

TREBALL DE FI DE GRAU

**Grau en Enginyeria Electrònica Industrial i Automàtica**

**DIGITAL SYSTEM FOR SPIKING NEURAL NETWORK  
EMULATION**



**Report - Cost Estimation - Annexes**

<b>Autor:</b>	Eduard-Guillem Merino Mallorquí
<b>Director:</b>	Jordi Cosp Vilella
<b>Departament</b>	Enginyeria Electrònica (EEL)
<b>Convocatòria:</b>	Juny 2017



## **Abstract**

The present project is about the design, simulation and an experimental test of a digital system in a single chip able to emulate the behavior of spiking neural networks, which is possible thanks to the use of mathematical models that emulate the behavior of these networks in the brain. A modular system has been proposed in order to provide the necessary flexibility and scalability for the simulation of different neural networks. At the same time the most flexible, simple and efficient option has been chosen in order to have a good performance without losing or reducing the necessary accuracy and exactitude for the emulation of the neural networks. The solution has been implemented by making use of different combinational blocks and totally synchronous flip-flops from a 100 MHz clock signal, besides, the description of the system was performed by using the high-level hardware description language VHDL. Finally, a neural network for pattern recognition has been implemented on a programmable logical device FPGA in order to demonstrate the correct operation of the digital system.

## Resum

El present projecte tracta en el disseny, simulació i test experimental d'un sistema digital en un sol xip capaç d'emular el comportament de xarxes neuronals d'impulsos, el qual és possible gràcies al ús de models matemàtics que emulen el comportament d'aquestes xarxes en el cervell. S'ha plantejat un sistema modular per tal de dotar-lo de la flexibilitat i escalabilitat necessària per a la realització de diferents xarxes neuronals. Alhora s'ha buscat la opció més simple i alhora eficient per tal de disposar d'un bon rendiment d'aquesta sense perdre o disminuir la precisió i exactitud necessària per a la emulació de les xarxes neuronals. La solució ha estat implementada fent ús de diferents blocs combinacionals i biestables totalment síncrons a partir d'un senyal de rellotge de 100 MHz, a més, la descripció del sistema s'ha realitzat mitjançant el llenguatge de descripció hardware d'alt nivell (VHDL). Finalment, per demostrar el correcte funcionament del sistema digital s'ha realitzat una xarxa neuronal per al reconeixement de patrons, la qual s'ha implementat sobre un dispositiu lògic programable FPGA.

## Resumen

El presente proyecto trata el diseño, simulación y test experimental de un sistema digital en un solo chip capaz de emular el comportamiento de redes neuronales de impulsos, el cual es posible gracias al uso de modelos matemáticos que emulan el comportamiento de estas redes en el cerebro. Se ha planteado un sistema modular para dotarlo de la flexibilidad y escalabilidad necesaria para la realización de diferentes redes neuronales. A la vez se ha buscado la opción más simple y a la vez eficiente para disponer de un buen rendimiento de esta sin perder o disminuir la precisión y exactitud necesaria para la emulación de las redes neuronales. La solución ha sido implementada haciendo uso de diferentes bloques combinacionales y biestables totalmente síncronos a partir de una señal de reloj de 100 MHz, además, la descripción del sistema se ha realizado mediante el lenguaje de descripción hardware de alto nivel (VHDL). Finalmente, para demostrar el correcto funcionamiento del sistema digital se ha realizado una red neuronal para el reconocimiento de patrones, la cual se ha implementado sobre un dispositivo lógico programable FPGA.



## Acknowledgements

First and foremost, I have to thank my thesis supervisor Jordi Cosp. Without his assistance and dedicated involvement in every step throughout the process, this project would have never been accomplished.

I am also grateful to Electronic Engineering Department, for providing me with all the necessary facilities for the development of this project.

Last but not the least, I would like to thank my family: my mother Marta and my sister Mireia for helping me and supporting me whenever I needed it.





# Index

<b>ABSTRACT</b>	<b>I</b>
<b>RESUM</b>	<b>II</b>
<b>RESUMEN</b>	<b>III</b>
<b>ACKNOWLEDGEMENTS</b>	<b>V</b>
<b>1. INTRODUCTION</b>	<b>1</b>
1.1. Project scope.....	1
1.2. Objectives.....	2
1.3. Biological background.....	2
1.3.1. Neural Networks.....	2
<b>2. ARTIFICIAL NEURAL NETWORKS</b>	<b>7</b>
2.1. Threshold Logic Unit .....	8
2.2. Multilayer Perceptron.....	9
2.3. Spiking Neural Networks.....	10
<b>3. PRELIMINARY STUDY OF ALTERNATIVES</b>	<b>13</b>
3.1. Bluehive.....	13
3.2. One million neuron single-FPGA neuromorphic system.....	14
3.3. SNAVA .....	15
<b>4. SPIKING NEURAL NETWORKS</b>	<b>17</b>
4.1. Neural models.....	17
4.1.1. Integrate-and-Fire model .....	17
4.1.2. Hodgkin-Huxley model .....	18
4.1.3. Izhikevich model.....	19
4.2. Neuronal connectivity.....	21
4.2.1. AER System .....	22
4.3. Learning of neural networks.....	23
4.3.1. Spike-Timing-Dependent Plasticity .....	23
<b>5. DESIGN AND IMPLEMENTATION OF A SNN</b>	<b>25</b>
5.1. Izhikevich neuron .....	26
5.1.1. Model adaptation.....	26
5.1.2. Design and architecture .....	27

5.1.3. Simulations.....	31
5.2. AER System .....	33
5.2.1. Design and architecture.....	33
5.2.2. Simulations.....	35
5.3. Spike-Timing-Dependent Plasticity .....	36
5.3.1. Design and architecture.....	36
5.3.2. Simulations.....	39
5.4. SNN Emulation .....	41
5.4.1. Design and architecture.....	42
<b>6. PATTERN RECOGNITION WITH A SNN.....</b>	<b>45</b>
6.1. Neural network model.....	45
6.2. Pattern recognition.....	46
6.3. Training method .....	47
6.4. FPGA implementation .....	49
6.4.1. Inputs and outputs.....	49
6.4.2. Design and architecture.....	50
6.4.3. Simulations.....	51
6.4.4. Experimental results .....	57
<b>CONCLUSIONS .....</b>	<b>65</b>
<b>COST ESTIMATION .....</b>	<b>69</b>
<b>REFERENCES .....</b>	<b>71</b>
<b>ANNEXES .....</b>	<b>75</b>
A1. User's Manual .....	75
A2. Computer files .....	81
A3. Blueprints.....	113





# 1. Introduction

One of the current challenges for the scientific community is to understand how the brain works and being able to reproduce it elsewhere, whether biologically or electronically, with the same properties. This is an ongoing research which is evolving, but unlike the search for a time-travel device or faster-than-light travel, there is a solid evidence that such quest is possible since we are the perfect example of an intelligent system [1].

Motivated by biological discoveries, many scientists take an interest on a modern approach of artificial intelligence. Therefore, they are able to replicate the behavior of the brain it may be possible to achieve a system which can become aware and react to the world that it is surrounded by.

Combining experimental studies of the biological nervous systems, from either animals or humans, several mathematical models have been developed, on which researchers try to produce a model that is sufficiently accurate and has a computational efficiency. Nevertheless, because of the available computational capabilities of the current electronic devices, it is already possible to simulate those neural systems to even fifty thousand times faster than their biological counterparts [2].

The digital system presented in this project is based on the fundamental characteristics of a biological neural network, in order to perform several functions associated with the brain like reasoning, speech recognition, movement or visual processing. Thus, the focus of this project is to proportionate a modular, flexible and scalable digital system, which is able to simulate the behavior of an artificial neural network for different uses.

## 1.1. Project scope

This project consists about the design and physical implementation of a digital system that is able to emulate the behavioral characteristics of biological neural networks in order to perform several functions associated with the brain; like reasoning, speech recognition, movement or visual processing. To achieve it, three fundamental main modules have to be designed in order to obtain a fully functional neural network.

Firstly, a module that provides the simulation of the properties, characteristics and behavior of a single neuron using an accurate, but at the same time computationally efficient mathematical model which can be adapted for its further implementation into the digital system.

Secondly, a communication or transmission system that is able to manage all the information flow of the neural network between an undetermined number of neurons.

Additionally, a training algorithm is designed to provide a functionality to the neural network, such as; like speech, movement recognition or visual processing as stated above.

Finally, the interconnection between them with the correct configuration will allow the simulation of a neural network with the given functionality. Therefore, the final digital system of this project has to be a modular, flexible and scalable platform, to cover all of the possibilities that involve the simulation of a neural network.

## 1.2. Objectives

Next, the main goals set from the very beginning of the project are written below, they are the essential requirements for the design and implementation of the digital system:

- Collecting information about the functionality of a neural network and its different models.
- Studying different models of neural networks and their most important characteristics.
- Choosing the most flexible and efficient neural network model to implement.
- Designing the digital system for neural network simulation.
- Making the description of the digital system through the high-level language VHDL.
- Verifying of the correct operation of the design through functional simulations of the digital system for neural network simulation.
- Implementation of the digital system in a programmable logic device (FPGA) and corroboration of its functionality.

## 1.3. Biological background

The aim of this section is to provide enough biological background, in order to understand the functionality of the design presented in this project. Therefore, the behavior of a neuron is described in detail, as well as the operational and transmission characteristics of the information flow between the neurons of a biological neural network.

### 1.3.1. Neural Networks

The brain is a very complex network with millions and millions of interconnected neurons which cooperate to efficiently process input signals in order to decide the required output action. Approximately every neuron sends its output signals to over 10.000 other neurons, thus complicates the flow of information. To put it mildly, we do not even understand the brain as well as we think. In fact, we do not totally understand the behavior of a single neuron [3].

Nevertheless, there is a rough concept of how neurons operate: neurons send out short pulses of electrical energy as signals, though only if they have received enough energy from pulses of other neurons. This simple mechanism has been translated to different mathematical models, so that it becomes possible to perform computer simulations of neurons.

#### 1.3.1.1. Neurons

The specialized cells of the nervous system are the neurons, they are the structural and functional unit of the nervous system. Their main characteristic is the electrical excitability of their plasmatic membrane, along with their specialization in the reception of the stimuli and conduction of the nerve impulses between them [4].

The structure of the neurons can be diverse, but they have a common pattern, which can be distinguished into three main parts:

- The cell body or soma: the wider area, which forms the nucleus of the neuron and where there are all the organs that drive the cellular activity by the neuron.
- Dendrites: these are the extensions of the cell body which allow the connection with other neurons, as well as being responsible for receiving nerve impulses.
- Axon: a long and unique extension per neuron, which has the function of sending nerve impulses generated by the body cell to the next neuron.

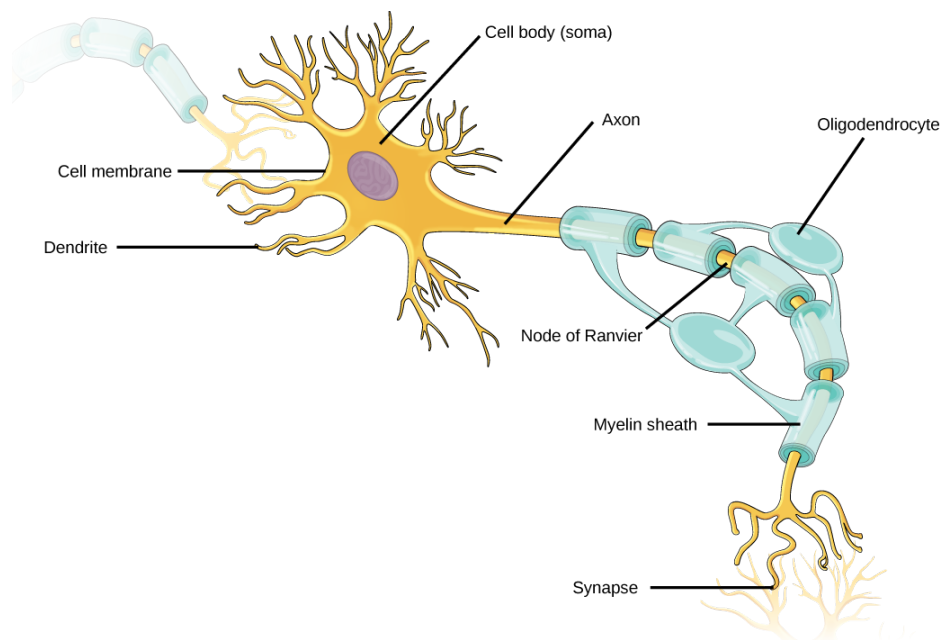


Figure 1.1. Schematic drawing of a neuron [4].

### 1.3.1.2. Neural transmission

In a neural network, the information is transmitted by nerve impulses that cause polarity changes in the membranes of the cells and propagate through the neurons as if they were small electrical currents, ranging from dendrites and passing through the neural body until the axon. To make this communication between the neurons possible, they establish connections called synapses.

Initially, inside the neuron there are proteins and ions with negative charge. This difference in ion concentration produces a potential difference between the outside and inside of the membrane of the neuron. In fact, the usual value is about -70 mV [5].

Moreover, when a nerve impulse reaches a neuron which is at its resting state the membrane depolarizes, achieving the variation of the potential difference that presents the neuron. In the event that the depolarization causes enough variation in the membrane potential, it is said that the neuron has reached the action potential and thus generates a nerve impulse, always with the same intensity, which is transmitted to the next neuron.

It has to be noted, that the transmission of nerve impulses follows the law of all or nothing. This means that if the membrane depolarization does not reach a minimum potential, called the threshold potential, the impulse is not transmitted.

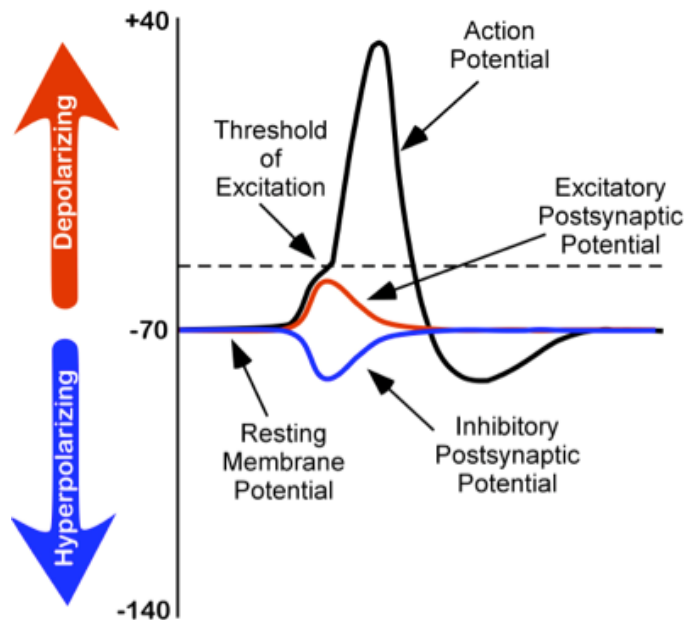


Figure 1.2. Evolution of the membrane potential of a neuron [5].



In addition, the post-synaptic impulses can be either positive or negative called, excitatory or inhibitory, respectively. One neuron receives about ten thousand potential synapses. Therefore, the sum of these potentials determines the value of polarization of the membrane of a neuron.

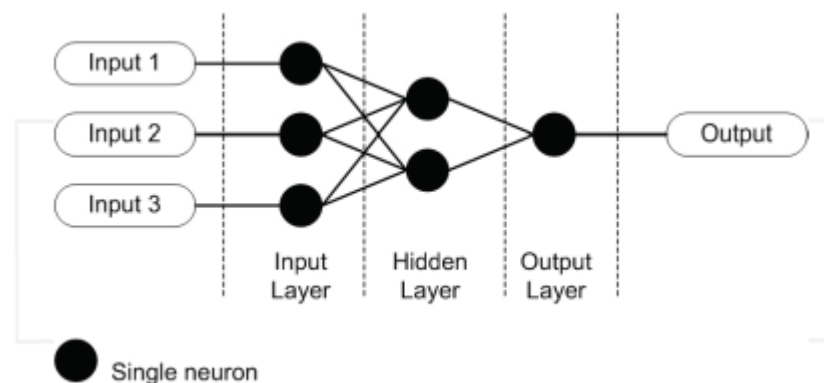
In short, as shown in Figure 1.2, if a neuron receives inhibitory potential, the polarization increases, however, if it receives an excitatory potential, the polarization of the membrane decreases. Only if the stimulus is sufficiently large, so that the depolarization reaches the threshold of excitation, the neuron sends a spike that is transmitted to the neural network. Then, the neuron enters into a short moment of rest, called the refractory period, in which it cannot send another spike again. Finally, the neuron will return to its initial state of rest if it does not receive more stimulus that perturbate its membrane potential charge.



## 2. Artificial Neural Networks

An artificial neural network (ANN) is a mathematical model that is based on the functionality of biological neural networks and thus, it can be defined as an emulation of biological neural systems. It is designed to produce and replicate the intelligent behavior, ANN's are at the vanguard of computational systems. Unlike the classical Artificial Intelligence approach, it intends to develop systems to directly simulate rational or logical reasoning, artificial neural networks aims lie at the reproduction of the underlying processing mechanisms that give a system its intelligence.

In order to take the full advantage of the artificial neural networks it is needed to interconnect the individual neural networks or its fundamental units, the neurons, in a topology that contributes to an easier, faster and more efficient problem solving. In the past, researchers have developed a series of "standardized" topographies of artificial neural networks that are suited for solving different types of problems. Therefore, after choosing the type of functionality that the neural networks need to offer, it is required to decide the appropriate topology and fine-tune it.



**Figure 2.1.** Example of a simple artificial neural network [6].

In addition, before using the neural network it is an indispensable condition to perform training to teach it problem solving. There are three major learning paradigms: supervised learning, unsupervised learning and reinforced learning. Even though these learning paradigms have their differences in their training methods they all have one thing in common: they train the neural network so it gives the desired output response in line with a series of input signals.

Some of the advantages of the artificial neural networks are [5]:

- It can be used to solve linear as well as non-linear programming tasks.

- If a component of an ANN fails, the net continues to operate (based on its highly parallel nature).
- A neural network learns and does not have to be re-programmed.
- An ANN can be used to solve classification, movement recognition or visual processing related problems.

On the other hand, the main cons of the artificial neural networks are:

- Most ANN's require a training phase to operate or function.
- As an ANN's architecture differs from microprocessors, they have to be simulated.
- Large ANN's require powerful hardware to run and accomplish reasonable execution times.

To sum up, artificial neural networks have been in use for some time now and their main application is in the field of robotics. They can be used to plan and direct the way of an autonomous vehicle, recognize obstacles or perform a classification of images. Essentially, artificial neural networks are capable of learning and generalizing from examples and experience to obtain solutions, and as its biological predecessor they are considered an adaptive system.

## 2.1. Threshold Logic Unit

The fundamental unit of an artificial neural network is the neuron. The first computational model for neural networks was based on mathematics and algorithms, which was the Threshold Logic Unit [7], developed by Warren McCulloch and Walter Pitts in 1943.

$$\varphi(v) = \begin{cases} 1, & v \geq 0 \\ 0, & v < 0 \end{cases} \quad (\text{Eq. 2.1})$$

As shown in equation 2.1, the output of a neuron takes on the value of 1 if the total internal activity level of that neuron is nonnegative and otherwise 0. This statement describes the all-or-nothing property of the McCulloch-Pitts model.

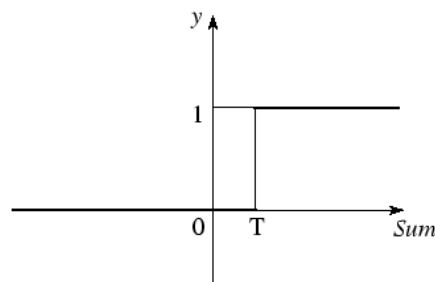
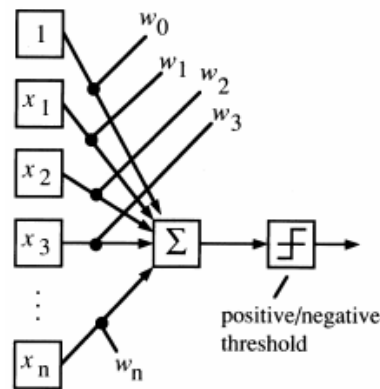


Figure 2.2. Threshold activation function [7].

Additionally, the Threshold Logic Unit model of a neuron is simple yet has substantial computing potential and a precise mathematical definition. Nevertheless, it is so simple that the weight and threshold values are fixed.

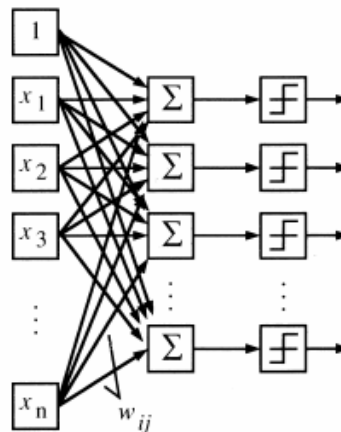
## 2.2. Multilayer Perceptron

The simplest form of an artificial neural network used for the classification of patterns, which are linearly separable, is the perceptron. The single-layer perceptron consists of a single McCulloch-Pitts neuron with adjustable synaptic weights and threshold. Such network is only able to perform pattern classification with only two classes [9].



**Figure 2.3.** Single-layer perceptron [9].

In order to perform more complex functions, several single-layer perceptron can be combined to obtain a neural network of multilayer perceptron as shown in figure 2.4.



**Figure 2.4.** Multilayer perceptron [9]

When the perceptrons were invented, many researchers speculated that the intelligent systems could be developed out of perceptrons. Nevertheless, as the ongoing research was evolving it turned out that it was impossible to develop a convenient learning algorithm. As an example, the exclusive-or (XOR) operation could not be solved. Only when McCulloch-Pitts neurons were replaced by neural models with a differentiable activation function, a back-propagation learning algorithm was invented.

## 2.3. Spiking Neural Networks

The first ideas and models of artificial neural networks are over fifty years old, hence they are already becoming an old technique within the computer science field. The first generation of artificial neural networks consisted of McCulloch-Pitts threshold neurons, as explained above. Rather than using a step or threshold function to compute its output signals, the second generation uses a continuous activation function, making them acceptable for analogue input and output. The last and third generation is what we call the spiking neural networks [10].

The spiking neural networks are the third generation of neural networks which raise the level of biological realism by the use of individual spikes. Just like real neurons do, this functional characteristic allows the codification of spatial-temporal information in communication and computation. Therefore, instead of using rate coding this type of neural networks use mechanisms where neurons receive and do send out individual pulses, called pulse coding, allowing the codification of information as frequency and amplitude of sound.

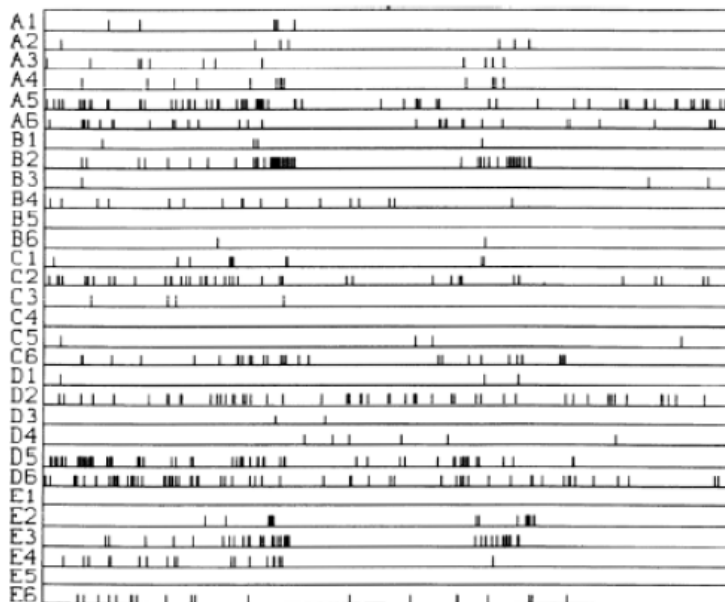


Figure 2.5. Spike-trains [10].

The neuronal signal of a spiking neural network consists of short electrical pulses called spikes. These pulses, called action potentials or spikes, have an amplitude of about 100 mV and a duration of 1-2 ms. As shown in Figure 2.5 individual neurons send out sequences of spikes, or spike-trains, which alter dramatically in frequency over a short period of time. Thus, neurons have to use spatial and temporal information of incoming spike patterns to encode their message to other neurons.

In short, since all neuron spikes of a spiking neural networks look alike, the form of the action potential does not carry any information. Therefore, it is the number and timing of these spikes which actually matter.



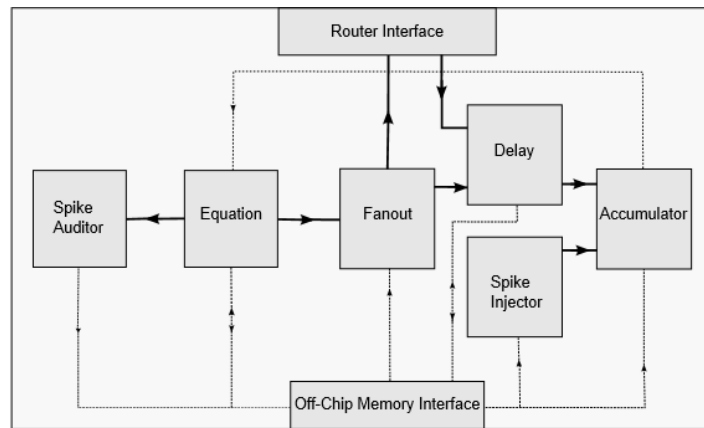


### 3. Preliminary study of alternatives

This chapter presents a few hardware implementations of artificial neural networks on FPGAs. A general analysis of these implementations has been performed in order to point out their benefits and drawbacks of their computational structures.

#### 3.1. Bluehive

Bluehive is a field-programable custom computing machine for extreme-scale real-time neural network simulation, which is capable of emulating 64k neurons along with 64M synapses per FPGA, aimed to be used for scientific simulations with high demanding communication requirements [11].



**Figure 3.1.** Processing engine of a node in Bluehive [11].

The design places the focus on the communication mechanism and it uses the Izhikevich neural model for neural networks simulations. The core SNN emulation is done by the processing engine that includes the following functional components:

- **Equation Processor:** calculates the equation of the Izhikevich neural model to performs the neuron computation.
- **Fan-out Engine:** takes the neuron firing events, looks up the destination nodes to be notified and the delay to be implemented and farms it out.
- **Delay-Unit:** performs the first part of the fan-in phase. Messages are placed into one of the sixteen 1ms bins, thereby delaying them until the right 1ms simulation time step.
- **Accumulator:** performs the second part of the fan-in phase, accumulation weights to produce an I-value for each neuron.

- **Router:** routes firing events destined for other processing nodes.
- **Spike auditor:** records spike events to output as the simulation results.
- **Spike injector:** allows external spike events to be injected into the simulated network. This is used to provide an initial stimulus. It could also be used to interface to external systems.

### 3.2. One million neuron single-FPGA neuromorphic system

The one million neuron single-FPGA neuromorphic system is an architecture that gives the approach for building a one million neuron system on a single FPGA. It is capable of implementing several neurons like the simple integrate and fire or the Izhikevich models, with the objective to use the system for multimodal scene analysis [12].

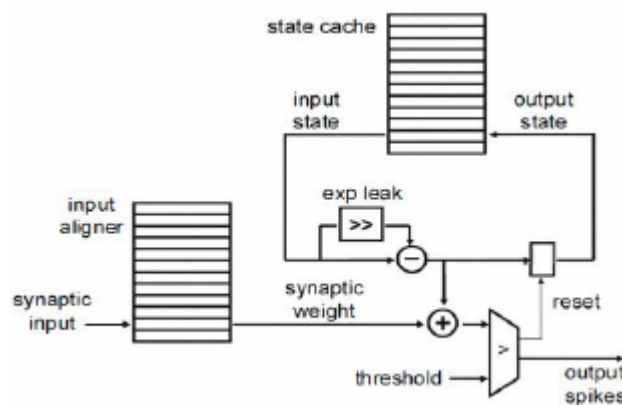


Figure 3.2. Neuron block diagram of the implementation [12]

This implementation uses the Address Event Representation (AER) communication system to handle the transmission of spikes between the neurons of the simulated neural network while the mapping of the network and the synaptic weights are stored in an external memory and fetched every cycle.

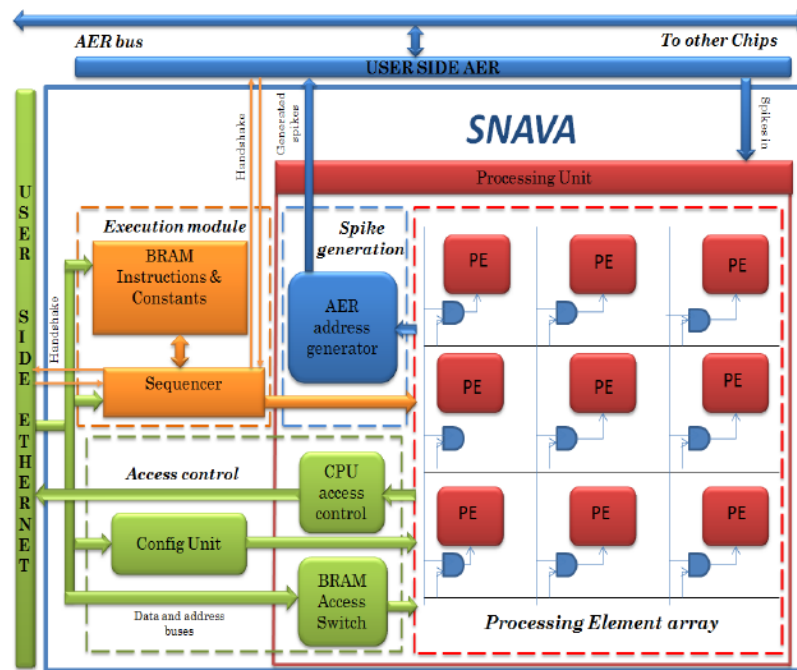
Figure 3.2 shows the neuron block diagram that uses a state cache to save the state of the several neurons that can be simulated in the same physical engine. Therefore, this block uses time multiplexing to simulate several neurons.

Finally, this design is capable of emulating up to 1 million of neurons, yet there are some drawbacks to it as it needs a large state cache and there is a significant time consumption due to the time multiplexing, as explained above.

### 3.3. SNAVA

Designed to be a flexible spiking neural network emulator, SNAVA (Spiking Neural-network Architecture for Versatile Applications) is a Harvard hardware architecture capable to simulate any spiking neural network model in which the transmission of information between the neurons take place through spikes [13].

Therefore, the architecture has been designed with the aim to ensure that it can be reprogrammed to simulate different spiking neural networks models in an efficient way, giving a high level of flexibility and scalability.



**Figure 3.3.** Block diagram of the SNAVA architecture [13].

SNAVA consists in a scalable array of SIMD (Single Instruction Multiple Data) processing elements that assure a complete parallel execution of operations which is highly effective for the simulation of SNN since they are parallel by nature.

Furthermore, this architecture uses two communications protocols to provide the flow of information required for a SNN simulation. First the Address-Event Representation (AER) communication system, and second the Ethernet protocol. Moreover, this architecture enables the interconnection of several FPGAs to perform a SNN emulation.



## 4. Spiking neural networks

The next chapter seeks to make a brief description and analysis of the different possibilities that exist for the implementation of a spiking neural network. It has the aim of designing a flexible, modular, scalable and efficient system without losing the precision and accuracy needed for the simulation of a neural network in order to achieve its subsequent implementation. Consequently, this chapter introduces different neural models, as well as a communication system for transmitting data between different neurons and, finally, a training method for spiking neural networks.

### 4.1. Neural models

The neural models are a mathematical approach of the behavior and the properties inspired by biological neural networks, which provide the capacity to reproduce their functionality in the field of computer simulations.

There are many different models for a spiking neural network. However, the type of model to use will depend on the needs that are required for the artificial neural network to simulate, as well as the available resources for such purpose. In other words, the challenge lies in finding a model that is sufficiently accurate and computationally efficient.

Further below, several neural models were analyzed to validate their suitability in the design of the proposed spiking neural network, always looking for the compromise between the model complexity and the cost of the implementation.

#### 4.1.1. Integrate-and-Fire model

The Integrate-and-Fire model [14], proposed in 1907 by the French neuroscientist Louis Lapicque, is recognized as one of the simplest neural models to establish itself as a canonical model to make way for a large number of variants.

In this model, a neuron is represented in time with the following equation:

$$I(t) = C_m \frac{dV_m(t)}{dt} \quad (\text{Eq. 4.1})$$

Where  $I(t)$  is the current injected to the neuron,  $C_m$  is the neuron membrane capacitance and  $V_m(t)$  is the neuron membrane potential. Therefore, when a current is introduced to the input, the voltage

of the membrane increases with time until it reaches a constant threshold  $V_{th}$ , point in which the neuron generates a spike and the membrane voltage is reset to its initial value.

Moreover, you can get a more accurate model entering a refractory period  $t_{ref}$  that limits the frequency of firing of the neuron, preventing the generation of spikes during this period. In such a scenario, the function of the firing frequency with a constant input current is:

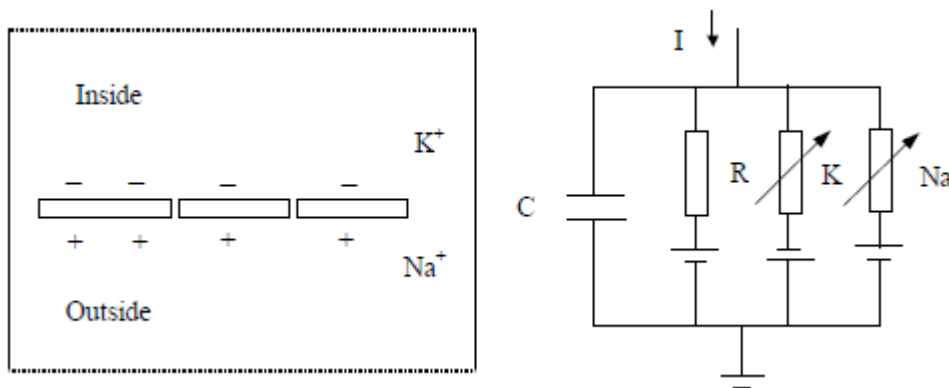
$$f(I) = \frac{I}{C_m V_{th} + t_{ref} I} \quad (\text{Eq. 4.2})$$

Ultimately, due to the lack of time memory this model has the disadvantage that, at any time, if you get a signal below the threshold, the voltage of the membrane will suffer an increase but it will not return to its initial value until it fires again. Consequently, the spike frequency of this model is constant. Thus, this particular characteristic does not correspond within the observed behavior of neurons in a real biological neural network.

#### 4.1.2. Hodgkin-Huxley model

The Hodgkin-Huxley model [15] was described by physiologists and biophysicists Alan Hodgkin and Andrew Huxley in 1952 to explain the ionic mechanisms underlying the initiation and propagation of action potentials in the axon of a giant squid. It is a complex mathematical model but really accurate that describes how the potentials in neurons are initiated and transmitted. Thanks to this work they received the Nobel prize in physiology and medicine in 1963.

This model is easily explained by Figure 4.1. Firstly, the semipermeable membrane of the neuron separates the inside of the extracellular fluid and acts as a capacitor.



**Figure 4.1.** Schematic diagram of the Hodgkin-Huxley model [15].

If an input current  $I(t)$  is injected into the neuron, it can charge the capacitor or provide a leakage current through the different channels of the membrane of the neuron. Due to the transport of ions across the cell membrane, the ion concentration in the neuron is different than the one in the extracellular fluid. Therefore, the conservation of electric charge in the membrane implies that the applied current  $I(t)$  can be divided into a capacitive current  $I_{cap}$ , which charges capacitor  $C$  and a current  $I_k$  that is injected from presynaptic neurons.

$$I(t) = I_{cap}(t) + \sum_k I_k(t) \quad (\text{Eq. 4.3})$$

Thus, the sum extends over all the ions. In the Standard model of Hodgkin-Huxley there are only three types of channels:

- Sodium channel with index Na.
- Potassium channel with index K.
- Unspecified leakage channel with resistance R.

From the definition of the Capacity,  $C = \frac{Q}{u}$ , where  $Q$  is a charge and  $u$  is the voltage across the capacitor. Since the current of the capacitor is  $I_{cap}(t) = C \frac{du}{dt}$ , then:

$$C \frac{du}{dt} = - \sum_k I_k(t) + I(t) \quad (\text{Eq. 4.4})$$

Therefore, in biological terms,  $u$  is the voltage across the membrane and  $\sum_k I_k(t)$  is the sum of the ionic currents that pass through the membrane of the neuron. In short, Hodgkin-Huxley formulated the three components with the following equation:

$$\sum_k I_k(t) = g_{Na} m^3 h (u - E_{Na}) + g_K n^4 (u - E_K) + g_L (u - E_L) \quad (\text{Eq. 4.5})$$

Where  $E_{Na}$ ,  $E_K$  and  $E_L$  are the reversal potentials and  $g_{Na} m^3 h$ ,  $g_K n^4$  i  $g_L$  are the conductance resistances.

#### 4.1.3. Izhikevich model

The Izhikevich model [16] was designed with the intention to develop large-scale models of brain impulses by neural networks. Therefore, to achieve this, you need a model for a single neuron that is computationally simple, but capable of producing pulse patterns exhibited by biological neurons.

Under these premises, Eugene M. Izhikevich in 2003 presented a simple neuron model for spiking neural networks, which is as biologically plausible like the Hodgkin-Huxley model, and at the same time as computationally efficient as the Integrate-and-Fire model.

The author has reduced the accurate biophysiological models of Hodgkin-Huxley neurons into a two-dimensional system of differential equations:

$$v' = 0,04v^2 + 5v + 140 - u + I \quad (\text{Eq. 4.6})$$

$$u' = a(bv - u) \quad (\text{Eq. 4.7})$$

If

$$v \geq 30 \text{ mV}, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \quad (\text{Eq. 4.8})$$

Where  $v$  and  $u$  are dimensionless variables, and  $a$ ,  $b$ ,  $c$ , and  $d$  are dimensionless parameters, and  $' = \frac{d}{dt}$ , where  $t$  is the time. In biological terms, the variable  $v$  represents the membrane potential of the neuron and  $u$  represents a membrane recovery variable, which simulates the activation of  $K^+$  ionic currents and inactivation of  $Na^+$  ionic currents, and it provides negative feedback to the membrane potential of the neuron,  $v$ .

Regarding the behavior of the neuron, the positive synaptic currents  $I$  from other neurons increase the value of the membrane potential. In the case that these currents are not sufficient to make the neuron generate an impulse or spike, the voltage of the membrane is reset to its initial value. Alternatively, if the neuron generates a spike (+30 mV) due to sum of their input current, the membrane voltage  $v$  and the recovery variable  $u$  are reset according to equation 4.8.

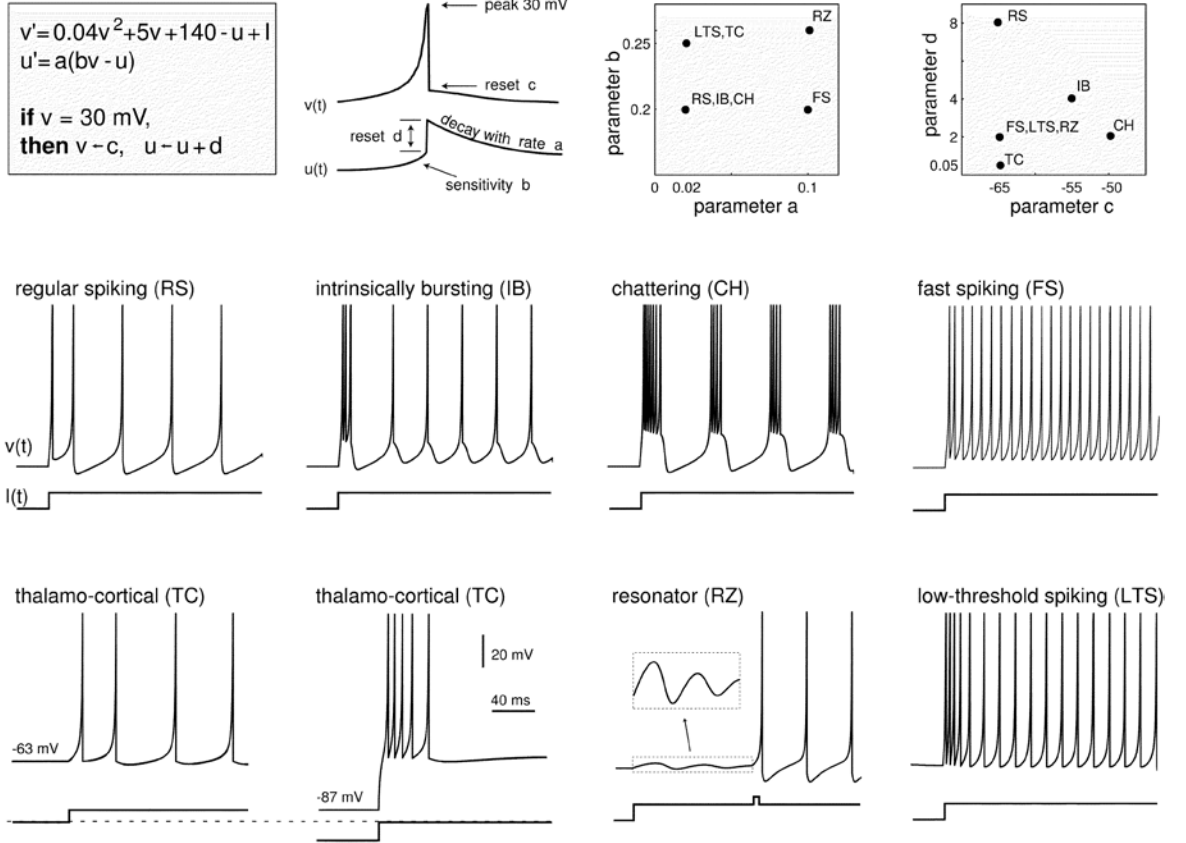
The resting membrane voltage in this model is between -70 and -60 mV depending on the value of the parameter  $b$ . In addition, just like real neurons, the model does not have a fixed threshold, hence depending on the history of the membrane potential before the generation of a spike the potential threshold can be as low as -55 mV or as high as -40 mV.

The following considerations have to be taken into account for the use of the parameters exhibited in this model:

- The parameter  $a$  describes the time scale recovery of the variable  $u$ . Small values result in a slow recovery. A typical value is  $a = 0,02$ .
- The parameter  $b$  describes the sensitivity of the recovery variable  $u$  to the subthreshold fluctuations of the membrane potential  $v$ . A typical value is  $b = 0,2$ .



- The parameter  $c$  describes the after-spikes reset value of the membrane potential  $v$ . A typical value is  $c = -65 \text{ mV}$ .
- The parameter  $d$  describes the after-spike reset of the recovery variable  $u$ . A typical value is  $d = 8$ .



**Figure 4.2.** Different types of neurons based on the parameters  $a$ ,  $b$ ,  $c$  and  $d$  [16].

Conclusively, as shown in Figure 4.2, setting different parameters values result in different patterns of intrinsic activation, which allows the behavior emulation of diverse real biological neurons.

## 4.2. Neuronal connectivity

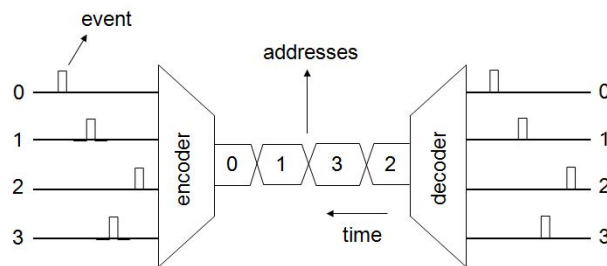
The digital systems implemented for neural networks emulation are still far from an equal efficiency in neural computation or neural coding like the real biological neural networks. Computers use a million times more energy for an operation than a real brain. Video cameras use a thousand times more bandwidth per bit of information than retinas do [17]. Due to these and other shortcomings, today we still cannot replace the damaged parts of the nervous system.

Due to this, it is not surprising that a small but growing community of engineers are trying to build systems that meet the efficiency and effectiveness of their biological references, in order to match the efficiency of the performance and computational communication by nature.

One of these problems arises when trying to establish the communications between neurons of an emulated neural network. In fact, the neural systems need to connect millions of neurons and thus establish a connection-efficient chip implementation, which creates a major challenge. Therefore, this section will discuss the AER system (Address-Event Representation) which aims to reduce some of these shortcomings and has been implemented in the emulated neural network of this project.

#### 4.2.1. AER System

Mahowald and Sivilotti proposed a system of event representation to be able to transmit pulses of a number of neurons on a chip to the appropriate location in an array of neurons in a second chip.



**Figure 4.3.** Schematic of the AER system [18].

In the schematic shown in Figure 4.3 an encoder assigns a unique address for each neuron that generates a spike. Then, a bus transmits these addresses to a decoder that selects the appropriate location of the spike.

This communication system is quite efficient to avoid the bottlenecks that occur when the information needs to be exchanged in a massively interconnected system, like the neural networks impulses or spikes.

Nevertheless, a number of issues should be considered in order to achieve an efficient implementation of the AER system. One of them is the case in which two or more events occur at the same time, then the system needs to decide in which order they are transmitted through the bus, since it can only transmit one address per time unit.

### 4.3. Learning of neural networks

To understand how the mammalian cerebral cortex performs its calculations, it is necessary to understand mainly two aspects. First, we must have a good understanding of the neuronal processing units, the neurons; and secondly, we must gain a better understanding of how the mechanisms of these neurons combine to build functional systems.

This section talks about the STDP (Spike-Timing-Dependent Plasticity) as a method for building artificial neural networks to perform complex computational operations or solving pattern recognition tasks.

#### 4.3.1. Spike-Timing-Dependent Plasticity

The STDP (Spike-Timing-Dependent Plasticity) [19] is a biological process responsible for altering the connections, or synapses, of all the neurons in a spiking neural network. To do so, it strengthens or weakens the connectivity between the neurons based on the degree of synchronization of their spikes. This degree of connectivity is commonly known as the weight of the link or synapse.

This rule is a method of unsupervised learning, the concept of which is to strengthen synapses that contribute to the generation of an output spike, while those that do not contribute, i.e. those that generate spikes after the output spike, are weakened.

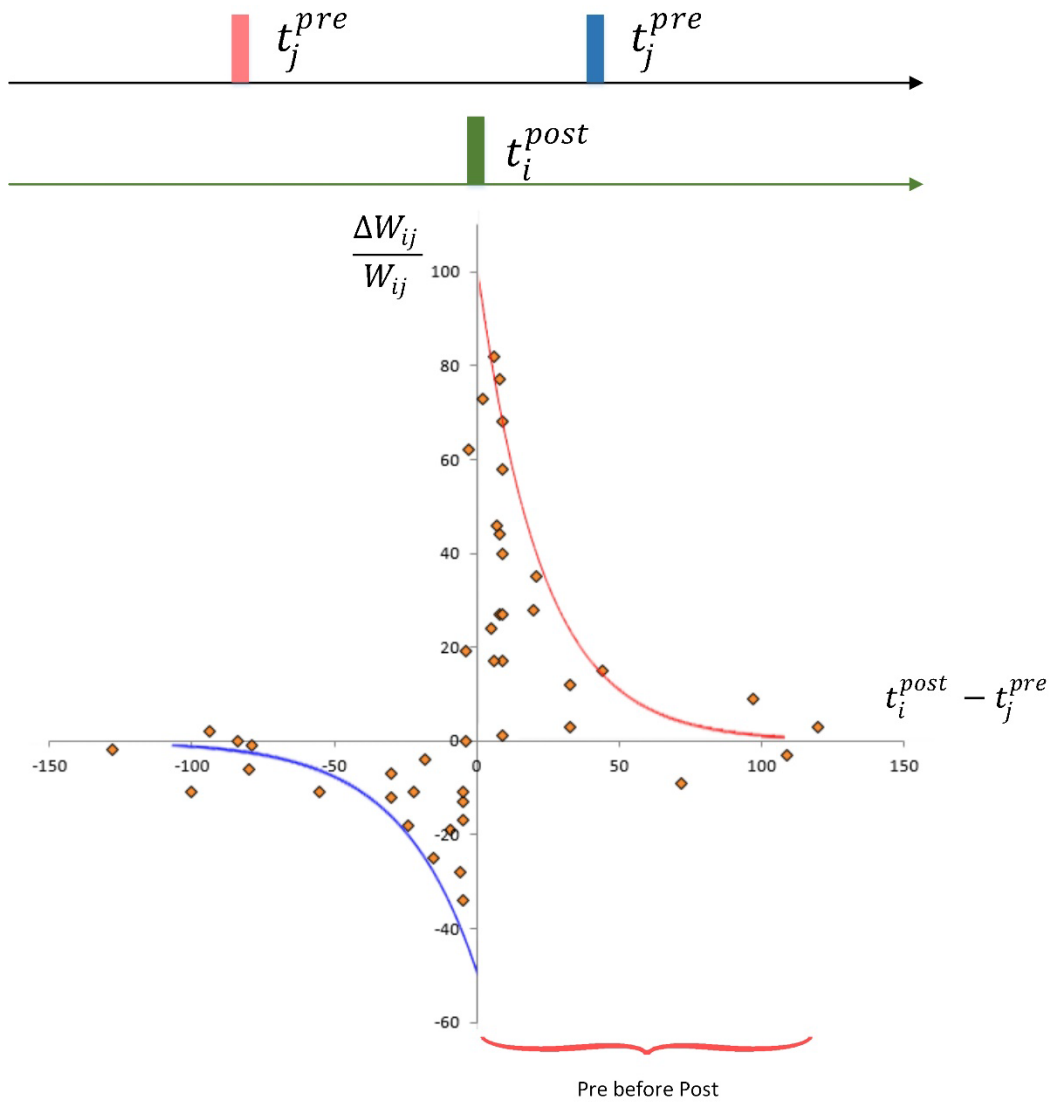
Considering a presynaptic neuron  $i$  and a postsynaptic neuron  $j$ , the function that characterizes the modification of the synaptic weight is as follows:

$$\Delta w_j = \sum_{k=1}^N \sum_{l=1}^N W(t_j^l - t_i^k) \quad (\text{Eq. 4.9})$$

Along with the function, which defines the degree of increase or decrease of the synaptic weight based on the timing of the impulses between the pre- and postsynaptic neurons expressed as:

$$W(x) = \begin{cases} A_+ \exp\left(-\frac{x}{\tau_+}\right) & \text{if } x > 0 \\ A_- \exp\left(\frac{x}{\tau_-}\right) & \text{otherwise.} \end{cases} \quad (\text{Eq. 4.10})$$

In equations 4.9 and 4.10,  $t_j^l$  represents the activation time  $l^{th}$  of the neuron  $j$ ; similarly,  $t_i^k$  represents the activation time  $k^{th}$  of the neuron  $i$ ;  $A_+$  and  $A_-$  are the constants that define the extent of the change in the synaptic weight (at  $t = 0_+$  and  $t = 0_-$  respectively); and,  $\tau_+$  and  $\tau_-$  are the constants of the exponential decrease in the change of the synaptic weight.



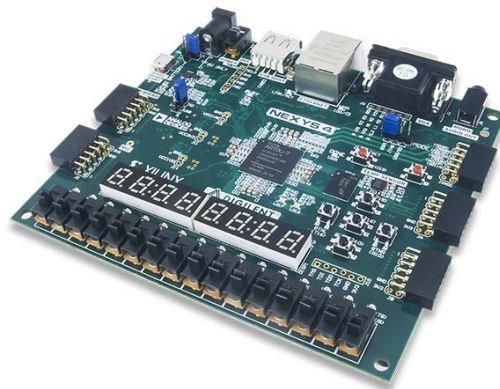
**Figure 4.4.** Graphical representation of the STDP learning rule [20].

Figure 4.4 represents equation 4.10 of relative weight changes based on the time between the pre-spikes and post-spikes of the synapsis between two neurons. Therefore, it shows the reduction of the synaptic weight when a presynaptic neuron fires after a postsynaptic neuron; and on the contrary, an increase of the synaptic weight from a presynaptic neuron to a postsynaptic neuron if a presynaptic neuron fires before the postsynaptic neuron.

## 5. Design and implementation of a SNN

The simulations of neural networks, due to their intrinsic characteristic of being formed by thousands of interconnected neurons, require high computing power which in turn requires high computational power that can exceed the computing power of a generic microprocessor.

Thanks to the technological development, today we have sufficient tools to provide customized hardware systems with the ability to obtain a high computing power by reducing the energy consumption and needed resources. The fact of designing and implementing a microprocessor which is designed to perform a particular task allows us to optimize and maximize its performance with respect to a generic microprocessor.



**Figure 5.1.** Nexys 4 Artix-7 FPGA [21]

One of the tools that allow the development and implementation of microprocessors or digital systems are the FPGA (Field-Programmable Gate Array). A FPGA is a semiconductor device based on a matrix of configurable logic blocks connected via programmable interconnections. The aim of this chapter is to explain the design and architecture of the different modules that form the spiking neural network and their interconnection to achieve its implementation in a FPGA.

In order to provide the maximum flexibility and scalability possible for the emulation of SNNs a modular system has been proposed. In the following sections, it is explained the design and architecture of a neuron, the AER communication system and the STDP learning method which interconnected allow the emulation of a SNN. Finally, in the next chapter, a SNN for pattern recognition is proposed as a proof-of-concept application.

## 5.1. Izhikevich neuron

To simplify the computing power necessary to reproduce the neuron model of Izhikevich and optimize the resources used in the FPGA, a series of adaptations have been made, similar to the fixed-point implementation of the model proposed in [22].

Due to the transition of the neuron model equations from the real numbers to the digital domain, all the properties of binary numbers have been studied in order to avoid performing complex multiplications and divisions that could complicate the implementation and exceed the resources available in a FPGA for the emulation of large-scale neural networks.

### 5.1.1. Model adaptation

Initially, the numbers of the equations for the digital system are represented with a signed vector of 13 bits, therefore with one bit for the sign of the number. In order to start working with the equations, the parameters corresponding to the variables  $a$ ,  $b$ ,  $c$  and  $d$  have been set to their recommended values in order to obtain a regular spiking neuron model proposed by Izhikevich. Therefore, the following equations have been obtained:

$$v' = 0,04v^2 + 5v + 140 - u + I \quad (\text{Eq. 5.1})$$

$$u' = 0,02(0,2v - u) \quad (\text{Eq. 5.2})$$

If

$$v \geq 30 \text{ mV}, \text{ then } \begin{cases} v \leftarrow -65 \text{ mV} \\ u \leftarrow u + 8 \end{cases} \quad (\text{Eq. 5.3})$$

Secondly, it has been decided to work with a binary representation of integers. Hence, due to the lack of decimal numbers in a binary vector representation a multiplication per ten has been performed. Thus, the membrane potential,  $v$ , along with the recovery voltage,  $u$ , are multiplied by ten, obtaining the following model:

$$v' = \frac{1}{250}v^2 + 5v + 1400 - u + I \quad (\text{Eq. 5.4})$$

$$u' = \frac{1}{50} \left( \frac{1}{5}v - u \right) \quad (\text{Eq. 5.5})$$

If



$$v \geq 300, \text{ then } \begin{cases} v \leftarrow -650 \\ u \leftarrow u + 80 \end{cases} \quad (\text{Eq. 5.6})$$

On the other hand, a way to save resources in the FPGA implementation is doing a power of two multiplications or divisions. Therefore, if you calculate a power of two multiplication in a binary number, you only need to shift left the binary vector as many positions as power of two multiplications need to be made. On the contrary, if you calculate a power of two division a shift right operation in the binary vector is enough to obtain the result.

In short, the coefficients of the equations have been adjusted to achieve multiplications and divisions by power of two in order to optimize the design. Firstly, for the equation 5.4 the number 250 has been replaced by 256, since  $256 = 2^8$ . Secondly, for equation 5.5 the divisors 50 and 5 have been replaced by 64 and 4 respectively, since  $64 = 2^6$  and  $4 = 2^2$ . In addition, the multiplication  $5v$  and  $v$  term of the equation 5.4 have been replaced to implement a sum of six terms of the membrane potential. Since the sum of the six terms can be expressed as a sum of a  $2v$  and  $4v$ , this two power of two multiplications have been implemented as shift left operations. Therefore, the following equations for the adapted model have been obtained:

$$v[n + 1] = 2v[n] + \frac{1}{2^8} v^2[n] + 2^2 v[n] + 1400 - u[n] + I[n] \quad (\text{Eq. 5.7})$$

$$u[n + 1] = u[n] + \frac{1}{2^6} \left( \frac{1}{2^2} v[n] - u[n] \right) \quad (\text{Eq. 5.8})$$

If

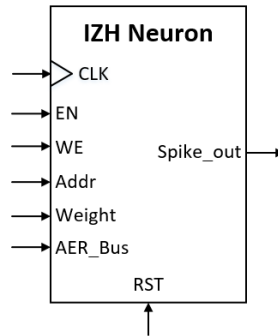
$$v[n + 1] \geq 300, \text{ then } \begin{cases} v[n + 1] \leftarrow -650 \\ u[n + 1] \leftarrow u[n + 1] + 80 \end{cases} \quad (\text{Eq. 5.9})$$

### 5.1.2. Design and architecture

As explained above, a modular, flexible and scalable digital system has been proposed in order to include all the different possibilities that can exist in the emulation of a spiking neural network, while trying to maintain its simplicity.

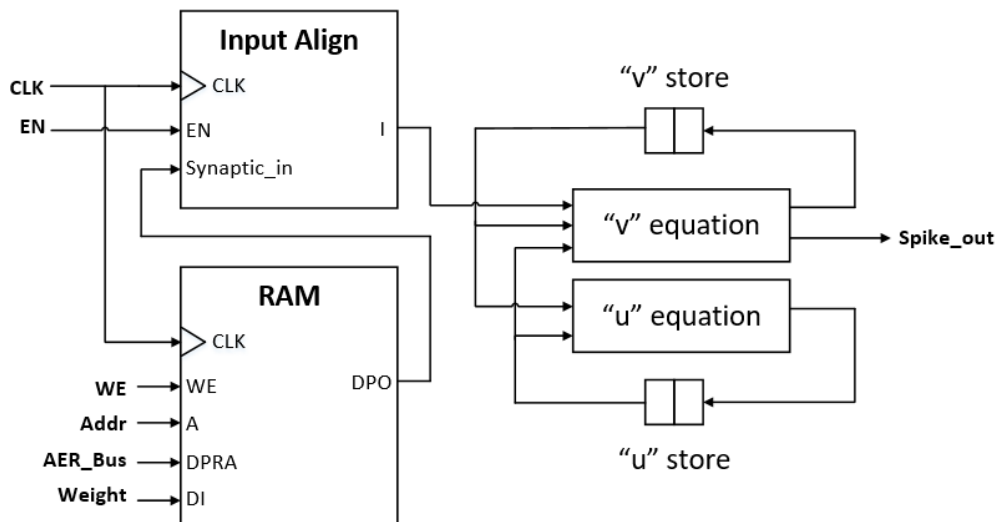
The designed module for the neuron implementation has seven entries, including the clock signal, and one output. Furthermore, the neuron has a small Random Access Memory (RAM), where it stores the different weights of the synaptic connections with other neurons.

Looking at Figure 5.2 and going in order, firstly, there is the clock signal (*CLK*), which is responsible for coordinating the different actions of the neuron; secondly, and activation signal (*EN*) which serves to activate the functioning of the neuron; thirdly, there are the signals write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) that are used to write to the internal RAM of the neuron; fourthly, there is the input of the AER bus (*AER\_Bus*) where the neuron reads which neuron of the neural network generated a spike; finally, there is a binary output signal (*Spike\_out*) which indicates whether the neuron generated a spike.



**Figure 5.2.** Digital bloc of a neuron.

Lowering a level in the implementation of the neuron, shown in Figure 5.3, there are the different sequential and combinational digital blocks that operate the actions of the neuron. These are: an internal RAM, a sequential block for the weight synaptic input, two registers for the membrane potential and voltage recovery, and two combinational blocks corresponding to the implementation of the differential equations presented above.



**Figure 5.3.** Block diagram of the neuron.



The internal RAM of the neuron is a matrix that forms a column of synaptic weights. As shown in Table 5.4, each position in the column corresponds to the weight of a synaptic connection with a neuron, so the first column corresponds to the synaptic weight of the connection with the zero neuron of the neural network.

NEURON	SYNAPTIC WEIGHT
0	70
1	-40
2	0
3	120
4	-15

**Table 5.1.** RAM of the neuron.

Therefore, if the value of the synaptic weight is zero, it means that there is no link between these two neurons. In addition, the used model allows the use of excitatory and inhibitory synaptic weights, i.e. positive and negative respectively. Also, as mentioned above, it is possible to modify the synaptic weights of the connections with write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals. Moreover it is possible to initialize the RAM of a neuron with a .mif text file.

Sequentially the functionality process of the neuron is as follows: firstly, it reads the AER communication bus for the neuron that it fired. Then, the RAM is responsible for reading the synaptic weight associated with the number written on the AER bus and it sends it to the *Input Align* sequential bloc. This bloc will dispatch the synaptic weight to the combinational block of the potential membrane equation, also it is responsible for adding the synaptic weights in the case that several neurons fired at the same time. For this purpose, as it will be explained in the next section, the AER bus stops the activity of the neuron deactivating the *EN* signal, since it cannot transmit more than one spike at once and writes one of the addresses of the neurons that fired in each clock cycle.

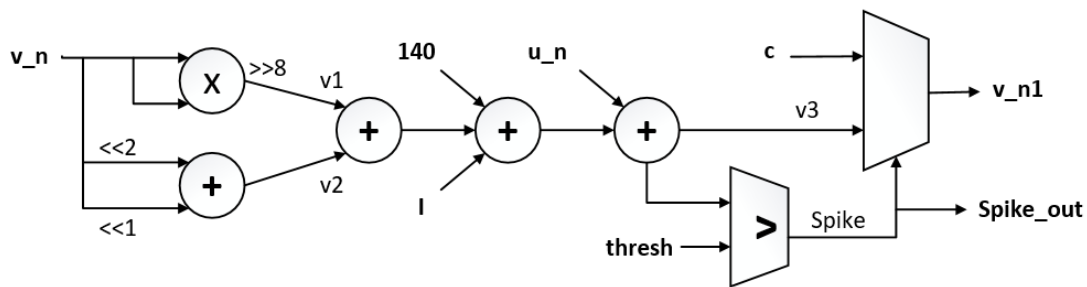
Also, the Input Align block limits the negative value that a synaptic weight can have to -140 mV because for values under -140 mV the neuron ends up generating a spike when, biologically, the neuron should not excite for inhibitory synaptic weights. To verify that this anomaly was not a consequence of the adapted model for this implementation, several simulations were performed with the original model and it reproduced the exact same behavior with high negative synaptic weights.

Finally, the combination blocks corresponding to the model equations perform their operations with the registers "*v*" store and "*u*" store where the signals  $v_n$ ,  $v_{n1}$  and  $u_n$  and  $u_{n1}$  are stored respectively.

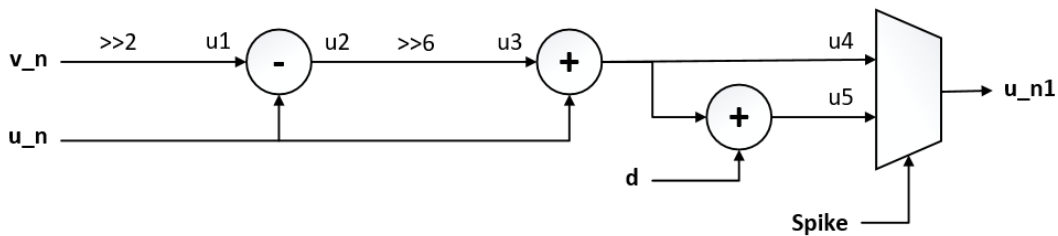
The digital implementation of the adapted equations can be seen in detail in Figure 5.5. The top of the diagram corresponds to the equation 5.7 where; firstly, the signal  $v1$  is generated by the square of  $v_n$

signal and a binary shift right operation of eight positions; secondly,  $v_2$  signal is generated by the sum of a binary shift left operation to  $v_n$  of one and two positions respectively; thirdly, the equation is followed by performing the sum of other factors including the synaptic weight; finally, using a comparator the design determines whether the value of the membrane voltages exceeds the threshold indicated in equation 5.9. If so, the spike signal is activated indicating the firing of the neuron. In addition, if the neuron exceeds the threshold value the membrane potential is reset to the  $c$  value, otherwise it continues to operate with signal  $v_3$  until the next clock cycle in which the “ $v$ ” store register will replace the  $v_n$  signal for the  $v_{n1}$  signal.

### “ $v$ ” Equation implementation



### “ $u$ ” Equation implementation



**Figure 5.4.** Digital implementation of the adapted neuronal equations.

Following with Figure 5.4, the bottom of the diagram corresponds to equation 5.8 where; primarily, the signal  $u_1$  is generated by performing a two binary shift right operation of the membrane voltage  $v_n$ ; leading on from this, the signal  $u_3$  is obtained by signal  $u_1$  minus  $u_n$  and a six binary shift right operation. Finally, the signals  $u_4$  and  $u_5$  are driven to a multiplexer with a channel selection based on whether the neuron fired or not (Spike signal) to set the recovery value of the membrane  $u_{n1}$  as equation 5.9 indicates.

Additionally, Figure 5.5 shows the comparison between the resources utilization of the adapted (left) and non-adapted (right) neuron models corresponding to equations 5.4-5.6 and 5.7-5.9 respectively.

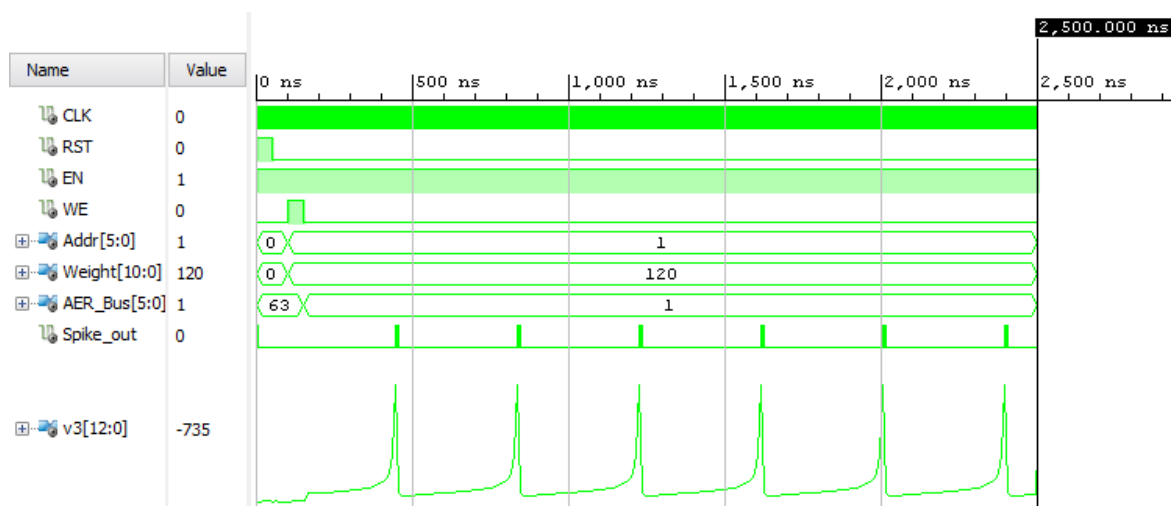
Resource	Utilization	Available	Utilization %	Resource	Utilization	Available	Utilization %
LUT	134	63400	0.21	LUT	786	63400	1.24
LUTRAM	16	19000	0.08	LUTRAM	16	19000	0.08
FF	130	126800	0.10	FF	169	126800	0.13
DSP	1	240	0.42	DSP	1	240	0.42
IO	28	210	13.33	IO	28	210	13.33
BUFG	1	32	3.13	BUFG	1	32	3.13

**Figure 5.5.** Resource utilization comparison of a single Izhikevich neuron between the adapted (left) and non-adapted (right) digital model.

As it can be observed, for the implementation of a single neuron into a Nexys4 Artix-7 FPGA the non-adapted model uses about 6 times more look-up tables than the adapted model. Therefore, this difference will greatly increase when building a neural network of ten, one hundred or more neurons. Finally, by doing some rough estimations it could be said that the used FPGA could implement almost 500 neurons of the adapted model. Whereas, only 80 neurons of the non-adapted model, which equates to a difference which cannot be ignored.

### 5.1.3. Simulations

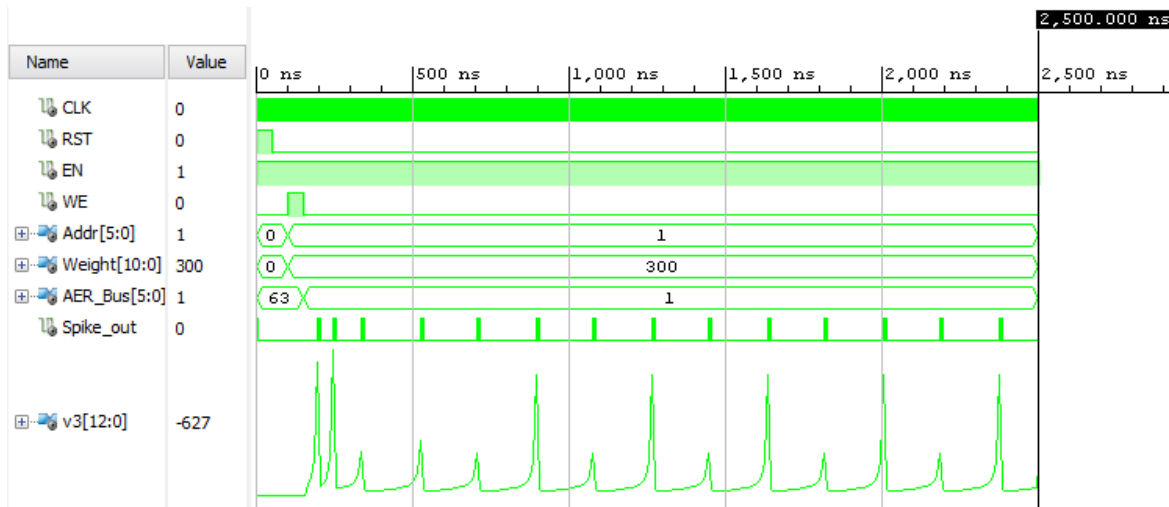
Here are various simulations of the implemented model made with Vivado in order to show the different behaviors exhibited by the neuron against different scenarios.



**Figure 5.6.** Timeline of inputs, outputs and membrane potential of the neuron to an input step of 12 mV.

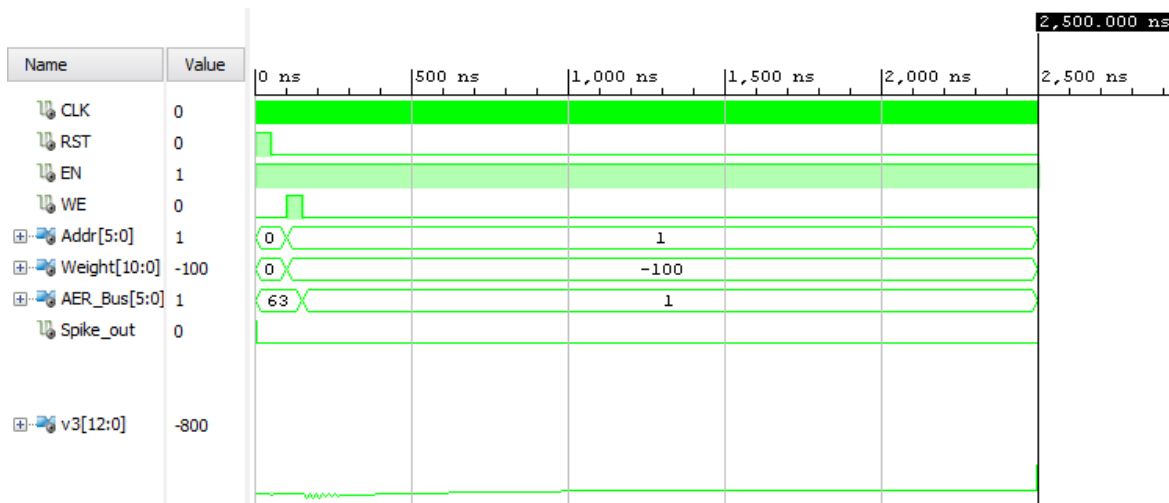
In Figure 5.6 the response of the neuron to an input step of 12 mV is shown. Firstly, in the 50 ns a synaptic weight of 12 mV with neuron 1 is written in the internal RAM, keeping the *WE* signal activated. A few clock cycles later the address of the neuron 1 is written in the AER bus, simulating the constant firing of this neuron and therefore, generating an input step of 12 mV for the simulated neuron. Finally,

what can be seen is how the neuron starts generating neuronal impulses to the output signal *Spike\_out*.



**Figure 5.7.** Timeline of inputs, outputs and membrane potential of the neuron to an input step of 30 mV.

Secondly, in Figure 5.7 the neuron exhibits a similar behavior as in Figure 5.6 as a response to an input step of 30 mV and therefore generating a higher frequency of spikes.



**Figure 5.8.** Timeline of inputs, outputs and membrane potential of the neuron to an input step of -10 mV.

Finally, the neuron shows the response to an input step of -10 mV in Figure 5.8, thus since it is an inhibitory input the neuron does not generate any spikes. In essence, the implemented neuron shows the behavior exhibited by the original model proposed by Izhikevich, as shown in Figure 4.2 of the previous chapter.

## 5.2. AER System

Due to the design of the neuron, this is capable of reading the information written on the AER (Address-Event Representation) bus and through its own RAM, it will apply itself to the corresponding synaptic weight of the link between itself and the firing neuron.

Consequently, unlike the system proposed in the previous chapter, consisting of an encoder and decoder, the design of the AER system of this project must consist of an encoder that reading the spikes of all neurons of the SNN is capable of translating them to their corresponding neural address, to finally write in the AER communication bus, the neuron that fired.

Moreover, this design has addressed the problem of processing two or more events at the same time due to the inability to transmit more than one address through the AER communication bus, adding a condition of priority to the encoder.

### 5.2.1. Design and architecture

The AER system block diagram is shown in Figure 5.9, where at the top there is the block of the AER system that has two inputs and two outputs corresponding to the clock signal (*CLK*), spikes vector (*Spikes*), and an activation signal for the neurons (*EN\_Neuron*) and the AER communication bus (*AER*) respectively.

The main signal is the spikes vector (*Spikes*) that is a vector which contains as many bits as neurons form the neural network. Therefore, in the case of a neural network of five neurons the spikes vector is as: "00000". Where with a "1" it indicates the firing of a neuron, thus if neuron zero and three generate a spike at the same time, the AER system will have to process the following vector: "01001". As a result, the number 3 will be written at the output of the AER bus and at the next clock cycle the number 0 will be written in order to avoid collapsing the bus by writing two address at once. Additionally, due to the delay caused by the transmission of an impulse for each clock cycle, the AER system has a *EN\_Neuron* signal used to stop all neurons connected to the neural network, so they have time to read the synaptic weight associated with each neuron that has fired and to compute their sum.

Going down one level in the implementation, as shown in Figure 5.9, the AER system is composed of four components, a type of buffer or memory called FIFO (First in, first out), a multiplexer, a priority encoder and a comparator. This AER system has two types of operation depending on whether there is one or more of a neural spike at the same time.

Continuing with the previous spikes vector, "01001", the AER system works as follows. First, the buffer called FIFO drives the vector to the multiplexer, on which if there have not been any spikes the signal *FIFO\_En* will be activated and it will transmit the vector directly to the priority encoder.

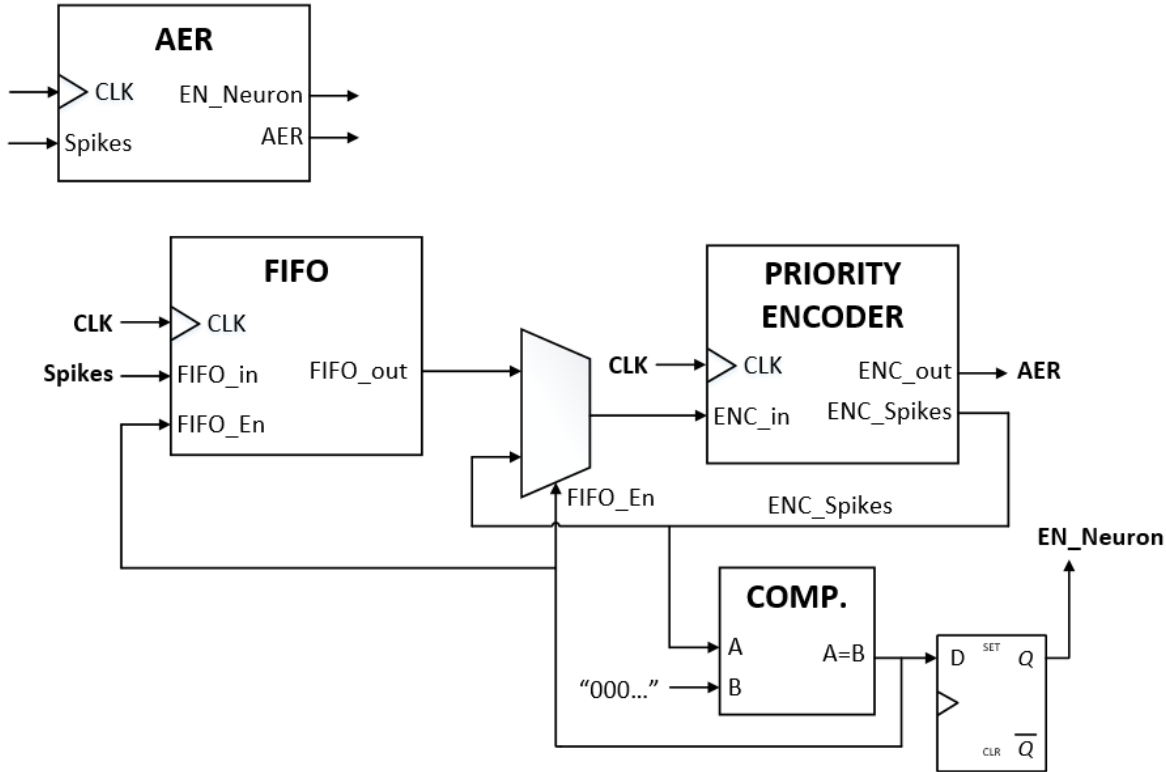


Figure 5.9. AER system block diagram.

One clock cycle later, the priority encoder works with the vector "01001" starting to read every bit from left to right and therefore, detecting a "1" in position four of the vector which corresponds to the third neuron. Finally, it writes the address 3 to the AER bus and generates a second signal, *ENC\_Spikes*, which is the same vector but with the spike of the neuron 3 being reset: "00001".

Then this vector is written to the comparator to determine if it is equal to a zero vector, i.e. a vector with all bits to zero. If not, the signal *FIFO\_en* along with *EN\_Neuron* are disabled, stopping all the neurons. Also, the multiplexer sends to the priority encoder the vector *ENC\_Spikes*, "00001", so it can continue to work with the rest of the other spikes. Meanwhile the FIFO performs an operation similar to the one shown in Figure 5.10. It is stopping the output of data and starting the storage of the different input spikes vectors that can arrive while the priority encoder is working with vector "01001".

Once the priority encoder finished to write all the spikes to the AER communication bus, the comparator will reset the *FIFO\_En* signal and the FIFO will start transmitting all the vectors that it saved in its memory.

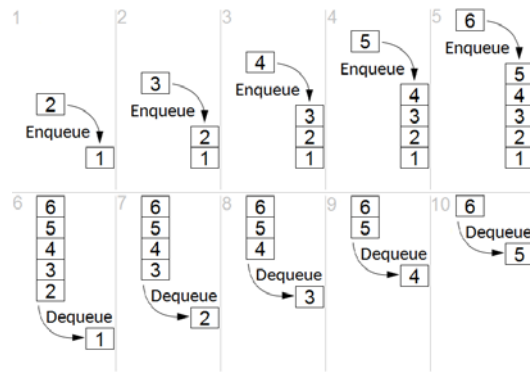


Figure 5.10. Representation of FIFO functionality [23].

Note that, in principle, FIFO memory should not keep any spikes vector while the priority encoder is working since all the neurons of the neural network are stopped with *EN\_Neuron* signal. However, this system is useful just in case external impulses arrive during this period, such as the spikes of the first layer of the neural network.

### 5.2.2. Simulations

A simulation of the AER system made with VIVADO is shown in Figure 5.11 in order to show the different behaviors exhibited this communication bus in front of the spikes generated by a neural network.

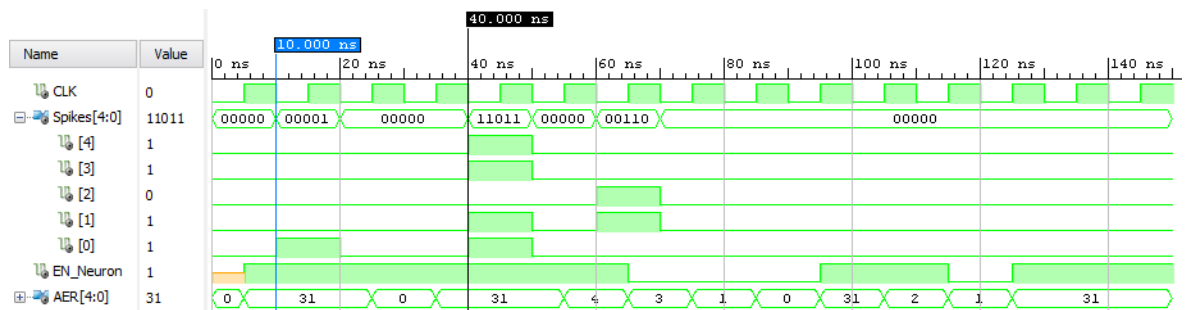


Figure 5.11. Timeline of the AER system behavior.

Firstly, as it can be seen in Figure 5.11 when the AER bus does not detect any spikes it writes all of their bits to “1”, so in this particular case, it shows the number 31, i.e. the AER bus can show up to 31 addresses from 0 to 30, including 0.

Secondly, at 10 ns a spike is generated by neuron number zero, therefore, according to the functionality described above, the AER bus writes the 0 address in the next clock cycle in its output.

Finally, at time 40 ns several spikes are generated by neurons number 4, 3, 1 and 0. Thus, the priority encoder starts working and the *EN\_Neuron* signal is deactivated to stop the activity of all the neurons

so it can write, in every clock cycle, the address of the firing neurons. Besides, during the transmission of these spikes neurons number 1 and 2 fire. Then, FIFO stores the corresponding spikes vector and the addresses of these two neurons are written once the priority encoder finishes the spikes transmission of neurons 4, 3, 1 and 0, implementing the functionality as shown in Figure 5.10 of the previous section.

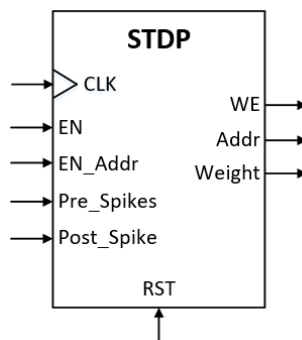
### 5.3. Spike-Timing-Dependent Plasticity

Recent efforts in artificial intelligence studies suggest that the software can be trained and taught to obtain a behavior that goes beyond the reproduction of a fixed sequence of events. Learning is a distinction that separates the intelligent systems from the unintelligent. Thus, researchers are directing considerable effort in developing learning skills for neuronal networks and other synthetic systems.

Therefore, this chapter proceeds to explain the design and implementation of the Spike-Timing-Dependent (STDP) learning rule, that modifies the synaptic weights of the connections between neurons depending on the synchrony of their firing. That is, the synapses that contribute to the generation of an output spike of the neural network should be enhanced, while those not contributing to the generation of a spikes in the output must be weakened.

#### 5.3.1. Design and architecture

The digital block shown in figure 5.12 has been designed for the implementation of the STDP learning rule based on the digital logic approach from [24]. This block has six inputs, including the clock signal, along with three outputs corresponding to the write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals of the internal RAM of the neuron.



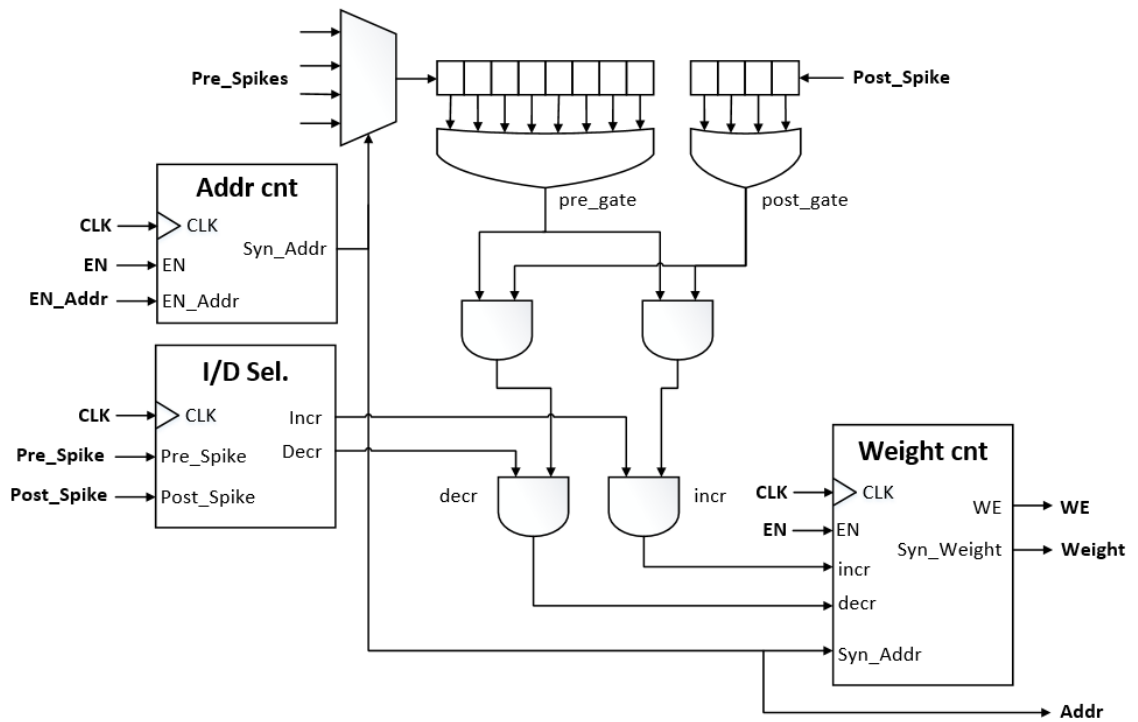
**Figure 5.12.** Digital block of the STDP module.

The training system described in the previous chapter is responsible of modifying the weights of all the connections of the neural network. Due to the complexity of creating a module to handle all the



synaptic weights of the neural network the following solution has been proposed: create a training module that is in charge of the connections of a single neuron. Therefore, having a learning module for each neuron of the neural network that is responsible of all the links coming to that neuron, and thus to maintain and update the RAM of it.

Observing the Figure 5.12 and going in order, primarily, there is a clock signal (*CLK*) which is responsible for coordinating the different actions of the learning module; following this, an activation signal (*EN*) which serves to activate the learning; next, an activation signal (*EN\_Adr*) that is responsible for changing the connection in which the STDP rules is applied; thus, two signals *Pre\_Spikes* and *Post\_Spike* that are the responsible of reading the firings of the previous neurons and the spike of the neuron where the STDP module is connected respectively; and finally, at the output there are the write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals that allow to write in the RAM of the neuron. In addition, there is a reset signal (*RST*) that along with the enable signal (*EN*) allow to reset all the synaptic weights of the neuron's RAM.



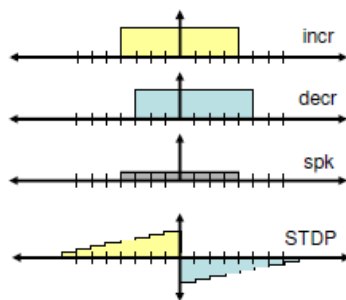
**Figure 5.13.** Block diagram of the STDP module.

As shown in Figure 5.13, there are several interconnected combinational and sequential blocks for the STDP module to function. These are: a counter address (*Addr cnt*) to select on which synaptic link the STDP rule is applied, an increment or decrement link selector (*I/D Sel.*) that activates the corresponding signal whether the pre-spike happens before or after the post-spike of the connection, a synaptic weight counter (*Weight cnt*) that is responsible for storing and modifying the synaptic weight of all the

connections of the neuron, and finally a set of combinational blocks that allow the digital logic implementation of equations 4.9 and 4.10 from the STDP learning rule.

Sequentially the operational process of the learning module is as follows: initially, the pre-spikes and post-spike signals are being read while the address counter is in charge of selecting the connection to apply the STDP rule with the *Syn\_Addr* signal, which at the same time serves as a selector channel for the multiplexer, indicates to the weight counter the link to modify and proportionate the address of the neuron's RAM.

After, in the event of a spike from the previous neurons of the neuron in which the STDP module is connected, the *I/D Sel* block activates the corresponding output signal to indicate whether the weight counter needs to increase or decrease the connection's value of the synaptic weight. Also, the spike is propagated through the shift register and activating the *pre\_gate* signal. Then, if the neuron of the training module fires, its spike is propagated by the corresponding shift register activating the *post\_gate* signal. The activation of these two logic gates along with the increased signal provided by the *I/D Sel* block generates an increment pulse for the synaptic weight counter. Therefore, depending on the duration of this pulse counter it will increase more or less the weight of the synaptic connection. Moreover, it will activate the write enable signal (WE) to allow the update of the neuron's RAM.



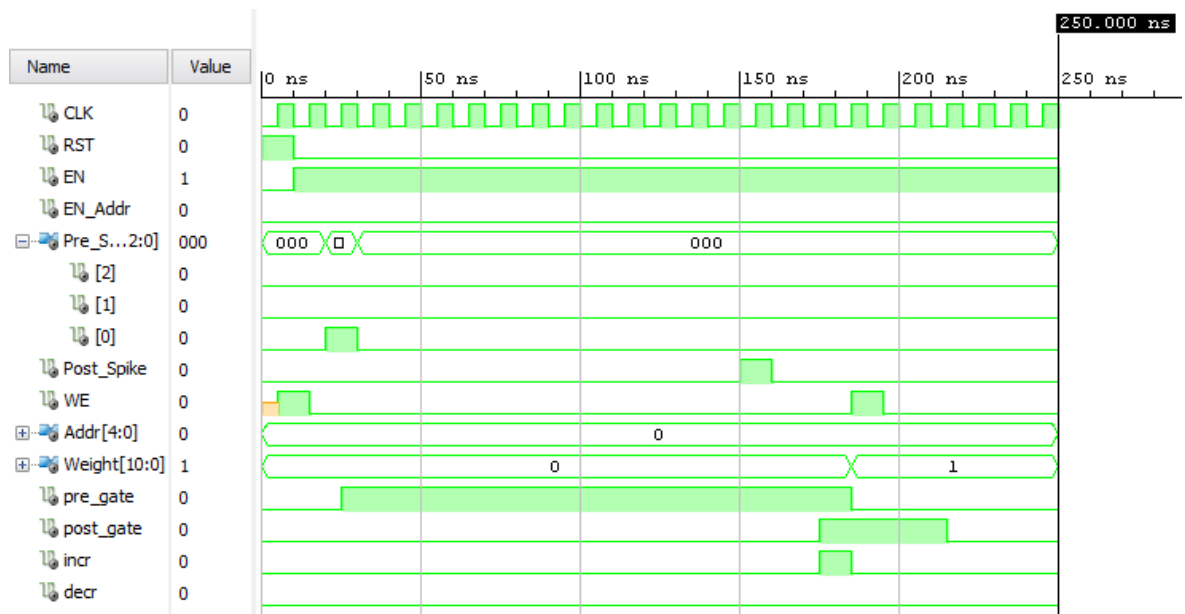
**Figure 5.14.** Timeline of the implemented STDP learning rule [24].

In short, as shown in Figure 5.14, depending on the increasing or decreasing pulse duration the value of the synaptic weight will increase or decrease respectively. That is, if the neuron fires after its preceding neurons generated a spike, the almost synchronous activation of the logic gates *pre\_gate* and *post\_gate* will create a long pulse length for the synaptic weight counter, which increases the weight of the connection for each clock cycle.

Finally, this design allows the regulation of the STDP function by modifying the length of the registers corresponding to the neuron's spikes. Therefore, creating a more or less sensible STDP learning rule to the synchrony of the neural impulses.

### 5.3.2. Simulations

In this section, there are the various simulations of the implemented module made with Vivado in order to show the different behaviors exhibited by the learning module. In the timeline simulations, there are the inputs and outputs of the STDP module along with the *pre\_gate*, *post\_gate*, *incr* and *decr* internal signals for a better understanding of the inner workings of the design.



**Figure 5.15.** Timeline of inputs and outputs of the STDP learning module (Case 1).

First, for the preparation of the timelines a neuron connected to a previous layer of three neurons has been simulated. Therefore, the *Pre\_Spikes* signal is a vector of three binary numbers corresponding to the three synaptic connections and the *Post\_Spike* signal is a single bit that corresponds to the firing of the neuron that is connected to the STDP learning module.

Looking at figure 5.15, around the 20 ns, there is a spike from the first synaptic connection, moments later, the neuron generates a spike that is read through the *Post\_Spike* signal. Because of this, the internal signals *pre\_gate* and *post\_gate* activate respectively and when these two come together in time, the signal *incr* generates a pulse of one clock cycle that is sent to the weight counter. Finally, the value of the synaptic weight, which at the start was 0, is updated to the neuron's RAM with a value of 1 by the write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals.

Continuing with Figure 5.16, the same functionality is exhibited with the difference that the time between the two spikes is lower, therefore the value of the synaptic weight is increased to a greater extent, i.e. from 0 to 4.

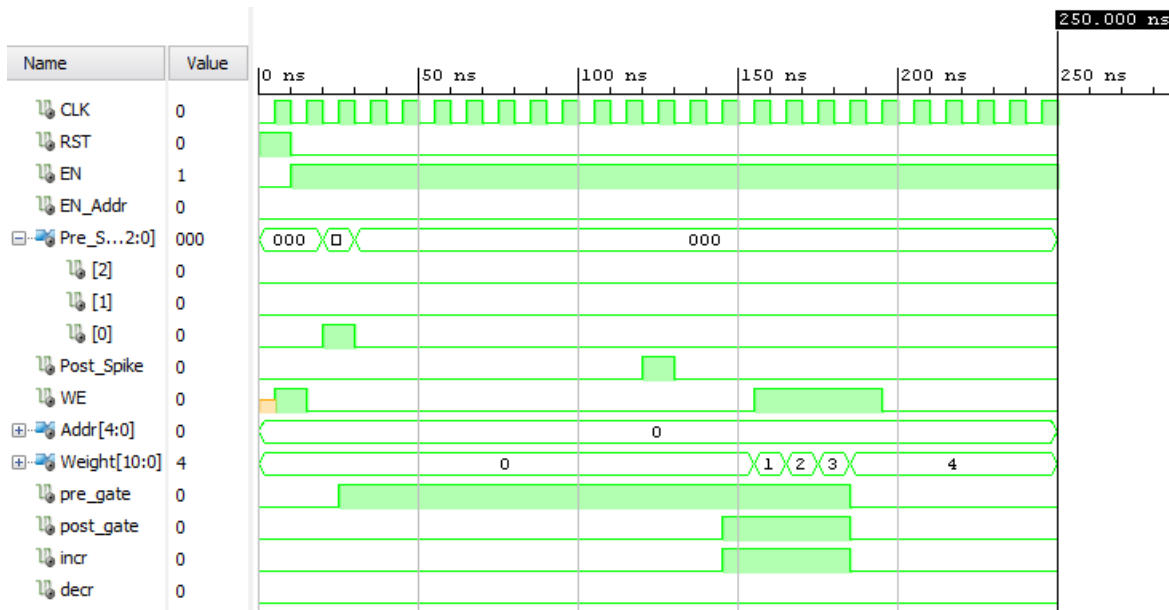


Figure 5.16. Timeline of inputs and outputs of the STDP learning module (Case 2).

Moreover, Figure 5.17 shows the case in which a neuron of the input layer fires after the neuron where the STDP module is connected fired. Hence, due to the  $I/D\ SeI$  block and the digital combinational logic explained above, a decrement pulse is generated, indicating to the synaptic weight counter the decline in the value of the link connectivity from 0 to -2.

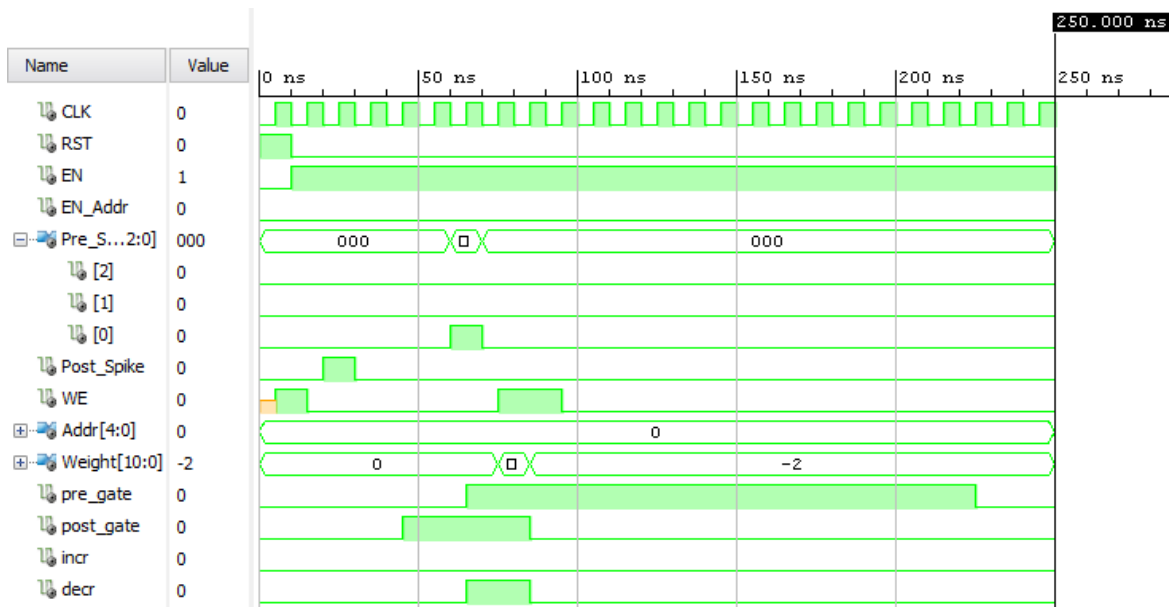
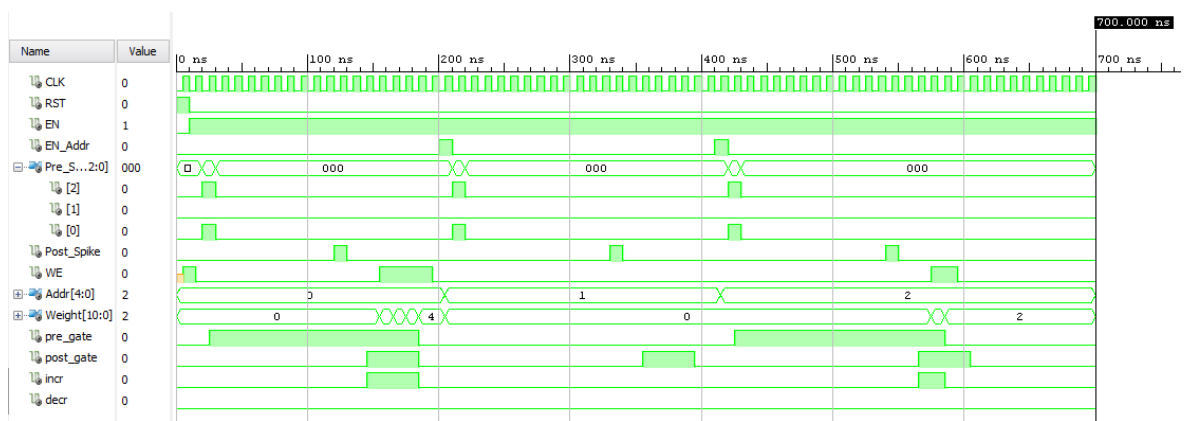


Figure 5.17. Timeline of inputs and outputs of the STDP learning module (Case 3).

Finally, the functionality of the learning module for this small neural network is shown in Figure 5.18. As it can be observed, approximately every 200 ns the neuron receives the same trend of spikes from the neurons which is connected to. Thus, the learning module has the ability to alter one synaptic connection for each time that it receives the spikes.

Firstly, the first synapsis that corresponds to the first bit of the *Pre\_Spikes* vector is updated with a value of 4. Secondly, the *EN\_Addr* signal is activated so the learning module operates for the second synapsis, since there is no spike from it the value is not modified. Finally, the *EN\_Addr* signal is triggered again to apply the STDP learning in the third synapsis, and the weight value is updated to 2 due to the time difference between the two spikes.



**Figure 5.18.** Timeline of inputs and outputs of the STDP learning module (Case 4).

Ultimately, the goal of this module is to sequentially update all the synapses to which the neuron is connected to in order to implement the STDP learning rule. This way, the number of resources needed to implement this system is greatly reduced in comparison to a module that took care of all the synapses of the neural network at once.

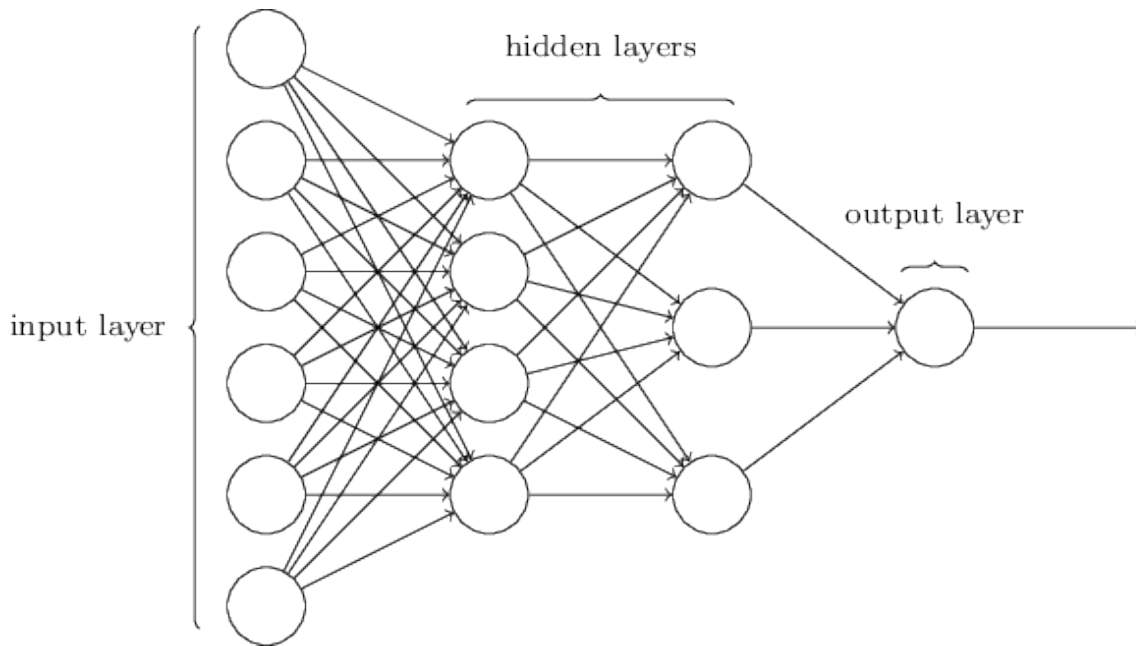
## 5.4. SNN Emulation

As explained in the introduction to this chapter, to give the neural network the highest possible flexibility and scalability a modular system of three main entities or digital modules has been proposed. These are: first, the neuron; second, the AER communication bus; and thirdly, the STDP learning system.

Once explained in the previous sections the internal functionality of each digital block, this section aims to explain how to perform an SNN emulation by interconnecting each one of the proposed digital modules. Meaning, how to connect and replicate these main digital modules in order to obtain a neural network of two, ten or thousands of neurons if the available resources allow it.

### 5.4.1. Design and architecture

To start with, a neural network made up of several layers is shown in figure 5.19. Firstly, an input layer which receives all external stimuli, second, a variable number of hidden layers that are responsible for performing the operations of the neural network, and then thirdly, an output layer in which the neural network broadcasts its stimuli or responses to the outside.

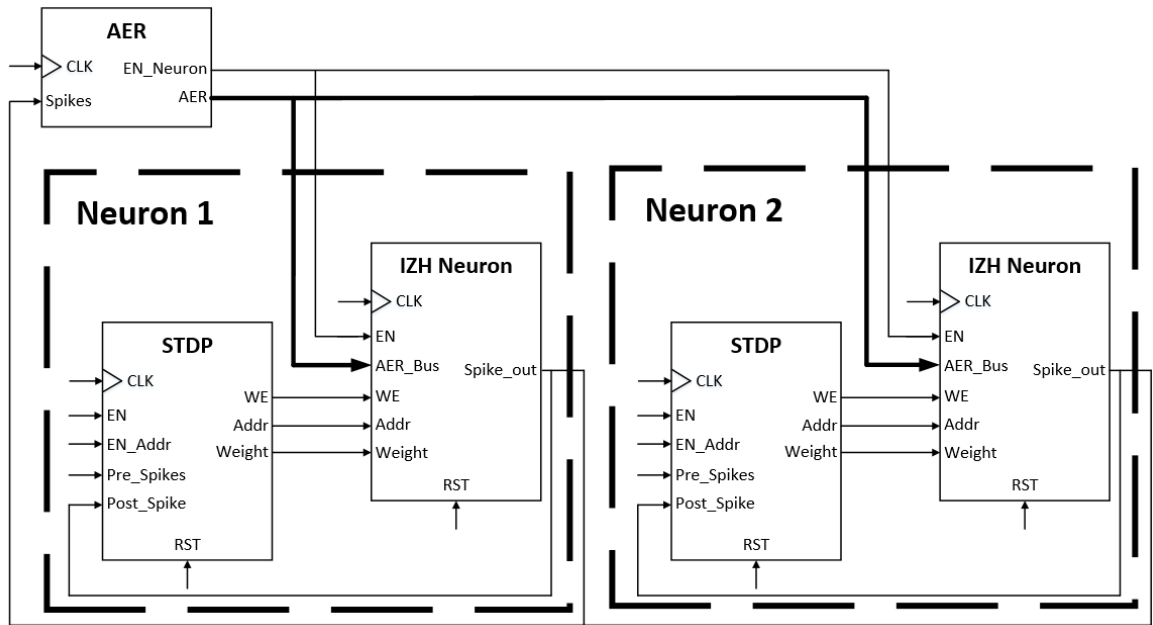


**Figure 5.19.** Architecture of a neural network [25].

To make it simple, Figure 5.20 shows a neural network of two neurons. However, this architecture allows the implementation of as many neurons as needed.

To begin with, what can be seen is that the AER system is responsible for establishing the communication between all the neurons, i.e. read the spikes to translate them into the appropriate address and transmit them to the AER bus. All neurons are connected to the *EN\_Neuron* signal which allow to stop the activity of the neurons, since as explained before, the AER bus can only transmit one address per clock cycle. Moreover, the neurons of the input layer are merely external stimuli, so the introduction of such into the neural network is done by treating the *spikes* vector.

Additionally, each neuron is formed by its digital module and a STDP learning module. The interconnection between these two is established with the write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals that allow to write in the neuron's RAM.



**Figure 5.20.** Interconnection of the different blocks for the emulation of SNN of two or more neurons.

Ultimately, the STDP modules the *EN* and *EN\_Addr* signals along with the *Pre\_Spikes* signal need to be treated with specific digital blocs. These blocks were not included in the diagram of the Figure 5.20, because they depend either from the layer architecture or the functionality of the neural network to emulate. Therefore, it is not possible to provide a standard solution for the many neural networks that this design can implement. However, in the next chapter a SNN for pattern recognition is proposed where the various blocks that allow the configuration of the STDP modules are explained.





## 6. Pattern recognition with a SNN

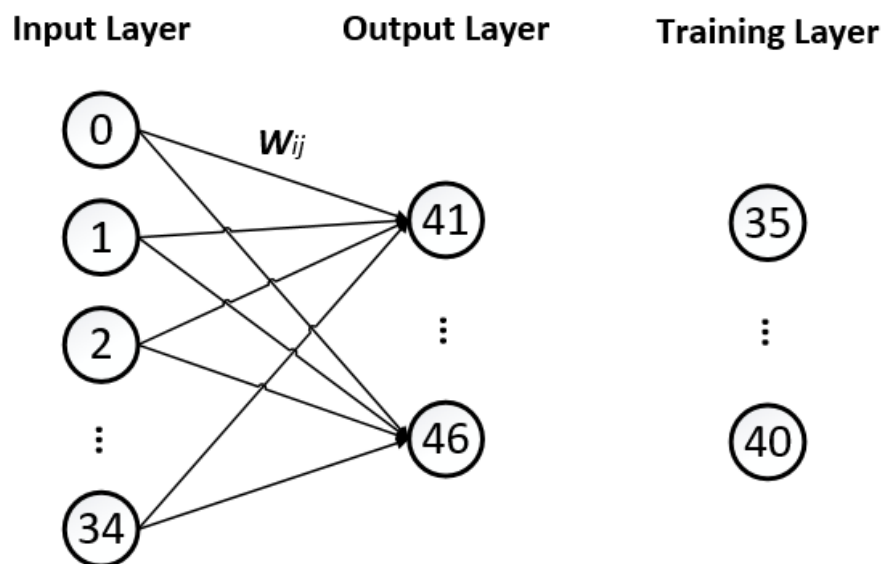
Below is simple spiking neural network that is presented as an example to demonstrate the functionality of the design of the previous chapter.

More specifically, it is a SNN with the functionality to recognize patterns and generate a response based on these types of networks, which are mainly used for image processing, for example to identify the letters of the alphabet or the numbers displayed in an image, recognize the movement of an object in a sequence of images or to detect cars in each lane of a motorway.

This chapter explains in detail the design of this SNN for pattern recognition beginning with the model of the network used, the patterns to recognize and the training method [20]. Finally, several simulations are carried and the final design is implemented into a FPGA to verify its functionality.

### 6.1. Neural network model

The neural network developed in this project as an example for pattern recognition tasks is presented in Figure 6.1. This network is dedicated to recognizing patterns in images of 5x7 pixels, i.e. 35 pixels.



**Figure 6.1.** Representation of the neural network developed for the recognition of six patterns.

It is composed of three layers, the first two, the input layer and output layer, are the ones that form the functional neural network. In addition, there is an extra layer, called training layer, which functionality will be explained later.

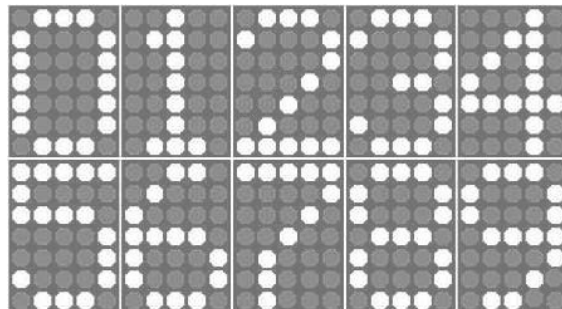
Briefly, the neural network is composed of:

- 35 input neurons corresponding to each pixel of the image.
- 6 output neurons in the output layer corresponding to each pattern. (first neuron corresponding to the first pattern, second neuron to the second pattern...)
- 6 training neurons for each training pattern.

Concisely, the functionality of the neural network is to recognize up to six different patterns depending on the emission of nerve impulses from the corresponding output neuron based on the stimuli of a 35 pixels' image through the input layer.

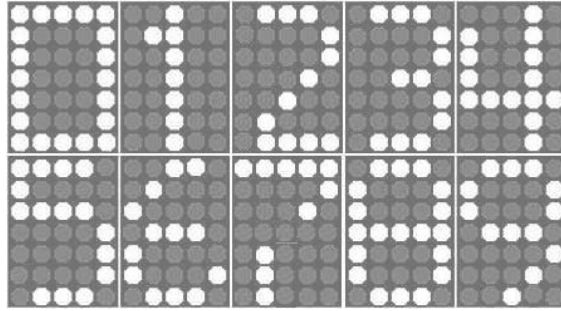
## 6.2. Pattern recognition

The pattern recognition task consists in differentiate up to six numbers from the digits presented in Figure 6.2. As seen in the picture, each number is represented by a 5x7 pixel image, therefore, can be used to stimulate the input layer of the neural network. Furthermore, because each pixel has a binary representation, black or white, these constitute the existence or absence of nerve stimulation of the input neurons.



**Figure 6.2.** Input patterns for the neural network stimulation [26].

In short, how to encode the digit corresponding to the zero digit in order to introduce it into the first input layer of the neural network is as follows. Firstly, the first row of five pixels corresponds to the zero to fourth neurons which its vector of spikes is: "01110". Secondly, the second row corresponds to the fifth to ninth neurons which its vector of spikes is: "10001". Finally, following this process a vector with a length of 35 bits is obtained and can be used to stimulate all the neurons of the input layer.



**Figure 6.3.** Input patterns with noise for the neural network stimulation.

In addition, some noise has been added to the digits presented in Figure 6.2 in order to prove the functionality of the neural network. Therefore, the neural networks should still be able to recognize the corresponding numbers using the digits from Figure 6.3 as stimuli after its training.

### 6.3. Training method

The training method [20] is based on the STDP learning rule, which as explained above, modifies the weights of the synaptic connections depending on the synchrony of firing of the neurons. That is, the synapses that contribute to the generation of an output spike of the neural network should be enhanced, while those not contribution to the generation of an output spikes must be weakened.

Leveraging the features of the STDP rule, a training layer has been implemented into the neural network that allows to train for each output neuron which pattern must recognize. Meaning, to what digit of Figure 6.2 the neurons needs to generate spikes and remain at rest for others digits.

Initially, the first layer of the neural network needs to be stimulated with a certain frequency. This period of time should be enough for all the neurons to return to their initial state of rest to achieve a reliable and robust training.

Leading on from this, all the synapses of the neural network between the input and output layer must be initialized to zero. Thus, for any stimulus in the neurons of the input layer, the output neurons cannot generate any spikes and must stay in their resting state.

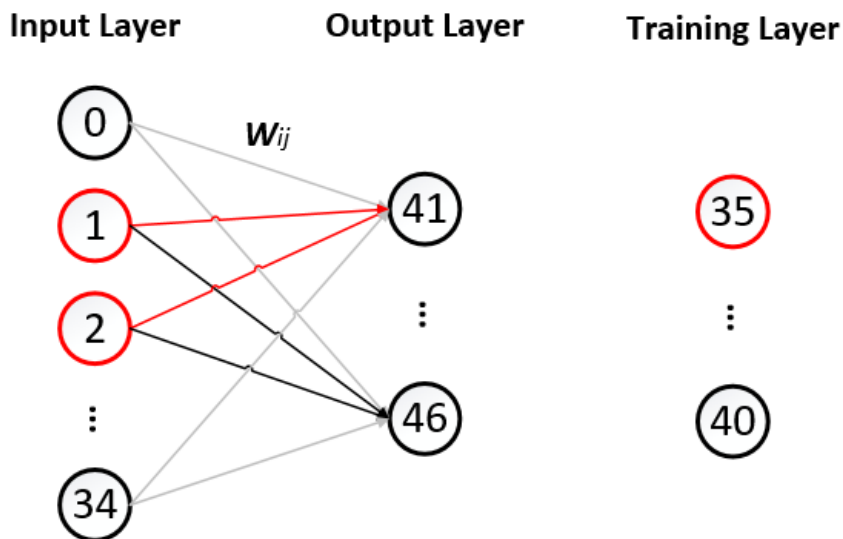
Lastly, the STDP training module of each output neuron is connected to his counterpart training neuron. Hence, the training module reads the spikes of the output neuron layer along with the firings from the training neurons.

Only when these three conditions are met the training phase can begin. Firstly, the pattern to train needs to be introduced into the network and moments later the training neuron corresponding to the

output neuron, which is wanted for the recognition of the pattern, has to be fired manually. Due to the firing of the training neuron with the spikes of the input neurons, the STDP training module will potentiate the connections from the active input neurons to the output neuron.

Alternatively, the other training neurons have to be fired manually moments before the stimuli to the input layer is introduced. Thus, the firing of these training neurons with the spikes of the input neurons promotes a decrease in the connectivity of the synapses that connect the input neurons with the output neurons corresponding to these training neurons.

Figure 6.4 shows the modification of the synaptic weights based on the firing of neurons. Therefore, in the case when the training pattern fires neurons 1 and 2 altogether with the training neuron 35, the connectivity of the synapses between neurons 1 and 2 and the output neuron 41 is increased (red). On the contrary, the connectivity of the synapses between neurons 1 and 2 and the rest of the output neurons is decreased. Moreover, the synaptic weights from the neurons 0 and 34 are not modified because they do not take part in the learning pattern input.



**Figure 6.4.** Modification of the synaptic weights based on the STDP learning rule.

Therefore, as long as the input layer is being stimulated along with the training neurons, the connections that contribute to the firing of the corresponding output neuron are potentiated. The synapses that could generate undesired spiking are decreased and those that do not intervene in the stimulation of the input layer are not modified.

Finally, an important variable to take into account for the proper learning of the neural network in the application of this method is the time period in which the STDP is applied. For excessively long time periods all the output neurons can end up firing for any pattern, while for short periods some or all the

output neurons can end up at rest for any pattern. The exact time period depends on the number of input neurons and the training patterns themselves, therefore the challenge lies in finding the appropriate period of time in order to achieve that each neuron fires for its trained pattern.

## 6.4. FPGA implementation

The implementation of the neural network for pattern recognition has been made on a FPGA model Nexys 4 Artix-7. This is a development platform ready for the implementation of digital circuits for industrial applications, plus it incorporates USB, ethernet and other ports to accommodate designs ranging from introductory combinational circuits to powerful embedded microprocessors.

This section explains in detail the implementation of the neural network for pattern recognition, and how the different elements of input and output communication of the board have been used for the introduction of different stimuli to the neural network and view its response.

### 6.4.1. Inputs and outputs

In the implementation of the neural network for pattern recognition, it requires sixteen switches, three buttons and six logic outputs, which have been used from the FPGA.

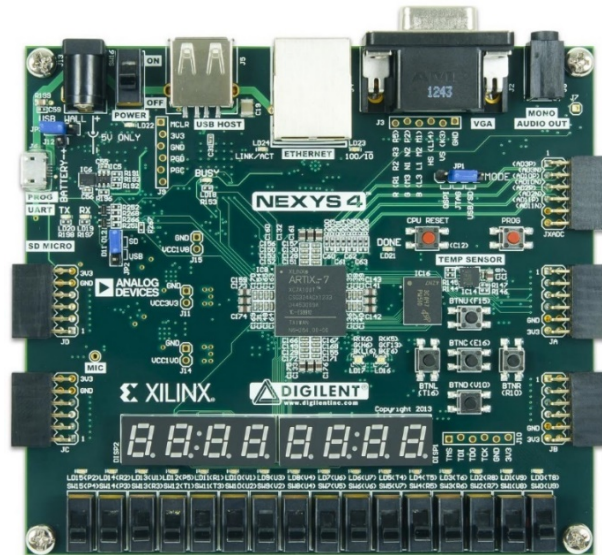


Figure 6.5. Nexys 4 development board [21].

The first ten switches (SW0 to SW9) allow to select the stimulus for the input layer of the neural network, i.e. any of the ten digits shown in figure 6.2. Moreover, the six remaining switches (SW10 to SW15) allow to select the training neuron to which the introduced pattern wants to be trained.

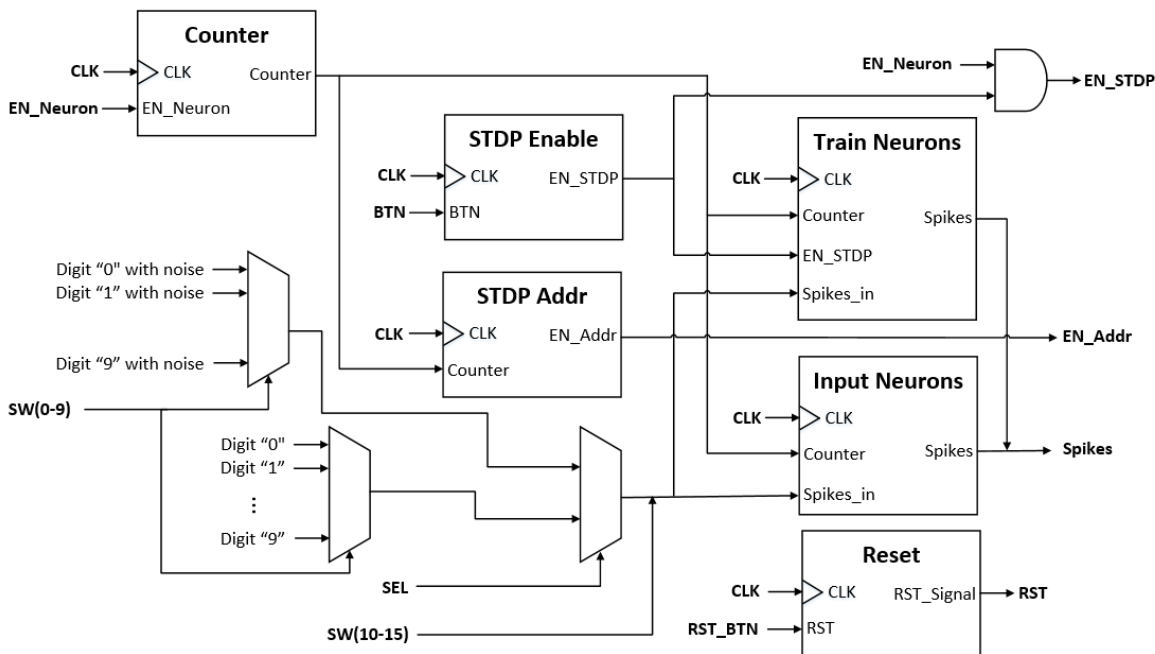
A button is then sending a reset signal for the whole design and a second button to send a training pulse of about 200 input stimuli for the neural network. Also, by pulsing the reset and training buttons together allows to reset all the synaptic weights to zero. Moreover, a third button is used for selecting the digits with noise presented in Figure 6.3 as stimuli for the neural network.

Finally, six analog outputs that allow the reading of the spikes generated by the output neurons of the neural network with an oscilloscope.

#### 6.4.2. Design and architecture

Regarding the implementation of the design, in Figure 6.6 all the necessary elements to introduce stimuli to the neural network with a certain frequency, for the treatment of training neurons and the signals for the correct configuration of the STDP module are shown.

Firstly, the entire structure is governed by a counter that, as mentioned above, provides a period of 150 clock cycles to introduce the stimuli to the neural network. This period is sufficient for all the output neurons to return at their resting state.



**Figure 6.6.** Block diagram for the treatment of the input stimuli and training of the neural network.

Secondly, to select the input stimuli two multiplexers have been implemented which generate a 35-bit vector at its output. Together with a third multiplexer controlled by the SEL button, it is possible to select between the normal digits and the digits with noise. In addition, the six bits from the training neurons are concatenated to this vector to indicate which neurons have to fire with the selected image.

Next, the *Input Neurons* block is responsible for transmitting to the AER system the 35-bit vector corresponding to the pixels of the image when the value of the counter is zero. Moreover, the *Train Neurons* block transmits to the AER system the 6-bit vector of the training neurons in the corresponding time instants in order to strengthen or weaken the synapses of the network.

Thirdly, the *STDP Enable* block translate a keystroke of the BTN button from the FPGA board to perform a training pulse for approximately 200 input stimuli. In addition, the *STDP Addr* block is responsible for changing the address of the STDP modules for each output neuron.

To summarize, while Figure 6.7 represents a block diagram necessary for the treatment of the input and training stimuli, the implementation of the neural network where the six layer output neurons (from 41 to 46) are emulated is shown in figure 6.6. Also, the *CLK*, *RST*, *EN\_Neuron*, *Spikes*, *EN\_STDP* and *EN\_Addr* signals that are interconnected between the two diagrams can be seen.

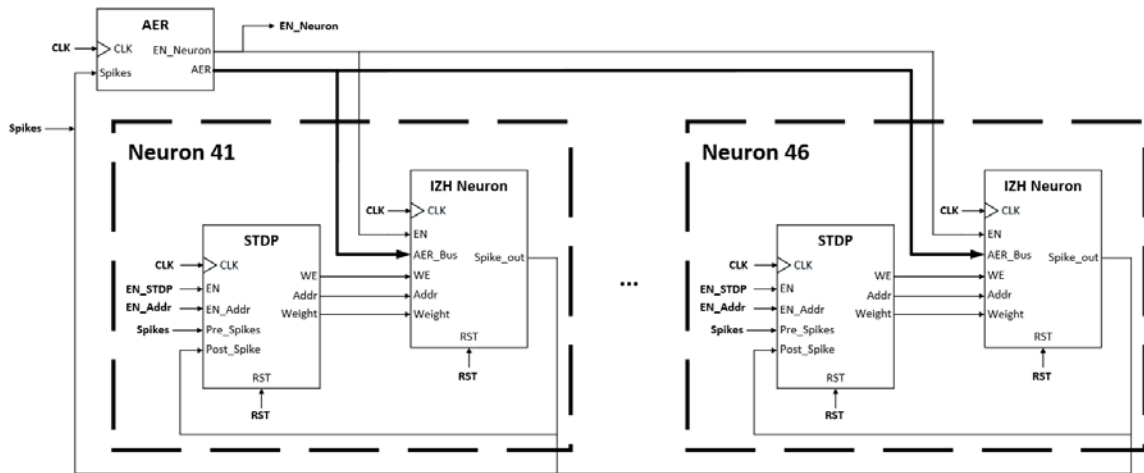


Figure 6.7. Block diagram of the neural network.

### 6.4.3. Simulations

Next are the different simulations of the neural network for pattern recognition implementation made with Vivado in order to demonstrate its functionality.

Figure 6.8 illustrates a complete simulation of the functionality of the SNN. Firstly, patterns from digits 0 to 5 have been taught to neurons 41 to 46 respectively. Therefore, as it can be seen through the pulses of the *EN\_STDP* signal, six learning phases are conducted, one for each output neuron of the SNN.



Figure 6.8. Complete simulation of the SNN for pattern recognition digits 0 to 5.

Therefore, in the first phase of the training where the image of digit 0 is selected along with the training neuron that corresponds to the output neuron 41, it can be seen that the output neuron does not fire, but as long as the training advances in time, the synapses that contribute to its firing are modified and the output neuron 41 ends up learning to generate spikes for the selected pattern.

Then, in the third phase of the training the image of digit 2 and the training neuron of the output neuron 43 is selected. In this case when the third phase is just starting the output neuron 41 is firing, however, as the training phase advances stops firing and it is only the output neuron 43 that generates spikes for the selected pattern.

In the following phases of the training the process is the same, until the training is complete in about 9 ms. Then, the correct pattern recognition of the neural network is tested and for one millisecond the images from digits 0 to 5 are introduced and the corresponding output neurons fire only for their respective trained digit.

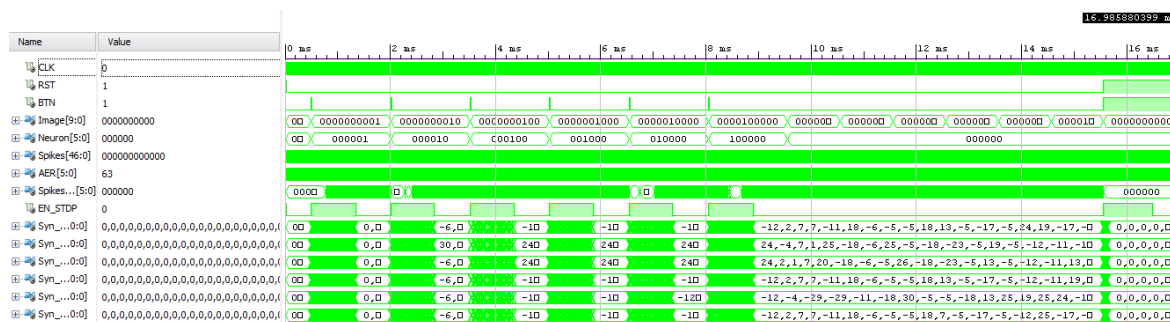


Figure 6.9. Modification of the synaptic weights of the SNN.



The modification of the synaptic weights of the SNN are shown in Figure 6.9. As it can be observed this are only modified when the *EN\_STDP* pulse is triggered, which means they are only modified during the different training phases. Also, as explained before when triggering the reset signal along with the training button all the synaptic weights values are reset to zero, getting the SNN ready to start another training.

Other behaviors described in the previous sections on the input stimulus and training neurons can be seen by looking at Figure 6.10, which is the same simulation performed on 6.8 and 6.9. Firstly, the position of the marker corresponds to the introduction of stimuli corresponding to the image of the 0 digit, from bit 0 to 34 in the *Spikes* vector. A few clock cycles later the training neuron 35 fires in order to train the output neuron 41. On the contrary, the rest of the training neurons fire before the stimulus of the neural network in order to weaken the corresponding synapses. After some more input stimuli, neuron 41 ends up learning and firing in its own as shown in Figure 6.11.

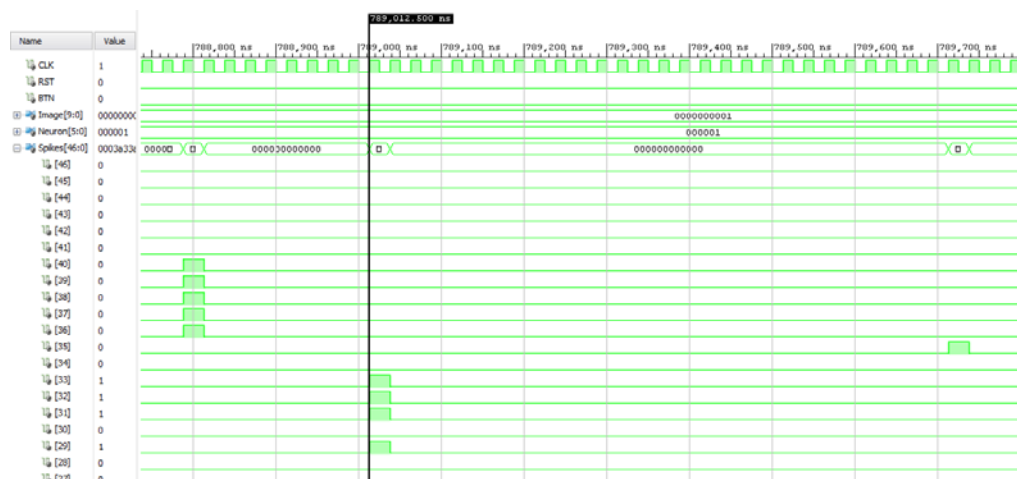


Figure 6.10. Firing of input, training and output neurons of the SNN.

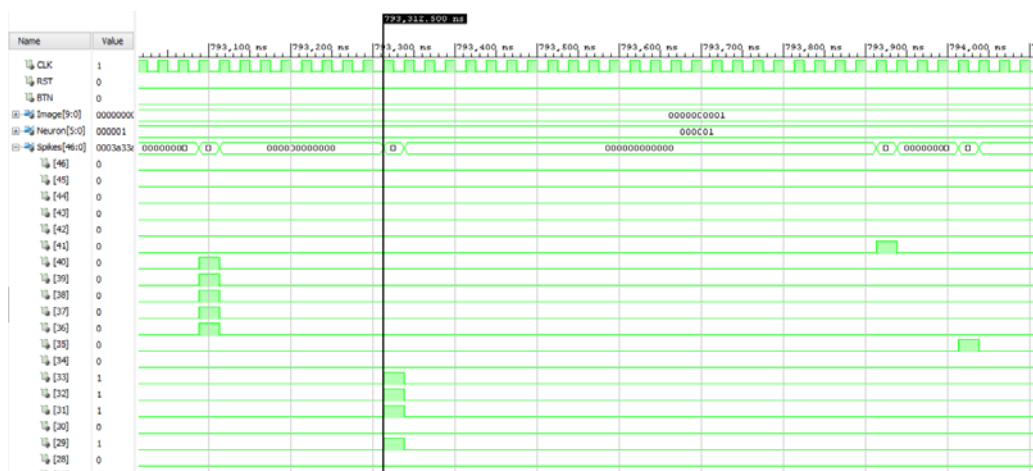
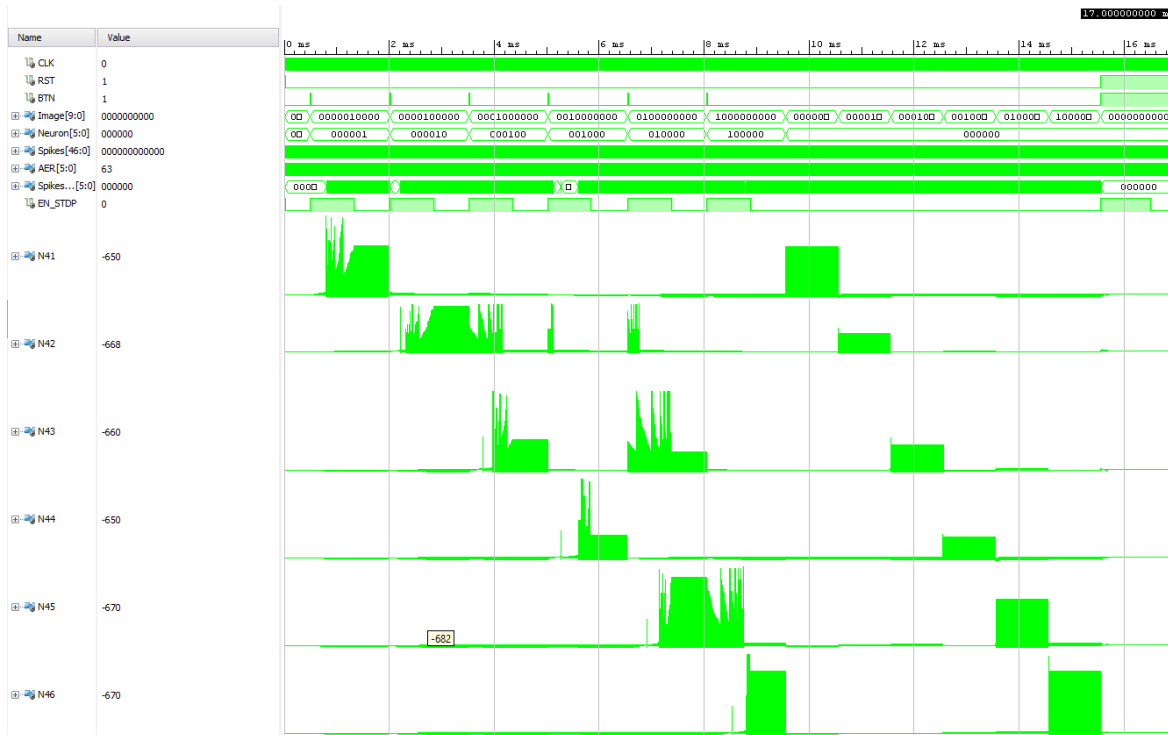


Figure 6.11. Firing of input, training and output neurons of the SNN.

Digits from 0 to 5 are not the only patterns that the SNN can be trained to recognize. In Figure 6.12 the output neurons 41 to 46 have been trained with the patterns of digits 4 to 9 respectively, as it can be seen with the values of the *Image* vector (“0000000001” corresponds to number 0, “0000000010” to number 1, until “1000000000” for number 9).



**Figure 6.12.** Complete simulation of the SNN for pattern recognition digits 4 to 9.

Moreover, the influence of the training phase time can be observed in Figures 6.13 and 6.14. Following on from the same learning for pattern recognition of digits 4 to 9, in Figure 6.13 not all the output neurons end up generating spikes due to an insufficient training time. However, in Figure 6.14 due to an excessive training time the output neuron corresponding to the digit 8 fires when a digit 6 and 9 are introduced as stimuli to the SNN.

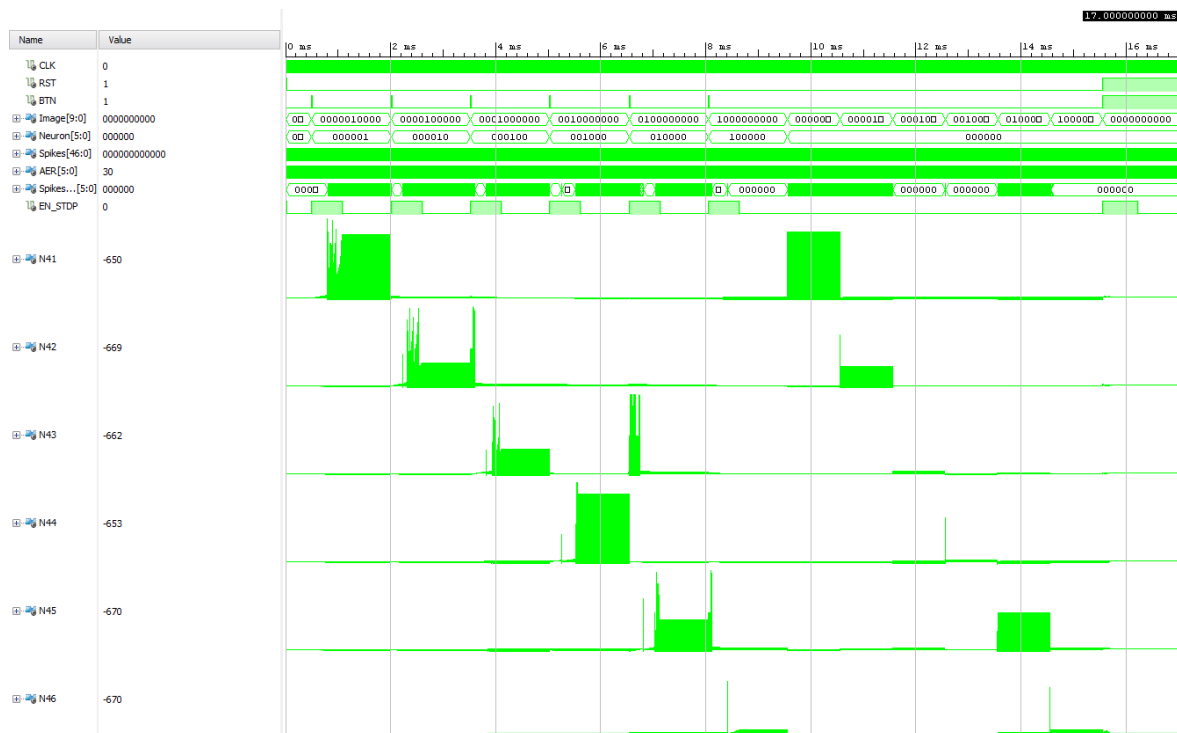


Figure 6.13. Complete simulation of the SNN for pattern recognition digits 4 to 9 (insufficient training time).

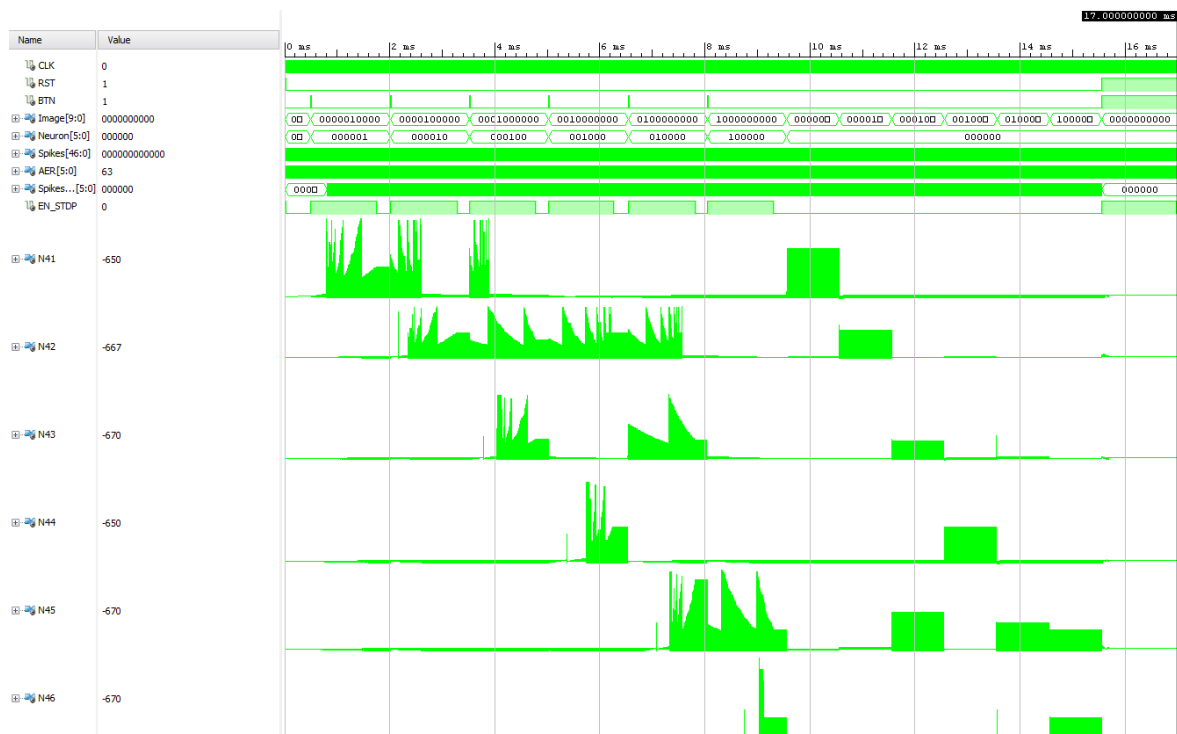
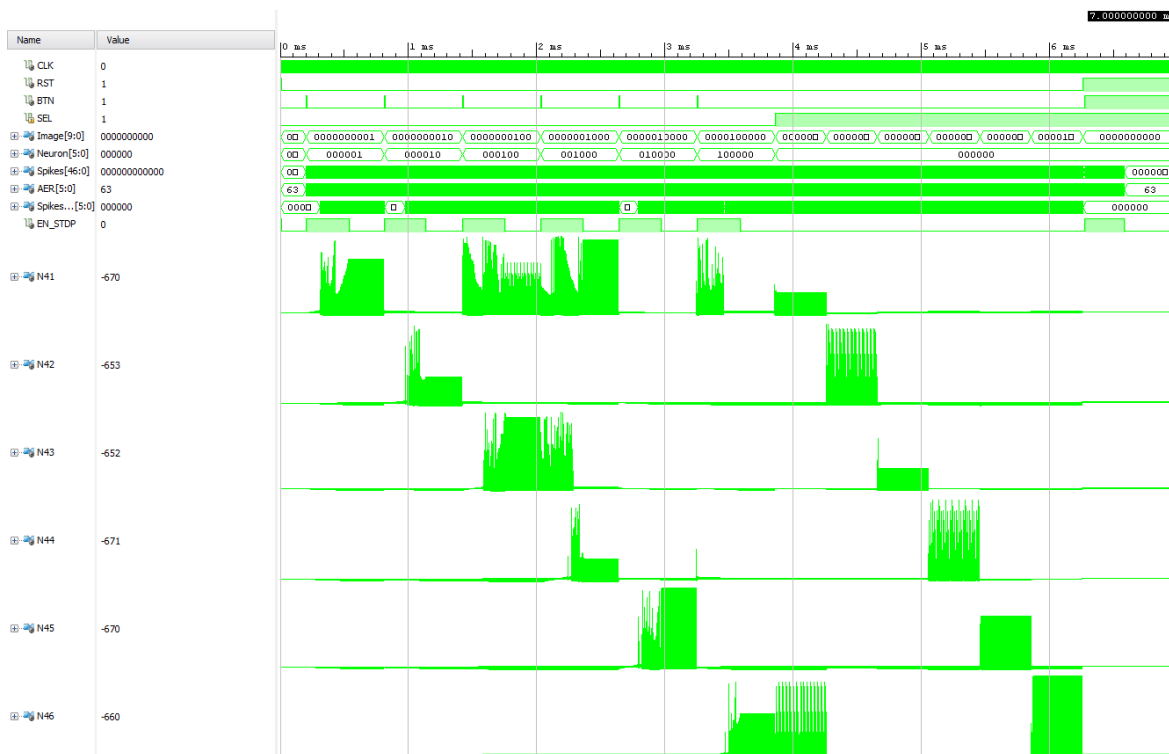


Figure 6.14. Complete simulation of the SNN for pattern recognition digits 4 to 9 (excessive training time).

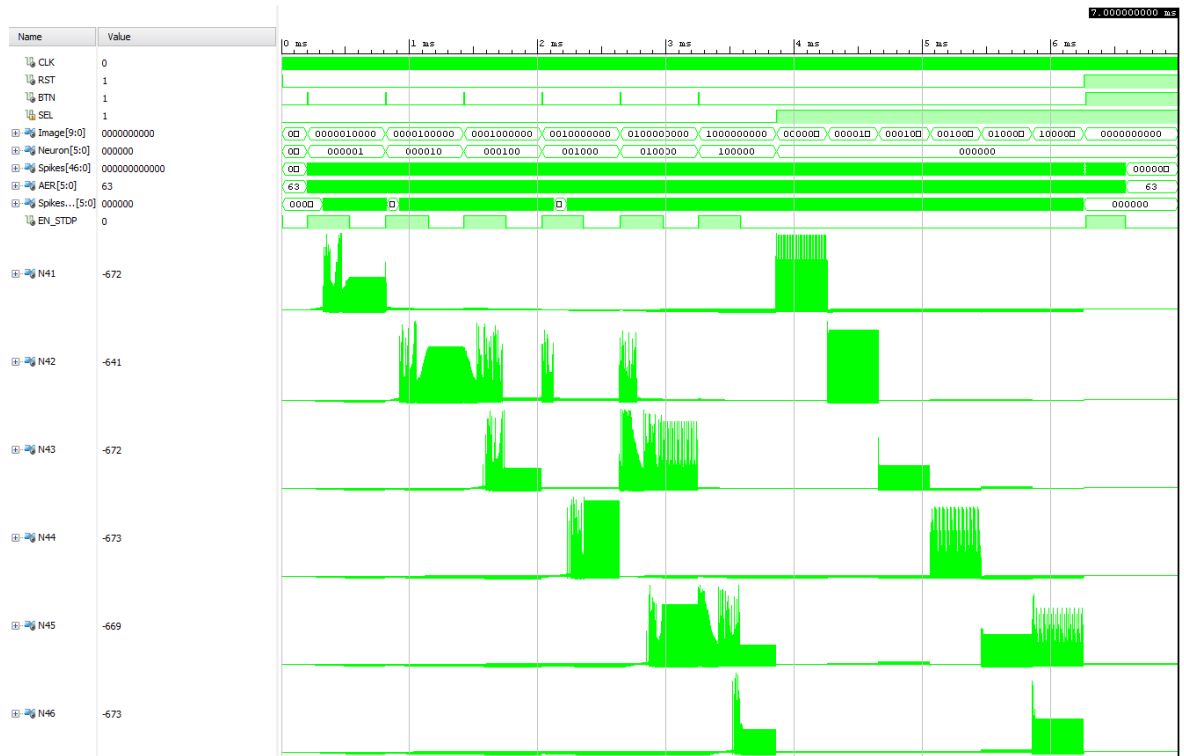
Moreover, the influence of the training phase time can be observed in Figures 6.13 and 6.14. Following on from the same learning for pattern recognition of digits 4 to 9, in Figure 6.13 not all the output neurons end up generating spikes due to an insufficient training time. However, in Figure 6.14 due to an excessive training time the output neuron corresponding to the digit 8 fires when a digit 6 and 9 are introduced as stimuli to the SNN.

Additionally, Figure 6.14 and Figure 6.15 represent a complete simulation of the SNN for pattern recognition of digits 0 to 5 and digits 4 to 9 respectively, as shown before. Nevertheless, in these simulations the difference resides in the stimuli used after the training phases have concluded. Instead of using the same digits for the whole simulation, after the training phase the digits with noise from Figure 6.3 are introduced into the SNN thanks to the activation of the signal *SEL*. Regardless of the difference in several pixels due to the noise of the images, since the main shape of the digits remains unchanged the neural network is capable of recognizing the correct numbers. Therefore, it proves the functionality of recognizing similar patterns.



**Figure 6.15.** Complete simulation of the SNN for pattern recognition digits 0 to 5.

Having said that, in Figure 6.15, the neuron corresponding to digit 8 fires for either digits 8 and 9 with noise, which is something to expect since the main shape of this two numbers are quite similar and therefore the SNN recognizes them as the same.



**Figure 6.16.** Complete simulation of the SNN for pattern recognition digits 4 to 9.

Finally, the simulations which were performed enabled a corroboration of what has previously been said. For long training time periods a less restrictive training is performed which, in excess, can trigger the firing of all output neurons for any given pattern. On the contrary, for short training time periods a more restrictive training is performed which cannot be enough to trigger the firing of the output neurons.

Succinctly, although the SNN has been able to learn different patterns (digits 0 to 5 and 4 to 9) with the same training time, it is not required to be similar for other pattern combinations because, at the end, each pattern combination has its appropriate training time period in order to trigger the firing of the corresponding output neuron.

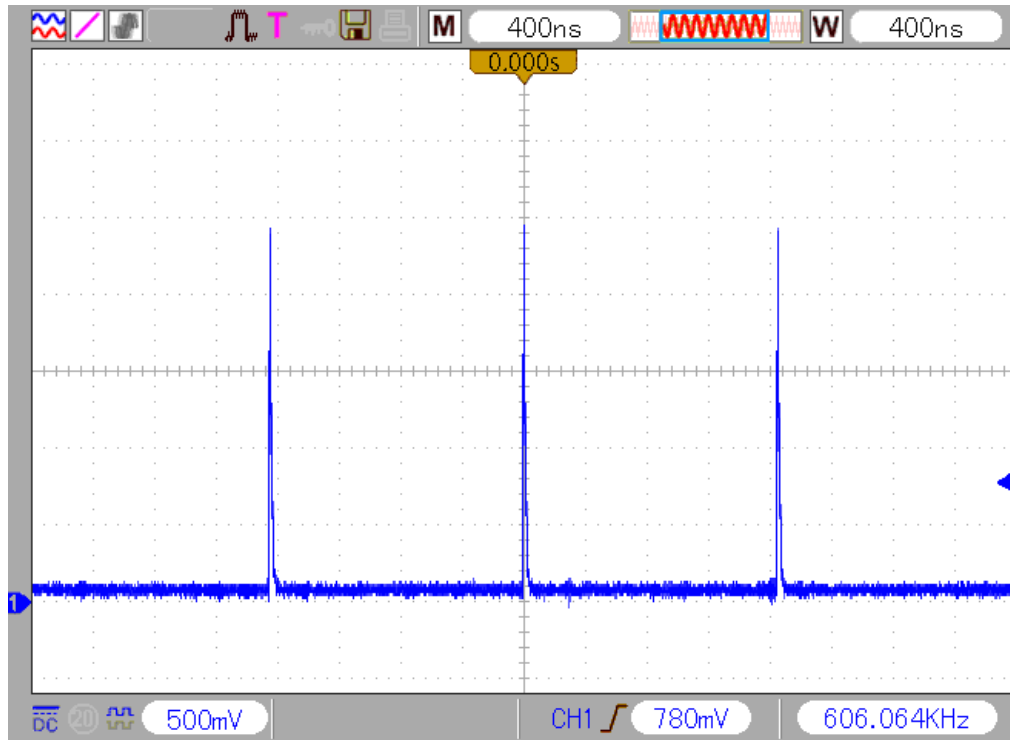
#### 6.4.4. Experimental results

The results of the physical implementation of the design into the FPGA are presented in this section. As said before the FPGA used to verify the functionality of the digital system is a Nexys 4 Artix-7 model at 100 MHz.

Initially, in order to read the spikes from all the output neurons from the FPGA analog outputs two oscilloscopes has been used. Firstly, the AD Instruments DS2202A with a 200 MHz bandwidth that allowed to view the form of a single spike. Secondly, the RIGOL DS1102D, a digital oscilloscope of 100

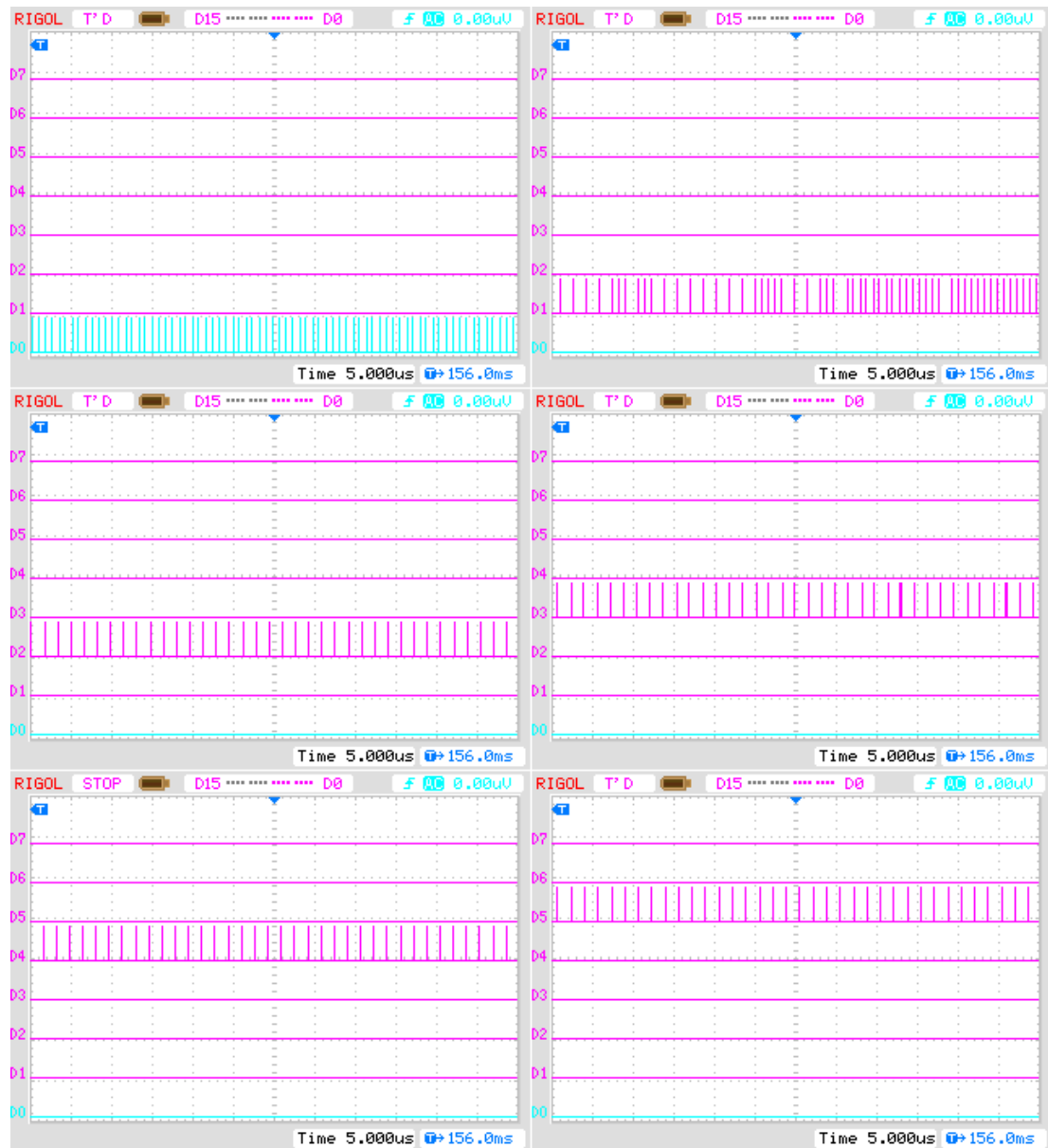
MHz bandwidth with a logic analyzer channel that can read up to 16 signals, which has been used to read the spikes of the six output neurons of the neural network at the same time.

Observing the 6.17 picture, taken with the DS2202A oscilloscope, there are three spikes of an output neuron of the SNN with a frequency of about 625 KHz. This is the time period between the input of the pattern, so the SNN has time to generate a response to the introduced pattern and all neurons have enough time to return to their initial state of rest.



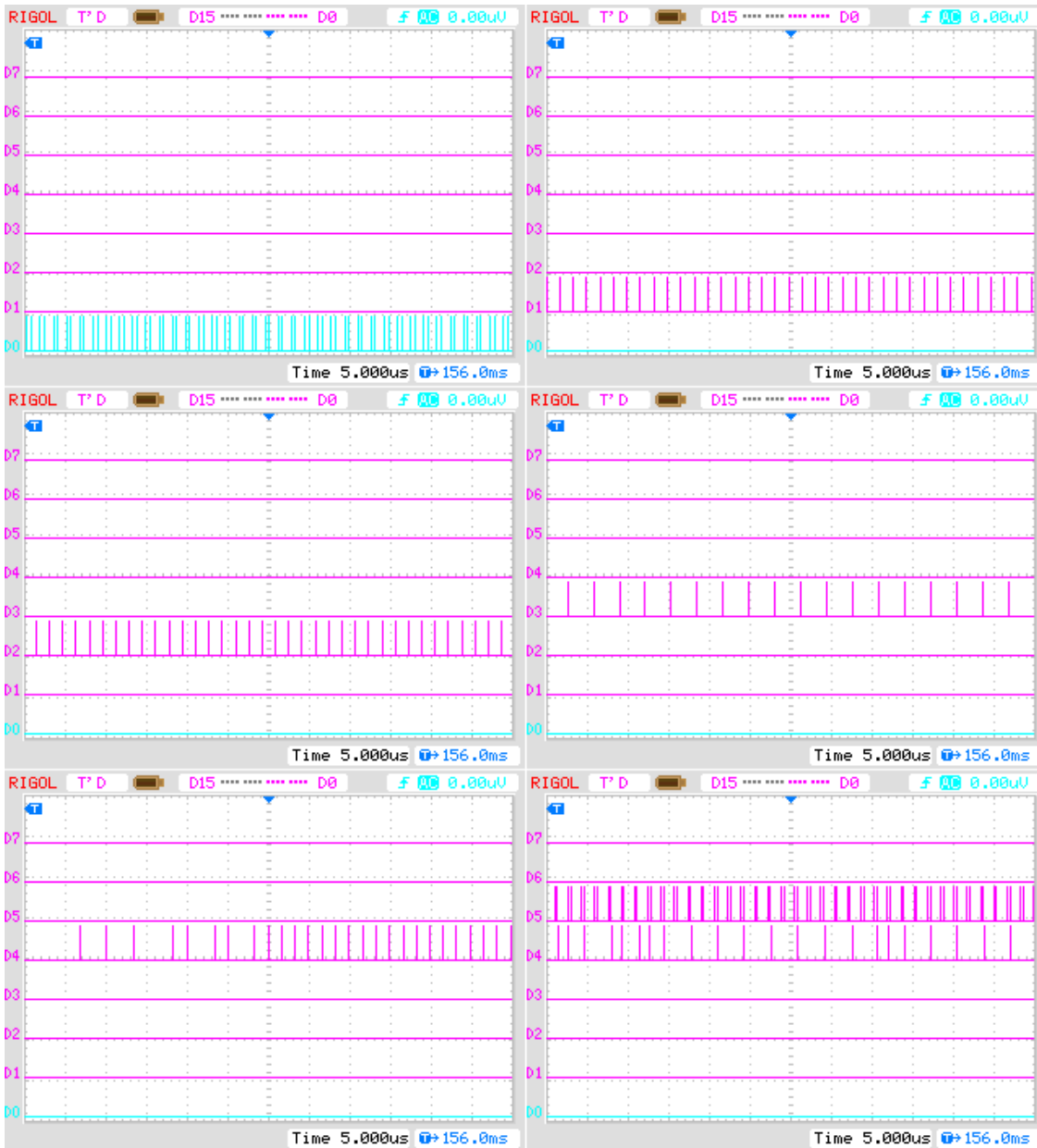
**Figure 6.17.** Spikes from output neuron 0.

Next, the SNN has been trained for the pattern recognition of digits 0 to 5, as did before with the simulations. The results are shown in figure 6.18, with six captures made with the DS1102D digital oscilloscope, one for each input pattern. Therefore, first image corresponds to the stimuli of the SNN with digit 0, second image with digit 1, until sixth image with digit 5. As it can be observed, the SNN exhibits the expected behavior recognizing the input pattern by generating spikes with the corresponding output neuron.



**Figure 6.18.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 0 to 5 respectively (Digit 0 as input pattern for the first image, digit 1 for second image...).

Moreover, in Figure 6.19 the SNN has been trained for the pattern recognition of digits 4 to 9, as did before with the simulations. Therefore, first image corresponds to the stimuli of the SNN with digit 4, second image with digit 5, until sixth image with digit 9. As it can be observed, the SNN exhibits the expected behavior recognizing the input pattern by generating spikes with the corresponding output neuron. Nevertheless, for input pattern of digit 9 the output neuron corresponding to digit 8 recognition also generates spikes. This shows one of the fundamental properties of the SNN, which are the recognition of similar patterns. Since the shape of digit 9 is close to the shape of digit 8 the SNN mistakes the 9th digit shape as digit 8.



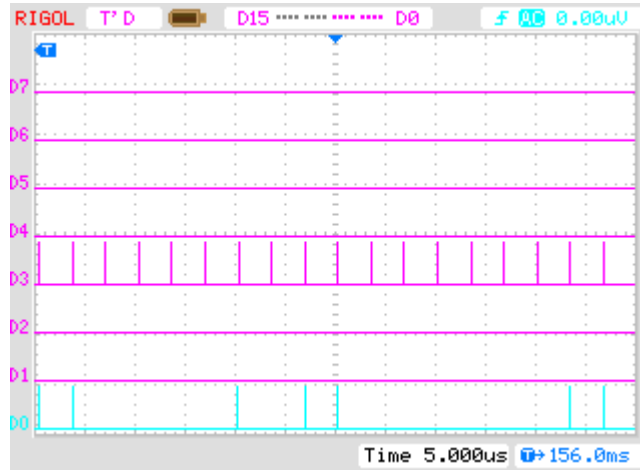
**Figure 6.19.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 4 to 9 respectively (Digit 4 as input pattern for the first image, digit 5 for second image...).

As explained above, the challenge resides on finding the appropriate training time period in order to trigger the firing of the corresponding output neuron. Therefore, to improve the training in the digit 4 to 9 recognition the train time period needs to be adjusted or output neuron 5 could send inhibition pulses to the output neuron 8 in order to prevent its firing.

In order to test out the property of the SNN to recognize similar patterns, the inputs of digits 6 to 9 has been introduced to the SNN presented in Figure 6.18 and the inputs of digits 0 to 3 into the SNN presented in Figure 6.19.

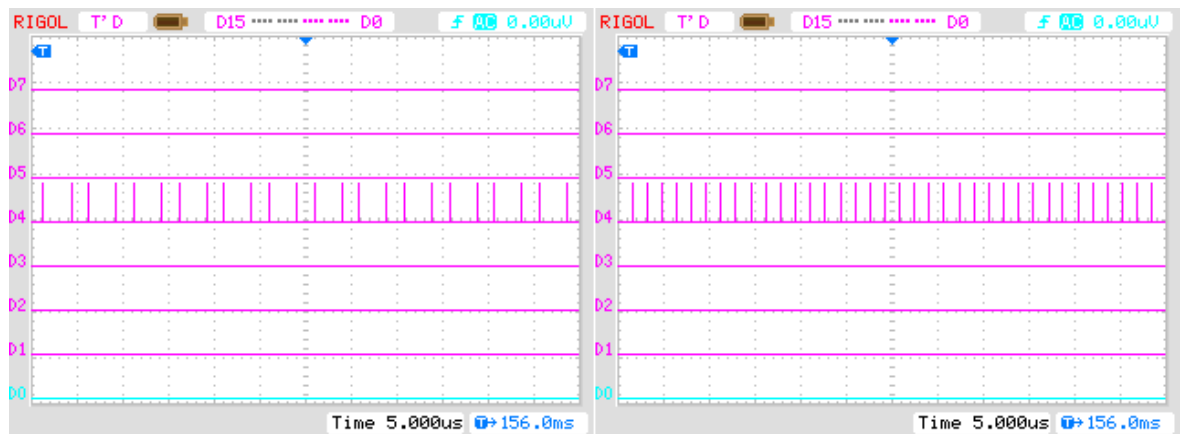


The response of the SNN of Figure 6.18 has been negative to the inputs of digits 6, 7 and 9 except for digit 8. Therefore, when introducing the stimuli to the SNN corresponding to digit 8 the output neuron 0 and 3 start to generate spikes as shown in Figure 6.20. This behavior is expected since digit 8 shape is similar to digit 0 and 3 shapes.



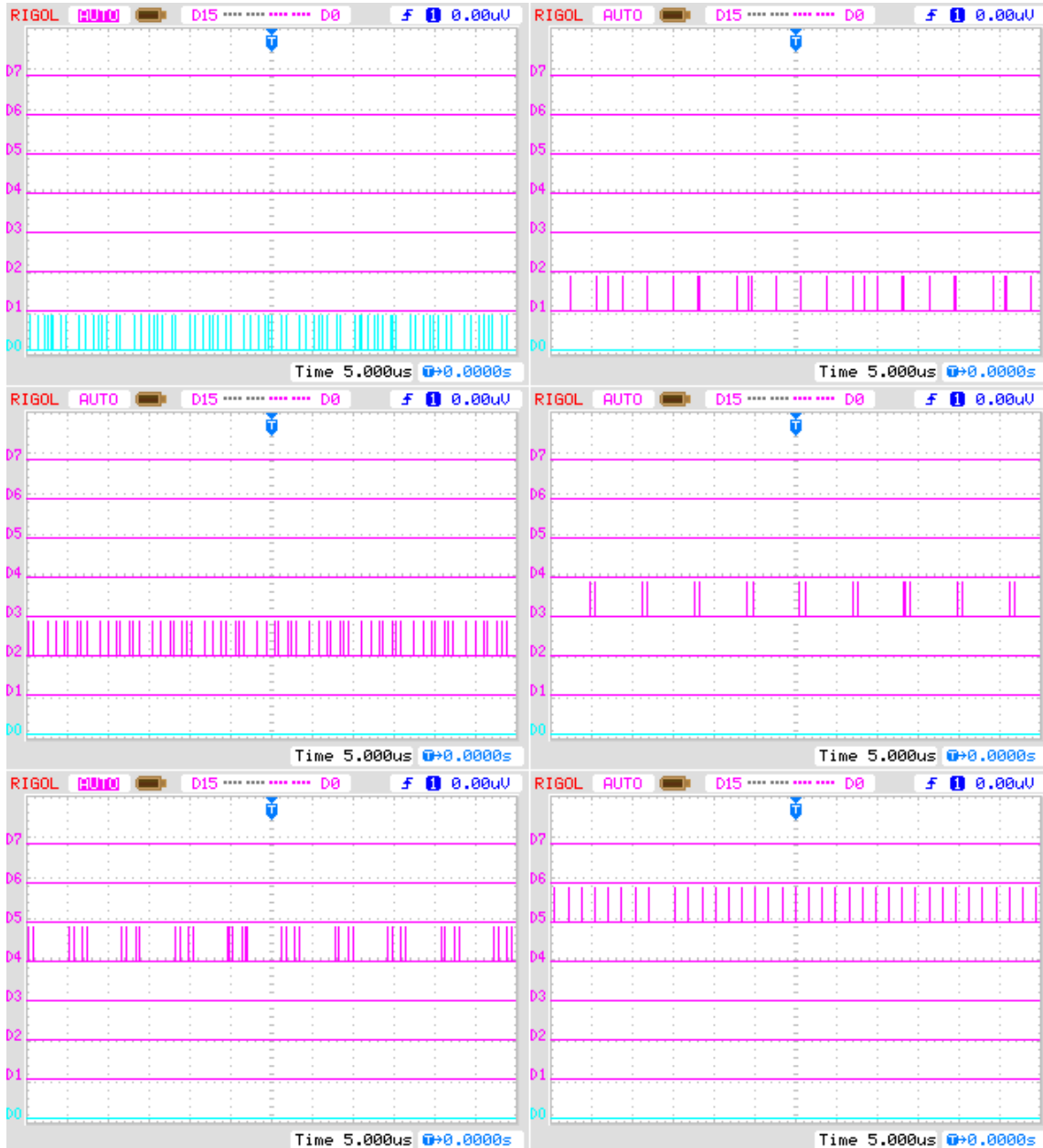
**Figure 6.20.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 0 to 5 respectively (Digit 8 as input pattern).

On the other hand, the response of the SNN of figure 6.19 has been null to the inputs of digits 1 and 2, but for inputs of digits 0 and 3 the output neuron corresponding to digit 8 detection starts firing. Therefore, exhibiting the same behavior explained above but just the other way around as shown in Figure 6.21.

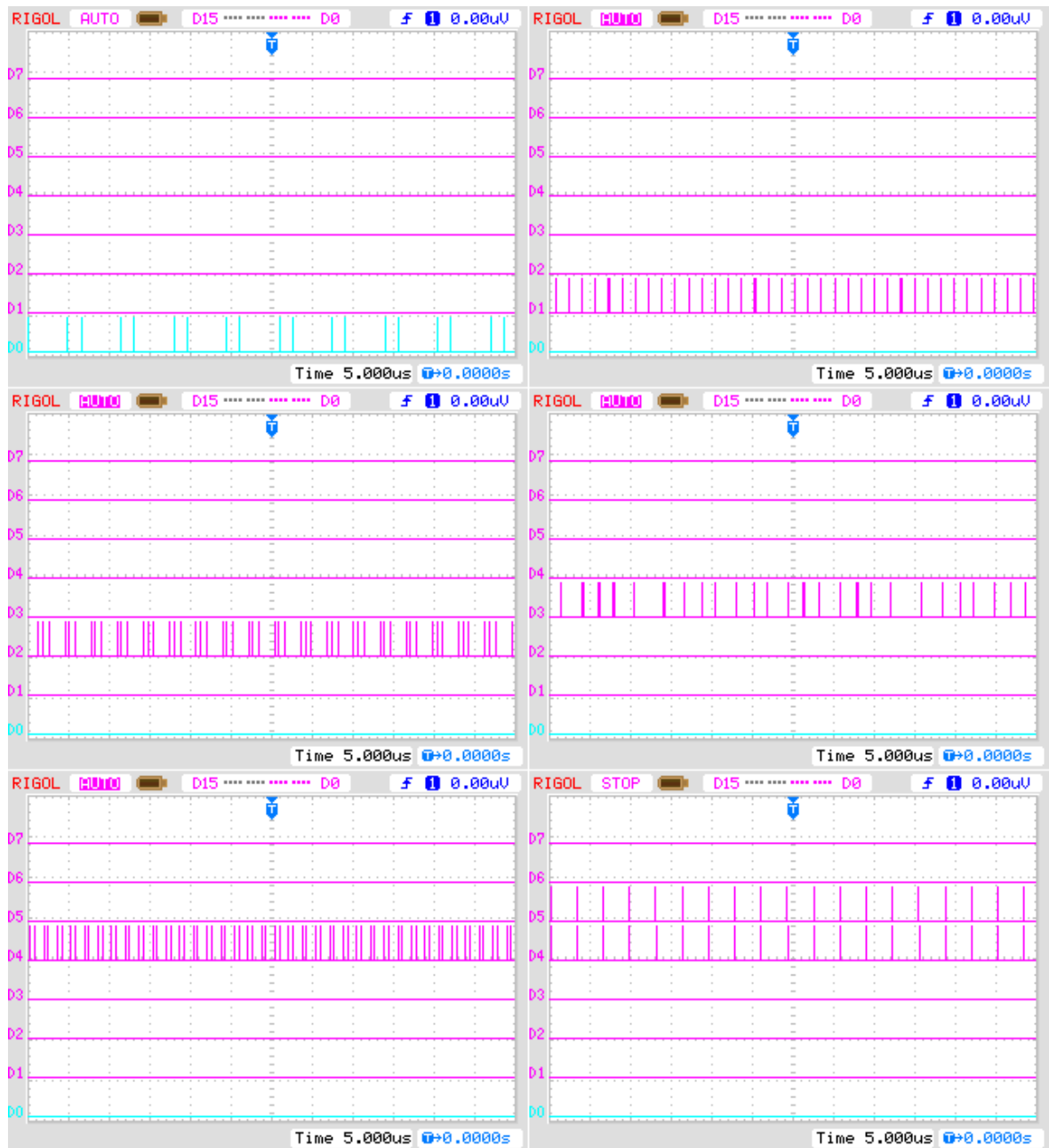


**Figure 6.21.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 4 to 9 respectively (Digit 0 as input pattern for the first image and digit 3 for the second image).

Finally, as done in the simulation section, the digits with noise of Figure 6.3 have been introduced to the SNN in order to test its functionality of recognizing similar patterns. Meaning that the training has been performed with the regular digits, while the final test has been done with the digits with noise. Firstly, for the SNN trained for the recognition of digits 0 to 5 and secondly, for the SNN trained for the recognition of digits 4 to 9. The final results are shown in Figures 6.22 and 6.23 respectively, and as it can be observed the SNN is able to recognize the corresponding digit regardless of the noise in its image.



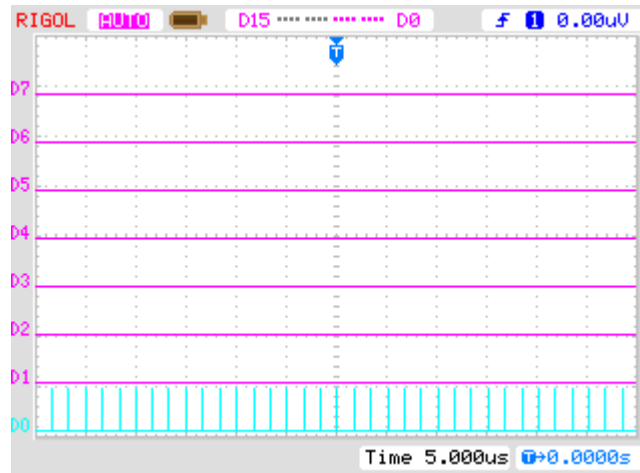
**Figure 6.22.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 0 to 5 with noise respectively (Digit 0 as input pattern for the first image, digit 1 for second image...).



**Figure 6.23.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 4 to 9 with noise respectively (Digit 0 as input pattern for the first image, digit 1 for second image...).

As done before, the inputs of digits 6 to 9 with noise has been introduced to the SNN presented in Figure 6.21 and the inputs of digits 0 to 3 with noise into the SNN presented in Figure 6.22.

As it can be observed in Figure 6.23 when introducing an 8 with noise, the neuron corresponding to digit 0 fires while the neuron corresponding to digit 3 stays at rest unlike the behavior shown at Figure 6.20.



**Figure 6.24.** Spikes from output neurons 0 to 5 of the neural network for the pattern recognition of digits 0 to 5 respectively (Digit 8 with noise as input pattern).

On the other hand, the response of the SNN of figure 6.23 has been null to the all inputs of digits 0 to 3 with noise unlike the behavior shown at Figure 6.21 where the neuron corresponding to digit 8 would fire when introducing a 0 or 3 as stimuli to the SNN.

## Conclusions

The realization of this project has consisted in the achievement of the main goals set from the very beginning. There was a clear initial idea, an artificial neural network, yet the amount of possibilities to approach this endeavor presented a difficult task in order to decide how it could be done, and this has been an ongoing process during the design of the digital system.

Firstly, by starting to collect different information regarding artificial neural networks and the different types of those, the SNNs (Spiking neural networks) came out to the surface as one of the most realistic and modern approach to artificial neural networks.

The next step was to decide which model of SNN to use that was suitable for a digital implementation, meaning that it was computationally simple but capable of producing pulse patterns exhibited by biological neurons. The Izhikevich neuron model was found, which was developed in 2003 with this particular aim.

The design of the digital system started by the implementation of a single Izhikevich neuron, but since the provided equations were using all kind of operations like divisions, multiplications and decimals number they needed to be adapted for a proper digital implementation. One of the first approaches to such an adaptation was a fixed-point model implementation presented by A. Cassidy and A. G. Andreou at [21], yet the use of custom libraries for the use of fixed-point representation numbers presented a difficulty for the design. Therefore, a custom adaptation of the neuron model was made, inspired on the work by A. Cassidy and A. G. Andreou, in which all the divisions and multiplications were implemented as static shift operations, greatly reducing the amount of needed resources for the digital implementation of the neuron. Finally, by doing numerous rough estimations, the used FPGA could emulate almost 500 neurons of the adapted model. Meanwhile only 80 neurons of the non-adapted model, which correlated into an unignorable difference.

Once the first neuron was implemented, the next aim was to instantiate as many neurons as possible and connect them in order to develop a neural network. By performing a research on several digital implementations of SNNs, it was found that one of the most used system for the communication of this type of networks is the AER (Address-Event Representation) system. Since a template or digital design of this system was not found but its functionality was clear, a custom approach to this digital system was designed. Therefore, all the spikes of the neurons could be read and translated to the corresponding addresses.

Furthermore, the magnitude of the digital design was growing and every neuron needed to store all of their synapsis weights. Thus, a RAM was implemented to the neuron model so the neuron could be

able to read which synaptic weight should be applied to itself depending on the neuron that spiked. Also, it was given the capacity to initialize all the values of this RAM through a text file in order to establish the connections between neurons and create different neural networks.

At this point, the digital system was capable of emulating any SNN and simple combinations were proposed, yet they would not present any functionality. First, a SNN was searched in order to obtain the connections and synaptic weights so it could be implemented in the digital system and prove its correct functionality, yet none was found. Therefore, several learning methods were studied in order to be able to train any given neural network and implement some sort of functionality. The most used training method for SNNs is the STDP (Spike-Timing-Dependent Plasticity) and since a SNN with the given synaptic weights was not found it was proceed to the digital implementation of this learning rule, which ended up being one of the inflection points of the project. Finally, the STDP was implemented thanks to the low complexity combinational digital logic approach presented in [23].

At the end, a custom SNN was presented for pattern recognition tasks thanks to the simple training method provided in [19]. And was successfully implemented into a FPGA, giving the ability to the user to decide which patterns to learn and train to the neural network for their recognition.

Since the digital system designed in this project aims to be a default template for the emulation of any SNN, three fundamental and independent modules were implemented in order to provide a flexible, simple, efficient and scalable solution. As explained above, these are: first, the neuron; second, the AER communication bus; and thirdly, the STDP learning system. Therefore, connecting and replicating these main digital modules the emulation of a neural network of two, ten or thousands of neurons is possible if the available resources allow it.

Moreover, a user's manual is provided in the annexes of this project in order to explain the different variables of importance for each module in order to modify the custom SNN presented as an example for pattern recognition tasks or for the emulation of any SNN.

Finally, one of the issues faced during the development of the project was the inability to communicate the FPGA with a computer. Therefore, the pattern recognition tasks were limited to images of 35 pixels which were implemented by hand into a multiplexer as stimuli of the SNN. One possible future work would be to develop a FPGA communication system with a computer in order to be able to work with much larger images and modify the given SNN to prove its flexibility and scalability. In addition, another future work would be to develop a top entity which could allow the user to define the number of layers and neurons per layer and implement the specified SNN with the correct interconnection of the three given modules in this project.

Definitely, in this project a digital system for SNN emulation has been implemented from and a custom SNN for pattern recognition tasks has been designed in order to successfully validate the functionality of the system, which allowed to acquire a lot of knowledge about the artificial neural networks and proved difficult during some stages but very rewarding with the final outcome of it.





## Cost estimation

This annex is about the cost estimation to carry out the totality of this final project, which takes into account the money spent on the devices, licenses and hours of work required to design, implement and verify the results of the developed digital system in the laboratory.

Firstly, the used devices for the successful realization of the digital system are: a personal computer to design, synthesize and simulate the design along with the creation of the different schematics and block diagrams presented in this report, a Nexys4 DDR Artix-7 FPGA trainer board where the design is implemented to demonstrate the functionality of the digital system, a RIGOL DS1022CD digital oscilloscope that allows to capture the spikes of the spiking neural network and an AD Instruments DS2202A analog oscilloscope to show the shape of a single spike.

	Total price (€)	Useful life (yr.)	Time used (yr.)	Eqv. price (€)
Computer	800,00	4	0.5	100,00
Nexys4 DDR Artix-7	294,65	4	0.5	36,83
RIGOL DS1022CD	715,00	6	0.5	59,58
AD INSTR. DS2202A	595,00	6	0.5	49,58
<b>Total</b>	<b>1.809,65</b>	<b>-</b>	<b>-</b>	<b>245,99</b>

**Table 1.** Equivalent price of used devices.

Secondly, the cost of the licenses of the different software used to develop this project are; the Vivado WebPack for students license to design, synthesize, simulate and implement the digital system into the FPGA board, the Microsoft Office package to create the different schematics and block diagrams along with the report of this project, and a Windows 10 license for the computer.

	Price per year (€/yr.)	Time used (yr.)	Eqv. price (€)
Vivado WebPack	0,00	0.5	0,00
Microsoft Office	29,89	0.5	14,95
Windows 10	44,00	0.5	22,00
<b>Total</b>	<b>73,89</b>	<b>-</b>	<b>36,95</b>

**Table 2.** Equivalent price of software licenses.

Furthermore, there is the cost of the working hours to develop the digital system of this project. Since this is the work of an engineering student, a price of approximately 8€ per hour has been established as a recommendation in the educational cooperation agreement between the UPC and the companies that offer academic practices for engineering students.

	Work (hours)	Price per hour (€)	Total price (€)
Engineering student	600	8	<b>4800,00</b>

**Table 3.** Total price of the engineering student.

Ultimately, the total price to carry out the totality of this final project is obtained by computing the sum of all the calculated prices above starting with the used devices, software licenses and ending with the cost of an engineering student's work hours.

	Price (€)
Devices	245,99
Software licenses	36,95
Engineering student	4800,00
<b>Total</b>	<b>5082,94</b>

**Table 4.** Total price of the project.

## References

- [1] Cassidy, a, S Denham, P Kanold, and a Andreou. 2007. "FPGA Based Silicon Spiking Neural Array." *Biomedical Circuits and Systems Conference, 2007. BIOCAS 2007. IEEE*, no. 1: 75–78. doi:10.1109/BIOCAS.2007.4463312.
- [2] Pirrone, Vito. n.d. "A Large-Scale Spiking Neural Network Emulation," 1–95.
- [3] Commons, Creative, and Attribution License. 2002. "Neurons and Glial Cells Cellule Gliali," 29–32. <https://cnx.org/contents/c9j4p0aj@4/Neurons-and-Glial-Cells>.
- [4] Furtak, S. (2017). Neurons. In R. Biswas-Diener & E. Diener (Eds), Noba textbook series: Psychology. Champaign, IL: DEF publishers. DOI:nobaproject.com
- [5] Krenker, Andrej, Andrej Kos, Janez Bešter, and Andrej Kos. 2011. "Introduction to the Artificial Neural Networks." *European Journal of Gastroenterology & Hepatology* 19 (12): 1046–54.
- [6] Ratika, Pradhan, Mohan Pradhan P., Ashish Bhusan, Ronak. K. Pradhan, and M. K. Ghose. 2010. "An Introduction to Artificial Neural Networks ( ANN ) - Methods , Abstraction , and Usage." *Journal of Computing* 2 (3): 1–8.
- [7] Kiyoshi Kawaguchi. 2000. "2.3.1 The McCulloch-Pitts Model of Neuron." <http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html>.
- [8] Hajek, M. 2005. "Neural Networks," 10–13. doi:10.1016/j.neunet.2004.10.001.
- [9] Song, S, K D Miller, and L F Abbott. 2000. "Competitive Hebbian Learning through Spike-Timing-Dependent Synaptic Plasticity." *Nature Neuroscience* 3 (9): 919–26. doi:10.1038/78829.
- [10] Moore, Simon W., Paul J. Fox, Steven J.T. Marsh, A. Theodore Markettos, and Alan Mujumdar. 2012. "Bluehive - A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation." In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 133–40. IEEE. doi:10.1109/FCCM.2012.32.
- [11] Cassidy, Andrew, Andreas G. Andreou, and Julius Georgiou. 2011. "Design of a One Million Neuron Single FPGA Neuromorphic System for Real-Time Multimodal Scene Analysis." In *2011 45th Annual Conference on Information Sciences and Systems*, 1–6. IEEE. doi:10.1109/CISS.2011.5766099.
- [12] Sripad, Athul, and Jordi Madrenas. 2013. "MSc Thesis SNAVA : A Generic Threshold-Based-

SNN Emulation Solution Master of Science in Information and Communication Technologies ( MINT ) Author : Tiruvendipura Achyutha Raghavan Date : September 2013,” no. September.

- [13] Kravchuk, Kseniia. 2016. “Leaky Integrate-and-Fire Neuron under Poisson Stimulation.” In *2016 II International Young Scientists Forum on Applied Physics and Engineering (YSF)*, 203–6. IEEE. doi:10.1109/YSF.2016.7753837.
- [14] Nelson, Mark, and John Rinzel. 1990. “The Hodgkin-Huxley Model.” *Genesis* 125 (20): 29–50. doi:10.1063/1.2400034.
- [15] Izhikevich, E.M. 2003. “Simple Model of Spiking Neurons.” *IEEE Transactions on Neural Networks* 14 (6): 1569–72. doi:10.1109/TNN.2003.820440.
- [16] Boahen, Kwabena A. 2000. “Point-to-Point Connectivity between Neuromorphic Chips Using Address Events.” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47 (5): 416–34. doi:10.1109/82.842110.
- [17] Moreno, J. M., J. Madrenas, and L. Kotynia. 2009. “Synchronous Digital Implementation of the AER Communication Scheme for Emulating Large-Scale Spiking Neural Networks Models.” *Proceedings - 2009 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009*, 189–96. doi:10.1109/AHS.2009.14.
- [18] Diehl, Peter, and Matthew Cook. 2015. “Unsupervised Learning of Digit Recognition Using Spike-Timing-Dependent Plasticity.” *Frontiers in Computational Neuroscience* 9 (August): 99. doi:10.3389/fncom.2015.00099.
- [19] Mikkonen, Tommi, Vafa Andalibi, and Kai Koskimies. n.d. “Pattern Recognition with Spiking Neural Networks : A Simple Training Method.”
- [20] Digilent. 2017. “Nexys 4 Artix-7 FPGA Trainer Board (LIMITED TIME); See Nexys4 DDR - Digilent.” Accessed May 14. <http://store.digilentinc.com/nexys-4-artix-7-fpga-trainer-board-limited-time-see-nexys4-ddr/>.
- [21] Cassidy, Andrew, and Andreas G. Andreou. 2008. “Dynamical Digital Silicon Neurons.” *2008 IEEE-BIOCAS Biomedical Circuits and Systems Conference, BIOCAS 2008*, 289–92. doi:10.1109/BIOCAS.2008.4696931.
- [22] Linares-Barranco, Alejandro. 2003. “Estudio Y Evaluación de Interfaces Para Conexión de Sistemas Neuromórficos Mediante Address-Event Representation (AER),” 228.
- [23] Cassidy, Andrew, Andreas G. Andreou, and Julius Georgiou. 2011. “A Combinational Digital Logic Approach to STDP.” *Proceedings - IEEE International Symposium on Circuits and Systems*, 673–76. doi:10.1109/ISCAS.2011.5937655.
- [24] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

doi:10.1038/nmeth.3707.

- [25] European Union, CORDIS, [www.cordis.europa.eu](http://www.cordis.europa.eu). 2015. "Thinking about Numbers - Mathematics - Information Science - Mathematics and Statistics - v.EN, Science - Studentnews.eu." <http://science.studentnews.eu/s/3942/76611-Mathematics/4074323-Thinking-about-numbers.htm>.
- [26] Bichler, Olivier, Damien Querlioz, Simon J. Thorpe, Jean Philippe Bourgoin, and Christian Gamrat. 2012. "Extraction of Temporally Correlated Features from Dynamic Vision Sensors with Spike-Timing-Dependent Plasticity." *Neural Networks* 32. Elsevier Ltd: 339–48. doi:10.1016/j.neunet.2012.02.022.
- [27] Izhikevich, E M. 2006. "Polychronization: Computation with Spikes." *Neural Computation* 18 (2): 245–82. doi:10.1162/089976606775093882.
- [28] Izhikevich, Eugene M. 2004. "Which Model to Use for Cortical Spiking Neurons?" *IEEE Transactions on Neural Networks* 15 (5): 1063–70. doi:10.1109/TNN.2004.832719.
- [29] Linares-Barranco, Alejandro. 2003. "Estudio Y Evaluación de Interfaces Para Conexión de Sistemas Neuromórficos Mediante Address-Event Representation (AER)," 228.
- [30] Rivera, Giovanni Sanchez. 2014. "Efficient Multiprocessing Architectures for Spiking Neural Network Emulation Based on Configurable Devices." *Thesis UPC*.
- [31] Zupan, Jure. 1994. "Introduction to Artificial Neural Network (ANN) Methods: What They Are and How to Use Them." *Acta Chimica Slovenica* 41 (September): 327–52. <http://www2.ccc.uni-erlangen.de/publications/ANN-book/publications/ACS-41-94.pdf>.



## Annexes

### A1. User's Manual

#### 1. Introduction

This document is intended to be used by any individual interested in developing the emulation of a spiking neural network (SNN) by using the digital system presented in this project. For such an aim, a software suite for synthesis and analysis of VHDL designs is needed, for this project Vivado Design Suite has been used with the free WebPack license.

#### 2. Overview

The digital system is formed by several VHDL files for different purposes:

- **Design Sources:** *Top.vhd, AER\_Bus.vhd, IZH\_Neuron.vhd, RAM\_09.vhd, STDP.vhd.*
- **Constraints:** *Nexys4\_Master.xdc.*
- **Simulation Sources:** *Testbench - Top.vhd, Testbench - IZH\_Neuron.vhd, Testbench - AER\_Bus.vhd, Testbench - STDP.vhd.*

First of all, the design sources, which include the AER system, an Izhikevich neuron with its RAM and the STDP training module that can be used as default templates for developing any type of SNN. In addition, a *Top* entity is provided in which an example of a SNN for pattern recognition is designed.

Second, a constraint file that together with all the provided design sources gives the ability to implement the SNN for pattern recognition into a Nexys 4 Artix-7 FPGA.

Finally, four test bench, one to simulate the example of a SNN for pattern recognition and three to simulate each one of the modules proposed as default templates (*AER, IZH\_Neuron* and *STDP*) for any type of SNN emulation.

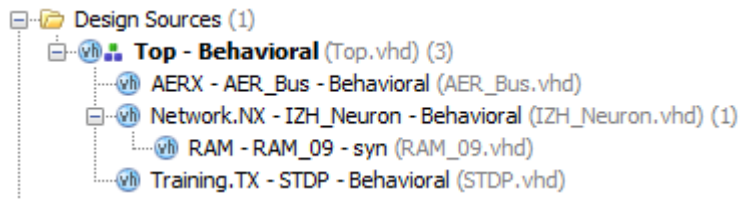
#### 3. Design sources

This section's aim is to provide the general instructions for the use of the different design sources of the design for the emulation of any SNN.

### 1.1. Top entity

The *Top* entity of the design is provided through the *Top.vhd* file. It is the responsible for implementing a type of SNN for pattern recognition making use of the rest of the design sources. For a custom SNN refer to last section to learn how to use the *IZH\_Neuron*, *AER* and *STDP* entities as default templates for your design.

The structure of the files is shown in the following picture from the Vivado's project manager:



**Figure 1.** Design Sources' files structure.

The generic variables are at the top of the file and they can be easily modified in order to adjust the SNN before the synthesis and implementation of the design. These are the following:

GENERIC	Description	Default value
<i>image_num</i>	Number of images or patterns. (If 9 then 10 images since "9 down to 0"). Multiplexer needs to be updated if <i>image_num</i> changed.	9
<i>rest_time</i>	Number of clock cycles between the input stimuli for the SNN (Should be enough for all the neurons of the SNN to return to their default state).	149
<i>train_time</i>	Number of input stimuli for a training phase.	200
<i>train_spike</i>	Number of clock cycles after the input stimuli to generate the spike for the selected training neuron.	10
<i>untrain_spike</i>	Number of clock cycles before the input stimuli to generate the spike for the unselected training neurons.	5
<i>pre_reg</i>	STDP pre-spike or increment register width.	15
<i>post_reg</i>	STDP post-spike or decrement register width.	5
<i>width</i>	Number of bits plus one for equation variables of <i>IZH_Neuron</i> .	12
<i>neuron_adr</i>	Number of bits plus one for neuron addresses (If 5 then up to 64 neuron addresses since $2^6=64$ ).	5
<i>weights</i>	Number of bits plus one for synaptic weights (signed vector).	10
<i>input_neuron_num</i>	Number of input and training neurons minus one.	40
<i>training_neuron_num</i>	Number of training neurons.	6
<i>neuron_num</i>	Total number of neurons.	46

**Table 2.** Top's entity generics.



## 1.2. Izhikevich neuron

The Izhikevich neuron module of the design is provided through the *IZH\_Neuron.vhd* and *RAM\_09.vhd* files. It is the responsible for implementing a neuron of the Izhikevich model.

The structure of the files is shown in the following picture from the Vivado's project manager:



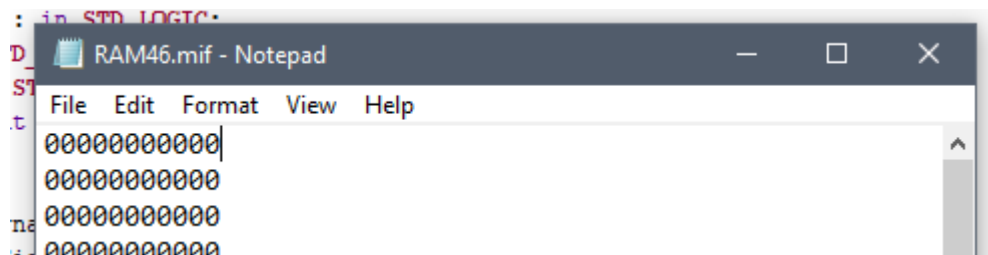
**Figure 2.** Izhikevich neuron's files structure.

The generic variables are at the top of the file and they can be easily modified in order to adjust the neuron before the synthesis and implementation of the design. These are the following:

GENERIC	Description	Default value
<i>number</i>	Address number assigned to the neuron.	0
<i>width</i>	Number of bits plus one for equation variables of <i>IZH_Neuron</i> .	12
<i>neuron_adr</i>	Number of bits plus one for neuron addresses (If 5 then up to 64 neuron addresses since $2^6=64$ ).	5
<i>weights</i>	Number of bits plus one for synaptic weights (signed vector).	10

**Table 2.** Izhikevich neurons' generics.

In addition, a .mif file can be provided in order to initialize the synaptic weights values of the neuron's RAM. These files can be easily created with a text editor or notepad always making sure the number of bits is correct. In Figure 3 an 11-bit signed vector is provided for each address of the RAM, which corresponds to the default value of the *weights* generic.



**Figure 3.** Path for the initialization file of the RAM for each neuron.

Furthermore, the path of the file for the initialization of the synaptic weight values of the neurons' RAM needs to be defined in the *RAM\_09.vhd* file in order to synthesize the design. This can be found at line 46 as shown in the following picture of the code:

```
46 signal RAM : ram_type := init_mem("C:\Users\emerino\Desktop\SNN_1\RAM\RAM" & INTEGER'image(number) & ".mif");
```

**Figure 4.** Path for the initialization file of the RAM for each neuron.

Failing to define the path and create the corresponding .mif files will cause an error when synthesizing the design. However, if such initialization is not needed it can be removed from the code by replacing this line for: *signal RAM : ram\_type := (others => '0');*

### 1.3. AER system

The AER communication system of the design is provided through the *AER\_Bus.vhd* file. It is the responsible for implementing the AER communication system.

The generic variables are at the top of the file and they can be easily modified in order to adjust the neuron before the synthesis and implementation of the design. These are the following:

GENERIC	Description	Default value
<i>neuron_adr</i>	Number of bits plus one for neuron addresses (If 5 then up to 64 neuron addresses since $2^6=64$ ).	5
<i>neuron_num</i>	Total number of neurons.	46

**Table 3.** AER system's generics.

### 1.4. STDP (Spike-Timing-Dependent Plasticity)

The STDP module of the design is provided through the *STDP.vhd* file. It is the responsible for implementing the STDP learning rule.

The generic variables are at the top of the file and they can be easily modified in order to adjust the neuron before the synthesis and implementation of the design. These are the following:

GENERIC	Description	Default value
<i>pre_reg</i>	STDP pre-spike or increment register width.	15
<i>post_reg</i>	STDP post-spike or decrement register width.	5
<i>neuron_adr</i>	Number of bits plus one for neuron addresses (If 5 then up to 64 neuron addresses since $2^6=64$ ).	5
<i>weights</i>	Number of bits plus one for synaptic weights (signed vector).	10
<i>input_neuron_num</i>	Number of input and training neurons minus one.	40
<i>training_neuron_num</i>	Number of training neurons.	6

**Table 4.** STDP's generics.

## 4. Constraints

The constraint provided for this digital system is meant for a Nexys 4 Artix-7 FPGA and the emulated SNN for pattern recognition using the *Top.vhd* entity. For the master file constraint of your FPGA refer to your device's manufacturer.

This file links the ports of the *Top.vhd* entity with the different FPGA I/O ports. Nevertheless, since this design has been implemented with Vivado certain timing requirements had to be accomplished. The following input and output delays have been set for a proper implementation of the design:

```
set_input_delay -clock sys_clk_pin 1.0 [get_ports RST]
set_input_delay -clock sys_clk_pin 1.0 [get_ports BTN]
set_input_delay -clock sys_clk_pin 1.0 [get_ports Image]
set_input_delay -clock sys_clk_pin 1.0 [get_ports Neuron]
set_output_delay -clock sys_clk_pin -1.5 [get_ports Spikes_out]
```

Figure 5. Input and output delays.

## 5. Simulation sources

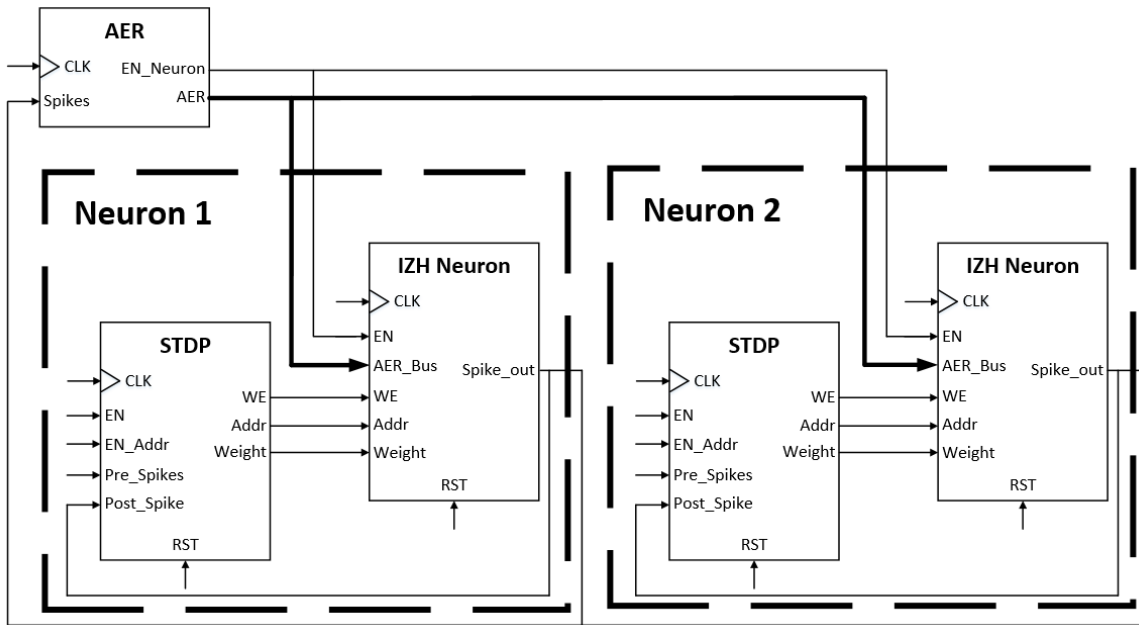
Four test benches, one to simulate the example of a SNN for pattern recognition and three to simulate each one of the modules proposed as default templates (*AER*, *IZH\_Neuron* and *STDP*) for any type of SNN emulation. The generic variables of each are the same as the ones described in the design sources section.

## 6. Custom SNN emulation

Figure 6 shows the interconnection of the three fundamental design sources: *IZH\_Neuron*, *AER* and *STDP* for a custom SNN emulation of two neurons. However, this architecture allows the implementation of as many neurons as needed.

Firstly, one *AER* system is implemented in order to establish the communication between all the neurons, i.e. read the spikes to translate them into the appropriate address and transmit them to the *AER* bus. In addition, all neurons are connected to the *EN\_Neuron* signal which allows to stop the activity of the neurons since the *AER* bus can only transmit one address per clock cycle.

Secondly, each neuron is formed by its digital module and a *STDP* learning module. The interconnection between these two is established with the write enable (*WE*), address (*Addr*) and synaptic weight (*Weight*) signals that allow to write in the neuron's RAM.



**Figure 6.** Interconnection of the different blocks for the emulation of SNN of two or more neurons.

Finally, from the STDP modules the *EN* and *EN\_Addr* signals along with the *Pre\_Spikes* signal need to be treated with specific digital blocs in order to work for the custom SNN design.

## A2. Computer files

### 1. Design Sources

#### 1.1. Top entity

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorqui
-- Create Date:   10:59:16 02/26/2017
-- Module Name:   Top - Behavioral
-- Project Name:  Digital System for Neural Network Emulation
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity Top is
    Generic ( image_num : integer := 9;           -- number of images
              rest_time : integer := 149;        -- rest time for a
              neuron to return to their default state
              train_time : integer := 200;        --
              train_time*rest_time
              train_spike : integer := 10;        -- time after the
              stimuli to generate a spike
              untrain_spike : integer := 5;       -- time before the
              input stimuli to generate a spike
              pre_reg : integer := 15;           -- STDP Pre-Spike
              register width
              post_reg : integer := 5;           -- STDP Post-Spike
              register width
              width : integer := 12;            -- IZH_Neuron eq
              variables width
              neuron_adr : integer := 5;         -- Up to 64
              neuron_adr
              weights : integer := 10;          -- 255 downto -256
              input_neuron_num : integer := 40;  -- Number of virtual
              and training neurons +1
              training_neuron_num : integer := 6; -- Number of training
              neurons
              neuron_num : integer := 46);       -- Number of neurons
              +1
    Port ( CLK : in STD_LOGIC;
           RST : in STD_LOGIC;
           BTN : in STD_LOGIC;
           SEL : in STD_LOGIC;
           Image : in STD_LOGIC_VECTOR(image_num downto 0);
           Neuron : in STD_LOGIC_VECTOR(neuron_num-input_neuron_num-1
              downto 0));
           Spikes_out : out STD_LOGIC_VECTOR(neuron_num-input_neuron_num-
              1 downto 0));
end Top;

```

**architecture** Behavioral **of** Top **is**

```

COMPONENT AER_Bus
GENERIC(
    neuron_adr : in integer;
    neuron_num : in integer);
PORT(
    CLK : in STD_LOGIC;
    Spikes : in STD_LOGIC_VECTOR(neuron_num downto 0);
    EN_Neuron : out STD_LOGIC;
    AER : out STD_LOGIC_VECTOR(neuron_adr downto 0));
end COMPONENT;

```

```

COMPONENT IZH_Neuron
GENERIC(
    number : in integer;
    width : in integer;
    neuron_adr : in integer;
    weights : in integer);
PORT(
    CLK : in std_logic;
    RST : in std_logic;
    EN : in std_logic;
    WE : in std_logic;
    Addr : in std_logic_vector(neuron_adr downto 0);
    Weight : in std_logic_vector(weights downto 0);
    AER_Bus : in std_logic_vector(neuron_adr downto 0);
    Spike_out : out std_logic);
end COMPONENT;

```

```

COMPONENT STDP
GENERIC(
    neuron_adr : in integer;
    weights : in integer;
    input_neuron_num : in integer;
    training_neuron_num : in integer;
    pre_reg : in integer;
    post_reg : in integer);
PORT(
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    EN : in STD_LOGIC;
    EN_Addr : in STD_LOGIC;
    Pre_Spikes : in STD_LOGIC_VECTOR(input_neuron_num-
training_neuron_num downto 0);
    Post_Spike : in STD_LOGIC;
    WE : out STD_LOGIC;
    Addr : out STD_LOGIC_VECTOR(neuron_adr downto 0);
    Weight : out STD_LOGIC_VECTOR (weights downto 0));
end COMPONENT;

```

```

signal RST_Signal : std_logic;
signal Image_Signal : std_logic_vector(image_num downto 0);
signal Neuron_Signal : std_logic_vector(neuron_num-input_neuron_num-1
downto 0);
signal Spikes_Signal : std_logic_vector(neuron_num-input_neuron_num-1
downto 0) := (others => '0');

```

```

    signal Counter : unsigned(7 downto 0) := (others => '0');

    signal Pixels, Digit, Digit_Noise : std_logic_vector(input_neuron_num-
training_neuron_num downto 0);
    signal Spikes_in : std_logic_vector(input_neuron_num downto 0);
    signal Spikes : std_logic_vector(neuron_num downto 0) := (others =>
'0');
    signal AER : std_logic_vector(neuron_adr downto 0);
    signal EN_Neuron : std_logic;

    type TYPES is (NP,P0,P1);
    signal STATE0,STATE1: TYPES;

    signal BTN_Rebound : unsigned(19 downto 0);
    signal BTN_Signal : std_logic;
    signal EN_Pulse : std_logic;
    signal Pulse : unsigned(16 downto 0) := (others => '0');

    signal EN_STDP : std_logic;
    signal EN_Train : std_logic;
    signal EN_Addr : std_logic;
    signal WE : std_logic_vector(neuron_num downto input_neuron_num+1);
    signal Pre_Spikes : std_logic_vector(input_neuron_num-
training_neuron_num downto 0);
    type addr_type is array (input_neuron_num+1 to neuron_num) of
std_logic_vector(neuron_adr downto 0);
    signal Addr : addr_type := (others => (others => '0'));
    type weight_type is array (input_neuron_num+1 to neuron_num) of
std_logic_vector(weights downto 0);
    signal Weight : weight_type := (others => (others => '0'));

begin

-- RST Signal

    process(clk)
    begin
        if (CLK='1' and CLK'event) then
            if RST='1' then
                RST_Signal<='1';
            else
                RST_Signal<='0';
            end if;
        end if;
    end process;

-- Counter

    process(clk)
    begin
        if (CLK='1' and CLK'event) then
            if (EN_Neuron='1') then
                if Counter=rest_time then
                    Counter<=(others => '0');
                else
                    Counter <= Counter + 1;
                end if;
            end if;
        end if;
    end process;

```

```

        end if;
    end process;

-- Input neurons

    process(clk)
    begin
        if (CLK='1' and CLK'event) then
            Image_Signal <= Image;
            Neuron_Signal <= Neuron;
        end if;
    end process;

    Digit <= "01110100011000110001100011000101110" when
Image_Signal(0)='1' else
    "00100011000010000100001000010001110" when
Image_Signal(1)='1' else
    "01110100010000100010001000100011111" when
Image_Signal(2)='1' else
    "01110100010000100110000011000101110" when
Image_Signal(3)='1' else
    "00010001100101010010111110001000010" when
Image_Signal(4)='1' else
    "11111100001111000001000011000101110" when
Image_Signal(5)='1' else
    "00110010001000011110100011000101110" when
Image_Signal(6)='1' else
    "1111100001000100010001000100001000" when
Image_Signal(7)='1' else
    "01110100011000101110100011000101110" when
Image_Signal(8)='1' else
    "01110100011000101111000010001001100" when
Image_Signal(9)='1' else
    (others => '0');

    Digit_Noise <= "11111100011000110001100011000111111" when
Image_Signal(0)='1' else
    "00100011000010000100001000010000100" when
Image_Signal(1)='1' else
    "01110000010000100010001000100001111" when
Image_Signal(2)='1' else
    "01110000010000100110000010000101110" when
Image_Signal(3)='1' else
    "00010100101001010010111110001000010" when
Image_Signal(4)='1' else
    "11110100001111000001000010000101110" when
Image_Signal(5)='1' else
    "00110010001000001110100001000101110" when
Image_Signal(6)='1' else
    "11111000010001000000010000100001000" when
Image_Signal(7)='1' else
    "01110100011000111111100011000101110" when
Image_Signal(8)='1' else
    "01110100011000101110000010001000100" when
Image_Signal(9)='1' else
    (others => '0');

    Pixels <= Digit_Noise when SEL='1' else Digit;

```



```

    Spikes_in(input_neuron_num downto 0) <= Neuron_Signal & Pixels;

    Input_Neurons : for I in 0 to input_neuron_num-training_neuron_num
generate
    process(clk)
    begin
        if (CLK='1' and CLK'event) then
            if Counter=rest_time then
                Spikes(I)<=Spikes_in(I);
            else
                Spikes(I)<='0';
            end if;
        end if;
    end process;
end generate Input_Neurons;

-- STDP Change synaptic addr

process(clk)
begin
    if (CLK='1' and CLK'event) then
        if Counter=rest_time-1 then
            EN_Addr<='1';
        else
            EN_Addr<='0';
        end if;
    end if;
end process;

-- Training Neurons

    Training_Neurons : for I in input_neuron_num-training_neuron_num+1 to
input_neuron_num generate
    process(clk)
    begin
        if (CLK='1' and CLK'event) then
            if(EN_STDP='1') then
                if(Spikes_in(I)='1' and Counter=train_spike) then
                    Spikes(I)<='1';
                elsif(Spikes_in(I)='0' and Counter=rest_time-
untrain_spike) then
                    Spikes(I)<='1';
                else
                    Spikes(I)<='0';
                end if;
            else
                Spikes(I)<='0';
            end if;
        end if;
    end process;
end generate Training_Neurons;

-- Output Spikes

process (CLK)
begin
    if (CLK='1' and CLK'event) then

```

```

        Spikes_Signal<=Spikes(neuron_num downto input_neuron_num+1);
        Spikes_out<=Spikes_Signal;
    end if;
end process;

-- State Machine to detect the pushbuttons

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        case STATE0 is
            when NP =>
                if BTN='1' then
                    STATE0 <= P0; BTN_Signal <= '0';
                else
                    STATE0 <= NP; BTN_Signal <= '0';
                end if;
            when P0 =>
                STATE0 <= P1; BTN_Signal <= '1';
            when P1 =>
                if BTN='1' then
                    STATE0 <= P1; BTN_Signal <= '0';
                elsif BTN_Rebound=2000 then
                    STATE0 <= NP; BTN_Signal <= '0';
                end if;
            end case;
        end if;
    end process;

process(CLK)
begin
    if (CLK='1' and CLK'event) then
        if (BTN_Signal='1') then
            BTN_Rebound <= (others=>'0');
        else
            if BTN_Rebound=2000 then
                BTN_Rebound <= (others=>'0');
            else
                BTN_Rebound <= BTN_Rebound + 1;
            end if;
        end if;
    end if;
end process;

-- Enable signal for the STDP Module

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        case STATE1 is
            when NP =>
                if BTN_Signal='1' then
                    STATE1 <= P0;
                    En_Pulse <= '1';
                    EN_STDP <= '1';
                else
                    En_Pulse <= '0';
                    EN_STDP <= '0';
                end if;
            end case;
        end if;
    end process;

```

```

        end if;
    when P0 =>
        STATE1 <= P1;
    when P1 =>
        if Pulse=train_time*rest_time then
            STATE1 <= NP;
        else
            En_Pulse <= '1';
            EN_STDP <= '1';
        end if;
    end case;
end if;
end process;

process(clk)
begin
    if (CLK='1' and CLK'event) then
        if (EN_Neuron='1') then
            if(En_Pulse='1') then
                if Pulse=train_time*rest_time then
                    Pulse<=(others => '0');
                else
                    Pulse <= Pulse + 1;
                end if;
            end if;
        end if;
    end if;
end process;

-- AER

AERX:AER_Bus
GENERIC MAP(
    neuron_adr => neuron_adr,
    neuron_num => neuron_num)
PORT MAP(
    CLK => CLK,
    Spikes => Spikes,
    EN_Neuron => EN_Neuron,
    AER => AER);

-- IZH Neurons

Network : for I in input_neuron_num+1 to neuron_num generate
    NX:IZH_Neuron
    GENERIC MAP(
        number => I,
        width => width,
        neuron_adr => neuron_adr,
        weights => weights)
    PORT MAP(
        CLK => CLK,
        RST => RST_Signal,
        EN => EN_Neuron,
        WE => WE(I),
        Addr => Addr(I),
        Weight => Weight(I),
        AER_Bus => AER,

```

```

        Spike_out => Spikes(I));
    end generate Network;

-- STDP

    Pre_Spikes <= Spikes(input_neuron_num-training_neuron_num downto 0);

    EN_Train <= EN_STDP and EN_Neuron;

    Training : for I in input_neuron_num+1 to neuron_num generate
        TX:STDP
        GENERIC MAP(
            neuron_adr => neuron_adr,
            weights => weights,
            input_neuron_num => input_neuron_num,
            training_neuron_num => training_neuron_num,
            pre_reg => pre_reg,
            post_reg => post_reg)
        PORT MAP(
            CLK => CLK,
            RST => RST_Signal,
            EN => EN_Train,
            EN_Addr => EN_Addr,
            Pre_Spikes => Pre_Spikes,
            Post_Spike => Spikes(I-training_neuron_num),
            WE => WE(I),
            Addr => Addr(I),
            Weight => Weight(I));
    end generate Training;

end Behavioral;

```

## 1.2. Izhikevich neuron

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorqui
-- Create Date:   11:53:08 02/12/2017
-- Module Name:   IZH_Neuron - Behavioral
-- Project Name:  Digital System for Neural Network Emulation
-----

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

entity IZH_Neuron is
  Generic ( number : in integer;
            width  : in integer;
            neuron_adr : in integer;
            weights : in integer);
  Port ( CLK : in std_logic;
        RST : in std_logic;
        EN  : in std_logic;
        WE  : in std_logic;
        Addr : in std_logic_vector(neuron_adr downto 0);
        Weight : in std_logic_vector(weights downto 0);
        AER_Bus : in std_logic_vector(neuron_adr downto 0);
        Spike_out : out std_logic);
end IZH_Neuron;

architecture Behavioral of IZH_Neuron is

  COMPONENT RAM_09
  GENERIC(
    width : in integer;
    neuron_adr : in integer;
    weights : in integer;
    number : in integer);
  PORT(
    clk : in std_logic;
    we : in std_logic;
    a : in std_logic_vector(neuron_adr downto 0);
    dpra : in std_logic_vector(neuron_adr downto 0);
    di : in std_logic_vector(weights downto 0);
    dpo : out std_logic_vector(weights downto 0));
  end COMPONENT;

  signal c,d,thresh : signed(width downto 0);
  signal I,v_n,v_n1,u_n,u_n1 : signed(width downto 0) := (others => '0');
  signal v1,v2,v3,u1,u2,u3,u4,u5 : signed(width downto 0);

  signal Synaptic_in : std_logic_vector(weights downto 0);
  signal Spike : std_logic;

  type signed_array is array (0 to 1) of signed(width downto 0);
  signal I_store, v_store, u_store : signed_array := (others => (others=>'0') );

begin

```

```

-- RAM

RAM:RAM_09
GENERIC MAP(
    number => number,
    width => width,
    neuron_adr => neuron_adr,
    weights => weights)
PORT MAP(
    clk => CLK,
    we => WE,
    a => Addr,
    dpra => AER_Bus,
    di => Weight,
    dpo => Synaptic_in);

-- Input align

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if (EN='1') then
            I_store(0)<=resize(signed(Synaptic_in),I_store(0)'length);
            I<=I_store(1);
            if I_store(0)>-140 then
                I_store(1)<=I_store(0);
            else
                I_store(1)<=to_signed(-140,I_store(0)'length);
            end if;
        else
            I_store(0)<=I_store(0)+resize(signed(Synaptic_in),I_store(0)'length);
        end if;
    end if;
end process;

-- "v" Store

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if (EN='1') then
            v_store(0)<=v_n1;
            v_store(1)<=v_store(0);
            v_n<=v_store(1);
        end if;
    end if;
end process;

-- "u" Store

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if (EN='1') then
            u_store(0)<=u_n1;
            u_store(1)<=u_store(0);
            u_n<=u_store(1);
        end if;
    end if;
end process;

```

```

        end if;
    end if;
end process;

-- Parameters

c <= to_signed(-650,width+1);
d <= to_signed(80,width+1);
thresh <= to_signed(300,width+1);

-- "v" Pipeline

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if (EN='1') then
            v3 <= resize( shift_right(v_n*v_n,8)           -- v_n^2/256
                + shift_left(v_n,1) + shift_left(v_n,2)    -- v_n + 5*v_n
                + to_signed(1400,width+1)                  -- + 1400
                - u_n + I,v3'length);                      -- - u_n + I
            end if;
        end if;
    end process;

    Spike <= '1' when v3 > thresh else
        '0';

    v_n1 <= c when RST = '1' else
        v3 when Spike = '0' else
        c when Spike = '1' else
        (others => '0');

    Spike_out <= Spike;

-- "u" Pipeline

u1 <= shift_right(v_n,2);    -- v_n/4
u2 <= u1-u_n;                -- v_n/4 - u_n
u3 <= shift_right(u2,6);     -- (v_n/4 - u_n)/64
u4 <= u_n + u3;              -- u_n + (v_n/4 - u_n)/64
u5 <= u4+d;                  -- u_n + (v_n/4 - u_n)/64 + d

u_n1 <= u4 when RST = '1' else
    u4 when Spike = '0' else
    u5 when Spike = '1' else
    (others => '0');

end Behavioral;

```

### 1.3. RAM

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   10:00:00 03/05/2017
-- Module Name:   RAM_09 - Behavioral
-- Project Name:  Digital System for Neural Network Emulation
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity RAM_09 is
    generic ( number : in integer;
              width  : in integer;
              neuron_adr : in integer;
              weights : in integer);
    port (
        clk : in std_logic;
        we  : in std_logic;
        a   : in std_logic_vector(neuron_adr downto 0);
        dpra : in std_logic_vector(neuron_adr downto 0);
        di  : in std_logic_vector(weights downto 0);
        dpo : out std_logic_vector(weights downto 0));
end RAM_09;

architecture syn of RAM_09 is
    type ram_type is array (0 to 63) of std_logic_vector(weights downto 0);

    impure function init_mem(mif_file_name : in string) return ram_type
    is
        file mif_file : text open read_mode is mif_file_name;
        variable mif_line : line;
        variable temp_bv : bit_vector(weights downto 0);
        variable temp_mem : ram_type;
    begin
        for i in ram_type'range loop
            readline(mif_file, mif_line);
            read(mif_line, temp_bv);
            temp_mem(i) := to_stdlogicvector(temp_bv);
        end loop;
        return temp_mem;
    end function;

    signal RAM : ram_type :=
init_mem("C:\Users\emerino\Desktop\SNN_1\RAM\RAM" & INTEGER'IMAGE(number)
& ".mif");

begin
    process (clk)
    begin
        if (clk'event and clk = '1') then

```



```
        if (we = '1') then
            RAM(conv_integer(a)) <= di;
        end if;
    end if;
end process;

dpo <= RAM(conv_integer(dpra));

end syn;
```

## 1.4. AER system

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   10:59:16 02/26/2017
-- Module Name:   AER Bus - Behavioral
-- Project Name:  Digital System for Neural Network Emulation
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity AER_Bus is
  Generic ( neuron_adr : in integer;
            neuron_num : in integer);
  Port ( CLK : in STD_LOGIC;
        Spikes : in STD_LOGIC_VECTOR(neuron_num downto 0);
        EN_Neuron : out STD_LOGIC;
        AER : out STD_LOGIC_VECTOR(neuron_adr downto 0));
end AER_Bus;

architecture Behavioral of AER_Bus is

  type memory_type is array (0 to 7) of std_logic_vector(neuron_num downto 0);
  signal memory : memory_type := (others => (others => '0'));

  signal FIFO_En : std_logic := '1';
  signal FIFO_in, FIFO_out : std_logic_vector(neuron_num downto 0);
  signal FIFO_ptr : unsigned(2 downto 0) := "000";

  signal ENC_in, ENC_Spikes : std_logic_vector(neuron_num downto 0) := (others => '0');
  signal ENC_out : std_logic_vector(neuron_adr downto 0) := (others => '0');

begin

  -- FIFO

  FIFO_in <= Spikes;

  process(clk)
  begin
    if (CLK='1' and CLK'event) then
      if(FIFO_En='1') then
        if(FIFO_ptr>0) then
          FIFO_out <= memory(0);
          for I in 0 to 6 loop
            memory(I)<=memory(I+1);
          end loop;
          if(FIFO_in>0) then
            memory(to_integer(FIFO_ptr)) <= FIFO_in;
          else
            FIFO_ptr <= FIFO_ptr - 1;
          end if;
        end if;
      end if;
    end if;
  end process;

```

```

        end if;
    else
        FIFO_out <= FIFO_in;
    end if;
else
    if(FIFO_ptr /= "110" and FIFO_in>0) then
        FIFO_ptr <= FIFO_ptr + 1;
        memory(to_integer(FIFO_ptr)) <= FIFO_in;
    end if;
end if;
end if;
end process;

-- Demux of spikes

ENC_in <= FIFO_out when FIFO_En='1' else
ENC_spikes when FIFO_En='0' else
(others=>'0');

-- Encoder

process(clk)
begin
    if (CLK='1' and CLK'event) then
        for I in ENC_in'range loop
            if (ENC_in(I) = '1') then
                ENC_out <=
std_logic_vector(to_signed(I,neuron_adr+1));
                ENC_Spikes <= ENC_in;
                ENC_Spikes(I) <= '0';
                exit;
            else
                ENC_out <= (others=>'1');
            end if;
        end loop;
    end if;
end process;

AER <= ENC_out;

FIFO_En <= '1' when ENC_Spikes="00000" else '0';

-- Neuron's Enable

process(clk)
begin
    if (CLK='1' and CLK'event) then
        if FIFO_En='1' then
            EN_Neuron<='1';
        else
            EN_Neuron<='0';
        end if;
    end if;
end process;

end Behavioral;
```

## 1.5. STDP (Spike-Timing-Dependent Plasticity)

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   12:14:40 01/04/2017
-- Module Name:   STDP - Behavioral
-- Project Name:  Digital System for Neural Network Emulation
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_misc.all;

entity STDP is
  Generic(
    neuron_adr : in integer;
    weights : in integer;
    input_neuron_num : in integer;
    training_neuron_num : in integer;
    pre_reg : in integer;
    post_reg : in integer);
  Port ( CLK : in STD_LOGIC;
        RST : in STD_LOGIC;
        EN : in STD_LOGIC;
        EN_Addr : in STD_LOGIC;
        Pre_Spikes : in STD_LOGIC_VECTOR(input_neuron_num-
training_neuron_num downto 0));
        Post_Spike : in STD_LOGIC;
        WE : out STD_LOGIC;
        Addr : out STD_LOGIC_VECTOR(neuron_adr downto 0);
        Weight : out STD_LOGIC_VECTOR (weights downto 0));
end STDP;

architecture Behavioral of STDP is

  signal pre_shift_reg : std_logic_vector(15 downto 0) := (others => '0');
  signal post_shift_reg : std_logic_vector(5 downto 0) := (others => '0');
  signal Pre_Spike,pre_gate,post_gate,decr_sel,incr_sel,decr,incr:
std_logic := '0';
  signal Syn_Addr : unsigned(neuron_adr downto 0) := (others => '0');

  type memory_weight is array (0 to input_neuron_num-training_neuron_num)
of signed(weights downto 0);
  signal Syn_Weight : memory_weight := (others => (others => '0'));

  type TYPES is (NP,P0);
  signal STATE: TYPES;

begin

  -- Synaptic Addr counter and Pre-Spike selector

  process (CLK)
  begin
    if (CLK='1' and CLK'event) then
      if (EN='1') then

```

```

        if EN_Addr='1' then
            if Syn_Addr = input_neuron_num-training_neuron_num
then
                Syn_Addr<=(others => '0');
            else
                Syn_Addr<=Syn_Addr+1;
            end if;
        end if;
    end if;
end if;
end process;

Pre_Spike<=Pre_Spikes(to_integer(Syn_Addr));
Addr <= std_logic_vector(Syn_Addr);

-- Pre-Spike Shift Register

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if(RST='1') then
            pre_shift_reg <= (others=>'0');
        elsif (EN='1') then
            pre_shift_reg(pre_reg) <= Pre_Spike;
            for I in pre_reg-1 downto 0 loop
                pre_shift_reg(I) <= pre_shift_reg(I+1);
            end loop;
        end if;
    end if;
end process;

-- Pre-Spike OR Gates

pre_gate <= or_reduce(pre_shift_reg);

-- Post-Spike Shift Register

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if(RST='1') then
            post_shift_reg <= (others=>'0');
        elsif (EN='1') then
            post_shift_reg(post_reg) <= Post_Spike;
            for I in post_reg-1 downto 0 loop
                post_shift_reg(I) <= post_shift_reg(I+1);
            end loop;
        end if;
    end if;
end process;

-- Post_Spike OR Gate

post_gate <= or_reduce(post_shift_reg);

-- I/D Sel

process (CLK)

```

```

begin
    if (CLK='1' and CLK'event) then
        if(RST='1') then
            incr_sel <= '0';
            decr_sel <= '0';
        elsif (EN='1') then
            if(Pre_Spike='1') then
                incr_sel <= '0';
                decr_sel <= '1';
            end if;
            if(Post_Spike='1') then
                decr_sel <= '0';
                incr_sel <= '1';
            end if;
        end if;
    end if;
end process;

-- Decr and Incr Gates

decr <= pre_gate AND (decr_sel AND post_gate);
incr <= pre_gate AND (incr_sel AND post_gate);

-- Synaptic Weight Counter

process (CLK)
begin
    if (CLK='1' and CLK'event) then
        if(RST='1') then
            Syn_Weight(to_integer(Syn_Addr))<=(others=>'0');
            WE <= '1';
        elsif(EN='1') then
            if(decr='1') then
                if Syn_Weight(to_integer(Syn_Addr))>-100 then
                    Syn_Weight(to_integer(Syn_Addr)) <=
Syn_Weight(to_integer(Syn_Addr)) - to_signed(1,weights+1);
                    WE <= '1';
                end if;
            elsif(incr='1') then
                if Syn_Weight(to_integer(Syn_Addr))<300 then
                    Syn_Weight(to_integer(Syn_Addr)) <=
Syn_Weight(to_integer(Syn_Addr)) + to_signed(1,weights+1);
                    WE <= '1';
                end if;
            else
                WE <= '0';
            end if;
        end if;
    end if;
end process;

Weight <= std_logic_vector(Syn_Weight(to_integer(Syn_Addr)));

end Behavioral;

```

## 2. Constraints

### 1.1. Nexys4 Master

```
## This file is a general .xdc for the Nexys4 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to
the top level signal names in the project

## Clock signal
##Bank = 35, Pin name = IO_L12P_T1_MRCC_35,          Sch name =
CLK100MHZ
set_property PACKAGE_PIN E3 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5.0}
[get_ports CLK]

set_property CFGBVS Vcco [current_design]
set_property config_voltage 3.3 [current_design]

set_input_delay -clock sys_clk_pin 1.0 [get_ports RST]
set_input_delay -clock sys_clk_pin 1.0 [get_ports BTN]
set_input_delay -clock sys_clk_pin 1.0 [get_ports SEL]
set_input_delay -clock sys_clk_pin 1.0 [get_ports Image]
set_input_delay -clock sys_clk_pin 1.0 [get_ports Neuron]
set_output_delay -clock sys_clk_pin -1.5 [get_ports Spikes_out]

##Buttons
##Bank = 15, Pin name = IO_L11N_T1_SRCC_15,          Sch name =
BTNC
set_property PACKAGE_PIN E16 [get_ports BTN]
    set_property IOSTANDARD LVCMOS33 [get_ports BTN]
##Bank = 15, Pin name = IO_L14P_T2_SRCC_15,          Sch name =
BTNU
set_property PACKAGE_PIN F15 [get_ports RST]
    set_property IOSTANDARD LVCMOS33 [get_ports RST]
##Bank = CONFIG, Pin name = IO_L15N_T2_DQS_DOUT_CSO_B_14, Sch name =
BTNL
set_property PACKAGE_PIN T16 [get_ports SEL]
    set_property IOSTANDARD LVCMOS33 [get_ports SEL]

## Switches
##Bank = 34, Pin name = IO_L21P_T3_DQS_34,          Sch name =
SW0
set_property PACKAGE_PIN U9 [get_ports {Image[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[0]}]
##Bank = 34, Pin name = IO_25_34,                  Sch name =
SW1
set_property PACKAGE_PIN U8 [get_ports {Image[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[1]}]
##Bank = 34, Pin name = IO_L23P_T3_34,              Sch name =
SW2
set_property PACKAGE_PIN R7 [get_ports {Image[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[2]}]
```

```

##Bank = 34, Pin name = IO_L19P_T3_34,                      Sch name =
SW3
set_property PACKAGE_PIN R6 [get_ports {Image[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[3]}]
##Bank = 34, Pin name = IO_L19N_T3_VREF_34,                  Sch name =
SW4
set_property PACKAGE_PIN R5 [get_ports {Image[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[4]}]
##Bank = 34, Pin name = IO_L20P_T3_34,                      Sch name =
SW5
set_property PACKAGE_PIN V7 [get_ports {Image[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[5]}]
##Bank = 34, Pin name = IO_L20N_T3_34,                      Sch name =
SW6
set_property PACKAGE_PIN V6 [get_ports {Image[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[6]}]
##Bank = 34, Pin name = IO_L10P_T1_34,                      Sch name =
SW7
set_property PACKAGE_PIN V5 [get_ports {Image[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[7]}]
##Bank = 34, Pin name = IO_L8P_T1_34,                      Sch name =
SW8
set_property PACKAGE_PIN U4 [get_ports {Image[8]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[8]}]
##Bank = 34, Pin name = IO_L9N_T1_DQS_34,                  Sch name =
SW9
set_property PACKAGE_PIN V2 [get_ports {Image[9]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Image[9]}]
##Bank = 34, Pin name = IO_L9P_T1_DQS_34,                  Sch name =
SW10
set_property PACKAGE_PIN U2 [get_ports {Neuron[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[0]}]
##Bank = 34, Pin name = IO_L11N_T1_MRCC_34,                Sch name =
SW11
set_property PACKAGE_PIN T3 [get_ports {Neuron[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[1]}]
##Bank = 34, Pin name = IO_L17N_T2_34,                    Sch name =
SW12
set_property PACKAGE_PIN T1 [get_ports {Neuron[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[2]}]
##Bank = 34, Pin name = IO_L11P_T1_SRCC_34,                Sch name =
SW13
set_property PACKAGE_PIN R3 [get_ports {Neuron[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[3]}]
##Bank = 34, Pin name = IO_L14N_T2_SRCC_34,                Sch name =
SW14
set_property PACKAGE_PIN P3 [get_ports {Neuron[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[4]}]
##Bank = 34, Pin name = IO_L14P_T2_SRCC_34,                Sch name =
SW15
set_property PACKAGE_PIN P4 [get_ports {Neuron[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Neuron[5]}]

##Pmod Header JA
##Bank = 15, Pin name = IO_L1N_T0_AD0N_15,                 Sch name =
JA1
set_property PACKAGE_PIN B13 [get_ports {Spikes_out[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[0]}]

```



```
##Bank = 15, Pin name = IO_L5N_T0_AD9N_15,          Sch name =
JA2
set_property PACKAGE_PIN F14 [get_ports {Spikes_out[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[1]}]
##Bank = 15, Pin name = IO_L16N_T2_A27_15,          Sch name =
JA3
set_property PACKAGE_PIN D17 [get_ports {Spikes_out[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[2]}]
##Bank = 15, Pin name = IO_L16P_T2_A28_15,          Sch name =
JA4
set_property PACKAGE_PIN E17 [get_ports {Spikes_out[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[3]}]

##Pmod Header JB
##Bank = 15, Pin name = IO_L15N_T2_DQS_ADV_B_15,    Sch name
= JB1
set_property PACKAGE_PIN G14 [get_ports {Spikes_out[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[4]}]
##Bank = 14, Pin name = IO_L13P_T2_MRCC_14,        Sch name =
JB2
set_property PACKAGE_PIN P15 [get_ports {Spikes_out[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Spikes_out[5]}]
```

### 3. Simulation Sources

#### 1.1. Top entity – Testbench

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   11:40:19 03/03/2017
-- Module Name:    Top - Testbench
-- Project Name:   Digital System for Neural Network Emulation
-----

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

ENTITY Testbench IS
Generic ( image_num : integer := 9;
         width : integer := 12;
         neuron_adr : integer := 5; -- Up to 32 neuron_adr
         weights : integer := 10; -- 255 downto -256
         input_neuron_num : integer := 40; -- Number of virtual neurons
+1
         training_neuron_num : integer := 6; -- Number of training
neurons
         neuron_num : integer := 46); -- Number of neurons +1
END Testbench;

ARCHITECTURE behavior OF Testbench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Top
    PORT(
        CLK : in STD_LOGIC;
        RST : in STD_LOGIC;
        BTN : in STD_LOGIC;
        SEL : in STD_LOGIC;
        Image : in STD_LOGIC_VECTOR(image_num downto 0);
        Neuron : in STD_LOGIC_VECTOR(neuron_num-input_neuron_num-1 downto
0);
        Spikes_out : out STD_LOGIC_VECTOR(neuron_num-input_neuron_num-1
downto 0));
    END COMPONENT;

    --Inputs
    signal CLK : std_logic := '0';
    signal RST : std_logic := '0';
    signal BTN : std_logic := '0';
    signal SEL : std_logic := '0';
    signal Image : std_logic_vector(image_num downto 0) := (others =>
'0');
    signal Neuron : std_logic_vector(neuron_num-input_neuron_num-1 downto
0) := (others => '0');

    --Outputs

```

```

    signal Spikes_out : STD_LOGIC_VECTOR(neuron_num-input_neuron_num-1
downto 0) := (others => '0');

    -- Clock period definitions
    constant CLK_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: Top PORT MAP (
        CLK => CLK,
        RST => RST,
        BTN => BTN,
        SEL => SEL,
        Image => Image,
        Neuron => Neuron,
        Spikes_out => Spikes_out
    );

    -- Clock process definitions
    CLK_process : process
    begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        RST<='1';
        wait for CLK_period*5;
        RST<='0';
        wait for 200us;
        Image <= (0=>'1',others => '0'); -- 0
        Neuron <= (0=>'1',others => '0');
        BTN<='1';
        wait for 10us;
        BTN<='0';

        wait for 600us;
        Image <= (1=>'1',others => '0'); -- 1
        Neuron <= (1=>'1',others => '0');
        BTN<='1';
        wait for 10us;
        BTN<='0';

        wait for 600us;
        Image <= (2=>'1',others => '0'); -- 2
        Neuron <= (2=>'1',others => '0');
        BTN<='1';
        wait for 10us;
        BTN<='0';

        wait for 600us;
        Image <= (3=>'1',others => '0'); -- 3

```

```

    Neuron <= (3=>'1',others => '0');
    BTN<='1';
wait for 10us;
    BTN<='0';

wait for 600us;
    Image <= (4=>'1',others => '0'); -- 4
    Neuron <= (4=>'1',others => '0');
    BTN<='1';
wait for 10us;
    BTN<='0';

wait for 600us;
    Image <= (5=>'1',others => '0'); -- 5
    Neuron <= (5=>'1',others => '0');
    BTN<='1';
wait for 10us;
    BTN<='0';

wait for 600us;
    SEL<='1';
    Neuron <= (others => '0');
    Image <= (0=>'1',others => '0');
wait for 400us;
    Image <= (1=>'1',others => '0');
wait for 400us;
    Image <= (2=>'1',others => '0');
wait for 400us;
    Image <= (3=>'1',others => '0');
wait for 400us;
    Image <= (4=>'1',others => '0');
wait for 400us;
    Image <= (5=>'1',others => '0');
wait for 400us;
    Image <= (others => '0');
    RST <='1';
wait for 20us;
    BTN<='1';

wait for 1000us;
    RST <='0';
    BTN<='0';
    Neuron <= (others => '0');
    Image <= (0=>'1',others => '0');
wait for 400us;
    Image <= (1=>'1',others => '0');
wait for 400us;
    Image <= (2=>'1',others => '0');
wait for 400us;
    Image <= (3=>'1',others => '0');
wait for 400us;
    Image <= (4=>'1',others => '0');
wait for 400us;
    Image <= (5=>'1',others => '0');
wait for 400us;
    Image <= (others => '0');

```

```
        wait;  
    end process;  
  
END;
```

## 1.2. Izhikevich Neuron – Testbench

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   14:27:19 02/17/2017
-- Module Name:   IZH_Neuron - Testbench
-- Project Name:  Digital System for Neural Network Emulation
-----

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

ENTITY Testbench IS
  GENERIC(
    number : integer := 0;
    width  : integer := 12;
    neuron_adr : integer := 5;
    weights : integer := 10);
END Testbench;

ARCHITECTURE behavior OF Testbench IS

  -- Component Declaration for the Unit Under Test (UUT)

  COMPONENT IZH_Neuron
  GENERIC(
    number : in integer;
    width  : in integer;
    neuron_adr : in integer;
    weights : in integer);
  PORT(
    CLK : in std_logic;
    RST : in std_logic;
    EN  : in std_logic;
    WE  : in std_logic;
    Addr : in std_logic_vector(neuron_adr downto 0);
    Weight : in std_logic_vector(weights downto 0);
    AER_Bus : in std_logic_vector(neuron_adr downto 0);
    Spike_out : out std_logic);
  END COMPONENT;

  --Inputs
  signal CLK : std_logic := '0';
  signal RST : std_logic := '0';
  signal EN  : std_logic := '0';
  signal WE  : std_logic := '0';
  signal Addr : std_logic_vector(neuron_adr downto 0) := (others =>
'0');
  signal Weight : std_logic_vector(weights downto 0) := (others => '0');
  signal AER_Bus : std_logic_vector(neuron_adr downto 0) := (others =>
'0');

  --Outputs
  signal Spike_out : std_logic;

```

```

-- Clock period definitions
constant CLK_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: IZH_Neuron
        GENERIC MAP (
            number => number,
            width => width,
            neuron_adr => neuron_adr,
            weights => weights)
        PORT MAP (
            CLK => CLK,
            RST => RST,
            EN => EN,
            WE => WE,
            Addr => Addr,
            Weight => Weight,
            AER_Bus => AER_Bus,
            Spike_out => Spike_out
        );

    -- Clock process definitions
    CLK_process :process
    begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        EN<='1';
        RST<='1';
        AER_Bus <= (others=>'1');
        wait for CLK_period*5;
        RST<='0';
        wait for 50 ns;
        WE<='1';
        Weight <= std_logic_vector(to_signed(120,Weight'length));
        Addr <= std_logic_vector(to_signed(1,Addr'length));
        wait for 50 ns;
        WE<='0';
        wait for 50 ns;
        AER_Bus <= std_logic_vector(to_signed(1,AER_Bus'length));
        wait;
    end process;

END;
```

### 1.3. AER Bus – Testbench

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   11:40:19 03/03/2017
-- Module Name:   AER_Bus - Testbench
-- Project Name:  Digital System for Neural Network Emulation
-----

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

ENTITY Testbench IS
Generic ( neuron_adr : integer := 4;    -- Up to 32 neuron_adr
         neuron_num  : integer := 4);  -- Number of neurons +1
END Testbench;

ARCHITECTURE behavior OF Testbench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT AER_Bus
    GENERIC(
        neuron_adr : in integer;
        neuron_num : in integer);
    PORT(
        CLK : in STD_LOGIC;
        Spikes : in STD_LOGIC_VECTOR(neuron_num downto 0);
        EN_Neuron : out STD_LOGIC;
        AER : out STD_LOGIC_VECTOR(neuron_adr downto 0));
    END COMPONENT;

    --Inputs
    signal CLK : std_logic := '0';
    signal Spikes : std_logic_vector(neuron_num downto 0) := (others =>
'0');

    --Outputs
    signal EN_Neuron : STD_LOGIC := '0';
    signal AER : STD_LOGIC_VECTOR(neuron_adr downto 0) := (others => '0');

    -- Clock period definitions
    constant CLK_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: AER_Bus
    GENERIC MAP (
        neuron_adr => neuron_adr,
        neuron_num => neuron_num)
    PORT MAP (
        CLK => CLK,
        Spikes => Spikes,
        EN_Neuron => EN_Neuron,

```



```
        AER => AER
    );

-- Clock process definitions
CLK_process :process
begin
    CLK <= '0';
    wait for CLK_period/2;
    CLK <= '1';
    wait for CLK_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    Spikes <= "00000";
    wait for 10 ns;
    Spikes <= "00001";
    wait for CLK_period;
    Spikes <= "00000";
    wait for 20 ns;
    Spikes <= "11011";
    wait for CLK_period;
    Spikes <= "00000";
    wait for CLK_period;
    Spikes <= "00110";
    wait for CLK_period;
    Spikes <= "00000";
    wait;
end process;

END;
```

## 1.4. STDP – Testbench

```

-----
-- Engineer:      Eduard-Guillem Merino Mallorquí
-- Create Date:   14:15:20 05/05/2017
-- Module Name:   STDP - Testbench
-- Project Name:  Digital System for Neural Network Emulation
-----

```

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;

```

```

ENTITY Testbench IS
Generic ( neuron_adr : integer := 4;
          weights    : integer := 10;
          input_neuron_num : integer := 2;
          training_neuron_num : integer := 0;
          pre_reg     : integer := 15;
          post_reg    : integer := 5);
END Testbench;

```

```

ARCHITECTURE behavior OF Testbench IS

```

```

    -- Component Declaration for the Unit Under Test (UUT)

```

```

    COMPONENT STDP

```

```

    GENERIC(

```

```

        neuron_adr : in integer;
        weights    : in integer;
        input_neuron_num : in integer;
        training_neuron_num : in integer;
        pre_reg     : in integer;
        post_reg    : in integer);

```

```

    PORT( CLK : in STD_LOGIC;

```

```

          RST : in STD_LOGIC;

```

```

          EN : in STD_LOGIC;

```

```

          EN_Addr : in STD_LOGIC;

```

```

          Pre_Spikes : in STD_LOGIC_VECTOR(input_neuron_num-
training_neuron_num downto 0));

```

```

          Post_Spike : in STD_LOGIC;

```

```

          WE : out STD_LOGIC;

```

```

          Addr : out STD_LOGIC_VECTOR(neuron_adr downto 0);

```

```

          Weight : out STD_LOGIC_VECTOR (weights downto 0));

```

```

    END COMPONENT;

```

```

--Inputs

```

```

signal CLK : std_logic := '0';

```

```

signal RST : std_logic := '0';

```

```

signal EN : std_logic := '0';

```

```

signal EN_Addr : std_logic := '0';

```

```

signal Pre_Spikes : std_logic_vector(input_neuron_num-
training_neuron_num downto 0) := (others => '0');

```

```

signal Post_Spike : std_logic := '0';

```

```

--Outputs

```



```

signal WE : STD_LOGIC := '0';
signal Addr : STD_LOGIC_VECTOR(neuron_adr downto 0) := (others =>
'0');
signal Weight : STD_LOGIC_VECTOR(weights downto 0) := (others => '0');

-- Clock period definitions
constant CLK_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: STDP
    GENERIC MAP (
        neuron_adr => neuron_adr,
        weights => weights,
        input_neuron_num => input_neuron_num,
        training_neuron_num => training_neuron_num,
        pre_reg => pre_reg,
        post_reg => post_reg)
    PORT MAP (
        CLK => CLK,
        RST => RST,
        EN => EN,
        EN_Addr => EN_Addr,
        Pre_Spikes => Pre_Spikes,
        Post_Spike => Post_Spike,
        WE => WE,
        Addr => Addr,
        Weight => Weight
    );

    -- Clock process definitions
    CLK_process :process
    begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        RST <= '1';
        wait for 10 ns;
        RST <= '0';
        EN <= '1';
        wait for CLK_period;
        Pre_Spikes <= "101";
        wait for CLK_period;
        Pre_Spikes <= "000";
        wait for CLK_period*9;
        Post_Spike <= '1';
        wait for CLK_period;
        Post_Spike <= '0';

        wait for 70 ns;
    end process;

```

```
    EN_Addr <= '1';
    wait for CLK_period;
    EN_Addr <= '0';
    Pre_Spikes <= "101";
    wait for CLK_period;
    Pre_Spikes <= "000";
    wait for CLK_period*11;
    Post_Spike <= '1';
    wait for CLK_period;
    Post_Spike <= '0';

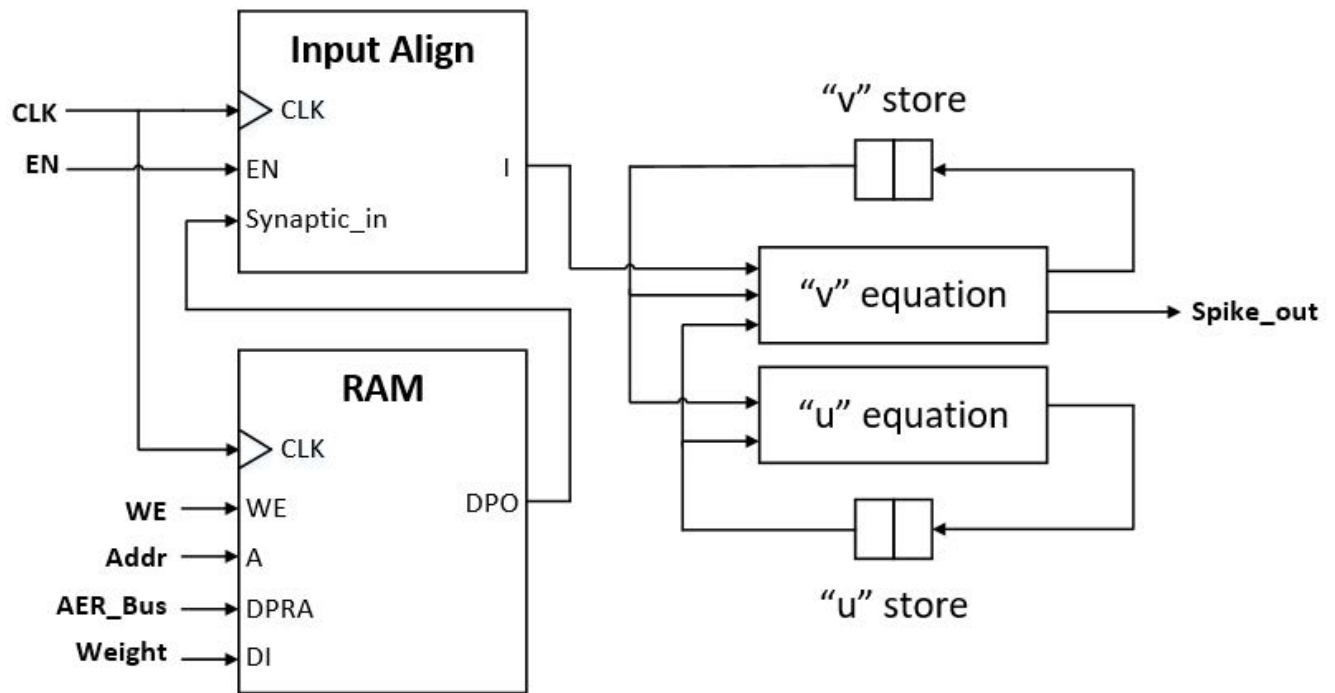
    wait for 70 ns;
    EN_Addr <= '1';
    wait for CLK_period;
    EN_Addr <= '0';
    Pre_Spikes <= "101";
    wait for CLK_period;
    Pre_Spikes <= "000";
    wait for CLK_period*11;
    Post_Spike <= '1';
    wait for CLK_period;
    Post_Spike <= '0';

    wait;
end process;

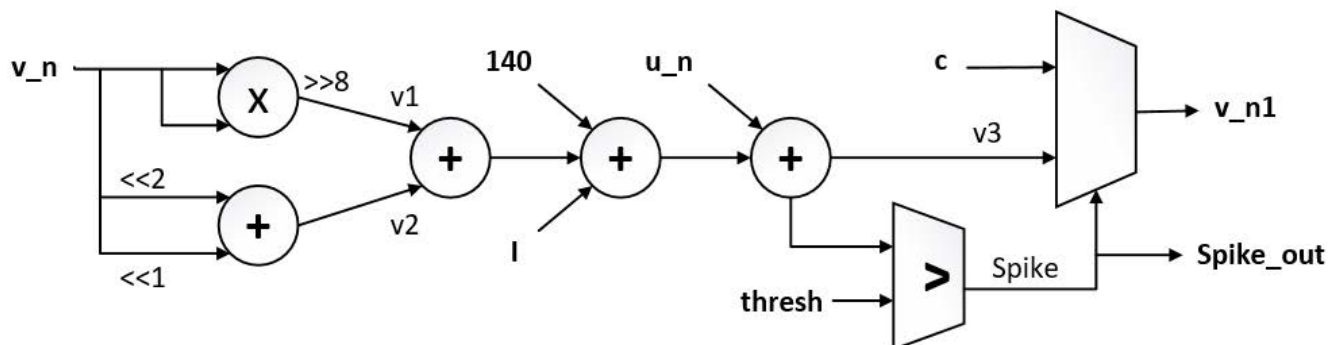
END;
```

## **A3. Blueprints**

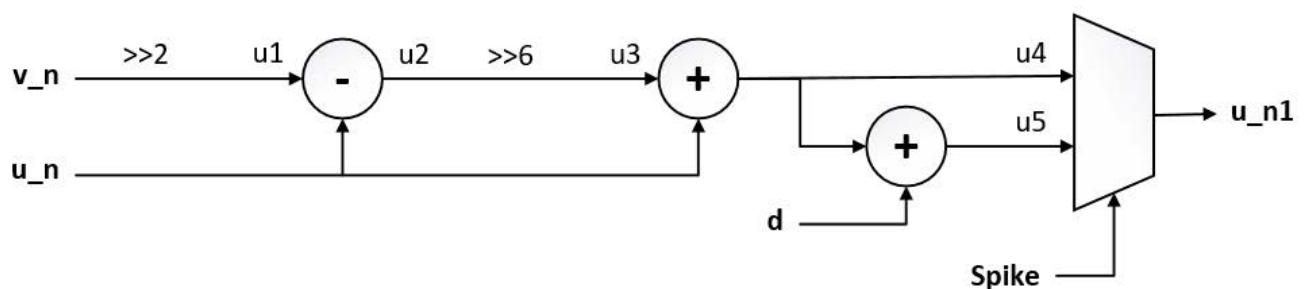





### "v" Equation implementation

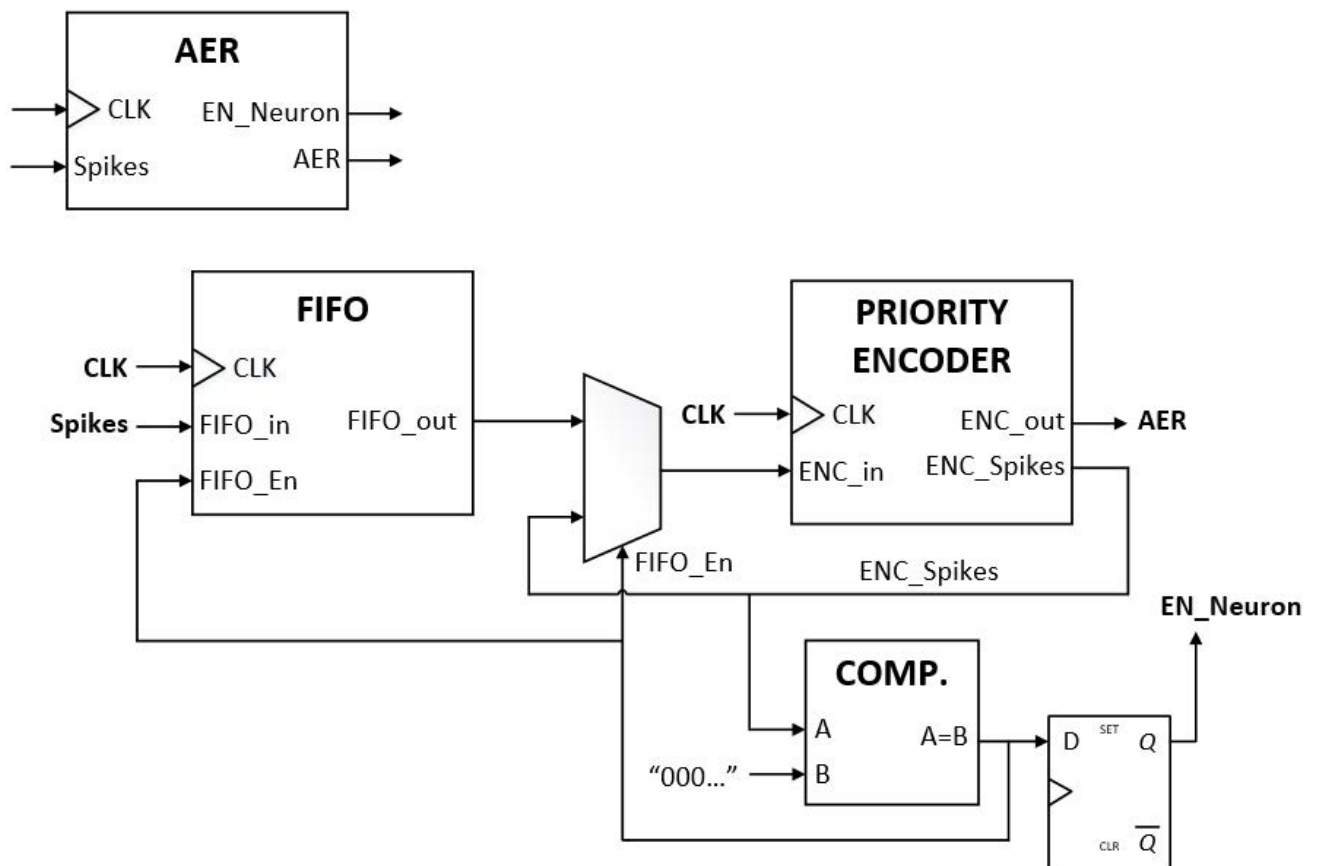



### "u" Equation implementation



Unit  mm		Name	Date	Digital System For Spiking Neural Network Emulation	
	Created	Eduard Merino	30/04/17		
	Revision	Eduard Merino	30/04/17		
Scale  1:1	Drawing Title  Digital diagram of a Neuron			Drawing No.  1	 UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH Escola d'Enginyeria de Barcelona Est

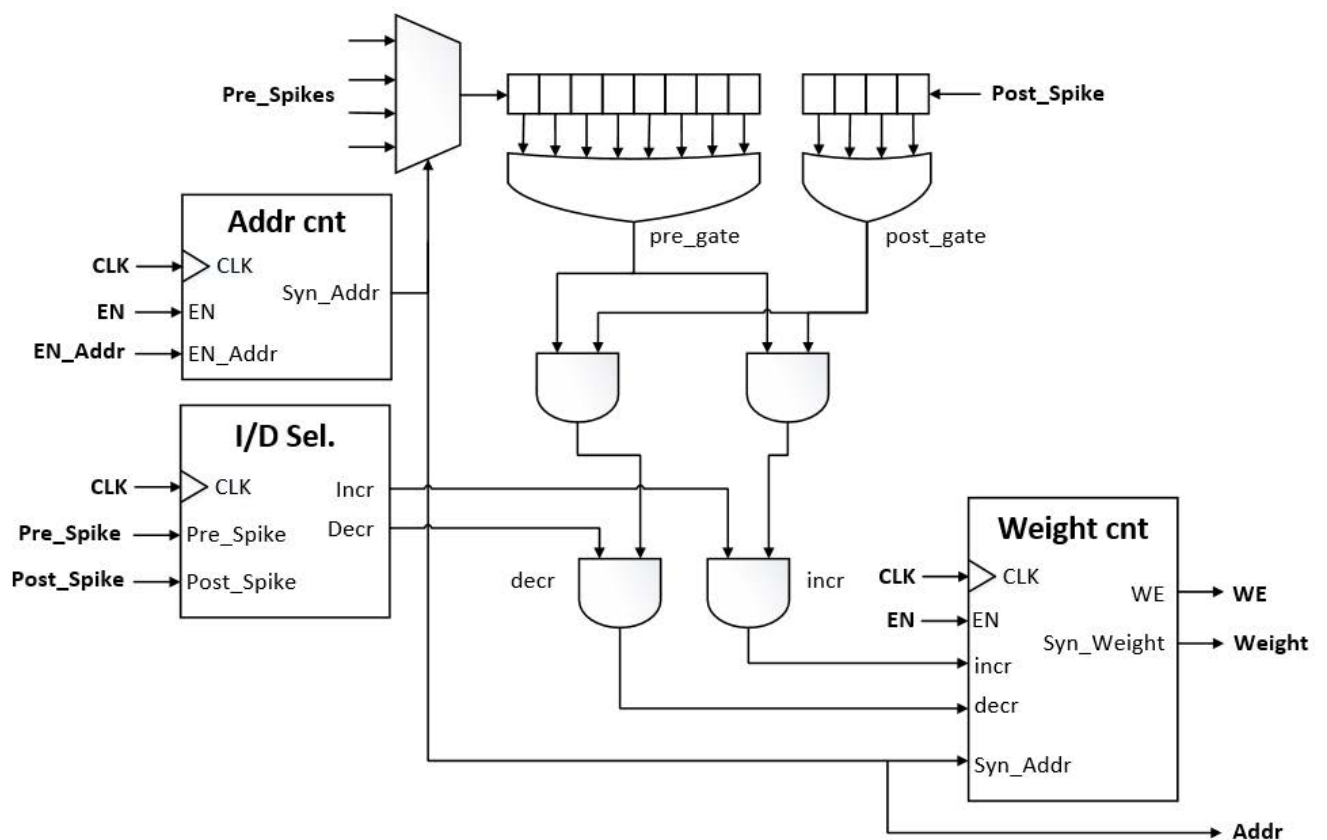
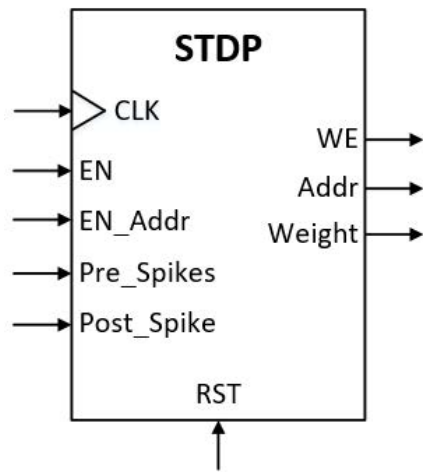




Unit mm		Name	Date	Digital System For Spiking Neural Network Emulation	
	Created	Eduard Merino	30/04/17		
	Revision	Eduard Merino	30/04/17		
Scale	Drawing Title			Drawing No.	 UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH Escola d'Enginyeria de Barcelona Est
1:1	Digital diagram of the AER system			2	



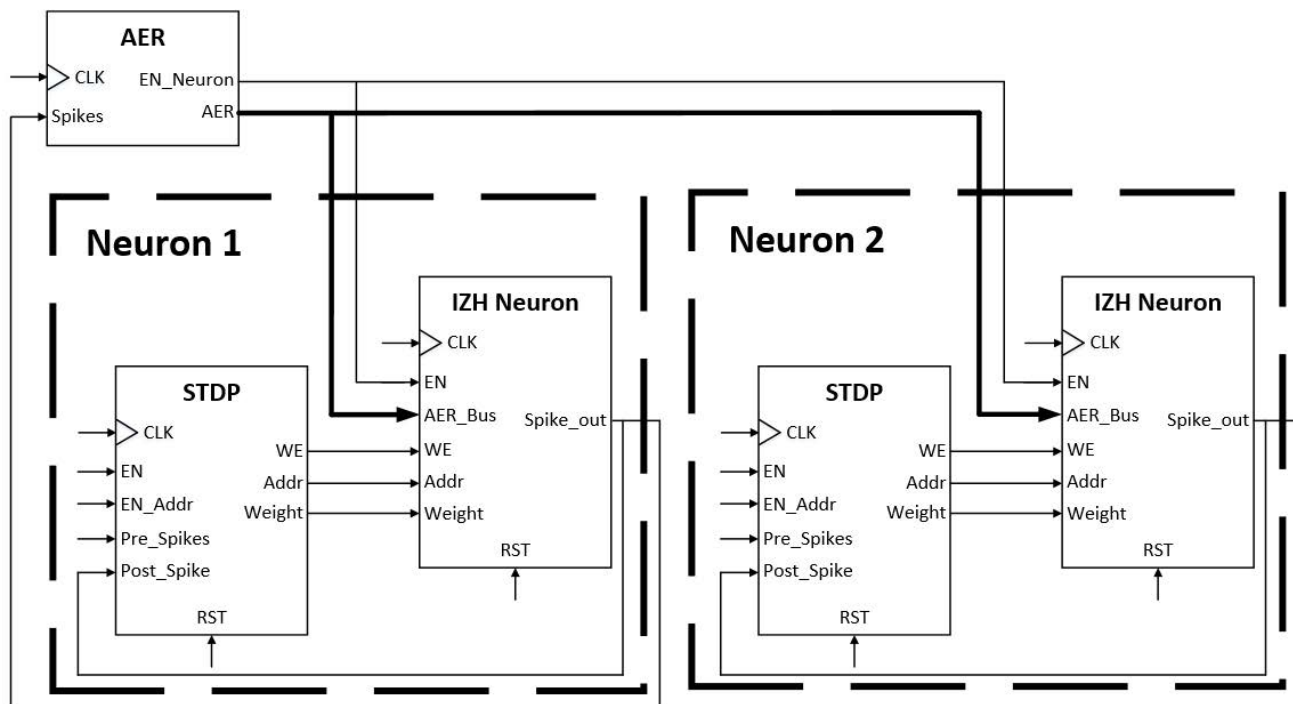




Unit mm		Name	Date	Digital System For Spiking Neural Network Emulation
	Created	Eduard Merino	30/04/17	
	Revision	Eduard Merino	30/04/17	
Scale 1:1	Drawing Title Digital diagram of the STDP			Drawing No. 3



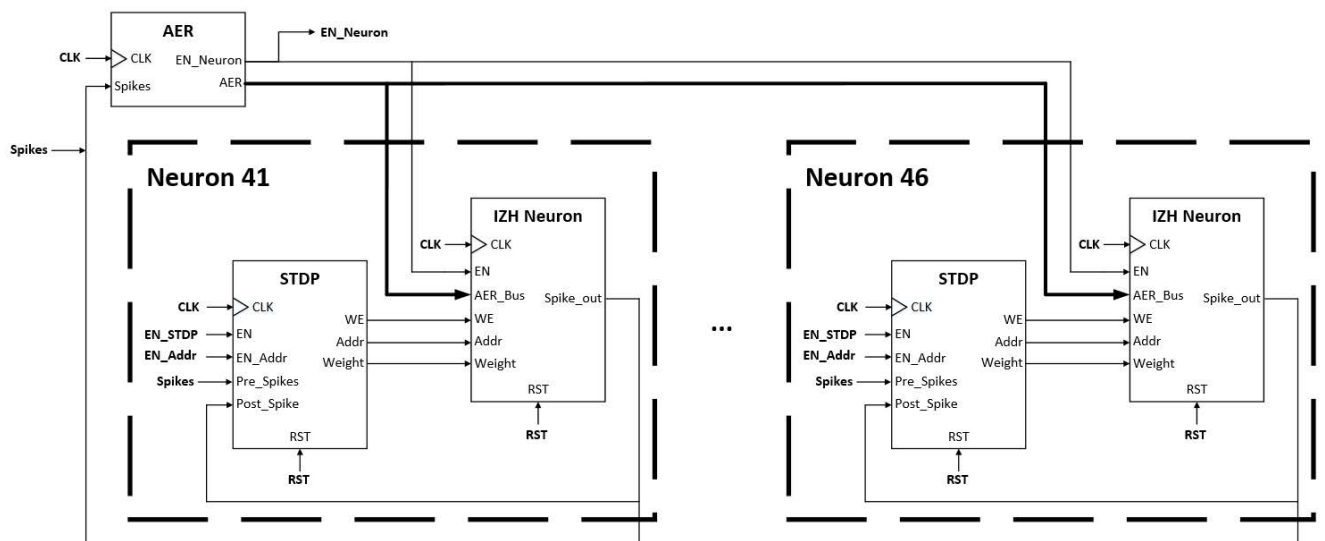
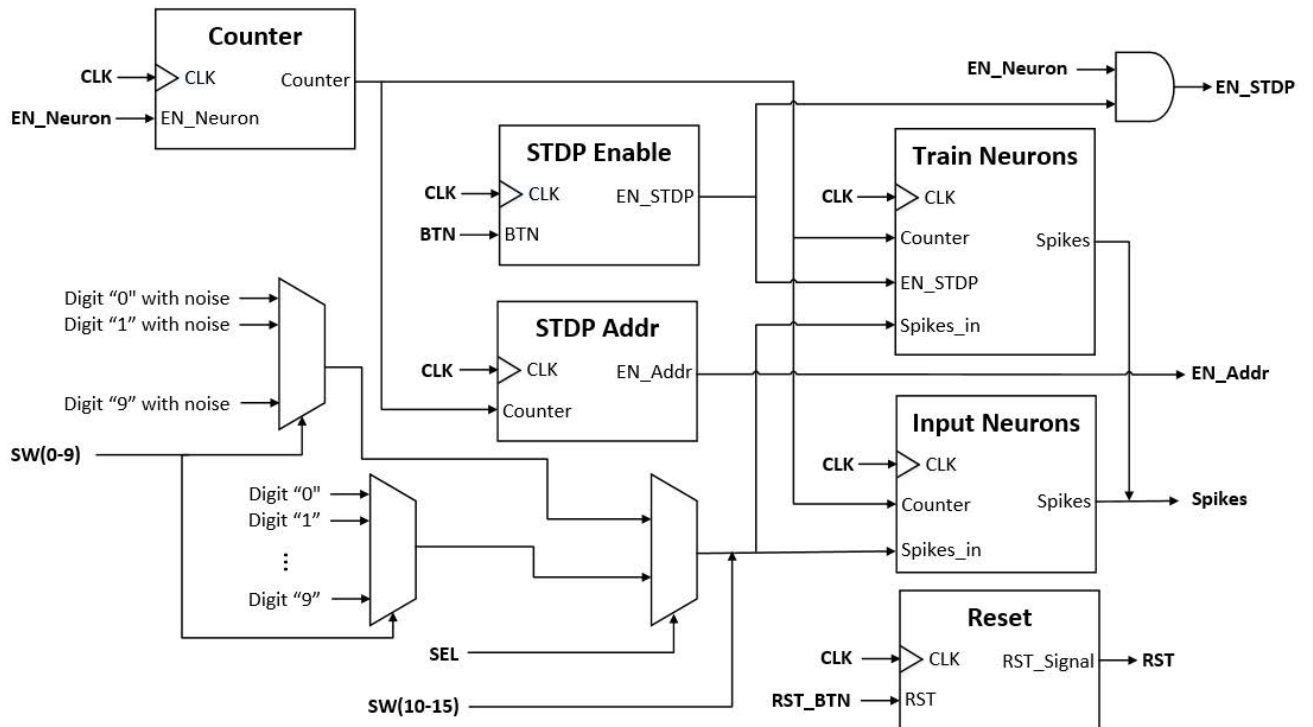




Unit mm		Name	Date	Digital System For Spiking Neural Network Emulation
	Created	Eduard Merino	30/04/17	
	Revision	Eduard Merino	30/04/17	
Scale 1:1	Drawing Title Digital diagram of a SNN			Drawing No. 4







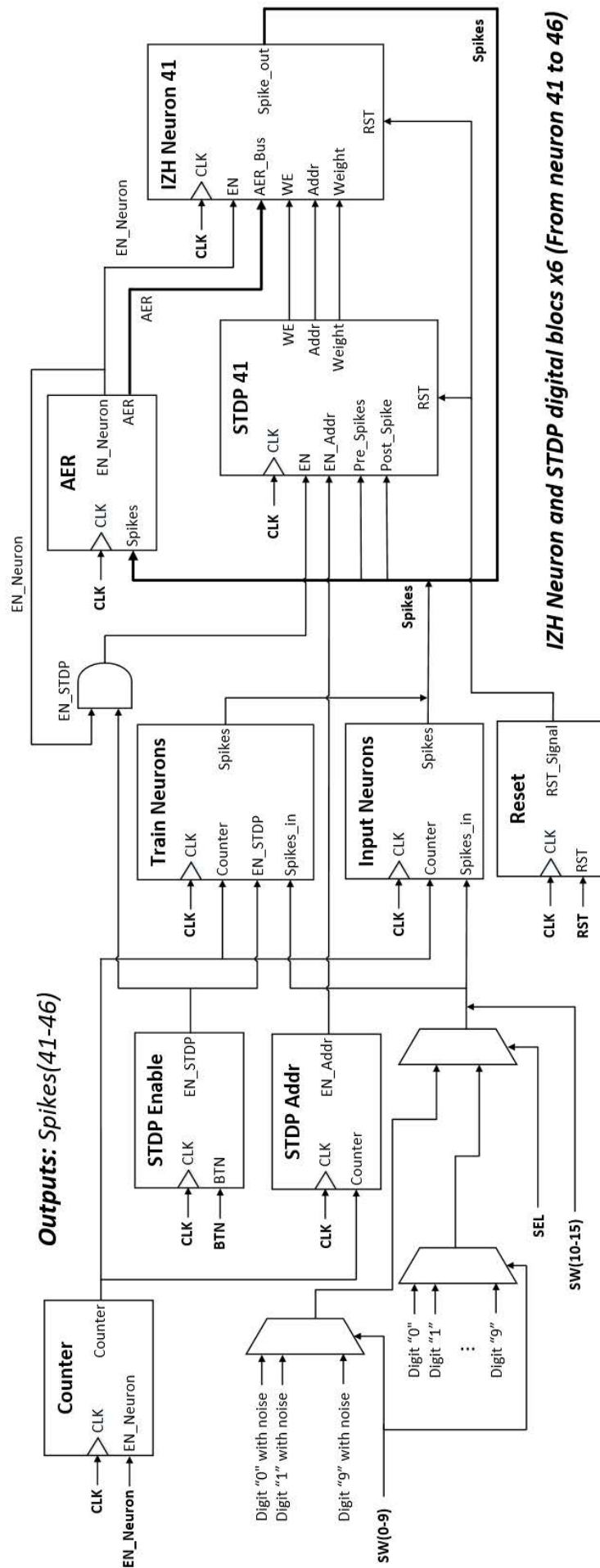
Unit mm		Name	Date	Digital System For Spiking Neural Network Emulation
	Created	Eduard Merino	30/04/17	
	Revision	Eduard Merino	30/04/17	
Scale	Drawing Title			Drawing No.
1:1	Digital diagram of a SNN for pattern recognition			5






**Inputs:** CLK, RST, BTN, SEL, SW(0-15)

**Outputs:** Spikes(41-46)



**IZH Neuron and STDP digital blocs x6 (From neuron 41 to 46)**

Unit mm		Name	Date	Digital System For Spiking Neural Network Emulation	
	Created	Eduard Merino	30/04/17		
	Revision	Eduard Merino	30/04/17		
Scale 1:1	Drawing Title Digital diagram of a SNN for pattern recognition			Drawing No. 6	 UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH Escola d'Enginyeria de Barcelona Est



