, lookup table (LUT), bit-level mapping, and hybrid methods.

Generally *piecewise linear approximation* uses a series of linear segments to approximate the activation function. The number and locations of these segments are chosen such that error, processing time, and area utilization are minimized. The use of multipliers should be avoided for efficient hardware implementations employing this approximation method, as multipliers are expensive hardware components in terms of area and delay. This method is used in [23] for the hyperbolic tangent and sigmoid function implementation.

The *piecewise nonlinear approximation* is similar to the PWL method except that a nonlinear approximation is used in each segment. One example of it is polynomial approximations. Although High order polynomial approximations can give low-error implementations, they are generally not suitable for hardware implementation, because of the number of arithmetic operations

In the *LUT-based methods*, input range is divided to equal sub-ranges and each sub-range is approximated by a value stored in LUT. This method is used to implement the hyperbolic tangent.

Hybrid methods usually combine two or more of the previously mentioned methods to achieve better performance. The main challenges have always been one of how to choose the best interpolation points and how to ensure that look-up tables remain small. This approach is used in [22] to implement tangent sigmoid.

In this work three activation functions are implemented, *Tangent sigmoid*, *log sigmoid*, and *linear activation function*. The approach adopted to implement the nonlinear ones is the last one, the hybrid method that uses lookup table coupled with a piecewise linear approximation. It is mainly based on the work presented by Promod Kumar Meher in his article entitled “*An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks*”[22].

3.3.1 Implementation of Tangent Sigmoid and Log sigmoid

This function is defined by the following equation:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (3-1)$$

This equation produces an S-shape curve presented in Figure 3-4

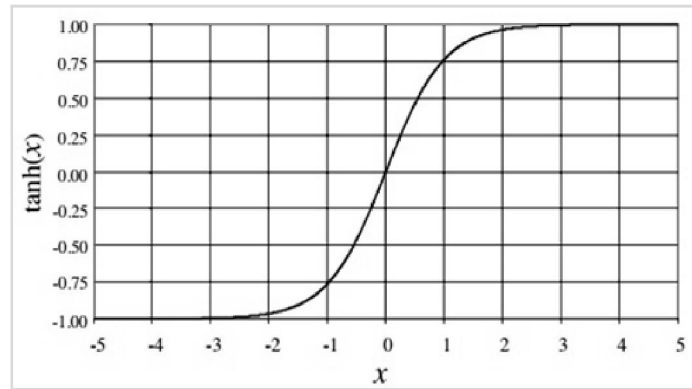


Figure 3-4: Hyperbolic Tangent Sigmoid Activation Function

As stated at before to implement this function, an optimized lookup table approach proposed in [22] is used, and the challenge in this is how divide intervals to ensure that look-up tables remain small, and result with minimum error.

3.3.1.1 Characteristics of Tanh and strategy of implementation

- *Property 1*

$$\tanh(-x) = -\tanh(x) \quad (3-2)$$

Considering this property (mirror symmetry about Y-axis), only the right-half of the curve in Figure 3-4 should be stored, i.e., for $x \geq 0$, the negative values of the input could be evaluated by negating the LUT-words stored for its corresponding positive values. This is done by performing the 2's complement operation on the LUT output if the input is negative.

- *Property 2*

$$\lim_{x \rightarrow 0} \tanh(x) = x \quad (3-3)$$

This means that for small values of the input, $\tanh(x)$ is linear, then storing values in the LUT for this region could be avoided, since the corresponding $\tanh(x)$ values could be obtained directly from input values. This could be implemented by a simple Multiplexer.

- Property 3

$$\lim_{x \rightarrow \infty} \frac{d \tanh(x)}{dx} = 0 \quad (3-4)$$

$$\lim_{x \rightarrow \infty} \tanh(x) = 1 \quad (3-5)$$

According to the two equations the variation of the $\tanh(x)$ is insignificant for big values of inputs, that is to say that for $|x| \geq 3$ the variation is less than 0.00015 and a value of $+1$ can be stored in the LUT for all values above 3.

By using these properties all together, the values of $\tanh(x)$ that should be stored in the LUT are for $\delta min < x < \delta max$, where δmin and δmax represent the limiting values which could be derived from accuracy requirements; such as $|\tanh(\delta min) - (\delta min)| \leq \epsilon$, and $|\tanh(\delta max) - (1)| \leq \epsilon$, where ϵ is the maximum allowable error.

δmax is the bound of the input values above which the $\tanh(x)$ stored in the LUT is $+1$. It takes generally values smaller than 3 for the reason that $|\tanh(x) - \tanh(3)| < 0.00015$, which is a very higher accuracy than what is required for many applications, with neural networks included. The Figure 3-5 shows clearly that for $x = 2$, $\tanh(x) > 0.96$, for $x = 2.4$, $\tanh(x) > 0.98$, and for $x = 2.7$, $\tanh(x) > 0.99$. Thus, $\tanh(x)$ can be approximated by $+1$ for $x > \delta max$ where δmax is 2, 2.4, or 2.7 if the maximum allowable errors are 0.04, 0.02, or 0.01 respectively.

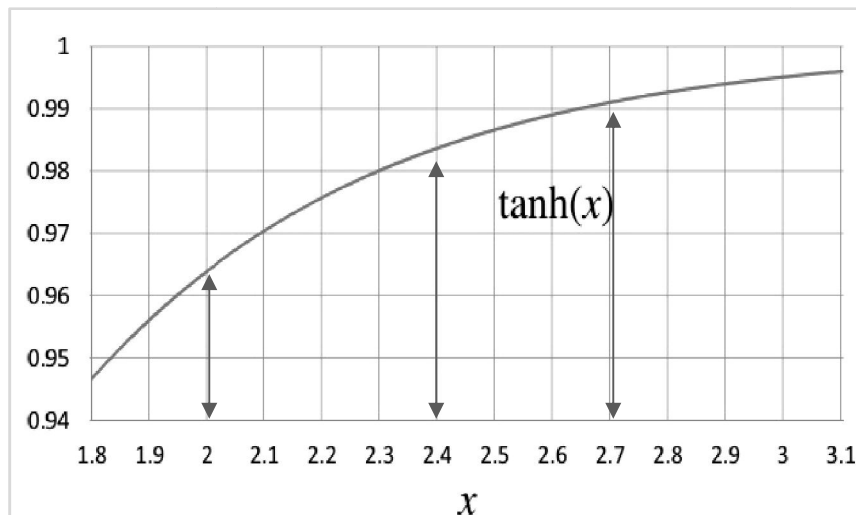


Figure 3-5: Saturation region of tangent hyperbolic

Similarly, to choose δmin we analyze the behavior of the $\tanh(x)$. The Taylor expansion of the $\tanh(x)$ near zero gives

$$\tanh(x) = x - \frac{x^3}{3} + \frac{x^5}{15} - \frac{17x^7}{315} + \dots, \quad x \rightarrow 0 \quad (3-6)$$

When x tends to zero high order terms can be ignored.

We can find δmin above which we can assume $\tanh(x) = x$ by using the nest equation:

$$\left| \delta min - \left(\delta min - \frac{\delta min^3}{3} + \frac{\delta min^5}{15} \right) \right| < \epsilon \quad (3-7)$$

By simplifying we find

$$\frac{\delta min^3}{3} - \frac{\delta min^5}{15} \leq \epsilon \quad (3-8)$$

For an error $\epsilon = 0.02$, δmin is found to be 0.390625 .

As seen in Figure 3-5, the rate of variation of $\tanh(x)$ when $\delta min < x < \delta max$ is not uniform, therefore all the values of inputs may not correspond to a different LUT value for a given accuracy. By knowing this, one single value can be stored for several input values forming sub-domains of the function.

3.3.1.2 Optimized Lookup table design for hyperbolic tangent

Unlike conventional lookup tables where each input value (address) word corresponds to one location in the LUT, Maher used in his work [22] used what is so called range-addressing, where one address corresponds to a range of input values that have the same value of $\tanh(x)$ stored in the LUT, reducing by that the number of words stored.

Apart from that, for a given sub-domain the value stored is the mean of the boundary values of the function in that sub-domain. This is unlike other works where they stored the function value corresponding to the lower-boundary address, and here the difference between the maximum and the minimum values of the function could be the double of the allowable error.

Designing the LUT is then done by following steps he proposed which are:

- 1) Determination of the upper and lower limits of LUT input (δmin and δmax):
For $\epsilon = 0.02$, $\delta min = 0.390625$ and $\delta max = 2.4$
- 2) Selection of the address width (precision):
By simulations in Matlab, it was found that a width of 9bits of input values represented in 2's complement allow to have $|\tanh(x_1) - \tanh(x_2)| \leq \epsilon = 0.02$ where x_1 and x_2 are two consecutive inputs.
- 3) Selection of Domain Boundaries:
The range of $\tanh(x)$ for $0.390625 < x < 2.4$ is divided into n sub-domains $R_i(x_{i1}, x_{i2})$ such that $|\tanh(x_{i1}) - \tanh(x_{i2})| \leq 2\epsilon$. n is determined such as the upper bound of the last domain (x_{n2})=2.4 then all (x_{i1}, x_{i2}) should be determined.
- 4) LUT assignment:
The stored is the mean of the boundary values of the function which means $\tanh(x_i) = [\tanh(x_{i1}) + \tanh(x_{i2})]/2$.

Based on these steps we developed a simple C++ program to generate the LUT words, results are given in the Table 3-1, unlike Maher work, the stored value is the nearest to the mean of the boundaries among its possible values that are imposed by the binary representation, which

mean that the error in a sub-domain can exceed the error imposed by the criterion $|\tanh(x_{i1}) - \tanh(x_{i2})| \leq 2\epsilon$. As we can see in the table the maximum error became 0.033

Table 3-1: LUT for the hyperbolic tangent activation Function

LUT word N°	The ranges limits		The mean $\tanh(x_i)$	Stored value	Stored value in binary	Maximum error
	x_{i1}	x_{i2}				
	...	0.390625	x	NA	NA	NA
1	0.390625	0.453125	0.398182	0.390625	00011001	0.0338394
2	0.453125	0.515625	0.44939	0.453125	00011101	0.0286606
3	0.515625	0.578125	0.497809	0.5	00100000	0.0256837
4	0.578125	0.640625	0.543313	0.546875	00100011	0.0255737
5	0.640625	0.703125	0.585836	0.578125	00100101	0.0282226
6	0.703125	0.78125	0.629886	0.625	00101000	0.0284236
7	0.78125	0.859375	0.67468	0.671875	00101011	0.0240605
8	0.859375	0.9375	0.715003	0.71875	00101110	0.0228145
9	0.9375	1.048675	0.757681	0.75	00110000	0.0312907
10	1.048675	1.171875	0.803082	0.796875	00110011	0.0279973
11	1.171875	1.328125	0.846831	0.84375	00110110	0.0250403
12	1.328125	1.53125	0.889714	0.890625	00111001	0.0218347
13	1.53125	1.859375	0.93163	0.9375	00111100	0.0268617
14	1.859375	2.90625	0.973329	0.96875	00111110	0.0252879
15	2.90625	NA	1	01000000	NA

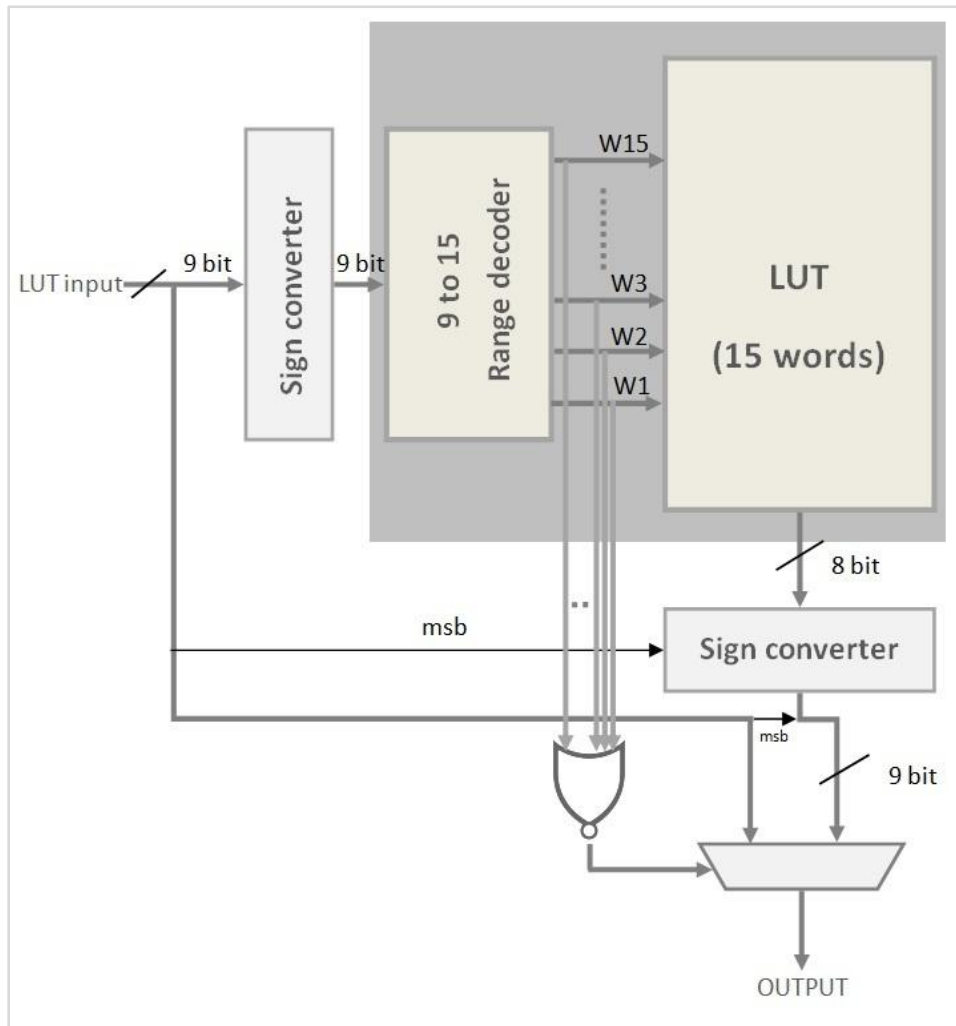


Figure 3-6 : Hyperbolic Tangent LUT block diagram

The Figure 3-6 shows that the hyperbolic tangent implementation consists of several blocks, that assure the functioning described previously.

A Sign converter: It has been implemented before and after the LUT bloc. The role of the first one is to calculate the magnitude of the input to feed it to the LUT, since this one deals with positive numbers only. So when the input is negative, the LUT word corresponding to the input magnitude is negated by the second sign converter to have the correct value $\tanh(x)$ of that negative input x .

A range decoder: it has been implemented in order to perform the range-addressing. That is to say that when it is fed with the input magnitude, it determines the corresponding range, through its word-select outputs (w_1, w_2, \dots, w_{15}).

A Multiplexer: its role is to let through either the LUT words or the input directly. The decision to let through the input happens in no word is selected by the range decoder, which means by reading the Table 3-1 that $|x| < 0.390625$. This could be implemented by using a NOR logic gate to all the word-select signals (range decoder outputs).

3.3.1.3 Implementation of Log-Sigmoid activation function:

We used the same previous implementation process to generate an optimized LUT for a sigmoid function:

$$\text{Sig}\alpha(x) = \frac{1}{1 + e^{-\alpha x}} \quad (3-9)$$

In our implementation we have chosen $\alpha = 2$ and that is to have an easy function to approximate its equation becomes:

$$\text{Sig}\alpha(x) = \frac{1}{1 + e^{-2x}} \quad (3-10)$$

This function has an S-shaped curve too; however unlike the tangent sigmoid this one ranges from 0 to 1 and not from -1 to 1 (See Figure 3-9a).

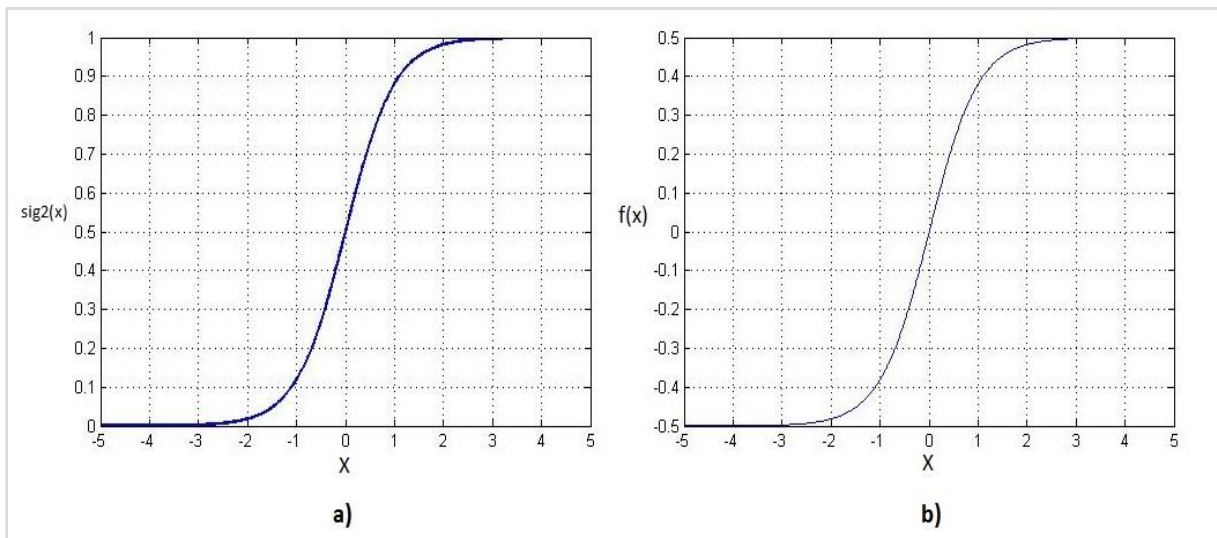


Figure 3-9: Sigmoid activation function

By doing the Taylor expansion to the first order we get:

$$\text{Sig2}(x) = \frac{1}{2} + x + \dots, \quad x \rightarrow 0 \quad (3-11)$$

To make the implementation easier, we considered implementing the function:

$$f(x) = \frac{1}{1 + e^{-2x}} - \frac{1}{2} \quad (3-12)$$

The new function has the shape shown in Figure 3-9b, clearly it has similar properties to $\text{anh}(x)$:

- $\lim_{x \rightarrow \infty} df(x)/dx = 0$
- $\lim_{x \rightarrow \infty} f(x) = 0.5$
- $f(-x) = -f(x)$
- $f(x) \approx x$ when $x \rightarrow 0$

After implementing the LUT of $f(x) = \frac{1}{1+e(-2x)} - \frac{1}{2}$ we can get the sigmoid by adding a 0.5 to its output.

We made a C++ code to generate the LUT for the function $f(x)$, the results are in the Table 3-2

Table 3-2 : LUT for $f(x) = \text{sig2}(x) - 0.5$

LUT word N°	The ranges limits		The mean sig 2 (x_i)	Stored value	Stored value in binary	Maximum error
	x_{i1}	x_{i2}				
	...	0.03125	x	NA	NA	NA
1	0.03125	0.09375	0.031179	0.03125	00000010	0.0156301
2	0.09375	0.15625	0.0621168	0.0625	00000100	0.0157618
3	0.15625	0.21875	0.0925793	0.09375	00000110	0.0162546
4	0.21875	0.28125	0.122347	0.125	00001000	0.0173368
5	0.28125	0.34375	0.151221	0.15625	00001010	0.0192192
6	0.34375	0.40625	0.179026	0.171875	00001011	0.020767
7	0.40625	0.46875	0.205618	0.203125	00001101	0.0154694
8	0.46875	0.546875	0.233841	0.234375	00001111	0.0157806
9	0.546875	0.625	0.263194	0.265625	00010001	0.0165378
10	0.625	0.71875	0.292684	0.296875	00010011	0.0195751
11	0.71875	0.8125	0.321775	0.328125	00010101	0.0200578
12	0.8125	0.921875	0.349438	0.34375	00010110	0.0196416
13	0.921875	1.0625	0.37835	0.375	00011000	0.0183094
14	1.0625	1.25	0.408726	0.40625	00011010	0.0178918
15	1.25	1.53125	0.43973	0.4375	00011100	0.0178191
16	1.53125	0.467493	0.46875	00011110	0.0134309

Simulations:

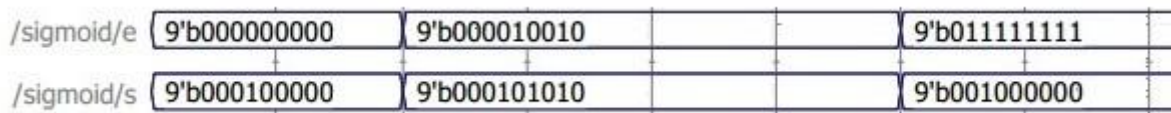


Figure 3-10 Log sigmoid2 simulation

The simulation shows the good functioning of the sigmoid2 function block. It gave the value 0.5 when the input was null and 1 when the input was in the saturation region.

The sig2 function $\text{sig2}(x) = f(x) + 1/2$ is implemented as it is shown in the following block diagram:

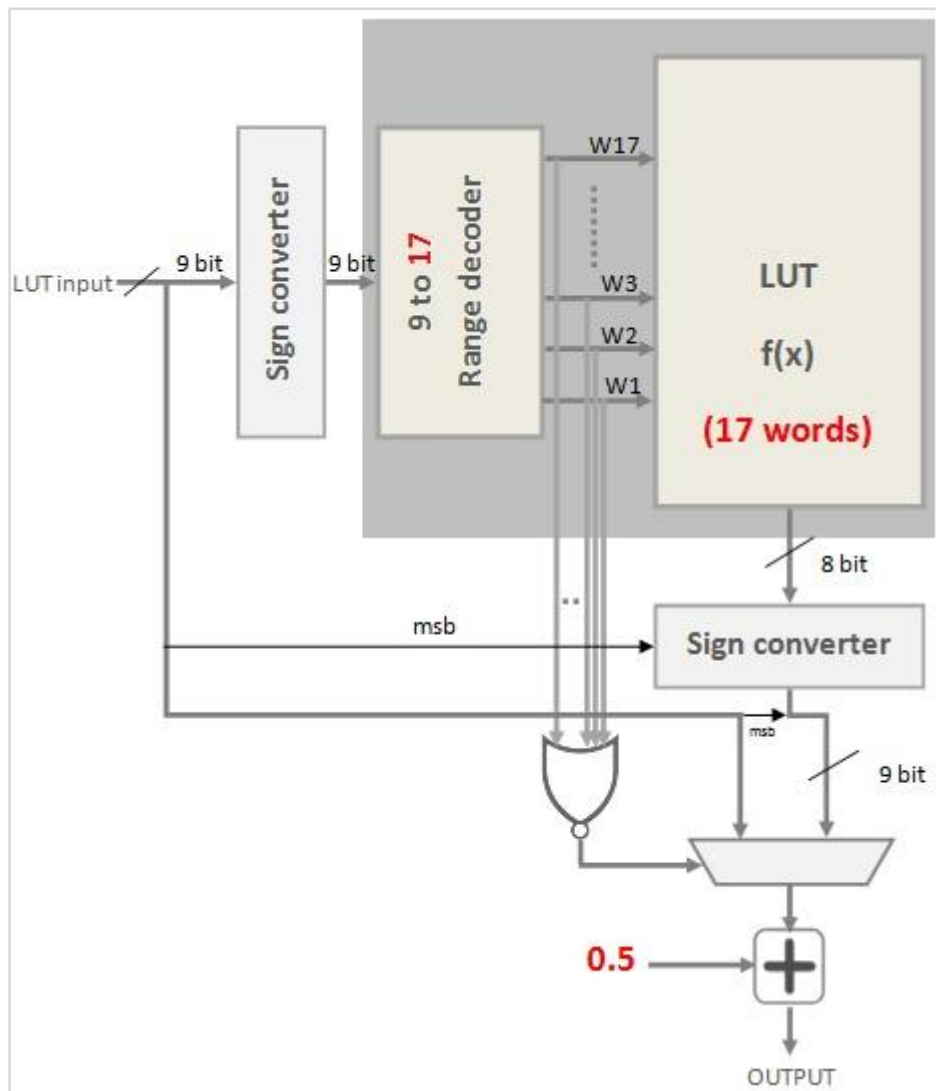


Figure 3-11: Sigmoid2 Implementation diagram

3.4 The architecture overview of the ANN generated by the C++ application

The choice of the anatomy of an ANN (the number of inputs, outputs, layers, and the number of neurons per layer) is specific to each application. In literature it has been reported that three-layer network with sigmoid activation function in the hidden layer and linear activation function in the output layer can virtually approximate any nonlinear function to any degree of accuracy provided sufficient number of neurons in a hidden unit is available. However To realize all types of nonlinearity using three layers, large number of neurons is needed and it may result in a huge NN.

For function approximation, multilayer networks have been found to be very useful as it is similar to a biological NN. However Implementations of multilayer networks will demand huge resource and will not be a feasible solution for real-time applications such as estimators for motor control. Thus we proposed an architecture that is based on the concept of layer

multiplexing presented in [24], where large ANNs could be implemented with minimum resources.

3.4.1 Concept of layer multiplexing

The data processed in a multilayer feed forward ANN propagates from one layer to another, and the computing happens in one layer at a time, hence we don't need to have all layers implemented in the same time, only the largest layer (the one with the maximum number of neurons) should be implemented, it calls itself repeatedly and behaves as different layers with the help of a control unit. The control block ensures the complete computation of NN using layer multiplexing by sequencing and placing the appropriate inputs, weights, biases, and value of excitation function (from LUTs) of each layer .

Unlike the architecture presented in [24], in our architecture, we have implemented both the largest layer and the output layer, which means that the largest layer behaves like the hidden layers only and not the output layer. That is to be able to have a different activation functions in the output layer. So implementing an ANN like the one in figure is reduced to implementation of the layer-multiplexed ANN in figure

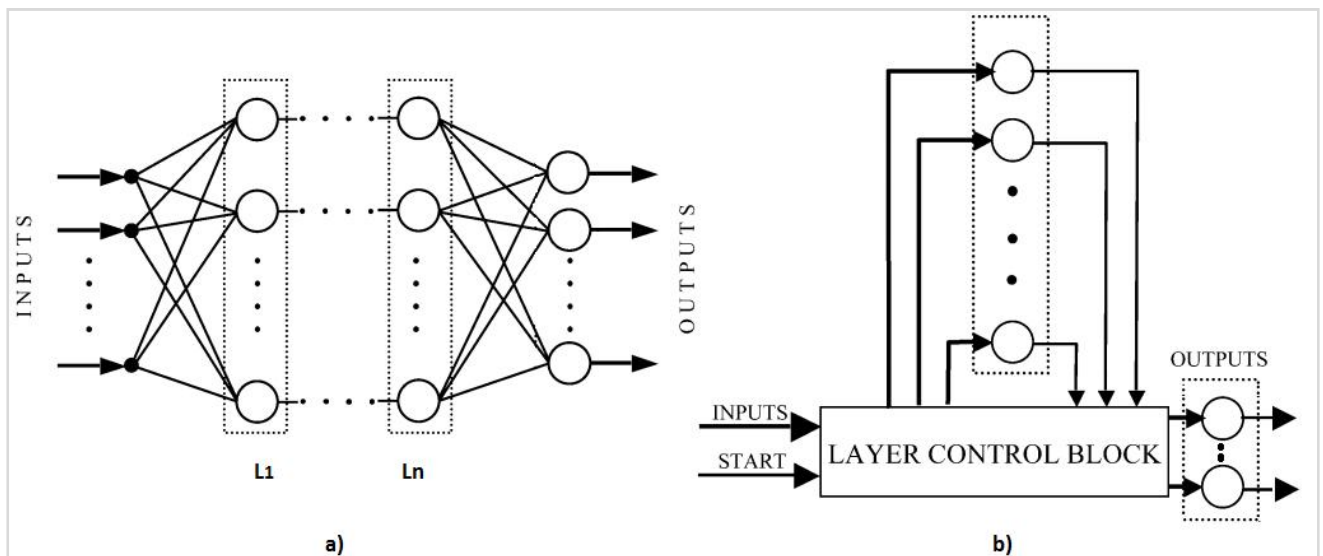


Figure 3-12 : Layer multiplexed ANN [24]

3.4.2 Single neuron architecture

As indicated in its mathematical model (Figure 1-4), the neuron does two major arithmetic operations; the first is a sum of products of inputs with their correspondent synaptic weights (1-1), then the result is fed to an activation function (1-2) to determine its output.

The sum-of-products implementation is done by the circuits in Figure 3-3, and the activation function is implemented by circuits shown in Figure 3-6, the global architecture of a neuron is shown in Figure 3-13.

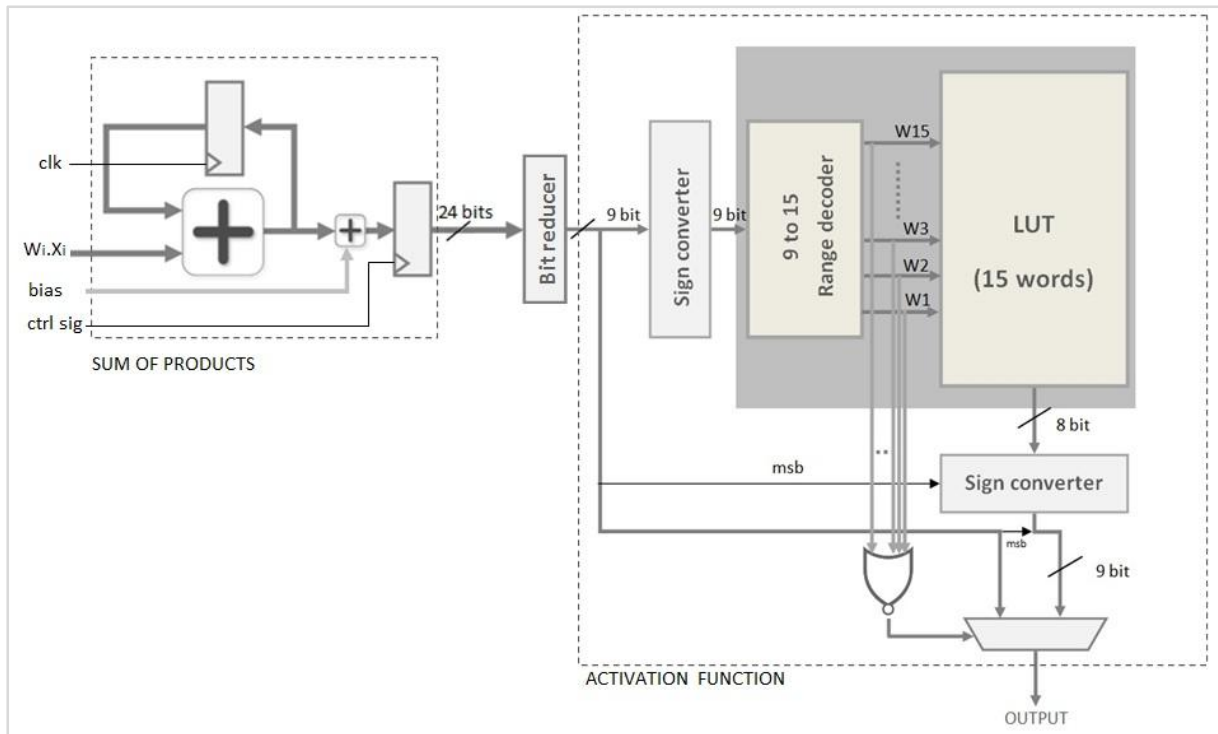


Figure 3-13: A Single neuron block diagram

The width used for the weights in this work is 15 bits; 7bits for the integer part and 8bits for the fractional part. Thus the product of the weight (15bits) with the input (9bits) gives a result in 24bits; this result is then fed to the adder which gives a result in 24bits too. The problem is how to adapt the width of the data coming from the adder, to the width of the input of activation function's block without affecting the results. We introduced a block called bit-reducer (see Figure 3-13) to do this adaptation.

Because of the saturation region in the hyperbolic tangent function, the output of the LUT for any value of its inputs x that verifies $|x| > 2.4$ is always 1. Thus if the adder's output exceeds 2.4 the bit-reducer changes it to a value that can be represented in 9bits, and which gives a result of 1 at the output of the LUT.

Since the LUT's input has 3bits for the integer part, then if $sum > 2.4$ the LUT's input is set to 3, and If the $sum < -2.4$ than the LUT's input is set to -4. This is because 3 and -4 give an output value equal to 1 and they can be represented in 9bits (in 3bits of the integer part).

3.5 C++ application

As it was mentioned before, our work consists of a C++ application that generates the VHD description of an optimized architecture of feed-forward layered neural networks. It is mainly an interface that demands from the user to insert the ANN anatomy, like the number of inputs, outputs, hidden layers and neurons of each layer, and then it generates its corresponding VHD description for that ANN.

This application is a kind of an abstraction to ANN VHD description, since the user has no longer the need to write any VHD code for any feed forward layered neural network, all what is needed for that with the application is a few clicks.

This application generates the implementation of the generalization phase only. That is to say that the neuron's training is performed offline with, so this application requires from the user to provide the weights and biases written in real representation (float), in a txt files and then it generates as output a set of files with a "vdh" extension which represent the different components of the ANN architecture.

The coding idea used to generate the VHD files is very simple. We used the commands of read and write from files, then the VHD description is written depending on the anatomy of the ANN specified by the user

Example:

```
fichier<<endl<<"library ieee ;";
```

This image shows an instruction of the C++ code to generate the first line of the VHD description. However the line showed is constant, it does not need to be changed with different ANN anatomies that are specified by the user, the challenge in this application was how to generate VDH codes of the blocks that are variable with different ANNs

Example:

```
fichier<<endl<<"      cs : out std_logic_vector (<<maxholayer<<" downto 1);";
```

In this example we can see the C++ instruction that generates a signal "cs", however this time this signal is used to command all the neurons in the output layer, each bit commands a neuron. Thus if a user wants an ANN with a 19 outputs this signal should have a width of 19, i.e., it becomes (19 downto 1). Another deal is how to initialize or assign values to this kind of signals. That was done by a set of simple functions like:

```
23  std::string toBinary( int n, int nmax);
24  std::string generatornchar(char c, int nbit);
25  std::string vectaffect(int c, int up,int down,int nbit);
26  std::string weightinitiate(float n);
27  std::string biasinitiate(float a);
28  std::string xtinitiate(float a);
```

The function in line 23 indicated in the previous image called “toBinary” takes a number n , and its maximum value and gives as an output in a string the representation of that number in 2’s complement representation with an appropriate width.

The functions in lines 26 and 27 do transform the values of biases and weights provided in files by the users, into their appropriate binary representation. The width of both biases and weights a general for all ANNs generated by this application.

The previous example shows how to generate and initiate signals whose width depends on the anatomy of the ANN specifies by the user; however there are much more complicated blocks whose functioning depends completely on the anatomy, and the most important one is the Control Unit. This block contains a finite state machine that produces signal to synchronize all the system (ANN), and the C++ code that generates it needs additional C++ functions and many conditions.

The figure shows the state diagram (a direct graph) of state machines for two different neural networks generated by the application:

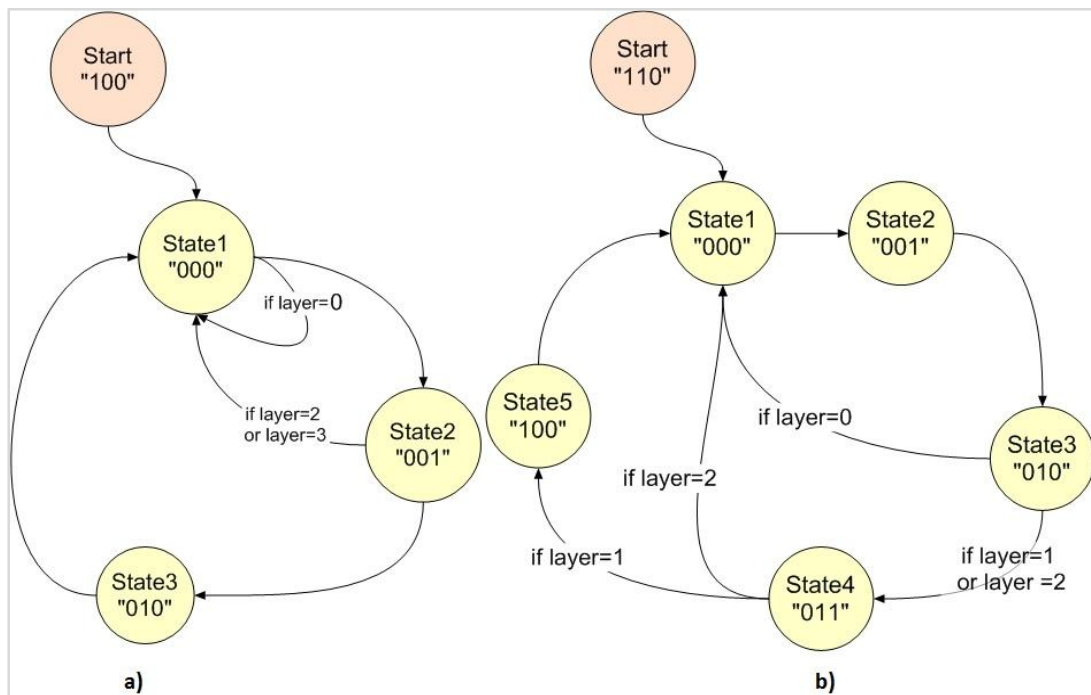


Figure 3-15: Flow state of 2 state machines

The first state machine (Figure 3-15 a) is for an ANN that has 3 layers plus an input layer that contains 3 inputs, the layer 1 has 1 neuron and both layers 2 and 3 have 2 neurons.

The second state machine (Figure 3-15 b) is for an ANN that has 2 layers plus an input layer which contains 3 inputs too, the layer 1 has 5 neurons and layer has 4 neurons.

It can be clearly seen that the state machines of the two neurons are totally different.

Simulations:

To test our application we have generated two different neural networks:

The first one has the anatomy 1_1_15 which means that it has 1 input, 1 neuron in the hidden layer, and 15 neurons in its output. The second one has the anatomy 1_1_5; (see Figure 3-16 and Figure 3-17)

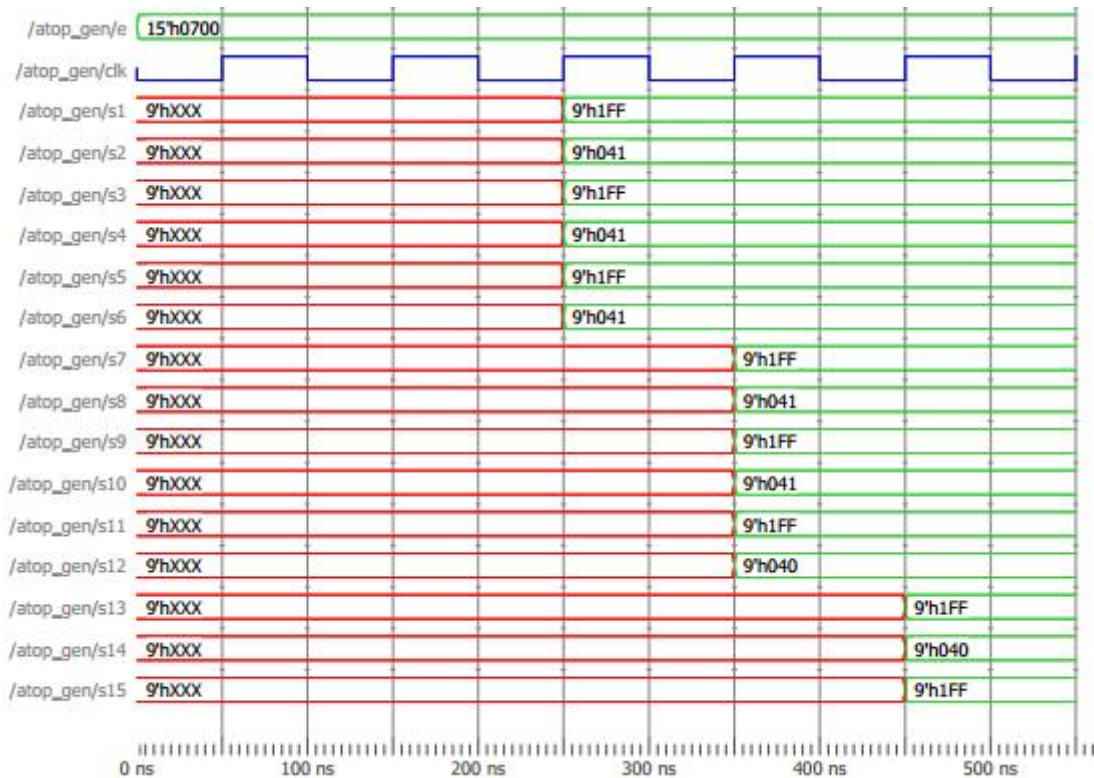


Figure 3-16 : ANN 1_1_15

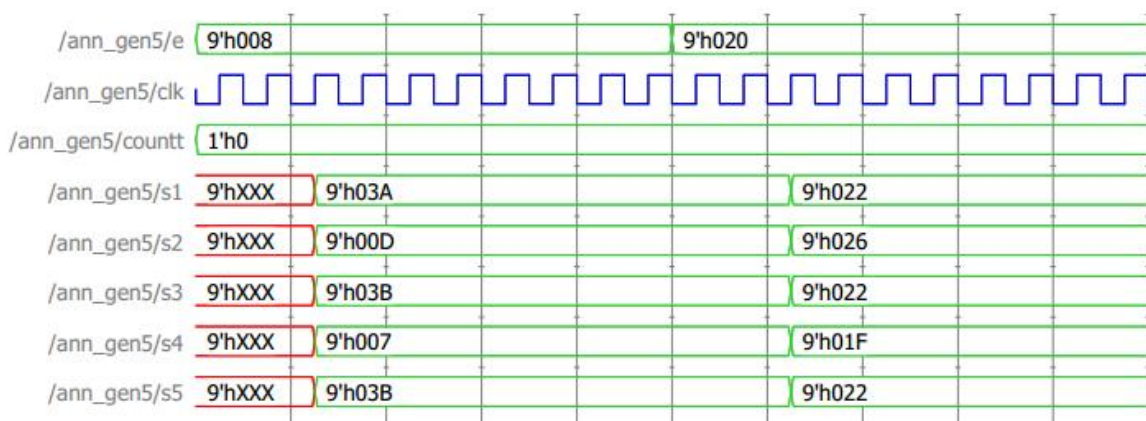


Figure 3-17 : ANN 1_1_5

Interpretation:

The Figure 3-16 shows a ModelSim simulation of the 1_1_15 ANN generated by the C⁺⁺, we can see clearly that all the output were assigned in less than four clock cycles. Normally all

the outputs should have being assigned their values in the third clk cycle, however the number of neurons of the output layer in this ANN is 15, it largely exceeds the number of products used (since the latter is very limited, in our FPGA and we used on multiplier per neuron, thus we assigned the outputs 6 by 6 and this created more latency (the two additional clk cycles). Unlike it, the second ANN's largest layer has 5 neurons in the output, that is why all its outputs got assigned in the third clk cycle)

3.6 Conclusion

In this chapter we have presented the hardware implementation of ANNs in FPGAs. We first introduced the most important issues and the challenges that should be dealt with by the designer when implementing ANNs; these include the parallelism required for real-time applications, the precision and the minimization of the cost (resources). After that we have presented an optimized method for implementing the activation function of a neuron by the use of an optimized LUT proposed by Maher. That same method was used in this work to implement two non linear activation functions, the hyperbolic tangent and the sigmoid function.

To implement large ANNs, huge resources are required. To avoid this, we have used the concept of layer multiplexing, where we needed to implement only the largest layer, this one calls itself repeatedly to behave sequentially like all hidden layers. This method proved to be very effective in term of reducing resources without a big compromise on the computing speed.

The architecture of a single neuron, and the whole artificial network were then presented respectively in the form of block diagrams and their components when presented after, and to conclude the chapter we talked about the C⁺⁺ program we created and that generates any feed forward ANN that the user wants, basing on all those previous optimized designing methods. After that we simulated two different generated ANNs, and analyzed the results.

CHAPTER 4

CHAPTER 4. ANN SHE PWM TECHNIQUE

4.1 Introduction

At the end of the last century, one of the results of the development of power electronics is the Pulse Width Modulation technique. It is the heart of the control of static converters. The objective of the PWM technique in controlling a voltage inverter is to have a fast response and high performance. The programmed PWM which is one of the two types of the PWM (generated and programmed) is based on a technique of elimination of unwanted harmonics that may produce vibrations and undulations of torque and many undesirable consequences. This technique is called SHE PWM (Selective Harmonics Elimination Pulse-Width Modulation). It was introduced by Turnbull in 1964 and developed later by Patel and Hofel in 1973.

4.2 SHE PWM

4.2.1 Introduction

This technique consists in forming the output wave of a succession of slots of variable and controllable widths. The switching angles are determined so as to eliminate certain disturbing harmonics in the output wave and improve the efficiency of the inverter-machine system by reducing torque ripples, as well as current peaks and losses in the machine. The calculation of these angles with this method is based on the nonlinear and transcendental equations. This has forced researchers to use numerical methods such as Newton-Raphson. The problem with this method is the choice of good initial values necessary for convergence. Moreover, the computation of these angles cannot be done on-line (in real-time), thus the angles should be stored in memory, which makes the system not optimal for applications whose changes in frequency and voltage are fast, such as the speed controller. This problem has led to the need to use a better algorithm. Recently, an algorithm based on the polynomial interpolation approach of the trajectory of the SHE PWM angles by ANN was proposed by GUELLAL Ammar [13]. This algorithm will be our application and be implemented using our ANNs to calculate the switching angles and generating the SHE PWM signals for controlling the voltage inverter on-line, .

4.2.2 Principle of operation

The PWM signals describing the three output voltages of the converter must have properties which help to orient their characteristics towards those of a sine wave. In order to approach them as much as possible, we may in some cases attribute to them the same properties of symmetry as a sinusoidal wave. The aim of this technique is to eliminate a certain number of low-order harmonics and to control the fundamental wave. The output voltage of the inverter is defined as a function of the exact switching angles $\alpha_1, \dots, \alpha_m$ (see Figure 4-1) corresponding to the switching times of the voltage from a positive value +E to a negative value -E or vice versa. The index m is the number of switching angles of the output voltage of the inverter per quarter wave. The output voltage of the inverter is constructed to have half-wave symmetry (odd function with respect to the angle π). This symmetry makes it possible to eliminate certain types of harmonics, which simplifies the Fourier series development of this voltage and reduces the harmonic ratio. Then, the amplitude of the fundamental is fixed to the value i_m and the amplitudes of the (m-1) first harmonics are canceled.

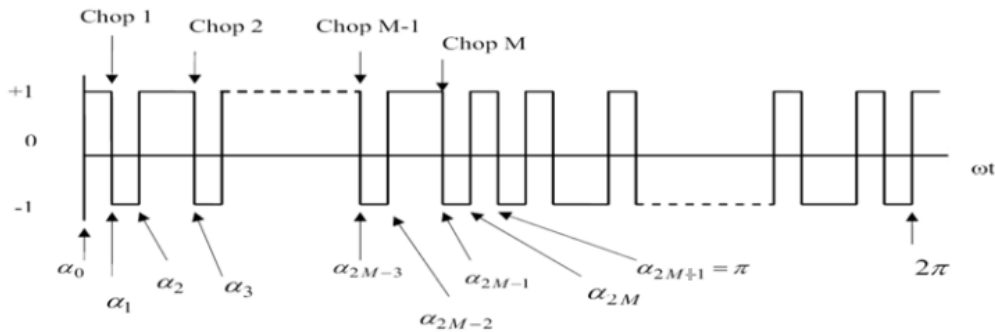


Figure 4-1 : The Inverters output normalized Voltage

i_m is the modulation rate defined by:

$$i_m = \frac{V}{E}$$

V is the tension of the fundamental.

It is assumed that the output voltage is periodic and of unit amplitude. Let f be the function representing the PWM signal as a function of α ($\alpha = \omega t$). We can write therefore:

$$f(\alpha) = -f(\alpha + \pi) \quad (4-1)$$

The function f can be decomposed into Fourier series:

$$f(\alpha) = a_0 + \sum_{n=1}^{\infty} (a_n \sin(n\alpha) + b_n \cos(n\alpha)) \quad (4-2)$$

Where:

$$a_0 = \frac{1}{2\pi} \int_0^{2\pi} f(\alpha) d\alpha \quad (4-3)$$

$$a_n = \frac{1}{\pi} \int_0^{2\pi} f(\alpha) \sin(n\alpha) d\alpha \quad (4-4)$$

$$b_n = \frac{1}{\pi} \int_0^{2\pi} f(\alpha) \cos(n\alpha) d\alpha \quad (4-5)$$

The calculation shows:

$$a_0 = 0$$

For n even:

$$a_n = b_n = 0$$

For n odd:

$$a_n = \frac{4}{n\pi} \left[1 + 2 \sum_{k=1}^M (-1)^k \cos(n\alpha_k) \right] \quad (4-6)$$

$$b_n = 0$$

In our study we used a three-phase inverter, so the harmonics of rank three and multiples of three are eliminated automatically. Thus, n takes odd values different from a multiple of 3.

4.2.3 The switching angles

Each equation (4-6) has m unknown variables $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m$. The problem is to calculate the values of these switching angles which make it possible to cancel the amplitudes a_n of the first (m-1) harmonics f_n and to assign an im value to the amplitude a_1 of the fundamental f_1 .

On the other hand, two voltage harmonics must be eliminated in order to eliminate a current harmonic. Since the amplitude of the fundamental is to be fixed at a given value, this sets the first value from m to 3 (m being the number of quarter-wavelength switching or number of cuttings per half wave).

The first value of m is set to 3 so that the amplitude of the fundamental is fixed to a given value (m being the number of quarter-wave switching per half wave). Consequently, when m is increased successively by 2, the number of current harmonics that will be eliminated is increased by 1.

It should be noted that the value of the modulation index im assigned to the fundamental is a dimensionless index varying from 0 to 1. To obtain the corresponding value in volt, multiply by E for the three-phase inverter.

The equations (4-7) form a system of m nonlinear equations with m unknown

$$\left\{ \begin{array}{l} f_1(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m) = \frac{4}{\pi} \left[1 + 2 \sum_{k=1}^m (-1)^k \cos(\alpha_k) \right] + im = 0 \\ f_2(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m) = \frac{4}{5\pi} \left[1 + 2 \sum_{k=1}^m (-1)^k \cos(5\alpha_k) \right] = 0 \\ f_3(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m) = \frac{4}{7\pi} \left[1 + 2 \sum_{k=1}^m (-1)^k \cos(7\alpha_k) \right] = 0 \\ \vdots \\ f_m(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m) = \frac{4}{n\pi} \left[1 + 2 \sum_{k=1}^m (-1)^k \cos(n\alpha_k) \right] = 0 \end{array} \right. \quad (4-7)$$

For this method to converge, we assign a negative value (-im) to the fundamental. This corresponds to a phase shift of π of the fundamental. This phase shift has no effect on the motor [13].

A simple technique to solve these equations is to use Newton-Raphson method. We must have a good initial estimate of the exact solution sought, so that this method accurate solution and good convergence. Alternatively, more complicated gradient search methods can be used to obtain the solutions. Indeed, the Taufik, Mellitt, and Goodman algorithm is used to quickly estimate the initial values of the nonlinear system solution.

A MATLAB program has been made to calculate the switching angles as a function of the modulation index im [13].

4.3 Implementation of ANN SHE PWM

As we have already stated, the PWM is a powerful tool in controlling the voltage inverter. It based on the SHE PWM technique which requires a very high computing time to calculate the switching angles using numerical methods which prevents speed control on-line so the algorithm of Patel and Hoft is off-line. Over the last years, various researches have addressed this topic. The results showed that it was not possible to implement this algorithm on a microprocessor. However, in 2007, GUELLAL managed to implement this algorithm on an FPGA circuit. With the emergence and development of new intelligent control techniques, a new algorithm based on artificial neural network (ANN) has been proposed. The aim of this algorithm is to calculate the switching times of the PWM signal with a precision very close to those calculated by the Patel and Hoft algorithm. This algorithm is going to be our application to test the performance of our ANN discussed in chapter III.

4.3.1 Architecture of the ANN SHE

The aim of the proposed algorithm is to build a Multi Layer Perceptron MLP which will be implemented on an FPGA circuit using the VHDL codes generated by our application.

4.3.1.1 The topology of neural network [13]

In order to simplify the implementation and reduce the consumed space in the FPGA circuit, a network composed of an input layer, a hidden layer and an output layer has been chosen, in addition the hidden layer contains a single neuron. This architecture also makes it possible to reduce the error and the time when calculating the switching angles since, if the number of layers or the number of neurons in the hidden layers is increased, the error in the switching angles and the calculation time will be multiplied.

Concerning the activation functions, the tangent sigmoid function was chosen between the input layer and the hidden layer to present the nonlinearity of the system and a simple linear function between the hidden layer and the output layer. The non-linear function between the input layer and the hidden layer has been placed so that this function is computed once, since if placed between the hidden layer and the output layer this function will be calculated for each switching angle; thus this choice makes it possible to reduce the complexity and the time of the computation.

The architecture of our ANNSHE PWM algorithm is presented in Figure 4-2 where:

im: The modulation index which is the input of the network.

W1: Weights matrix between the input layer and the hidden layer.

W2: Weights matrix between the hidden layer and the output layer.

b1: Biases matrix between the input layer and the hidden layer.

b2: Biases matrix between the hidden layer and the output layer.

α : Matrix of switching angles which presents the output of our network.

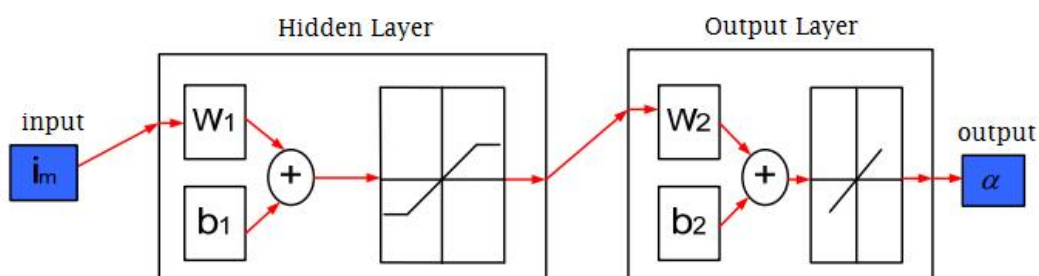


Figure 4-2 : ANN SHE Architecture

4.3.1.2 Database

In this section, the database that will be used in the off-line training is managed. This database gives a relation between the input of the artificial neural network which is the index modulation im and its output which is the switching angles matrix α . im takes values from 0 to 1. A database of 100 values in which im vary by a step of 0.01 has been constructed [13], and for each value of im , for each value of im , the system of equation of Patel and Hoft (equation (4-7) is solved in order to find the matrix α of the corresponding switching angles. Moreover, in order for the algorithm to be effective, that is to say the angles calculated by this algorithm are very close to the exact values computed by the iterative method of Newton-Raphson, and to allow the convergence of the learning step, the interval of variation of im is divided into six and for each interval a specific ANN (weights and biases) is constructed. The choice of number of switching angles (i.e. the number of harmonics to be eliminated) in each interval depends on the value of the index im and since the effect of the harmonics increases when im decreases we take the appropriate choice which is illustrated in the Table 4-1, this choice also makes it possible to optimize our algorithm and thus minimize the space consumed during the implementation.

Table 4-1: ANN SHE characteristic

The modulation index im	ANN	Number of the switching angles	Number of neurons in the hidden layers	Number of neurons in the output layer	Training algorithm	Activation function	Number of elements in database of the training	Training parameters	
								Performance	of number epochs
$0.01 \leq im < 0.16$	ANN-1	23	1	23	Back propagation algorithm	Hyperbolic Tangent Sigmoid + linear	23x 15	10^5	62561
$0.16 \leq im < 0.32$	ANN-2	19	1	19			19x 16	10^5	64982
$0.32 \leq im < 0.56$	ANN-3	15	1	15			15x 24	10^5	73214
$0.56 \leq im < 0.76$	ANN-4	7	1	7			7x 20	10^5	87327
$0.76 \leq im < 0.92$	ANN-5	5	1	5			5x 16	10^5	164338
$0.92 \leq im < 1$	ANN-6	3	1	3			3x 8	$1.5 \cdot 10^4$	200000

To find the parameters of six ANNs, a program on MATLAB which solves the system of equation of Patel and Hoft and generates the six databases has been developed [13]. These databases will be used in the learning phase. Some of switching angles are shown in the Table 4-2 using ANN-4.

Table 4-2 : An example of switching angles ($m=7$) generated by a Matlab program

im	0.58	0.59	0.60	0.61	0.62
α_1	11,1149755	11,0448428	10,9745607	10,904126	10,83353518
α_2	16,4743076	16,4952108	16,5158101	16,5360929	16,55604594
α_3	25,8659416	25,7882198	25,7101518	25,6317263	25,55293186
α_4	32,5003488	32,5384342	32,5761823	32,6135775	32,65060327
α_5	40,836138	40,7576594	40,6787918	40,5995203	40,51982873
α_6	48,3117705	48,3673372	48,4227855	48,4781084	48,53329855
α_7	56,1144861	56,0440286	55,9733672	55,9024937	55,83139934

4.3.1.3 Off-line training

In this stage, the parameters of six ANNs (ANN-i) which are the weights and biases will be calculated off-line. The database of each ANN calculated in the previous section, is used as input in a training program based on the gradient method. The inputs and the outputs of that program are automatically normalized. The training program has two shutdown conditions, performance and number of epochs. When one of these conditions is verified, training is stopped to generate the appropriate parameters (weights and biases).

The parameters of the training and the characteristics of each ANN-I are shown in the Table 4-1.

At the end of the training, the appropriate parameters for the used neurons is generated and used for the simulation in the next section. These results (Weights and biases) are shown in the Table 4-3 for the ANN-4.

Table 4-3 : weights and biases calculated in the training phase

W1	W2	b1	b2
0.9966	-1.0588	-0.5257	0.4770
	1.0551		0.5583
	-1.0581		0.4813
	1.0615		0.5459
	-1.0576		0.4826
	1.0602		0.5313
	-1.0582		0.4790

4.3.1.4 Simulation

In this stage, the efficiency and the accuracy of the algorithm is checked. Using ModelSim, the ANN-4 is simulated (Figure 4-3) for two different values of im (im=59.5 and im=61.5).

As we have mentioned earlier, the input is represented in 15 bits (10 for the integer part and 5 for the fractional part) and the output is represented in 9 bits (3 for the integer part and 6 for the fractional part).

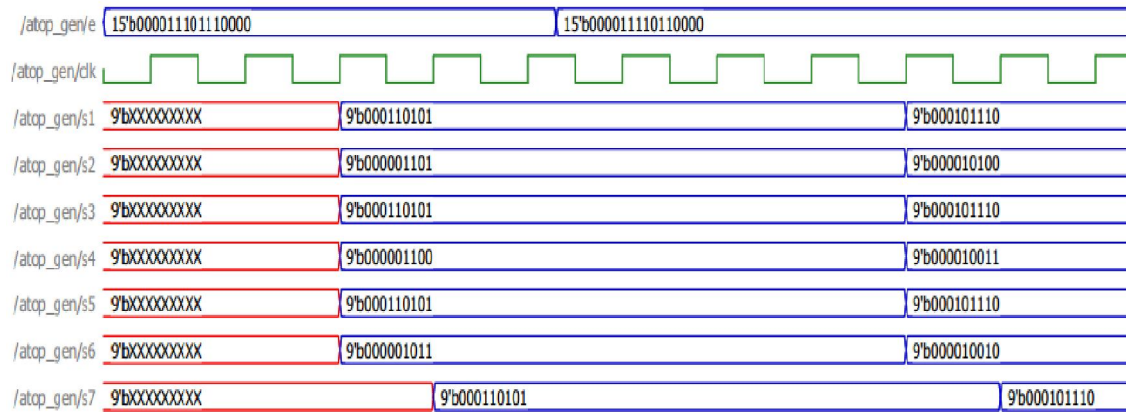


Figure 4-3: A ModelSim simulation to generate the angles ($im = 0.595$ & 0.615)

It is clear from the Figure 4-3 that the output is normalized, thus a reverse normalization should be done in order to get the real switching angles. The Table 4-4 shows the results after the reverse normalization.

Table 4-4 : The exact values for switching angles for $m=0.595$ and $m=0.615$

im	0.595	0.615
α_1	11,0218985	10,8736844
α_2	16,5020112	16,5399051
α_3	25,7570957	25,5895462
α_4	32,547983	32,6207589
α_5	40,724594	40,5544361
α_6	48,3779872	48,4910562
α_7	56,0188291	55,8686532

4.3.1.5 Results & interpretation

To check the performance and the accuracy of the ANN SHE, we make a comparative study between the exact switching angles calculated by Newton-Raphson method, and the ANN SHE switching angles given by the MATLAB program. Then the error between them has been shown in the Table 4-5.

Table 4-5 : A comparison between the exact and ANN SHE switching angles

im = 0.595			im = 0.615		
Exact angles	ANN SHE angles	error	Exact angles	ANN SHE angles	Error
11.0097	11,0218985	0,01219848	10,8689	10,8736844	0,00478443
16.5055	16,5020112	0,00348878	16,5461	16,5399051	0,00619486
25.7492	25,7570957	0,00789571	25,5924	25,5895462	0,00285378
32.5574	32,547983	0,009417	32,6321	32,6207589	0,01134113
40.7183	40,724594	0,00629396	40,5597	40,5544361	0,00526387
48.3951	48,3779872	0,01711281	48,5057	48,4910562	0,01464378
56.0087	56,0188291	0,01012914	55,867	55,8686532	0,00165325

We can clearly see that the ANN SHE angles are close to exact angles with a very small error.

4.3.2 Implementation of ANN SHE

To implement practically the proposed ANNSHE PWM algorithm and thus verify its accuracy and efficiency, an implementation on an FPGA circuit is described using our application described in the previous chapter.

4.3.2.1 Description

Our C++ application is used to implement ANN SHE. The ANN SHE is a specific application with its own characteristic, thus some extra blocks should be added to the main structure generated by our application:

Interval selector

As we have already stated earlier, the im interval has been divided into 6 sub-intervals. For each sub-interval, an artificial neural network (ANN- i) has been constructed. Accordingly, the purpose of this block is the selection of the network which is suitable for the input im (Figure 4-4). In the training phase, im was expressed as a percentage, i.e. im is between 0 and 100%. In the design we have dimensioned the im input on fifteen bits, ten bits for the integer part and five bits for the fractional part, so the variation pitch of im is 0.03125%. This choice can be changed according to the choice of the variation pitch of im . Moreover, in order to simplify the choice of each interval as a function of im , the limits of each interval were defined as indicated in the Table 4-6 [13].

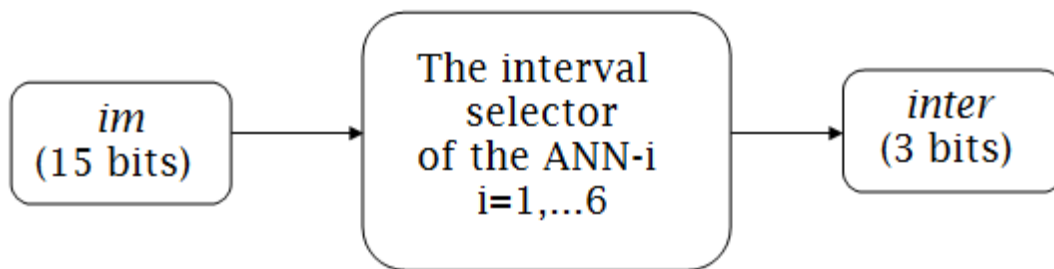


Figure 4-4 : Interval selector structure

Table 4-6: The intervals for the variation of im

ANN	Interval of variation of im in percentage $im_{\min} \leq im < im_{\max}$	The lower limit in binary	Interval
ANN-1	$01\% \leq im < 16\%$	000000000000	001
ANN-2	$16\% \leq im < 32\%$	001000000000	010
ANN-3	$32\% \leq im < 56\%$	010000000000	011
ANN-4	$56\% \leq im < 76\%$	011100000000	100
ANN-5	$76\% \leq im < 92\%$	100110000000	101
ANN-6	$92\% \leq im < 100\%$	101110000000	110

According to the Table 4-6 we find:

$$\begin{aligned}
ANN_1 &= \overline{im_{12}} \cdot \overline{im_{11}} \cdot \overline{im_{10}} \\
ANN_2 &= \overline{im_{12}} \cdot \overline{im_{11}} \cdot im_{10} \\
ANN_3 &= \overline{im_{12}} \cdot \overline{im_{11}} \cdot (\overline{im_{10}} + \overline{im_9}) \\
ANN_4 &= (\overline{im_{12}} \cdot im_{11} \cdot im_{10} \cdot im_9) + (im_{12} \cdot \overline{im_{11}} \cdot \overline{im_{10}} \cdot (\overline{im_9} + \overline{im_8})) \\
ANN_5 &= im_{12} \cdot \overline{im_{11}} \cdot ((\overline{im_{10}} \cdot im_9 \cdot im_8) + (im_{10} \cdot (\overline{im_9} \cdot \overline{im_8}))) \\
ANN_6 &= im_{12} \cdot ((\overline{im_{11}} \cdot im_{10} \cdot im_9 \cdot im_8) + (im_{11} \cdot \overline{im_{10}} \cdot \overline{im_9}) \cdot (\overline{im_8} \\
&\quad + (im_8 \cdot \overline{im_7} \cdot \overline{im_6})))
\end{aligned} \tag{4-8}$$

$$\begin{aligned}
Inter_1 &= ANN_1 + ANN_3 + ANN_5 \\
Inter_2 &= ANN_2 + ANN_3 + ANN_6 \\
Inter_3 &= ANN_4 + ANN_5 + ANN_6
\end{aligned} \tag{4-9}$$

According to the equations (4-8) and equations (4-9) a design based on the combinational logic of the "interval selector" module is created under VHDL to select the appropriate network at the im input. This block is added to the other blocks generated by our C++ application.

Normalization & Reverse normalization

In the training phase, one important parameter is the input interval values. Once this interval is chosen, the training is done its calculation considering the input as it is set in that interval. Thus a normalization block has been added to adjust any inputs of ANN to another normalized input.

In the calculation of the switching angles of the SHE PWM, the reverse normalization block at the output of the ANN is not needed since we have used the normalized output instead of the real one.

PWM signal generator

We have at the end of our main design followed by the extra blocks, the switching angles. This block is used to generate the PWM signals by converting these angles to times, thus we obtain the switching times.

The switching angles are calculated using the following equations:

$$\alpha_i = \frac{\alpha_{i \max} - \alpha_{i \min}}{\alpha n_{i \max} - \alpha n_{i \min}} (\alpha n_i - \alpha n_{i \min}) + \alpha_{i \min} \tag{4-10}$$

Where:

$\alpha_{i \max}$ and $\alpha_{i \min}$ are respectively the maximum angle and the minimum angle of the database corresponds to the angle α_i

$\alpha n_{i \max}$ and $\alpha n_{i \min}$ are respectively the maximum normalized angle and the minimum normalized angle of the database corresponds to the normalized angle αn_i .

In the training phase $\alpha_{i \max} = 1$ and $\alpha_{i \min} = 0$ has been used, Replacing in the equation (4-10) we find:

$$\alpha_i = (\alpha_{i \max} - \alpha_{i \min})\alpha n_i + \alpha_{i \min} \quad (4-11)$$

Posing

$$k_{1i} = (\alpha_{i \max} - \alpha_{i \min}) \text{ and } k_{2i} = \alpha_{i \min}$$

By replacing in (4-11):

$$\alpha_i = k_{1i}\alpha n_i + k_{2i} \quad (4-12)$$

As it has been mentioned above, in order to generate the PWM control signals it is necessary to transform the switching angles into switching times. In our application we have opted for the command $\frac{v}{f} = cte$. By using this property one finds:

$$\frac{v}{f} = \frac{v_0}{f_0} \quad (4-13)$$

which implies:

$$\frac{v}{v_0} = \frac{f}{f_0} = im \quad (4-14)$$

Where im is the modulation index and $f_0 = 50 \text{ Hz}$ the frequency for $im = 1$.

The relationship between the switching angle and the switching instants is given by the following relation:

$$t_i = \frac{\alpha_i}{360} \times \frac{1}{f} \quad (4-15)$$

Using the equations (4-14) and (4-15) one finds:

$$t_i = \frac{10^{-3}}{18} \times \frac{\alpha_i}{im} \quad (4-16)$$

Since α_i was expressed in percentage.

$$t_i = \frac{1}{180} \times \frac{\alpha_i}{im} \quad (4-17)$$

In addition, to generate the control signals from the switching times, a clock of 1 MHz was used, so the switching times must be expressed in μs .

$$t_i(\mu\text{s}) = \alpha_i \times \frac{10^6}{180} \times \frac{1}{im} \quad (4-18)$$

By combining (4-12) and (4-18) one finds

$$t_i(\mu\text{s}) = \frac{1}{im} \left(\frac{k_{1i} \times 10^6}{180} \alpha n_i + \frac{k_{2i} \times 10^6}{180} \right) \quad (4-19)$$

$$t_i(\mu s) = \frac{1}{im} \times \theta_i \quad (4-20)$$

with

$$\theta_i = \left(\frac{k_{1i} \times 10^6}{180} \alpha n_i + \frac{k_{2i} \times 10^6}{180} \right) \quad (4-21)$$

According to (4-20) we have a division on im which poses an implementation problem on an FPGA circuit. To avoid this division, in this step the values of θ_i given by the equation (4-21) are calculated. Then, in the step of generating control signals, an internal signal “counter” in the form of a counter is created.

The equation (4-20) gives:

$$im \times t_i(\mu s) = \theta_i \quad (4-22)$$

The “counter” represents the value of $im \times t_i(\mu s)$. It is initialized by 0 and incremented by im at each rising edge of the clock (1 μs) then we compare it, each time, with θ_i .

According to (4-21) we need a multiplier to calculate each value θ_i , where i can vary from 1 to 23, so if we calculate all θ_i in parallel we need 23 multipliers. Only two multipliers were used [13]. According to the signal generation block, in the first clock edge it is necessary to calculate θ_1 and θ_m (m is the number of switching angles), in the second front θ_2 and θ_{m-1} , and so on until calculation of all θ_i .

The objective of this block is to generate the PWM signals s_1, s_2, s_3 from the switching instants. At the beginning, the signal s_1 is at 1 then as we have mentioned the “counter” starts at 0 and incremented by im then it is compared to θ_1 . When “counter” become greater than θ_1 , s_1 is toggled and “counter” is compared then to θ_2 until it becomes greater than θ_2 then s_1 is toggled and so on until “counter” become greater than θ_m . We start then comparing “counter” to $\theta_\pi - \theta_m$ where θ_π is the value of θ correspond to the half-period ($\alpha = \pi$). When “counter” becomes greater than $\theta_\pi - \theta_m$, s_1 is toggled and “counter” is compared then to $\theta_\pi - \theta_{m-1}$ until it becomes greater than $\theta_\pi - \theta_{m-1}$ then s_1 is toggled and so on until “counter” becomes greater than $\theta_\pi - \theta_1$. Then we compare it to θ_π . When “counter” becomes greater than θ_π , s_1 is toggled, “counter” is reset to 0, then we repeat the process. According to the database used during training for all ANN-i networks, it was noted that $\theta_m < 60^\circ$. besides, s_2 is phase-shifted by 120° with respect to s_1 , besides $s_1=0$ after 120° and the next switching instant is at $\theta_\pi - \theta_m$. Consequently, at the beginning signal s_2 starts at 0, its counter "counter2" starts at $\frac{\theta_{2\pi}}{3}$ and compared to $\theta_\pi - \theta_m$ then the same process for generating s_1 is repeated. The same with s_3 , we find that it starts at 1 and its counter “counter3” starts at $\frac{\theta_\pi}{3}$ and compared to $\theta_\pi - \theta_m$ then the same process for generating s_1 is repeated.

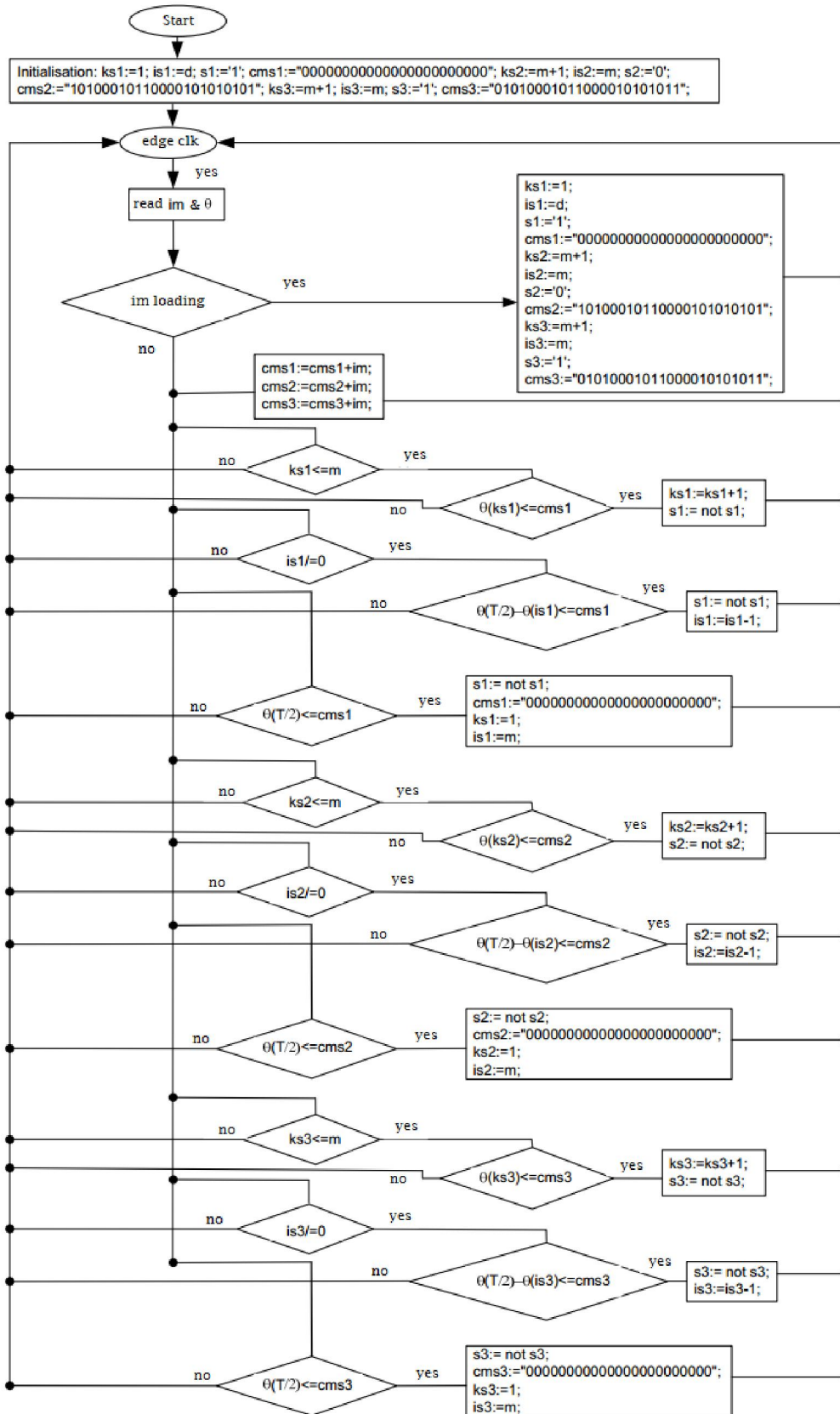


Figure 4-5 : PWM generator algorithm

4.4 Simulation and Results

In this section, the software ModelSim has been used to simulate the PWM signal generated by the algorithm ANN SHE designed by the VHDL codes. Some of these codes are generated by our C⁺⁺ based-application and the others are VHDL codes of the blocks stated in the previous section.

The ANNSHE PWM algorithm has two inputs, which are the modulation index im and the clock clk , and three outputs representing the three PWM commands out of phase by 120° .

Figure 4-6 and Figure 4-7 show under Modelsim a simulation of the implementation of the ANNSHE PWM algorithm on FPGA for different values of im .

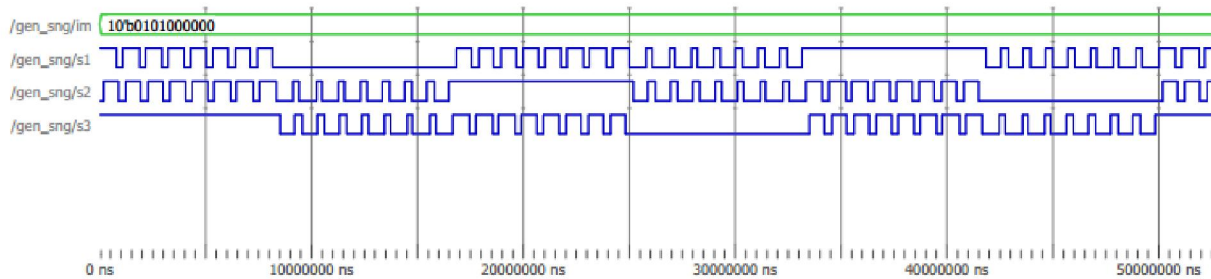


Figure 4-6: The three-phase PWM signals ($im=40\%$)

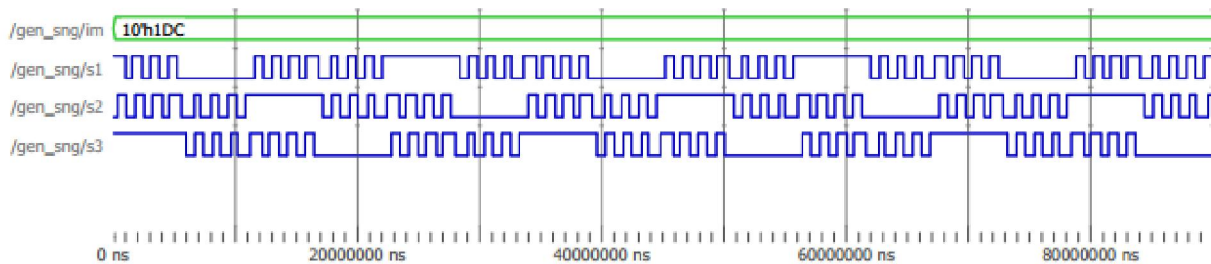


Figure 4-7: The three-phase PWM signals ($im=59.5\%$)

From the two previous figures, it can be seen that the three signals are generated in parallel and are independent of each other, and further from that figure it is noted that the signals are phase-shifted by 120° .

The signal s_1 from Figure 4-6 has 15 switching angles and it has a period $T = 50ms$, thus a frequency $f = \frac{1}{T} = 20 Hz$.

These results are confirmed by the fact that for $im = 40\%$ the selected ANN is the ANN3 which has 15 switching angles. And from the equation (4-14) we find $f = 20 Hz$.

The signal s_1 from the Figure 4-7 has 7 switching angles and it has a period $T = 33ms$, thus a frequency $f = \frac{1}{T} = 30.3 Hz$.

These results are confirmed as well by the fact that for $im = 59.5\%$ the selected ANN is the ANN4 which has 7 switching angles. And from the equation (4-14) we find $f = 29.75 \text{ Hz}$ which is close to 30.3 Hz .

The first signal s_1 in Figure 4-6 has been exported to Matlab. The Figure 4-8 shows its frequency spectrum:

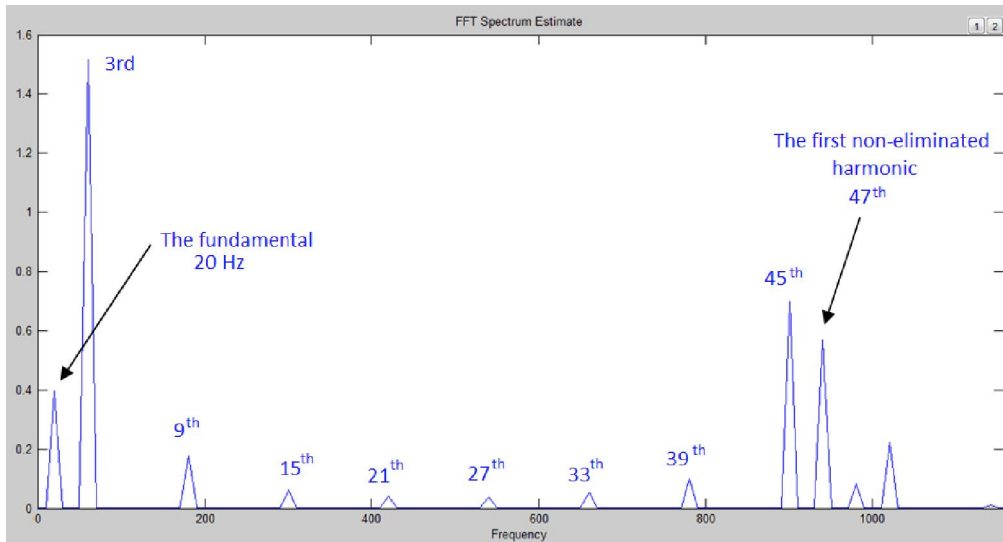


Figure 4-8 : Frequency spectrum of PWM signal for $Im=40\%$

The Figure 4-8 shows that the first non-eliminated harmonic for $m=15$ is the 47th.

The other harmonics in the figure are multiple of three harmonics. They are eliminated in the phase voltage.

This proves that the switching angles computed by the ANNs generated by our C⁺⁺ application are accurate. This confirms the good functioning of our application.

4.5 Conclusion

As a test for our C⁺⁺ based application proposed in the previous chapter, we have chosen a speed control of an inductor motor. Although the functionality of our application has already been checked, in this chapter, it is used in a real industrial application. The ANN is associated with the well-known SHE PWM to form the ANN SHE PWM algorithm. The results showed that the nearest harmonics are eliminated, thus it proves the accuracy of the ANNs generated by our application.

In this chapter we have presented at the beginning the SHE PWM and its principle of operation, then we introduced the notion of the switching angles and their calculation methods. We went after that to the technique of ANN SHE PWM, discussing the ANN topology, data representation and the training.

GENERAL CONCLUSION

In this work a C++ based-application has been created for the implementation of ANNs with a flexible topology. It generates ANNs descriptions in VHDL codes. These description topology parameters are introduced by the user who has no longer to type the whole script. This application has been then tested for a specific use which is an inductor motor control.

The desire to create such an application has come because of the importance of the artificial neural networks in today's technology development. They become more and more an open field for researches and it have increasingly been used in different industrial domains.

The hardware implementation of these networks has represented an issue, until the emergence of a highly developed FPGA circuits, which fit much better compared to other circuits such as ASICs.

To implement these ANNs in FPGA circuits, a description VHDL codes, and a development tools (e.g. ISE Xilinx) are required. Thus, to facilitate the task of writing the scripts and save times and efforts for the user, we have developed a C++ based application, with whom the user can get any ANN topology depending on his needs. In the implementation of the ANN, many challenges have been faced such as the problem of limited resource. It has been dealt with that limits by designing an optimized architecture with its own characteristic, ensuring the parallelism of the ANN. That architecture uses a specific data representation, to present the inputs, outputs and the inner signals. This representation is fit for the inner multipliers which are limited dimensions. Talking about multipliers the FPGA has a few amount of them, thus a multiplexing is needed between them. In our app only six multipliers has been used, thus six products can be calculated at the same time. Although that reduces the speed of calculation, it economizes so many resources. Moreover, a serial sum of product has been used to calculate the sum of products. To implement the activation function of the neurons, we used LUT-based method, optimized by Maher which requires 15 values to be stored for the Tangent Sigmoid, and 18 for Log Sigmoid, ensuring a minimum error.

To put our C++ based application in a real test. It has been used to generate a specific ANNs dedicated to a speed control of an asynchronous motor.

Controlling the speed of an asynchronous motor requires the use of a voltage inverter with a sinusoidal output that can vary in voltage and frequency. In real applications, control strategies produce unwanted harmonics in the output of the inverter. To solve the problem of undesirable harmonics, a well-known solution is the use of the SHE PWM command. However, the use of this command requires the resolution of a system of nonlinear equations, limiting this control strategy to off-line calculation. To get through this limitation, a new technique, based on SHE PWM and ANN, has been introduced, it is called ANN SHE PWM.

The results show that the switching angles calculated by ANN SHE PWM algorithm, are close to the exact values.

To implement the whole circuit that generates the PWM signals, we have generated the main blocks by our C⁺⁺ application which is responsible for the ANN VHDL description code, and the rest are added manually. All the blocks together formed a specific application to be implemented on FPGA. The implementation results show that the calculation of the switching instants and the generation of the PWM signals are on-line and very accurate. The three control signals are generated in parallel and are independent of each other.

In view of this work, we plan to continue working on our C⁺⁺ based application, to make it easier to the user by adding some extra options such as changing the number of the product used from a fixed number to variable, which make the ANN fit for the large and the low resources-FPGA. Besides, the data representation is fixed, and by making it variable it allows the user to introduce a larger interval of values for the inputs and other signals. Finally, implementing other types of layered neural network would offers more flexibility and diversity of the topology of the ANN.

BIBLIOGRAPHY

- [1] Simon Haykin, *Neural Networks - A Comprehensive Foundation, Second Edition*, Prentice Hall ed., 1998.
- [2] IZBOUDJEN Nouma, "Plateforme pour l' Implémentation des Réseaux de Neurones sur FPGA : Application à l' Algorithme de la Rétro Propagation du Gradient (RPG)," Ecole nationale Polytechnique, Algiers, Algeria, Thèse de Doctorat 2014.
- [3] Jeff Heaton, *Introduction to Neural Networks with C#*, WordsRU.com, Ed.: Heaton Research, Inc, 2008.
- [4] Bertram G. Katzung, *Basic and Clinical Pharmacology*.: McGraw-Hill Education, 2015.
- [5] Kevin Gurney, *An introduction to NEURAL NETWORKS*.: CRC Press, 1997.
- [6] Christopher MacLeod, *An Introduction to Practical Neural Networks and Genetic Algorithms For Engineers and Scientists.*, 2010.
- [7] Wiley Corning, "Topology of Neural Networks," New College of Florida, Sarasota, FL, Thesis June 2016.
- [8] Patrick van der Smagt Ben Krose, *An introduction to Neural Networks*.: The University of Amsterdam, 1996.
- [9] Paulo E. M. Almeida, Marcelo Godoy Simões Magali R. G. Meireles, "A Comprehensive Review for Industrial Applicability," *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, vol. 50, no. 3, JUNE 2003.
- [10] JAGATH C. RAJAPAKSE, AMOS R. OMONDI, *FPGA Implementations of Neural Networks*, Springer, Ed.
- [11] NEKKACHE Abdessalem BOUCHEKOUK Oussama, "Implémentation s'une commande ANN SHE PWM sur une carte FPGA pour un véhicule électrique," Ecole Nationale Polytechnique, Algiers, Projet de fin d'études 2015.
- [12] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 42-57, 1996.
- [13] GUELLAL Amar, "Contribution à l'étude et à l'implémentation des commandes en temps réel pour MAS," Ecole Nationale Polytechnique, Algiers, Algeria, Thèse de doctorat 2015.
- [14] BENDIB Douadi, "Etude et réalisation d'une commande MLI on-line sur circuit FPGA," Ecole Nationale Polytechnique, Algiers, Algeria, Mémoire de Magister 2009.

- [15] Marcian N. Cirstea, Eric Monmasson, "FPGA Design Methodology for Industrial Control Systems—A Review," *IEEE Transactions On Industrial Electronics*, vol. 45, no. 4, AUGUST 2007.
- [16] Zied Marrakchi, Habib Mehrez (auth.) Umer Farooq, *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*, Springer-Verlag ed. New York, 2012.
- [17] Juan J- Rodriquez-Andina, "Features, Design Tools, and Application Domains of FPGAs," *IEEE Trans. Ind. Electron.*, vol. 54, no. 4, AUGUST 2007.
- [18] G JAntoine Bioul J P Deschamps, "Synthesis of Arithmetic Circuits FPGA, ASIC, And Embedded Systems," *JOHN WILEY & SONS*, 2006.
- [19] Digilent Nexys 2 FPGA board. Data Sheet. [Online].
http://www.ece.umd.edu/class/enee245.F2016/nexys2_reference_manual.pdf
- [20] Spartan-3E FPGA family. Data sheet. [Online].
https://www.xilinx.com/support/documentation/data_sheets/ds312.pdf
- [21] Karl Leboeuf, Roberto Muscedere, Huapeng Wu, Majid Ahmadi, Ashkan Hosseinzadeh Namin, "Efficient hardware implementation of the hyperbolic tangent sigmoid function," *Proceedings - IEEE International Symposium on Circuits and Systems*, pp. 2117-2120, 2009.
- [22] Pramod Kumar Meher, "An Optimized Lookup-Table for the Evaluatiion of Sigmoid Function for Artificial Neural Networks," *IEEE/IFIP VLSI Syst. Chip Conf*, pp. 91-95, september 2010.
- [23] Jeen Shing Wang, Che Wei Lin, "A digital circuit design of hyperbolic tangent sigmoid function for neural networks," *Proceedings - IEEE International Symposium on Circuits and Systems*, pp. 856-859, may 2008.
- [24] D. Anitha, A. Muthuramalingam, S. Himavathi, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Transactions on Neural Networks*, vol. 18, pp. 880-888, 2007.