UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Enginyeria Electrònica

Efficient multiprocessing architectures for Spiking Neural Network emulation based on configurable devices

Tesi doctoral presentada per a l'obtenció del títol de Doctor per la Universitat Politècnica de Catalunya, dins el Programa de Doctorat en Enginyeria Electrònica

Giovanny Sánchez Rivera

Director: Jordi Madrenas Boadas

June, 2014

Abstract

The exploration of the dynamics of bio-inspired neural networks has allowed neuroscientists to understand some clues and structures of the brain. Electronic neural network implementations are useful tools for this exploration. However, appropriate architectures are necessary due to the extremely high complexity of those networks. There has been an extraordinary development in reconfigurable computing devices within a short period of time especially in their resource availability, speed, and reconfigurability (FPGAs), which makes these devices suitable to emulate those networks.

Reconfigurable parallel hardware architecture is proposed in this thesis in order to emulate in real time complex and biologically realistic spiking neural networks (SNNs). Some relevant SNN models and their hardware approaches have been studied, and analyzed in order to create an architecture that supports the implementation of these SNN models efficiently. The key factors, which involve flexibility in algorithm programmability, high performance processing, low area and power consumption, have been taken into account. In order to boost the performance of the proposed architecture, several techniques have been developed: time to space mapping, neural virtualization, flexible synapse-neuron mapping, and specific learning and execution modes, among others.

Besides this, an interface unit has been developed in order to build a bio-inspired system, which can process sensory information from the environment. The spiking-neuron-based system combines analog and digital multi-processor implementations. Several applications have been developed as a proof-of-concept in order to show the capabilities of the proposed architecture for processing this type of information.

Resumen

El estudio de la dinámica de las redes neuronales bio-inspiradas ha permitido a los neurocientíficos entender algunos procesos y estructuras del cerebro. Las implementaciones electrónicas de estas redes neuronales son herramientas útiles para llevar a cabo este tipo de estudio. Sin embargo, ls alta complejidad de las redes neuronales requiere de una arquitectura apropiada que pueda simular este tipo de redes. Emular este tipo de redes en dispositivos configurables es adecuado debido a su extraordinario desarrollo en cuanto a recursos, velocidad y capacidad de reconfiguración (FPGAs).

En esta tesis se propone una arquitectura hardware paralela y configurable para emular las complejas y realistas redes neuronales tipo spiking en tiempo real. Se han estudiado y analizado algunos modelos de neuronas tipo spiking relevantes y sus implementaciones en hardware, con el fin de crear una arquitectura que soporte la implementación de estos modelos de manera eficiente.

Se han tenido en cuenta varios factores clave, incluyendo flexibilidad en la programación de algoritmos, procesamiento de alto rendimiento, bajo consumo de energía y área. Se han aplicado varias técnicas en la arquitectura desarrollada con el propósito de aumentar su desempeño. Estas técnicas son: mapeado de tiempo a espacio, virtualización de las neuronas, mapeo flexible de neuronas y sinapsis, modos de ejecución, y aprendizaje específico, entre otras.

Además, se ha desarrollado una unidad de interfaz de datos con el fin de construir un sistema bio-inspirado, que puede procesar información sensorial del medio ambiente. Este sistema basado en neuronas tipo spiking combina implementaciones analógicas y digitales. Varias aplicaciones se han desarrollado usando este sistema como prueba de concepto, con el fin de mostrar las capacidades de la arquitectura propuesta para el procesamiento de este tipo de información.

I am very grateful with my parents Elpidio, Paulina, and my brother Humberto, who always have been supporting me and they never let me down, and my wife Albina and my daughter Angelina for encouraging throughout my life

Acknowledgments

First of all, I would like to thank all people who have been participated with me in the development of this project. It is very difficult to write it in few lines how they were important in this project.

I would like to thank Dr. Jordi Madrenas Boadas, who is the director of this thesis, for his supporting, and guiding me in the development of this thesis. I am very grateful with him for allowing me to do this thesis, also for his friendship. His valuable help has allowed me to develop this thesis and improve several professional aspects.

I would like to thank Dr. Alister Hamilton, and Dr. Thomas Jacob Koickal for their support, guidance during my research stage at *Institute for Micro and Nano Systems*, *School of Engineering and Joint Research Institute for Integrated Systems*, University of Edinburgh, Edinburgh, UK. The contribution with new ideas and observations given by Dr. Alister and Dr. Thomas has allowed improving this work.

I would like to thank to Adam Sokolnicki, Karthikeyan k., Salvo cambria, Vito pirrone for participating in the development of this thesis.

I would like to thank to Athul sripad for his important contributions in the development of this thesis. I am so grateful for his valuable help in the development and for his friendship.

I would like to thank to Sanjana Sekar for checking m~~y speling~~g spelling mistakes to improve my thesis and for her valuable friendship.

I would like to thank to Piotr Michalik, besides being my collage, he is my friend. I am so grateful with his help in several issues and friendship.

Quiero dar las gracias a mi amiga Rosario Zarate Maldonado, por su apoyo emocional que me ha brindado durante mi estancia en España.

I would like to thank to the Catalan Department of Innovation, Universities and Companies, and the European Social Fund for sponsoring this work.

Agradezco al Consejo de Ciencia y Tecnologia CONACyT por patrocinio para la realización de este proyecto de tesis.

Contents

List of figures

List of Tables

Introduction

1

1.1 Problem Statement

For several decades the anatomy of the human brain has fascinated many neuroscientists and engineers due to its complex functions and structures. Some of the functions performed by the brain are associated with reasoning, speech recognition, movement, visual processing [1]. These biological processes are carried out through the cells known as 'neurons' which constitutes the fundamental part of the brain. The neuron has the ability to propagate signals rapidly over large distances. Basically, these signals are electrical pulses called action potentials or, more simply, spikes, which travel through nerve fibers. The information in the neurons is represented by firing sequences of spikes in various temporal patterns. These patterns provide information to perform functions such as detection of the light, detection of the sound intensity, or motor actions in the form of action potentials [1]. The structure of a neuron can be divided into three parts where each part is analogous to the input (dendrites), processing unit (soma) and output of the system (axon), as is shown in Fig. 1.1. Basically, the structure of a neuron can be divided into three parts where each part is analogous to the input (dendrites), processing unit (soma) and output of the system (axon), as is shown in Fig. 1.1. Basically, the soma or cell body can be considered as the Central Processing unit of the neuron where all the information processing is done. The transfer of information from one neuron to another takes place through the synapse connection typically by means of a discharge of a chemical neurotransmitter. The neurotransmitters received from the dendrites are received in the soma producing a potential. If the increased potential in the soma is large enough to cross a threshold level, then an electrical pulse is produced. The pulse travels through the axon, activating the synapses of the other neurons. Pre-Synaptic is a term that is used for the transmitter neuron whereas the receiver neurons are called as Post-synaptic. The ability to change the strength or weakness of the excitatory or inhibitory synapses is called plasticity. This is one of the important mechanisms which are linked to the learning and memory processes in the human brain. Neuroscientists assume that this process is due to the change of the strength or weakness of the synapses [2]. This strength or weakness changes according to the response to the activity of both pre and post synaptic neurons. Alterations in the synaptic transmission can be roughly subdivided into two classes of mechanisms: long-term potentiation (LTP), and long-term depression (LTD), LTP is measured as a persistent increase in the amplitude of the excitatory postsynaptic potentials (EPSP), whereas LTD is measured as the persistent decrease in the amplitude of the EPSPs.

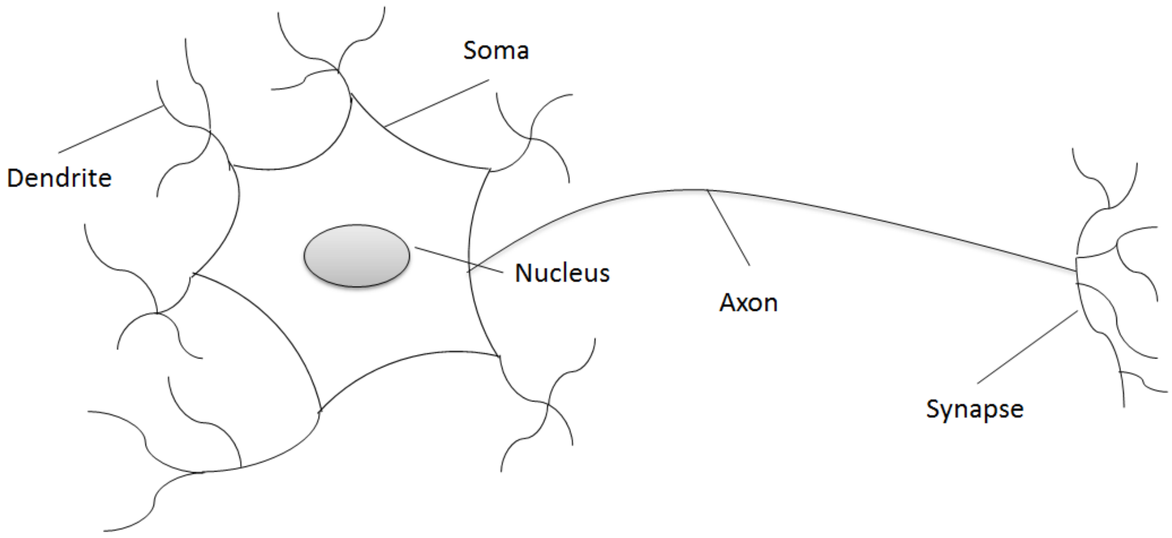


Figure 1.1: A single neuron in the schematic view. Dendrites, soma, and axon are the main components of the neuron and can be clearly distinguished.

In this present era, there has been a lot of research on the modelling of the cortex which is one of the vital parts of the human brain. This part performs some important functions such as thinking, information processing, perception, etc. The cortical models, also known as Spiking Neural Network (SNN) models intend to mimic the biological neuron by making the neural simulation real time. This is because the concept of the time is inherent in these types of models. So that, the representation of the information is through spikes, which makes the SNN model more biologically plausible. Therefore, Spiking neurons differ from conventional Artificial Neural Networks (ANN) models as information is transmitted by means of spikes rather than firing rates. The ANNs are computational models inspired by the structure and functionality of biological neurons. The ANNs have been classified in three generations according to the level of realism in the neural simulation. The first generation of ANN models involves binary networks (activation of 0 or 1). The second generation takes into account the networks where the activation is the representative of 'mean firing rate' of the neuron. These types of networks have been implemented in analogue devices where the firing rate values can be between 0 and 1. These values are normalized and hence it is also known as rate-coding scheme. It is called so because this implies a mechanism for averaging. The SNN models are classified as the third generation of these models, which includes the concept of time in their operating model. The neuron fires only when the membrane potential crosses a certain threshold producing a spike. This spike increases or decreases the potential of other neurons in accordance with this signal. For several years, conventional ANNs (second generation) have demonstrated their best performance as engineering tools and in many other domains like pattern recognition, control, bio-informatics, and robotics. Nevertheless, these models suffer from intrinsic limitations such as processing enormous amount of data or the ability to adapt to the changes in the environment. This is because of several characteristics, such as iterative learning algorithms or artificially designed neuron model and network architecture, that are strongly limited in comparison with the processes carried out in biological neural networks.

The development of neural network models has been progressing according to two vital applications as shown in Fig. 1.2. The first one is mainly given to the development of engineering applications where the efficiency of the model is given the highest priority without focusing on the biological process taking place inside the human brain. The second type of application is mainly dedicated for the simulation and modelling of the behaviour of the brain.

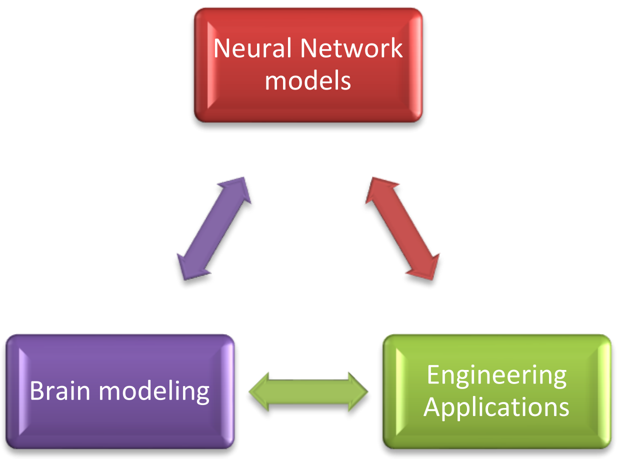


Figure 1.2: The development of Neural Networks models

A wide variety of SNN models have been proposed for several years. Most of them model the ion channels that are responsible for generating the spikes at the axon hillock. The most relevant works presented here is due to its importance in the modelling of the neurons, and its widespread use in several applications. One such SNN model was proposed by Hodgkin-Huxley [3]. This model describes the conductance-based neuron by reproducing electrophysiological measurements to a very high degree of accuracy [4]. Unfortunately, this model is very complex, usually difficult to analyze and computationally expensive in numerical implementations. Other SNN models which were derived from Hodgkin-Huxley model are integrate-and-fire model and Leaky-Integrate and fire model. These two models are extensively used in many applications, mostly in applications pertaining to processing of time-varying signals [5].These models are simple to understand, easy to implement and commonly used in the networks designed using spiking neurons. Apart from these, there have been a lot of efforts being put to propose SNN models with different properties. One such neural model was proposed by Izhikevich [6]. This model combines the biologically plausibility of the Hodgkin–Huxley-type dynamics and the computational efficiency of integrate-and-fire neurons to reproduce the spiking behavior of many neurons dynamics. The Bifurcation methodologies enable the author to reduce many biophysically accurate Hodgkin–Huxley-type neuronal models into a two-dimensional (2-D) system of ordinary differential equations. These equations have the ability to handle about 20 fundamental neuro-computational properties of biological neurons [6]. Finally, the SNN model, which was proposed by Iglesias and Villa [7] during the development of the European PERPLEXUS project [8], has served as a benchmark for this thesis. In this SNN model the neuron is modelled as a simple leaky integrate-and-fire model which makes it computationally efficient and as an important functional aspect of the cortical circuits. In particular, the spike-timing-dependent synaptic plasticity (STDP) of the synapses is modeled according to the learning rule suggested by Donald Hebb [2].

The computer simulation of the human brain has been considered as an important tool to understand its structure and dynamics. After simulating several large-scale SNN models, neuroscientists have tried to test their computational model hypothesis of the brain structure, dynamics and function by interacting with the real or virtual environment. The extraordinary performance of the human brain has propelled many engineers to design architectures that imitate the functionality of the brain in order to create bio-inspired machines or robots that can mimic these functions to solve complex perception problems as learning visual features, character recognition, and autonomous navigation. Also, this performance in the human brain imposes big challenges in the development of cortical emulators regarding several factors like: the complexity of the neural behaviour, the scale of the network, interconnection, plasticity in the synapses, and power consumption. The brain exhibits high performance in terms of processing speed, so that it could be considered as an interesting computer, because it performs approximately 3.6×10^{15} synaptic operations per second. Moreover, it is highly efficient in terms of power consumption (12 Watts). Even though the neurons work slowly, there is a tremendous amount of computation achieved in real time. Due to this, several questions are cropping up regarding the efficiency of the brain. The physical structure and composition of each neuron membrane in the design of the human brain, the wiring plans between the neurons and the astounding capability to learn are some of the reasons to justify the efficiency of the brain.

The simulation of large scale Spiking Neural Network models has been performed with different approaches which mainly involve supercomputers, general purpose computers, analogue circuits, multiprocessors, graphics processing units and field-programmable gate arrays. These approaches have been proposed as suitable platforms to be used to simulate such SNN models in order to explore the neural dynamics involved in the SNN models. However there are several aspects to be discussed and analysed in order to create this platform which involves the model flexibility, architectural scalability, power consumption, and area consumption.

- ***Supercomputers and general purpose computers***

During the last years, several projects have been proposed to create cortical simulators which try to emulate large-scale SNN models. They intend to emulate millions of neurons and billions of synapses. One of the most important emulator is implemented on a supercomputer. This project is the ‘Blue brain project’ which was started in 2005 at École Polytechnique Fédérale de Lausanne (EPFL) [9]. Several aspects of the brain were modelled and verified by EPFL. Now, attempts are being made to simulate the whole brain [10]. The simulation of thousands of neurons was done in the Blue brain project at ion-channel level which was implemented with the help of the details given in [9]. And billions of synapses were modelled with the help of non-linear differential equations [11]. But, the implementation of the neurons and synapses as per the Blue Brain project requires a high amount of power consumption (around 8.4 GW) [12]. The power consumption and size limitations are some of the ordeals this project faces right now. But there were certain ways to overcome them. The project focussed only in the reproduction of the dynamics of the membrane potential, which in turn reduced the energy as well as the time required for performing the simulation.

The software approach involves developing codes and algorithms for general purpose computers/High Performance Computing (HPC) modelling the neural behaviour. This is the approach adopted in the blue brain project in terms of HPCs and several others using general purpose computers. In case of conventional processors, the memory bandwidth and parallelism is minimal to be able to implement large scale real time simulation of complex spiking neural networks.

- *Digital implementations*

As mentioned before, in digital domain several works have been proposed to develop cortical emulators in compact digital devices such as multiprocessors, Graphic Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs). The important features of these digital implementations are flexibility, scalability and re-configurability. Although the digital implementation is less compact when compared to analog implementation, the cost of analog designs is much higher in comparison with digital or software implementations. Therefore, some research groups are aiming to make an SNN emulator which is flexible and scalable in order to offer a system which can be used as a platform for experimental research, which involves the study of the neural dynamics in certain part of the brain.

. *Multiprocessor units*

One the most representative work which has been developed a SNN emulator based on multiprocessor is proposed in [13]. This multiprocessor called SpiNNaker intends to simulate billion neurons and trillion-synapses using a network of 50K SpiNNaker chips. Each SpiNNaker chip is composed of 18 identical ARM processors with custom interconnections. Its programmable feature allows SpiNNaker to support different SNN models. This customized architecture promises to be a powerful platform in the simulation of large-scale of SNN models. However, there are some aspects to discuss about this architecture. One of them is regarding the communication system. They assume that the network will not be saturated and there is no mechanism of congestion. This mechanism of congestion is vital when large networks become more active. Another aspect is regarding the memory system of SpiNNaker. A large amount of data is transferred from the external memory to the processors. The high bandwidth memory data interfaces compensate the negative effect of transfer of data from the external memory to the processors.

. Graphic Processing Units

There have been efforts to create SNN emulators based on GPU implementations to simulate large-scale SNN models [14, 15]. Currently, the GPUs are becoming very popular because these types of digital devices offer excellent parallel computation, due to its inherent parallelization. However, despite having a parallel architecture it could be an excellent platform for the simulation of large-scale SNN models. This is because the calculation of the membrane potential of every spiking neuron exhibits significant parallel computation. There are many factors which reduce the performance of the simulation of these SNN models in GPUs architectures. However, many of these works have reported that the simulation of the SNN models on GPUs is around of 10 to 100 times faster when compared to the simulations on general purpose computers. The method used to measure the performance of the SNN model simulations on this type of digital device is not provided in [14, 15]. One of the factors which decrease the performance is related to the memory bandwidth [14]. This presents an important problem to the GPUs due to the enormous rate at which the processors read data from the memory or load data to the memory, which was reported in [14]. The other is related to the optimization of the parallel execution, which depends on the number of threads and the memory system in every GPU. These threads, which are in charge of transferring data from the memory to the processors or vice versa, are limited in number, so that, not all cores are used [14, 15]. The threads have been programmed with high level of complexity in order to exploit maximum number of processors that are available in the GPU. This solution limits the use of the algorithm for the simulation of the SNN model to a specific GPU architecture. The flexibility and scalability are the two vital factors to be noted in order to explore the neural dynamics in the SNN models. The work proposed by [15] intends to offer a programming method which can be applied in different GPUs, regardless of the structure of the memory of the GPUs and the number of threads. However, the number of neurons and synapses are decreased in number in comparison with [14]. The number of processors implemented in [15] is two times more than [14]. However, the number of neurons simulated in [15] is 1000 thousand times less than [14]. In these components the factor called scalability is limited. Almost all of the architectures presented here have simulated thousands of neurons by using a single GPU. Also, the absence of an efficient communication system in these devices generates an overhead memory access. This is because also the threads are in charge of the distribution of the spikes. GPUs devices exhibit high performance on certain algorithms which involves a large amount of computation. However, simulations of large-scale SNN require large computation and large high communication bandwidth.

. *Field-Programmable Gate Arrays*

Several works have been proposed in order to develop an SNN emulator which support emulate large scale SNN models in a FPGA device [8, 16, 17, 18, 19, and 20]. Only few of these works have achieved to simulate very large scale SNN models [19, 20]. One of these works is proposed by [19], who has achieved the implementation of 1000000 leaky integrate-and-fire neurons. The implementation of the spiking neurons was carried out in very simple processors that are many in number but integrated into a single FPGA. In this work the number of neurons implemented is of a good number but the number of synapses was unreported. But in case of the human brain there a large number of neurons present at the same time the number of synapses are much higher. In other words, neurons require a high level of connectivity. The connectivity of the neurons imposes a big challenge to develop such levels of connectivity in the current FPGAs. An architecture called Bluehive [20] is proposed to support large number of neurons and synapses. This architecture is composed of 64-FPGAs interconnected by high speed serial links. Each FPGA can emulate up to 64k Izhikevich neurons and 64000000 synapses per neuron [6]. The proposal of this work is focused to build a system with high speed communication by using high speed serial links available in the FPGA. However, there is no mechanism that can manage congestion in the SNN network in the case of saturation.

The mechanisms implemented in both architectures [19] and [20] process a huge amount of neurons based on fixed pipeline stages, which reduce the capacity of the system for supporting different SNN models. These architectures were designed for simulating a simple specific SNN model which does not implement the plasticity of the synapses which plays a major role to carry out the learning process. Despite the configurable features that the FPGA approaches offer when compared with the ASIC design, the design is fixed to a particular model. In case of any minor change to implement other SNN models or by adding the modelling of the plasticity of the synapses using the previously implemented model the whole architecture has to be defined.

- *Mixed signal implementations*

The sub threshold levels of the transistor dynamics is being exploited in case of analogue domain approach so as to model the neuron in silicon. One the most relevant works done in this domain is the BrainScaleS project [21], which proposes analogue circuits to simulate the spiking neurons. The spike distribution is carried out by digital packet-switched routers. These circuits are manufactured on a single wafer. Every wafer can contain 180k I&F neurons and $4x10^7$ synapses. A great energy efficiency and low area consumption can be obtained in the simulation of large-scale networks in these types of analogue-digital circuits. However, the programmability on these architectures is affected, which is very important when there is a need to explore the dynamics of the model, which is being studied and the final description is not yet defined. Therefore, the analogue implementations can be used for applications where the behaviour of the SNN is well characterized. Otherwise, the cost of fabrication will be costly and determining dynamics of the SNN models by using analogue approaches would be time consuming. Several other factors also decrease the performance of the spiking neural computation in analogue circuits. This is due to various non-idealities in circuit and reduced precision of the calculation. Also, the noise starts to accumulate in case of cascaded analog stages, thereby making it strenuous to build such complex systems using analog devices. A hybrid system could be another solution to create a configurable SNN emulator which exhibits high computational performance in the simulation of large-scale SNN models, with low power consumption. It represents a big challenge to develop a system that benefits from the fruits of both approaches (analog and digital). There have been developing architectures, which combine the digital and analog implementations. One of them is proposed by [22]. This approach proposes a hybrid analog/digital circuit with very large-scale integration. The simulation of spiking neurons in this architecture is suitable for applications which require real-time large-scale neural simulations. Nevertheless, the simulation of the spiking neurons gives some qualitative approximations in the behaviour when compared with the simulation on digital devices. This work offers an option for simulating SNN models on a “general purpose” silicon emulator.

As it has been mentioned, several projects have been developed in order to create SNN emulators, which can support the simulation of large scale SNN models, using different development platforms as High performance computing, general purpose computers, digital multiprocessors, FPGAs, GPUs, Mixed-signal aVLSI architecture, and VLSI architectures. There are some important aspects to be discussed, which are related to the portability and the maximum structural parallelism of these emulators. The first feature opens the possibilities to use such devices in applications as mobile robotics [23], because the SNN emulations on these compact devices require less power and area consumption when compared with HPC or general purpose computer simulations. The second aspect is related to the maximum structural parallelism in these architectures because the computation of the SNN models demands large parallel computation in order to achieve real time emulations [24]. Although, modern parallel architectures such as supercomputers are considered as powerful alternatives, for speeding large-scale SNN model simulations. They are facing several problems all of them are related to the energy consumption, area consumption.

Table 1.1: Feature summary of SNN implementations on compact devices and using a single chip

Implementation	SNN model used	# of neurons	# of synapses	Device used	Migration	Program Flexibility	Scalability
SpiNNaker [13]	Izhikevich Leaky Integrate-and-Fire	20000	2000000	ARM9 processing	NO	YES	YES
Nageswaran [14]	Izhikevich	Range: 50000 to 225000	Range: 100000 to 1000000	NVIDIA GTX280 GPU card	NO	YES	NO
Arista [15]	Izhikevich	7000	7000000	NVIDIA TESLA C2050	YES	YES	NO
Bluehive [20]	Izhikevich	64000	64000000	Altera Startix IV	YES	NO	YES
Cassidy [19]	Leaky-integrate-and-fire	1000000	1000000	Virtex 5 SX240T	YES	NO	YES
Vogelstein [22]	Leaky-integrate-and-fire	2400	1048576	3 mm x 3 mm chip in -0.5 µm CMOS technology	NO	YES	NO

As it can be observed from Table 1.1, the current architectures provide a SNN emulator to be used as tool for understanding the biological functions carried out in the brain through the simulation of large-scale spiking neurons. However, these works are facing problems which are related to the performance, processing speed, high level of interconnectivity, support of complex neural modelling, etc. Therefore, the development of an efficient and configurable emulator imposes big challenges in terms of program flexibility, computational performance, communication structure and low power and low area consumption.

The construction of an emulator that supports a large number of neurons in any device in either analog or digital or mixed signal, shall take into account some of the essential aspects as mentioned above. Even though analog circuits provide a large-scale support to emulate SNN in compact designs, while doing experiments to get results in a short period of time, flexibility becomes an important factor. Upgrading and expanding the network becomes a part of flexibility. The modern FPGAs offer exceptional performance and flexibility which supports large scale SNNs. Evidently, the FPGAs consumes a significant amount of area and there is a performance penalty in re-configurability when compared with the ASIC design. Thus, the idea of implementing the biological neuron's basic features from nature in modern programmable digital systems becomes a very exciting and high impact research topic.

The main focus of this work is to build a real time configurable emulator that can support large-scale spiking neural networks with vital features such as low-area consumption, minimum-power consumption, and good performance.

1.2 Work methodology

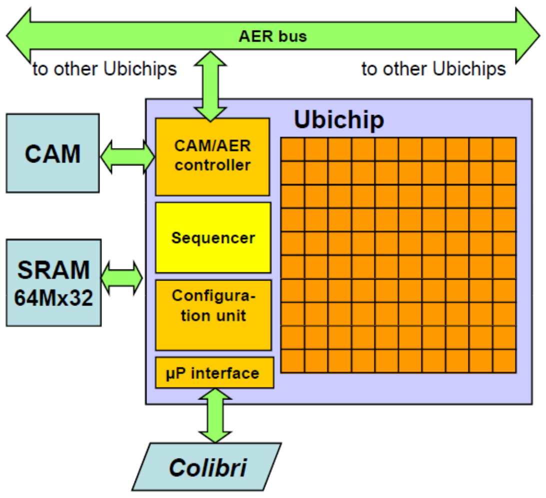
This work focuses on the development of a digital emulator, in particular using FPGAs, to support large scale SNN models. The proposed system maximizes its properties offering the following characteristics to emulate large SNN:

1. *Integrated large scale SNN models*
2. *Multi-model support*
3. *High processing speed (parameter calculation and spike distribution)*

4. *Easy upgradability*

5. *Low power and area consumption*

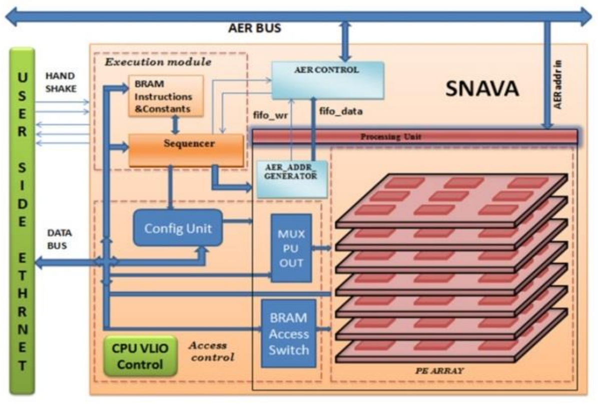
FIRST PHASE



- Study and analyze of existing multiprocessor architecture called Ubichip developed in PERPLEXUS project as a reference point.
- Implementation of SNN models of the major interest.
- Development of system to process sensory information

(a)

SECOND PHASE



- Design and development of the new multi-processor architecture called SNAVA.
- Development of advanced bio-inspired applications

(b)

Figure 1.3: Flowchart of development of this PhD thesis, a) Ubichip architecture [8], b) SNAVA architecture.

The proposed methodology used to achieve an architecture that meets the criteria specified above, is composed of two design phases. The starting point of this thesis is the analysis of an existing multiprocessor called Ubichip which was developed during PERPLEXUS project [8]. The performance of the Izhikevich model [6], the Iglesias and Villa model [7] were measured in Ubichip and new ideas emerged to generate the next architecture with better benefits and features. The new architecture was designed and evaluated in the second phase. Apart from this, advanced bio-inspired applications were also implemented on it. The Figure 1.4 shows the general flowchart, which resumes the process development of this thesis.

1.3 Outline of the thesis

The thesis is divided into two parts according to the development of this project. The first part, which comprises of Chapter 2, Chapter 3 and Chapter 4, is dedicated to analyze the Ubichip architecture and to develop applications on it in order to determine the extent to which the Ubichip architecture supports in different SNN models. The second part consists of Chapter 5 and Chapter 6, which explains the new architecture called SNAVA and the advanced applications which can be developed on it. A detailed description of the chapters is presented in the following paragraphs:

Chapter 2: This chapter introduces the operation of the multiprocessor called Ubichip which was developed in PERPLEXUS Project, and presents a detailed analysis of the performance of the Ubichip. As a result of this analysis, several bottlenecks were detected. Possible modifications and new ideas to generate advanced architecture are discussed in this chapter.

Chapter 3: A spiking-neuron-based system that combines analog and digital multiprocessor is reported. A data interface to establish the communication between analog-digital neuromorphic systems is developed and the key factors related to the synchronization of the communication are discussed in this chapter. This work was a collaboration project between the University of Edinburgh and the Universitat Politècnica de Catalunya.

Chapter 4: Perception environment applications using sensory information as proof of concept are presented in this Chapter. These applications are detection of frequency and amplitude of a signal and LEGION image segmentation. Experimental results are provided in this Chapter.

Chapter 5: The new improved architecture called SNAVA is described in this chapter. Results of implementation of SNAVA architecture were compared with other implementation using multiprocessors, GPU and FPGA. The motivation is to put light on the contribution of SNAVA architecture to realistic SNN models support by analysis of its efficiency.

Chapter 6: The frequency and amplitude detection application of chapter 4 was extended and implemented in SNAVA. This chapter proposes SNN topology to work with a wider bandwidth. The experimental results are presented in this Chapter.

Chapter 7: The conclusion of this research work and the future work are presented in this Chapter.

References

[1] J. Nolte, *"The Human Brain: An Introduction to Its Functional Anatomy,"* Elsevier, pp. 25-35, 2009.

[2] Hebb, D. O., "*The organization of behaviour,*" Wiley, conference on New York, 1949.

[3] A.L. Hodgkin, A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerves", Journal of Physiology 117, pp 500–544, 1952.

[4] Jolivet, R., T.J. Lewis, and W. Gerstner, "*Generalized Integrate-and-Fire Models of Neuronal Activity Approximate Spike Trains of a Detailed Model to a High Degree of Accuracy*", J. Neurophysiology, pp. 959-976, 2004.

[5] F. Gabbiani and C. Koch, "*Coding of time-varying signals in spike trains of integrate-and-fire neurons with random threshold*", *Neural Comput.*, vol. 8, pp. 44-66, 1996.

[6] Izhikevich, E.M., "*Simple model of spiking neurons*", Neural Networks, IEEE Transactions on, pp. 1569-1572, 2003.

[7] J. Iglesias, J. Eriksson, F. Grize, M. Tomassini, and A. E. P. Villa, *"Dynamics of pruning in simulated large-scale spiking neural networks," Biosystems*, vol. 79, Issues 1-3, 2005.

[8] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas, *"The Perplexus bio-inspired reconfigurable circuit*", in *Adaptive Hardware and Systems, AHS 2007, Second NASA/ESA Conference on*, pp. 600-605, 2007.

[9] H. Markram, "*The blue brain project", Nat Rev Neurosci*, vol. 7, pp. 153-60, Feb 2006.

[10] *https://www.humanbrainproject.eu/discover/the-project/*

[11] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "*The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses",* presented at the Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, pp. 1-12, 2009.

[12] T. Sharp, F. Galluppi, A. Rast, and S. Furber, "*Power-efficient simulation of detailed cortical microcircuits on SpiNNaker", Journal of Neuroscience Methods,* vol. 210, pp. 110-118, 9/15/2012.

[13] J. Xin, Luja, x, M. n, L. A. Plana, S. Davies, *et al.*, "*Modelling Spiking Neural Networks on SpiNNaker", Computing in Science & Engineering,* vol. 12, pp. 91-97, 2010.

[14] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum, "*A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors", Neural Networks,* vol. 22, pp. 791-800, 2009.

[15] A. Arista-Jalife and R. A. Vazquez, *"Implementation of configurable and multipurpose spiking neural networks on GPUs"*, in *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pp. 1-8, 2012.

[16] Bellis, S., et al. "*FPGA implementation of spiking neural networks - an initial step towards building tangible collaborative autonomous agents,"* in *Field-Programmable Technology, Proceedings, 2004 IEEE International Conference on*, pp. 245-250, 2004.

[17] Pearson, M.J., et al. "*Design and FPGA implementation of an embedded real-time biologically plausible spiking neural network processor*," in *Field Programmable Logic and Applications, 2005, International Conference on*, pp. 1205-1220, 2005.

[18] Teuscher, C., "*FPGA Implementations of Neural Networks," (Ormondi. A.R. and Rajapakse, J.C., Eds.; 2006).* Neural Networks, IEEE Transactions on, pp. 1550-1550, 2007.

[19] Cassidy A, Andreou AG, Georgiou J (2011), "*Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis"*, Information Sciences and Systems (CISS), 45th Annual Conference on, pp 1-6, 2011.

[20] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "*Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation*", in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 133-140, 2012.

[21] Schemmel, J. and Brüderle, D. and Grübl, A. and Hock, M. and Meier, K. and Millner, S., "A wafer-scale neuromorphic hardware system for large-scale neural modelling," Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, pp. 1947-1950, 2010.

[22] Vogelstein, R.J. and Mallik, U. and Vogelstein, J.T. and Cauwenberghs, G., "*Dynamically Reconfigurable Silicon Array of Spiking Neurons With Conductance-Based Synapses,"* Neural Networks, IEEE Transactions on, pp. 253-265, 2007.

[23] X. Wang, Z. G. Hou, M. Tan, Y. Wang, and Z. Huang, "*Spiking neural networks and its application for mobile robots*", in *Control Conference (CCC), 2011 30th Chinese*, pp. 4133-4138, 2011.

[24] T. Schönauer, A. Jahnke, U. Roth, H. Klar, "*Digital Neurohardware: Principles and Perspectives,"* in proceedings of neuronale netze in der anwendung, pp. 101-106, 1998.

Part I

Ubichip benchmarks and applications

Performance study of SNN model implementations on Ubichip

2

2.1 Introduction

According to the objective of this thesis, the development of the SNN emulator which guarantees high performance, low power and area consumption, and flexibility for supporting different large-scale SNN models is required. Several aspects were discussed in Chapter 1 in order to select the appropriate digital compact device (Graphical Processing Units, Multiprocessor, or Field Programmable Gate Array) to be the development platform to create such SNN emulator which fulfils the needs mentioned above. As a result of the study of these digital compact devices, the FPGA was selected to be the development platform for the new architecture in this work. This is because the modern programmable FPGAs offer attractive hardware interfaces, which involves high-bandwidth memory interfaces and high speed serial links. These interfaces allow to build a system where the high processing and high distribution system is required to simulate large-scale SNN models. Also, the new architecture can take advantage of the main characteristic of the FPGAs, which is called configurability. This will allows the easy implementation of different SNN models in a short time when compared to the ASIC design.

The Advanced Hardware Architectures (AHA) group at the Universitat Politècnica de Catalunya has been involved actively in Research projects in order to develop digital architectures for the emulation of bio-inspired systems. One of the most recent projects in this area was the PERPLEXUS project [1]. This project was funded by the European Commission through the 6th framework programme and established the collaboration eight research institutions from four different countries. One of the objectives of this project was to develop hardware capable of implementing biologically-inspired spiking neurons. The AHA group was responsible for developing the neural emulation architecture called Ubichip. Besides the emulation of SNN the Ubichip was developed also to support dynamic routing, mechanisms of self-replication. These features permit to carry out numerous bio-inspired mechanisms such as learning, growth, and evolution by simulating complex systems. A significant increase in the area consumption is the price for the implementation of all the above mechanisms. Therefore, the neural emulator was implemented with large area overhead. The Ubichip architecture was prototyped on FPGA for testing and verification for later to be transferred to an ASIC chip.

The idea of implementing the neural mechanism processing, using a parallel architecture based on Single Instruction, Multiple Data (SIMD) was introduced in Ubichip. It is an interesting approach which could be explored in order to take full advantage of all its capabilities, in order to emulate large-scale SNN models, which demands high parallel computation. As mentioned above, the current version of the Ubichip was designed under some limitations, so that it has generated architecture with poor processing efficiency. A great improvement of this architecture imposes a big challenge in order to create an emulator of spiking neurons efficiently, in addition maintaining low area and power consumption. This chapter is devoted to analyze how the bottlenecks of Ubichip affect its performance, from here possible improvements will be proposed to generate a new architecture

A performance evaluation of the Ubichip, in terms of processing and communication, has been carried out through the simulation of two Spiking Neural Networks (SNN) models. The result of the evaluation provides insight on further improvements. The first SNN model evaluated was proposed by Iglesias and Villa [2], which was used as a reference model for the development of the PERPLEXUS project. The second SNN model analyzed here is one of the most commonly used in the simulation of spiking neurons which was proposed by Izhikevich [3].

This Chapter gives a brief explanation of Ubichip architecture and its phases of operation in the emulation of an SNN model. And finally, the results of the performance evaluation of the two models are provided in detail along with the prominent architectural limitations affecting performance.

2.2 Ubichip architecture description

The configurable architecture called Ubichip was designed for supporting complex bio-inspired systems, the emulation of large-scale spiking neural network (SNN) being one of them. The Ubichip architecture is composed of array of Single Instruction, Multiple Data (SIMD) units. This type of architecture is classified as a parallel architecture according to the Flynn´s taxonomy [4]. There are two aspects which were considered in the development of Ubichip in order to simulate Spiking Neural Networks models on SIMD architecture. Firstly the nature of SNN models is completely parallel which represents the best challenge for parallel architectures to be used as the base of SNN emulators [5], and secondly the selected SIMD machine presents a simple architecture which makes the machine potentially inexpensive in terms of area and energy consumption. Therefore, the proposed architecture intends to simulate large scale SNN models by expending the minimum area resources. Several modes were implemented in Ubichip for supporting different applications. The concerning mode to develop this work is the multiprocessor mode. This mode was specially developed to emulate such bio-inspired SNN models [1, 6].

The Ubichip has three main modules, which are: Configurable array, System Manager, and the Address Event Representation (AER) module, as shown in Fig. 2.1. The configurable array consists of Processing Elements (PE), which are the basic building blocks of the system. The AER controller takes care of the communication between the processing elements within the chip and also between different chips. It consists of an encoder or control unit and a decoder or Content Address Memory (CAM) unit. The system manager comprises of the configuration unit, sequencer, memory controller and the CPU interface or Variable Latency Input Output unit (VLIO). Outside the FPGA there is an External SRAM that stores the synapse and neural parameters and the CPU that is used for initial configuration and access to the chip for response analysis.

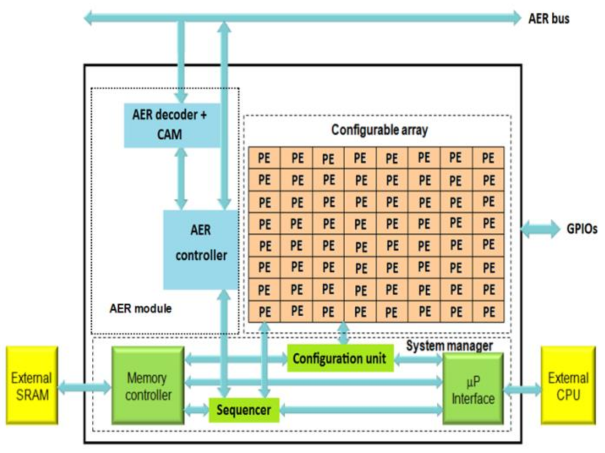


Figure 2.1: The Ubichip architecture for the multiprocessor mode, figure extracted from [1]

The functional operation of Ubichip was designed to work in two operational phases, which are the data processing (phase 1) and the spike distribution (phase 2). Therefore, the emulation of SNN models could be executed by these two periodic phases. During the processing phase, the synapse and neural parameters are calculated. The spikes generated by the neurons in phase 1 are distributed by the AER module through the SNN network. The spike distribution is carried out by the synchronous AER protocol defined in [5] in order to avoid overhead connection when a large-scale SNN models are implemented.

2.2.1 Architecture functional details

- Configurable array

The configurable array is based on parallel SIMD units, which were defined in this architecture as Processing Elements (PEs). Each PE is a 16 bit processor. It is built with two 16-bit register banks and a 16 bit ALU. The ALU is capable of performing arithmetic and logical operations like 2'complement addition and subtraction, shifting, and, or, xor, 2'complement and Negation. Multiplication is also possible but by software through repeated addition. The two sets of 8-register banks containing 16-bit registers are called the active and the shadow registers. The active register is the one the ALU operates on where one operand is always the register 0 also known as accumulator. The shadow register as the name implies serves as a temporary storage for the active registers providing space for complex algorithms. Data move operations are possible between the active and the shadow registers either as bulk or single. The structure of the PE is shown in Fig. 2.2.

This SIMD approach allows the removal of the local program memory from PEs at the cost of forcing the same program to be executed by all PEs. The program is thus stored in a common single memory, while the data which is being processed is locally stored in the PE register bank. The data is transferred to/from the common external SRAM (Static Random Access Memory) when necessary. The flexibility to change the values of synaptic parameters, neural parameters, as well as the structure of the connectivity pattern can be made only by changing the program and configuration parameters stored in an SRAM.

The selected parallelization approach consists of assigning a PE to each neuron and its associated input synapses. Thus, neurons are emulated in parallel, and the synapses of each neuron, serially. This approach exhibits a good trade-off between today´s utopia fully-parallel emulation and the serial approach.

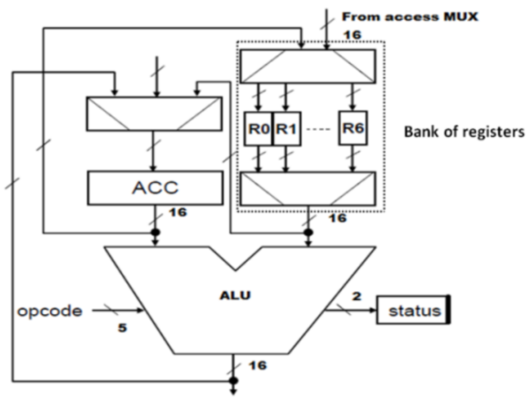


Figure 2.2: Processing Element path, figure extracted from [6]

- System Manager

The system manager is composed by the sequencer, configuration unit, memory controller and microprocessor interface. An explanation is of function of each module is provided as follows:

- A single sequencer, which is external to all PEs [7], is in charge of controlling the program flow. It performs the following tasks:
    - Fetching and decoding the instructions stored in a common memory.
    - Broadcasting the instructions to be executed by the SIMD PE array.
    - Executing the instructions which are specific for the sequencer itself.
    - Transferring the data between the external SRAM and PE array.
    - Interfacing with an AER/CAM controller [7].
- The configuration unit takes care of managing the configuration of the different building blocks of the Ubichip. It also permits to set the registers that drive the integrated debugging capabilities present in the device like setting the clock mode, debugging step by step, enabling and disabling the AER, Sequencer, resetting the array etc. Two modes of configuration are supported by the configuration unit: serial and parallel configuration. In serial configuration mode the Ubichip is configured through a serial interface driven by an external unit (CPU or CPLD). In parallel configuration mode the Ubichip is configured from an external CPU unit though its memory bus.
- A memory controller is in charge for generating the signals to access to the SRAM (Static RAM) from the sequencer.
- The microprocessor interface as well-known as the VLIO (Variable Latency Input-Output) interface is just a control state machine that synchronizes with a set of signals the external CPU access. The Ubichip is thus configured through the memory bus in VLIO mode consisting in a 26-bit address bus (CPU_addr), a 32-bit data bus (CPU_data). This interface controller is connected to a Colibri board (external CPU), which contains an XScale processor that is responsible for configuring, controlling and providing input-output functions to the Ubichip.

- The Address Event Representation (AER) module

The function of the AER module is dedicated to broadcast post-synaptic spikes from any Ubichip to all Ubichips connected to the AER bus (see Fig. 2.1). This module works under the synchronous AER protocol which was defined in [7] during the development of PERPLEXUS project. Also, the AER module works 10 times less than the frequency operation of the multiprocessor in order to achieve the communication between 100 Ubichips. The AER module consists basically of three components which are:

- An AER controller that reads the spikes produced by the multiprocessor array and sends them to the AER bus.

- An AER decoder, that detects the synapses connected to the AER spike and writes the corresponding pre-synaptic spikes into the suitable SRAM positions.
- A CAM (Content-Addressable Memory) models the synaptic connections between neurons. Each CAM has tags that have the ID's of the PE's connected to that particular PE. Thus the number of associated tags is the number of synaptic connections to the neuron.

2.2.1.1 AER system

This section describes briefly the AER protocol implemented in AER module of Ubichip. The AER protocol was developed during PERPLEXUS project [1]. This AER protocol theoretically allows 100 Ubichips to communicate through the common bus and avoids the interconnectivity overhead. Also, this protocol allows the synchronization in communication between the Ubichips. Fig. 2.3 shows the simplified view of the proposed network in PERPLEXUS project.

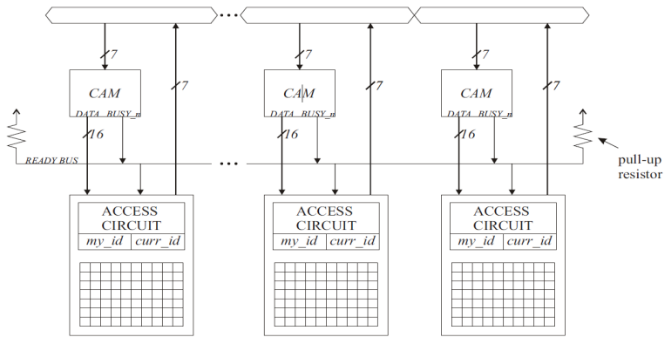


Figure 2.3: Simplified view of the PERPLEXUS framework, figure extracted from [7]

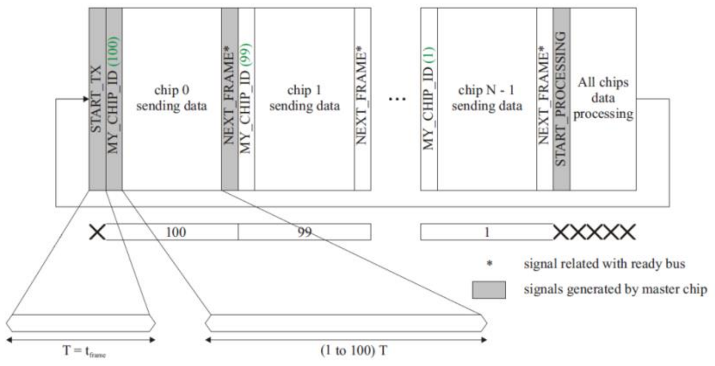


Figure 2.4: Conceptual view of the transmission through the common bus, figure extracted from [7]

The transmission of the spikes through the common bus is carried out sequencially. The operation can be briefly explained as follows. Every Ubichip has an ID in order to identify the current Ubichip that is given access to the bus. The transmission begins when the master Ubichip (with the highest ID) sends the START_TX signal and its chip ID. This Ubichip sends its spikes to other Ubichips. Once, the master

Ubichip has finished its transmission, the chip ID value is decremented by 1, thereby giving access to the next higher priority Ubichip. Similar pattern is followed for giving the access to subsequent Ubichips in the network. Fig. 2.4 shows the general idea in the synchronization between Ubichips (see [7], for further information).

2.3 Performance evaluation – Iglesias and Villa model

This section presents the performance analysis of the algorithm implementation of the complex bio-inspired SNN algorithm proposed by Iglesias and Villa [2] in assembly code of the Ubichip in multiprocessor mode. The implementation has been done following a structured manner [9], using certain procedures so as to simplify the maintenance and updating. The low-level programming allows optimizing the algorithm´s execution time. The algorithm, which has been written in assembler code for Ubichip, is provided in Annexure C.

2.3.1 Iglesias and Villa model description

The SNN model proposed by Iglesias and Villa includes the modelling of the neuron as leaky integrate and fire neuron but they include in the modelling the synapses important mechanisms like learning based on the Spike-Timing-Dependent-synaptic Plasticity (STDP). They also included the noise in the simulation of their model and the refractoriness in the neuron. The equation 2.1 that describes the membrane potential $V_i(t)$ in their model is calculated as follows:

$$V_i(t+1) = V_{rest[q]} + B_i(t) + \left(1 - S_i(t)\right)\left(\left(V_i(t) - V_{rest[q]}\right)k_{mem[q]}\right) + \sum_j \omega_{ji}(t) \qquad (2.1)$$

where $V_i(t+1)$ refers to the membrane potential of neuron type either excitatory or inhibitory, $V_{rest[q]}$ corresponds to the value of the resting potential for the units of class type (excitatory or inhibitory), $B_i(t)$ is the background activity noise, $S_i(t)$ is the postsynaptic unit state, $k_{mem[q]} = e^{\left(-\frac{1}{\tau_{mem[q]}}\right)}$ is the time constant associated to the current of leakage for the units of class type q (excitatory or inhibitory), and finally, $\omega_{ji}(t)$ is the postsynaptic potentials. Therefore, the postsynaptic potential $\omega_{ji}(t)$ is a function of the state of the presynaptic unit S_j, of the type of the synapse (excitatory or inhibitory) $P_{[qj,qi]}$, and of the activation level of the synapse $A_{ji}(t)$. This is expressed by the following equation:

$$\omega_{ji}(t+1) = S_j(t) \cdot A_{ji}(t) \cdot P_{[q_j,q_i]} \qquad (2.2)$$

The real-valued variable L_{ji} (t) was introduced in order to determine the activation level $A_{ji}(t)$ of a synapse. The variable L_{ji} (t) is used to implement the STDP rule for $A_{ji}(t)$. It is important to be noted that the STDP rule depends on the occurrence of presynaptic spikes, the generation of postsynaptic spikes and the correlation between both. It integrates the timing of the pre- and postsynaptic activities and decays itself overtime. STDP defines how the value of L_{ji} (t) at the time t is changed by the arrival of the presynaptic spikes, by the generation of postsynaptic spikes and the correlation existing between these events. That means if a neuron generates a postsynaptic spike ($S_i(t)$), the real valued variable is going to be incremented. Similarly, when a presynaptic spikes arrives at the synapse ($S_j(t)$), the variable L_{ji} (t) receives a decrement. The increments or decrements are reflected directly in the value of $A_{ji}(t)$. This was proposed to implement the plasticity in the synapses. The calculation of the real-valued variable L_{ji} is given by the following equation:

$$\mathrm{L_{ji}(t+1) = L_{ji}(t) \cdot k_{act[qj.qi]} + S_i(t) \cdot M_j(t) - S_j(t) \cdot M_i(t)} \tag{2.3}$$

As one can see, there is another dependency to the variables $M_j(t), M_i(t)$ and $k_{act\left[q_{j},q_{i}\right]}$. $M_j(t)$ and $M_i(t)$ describe the "memory" of latest inter-spike, they are explained later on. $k_{act\left[q_{j},q_{i}\right]}$ is responsible for decaying L_{ji} exponentially, this is due to the decreasing of the elapsed time from the previous postsynaptic spike.

$$k_{act\left[q\right]} = \mathrm{e}^{\left(-\frac{1}{\tau_{act[q]}}\right)} \tag{2.4}$$

The variables L_{ji} are user-defined boundaries of attraction $L_0 < L_1 < L_2 < \cdots L_{k-1} < L_k$. The distance between two neighbored boundaries is always a constant i.e. $L_{k+1} - L_k = const$. Overstepping one of these boundaries induces L_{ji} to reset to the middle of its upper or lower domain, where the L_{max} is defined as the maximum value of L_{ji}. The reset value of L_{ji} is calculated by the equation 2.5

$$L_{ji}(t+1) = \frac{L_{\max}}{2} \tag{2.5}$$

M_i and M_j can be viewed as the memory of the latest inter-spike interval. M_i stays for the memory of the latest presynaptic spike

$$M_i\left(t+1\right)=s_i(t)\cdot M_{\max[qi]}+\left(1-s_i\left(t\right)\right)\cdot M_i\left(t\right)\cdot k_{syn[qi]} \tag{2.6}$$

M_i however, is the memory of the latest postsynaptic spike and refers to the projected neuron (2.9) in other words it describes the neuron that has been fired, hierarchically it is not part of the synapse. The following formula represents the memory of the latest presynaptic spike: when spike is generated (presynaptic or postsynaptic) the according memory is reset to its maximum value. On the other side, when no spike was emitted the memory variable just will be decayed by the synaptic plasticity time constant k_{syn} .The formula for the latest postsynaptic spike looks similar, but must be assigned to the characteristics of the projected neuron.

$$M_j\left(t+1\right)=s_j(t)\cdot M_{\max[qj]}+\left(1-s_j\left(t\right)\right)\cdot M_j\left(t\right)\cdot k_{syn[qj]} \tag{2.7}$$

If a neurons membrane potential crosses a certain threshold value $\theta_{[q]}$ from below, it generates spike, also called action potential, and sends it out down the axon. Shortly after the membrane value resets to its resting potential.

$$s_i(t)=H\left(V_i\left(t\right)-\theta_{[q]i}\right) \tag{2.8}$$

$$H(x)=\begin{cases}0: x<0\\ 1: x\geq 0\end{cases} \tag{2.9}$$

Where, H is the Heaviside function and indicates, whether a spike is generated or not. This condition is also conditioned to the refractoriness of the neuron. After firing a spike, the neuron enters a short moment of rest, the absolute refractory period prevents that the neuron fires again. Even with very strong input, it is impossible to excite a second spike during or immediately after a first one. The minimal distance between two spikes defines the absolute refractory period of the neuron.

$$A_{ji}\left(L_{ji}\right)=\begin{cases}0, & if\left(\left(A_{ji}=1\right)\wedge\left(L_{ji}<L_{\min}\right)\right)\\ 1, & if\left(\left(A_{ji}=0\right)\wedge\left(L_{ji}<L_{\max}\right)\vee\left(A_{ji}=2\right)\wedge\left(L_{ji}<L_{\min}\right)\right)\\ 2, & if\left(\left(A_{ji}=1\right)\wedge\left(L_{ji}<L_{\max}\right)\vee\left(A_{ji}=4\right)\wedge\left(L_{ji}<L_{\min}\right)\right)\\ 4, & if\left(\left(A_{ji}=2\right)\wedge\left(L_{ji}<L_{\max}\right)\right)\\ A_{ji}, & if\left(\left(L_{ji}\geq L_{\max}\right)\wedge\left(L_{ji}\leq L_{\min}\right)\right)\end{cases} \tag{2.10}$$

The activation variable $A_{ji}(t)$ reflects the activity of a synapse. In the model it is distinguished between four different activation states for a synapse: 0,1,2,4 (N=4). In this case 0 means that there is no activity anymore, the synapse is dead – this characteristic is known as synaptic pruning. On the other side an activation level of 4 indicates the highest level of synapse activity. This has a significant impact for the postsynaptic neuron, because its membrane value will grow and spiking is provoked quicker.

The activation variable $A_{ji}(t)$ is directly dependent on the variable L_{ji}. Whenever L_{ji} exceeds the border from one of its domains to another $A_{ji}(t)$ is directly affected by it and jumps to one of its neighboured states, $[A_k]$ to $[A_{k+1}]$ for increasing and $[A_k]$ to $[A_{k-1}]$ for decreasing $A_{ji}(t)$.

2.3.2 Iglesias and Villa model implementation

The execution loop to emulate the Iglesias and Villa model is shown in Fig. 2.5. As it can be noted, after a short parameter initialization, the SNN algorithm is cyclically emulated by means of an infinite loop that executes phase 1 to emulate the neural network and stops for the AER decoder to execute the spike broadcasting of phase 2. In this phase, no instructions are executed, but the AER controller and AER decoder control units perform the required operations by means of finite state machines. When phase 2 is done, the sequencer resumes execution of phase 1 and this loop is made to run continuously.

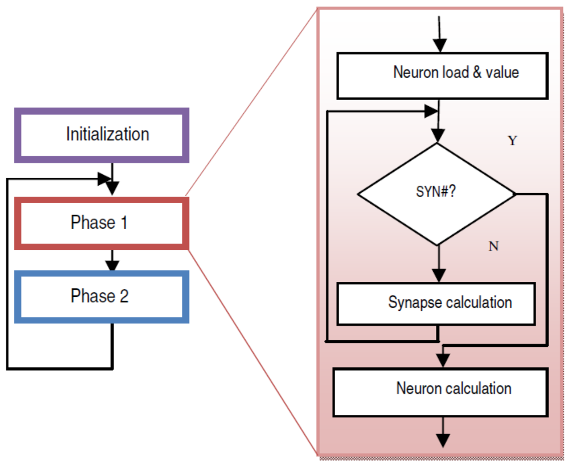
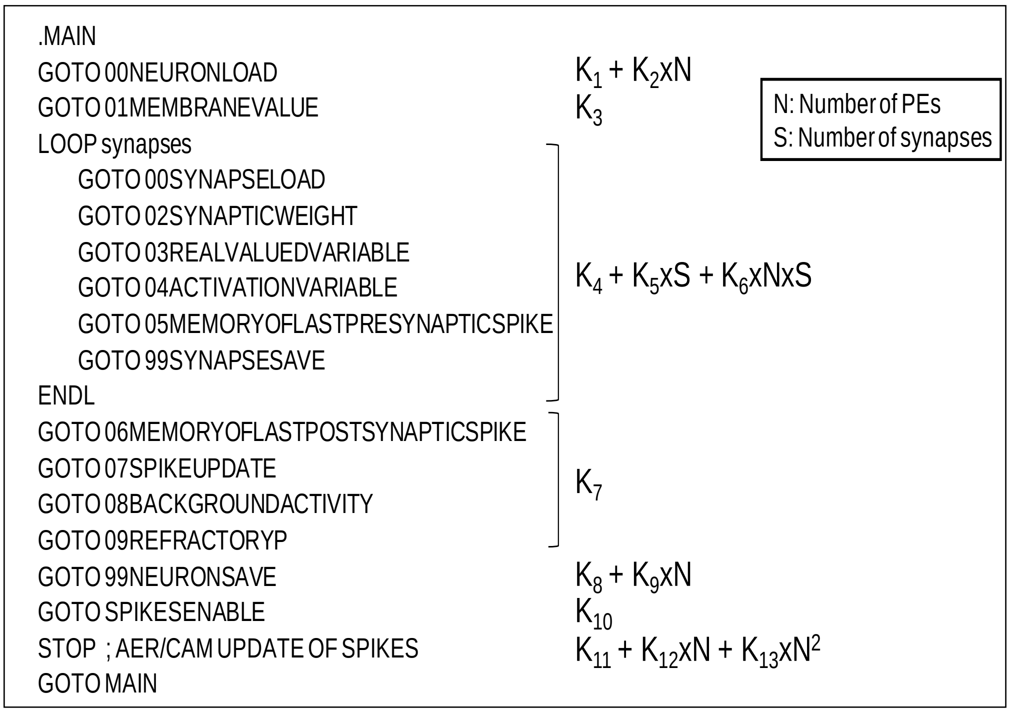


Figure 2.5: Execution loop for SNN emulation. Phase 1 main operations are detailed

In phase 1, when the neuron and synapse algorithms are executed, the neuron parameters are first loaded and then the membrane value is calculated. Then, the input synapses of the associated neuron are calculated. Finally, the neuron is updated taking into account the input synapses, background noise and refractory period, thereby determining whether it spikes or not.

As mentioned earlier in the beginning of this section, the algorithm to perform the Iglesias and Villa model in Ubichip was programmed in assembler language. The main loop of this assembler program is shown in Figure 2.6. This loop consists of 8 subroutine calls which are dedicated to calculate the neural parameters, and a synapse loop which is dedicated to compute the synaptic parameters. The number of times the synaptic loop is executed is equal to the defined number of synapses. This synaptic loop also includes 6 additional subroutines. Every subroutine requires a certain number of clock cycles to perform a specific part of the algorithm. The number of clock cycles per subroutine were measured and translated into a mathematical equation as a function of the constants K and the variables S or N. Where variables N and S are defined based on the number of neurons and the number of synapses to be emulated by the Ubichip multiprocessor. Also, the variable N is related to the number of times the sequencer can access to the external SRAM for loading and storing data from the PE and vice versa. Whereas, in case of spike distribution the sequencer gets access to the SRAM to save the data received from the AER module. This can be seen clearly in Fig 2.6, the subroutines which are dedicated to load and save parameters takes into account the variable N. Therefore, The constants $K_1 \cdot\cdot K_{13}$ and variables N and S allow the calculation of the number of clock cycles required for the execution of each subroutine as shown in Fig. 2.6.



```
.MAIN
GOTO 00NEURONLOAD                              K1 + K2xN
GOTO 01MEMBRANEVALUE                           K3
LOOP synapses                        ┐
    GOTO 00SYNAPSELOAD               │
    GOTO 02SYNAPTICWEIGHT            │
    GOTO 03REALVALUEDVARIABLE        │
    GOTO 04ACTIVATIONVARIABLE        │         K4 + K5xS + K6xNxS
    GOTO 05MEMORYOFLASTPRESYNAPTICSPIKE
    GOTO 99SYNAPSESAVE               │
ENDL                                 ┘
GOTO 06MEMORYOFLASTPOSTSYNAPTICSPIKE ┐
GOTO 07SPIKEUPDATE                   │         K7
GOTO 08BACKGROUNDACTIVITY            │
GOTO 09REFRACTORYP                   ┘
GOTO 99NEURONSAVE                              K8 + K9xN
GOTO SPIKESENABLE                              K10
STOP ; AER/CAM UPDATE OF SPIKES                K11 + K12xN + K13xN^2
GOTO MAIN
```

N: Number of PEs
S: Number of synapses

Figure 2.6: Main program of the SNN emulation assembly code and number of cycles of each subroutine.

2.3.3 Performance figures

The encoding for each subroutine in phase 1, which is indicated in Table 2.1, takes into account the expressions defined in Fig. 2.6. The initial conditions (IC) (initialization of Fig. 2.5) are also considered, although they do not have any relevance in the calculation. The encoding of subroutines contained in the synapse loop is shown in Table 2.2. Adding the number of cycles of these subroutines, the Cycle per Synapse (CS) figure of Table 2.1 is obtained.

Table 2.1: Main loop subroutine encoding and execution number of clock cycles

Symbol	Subroutine	Clock cycles *
I C	Initial conditions	24 + N
N L	Neuron Load	148 + 4N
M V	Membrane value	538
C S	Cycle per each synapse	(1392+4N)S
M O L P	Memory of last post-synaptic	496
S U	Spike Update	70
B A	Background activity	527+ N
R P	Refractory period	6
N S	Neuron save	92+4N
S E	Spike enable	8

* The number of clock cycles depends on the number of synapses (S) and number of neurons (N).

Table 2.2: Synapse loop routine encoding

Symbol	Subroutine	Clock cycles *
S L	Synapse Load	150 + 2N
S W	Synaptic weight	70
R V V	Real value variable	472
A V	Activation variable	100
M O L P	Memory of last pre-synaptic spike	450
S S	Synapse Save	150 + 2N

* The number of clock cycles depends on the number of neurons (N).

Adding all the contributions of Table 2.1, the number of clock cycles N_T that is required for the initialization and the phase 1 execution in one simulation cycle is obtained which is shown in equation (2.11).

$$N_T = 1909 + 10 \times N + 1392 \times S + 4 \times N \times S \tag{2.11}$$

As it can be observed from equation 2.11, the number of clock cycles varies with respect to the number of neurons (N) and synapses (S). It is important to consider both synapses and neurons when evaluating scalability. From inspection of eq. (2.11), clearly the major contribution to the delay is given by the number of synapses, while the number of neurons is contributing less to the delay when compared to the number of synapses. As indicated before, the maximum number of neurons to be emulated in Ubichip is limited by the number of PEs. This is because every neuron is associated to each processor. The maximum number of synapses is defined in function of the size of the Content Address Memory (CAMs), which contains the logic mapping of the connection of the SNN network.

In Table 2.3, the previous expression (2.11) is used to calculate execution times for different SNN emulation array sizes to infer scalability. The execution time depends on the system clock. Here, the conservative Ubichip prototype 50 MHz clock, or 20 ns period, is assumed, although the delays would be proportionally reduced as the clock period decreases. The distribution time (spike propagation) is also considered for the calculation, taking into account that in the PERPLEXUS implementation the AER bus is working at around 5 MHz [1]. The AER bus is working 10 times slower than the multiprocessor in order to ensure a successful communication between several Ubichips.

As shown in the Table 2.3, even working with the current prototype clock, the system performance is very close to the real-time emulation. For the 300-synapse 10000-neuron network proposed case, using 100 Ubichips, 78 spike/s rate is obtained for individual neurons, which is very close to the proposed target in the PERPLEXUS project [1]. Furthermore, the spiking phase is calculated on a worst-case, all-neuron spike basis, because not all neurons will be spiking at every simulation step.

Table 2.3: Execution time of one simulation cycle for different SNN size

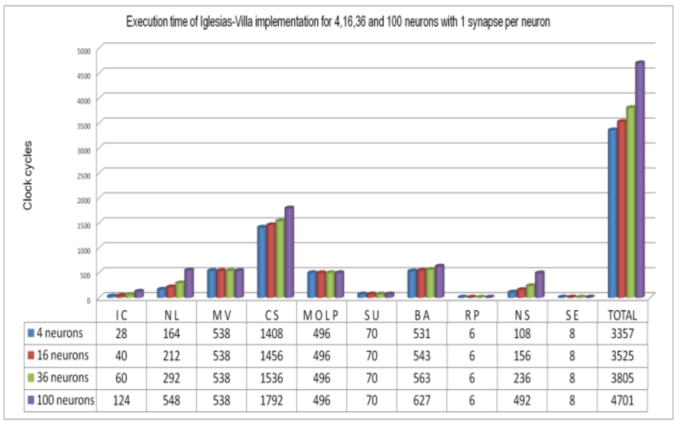
Array	#PE	#Syn	#Chip	Processing phase (clock cycles)					fCK	Total phase1	Spiking phase		Total phase2	**TOTAL**	**SPIKE RATE**
N	N	S	C	K1	K2*N	K3*S	K4*N*S	TOTAL	(MHz)	(ms)	AER	(MHz)	(ms)	**(ms)**	**(s^{-1})**
2x2	4	2	1	1909	40	2784	32	4765	50	0,095	8	5	0,0016	**0,10**	**10320**
2x2	4	3	1	1909	40	4176	48	6173	50	0,123	8	5	0,0016	**0,13**	**7996**
6x6	36	8	1	1909	360	11136	1152	14557	50	0,291	40	5	0,008	**0,30**	**3343**
6x6	36	12	1	1909	360	16704	1728	20701	50	0,414	40	5	0,008	**0,42**	**2370**
10x10	100	300	1	1909	1000	417600	120000	540509	50	10,810	104	5	0,0208	**10,83**	**92**
100x100	**100**	**300**	**100**	**1909**	**1000**	**417600**	**120000**	**540509**	**50**	**10,810**	**10202**	**5**	**2,0404**	**12,85**	**78**

In the following figures, the number of clock cycles required for one-step emulation of the SNN algorithm is analyzed. The purpose is to show the influence of every subroutine as a function of the number of neurons and synapses being emulated. The figures have been obtained from simulations and they have been verified for consistency with eq. (2.11).

To infer scalability, the Ubichip arrays that emulate 2x2, 4x4, 6x6 and 10x10 neuron networks are considered in the analysis. In fact, a 6x6 array has been mapped into the FPGA of the first Ubichip prototype and 10x10 was the array implemented in a standard-cell ASIC in the PERPLEXUS project [1].

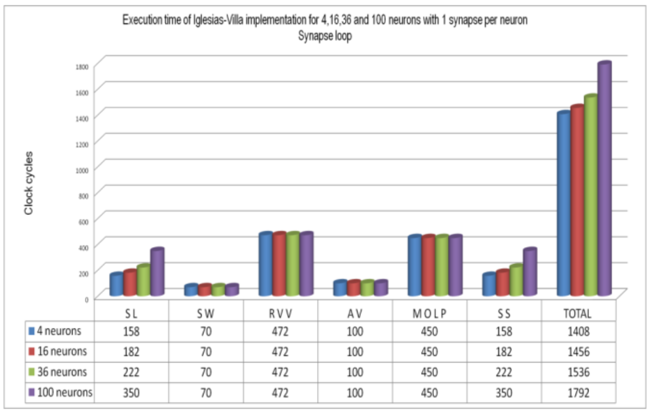
In Fig. 2.7, the required number of cycles per emulation step for the four configurations is shown. In this case, a single synapse is considered, to show scaling with the number of neurons. As predicted by eq. (2.1), the total execution time increases linearly with the number of neurons. The figure displays both the total number of cycles (in the last column) and its distribution among the main loop subroutines.

As it can be observed, the delay mostly depends on the synapse cycle, even for a single synapse. In Fig. 2.8, the number of cycles for the synapse loop (CS in Fig. 2.7) distributed among the internal subroutines is shown, also for the neural network arrays previously indicated. It can be observed that the only subroutines that increase their number of cycles with the number of neurons N are SL (synapse load) and SS (synapse save), i.e., when the SRAM is accessed. Since they are inside the synapse loop, they will linearly increase with the synapse number, so they provide a major contribution to the total delay.



	IC	NL	MV	CS	MOLP	SU	BA	RP	NS	SE	TOTAL
🟦 4 neurons	28	164	538	1408	496	70	531	6	108	8	3357
🟥 16 neurons	40	212	538	1456	496	70	543	6	156	8	3525
🟩 36 neurons	60	292	538	1536	496	70	563	6	236	8	3805
🟪 100 neurons	124	548	538	1792	496	70	627	6	492	8	4701

Figure 2.7: Required number of cycles for the execution of Iglesias-Villa implementation for 4, 16, 36 and 100 neurons with 1 synapse per neuron [10].

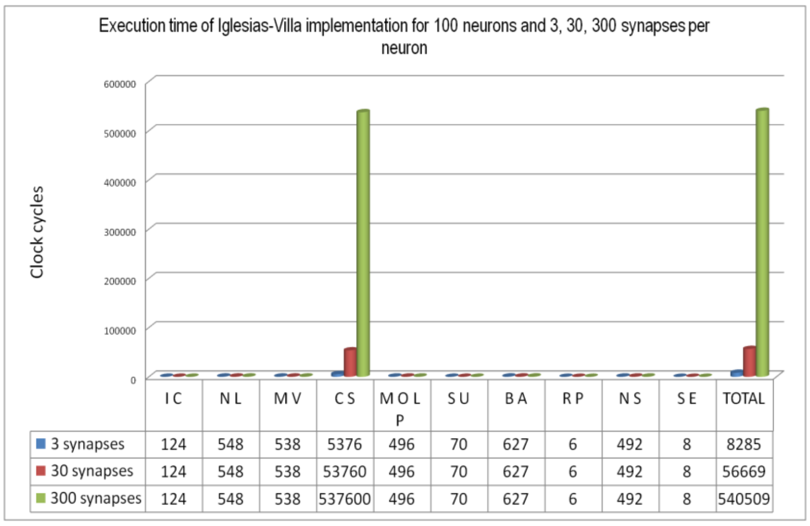


	SL	SW	RVV	AV	MOLP	SS	TOTAL
🟦 4 neurons	158	70	472	100	450	158	1408
🟥 16 neurons	182	70	472	100	450	182	1456
🟩 36 neurons	222	70	472	100	450	222	1536
🟪 100 neurons	350	70	472	100	450	350	1792

Figure 2.8: Required number of cycles for the execution of the synapse loop for 4, 16, 36 and 100 neurons with 1 synapse per neuron [10].

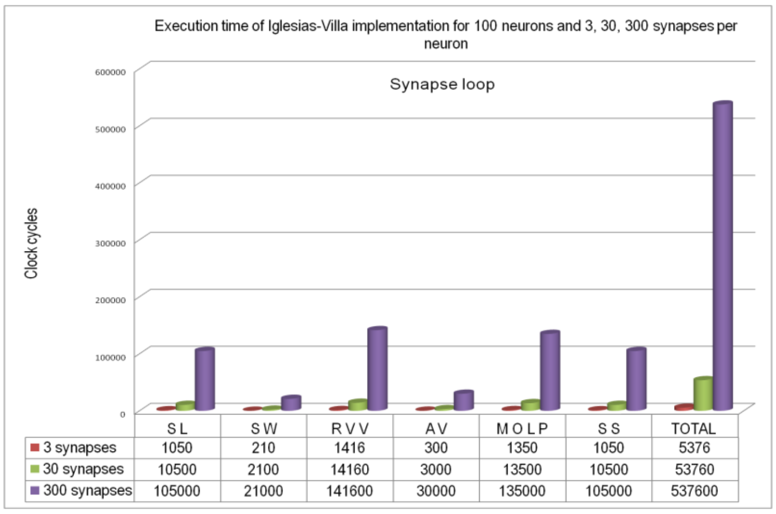
In Fig. 2.9, the required number of cycles per emulation step is presented from the synapse number point of view. For the target 100 (10x10) neuron array per Ubichip, a varying of number of synapses 3, 30 and 300 are analyzed.

The serial emulation of synapses implies that each synapse requires a synapse emulation loop. This is why CS (Synapse Cycle) is dominant even for a small number of synapses and the remaining subroutines become irrelevant, as Fig. 2.9 indicates. In fact, for the 100-neuron 300-synapse neuron array, 99.5% of the cycles are dedicated to synapse cycle.



	I C	N L	M V	C S	M O L P	S U	B A	R P	N S	S E	TOTAL
🟦 3 synapses	124	548	538	5376	496	70	627	6	492	8	8285
🟥 30 synapses	124	548	538	53760	496	70	627	6	492	8	56669
🟩 300 synapses	124	548	538	537600	496	70	627	6	492	8	540509

Figure 2.9: Required number of cycles for the execution of Iglesias-Villa SNN for 3, 30 and 300 synapses on a 100 neuron array [10].



	S L	S W	R V V	A V	M O L P	S S	TOTAL
🟥 3 synapses	1050	210	1416	300	1350	1050	5376
🟩 30 synapses	10500	2100	14160	3000	13500	10500	53760
🟪 300 synapses	105000	21000	141600	30000	135000	105000	537600

Figure 2.10: Required number of cycles for the execution of the synapse loop for 3, 30 and 300 synapses on a 100 neuron array [10]

The contribution to CS of Fig. 2.9 for each subroutine inside the synapse loop is shown in Fig. 2.10. Taking into account that the selected number of synapses varies exponentially, the exponential increase in the figure corresponds to a linear growth with the number of synapses, as expected.

2.3.4 Architecture limitations

The target of real-time emulation has been almost achieved with the current implementation. Simply by slightly increasing the current operating frequency (50 MHz) it would be totally fulfilled. Nevertheless, from the previous performance analyses, there are some possible modifications that would significantly boost the processing power of SNN algorithms based on the PERPLEXUS multiprocessor.

This section explains the limitations that were identified after the previous analysis. The current architecture´s bottlenecks will be clearly shown when a large SNN is considered, as in the case of 100 neurons and 300 synapses per neuron. Analyzing the synapse cycle operations, the main time-consuming tasks are memory access (LOAD and STORE instructions). This is because either neural or synaptic parameters are loaded and stored from the array of PEs to the external SRAM memory. Another bottleneck is the implementation of the exponential decays which are based on a software multiplication algorithm.

Table 2.4: Clock cycles devoted to LOAD, STORE, multiplication and all other instructions for the execution of Iglesias and Villa algorithm (100 neurons and 300 synapses per neuron)

Instructions and subroutine	#	# cycles/instruction	Total number of cycles
LOAD NEURONS	6	100	600
LOAD SYNAPSES	600	100	60000
STNC NEURONS	4	100	400
STNC SYNAPSES	600	100	60000
MULT. NEURONS	3	432	1296
MULT. SYNAPSES	600	432	259200
REMAINING INSTRUCTIONS			159013
		TOTAL	540509

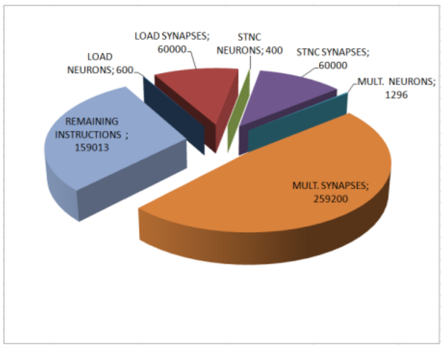


Figure 2.11: Clock cycle number distribution of instructions as classified in Table 2.3 [10]

The number of clock cycles required to execute the algorithm, LOAD, STORE (where STNC is a particular case of STORE) [6], and multiplication instructions are shown in Table 2.4. In the Remaining instructions row, all the other instructions that complete the algorithm are accounted.

As it can be observed in the pie chart of Fig. 2.11, the relative execution time of these instructions is displayed. Approximately, one-fourth of the processing time is devoted to LOAD and STORE, i.e., to SRAM access, one-half of the time to product instructions and the other one fourth, to the remaining instructions.

In order to speed up processing the following improvements can be considered:

- Parallelization of LOAD and STNC instructions.
- Hardware multiplier: Full parallel multiplier, radix-4 multiplier or parallel-serial multiplier.

- Parallelization of LOAD and STORE instructions feasibility

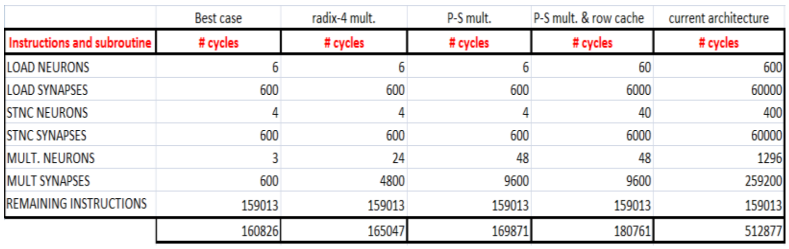
It would be feasible to parallelize the LOAD and STNC instructions only by amending the memory system architecture. Currently, modern FPGAs integrates thousands of block memory that are optimized for resource and power, so that, the possible modification could be based on the implementation of a distributed memory system. Every processor element could have a memory block. The data values could be sent from block memory to PE when the LOAD or STORE parallel instructions are executed in a single clock.

- Hardware multiplier feasibility

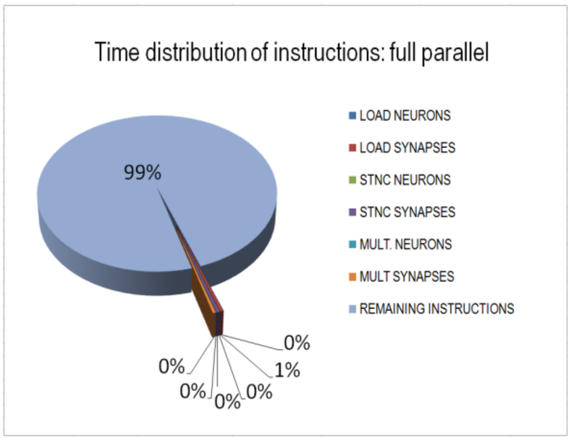
Regarding hardware multipliers, taking into account that the current architecture has 16-bit precision of the PE registers, the 16-bit multipliers mentioned above would require 1, 8, and 16 clock cycles, respectively. The PE area overhead is almost insignificant with respect to the implementation of the radix-4 and parallel-serial multiplier. But, for the full-parallel implementation, it would require further analysis. However, new FPGAs provide cores that generate parallel multipliers, and constant coefficient multipliers giving rise to a system with maximum performance and resource efficiency.

In Table 2.5, the calculated clock cycle number for several architectural modifications is shown. The first column indicates the best case, with fully parallel LOAD and STNC instructions and fully-parallel hardware multipliers. Second column replaces the full-parallel multiplier by a radix-4 multiplier, and a parallel-serial multiplier in the third column. The fourth column keeps the P.S. multiplier, but assumes that the LOAD and STNC instructions are row-serial and column-parallel (using row cache). Finally, the last column shows the current implementation figures.

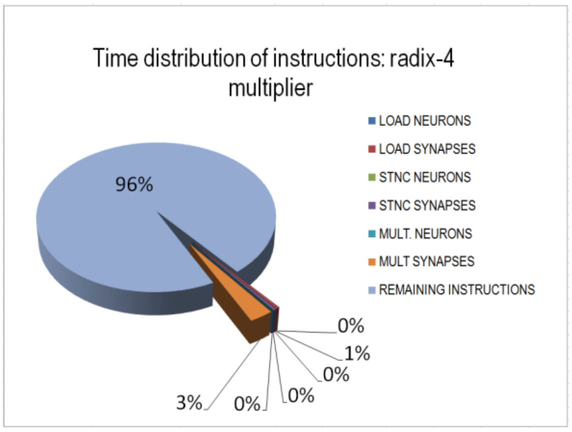
Table 2.5: Clock cycle number calculation for modified architectures



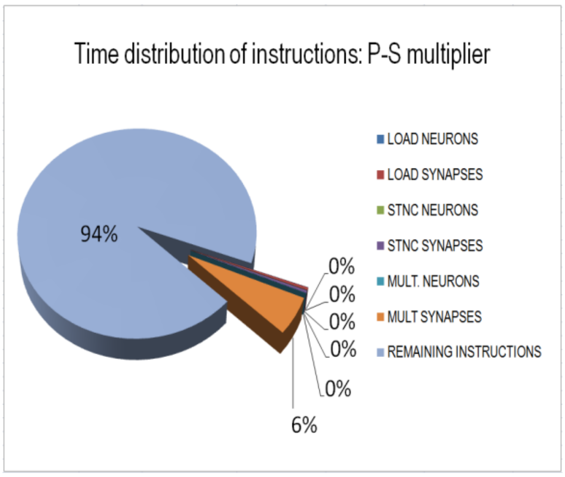
	Best case	radix-4 mult.	P-S mult.	P-S mult. & row cache	current architecture
Instructions and subroutine	# cycles	# cycles	# cycles	# cycles	# cycles
LOAD NEURONS	6	6	6	60	600
LOAD SYNAPSES	600	600	600	6000	60000
STNC NEURONS	4	4	4	40	400
STNC SYNAPSES	600	600	600	6000	60000
MULT. NEURONS	3	24	48	48	1296
MULT SYNAPSES	600	4800	9600	9600	259200
REMAINING INSTRUCTIONS	159013	159013	159013	159013	159013
	160826	165047	169871	180761	512877



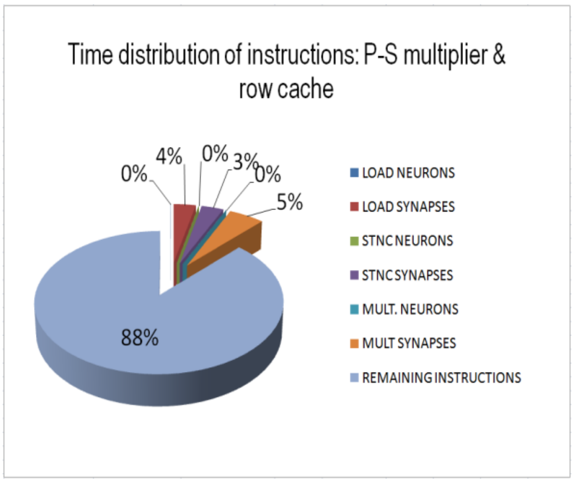
(a)



(b)



(c)



(d)

Figure 2.12: a) Clock cycle number distribution of instructions taking into account the proposed architecture improvements. a) Full parallel LOAD and STNC instructions and full-parallel multipliers; b) Full parallel LOAD and STNC instructions and radix-4 multiplier. c) Full parallel LOAD and STNC instructions and parallel-serial multiplier. d) Column-parallel row-serial LOAD and STNC instructions and parallel-serial multiplier [10].

In Fig. 2.12, the pie charts for the proposed improvements show how the execution time could be redistributed from the current implementation that was depicted in Fig. 2.11. As it can be seen, great improvements may be achieved. In the best case (Fig. 12a), the LOAD, STNC and product instructions become almost neglectable, but the hardware overhead will increase by the fully-parallel array of multipliers. In Fig. 12b, 12c and 12d, the percent of those instructions still remain significantly reduced. Table 2.6 points out the number of clock cycles required for each subroutine to compute the algorithm of Iglesias and Villa taking into account the four cases showed in Table. 2.5.

Table 2.6: Calculation of the clock cycle number in each subroutine for each proposed architecture change

Symbol	Subroutine	Clock cycles Parallel-parallel multiplier	Clock cycles Radix-4 multiplier	Clock cycles Parallel – serial multiplier	Clock cycles P.S. mult. & Row cache
I C	Initial conditions	22	22	22	$21 + \sqrt{n}$
N L	Neuron Load	158	158	158	$154 + 4\sqrt{n}$
M V	Membrane value	102	109	117	117
C S	Cycle per each synapse	$(534 \cdot s)$	$(548 \cdot s)$	$(564 \cdot s)$	$(560 + 4\sqrt{n}) \cdot s$
M O L P	Memory of last post-synaptic	61	68	76	76
S U	Spike Update	70	70	70	70
B A	Background activity	94	101	109	$108 + \sqrt{n}$
R P	Refractory period	10	10	10	10
N S	Neuron save	101	101	101	$97 + 4\sqrt{n}$
S E	Spike enable	8	8	8	8

Table 2.7: Performance improvement ratio for the proposed architecture changes (100 neurons and 300 synapses)

Type of improvement	Estimated execution time of the proposed architecture	Improvement ratio
Full parallel LOAD and STNC instructions & Full parallel multiplier	160826	3.3
Full parallel LOAD and STNC instructions & Radix-4 multiplier	165047	3.2
Full parallel LOAD and STNC instructions & Parallel-serial	169871	3.1
Parallel-serial & row cache	180761	2.9
Current implementation	540509	1

Considering LOAD and STNC parallel instructions, for any multiplier, the number of cycles follows the eq. (2.12) form:

$$N_T = K_1 + K_2 S \tag{2.12}$$

Notice that N_T does not grow anymore with the number of neurons. Of course, the expression is limited by the feasibility of block storage memory of all the synapse parameters in the synapse loop time. The values of the constants K_2 and K_2 are shown in the Table 2.8.

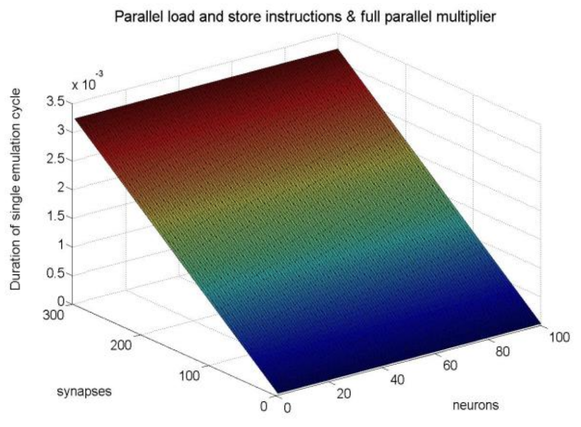
Table 2.8: Values of the constants K_1 and K_2 of equation 2.12 for the full parallel LOAD and STNC instructions and the three types of multipliers

Type of improvement	K_1	K_2
Full parallel multiplier	626	534
Radix-4 multiplier	647	548
Parallel-serial	671	564

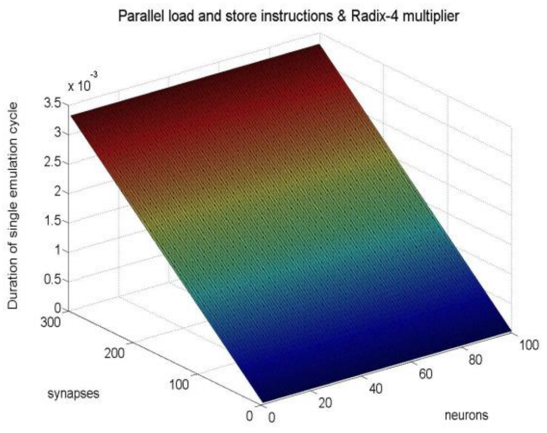
In case of column-parallel row-serial LOAD and STNC instructions (row cache) and parallel-serial multiplier, N_T takes the form of eq. (2.13), where the growth depends also on the square root of the number of neurons.

$$N_T = 661 + 10\sqrt{N} + 560S + 4S\sqrt{N} \tag{2.13}$$

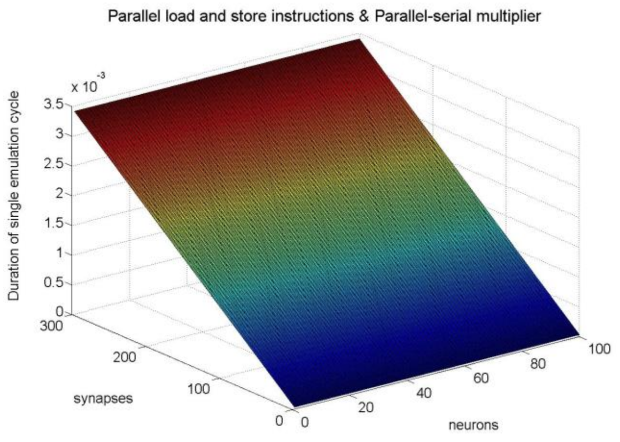
Figure 2.11 shows the execution time for a single simulation step by considering the four improvements mentioned above. The maximum number of neurons and synapses per neuron is assumed to be 100 and 300 respectively. These values were the target values of the PERPLEXUS project [1]. The number of clock cycles for the emulation of Iglesias and Villa model is calculated by using the equation 2.12, from which the values of K_1 and K_2 are determined and these are tabulated in Table 2.8. Equation 2.13 computes the number of clock cycles taken by the neural model of case (d).



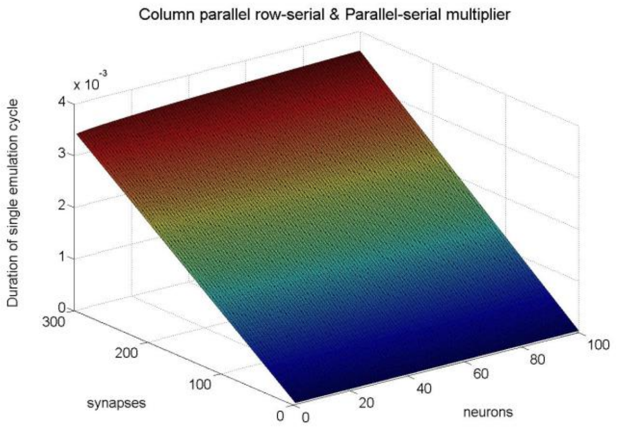
a)



b)



c)



d)

Figure 2.13: a) The execution time required to perform the Iglesias and Villa algorithm in a single step simulation, by taking into account the proposed architecture improvements. a) Full parallel LOAD and STNC instructions and full-parallel multipliers; b) Full parallel LOAD and STNC instructions and radix-4 multiplier. c) Full parallel LOAD and STNC instructions and parallel-serial multiplier. d) Column-parallel row-serial LOAD and STNC instructions and parallel-serial multiplier.

From the graphs shown in Figure 2.13, the value of the single emulation cycle was found to be in the range of 3.2 to 3.5 ms. From the results obtained, it has to be understood that the amendments proposed are not adequate enough to achieve an emulation cycle value of 1ms in case of Iglesias and Villa model which was one of the goals of PERPLEXUS project [1]. Hence further changes are to be thought and implemented in Ubichip in order to support the design of this neural model.

2.4 Performance evaluation – Izhikevich model

This section presents the evaluation of the Izhikevich model which has become quite popular for the simulation of Spiking Neural Networks [3]. This is because the SNN model exhibits various spiking bursting behaviours of cortical neuron. The Izhikevich model is claimed to be computationally as efficient as the integrate-and-fire model, and also this model require minimum hardware to emulate a large number of neurons [11, 12]. The algorithm, which has been written in assembler code for Ubichip, is provided in Annexure C.

2.4.1 Izhikevich model description

Izhikevich presented a simple spiking model (2.14), (2.15). The author has reduced many biophysically accurate Hodgkin–Huxley-type neuronal models to a two dimensional (2-D) system of ordinary differential equations of the form:

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{2.14}$$

$$u' = a\left(bv - u\right) \tag{2.15}$$

if

$$v \geq 30mV, then \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{2.16}$$

Here, v and u are dimensionless variables, and a, b, c, and d are dimensionless parameters, and $' = \frac{d}{dt}$, where t is the time. The variable v represents the membrane potential of the neuron and u represents a membrane recovery variable, which accounts for the activation of K^+ ionic currents and inactivation of Na^+ ionic currents, and it provides negative feedback to v. After the spike reaches its apex (+30 mV), the membrane voltage and the recovery variable are reset according to the (2.16). Synaptic currents or injected dc-currents are delivered via the variable I.

- Parameter a describes the time scale of the recovery variable u. Smaller values result in slower recovery.
- Parameter b describes the sensitivity of the recovery variable u to the sub-threshold fluctuations of the membrane potential v. Greater values couple v and u more strongly resulting in possible sub-threshold oscillations and low-threshold spiking dynamics. A typical value is b=0.2.
- Parameter c describes the after-spike reset value of the membrane potential v caused by the fast high-threshold K^+ conductances. A typical value is c = -65mV.

- Parameter d describes after-spike reset of the recovery variable u caused by slow high-threshold Na^+ and K^+ conductances. A typical value is d = 2.

Depending on the values of the parameters (a, b, c, d), the model can exhibit firing patterns of all known types of cortical neurons [3].

2.4.2 Izhikevich model implementation

The Izhikevich model implemented in this work does not include the STDP, dendrites and axon delays [3]. This is because the mechanism to implement the STPD and axon delay, and dendrites delay proposed by Izhikevich requires excessive hardware resources which makes the implementation very expensive in terms of communication and memory system [11, 12]. The Izhikevich model which includes the modelling of the STDP in the synapses is described in [13]. This model takes into account the delay in the axon and in the dendrites. The STDP rule is based on the Hebbian learning, which is a temporal correlation between the spikes of the pre and post synaptic neurons. In the case that repeated presynaptic spikes arrive before the neuron fires, its synapses stay in long-term potentiation (LTP), in the contrary case, after the neuron fires its synapses leads in long-term depression (LTD).

The implementation of the LTP and LTD in the Izhikevich model [13] is described by the following equations:

- LTP

$$\mathrm{sd(t)} = (\,\mathrm{t} + \Delta \mathrm{t}_1) * (\mathrm{sd(t)} + \mathrm{STDP(t)}) \tag{2.17}$$

- LTD

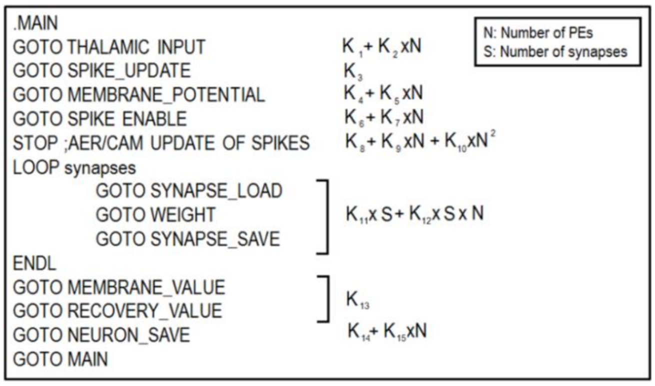
$$\mathrm{sd(t)} = \mathrm{Sj}(\,\mathrm{t} + \Delta \mathrm{t}_1) * (\mathrm{sd(t)} - 1.2 * \mathrm{STDP(t)}) \tag{2.18}$$

- Variable STDP

$$\mathrm{STDP\ (t) = STDP\ (t + 20)} \tag{2.19}$$

The variable sd is in charge of doing the correlation between spikes of the pre and post synaptic neurons in this model [13]. The LTP and LTD mechanisms use a common parameter called STDP (equations 2.17 and 2.18). The STDP variable is an exponential decay variable which is set to the value 0.1 when the postsynaptic neuron fires. The STDP variable takes into account the axon delay which correspond to the value of 20 ms. As can been seen in equation 2.19, the STDP (t) value takes the value of the same variable after 20 ms. The LTP and LTD takes into account the delay in the dendrites, which is indicated by $\Delta \mathrm{t}_1$. The values of $\Delta \mathrm{t}_1$ are generated by a random function and the range of these values is defined between 0 ms and 5 ms. Obviously, the future spike timing information is not available when the pre-synaptic spike S_j arrives because this process has not occurred yet. Therefore, the STDP mechanism proposed by Izhikevich requires high bandwidth memory to store the value of synaptic variables at different instants of time and a high bandwidth communication to exchange the parameter STDP between synapses. The implementation of this complex STDP mechanism demands a lot of hardware resources, so that many hardware emulators do not implement it in their platforms [12, 14]. It is important to be noted that the role of the synaptic plasticity is linked to the biological process of learning.

The proposed emulation of Izhikevich model is carried out by means of these seven subroutine calls and a synaptic loop as shown in Fig. 2.14. The synaptic loop is composed of 3 additional calls. The number of times the loop is executed is equal to the number of synapses. The constants K_1 to K_{15} allow the calculation of the number of clock cycles required to execute each procedure. In some cases the number of clock cycles is calculated as a function of the number of neurons and the number of synapses.



```
.MAIN                                                      N: Number of PEs
GOTO THALAMIC INPUT              $K_1 + K_2 xN$            S: Number of synapses
GOTO SPIKE_UPDATE                $K_3$
GOTO MEMBRANE_POTENTIAL          $K_4 + K_5 xN$
GOTO SPIKE ENABLE                $K_6 + K_7 xN$
STOP ;AER/CAM UPDATE OF SPIKES   $K_8 + K_9 xN + K_{10} xN^2$
LOOP synapses
          GOTO SYNAPSE_LOAD   ]
          GOTO WEIGHT         ]  $K_{11} x S + K_{12} x S x N$
          GOTO SYNAPSE_SAVE   ]
ENDL
GOTO MEMBRANE_VALUE           ]
GOTO RECOVERY_VALUE           ]  $K_{13}$
GOTO NEURON_SAVE                 $K_{14} + K_{15} xN$
GOTO MAIN
```

Figure 2.14: Main program of the SNN emulation assembly code

2.4.3 Performance figures

The value of the constants K for each subroutine is shown in Table 2.9. Table 2.10 shows the encoding of subroutines contained inside of the synapse loop.

Table 2.9: Main loop subroutine encoding and execution number of clock cycles

Symbol	Subroutine	Clock cycles*
T I	Thalamic Input	36 + N
S U	Spike Update	36
M P	Membrane Potential	52 + N
S E	Spike Enable	6 + N
C S	Cycle per each synapse	(128 + 8N)S
M V	Membrane Value	9840
R V	Recovery Value	1516
N S	Neuron save	6 + N

* The number of clock cycles depends on the number of synapses (S) and number of neurons (N).

Table 2.10: Synapse loop routine encoding

Symbol	Subroutine	Clock cycles *
S L	Synapse Load	1 + 4N
S W	Synaptic weight	111
S S	Synapse Save	16 + 4N

* The number of clock cycles depends on the number of neurons (N).

The equation 2.20 allows the calculation of the number of clock cycles required for the execution of phase 1 in one simulation step. This equation is obtained from the addition of all contributions of the Table 2.9:

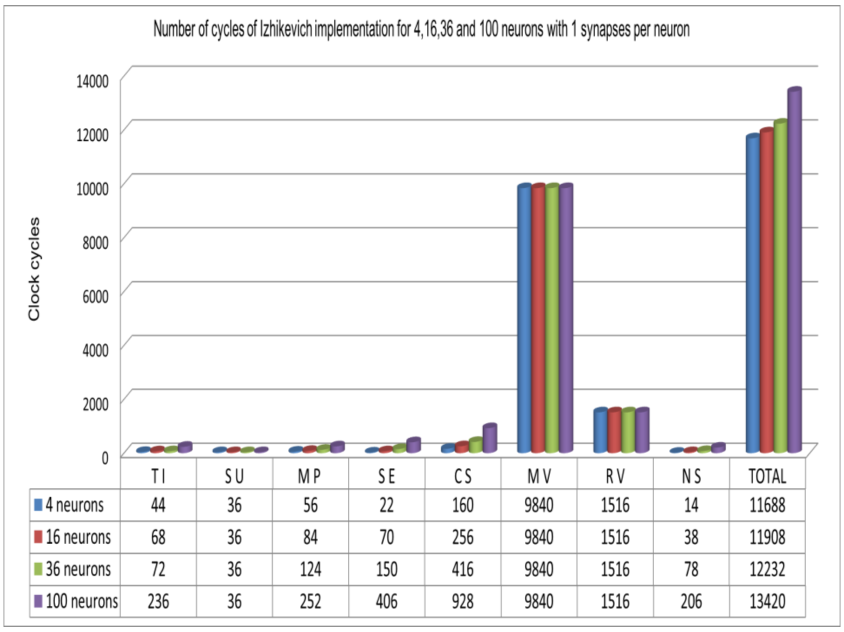
$$N_T = 11492 + 10xN + 128xS + 8xNxS \tag{2.20}$$

Table 2.11: Execution time of one simulation cycle for different SNN size

Array	# PE	# Synapse	# Chip	Processing phase (clock cycles)				FCK	Total phase 1	Spiking phase		Total phase 2	**TOTAL**	**SPIKE RATE**
N	N	S	C	K1	K2*N	K3*S	K4*N*S	(MHz)	(ms)	AER	(MHz)	(ms)	**(ms)**	**(s^{-1})**
2x2	4	2	1	11492	40	2784	64	50	0.2876	8	5	0.0016	**0.29**	**3458**
2x2	4	3	1	11492	40	4176	96	50	0.31608	8	5	0.0016	**0.32**	**3148**
6x6	36	8	1	11492	360	11136	2304	50	0.50584	40	5	0.008	**0.51**	**1946**
6x6	36	12	1	11492	360	16704	3456	50	0.64024	40	5	0.008	**0.65**	**1543**
10x10	100	300	1	11492	1000	417600	240000	50	13.40184	104	5	0.0208	**13.42**	**75**
100x100	100	300	100	11492	1000	417600	240000	50	13.40184	10202	5	2.0404	**15.44**	**65**

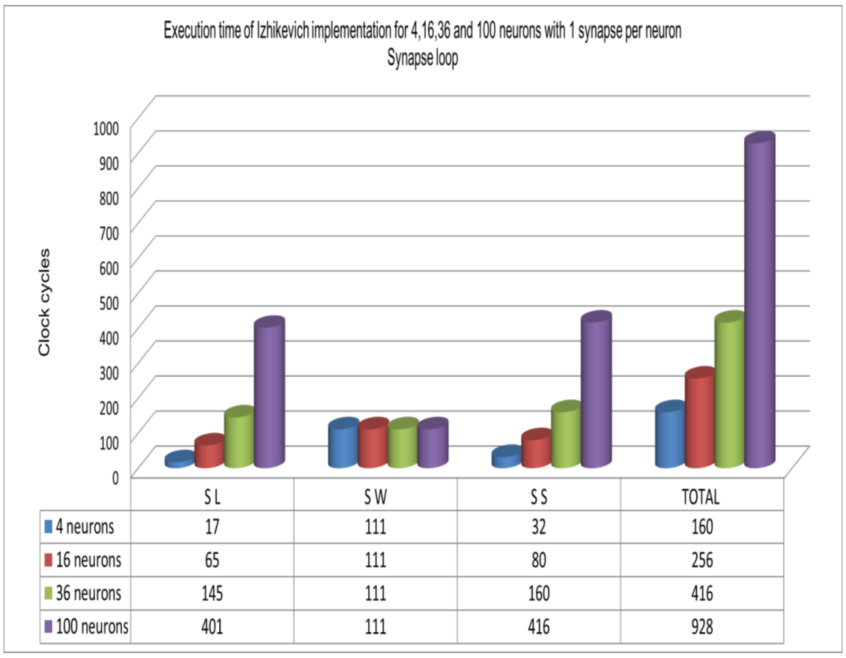
Table 2.11 shows the calculated execution times for different SNN emulation array sizes using the expression (2.20). The calculation considers the spike distribution time and the processing time. The execution time dedicated to processing phase is about 7 times longer than the phase of distribution. This is taking into account the proposed target in PERPLEXUS project [1] (300-synapse 10000-neuron, 100-ubichip). The spike distribution phase was calculated assuming that all neurons will be spiking at every simulation step. As it was indicated in previous analysis the spike rate was very close to the proposed target in PERPLEXUS project [1] (average spiking rate: 100-200 spikes/second) by obtaining 65 spike/s rate. Therefore, the following study will be dedicated to analyze the performance of the processing phase.

The method used to analyze the performance of the Izhikevich model has also been used for the Iglesias and Villa model. This method consists of two approaches to measure the number of clock cycle. First, by varying the size of the array (2x2, 4x4, 6x6 and 10x10) with one synapse per neuron, and second by varying the number of synapses (3, 30 and 300 per neuron), but keeping the number of neurons as a constant (100). The results show the measurement of the SNN algorithm in one-step emulation and they have been verified for consistency with eq. (2.20). The proposed strategy has helped us to understand how every subroutine call is affected either by increasing the number of neurons or synapses.



	T I	S U	M P	S E	C S	M V	R V	N S	TOTAL
🟦 4 neurons	44	36	56	22	160	9840	1516	14	11688
🟥 16 neurons	68	36	84	70	256	9840	1516	38	11908
🟩 36 neurons	72	36	124	150	416	9840	1516	78	12232
🟪 100 neurons	236	36	252	406	928	9840	1516	206	13420

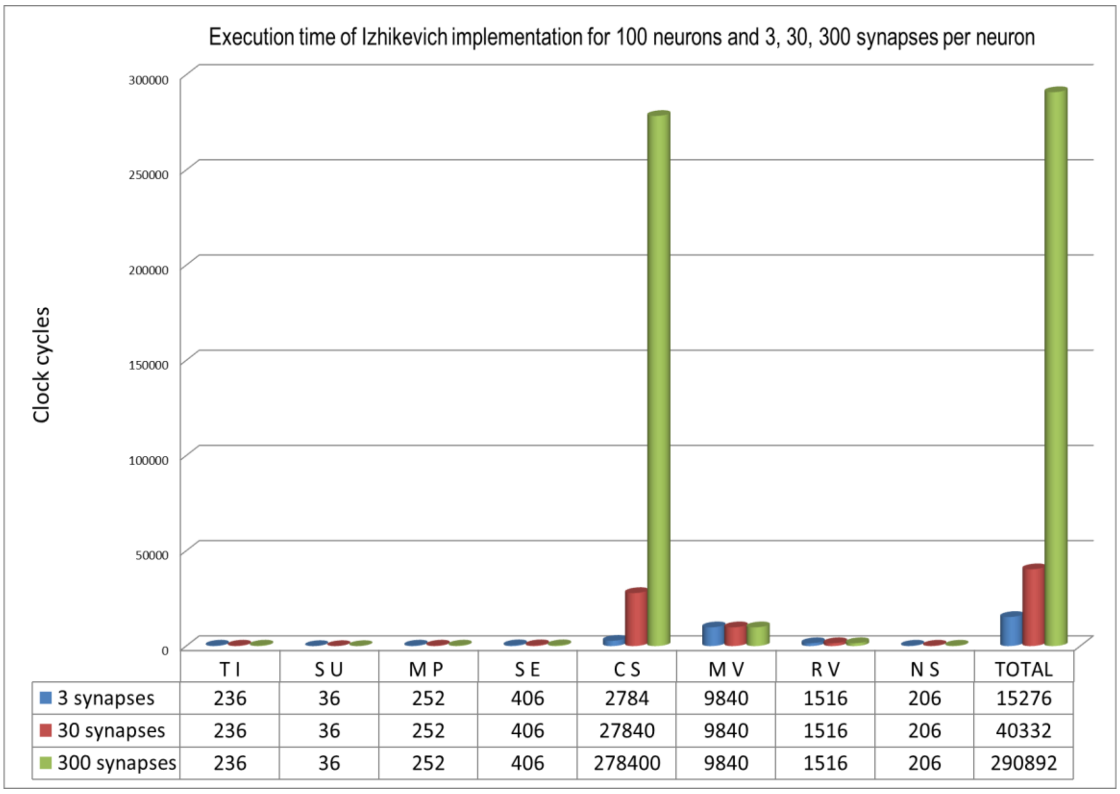
Figure 2.15: Required number of cycles for the execution of Izhikevich implementation for 4, 16, 36 and 100 neurons with 1 synapse per neuron.



	S L	S W	S S	TOTAL
🟦 4 neurons	17	111	32	160
🟥 16 neurons	65	111	80	256
🟩 36 neurons	145	111	160	416
🟪 100 neurons	401	111	416	928

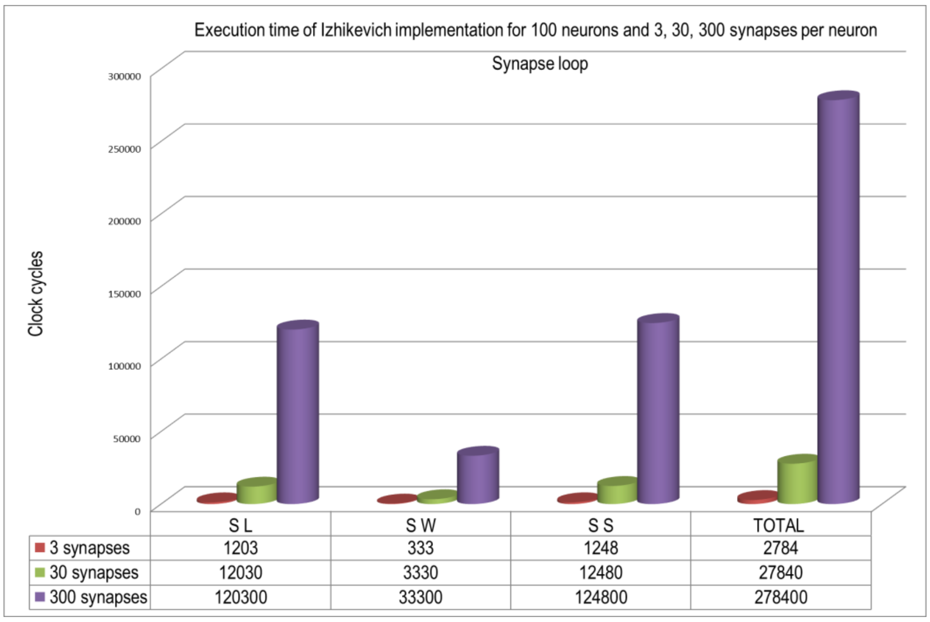
Figure 2.16: Required number of cycles for the execution of the synapse loop for 4, 16, 36 and 100 neurons with 1 synapse per neuron.

Figure 2.15 shows the required number of clock cycles per emulation step for the four configurations. It is clear that the major number of cycles is dedicated to execute the membrane voltage. This is due to the number of multiplications required to calculate it. The initial proposal for the implementation of the multiplication was to carry out by software in order to save area in the FPGA. The cost of this implementation is high in terms of processing speed because the saturated multiplication executed by software requires a lot of clock cycles as it can be observed in Figure 2.15 in the calculation of the membrane voltage (eq. 2.16). Another delay, which becomes important when the number of synapses is increasing, is the synapse cycle (SC). SL (synapse load) and SS (synapse save) provide the major contribution of the total delay as it is shown in Fig. 2.18. A better case to observe the impact produced by increasing the number of synapses per neurons is shown in Fig. 2.17, where the number of clock cycles increases linearly. In the case of 100-neurons 300-synapse neuron array it can be clearly observed that 99.5% of the clock cycles are dedicated to synapse cycle. Fig. 2.18 shows the contribution to CS for each subroutine inside the synapse loop in order to detect the subroutines that are making the major contribution to the delay. The identified subroutines are dedicated to storing and loading synaptic parameters.



	T I	S U	M P	S E	C S	M V	R V	N S	TOTAL
🟦 3 synapses	236	36	252	406	2784	9840	1516	206	15276
🟥 30 synapses	236	36	252	406	27840	9840	1516	206	40332
🟩 300 synapses	236	36	252	406	278400	9840	1516	206	290892

Figure 2.17: Required number of cycles for the execution of Iglesias-Villa SNN for 3, 30 and 300 synapses on a 100 neuron array.



	S L	S W	S S	TOTAL
🟥 3 synapses	1203	333	1248	2784
🟩 30 synapses	12030	3330	12480	27840
🟪 300 synapses	120300	33300	124800	278400

Figure 2.18: Required number of cycles for the execution of the synapse loop for 3, 30 and 300 synapses on a 100 neuron array.

2.4.4 Architecture limitations

The previous performance analysis indicates that the memory access system and the execution of arithmetic operations abate the effectiveness of the SIMD architecture for the computation of the Izhikevich model when the number of neurons and synapses to be emulated is increased. The improvements that were proposed in Section 2.3.4 are also considered in order to boost the processing power of this model.

Table 2.12: Clock cycles devoted to LOAD, STORE, multiplication with saturation, addition with saturation and all other instructions for the execution of Izhikevich algorithm (100 neurons and 300 synapses per neuron)

Instructions and subroutine	#	# cycles/instruction	Total number of cycles
LOAD NEURONS	2	100	200
LOAD SYNAPSES	600	100	60000
STNC NEURONS	2	100	200
STNC SYNAPSES	600	100	60000
MULT. NEURONS	10	642	6420
ADDER NEURONS	10	95	950
ADDER SYNAPSES	300	95	28500
REMAINING INSTRUCTIONS			156702
		TOTAL	312972

Table 2.12 shows the number of clock cycles required to execute the LOAD, STORE (conditional store family), and multiplication and adder instructions. Also the remaining instructions are considered in order to complete the Izhikevich algorithm.

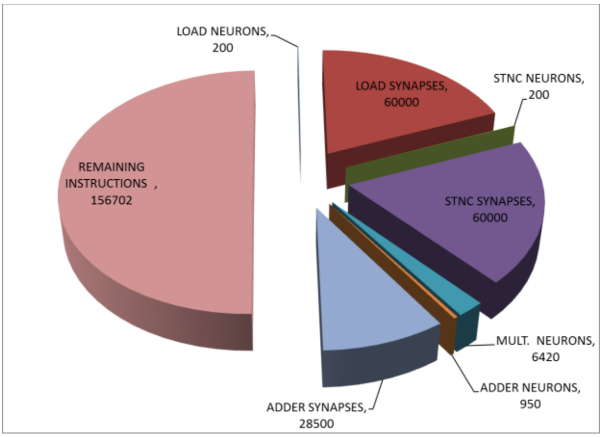
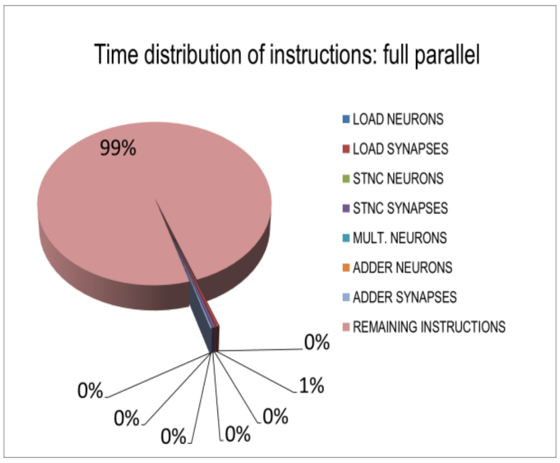
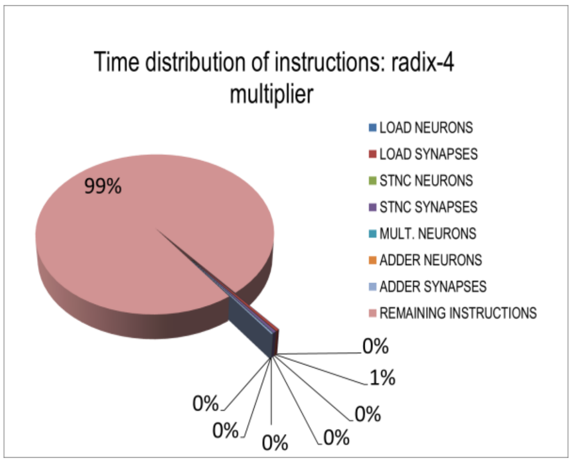


Figure 2.19: Clock cycle number distribution of instructions as classified in Table 2.12

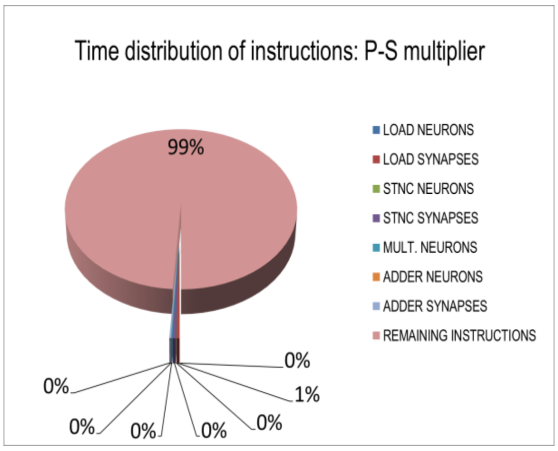
The pie chart of the Fig. 2.19 displays the distribution of clock cycles required for the instructions and subroutines indicated in Table 2.12. Almost the half of the total of the clock cycles are dedicated to executing the memory access and arithmetic operations. These figures indicate the need for implementing strategies to improve the performance of the system by keeping low-area consumption, in order to allocate a large number of neurons in a single FPGA.

The possible changes are shown in Table 2.13. These modifications are based on the implementation of three types of multipliers, saturation adders and distributed memory architecture. Therefore, fully-parallel hardware multiplier and fully parallel LOAD and STORE instructions would be able to carry out in a single clock cycle.

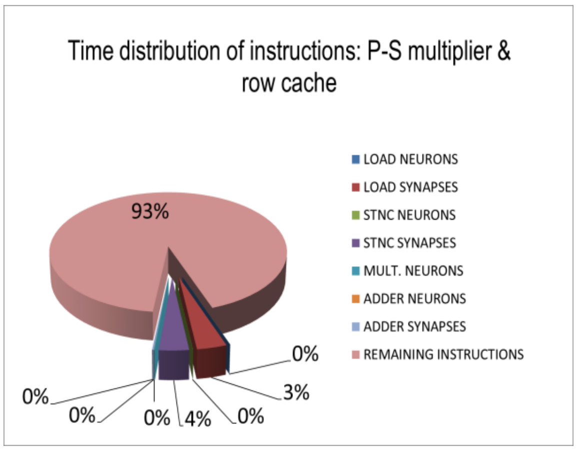




(a) (b)



(c)



(d)

Figure 2.20: a) Clock cycle number distribution of instructions taking into account the proposed architecture improvements. a) Full parallel LOAD and STNC instructions and full-parallel multipliers; b) Full parallel LOAD and STNC instructions and radix-4 multiplier; c) Full parallel LOAD and STNC instructions and parallel-serial multiplier; d) Column-parallel row-serial LOAD and STNC instructions and parallel-serial multiplier.

Figure 2.20 shows the distribution of the clock cycles for each instruction or subroutine which were indicated in Table 2.10 taking account the four proposed configurations. The best case (see Fig. 2.20 a)) represents an important improvement because LOAD, STNC, addition and product instructions are reduced significantly. For the remaining cases, a negligible increment in the percentage of these instructions is gained as it can be observed in Fig. 2.20 b), 2.20 c) and 2.20 d) respectively.

Table 2.13: Clock cycle number calculation for modified architectures

	Best case	radix-4 mult.	P-S mult.	P-S mult. & row cache	current architecture
Instructions and subroutine	# cycles	# cycles	# cycles	# cycles	# cycles
LOAD NEURONS	2	2	2	20	200
LOAD SYNAPSES	600	600	600	6000	60000
STNC NEURONS	2	2	2	20	200
STNC SYNAPSES	600	600	600	6000	60000
MULT. NEURONS	10	80	160	160	6420
ADDER NEURONS	10	10	10	10	950
ADDER SYNAPSES	300	300	300	300	28500
REMAINING INSTRUCTIONS	156702	156702	156702	156702	156702
TOTAL	158226	158296	158376	169212	312972

Table 2.14 shows the calculation of clock cycle number for each subroutine by considering the proposed improvements. Table 2.15 indicates the estimated performance improvement ratio taking into account these possible modifications.

Table 2.14: Calculation of the clock cycle number in each subroutine for each proposed architecture change

Symbol	Subroutine	Clock cycles Parallel-parallel multiplier	Clock cycles Radix-4 multiplier	Clock cycles Parallel – serial multiplier	Clock cycles P.S. mult. & Row cache
T I	Thalamic Input	37	37	37	$36 + \sqrt{N}$
S U	Spike Update	36	36	36	36
M P	Membrane Potential	53	53	53	$52 + \sqrt{N}$
S E	Spike Enable	7	7	7	$6 + 2 \cdot \sqrt{N}$
C S	Cycle per each synapse	$(136 \cdot S)$	$(136 \cdot S)$	$(136 \cdot S)$	$(128 + 4 \cdot \sqrt{N}) \cdot S$
M V	Membrane Value	67	88	112	112
R V	Recovery Value	46	60	76	76
N S	Neuron save	101	101	101	$6 + \sqrt{N}$

Table 2.15: Performance improvement ratio for the proposed architecture changes (100 neurons and 300 synapses)

Type of improvement	Estimated execution time of the proposed architecture	Improvement ratio
Full parallel multiplier	41147	7.6
Radix-4 multiplier	41182	7.5
Parallel-serial	41222	7.5
Parallel-serial & row cache	50774	6.1
Current implementation	312972	1

Considering LOAD and STNC parallel instructions, for any multiplier, and the saturated adder in hardware, the number of cycles follows the eq. (2.21) form:

$$N_T = K_1 + K_2 S \tag{2.21}$$

Notice that N_T does not grow anymore with the number of neurons. Of course, the expression is limited by the feasibility of block storage memory of all the synapse parameters in the synapse loop time. The values of the constants K_2 and K_2 are shown in the Table 2.16.

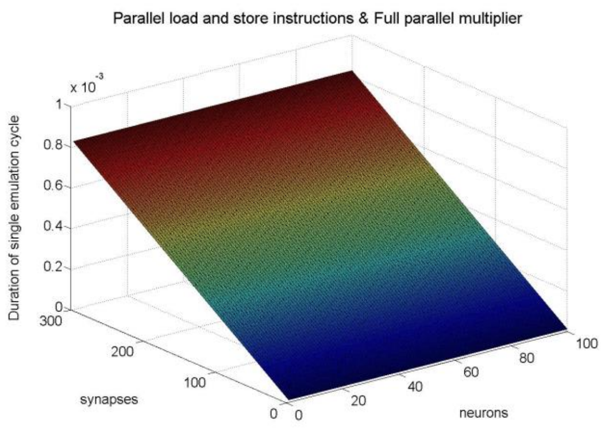
Table 2.16: Values of the constants K_1 and K_2 of equation 2.13 for the full parallel LOAD and STNC instructions and the three types of multipliers

Type of improvement	K_1	K_2
Full parallel multiplier	347	136
Radix-4 multiplier	382	136
Parallel-serial	422	136

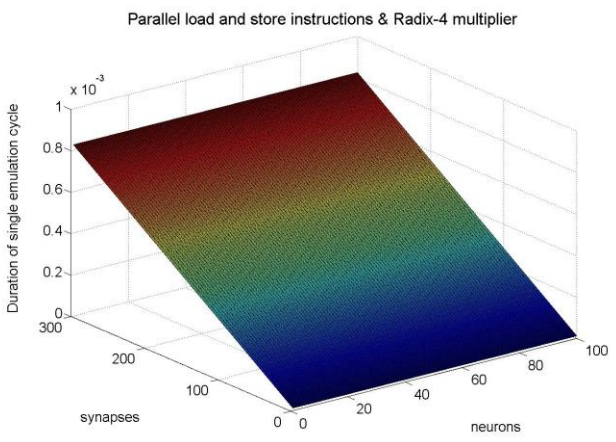
In case of column-parallel row-serial LOAD and STNC instructions (row cache) and parallel-serial multiplier, N_T takes the form of eq. (2.22), where the growth depends on the square root of the number of neurons.

$$N_T = 324 + 5\sqrt{N} + 128S + 4S\sqrt{N} \tag{2.22}$$

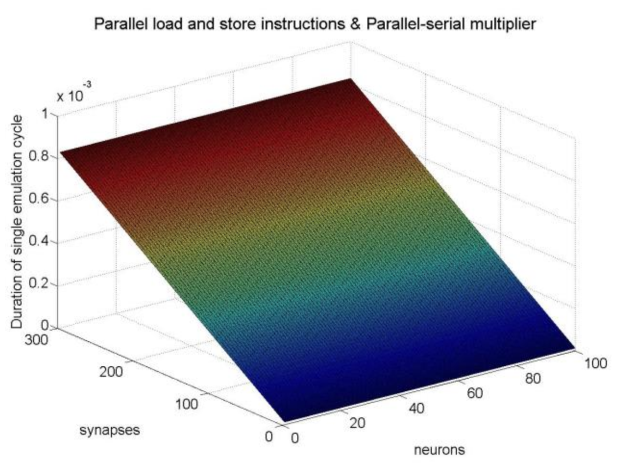
Figure 2.21 shows the execution time for a single simulation step by considering the four improvements mentioned above. The maximum number of neurons and synapses per neuron is assumed to be 100 and 300 respectively. These values were the target values of the PERPLEXUS project [1]. The number of clock cycles for the emulation of Izhikevich model is calculated by using the equation 2.21, from which the values of K_1 and K_2 are determined and these are tabulated in Table 2.16. Equation 2.22 computes the number of clock cycles taken by the neural model of case (d). The product of the number of emulation cycles with that of the clock cycles will provide the execution time. Hence this execution time is being calculated and found to be 50MHz.



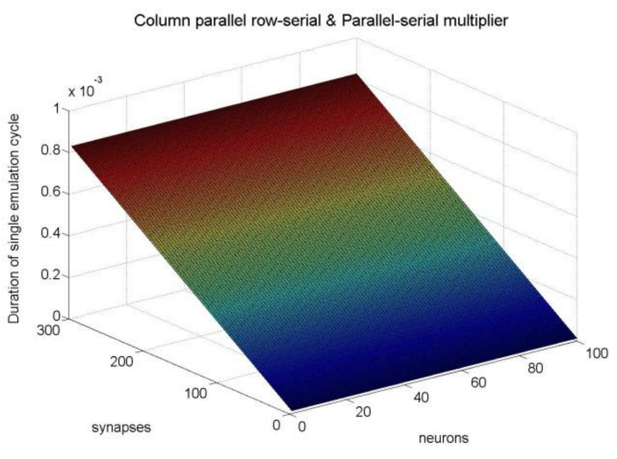
a)



b)



c)



d)

Figure 2.21: a) The execution time required to perform the Izhikevich algorithm in a single step simulation, by taking into account the proposed architecture improvements. a) Full parallel LOAD and STNC instructions and full-parallel multipliers; b) Full parallel LOAD and STNC instructions and radix-4 multiplier. c) Full parallel LOAD and STNC instructions and parallel-serial multiplier. d) Column-parallel row-serial LOAD and STNC instructions and parallel-serial multiplier.

From the graphs shown in Fig. 2.21 the value of the single emulation cycle was found to be in the range of 0.82 to 0.85 ms. According to Izhikevich, the model has to be executed for every 1ms. But this can be implemented only by taking into account the four changes proposed in the Ubichip architecture.

2.5 Conclusions

A detailed performance analysis of two SNN models with different levels of computational complexity has been carried out in the previous PERPLEXUS SNN multi-model architecture called Ubichip. The study of the performance has been evaluated in clock cycles only to give some real figures at the end it has been particularized for the nominal 50 MHz operation. Results show that the target objective in both cases (Iglesias-Villa model and Izhikevich model) has not been reached (12.85 milliseconds and 15.44 milliseconds respectively), where the target of the project is to simulate SNN models under 1 millisecond step time resolution. Some improvements were analysed to increase the performance of the Ubichip. These improvements involve different topologies of the memory, different type of multipliers. These improvements try to abate the main bottlenecks in Ubichip architecture.

Beyond the performance increase by means of frequency clock boosting, the inclusion of hardware multipliers in the current architecture and reduction in the RAM access bottleneck by introducing block RAMs and dedicated instruction memory improves the performance of Iglesias and Villa model and Izhikevich by a factor of 3 and 7 respectively, by considering the simulation of 100 neurons and 300 synapses per neuron in a single Ubichip. The improvement in the processing speed of the current architecture could guarantee a step time resolution of 1 ms in the emulation of Izhikevich model [3], taking into account the target (10000 neurons 300 synapses per neuron) which was proposed in the PERPLEXUS project [1].

In fact, given the simple architecture of the PE in the multiprocessor mode, larger PE arrays could be easily implemented with currently available CMOS technologies, and higher operation frequency could be easily achieved. Therefore, the performance would increase to 4 fold (50 MHz – 200 MHz) as the clock used in the modern FPGAs works at 200MHz.

References

[1] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas, "The Perplexus bio-inspired reconfigurable circuit," in Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on, 2007, pp. 600-605.

[2] J. Iglesias, J. Eriksson, F. Grize, M. Tomassini, and A. E. P. Villa, "Dynamics of pruning in simulated large-scale spiking neural networks," Biosystems, vol. 79, 2005.

[3] E. M. Izhikevich, "Simple model of spiking neurons," Neural Networks, IEEE Transactions on, vol. 14, pp. 1569-1572, 2003.

[4] Flynn, M.J., Rudd, K.W, "Parallel architectures. ACM Computation Surveys", pp. 67–70, 1996.

[5] Ferscha, "Parallel and distributed simulation of discrete event systems", Parallel and Distributed Computing Handbook, McGraw-Hill, pp. 1003–1041, 1996.

[6] J. M. Moreno and J. Madrenas, "A reconfigurable architecture for emulating large-scale bio-inspired systems," in Evolutionary Computation, 2009. CEC '09, IEEE Congress on, pp. 126-133, 2009.

[7] J. M. Moreno, J. Madrenas, and L. Kotynia, "Synchronous Digital Implementation of the AER Communication Scheme for Emulating Large-Scale Spiking Neural Networks Models," in Adaptive Hardware and Systems, NASA/ESA Conference on, pp. 189-196, 2009.

[8] Jordi Madrenas, "Specification of the Ubichip Sequencer," Internal report WP2-T2.3-UPC-1, Version 2.0, June 15, 2009.

[9] Michael Hauptvogel, "Design of a bio-inspired spiking network environment," Thesis of Master in network centre computing high performance computing and communication, Faculty of science, The university of reading, 17 of march 2008.

[10] G. Sanchez, J. Madrenas, and J. Moreno, "Performance Evaluation and Scaling of a Multiprocessor Architecture Emulating Complex SNN Algorithms," in Evolvable Systems: From Biology to Hardware, vol. 6274: Springer Berlin Heidelberg, pp. 145-156, 2010.

[11] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation," in Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, pp. 133-140, 2012.

[12] J. Xin, Luja, x, M. n, L. A. Plana, S. Davies, et al., "Modeling Spiking Neural Networks on SpiNNaker," Computing in Science & Engineering, vol. 12, pp. 91-97, 2010.

[13] E. M. Izhikevich, "Polychronization: Computation with Spikes," Neural Comput., vol. 18, pp. 245-282, 2006.

Development of a data interface between Analog and Digital Neuromorphic systems

3

3.1 Introduction

This chapter presents the development of data interface in order to achieve the communication between analog and digital multi-processor implementations for bio-inspired processing of sensors. This combination allows us to create a bio-inspired multiple-input sensor processing system for applications that mimic the perception of the environment such as vision, hearing or other modalities. This work has been a part of the Neural and Self-adaptive Sensory Integration for Environment-Perception Embedded Systems (NESSIE2) project [1]. The main motivation behind this project is to develop a pre-processing integrated system of physical information in order to be connected to the Digital Multi-Processor (DMP) system or as well known as Ubichip, which was developed as part of the PERPLEXUS project [2]. So that the DMP may be able of acquiring information from the environment and perception applications could be implemented by using this bio-inspired system (analogue – digital systems). Figure 3.1 shows the bio-inspired system proposed in the NESSIE2 project. The pre-processing system consists of three main components which are: the sensors (internal sensors or external sensors), sensor conditioning, and the pre-processing system circuits.

The information-preprocessing circuits are in charge of performing two important tasks, first task is the sensor conditioning and second one is the digitization of the data into a format that is compatible with the DMP (Digital Multi-Processor) by using the Address Event Representation (AER). Therefore, the challenge is to develop a parallel acquisition system i.e. a sensory unit to acquire analog data from multiple sensors and pre-process the raw data with an analog implementation/Analog pre-processor that translates the data from the sensory unit into a data which is compatible with the DMPs (Digital Multi-Processors) for processing.

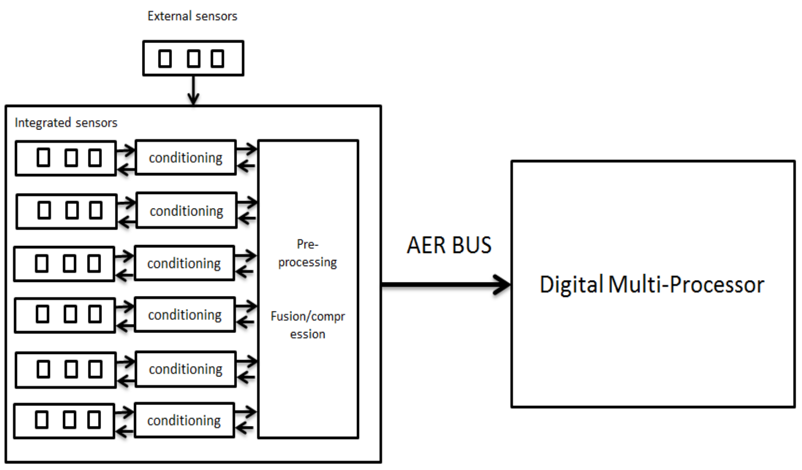


Figure 3.1: NESSIE2 system, drawing partially reproduced from [1]

During the last 5 years, the University of Edinburgh was taking efforts to develop an analog spike coder-decoder prototype [3]. This prototype is suitable to be used as an analog pre-processor in bio-inspired systems. This is because of two reasons. Firstly, the Analog spike coder-decoder is also a bio inspired design which works on the principles of SNN as the DMP. Secondly, it uses the AER communication protocol for spike communication and can be easily interfaced with the DMP. The AER protocol is one of the predominant solutions used in the latest neuromorphic systems in order to prevent interconnection overhead which occurs due to the reduction in the number of physical lines that are required for the interconnection of large neural networks [4]. The collaboration between the Institute of Micro and nano-systems, University of Edinburgh and the Advanced Hardware Architecture Research Group, Universitat Politècnica de Catalunya has shortened the development time.

The analog front-end, which was developed in the University of Edinburgh, encodes the input signal in a signed spike representation, which is further processed by means of a digital Spiking Neural Network (SNN) on a Single-Instruction Multiple-Data (SIMD) multiprocessor. The APP generates positive and negative spikes, but the sign is not used in this work because Ubichip was not designed to process negative spikes, however this information is not discarded and negative spikes are converted to positive spikes. The spike distribution for both the systems is based on Address-Event Representation (AER) scheme. AER is asynchronous for the Analog Pre-Processor (APP) [3] and synchronous for the Digital Multi-Processor (DMP) [5]. Therefore, the analogue pre-processor have been connected to the DMP through data interface which is responsible for the synchronization between two systems.

This chapter gives an outline of this analog-digital neuromorphic system, a brief explanation about the APP, designed by the University of Edinburgh, and a detailed explanation about the functionality of the transceiver interface between the APP and the DMP designed by the Universitat Politècnica de Catalunya and the metrics considered for the design. Finally, real-time results of the communication through the interface are provided.

3.2 The Analogue-Digital neuromorphic system

The system consists of 3 functional units which are the sensory unit, the transceiver unit and the Digital Multi Processors as shown in Fig. 3.2. As specified earlier, the sensory unit consists of the sensors and the Analog Pre-processor. The general idea of the proposed system is to acquire external environment information and process them in two steps. The signals are detected by the sensors (acoustic, infrared, or pressure, etc. which have analogue output). In the first step, the acquired signals are converted to spike-timing representation by the Analog Pre-processor. This module conveys the spikes to DMPs using the AER bus through a transceiver that converts the spikes into a format which the DMPs can comprehend.

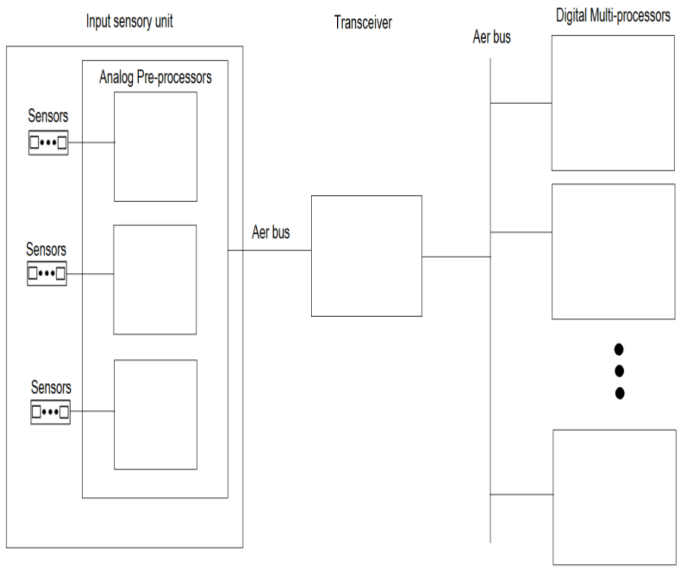


Figure 3.2: Functional block diagram of the proposed analog-digital bio-inspired system

The advantage of using the transceiver as an interface is that the input sensory unit can be changed and any other sensory unit can be replaced in the same system within a short duration of time. It is important to note that these replacements must work under AER protocol [4]. This allows faster prototyping. The transceiver plays a vital role in providing data in appropriate formats and the synchronization of data transactions on both sides. It is important to note that the APP and DMP were designed to support different applications based on Spiking Neural Networks. The APP was developed in order to process the input analog signal and the minimum inter-spike time/ spike transmission rate supported by the APP is in the order of microseconds. While the DMP was developed to emulate complex SNN models that resemble biological neurons and supports an inter-spike time/ spike transmission rate of several milliseconds. Another challenge to synchronization is that the APP works on Asynchronous AER and the DMP on Synchronous AER. These are the most important factors to be focused in order to achieve the synchronization between the APP and the DMP. In the second step, the data translated by the transceiver is analyzed and processed by the DMPs.

3.3 The Analogue Pre-Processor (APP)

The purpose of this section is to give a general description of the APP in order to give a better understanding of the system. This work has been developed by Dr. Luiz Gouveia as part of his PhD thesis in University of Edinburgh [3]. The APP consists of Configurable Analog Blocks (CABs). These CABs are programmed to perform analog computation and signal processing functions as is shown in Fig. 3.3.

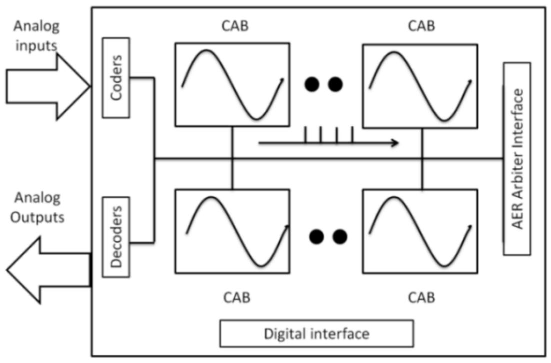


Figure 3.3: Architecture diagram. The array of Configurable Analogue Blocks (CABs) is connected using an asynchronous digital channel, this figure was extracted from [3]

The analog signals are encoded as spikes timed events and are transmitted between CABs via asynchronous AER bus. The coding and decoding process is implemented by spiking coders and decoders respectively, as shown in Fig. 3.4. In the coder, the error signal e(t) is the difference between the analog input x(t) and the reconstructed feedback signal z(t). The comparators in the feed-forward loop compare the error signal e(t) against their respective threshold values and their outputs form the input to a spike generator block. The spike generator block generates a positive spike if the comparator output state $C_1(t)$ goes high and a negative spike if the comparator output $C_2(t)$ is high. The spike events generated by the spike event block are outputted using an asynchronous AER interface. In the feedback loop, which is implemented inside of the coder, the spikes are integrated by the integrator block to form the reconstructed signal z(t). The spike to analog decoder is essentially the feedback path of the spike encoder. At the decoder, a low pass filter is added to the integrator output to improve the resolution of the reconstructed signal.

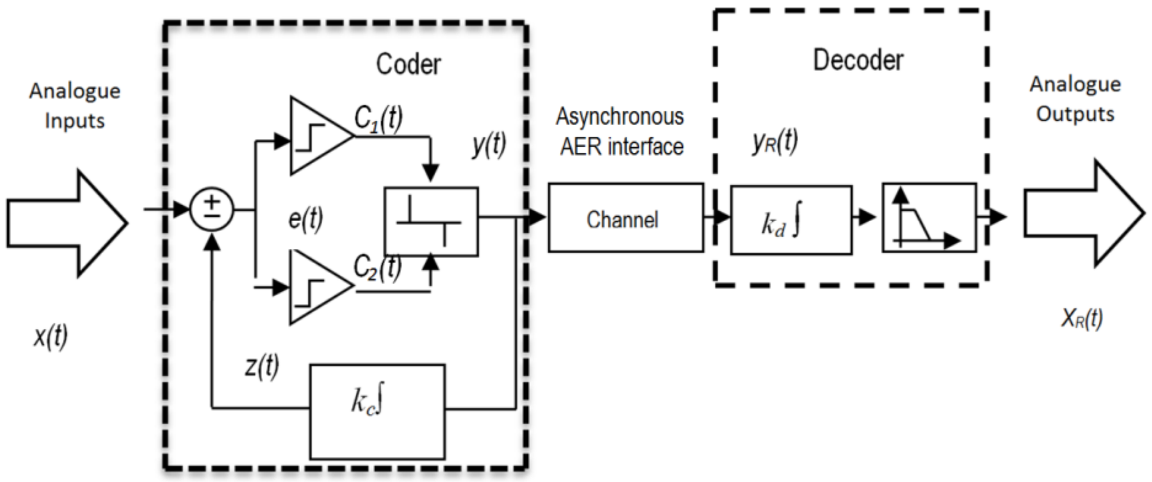


Figure 3.4: Event coding block diagram, this figure was extracted from [3]

3.3.1 AER channel

In the analog pre-processor architecture, an array of spike encoders shares a common digital AER bus through an AER interface. When a spike event is generated by one of the encoder blocks, the AER interface broadcasts asynchronously the digital address of that encoder. Each spike event is therefore identified by the digital address representing the address of the encoder. Usually, in neuromorphic systems, events are represented by two states, i.e. the presence or absence of a spike. In the encoder implementation, the analog signal is represented in the spike domain by three states: a positive spike, a negative spike and an absence of a spike. This extra information on the sign of the spike event is provided by appending an additional MSB bit in the digital address code, where a positive spike is represented by a 1 at the MSB location and a negative spike is represented by 0. For example, when an encoder with a digital address 00 generates a positive spike, the output at the AER bus will be represented by the digital code 100.

3.4 Transceiver functional description

In Subsections 2.2 and 3.3 the basic features of the DMP (Ubichip) and the APP were presented respectively. Thus, this section describes the transceiver design to synchronize the communication between DMP and APP using the AER protocol in two different operation modes, synchronous and asynchronous respectively. Synchronous AER is used to distribute spikes but not only inside of each DMP, but also the spikes are broadcasted to external chips through the synchronous AER bus; this feature allows processing external inputs that support the synchronous AER protocol. In order to develop an autonomous system (DMP) capable to interact with outer world, the analogue preprocessing circuit is used to provide such inputs in appropriate data form to DMP through the asynchronous AER. The AER transceiver is the key element that synchronizes the communication between both processors (see Fig. 3.5).

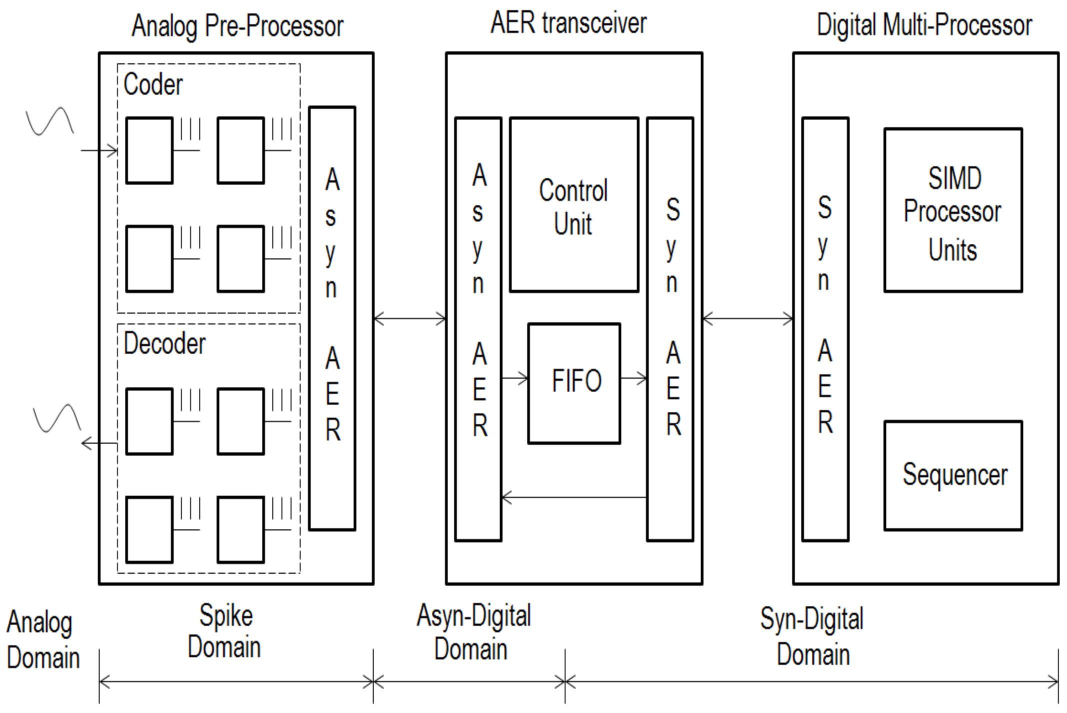


Figure 3.5: Functional block diagram of the analog-digital bio-inspired system

The analog-digital bio-inspired system processes the sensor information through four domains, providing the analysis of characteristics of the signal. The sensor information in the analog domain is acquired by the Analog-Preprocessor, which processes the information in the analog spike domain and converts it into Asynchronous-digital spikes. These spikes are read by the transceiver that converts it into the Synchronous digital spikes. Then these spikes are forwarded to the DMP. But this depends on, if the DMP is in its processing phase or execution phase. If it’s in the processing phase then the spikes are stored in the transceiver, if not then the spikes are forwarded to the DMP (see section 2.2.1). The transceiver acts as a master that controls spikes to be transmitted to the DMP by using the synchronous AER protocol. According to the AER protocol defined and implemented in the DMP [5], the connection between several DMPs through a common AER bus forms the DMP network. In an only-DMP network, the distribution of the spikes is started by the master DMP. For this particular case, it has been decided that the configuration of the AER transceiver module performs as the AER Master; therefore, other DMPs that are connected to the synchronous AER perform as slaves (for further details see Section 2.2.1.1 in Chapter 2).

From the interfacing point of view, the transceiver does two functions. Firstly, it adapts to the AER protocols in each side. In the current implementation, the APP has a 3-bit asynchronous AER protocol and the DMP an 8-bit synchronous one. Secondly, it buffers the data to be sent to the DMP, when it is not ready to accept spikes (in phase 1) by means of FIFO (First-In First-Out) stacks.

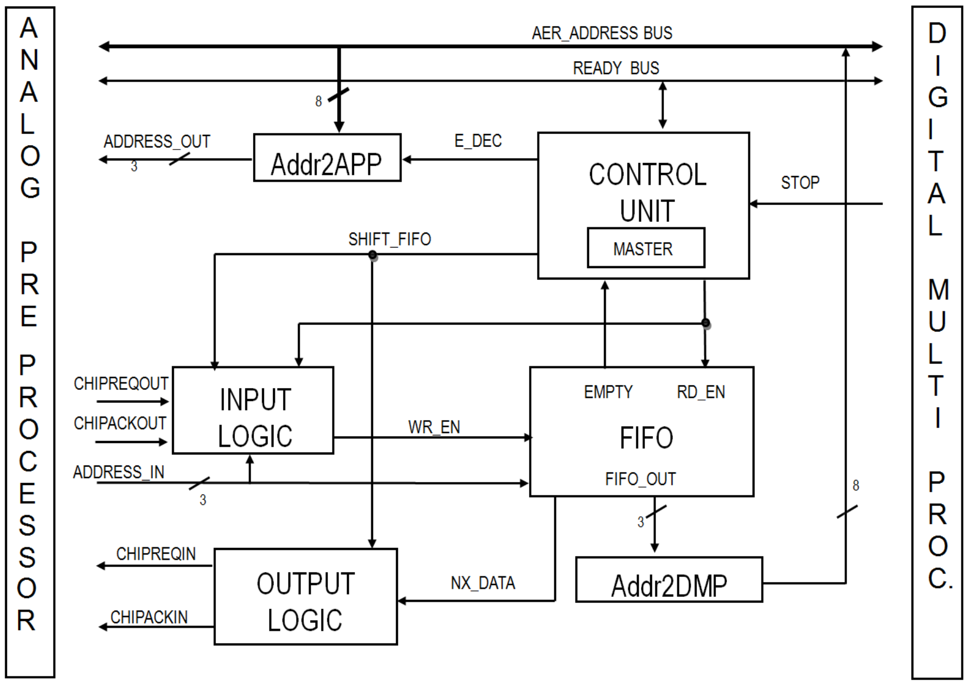


Figure 3.6: Functional blocks of the AER transceiver

In Fig. 3.6, the AER transceiver is shown. It consists of the following elements:

- Control Unit that is responsible for generating internal and external signals to control the DMP and the APP.

- Input logic was designed to detect the asynchronous events generated by the APP coder. A simple method to avoid meta-stability on the bus has been implemented to guarantee fidelity of the information. This method consists on registering the data during at least two clock cycles and validates the same when stable.

- Output logic generates the acknowledge signal to indicate to APP that the current sample has been successfully stored, and the next sample is ready to be received.

- Addr2DMP converts the AER bus width from 3 bits (FIFO) to 8 bits (DMP) to make it compatible.

- Addr2APP converts the AER bus width from 8 bits (DMP) to 3 bits (APP) to make it compatible.

- FIFO buffers the AER input address_in in order to sample each spike generated by APP, due to the DMP is capable of reading spikes only when it is in its distribution phase. For this reason the spikes must be stored during the processing phase.

3.5 Metrics considerations in the design of the data interface

One of the most important parameter to be analyzed in this work is the spike transmission rate. It is the key to allow synchronizing between the APP and the DMP, so that the spikes are properly processed by the whole bio-inspired system. Since spikes generated by analog coders are read by the DMP only during phase 2 (see Section 2.2.1), spikes cannot be accepted by the DMP when it is in phase 1. During the processing time neural parameters and synaptic parameters are calculated. Considering the Iglesias & Villa model [6] the processing time P_T required by the DMP to execute phase 1 is

$$P_T = N_{TP} * T_{CLK} \tag{3.1}$$

where: N_{TP} is the number of clock cycles needed to execute the algorithm, which depends on the number of neurons and number of synapses per neuron (see section 2.3.2), and T_{CLK} is the DMP clock period (20ns in the DMP prototype). N_{TP} is formulated by:

$$N_{TP} = 1909 + 10 \times N + 1392 \times S + 4 \times N \times S \tag{3.2}$$

where: N is the number of neurons and S represents the number of synapses per neuron emulated by the DMP.

Since the main delay is produced by the DMP in phase 1, it is necessary to calculate the minimum time between two successive output spikes, or thus, maximum inter-spike frequency $f_{spike(max)}$ produced by the APP spike generator [3] for a sinusoidal input signal:

$$f_{in(\max)} = \frac{\omega_{in(\max)}}{2 \cdot \pi} = \frac{1}{2 \cdot \pi} \cdot \frac{\delta}{A} \cdot f_{spike(\max)} \tag{3.3}$$

where: $f_{in(max)}$ is the maximum input frequency, the amplitude is represented by A and δ defines the tracking step of the input magnitude for maximum input signal in range:

$$\delta = \frac{2A}{2^{N_B} - 1} \tag{3.4}$$

where: N_B is the desired resolution in bits.

As observed in (3.3), the maximum inter-spike frequency, or thus, the minimum inter spike time is a function of the input signal frequency. Therefore, the maximum input frequency $f_{in(max)}$ is obtained by assuming that the minimum inter-spike period is limited by the data processing delay (phase 1). In order to calculate the $f_{in(max)}$, the following values were taken: N = 30, S = 30, A = 1 V, and 8-bit resolution. The maximum spike frequency is

$$f_{spike(max)} \approx \frac{1}{T_{spike(max)}} \approx \frac{1}{P_T} \approx \frac{1}{930\ \mu s} \approx 1075 Hz \tag{3.5}$$

by replacing $f_{spike(max)}$ in (3.3), the corresponding input signal frequency becomes $f_{in(max)} \approx 1.3$ Hz.

In order to cope with higher input frequency, a bio-inspired spatial encoding is proposed. This mechanism consists in performing an input spike time-to-space translation. i.e., for a given time slot, each one of the spikes that would otherwise be lost, is mapped to a different input neuron of the DMP. Thus many input neurons in a specific time will be devoted to process the output spikes of an individual APP coder as is shown in Fig. 3.7.

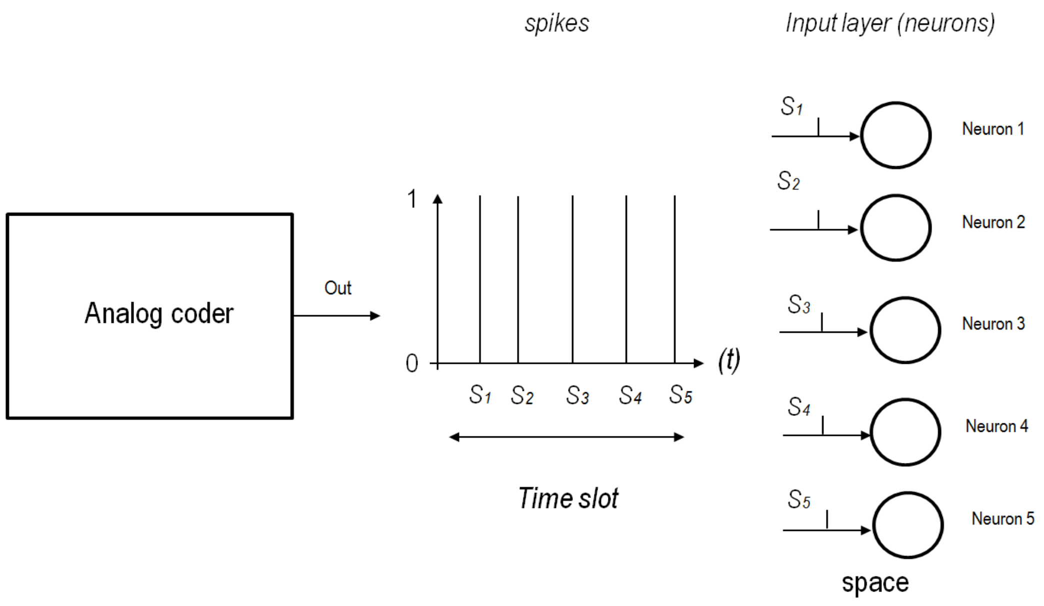


Figure 3.7: Time-to-space translation diagram

Taking into account the new conditions, the maximum spike frequency is determined by:

$$f_{spike(\max)} = \left(\frac{N_i}{P_T} \right) \tag{3.6}$$

where: N_i is the number of neurons in charge of detecting the input spikes of a single analog coder. In the current implementation of the application example, $N_i = 24$, so the maximum input frequency is increased by this number: $f_{in(max)} \approx 32$ Hz. Larger number of neurons and higher clock frequency will allow higher frequency processing in the next implementations.

3.6 Results

This section presents a simple communication sequence between the transceiver and a single DMP. Both are using the AER protocol [5] as shown in Fig. 3.8. The beginning of the transmission process is carried out by the Transceiver (master) by detecting the STOP signal, which is controlled by the DMP. This signal indicates the end of the DMP processing phase and thus the beginning of the spike distribution phase. The process starts with the transmission of the spikes from the analogue circuit to the DMP. This is indicated by the word FE in hexadecimal. After that, the master sends its Chip ID, in this case it is 2. Then, the spike transmission starts. The spikes are encoded by the transceiver to be properly addressed to the desired DMP neuron. As it can be observed from Fig. 3.8, four spikes were encoded by the analogue circuit. Every CAB consists of 4 coders which generates four different addresses. These four spikes correspond to the address of the analog coders of the Configurable Analog Block (CAB). After that, the FB code indicates that all spikes of chip ID 2 were sent to the DMP. The FF code indicates the beginning of the following frame, therefore, the DMP send its Chip ID. In this case it is 1. It is important to note that the network can be configured to connect several DMPs through the AER common bus. As it can be observed from Fig. 3.8, three spikes were generated by the DMP and they are sent to the transceiver, which, in turn, processes and propagates them to the analogue board, where they are decoded. Once all the spikes have been sent, the FB word is sent by the DMP to indicate the end of frame. Finally, the FF word indicates the last frame. When the master sends FD, it indicates START_PROCESSING which means that the spike transmission phase is over and the DMP will resume the data processing phase. After that, the FC code sent by the master indicates that all DMPs operate in the data processing phase.

The following list describes the function of each signal shown in Figure 3.8.

- Stop: this signal is generated by the DMP internal sequencer and it indicates that the DMP is in phase 2 (spike broadcast mode) and thus the AER communication is ready to be executed.

- Syn_address: It is a common bus that transmits the spike address between several DMPs based on the synchronous AER protocol. In this work, the transceiver first sends the spikes generated by the analog encoders to the DMP and later, the DMP transmits its spikes to the transceiver.

- Asy_address_out: The spikes generated by the DMP are received by the transceiver, which sends them by means of these lines to the analog decoders.

- ChipReqin: This signal is generated by the transceiver to indicate that the Asy_address_out is ready to be read by the decoders inside the analogue circuit. According to Fig. 3.8, three spikes generated by the DMP and processed by the transceiver are ready to be decoded by the analogue board.

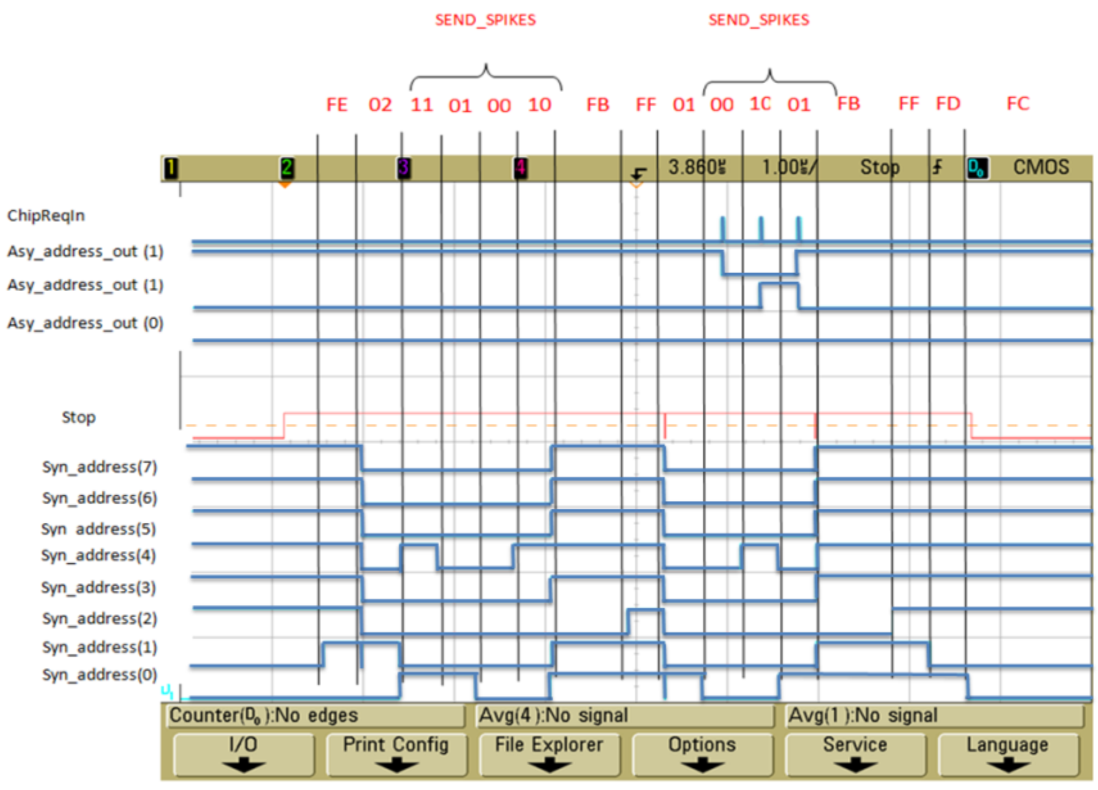


Figure 3.8: Spike transmission process between the Analog chip, the transceiver and the DMP.

3.7 Conclusions

The architecture and proof-of-concept implementation of an analog-digital spiking-neuron-based system capable of processing multiple-input sensor information is presented in this work. For successful communication between analog and digital processors, a transceiver has been developed and the maximum input frequency has been calculated. A spatial encoding has been proposed to increase this frequency limit, with a direct trade-off between frequency and DMP input layer neurons. The maximum frequency of the input signal that can be processed by the current Ubichip was increased from 1.2Hz to ~32Hz by applying the proposed encoding. The challenge in synchronization such as the different modes of AER operation (Asynchronous in APP and Synchronous in DMP) has been discussed. A successful communication through the interface between the APP and DMP has been verified experimentally.

The synergy between UPC and UoE has been very successful with fruitful research collaboration. A simple prototype of Analog-Digital neuromorphic system has been implemented and experimental results have been obtained.

References

[1] Integración Sensorial Neuronal y Autoadaptativa para sistemas empotrados de percepción del entorno (Nessie2) -Neural and Self-adaptive Environment-Perception Embedded Systems - funded by Spanish Ministerio de Ciencia e Innovación (MICINN) - TEC2008-06028, Project proposal, unpublished, 2008.

[2] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas, "The Perplexus bio-inspired reconfigurable circuit," in Adaptive Hardware and Systems, AHS 2007, Second NASA/ESA Conference on, pp. 600-605, 2007.

[3] Gouveia, L.C., T.J. Koickal, and A. Hamilton, "An asynchronous spike event coding scheme for programmable analog array," in Circuits and Systems, 2011, ISCAS 2011, IEEE International Symposium on, pp. 791-799, 2011.

[4] Boahen, K.A., "Point-to-point connectivity between neuromorphic chips using address events," Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, pp. 416-434, 2000.

[5] Moreno, J.M., et al. "Synchronous Digital Implementation of the AER Communication Scheme for Emulating Large-Scale Spiking Neural Networks Models," in Adaptive Hardware and Systems, pp. 189-196, 2009.

[6] Iglesias, J., et al., "Dynamics of pruning in simulated large-scale spiking neural networks," Biosystems, pp. 11-20, 2005.

Application on Ubichip

4.1 Introduction

The upcoming engineering systems are inspired by the neural processing performed in the brain. This processing enables faster perceptual decisions based on the evolution of incoming sensory information from the environment. The term “perception” defines a characteristic of the biological organisms, which interacts with the environment. This term involves cognitive, sensorial and controllable processes, which are related to vision, hearing or other modalities [1]. The continued development in the field of embedded systems (for supporting information processing of the environment) which is inspired by the biological process carried out in the brain through its neurons, has allowed carrying out these types of applications related to perception such as speech recognition [2], vision [3], olfactory pathway [4], textual and image content recognition [5], and robotic control [6] in compact devices either in analogue or in digital domain. These systems involve the implementation of large interconnected neurons, and mechanisms of learning and evolution. One way to support such enormous amount of neurons and these types of mechanisms was proposed in the PERPLEXUS project [7]. In order to demonstrate the capabilities of Ubichip to process the sensory information, a bio-inspired engineering application was proposed in this work. This application was developed in order to detect the frequency of an input signal from low-frequency sensors such as olfactive or auditory. A point to be noted here is that the application mentioned above was carried out using the bio-inspired sensory system (see Chapter 3). As indicated before, the system has been developed in collaboration with the University of Edinburgh. Some modifications were done in the mixed signal bio-inspired system according to our needs. The original system is composed of three elements, which are: the Analogue Pre-Processor (APP), the transceiver and the Digital Multi-Processor (DMP). The analogue coders of the APP [8] were replaced by digital spike coders (for further details see Annexure E), which generate spikes under the same principle of operation as the analogue coders (see Chapter 3), except that the sign is not detected. This digital implementation was done because the analogue coder prototype is located at University of Edinburgh and the majority of our experiments were done at Universitat Politècnica de Catalunya. The system that is used to carry out this application has been developed in digital domain. The details of the digital bio-inspired system are provided in the next section.

4.2 Frequency level detection system

This section presents an engineering application which is based on the detection of frequency of a sinusoidal signal by using a digital bio-inspired system. The digital bio-inspired system, which is proposed in this work, is composed of two modules which are: the digital spike coder module and the Digital Multi-Processor, as shown in Fig. 4.1. In this application, a sinusoidal input signal was created artificially, by recording the values of a sinusoidal waveform in a ROM memory. The sinusoid module can be replaced by an external signal by adding a sensor at the input. The analogue output of the sensor can be converted to digital in order to be processed by the spike coder using an Analogue Digital Converter (ADC) module. Therefore, the Sinusoid module replaces the real input sensory information provided by the ADC module in this application. The digital spike coder module integrates modules like the sinusoid, the spike coder, and the time to space converter. The communication between the digital spike coder module and the Digital Multiprocessor is carried out by the synchronous Address Event Representation (AER) bus.

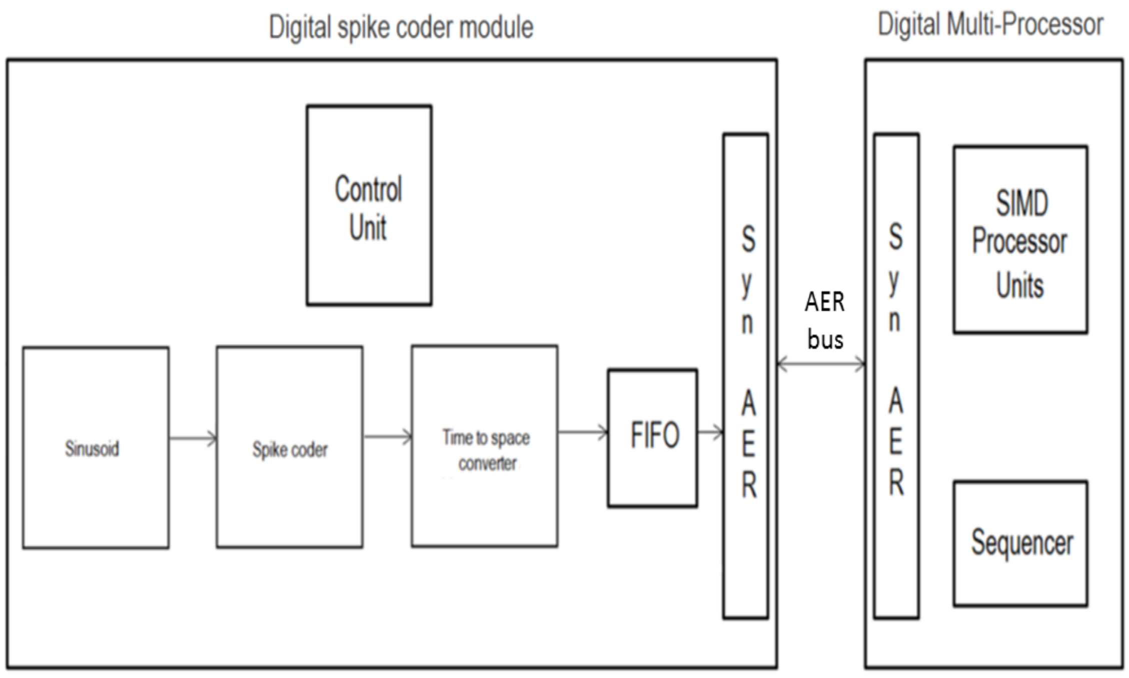


Figure 4.1: Functional block diagram of the digital bio-inspired prototyping and debugging system for the frequency detection application

The functionality of the new bio-inspired system is as follows: the sinusoidal module (see Fig. 4.1) generates the sinusoidal wave (see Fig. 4.2a), which is the sampled version of unit amplitude, unit frequency sine wave and it has been stored as look up tables. The data is fetched and manipulated from this lookup table with respect to the amplitude and frequency and it is the input to the spike coder. The spike coder is responsible of producing spikes using the time step differentiation. The principle of operation of the digital coders is the same as that of the analogue coders. A spike is generated if the present sample value is greater than the previous sample value by a predetermined threshold value. It is important to be noted that the analogue coder generates positive and negative spikes. The negative spikes are converted to positive spikes in this application. This is because the DMP was designed to process only positive spikes. The spikes generated by the spike coder are sent to the time to space converter module which is responsible for assigning every spike (which is produced by the spike coder) to a specific neuron (see Fig. 3.7 in Chapter 3). The First Input First Output (FIFO) stores the address of the neuron which has been assigned with a spike by the time to space converter module. The values stored in the FIFO are sent to the Digital Multiprocessor, only when the Multiprocessor is in the spike distribution phase. The DMP executes the spiking neural network algorithm, which performs a frequency classification. The details of the frequency classifier will be provided in the section 4.2.2.

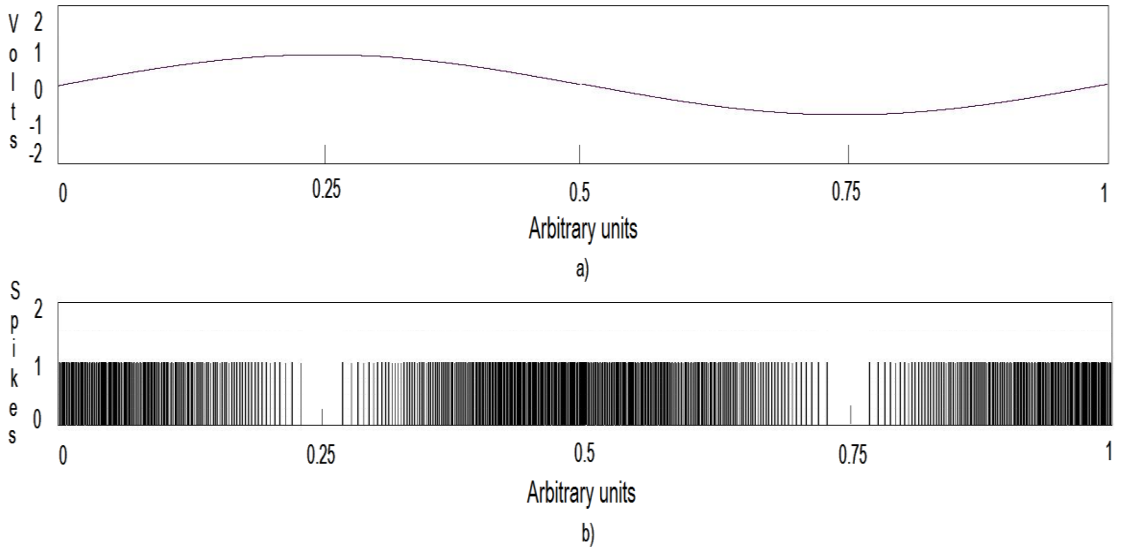


Figure 4.2: a) sinusoidal input signal; b) corresponding spike train calculated by the digital encoder

4.2.1 SNN model

The spiking neural model used in this application is proposed by Iglesias and Villa [9], which was already described in Chapter 2. The assembler code of this algorithm is provided in Annexure C.

4.2.1.1 Parameter values

The spiking neural network is initialized with the values presented in Table 4.1. Some of these values were modified from the original bio-inspired model [9]. Two of them are the threshold voltage and the post-synaptic potential (excitatory or inhibitory) of the synapses. The reason behind the modification of these two values in this application is due to the proposed topology and network size to detect the frequency level. As it can be seen from Fig. 4.3, the proposed topology consists of two layers. The input layer is in charge of processing the spikes, which are generated by the digital coders and the second layer is responsible of indicating the level of the input signal. The details of this SNN topology are provided in the next section. In this application, the neurons of the first layer must fire after receiving the pre-synaptic spikes in order to produce the excitation or inhibition in the second layer (output layer). This way allows detecting the frequency level by sending these spikes to the second layer in few milliseconds. One way to achieve this was to reduce the threshold Θ_i so that the membrane potentials of the input layer cross the threshold with few excitations of its excitatory synapses P_1. The threshold Θ_o of the second layer was modified in order to have a dynamic range of 50mV between the membrane resting potential and the threshold Θ_o. It is worth to mention that the Ubichip architecture support of individual neuron parameters allows for these settings.

Table 4.1: Parameter list of the main variables used for leaky integrate-and-fire neurons

Variable	Original values	Modified values	Hexadecimal representation for the modified values	Short description
P_1	0.84 mV	2 mV	00C8	Excitatory Post synaptic potential
P_2	-1.40 mV	-8 mV	FCE0	Inhibitory Post synaptic potential
V_{rest}	-78 mV	-300 mV	E188	Membrane resting potential
Θ_i	-40 mV	-299 mV	E1BA	Membrane threshold potential of input layer neurons
Θ_o	-40 mV	-250 mV	9E58	Membrane threshold potential of the output layer neurons
$t_{refract}$	3 ms	3 ms	0003	Absolute refractory period
τ_{mem}	15 ms	500 ms	FFAE	Membrane time constant

The membrane resting potential was also changed with respect to the original SNN model. The architecture of Ubichip suffers from some deficiencies. One of the deficiencies is linked to the absence of native saturated operations such as multiplication, addition and subtraction. In order to avoid the overflow in the arithmetic operations, the dynamic range of the membrane voltage was increased. Also, the multiplier affects the calculation of the membrane decay due to reduced precision in the calculation of this variable. Therefore, the membrane time constant was increased in order to ensure that there is a proper decay in the membrane value. It is important to indicate that the problem of saturation and precision can be solved by software, however the number of clock cycles required to implement these mechanisms are: 6420 clock cycles for every saturated multiplication and 950 clock cycles per each saturated addition (see Table 2.3 in Chapter 2). If these mechanisms are implemented it will generate an important delay.

4.2.2 Frequency classifier

This section presents a general network topology for the frequency classifier. The spiking neural network is divided into several blocks; each one represents a frequency level estimator. One block is composed of four input neurons and one output neuron. In the case of reducing the number of input neurons the resolution of the frequency levels by the output layer increase, as it will be demonstrated in the next section. In other words, a range of frequencies can be detected by means of the SNN network by the firing of a particular output neuron. The range of frequencies a particular output neuron indicates depends on the number of neurons that can be supported in the FPGA and the distribution of the input layer per each block. In the initial experiments, it has been decided arbitrarily that the output neuron is connected to the four input neurons through the excitatory synapses and to four input neurons of the upper block through inhibitory synapses, as shown in Fig. 4.3. The reason to select four input neurons is to provide better stability to the output level.

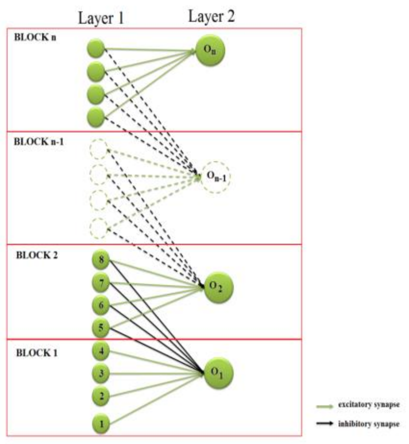


Figure 4.3: Proposed network topology

Frequency level is determined by the excitatory potential of the output neurons. And the firing of several output neurons at once is prevented by the inhibitory connection from the input neurons of a higher block. Consider the case that 5 spikes were generated by the spike coder in such a way that the first 5 neurons receive one spike each. In other words, if there is an input up to the 5th neuron (see Fig. 4.3), then according to the spatial distribution, the spikes would be distributed from N_1 to N_5. In the absence of inhibitory connection of the synapses it leads to the generation spikes in both O_1 and O_2. But when there is an inhibitory connection from the input neuron of the block 2 to the output neuron of block 1, as shown in Fig. 4.3, only O_2 would fire indicating that the amplitude of the input has reached level 2. This is mainly due to the presence of these inhibitory connections which will prevent the output neurons from crossing the threshold voltage immediately. In other words, the potential provided by inhibitory connections will be more negative than the excitatory connections. From Table 4.1 it is evident that the potential provided by the inhibitory connections is 4 times more than the potential provided by the excitatory connections.

4.2.3 Experimental results

This section shows the experimental results in the implementation of the frequency detection of a sinusoidal input signal in Ubichip. The Ubichip prototype implemented in a FPGA can support 36 neurons and 30 synapses per neuron. In our first experiments, the number of neurons and synapses were distributed according to the requirements of this experiment. The maximum number of neurons used in this experiment is 30 and 30 synapses per neuron. In other words a full connected network was created. All calculations carried out in this experiment were done by considering these values. Once achieving the detection of the frequency by the SNN, the numbers of synapses were fixed to 8 and the remaining synapses were disabled by software.

The amplitude of the signal is fixed to 1 V. It is important to notice that the amplitude needs to be fixed to properly detect frequency. The range of frequencies that are used to test the bio-inspired system is selected from 1 Hz to 32 Hz. This is because the maximum frequency of the input signal in the available Ubichip FPGA implementation is 32 Hz. Several aspects were discussed in Chapter 3 in order to increase the frequency of the input signal which can be processed by the Ubichip (see Section 3.5 for further details). The experiments carried out in this work consist of generating artificially a sinusoidal waveform. The values of this sinusoidal waveform are stored in the FPGA LUTs (lookup tables), by means of Direct Digital Synthesis (DDS) while the frequency value of the signal is set by the user through the circuit input, as shown in Fig. 4.4.

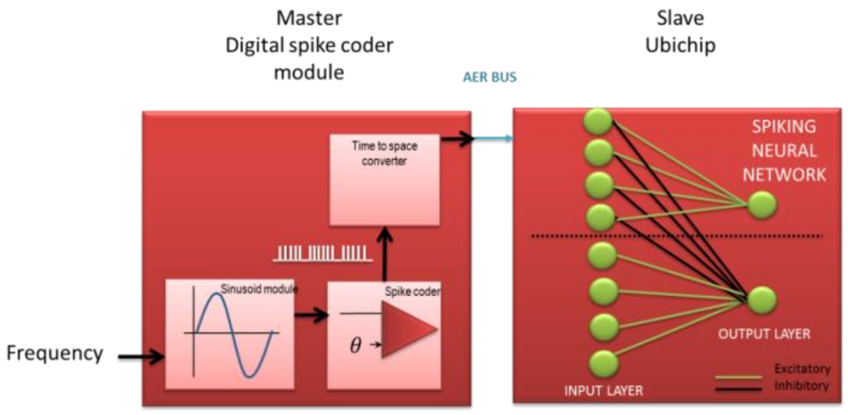


Figure.4.4: Phases of operation of Ubichip and generation of spikes by the digital

The frequency classifier requires 30 neurons to perform the detection of 6 levels of frequency. These 30 neurons are distributed into two feed-forward layers in order implement the frequency classifier according to the mapping proposed in the Fig. 4.3. The 24-neuron input layer maps in space the input spike frequency and the second layer consists of 6 neurons that indicate the frequency level by means of excitation and inhibition from the previous layer. The number of neurons in the input layer is calculated taking the equation 4.3, since the main delay is produced by the DMP in processing phase. This is because the DMP cannot process spikes during this phase, only in the distribution phase, and the generated spikes need to be mapped in space (see Fig. 3.7 in Chapter 3). The expression 4.3, allows calculating the number of input neurons N_i based on the number of spikes which are generated by the spike coder during the processing time of the Ubichip. Let us assume that every spike is produced every $T_{spike(max)}$, in the worst case.

$$N_i = \frac{P_T}{T_{spike(max)}} \tag{4.3}$$

It is necessary to calculate the minimum time between two successive output spikes, or thus, maximum inter-spike frequency $f_{spike(max)}$ produced by the APP spike generator [8] for a sinusoidal input signal, in order to ensure that the allocation of the spikes reaches up to 24 neurons in the input layer

$$f_{spike(max)} = \frac{2 * \pi * f_{in(max)}}{\delta} \tag{4.4}$$

Where, $f_{in(max)}$ is the maximum input frequency, and δ defines the tracking step of the input magnitude for maximum input signal in range, and the calculation of δ is given by expression (4.4):

$$\delta = \frac{2A}{2^{N_B} - 1} \tag{4.5}$$

.

Where: A is the amplitude is represented, and N_B is the desired resolution in bits.

The maximum input frequency $f_{in(max)}$ to be processed by the SNN network classifier is 32Hz according to the maximum frequency $f_{in(max)}$ calculated in Chapter 3. The $f_{spike(max)}$ is calculated taking into account the following values: N = 30, S = 30, A = 1 V, and 8-bit resolution. By replacing these values in equation 4.5 and 4.4, the $f_{spike(max)}$ is 25635.51 Hz or $T_{spike(max)}$ = 39 µs.

$$\delta = \frac{2A}{2^{N_B} - 1} \quad \delta = \frac{2A}{2^{N_B - 1}} = \frac{2(1)}{2^{8} - 1} = 7.84\text{x}10^{-3}$$

$$f_{spike(max)}=\frac{2*\pi*f_{in(max)}}{\delta}=\frac{2*\pi*32Hz}{7.8431x10^{-3}}=25635Hz$$

$$T_{spike(max)}=\frac{1}{f_{spike(max)}}=\frac{1}{25635.51Hz}=39\mu s$$

As indicated before, the digital spike coder works under the same principle of operation of the analogue coders. Thus, the time between the spikes, which are generated by the digital spike coder, for the worst case is 39 µs, and the processing phase in the Ubichip lasts for 930 µs. This is graphically shown in Fig. 4.5. The calculation of the processing phase P_T was done in Chapter 3 (for further information see Section 3.5). The calculation of the maximum number of neurons of the input layer is calculated by equation (4.3). As it can be observed from Fig. 4.5, the time of the distribution phase is shown. This spike distribution time T_D is obtained by considering the worst case where all neurons fire every emulation cycle, in this case $N_D = 30$. This value is calculated by using the expression (4.6)

$$N_i=\frac{P_T}{T_{spike(max)}}=\frac{930\mu s}{39\mu s}=24\ \text{neurons}$$

$$T_D = N_D * 5Mhz \tag{4.6}$$

Where: N_D is the number of neurons that fire every simulation cycle and it is assumed that the AER module works at 5 MHz. The worst case is $T_D = 6$ µs, much lower than the processing phase time.

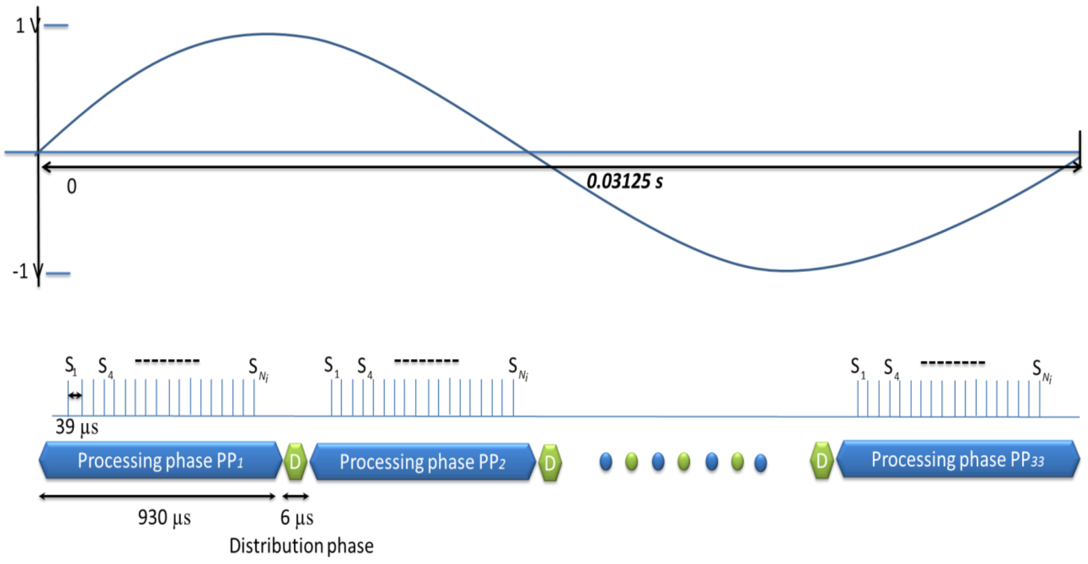


Figure.4.5: Phases of operation of Ubichip and generation of spikes by the digital coder.

The number of neurons of the input layer for different frequencies was calculated following the same procedure as discussed above. Table 4.2 summarizes the calculation of the number of neurons in the input layer for the frequencies in the range of 1 Hz to 32 Hz.

Table 4.2: Number of excited neurons of input layer for several frequencies (1-32 Hz)

Frequency (Hz)	N_i	Frequency (Hz)	N_i	Frequency (Hz)	N_i	Frequency (Hz)	N_i
32	24	24	17	16	11	8	5
31	23	23	16	15	10	7	4
30	22	22	15	14	9	6	4
29	21	21	14	13	8	5	3
28	20	20	14	12	8	4	2
27	19	19	13	11	7	3	2
26	18	18	12	10	6	2	1
25	17	17	11	9	5	1	1

The frequency level is indicated by the corresponding output neuron (O_1 to O_6) which fires as it can be observed in Fig. 4.6. Hence six levels of frequencies can be distinguished because the numbers of block (n) is set to 6 (see Fig. 4.3). Taking the data from Table 4.2, in specific, the number of neurons of layer 1 required for each the selected frequency. The calculation of the level will be indicated by the neurons (O_1 to O_6). Every level is composed of 4 input neurons and one output neuron. Therefore, the N_i obtained in Table 4.2 is divided into 4, the criteria to define the number of output level, based on the result of the division, was to rounded values greater that 0.5 to the top value. The Table 4.3 shows the neurons (O_1 to O_6) which indicate the level of the frequency of the input signal.

Table 4.3: Output neural layer for several frequencies (1-32Hz)

Frequency (Hz)	O_n	Frequency (Hz)	O_n	Frequency (Hz)	O_n	Frequency (Hz)	O_n
32	6	24	4	16	3	8	1
31	6	23	4	15	3	7	1
30	6	22	4	14	3	6	1
29	5	21	4	13	2	5	1
28	5	20	4	12	2	4	1
27	5	19	4	11	2	3	1
26	5	18	3	10	2	2	0
25	5	17	3	9	2	1	0

In this experiment, the frequency of the input signal was modified by the user while the Ubichip was executing its two phases of operation. The main objective of the following experiment is to see how Ubichip can process the signals varying on the time. The test lasted for 800 ms, and the value of the frequency was changed by four times. Table 4.4 shows the values of each selected frequency to test the system.

An experimental spike raster plot can be observed in Fig. 4.6. The input signal was maintained at the same value for four different changes. These times are indicated by the labels T_1, T_2, T_3, and T_4 on Figure

4.6 in order to validate the level of the frequency for each input frequency indicated by their corresponding output neuron (O_1 to O_6). The input frequency is at its maximum (32 Hz) at the beginning of the emulation during the time defined by T_1, which is around 280 ms. The output neuron O_6 level indicates 6 for this first frequency. The first change was done to set the frequency of the input signal at 24 Hz. During the time T_2, which lasts for 100 ms, the frequency classifier indicated that the level of this frequency is 5 by firing the output neuron $O_{5.}$ The frequency of the input decreases to 11 Hz. Hence, the corresponding level of output neuron was 2 since the output neuron O_2 fired during the time T_3, which lasted for 180ms. Finally, the frequency of the input signal was decreased to 5 Hz. The sinusoidal input signal was fed to the amplitude classifier during the time T_4, which lasted for 180 ms. Neuron which fired to indicate this level of the frequency was O_1.

Table 4.4: Frequencies to test the system

Frequency	Value
F_1	32 Hz
F_2	24 Hz
F_3	11 Hz
F_4	5 Hz

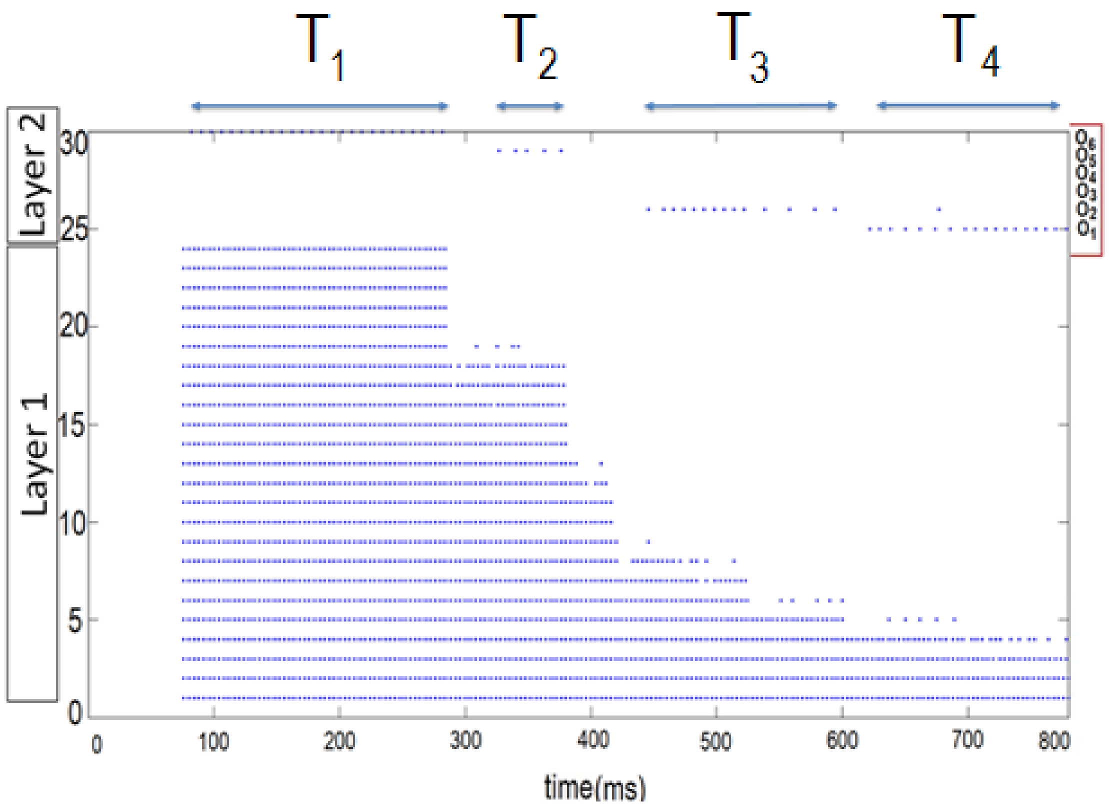


Figure.4.6: Raster plot of neuron spikes [10].

In Fig. 4.7, it can be observed how the membrane voltage of the output neurons was inhibited by their upper blocks. As the frequency was decremented over the time, the membrane voltage of the lower blocks started to increase their membrane potential.

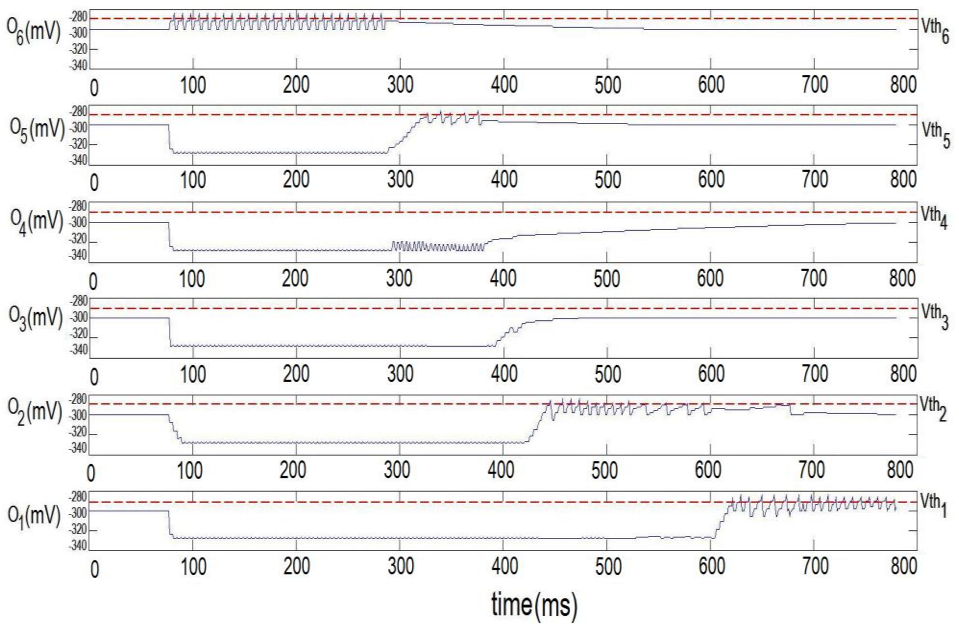


Figure 4.7: Membrane potential of the output layer; the dashed line represents the potential threshold (-290 mV) [10].

Table 4.5: Output levels theoretically estimated and experimentally obtained by applying four different frequencies

Frequency	Value	Theoretical Value $N_{output(1-6)}$	Experimental Value $N_{output(1-6)}$
F_1	32 Hz	6	6
F_2	24 Hz	4	5
F_3	11 Hz	2	2
F_4	5 Hz	1	1

As it can be observed from Table 4.5, there is one value which deviates from the estimated value. This input frequency signal corresponds to the frequency of 24 Hz. As, it can see from the Table 4.3, the value of 24 Hz is in the border between the level 4 or 5. The ambiguity in determining the several frequencies by one output neuron per each block could be reduced by decreasing the number of input neurons per each block. This depends mainly in the number of neurons which can be allocated in a single FPGA and its distribution per each layer. As it can be observed from Table 4.3, a single neuron output can detect up to 6 Hz bandwidth.

4.3 Conclusions

In this work an application based on the sensory information process performed by Ubichip is presented. As a proof-of-concept, the frequency detection application has been experimentally demonstrated.

The maximum input frequency is limited by the processing time of the Ubichip. This was studied in Chapter 3. A spatial encoding has been proposed to increase this frequency limit, with a direct trade-off between the frequency and DMP of input layer neurons. In its present form, the system could be applied to embedded neuromorphic systems using olfactory sensors that have typical operating frequencies under 10Hz [4]. In the ongoing work, the system is being adapted to support audio signal processing.

References

[1] G. Duckworth, "Concepts and Mechanisms of Perception," Psychologycal Medicine, volume 5, pp. 55-60, 1974.

[2] Schaik A., Fragniere E., Vittoz E., “Improved silicon cochlea using compatible lateral bipolar transistors,” MIT Press, pp 671–677, 1996.

[3] Koch C, Li H, “Vision Chips: Implementing Vision Algorithms with Analog VLSI Circuits,” Institute of Electrical & Electronics Engineering, pp 234-237, 1995.

[4] T. J. Koickal, A. Hamilton, S. L. Tan, J. A. Covington, J. W. Gardner, and T. C. Pearce, "Analog VLSI Circuit Implementation of an Adaptive Neuromorphic Olfaction Chip," Circuits and Systems I: Regular Papers, IEEE Transactions on, vol. 54, pp. 60-73, 2007.

[5] Bhuiyan MA, Jalasutram R, Taha TM, “Character recognition with two spiking neural network models on multicore architectures,” In: Computational Intelligence for Multimedia Signal and Vision Processing, 2009 CIMSVP '09 IEEE Symposium on, pp 29-34, 2009.

[6] Sasaki H., Kubota N., “Distributed behavior learning of multiple mobile robots based on spiking neural network and steady-state genetic algorithm,” In: Robotic Intelligence in Informationally Structured Space, 2009 RIISS '09 IEEE Workshop on, pp 73-78, 2009.

[7] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas, “The Perplexus bio-inspired reconfigurable circuit”, in Adaptive Hardware and Systems, AHS 2007, Second NASA/ESA Conference on, pp. 600-605, 2007.

[8] Gouveia, L.C., T.J. Koickal, and A. Hamilton. "An asynchronous spike event coding scheme for programmable analog arrays", in Circuits and Systems, ISCAS 2008. IEEE International Symposium on, pp. 791-799 ,2008.

[9] J. Iglesias, J. Eriksson, F. Grize, M. Tomassini, and A. E. P. Villa, "Dynamics of pruning in simulated large-scale spiking neural networks", Biosystems, vol. 79, pp. 15-27, 2005.

[10] Sanchez, G.; Koickal, T.J.; Sripad, T.A.A.; Gouveia, L.C.; Hamilton, A.; Madrenas, J., "Spike-based analog-digital neuromorphic information processing system for sensor applications," Circuits and Systems (ISCAS), 2013 IEEE International Symposium on, pp. 1624-1627, 19-23 May 2013.

Part II

SNAVA architecture and applications

SNAVA: Spiking Neural-network Architecture for Versatile Applications

5

Etymology

The word ‘snava’ is derived from the Sanskrit root which refers to a“neuron”. This word has equivalent translations, for instance: ‘nervus’ is in Latin, ‘neuron’ in Greek. From these roots the word “Neuron”has been created in English. As per the Sanskrit-English Dictionary by Monier-Williams, Ernst Leumann and Carl Cappeller “snava” refers to a tendon/sinew/muscle/nerve.

5.1 Introduction

The goal of SNAVA is to implement a reconfigurable and scalable digital architecture which can be a platform to emulate Spiking Neural Network (SNN) models that involve bio-inspired mechanisms. This provides an opportunity to explore the neural dynamics involved in the SNN modelling through the experimental investigation.

As mentioned in Chapter 1, the predecessor of SNAVA is Ubichip, which offers interesting features for the emulation of SNN models like SIMD processing, multi-model support and scalability. This architecture was proposed in the PERPLEXUS project [1]. As the starting point in the development of this thesis, Ubichip was studied in order to measure the performance in terms of processing speed and spike distribution time. The result of this study indicates clearly that the Ubichip´s performance is affected drastically due to multiple bottlenecks in its Memory System, Processing System and Communication System. Every Ubichip can support 36 number of neurons and 30 number of synapses at the maximum. This limitation is mainly given by the area occupancy of the FPGA that was used to implement it. The Ubichip architecture was implemented in an FPGA Spartan 3 (XC3S5000). Better results can be obtained with the use of bigger FPGA, because the architecture was designed to be scalable in terms of number of processing elements to be implemented in FPGAs with better area resources. Also, migration is an important feature, which makes it feasible to translate this architecture into bigger FPGAs. Therefore, a new architecture named SNAVA is proposed to guarantee high performance execution and flexibility in order to support large-scale SNN models. The term flexibility is defined in terms of programmability which allows easy implementation of the synapses as well as neuron modelling on SNAVA. This has been achieved by analyzing and customizing the instructions according to the need for processing different SNN models to achieve maximum performance with minimum computation.

This chapter gives a brief explanation of the SNAVA architecture and its modules, the results of detailed evaluation of the performance of SNAVA in terms of speed, area and power, and finally the contribution of our system to the state-of-the art when compared with another approaches.

5.2 SNAVA description

The SNAVA architecture is composed of an array of Single Instruction Multiple Data (SIMD) units. Several aspects regarding the use of the SIMD architecture for simulating SNN models were presented in Chapter 2. Basically, three ideas have contributed to achieve high performance regarding processing and communication speed. These ideas are linked to the processing system, memory system and communication system.

- Processing system/Virtualized topology

The virtualization concept also known as time-multiplexing of neural computations increases the capacity of the architecture in order to support various neurons by using the same core. This technique tries to minimize the consumption of hardware in exchange for the increase in the execution time. The implementation of the virtualization concept is feasible in SNAVA architecture. This is because SNAVA was designed to execute SNN models at high processing speed achieving less than 1 millisecond for every time step simulation. Here the time resolution in the biological neurons is considered to be around 1 millisecond. Therefore, several neurons could be emulated in 1 millisecond by using the same core.

- Memory system

A distributed memory system has been implemented in the current architecture. The memory system allows accessing the memory in each processor by spending a single clock cycle. Putting into practice of such a system was possible since modern FPGAs have thousands of Blocks of RAM integrated in them which could be used for this purpose. Besides, the BRAMs have been manufactured in such a way that they optimize the area and power consumption.

- Communication System

The technological advancements in terms of communication have enabled the development of new protocols of communication at very high speeds. This important aspect has been considered in design of SNAVA in order to be upgradeable with the newest technology without having to make radical changes in the architecture.

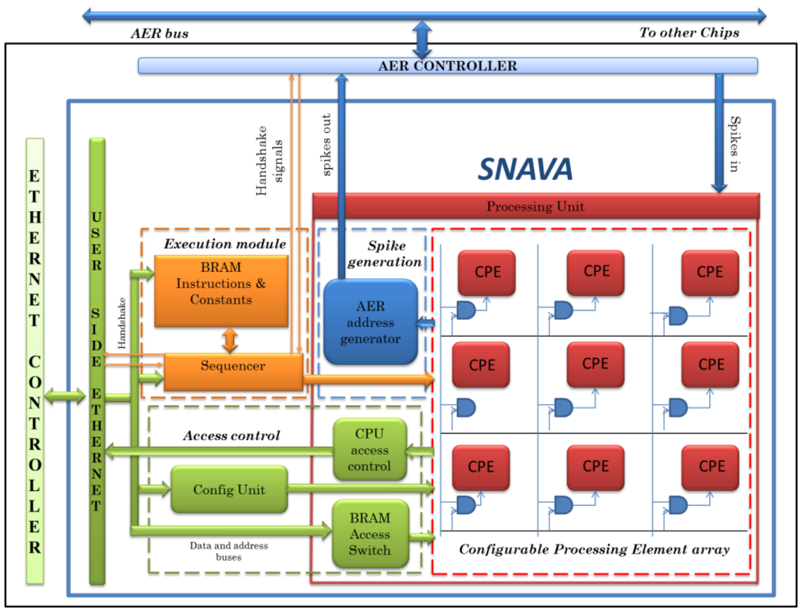


Figure 5.1: Architectural Overview of SNAVA

SNAVA architecture is composed of four modules (Fig. 5.1):

1. The Configurable Processing Element array
2. Execution module
3. Access control module
4. Spike generation

The flow of the input and output data on SNAVA is carried out by two communication protocols (Address Event Representation and Ethernet). These two communication protocols manage the spike communication between the neurons and the communication between the user and SNAVA, respectively. The overall organization of the SNAVA is shown in Figure 5.1. The function of each module is provided in the next sections. And the technical details of the SNAVA architecture are provided in [2].

5.2.1 The Configurable Processing Element array

Each Configurable Processing Element (CPE) is equipped with all the necessary elements to carry out the processing information in a neuron similar to what happens in the biological process. This module consists of a processing element, synaptic BRAM, Content Address Memory (CAM), and spike register, as shown in Fig. 5.2.

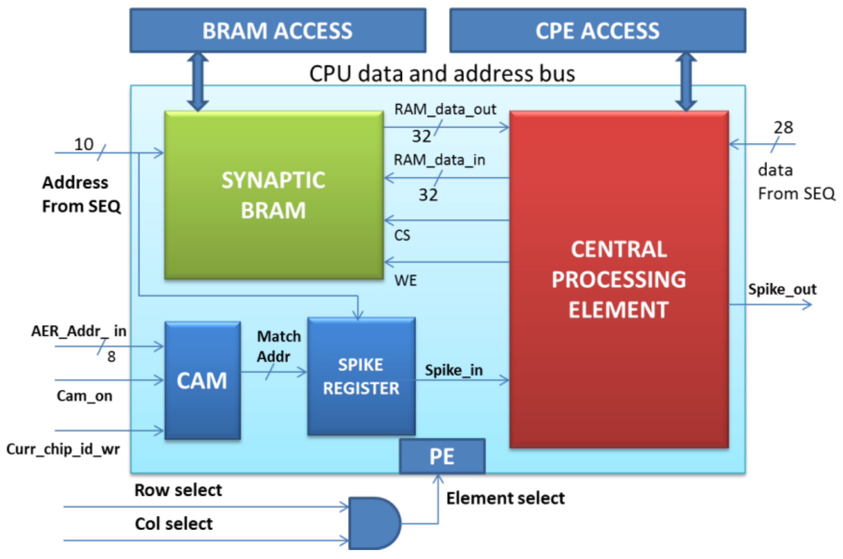


Figure 5.2: Functional Block diagram: Configurable Processor Element (CPE)

Each component of the CPE was designed for specific uses according to the needs to emulate the neuron efficiently. It is explained as follows:

1. Processing Element

Figure 5.3 illustrates the data path of the PE. Each PE contains n register banks where each bank contains eight 16-bit registers. The number of neurons that each PE can emulate is a function of the number of banks of registers. The neural time multiplexing is applied in SNAVA by storing the neural parameters for each neuron using its bank of registers. Hence several neurons can be implemented without a significant increase in the PE area consumption at the cost of time. Now, since each neuron is implemented in a single register bank instead of being implemented in an entire PE, the total area consumed will be reduced to a significant amount.

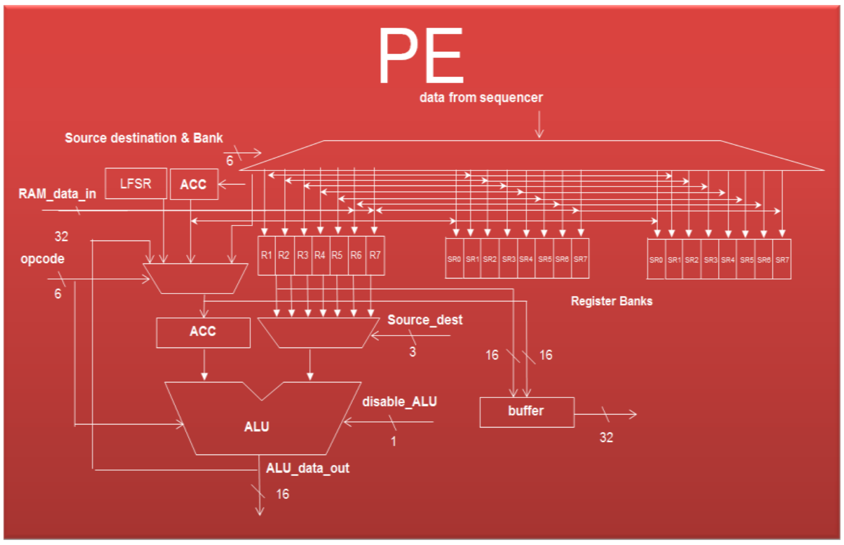


Figure 5.3: PE data paths for simplicity of the drawing only three bank of registers are shown, one active register and two shadow register banks

The key role of the processing element is to perform several mathematical computations involved in an algorithm. These computations involve Arithmetic, Boolean, logical and SNN-customized operations. Several changes were applied to the structure of the PE in Ubichip to optimize the computation of SNN algorithms. The description of every change is listed as follows:

- The structure of the PE in SNAVA was modified with respect to the PE implemented in Ubichip so that each PE can support more number of neurons. One of the goals in the proposal of SNAVA was to implement seven virtual layers containing the array of processing elements. Hence, each processing element in a virtual layer would emulate one neuron. While in Ubichip there is no concept of virtualization being implemented. The mechanism to process every virtual layer, specifically the neural parameters, is explained in Annexure A (see Section A.4).

- The multiplier implemented in Ubichip was based on software multiplier. The execution of this multiplier takes several clock cycles to perform the saturated multiplication operation. This affects the performance of the SNN models as it has been demonstrated in Chapter 2. A 16x16 bit hardware multiplier is implemented in the ALU of the PE. Several protection mechanisms were implemented in order to prevent overflow in the operations. These mechanisms involve saturated multiplication for positive values and negative values. Also, these protection mechanisms were implemented in the addition and subtraction operations. The saturated multiplier, saturated addition and saturated subtraction perform the multiplication, addition, and subtraction operation in two clock cycles respectively.

- The Pseudo random number generator implemented in Ubichip is a 64-bit LFSR in Galois configuration. This LFSR register is shared by all PEs. Therefore, the distribution of every pseudorandom value in each PE is carried out serially by the sequencer. An LFSR register is implemented in each PE of SNAVA, so that the pseudorandom value is loaded in parallel to the entire array of PEs. This feature is relevant since most neural algorithms require the presence of noise for a correct behaviour. The local noise support greatly enhances the multiprocessor performance, clearly compensating the cost of a larger area overhead.

- The number of synapses per neuron is constant in Ubichip but can be variable in SNAVA. This feature in SNAVA permits to create SNN networks using the maximum number of synapses available at each PE. Because not all applications require the same number of synapses per neuron, so that some synapses are not used. By applying the virtualization concept in SNAVA, the total number of synapses can be distributed to every layer such that each layer can have different number of synapses. This distribution of the synapses for each layer is to be done by the user. It is important to be noted that the maximum number of synapses available in the whole array of PEs is a constant. So, the number of synapses per PE is distributed into number of layers according to the requirements of the application.

- The monitor buffer has been implemented in every PE in order to send the calculated parameters to the CPU for visualization, as shown in Fig. 5.3. This was done to make the system transparent. This would reduce the bottleneck in the flow of the data to be sent for visualization. In Ubichip the data are sent to the CPU, only after the all parameters has been processed. In SNAVA, every parameter is sent to the monitor after it has been processed.

2. Synaptic BRAM

This memory block is dedicated to store the synaptic parameters. Every 32-bit memory word is allocated for each synapse in order to store its synaptic parameters. These synaptic parameters are hardwired into the internal registers of the corresponding PEs, as shown in Fig. 5.4, in order to ensure that a single cycle instruction fetches all the parameters for every synapse at a time, by executing the LOADSP instruction. Similarly, the STORESP instruction is made to be used for storing the newly computed parameters back to the memory. The LOAD and STORESP are the customized instructions that are dedicated to store or save the data from the PE to the synaptic BRAM or vice versa.

3. Content Addressable Memory – Spike register

The Content Addressable Memory (CAM) represents the detailed synapse formation and functionality due to interconnection of several neurons to form a certain topology. The role of the CAM is to create matches during phase 2 by reading the addresses that are broadcasted on the AER bus. The encoded spikes are stored in the spike register. The functionality of the CAM implemented in SNAVA is the same as in Ubichip but there were two amendments made in the structure of the CAM. Firstly, every PE was allotted with its own CAM. In other words, the implementation of the CAM has been distributed. Secondly, due to this distributed CAM, multiple spikes are processed at every clock cycle. In case of Ubichip there is only one CAM which is shared by all PEs. Hence only one spike can be processed in every 2 clock cycles.

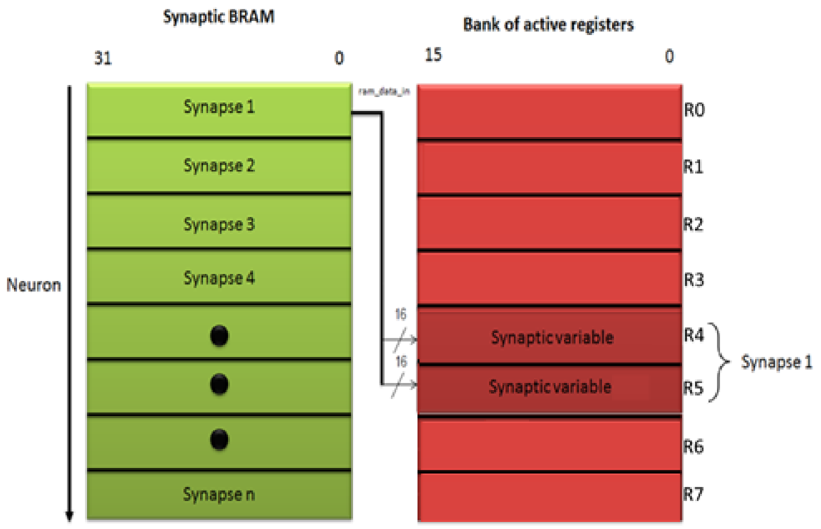


Figure 5.4: Synaptic BRAM wired to active registers

5.2.2 The execution module

The sequencer and the instruction Block RAM constitute the execution unit. The structure of the SNAVA has been defined as Harvard architecture while the structure in Ubichip was defined to operate as Von Neumann architecture. The pipeline strategy was implemented in the sequencer of SNAVA. This technique includes 3 stages, which are: fetch, decode and execute. These two changes have decreased the processing time of SNAVA in the computation of SNN algorithms in comparison with Ubichip. The sequencer is responsible for the entire control flow of the system. The emulation of the SNN models is carried out through two phases (processing phase and distribution phase) as it was defined in Ubichip operation. Firstly, the synapse and neural parameters are calculated and the possible spikes are generated which is marked by the termination of the sequencer during the first phase. Secondly, a signal is being generated by the sequencer along with the AER address generator which indicates the beginning of the distribution phase also known as phase 2. The sequencer begins its operation during phase 1 upon receiving a notification from the AER controller. The instruction set of SNAVA is provided in Annexure A.

5.2.3 Access Control Module

The Access control module controls external access to entities in SNAVA. All the access from the host CPU is only through the User-side Ethernet. Access to any entity is allowed only when SNAVA is not processing neural and synaptic parameters or distributing spikes. The access from the CPU is allowed when the sequencer is in halt state. The access control consists of the following components:

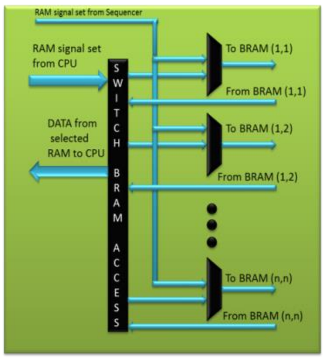


Figure 5.5: Switch BRAM Access

- BRAM Access switch allows accessing the synaptic BRAM from the CPU in order to load the initial values for the synaptic parameters. It also delivers the data from the synaptic BRAM to the CPU when requested. The structure of the switch is shown in Fig. 5.5.

- CPU access control is basically a multiplexer and de-multiplexer that allows the CPU to access to each PE in order to initialize the neural parameters and the LFSR register. Also, the sequencer can have access to PE in order to give the data to be processed. This is done when the operation is being performed by the ALU of the PE, as shown in Fig. 5.6.

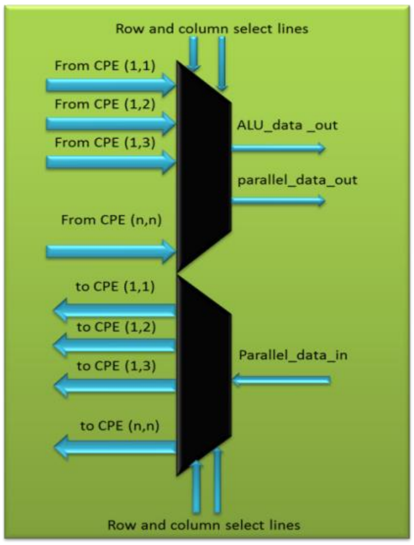


Figure 5.6: mux-demux CPE access

- Config Unit consists of a register bank for global SNAVA control. The register bank consists of seven 16 bits registers. The description of each register is summarized in the Table 5.1.

Table 5.1: Details of Configuration Registers

Register number	Description	Function
0	config_done_int register (0)	1= it indicates that the configuration has been completed, and the SNAVA can start its regular operation
1	clk_mode register(0)	0 = the system clock runs in free-run mode 1= the system clock runs in step-by-step mode.
2	dec_clk_counter register	In step-by-step mode, it stores the number of clock steps to execute, decreasing at each clock cycle. It is a 16 bit register.
4, 5, 6	inc_clk_counter	They form a 48-bits clock counter allowing having a time foot print for recovered states of the chip. It increases at each clock cycle.
7	contr_reset register (0)	This register can be set from the CPU in order to reset every reconfigurable unit. This signal resets the CPE array, and sequencer.

5.2.4 AER address generator

This module is responsible for sending spikes that are produced during the processing phase to the AER module. These spikes are stored in a FIFO, which is included in the AER module. The instruction SPIKEDIS has been implemented in the sequencer to indicate the AER that it can start to perform the distribution of the spikes through network once the sequencer has finished the processing phase. One of the features of this module is to inform the AER control unit the details of the neurons that have fired by sending the address of the neuron to it. The AER address generator reads the LSB of the accumulator from every PE, which contains the spike generated by the neuron. This address is composed by three fields, which are: the row, the col, and the position of the virtual layer where the neuron fired. The length of the address is the 11 bits, which is composed by 4 bits for column, 4 bits for the row and 3 bits for the virtual layer. The spike generator sends to the AER module the address of the fired neuron every time that a virtual layer has been processed. It is important to note that the execution of every layer is carried out serially. Detailed information regarding the process of sending the spikes from the array of CPEs to the AER modules is provided in the Annexure A (see section A.4).

5.2.5 Ethernet user side

Ethernet user side provides the control to the user to send Neural and Synaptic information to the Host CPU for monitoring the network at any point in the algorithm. There are two modes in which the Ethernet user side operates which are online scan and offline scan:

1. Online scan: In this mode the sequencer is not halted and the Ethernet User side scans the whole array for data in the monitor buffers and sends it to the Ethernet core which in turn sends the data to the external CPU. The instruction STOREB is one of the instructions of the sequencer which was implemented in order to indicate to the Ethernet user side that this module can read the values of each PE. These values are allocated in the buffer of each PE. The buffer contains the neural and synaptic parameters which have been processed by the PE when SNAVA is in the processing phase. The advantage of using this mode is that visualization of the parameters is possible and the parameters can be sent to the monitor once they have been processed. The sequencer does not interrupt its operation except if the Ethernet transmitter is busy.

2. Offline Scan: In this mode the processor is halted and the Ethernet User side scans the whole array for data in the monitor buffers and sends it to the Ethernet core which in turn sends the data to the external CPU. The offline was mainly developed to carry out the process of debugging of SNAVA. In this mode the user can access to the components to SNAVA to verify the value in the registers of a component. These components are: the

sequencer, the processing element, the synaptic BRAM, and the BRAM instructions. The SNAVA memory map to access to these components from the CPU is provided in Annexure B.

5.2.6 Phases of operation

The process for the simulation of any SNN model is carried out in SNAVA by means of two periodic phases, which were introduced in Section 5.2.2. The events happening in SNAVA is similar to that in a biological neuron. Figure 5.7 shows how the biological process in the neuron is being mimicked by the SNAVA architecture. In case of the biological process, the neuron receives chemical and electrical signals via the synapses which are the interconnections made between neurons. These signals travel through the dendrites to be processed by the soma. The soma fires a spike when its membrane potential reaches the threshold voltage. The spike flows through the axon and is propagated to the dendrites of other neurons. SNAVA tries to mimic the biological process described above through its processing phases, the function of dendrites are carried out by the synaptic register (see Fig. 5.2) which store the spikes from the previous distribution phase. Spikes are processed by the PE only in the processing phase according to the SNN algorithm. The soma is the important region of the neuron which consists of the Processing Element, the synaptic parameter BRAM, Instruction BRAM and the Sequencer. Since the concept of virtualization has been implemented in SNAVA, when a layer of neurons complete the processing phase, a corresponding series of spikes are generated by the AER address Generator module. When the execution of the algorithm is completed i.e. when all layers of neurons have been processed, it marks the end of the processing phase. Once the processing phase is completed, the spike distribution begins. This phase is indicated by the flow of spikes to the dendrites of the destination neurons. The transmission of the spikes is emulated by means of broadcasting these spikes in the AER bus and then creating the synaptic contact in the CAM that is located in the PE of each neuron. The spikes that enter the neuron are stored in the Synaptic register (Dendrites) that corresponds to that particular neuron.

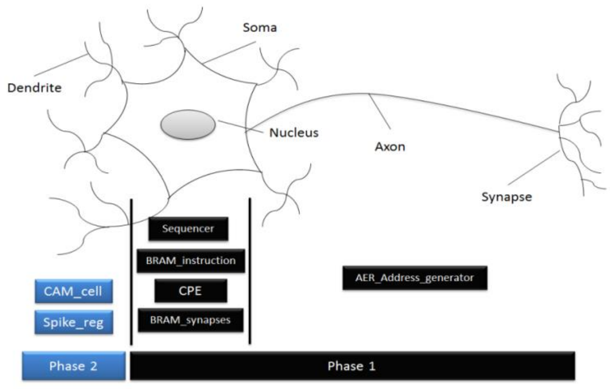


Figure 5.7: Biology and SNAVA based on [2]

5.3 Implementation and performance

The SNAVA prototype is implemented on the KC705 board kit which includes a Xilinx Kintex7 FPGA embedded on it. This board offers advanced modules of hardware which involve high speed serial links and advanced memory interfaces. Therefore, the use of these has facilitated the development of the present architecture with high performance in terms of communication and processing. In regards to communication, two protocols have been implemented on SNAVA in order to manage the flow of the data, as shown in Fig. 5.8.

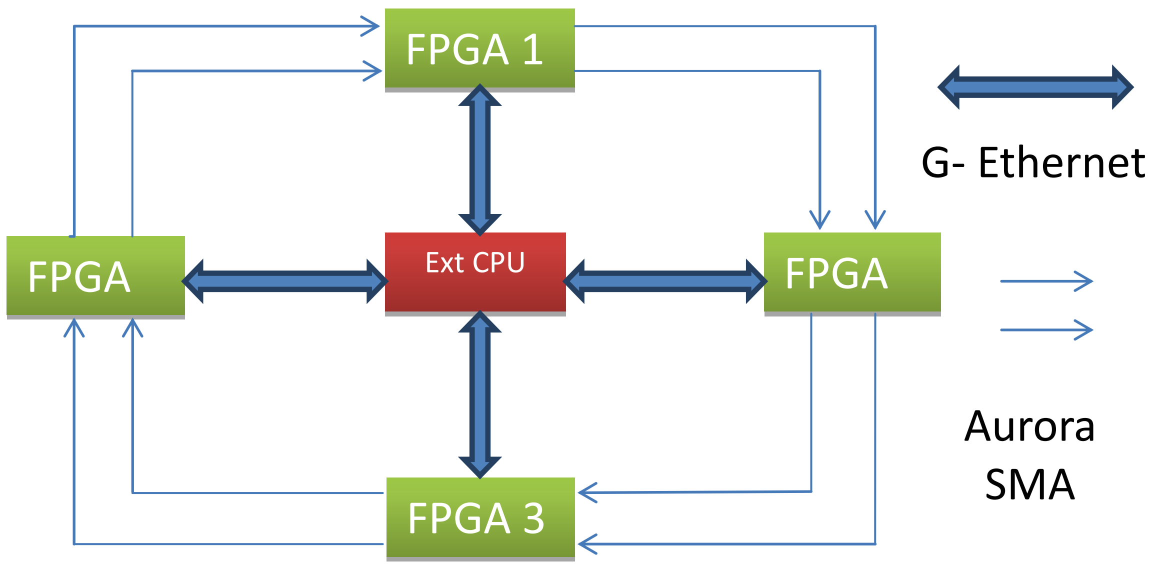


Figure 5.8: SNAVA Communication Network. Figure extracted from [2]

1. Aurora is a communication protocol which is used to transmit data point to point through fast serial links. This protocol offers several benefits like high bandwidth transmission, support Full Duplex & Simplex channels and minimum area consumption. Therefore, it has been decided to use this protocol, to carrying out the communication between neurons due to its interesting features, in order to achieve the maximum performance in communication on SNAVA by maintaining the low latency and low power consumption. It is important to note that this work has been developed by Mr.Taho Dorta Pérez collaborating with the SNAVA project [3].

2. Ethernet MAC protocol is one the most popular networking protocol adopted in FPGAs due to its features like: flexibility, performance and reliability. These features has been taken into account in development of SNAVA in order to create an efficient interface that allows to the user take the control of the flow of data between SNAVA and the external CPU.

The scalability is one of the main features offers by SNAVA which allow the emulation of large scale SNN by connecting multiple FPGAs boards. The proposed network topology is based in a ring configuration which enables the efficient use of the resources available, namely the construction of the network does not require extra hardware to connect a large number of FPGA boards. This reduces the effort for making expensive dedicated interfaces to connect large number of boards like [7]. Besides, the pipeline operation carry in every board increases the performance of the communication through the network. But SNAVA is not only scalable respect to the number of boards is also scalable in the number of processing elements. Therefore, the user can define the number of processing elements that are required for a specific application enabling the creation of an optimized architecture.

The following sections present the study of the area and power consumption of SNAVA architecture by testing 2x2, 4x4, 6x6, 8x8, and 10x10 configurable processing array sizes where all processing elements have enabled 2 level of virtualization. The experiments were done by considering two cases, the first consist of a network which one layer is full connected, and the second case considers that a virtual layer has a single synapse in order to clarify the logic utilization for the implementation of the two modules: CAMs and the processing elements. The results presented in these evaluations were taken from report utilization and report power, which are provided by the utilization of VIVADO software tool. This software is released by Xilinx Company in order to implement the designs efficiently on the FPGAs.

5.3.1 Area consumption

It can be seen clearly that the biggest consumption hardware resources is due to the use of LUTs, which increases when a large number of processors are implementing on SNAVA, when the Table 5.2 and 5.3 compares. In the case of 100 processors and the communication controllers the area consumption is around 80% of the total for both cases. Then the number of synapses has a small impact on the consumption by considering an increment around 4%. Therefore, it is important to analyze the area consumption of each component in order to optimize the current implementation in order to achieve better performance and a large number of synapses per neuron.

The following figures points out the area consumption for each module in the design hierarchically, namely from the top level until the processing elements. As it can be observed from Fig. 5.9, the consumption of the Ethernet controller and the AER controller are negligible when is compared with the area consumption of the SNAVA. Fig. 5.10 shows the SNAVA consumption, this architecture is composed by the sequencer, the array of Processing Elements, and the configuration module. It is clear that main area consumption is due the implementation of configurable processing units which consumes around the 93% of the total. Figure 5.11 shows clearly that the main area consumption is due to the implementation of the Processing Element. The major consumption of LUTS is attributed to the implementation of the multiplexors inside of the PE to carry out the customized instructions.

Table 5.2: Utilization Summary of Fully Connected SNAVA

Resource	2x2 (3 synapses)	4x4 (15 synapses)	6x6 (35 synapses)	8x8 (63 synapses)	10x10 (99 synapses)	Available
Flip-Flops	6623 – 2%	15282 – 6%	31532 – 8%	58257 – 14%	99487 – 24%	407600
LUTs	9805 – 5 %	29595 – 15%	67727 – 33 %	109801 – 54%	171291 – 84%	203800
BRAMs	39 – 4%	51 – 6%	71 – 8%	99 – 11%	135 – 15%	890

Table 5.3: Utilization Summary of Single Synapses SNAVA

Resource	2x2 (1 synapses)	4x4 (1 synapses)	6x6 (1 synapses)	8x8 (1 synapses)	10x10 (1 synapses)	Available
Flip-Flops	6606 – 2%	14609 – 4%	27861 – 7%	46351 – 11%	69996 – 17%	407600
LUTs	10421 – 5 %	29317 – 14%	60212 – 30 %	104533 – 51%	161158 – 79%	203800
BRAMs	39 – 4%	51 – 6%	71 – 8%	99 – 11%	135 – 15%	890

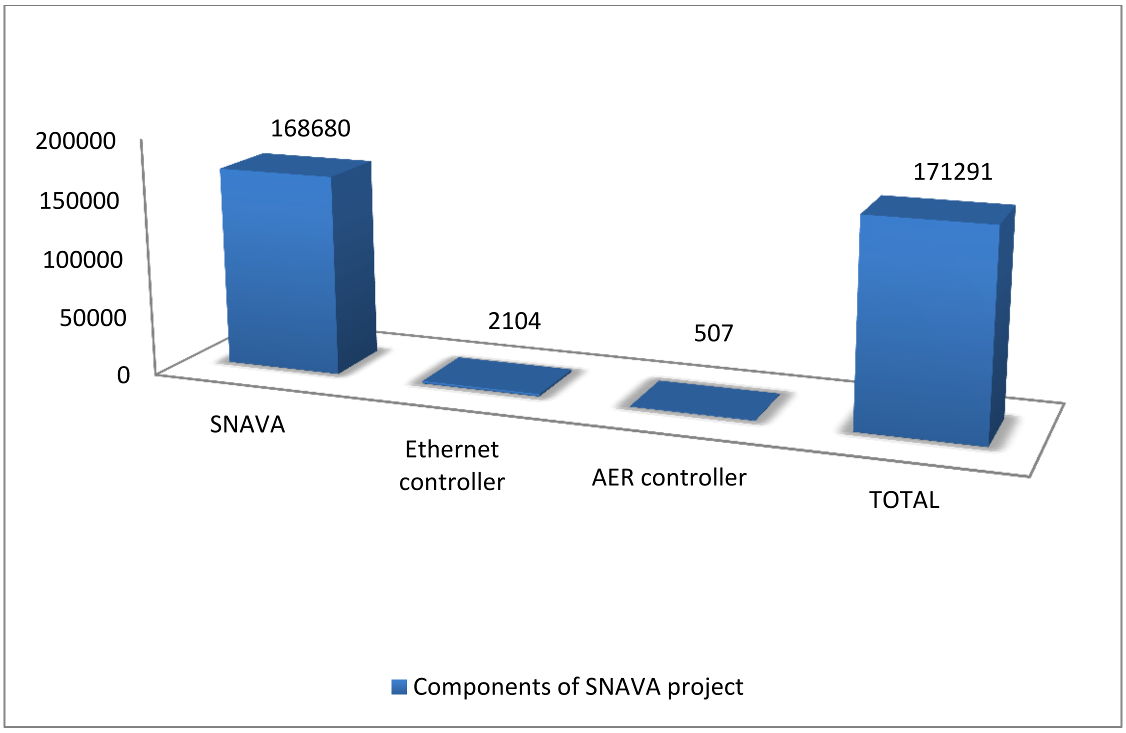


Figure 5.9: Utilization Representation of fully connected SNAVA project 10x10

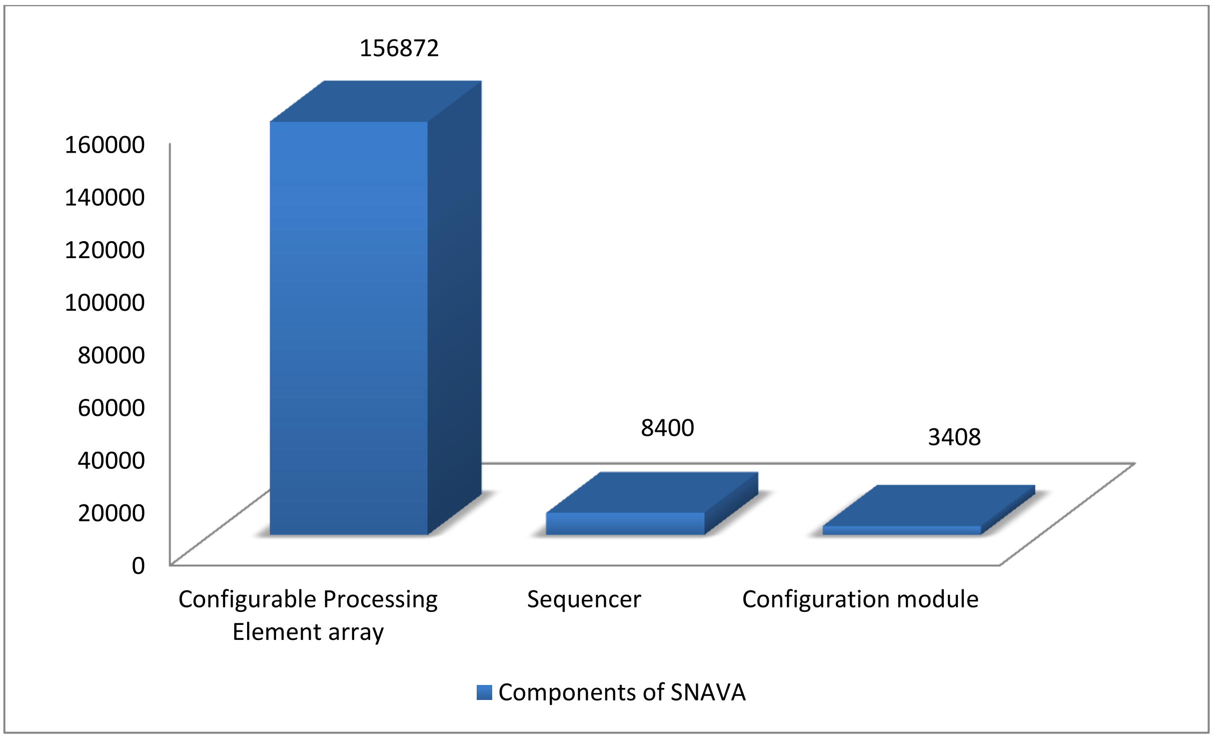


Figure 5.10: Utilization Representation of fully connected 10x10 SNAVA

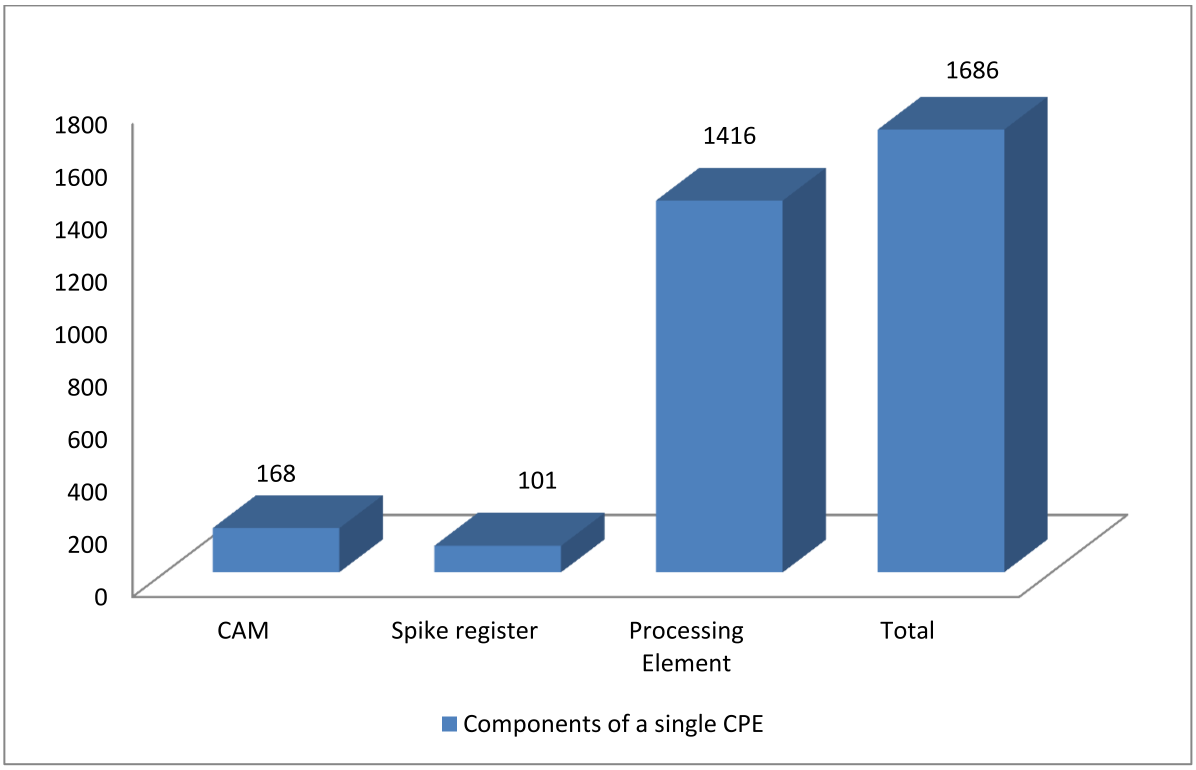


Figure 5.11: Utilization Representation of a Single Processor Element in fully connected SNAVA 10x10

5.3.2 Power consumption

The total power consumption of SNAVA project has been estimated around 0.891 Watts, which can be calculated by adding the static power and dynamic power. It is important to be noted that the confidence level of this estimation is low. This is because the software calculates the power consumption by testing the architecture with test vectors. This vector of test enables all the nodes of the architecture, so that the architecture is evaluated by considering the worst case. Namely, the architecture is working with all the components all time.

As it can be observed from Fig. 5.12, the static power is around 0.169 W while the dynamic power is around 0.931 W. Regarding the static power would be constant for all the array sizes and dynamic power, which varies based on the design. Therefore, the dynamic power of a fully connected SNAVA is analysed in order to make clear the power consumption for each its modules. The results of power consumption reported in this section were obtained by enabling the settings on VIVADO to optimize the power on the design. These options are:

1. power opt design
2. post placed power opt design

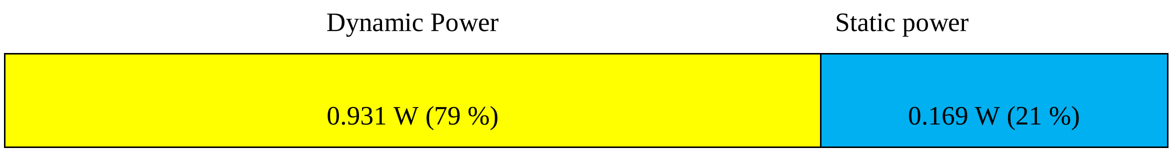
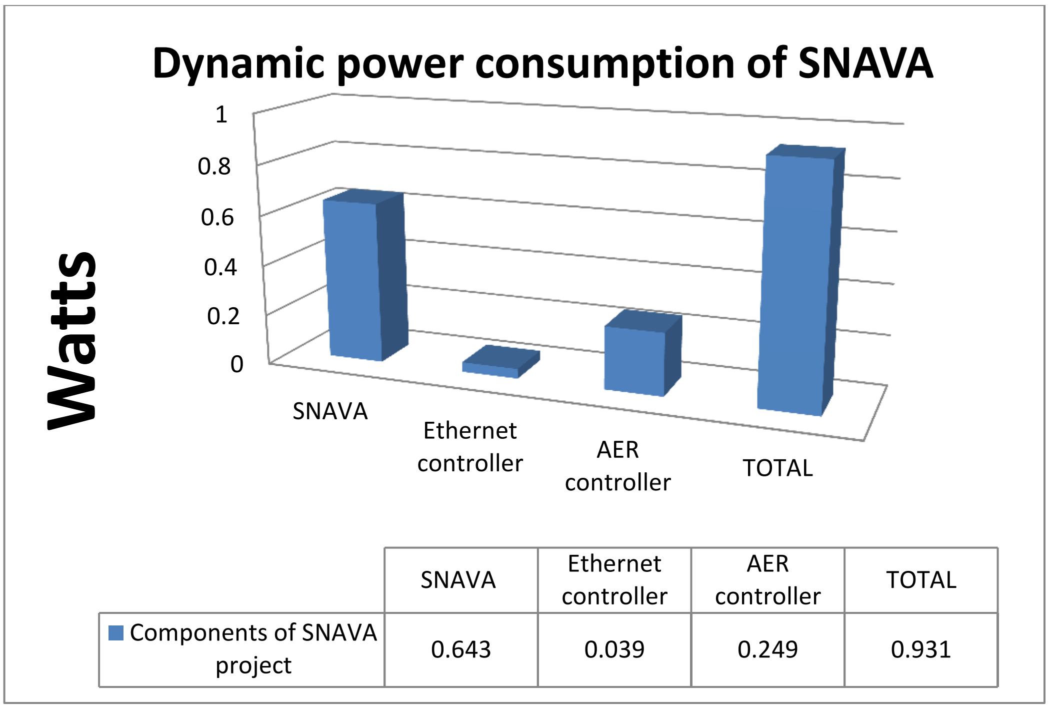


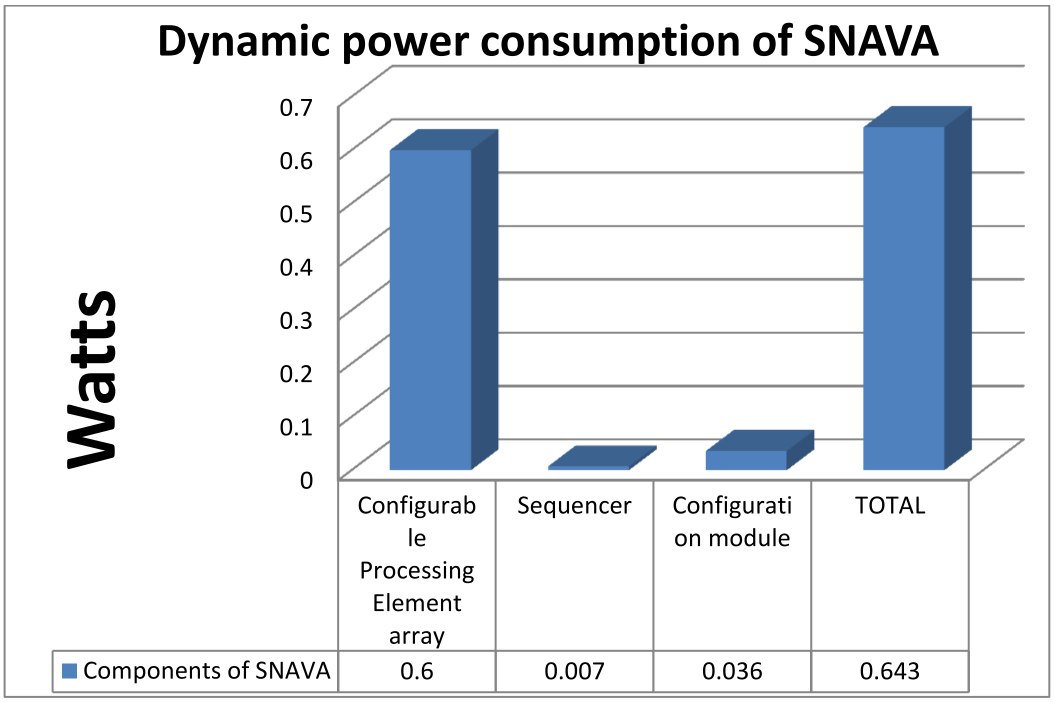
Figure 5.12: Power consumption of fully connected SNAVA 10 x 10

Figure 5.13 shows the power consumption for the SNAVA implementation which is composed by SNAVA, and AER controller and Ethernet controller. As can be seen from this table, the power consumption of Ethernet controller is negligible when compared to SNAVA and AER controller. Almost the half of the power consumption is spend by SNAVA architecture while the AER controller takes around of one quarter of the total as shown in the figure. As it has been expected, the large power consumption is spent by the processing elements as shown in Fig. 5.14 and Fig. 5.15. The processing element contributes to the major consumption of area and power of the Configurable Processing element module, as shown in Fig. 5.11 and 5.15, respectively. Only two 2 layers/2 bank of registers on the current SNAVA architecture have been implemented, so that the implementation of more bank of registers will increase the area and power consumption in order to support large-scale SNN models. Therefore, some modifications have been proposed in this work to increase the capability of SNAVA to support large-scale spiking neurons by decreasing the area consumption at the cost of processing speed. These new ideas have led to develop the new version with better performance will be presented in Section 5.6.



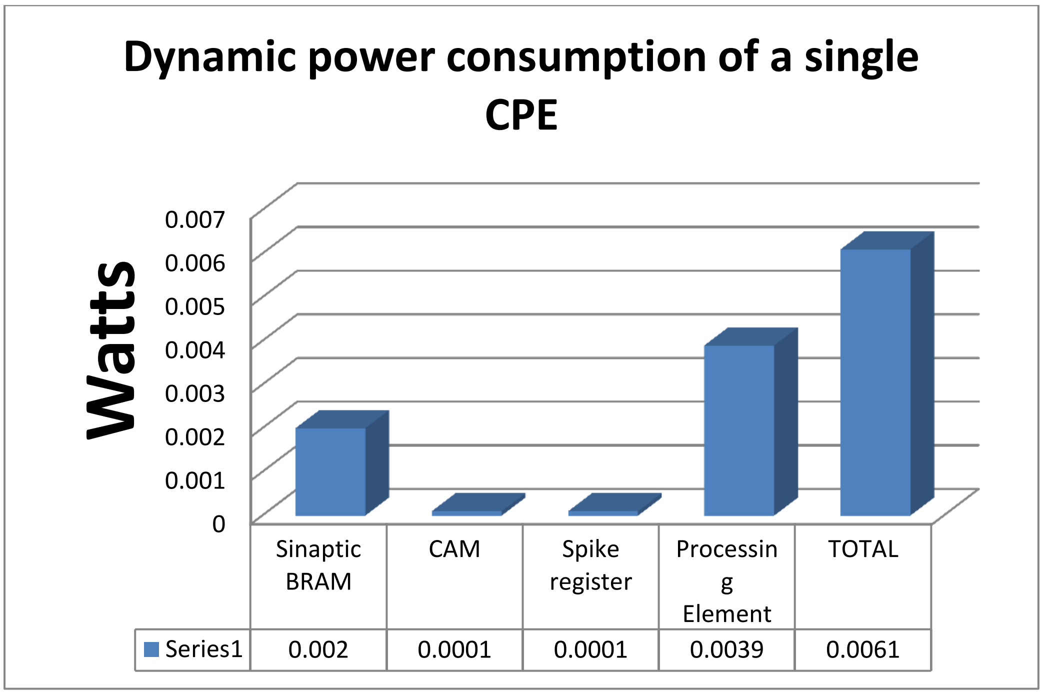
	SNAVA	Ethernet controller	AER controller	TOTAL
🟦 Components of SNAVA project	0.643	0.039	0.249	0.931

Figure 5.13: Dynamic Power distribution of fully connected SNAVA 10 x 10 – SNAVA project



	Configurable Processing Element array	Sequencer	Configuration module	TOTAL
🟦 Components of SNAVA	0.6	0.007	0.036	0.643

Figure 5.14: Dynamic Power distribution of fully connected SNAVA 10 x 10 – SNAVA



	Sinaptic BRAM	CAM	Spike register	Processin g Element	TOTAL
■ Series1	0.002	0.0001	0.0001	0.0039	0.0061

Figure 5.15: Dynamic Power distribution of fully connected SNAVA 10 x 10 – Single CPE

5.4 Performance evaluation

SNAVA exhibits many advances with respect to Ubichip in terms of processing speed and spike distribution. Many of these are due to the improvements performed on it. SNAVA tries to exploit the benefits of the parallel SIMD architecture, so a majority of its instructions are carried out in a single clock cycle in all Processing Elements. The important changes made on SNAVA with respect to Ubichip architecture are listed below:

1. Two strategies related to the structure of memory system have been implemented in SNAVA in order to exploit the available Block RAMs on the FPGA. The first strategy consisted on separating BRAM for storing the instructions and global constants was implemented as these are common for all the PEs. And second, One BRAM per each Processing Element for storing the synapse parameters was implemented and the output of the BRAM was hardwired to the internal registers of the PEs, as shown in Fig. 5.4. Then a single cycle instruction is used to fetch all the parameters for individual synapse at a time. In the same way an instruction can be made use of to store the computed new parameters back to the memory. These strategies allow the best use of the available on-FPGA resources.

2. The neural parameters are specific to each PE. Hence these parameters can be stored locally in the register bank instead of storing them in a common memory. This will save the time and complexity in fetching them each time. Instructions capable of swapping bulk data between the active and shadow registers have been implemented.

3. The CAM was modified to detect multiple matches in a single clock cycle. This change has accelerated the spike distribution. Therefore, the encoding of the spikes is in parallel, which represents the best improvement with respect to Ubichip. The Ubichip can detect only one spike by using two clock cycles. Also, a spike register array corresponding to the neurons, that stores the matches in the specific synapse numbers, was implemented.

4. There are several hardware multipliers available in the FPGA. These multipliers have been used in SNAVA to accelerate this operation used in most SNN algorithms.

All these changes have helped to boost both processing speed and spike distribution in order to achieve real-time simulation of large number of neurons below 1ms resolution. The following evaluations were carried out in order to obtain the figures of performance of the SNAVA, in terms of processing speed and spike distribution time, by considering the emulation of Iglesias and Villa model [4], Izhikevich model [5] and Leaky integrate and fire model in 16-bit fixed point arithmetic operations. The following evaluation in the performance of SNAVA considers this amount of neurons and synapses. The algorithms for the emulation of the mentioned models have been programmed in assembler code in order to achieve the maximum efficiency in terms of the execution time. Also, the program was defined in a structured manner in order to simplify the process of update. Taking into account the results of the previous analysis regarding the area consumption, the maximum number of neurons and synapses per processor that can be supported in the current FPGA are: 200 and 100 respectively (with 100 processors and 2 levels of virtualization).

5.4.1 Iglesias and Villa model execution analysis

The description of the Iglesias and Villa model is provided in Chapter 2, and the algorithm, which has been written in assembler code for SNAVA, is provided in Annexure C. The performance of SNAVA has been evaluated in clock cycles in order to obtain real figures. The required number of cycles to execute each subroutine in phase 1 is indicated in Table 5.4. The encoding of subroutines contained in the synapse loop is shown in Table 5.5.

Table 5.4: Main loop subroutine encoding and execution number of clock cycles

Symbol	Subroutine	Clock cycles *
M V	Membrane value	37
C S	Cycle per each synapse	(122)·S
M O L P	Memory of last post-synaptic	28
S U	Spike Update	24
B A	Background activity	34
R P	Refractory period	5
N S	Neuron save	19
S E	Spike enable	5

* The number of clock cycles for the subroutine CS depends on the number of synapses (S).

Table 5.5: Synapse loop routine encoding

Symbol	Subroutine	Clock cycles
S L	Synapse Load	1
S W	Synaptic weight	19
R V V	Real value variable	30
A V	Activation variable	43
M O L P	Memory of last pre-synaptic spike	24
S S	Synapse Save	5

Adding all the contributions of Table 5.4, except the initialization process, the number of clock cycles N_T that is required to carry out the phase 1 in one simulation cycle is obtained in equation (5.1), where the growth depends on virtual layers and number of synapses per each processor.

$$N_T = 152 \cdot N_v + 122 \cdot S \tag{5.1}$$

Where: N_T is the number of clock cycles, N_v is the number of virtual layers and, S is the number of synapses per processor

Monitoring the synaptic parameters and neural parameters on real time is essential to observe the SNN dynamics. Thus, SNAVA allows the user to define the number of synaptic parameters or neural parameters to be displayed on the monitor. Neural and synaptic parameters in phase 1 could be shown on the monitor once it has been processed, this implies that the sequencer stops its operation and the Ethernet user side module reads the data stored in every buffer serially and these data are sent to the computer through its Ethernet bus. Sequencer resumes its operation only when the user side module finishes reading all the buffers. Equation 5.2 takes into account the number of clock cycles required by the user side module to read all buffers. The size of Ethernet bus and the size of the buffer in the current implementation are 8 bits and 32 bits respectively.

$$N_{TD} = 152 \cdot N_v + 122 \cdot S + P\left(\frac{BS}{B}\right)\left(S \cdot SD + N_v \cdot ND\right) \tag{5.2}$$

Where: N_{TD} is the number of clock cycles, N_v is the number of virtual layers, S is the number of synapses per processor, P is the number of Processing Elements (PEs), B is the Ethernet bus width, SD is the number of synapse parameters to display, ND is the number of neuronal parameters to display and BS is the Buffer size.

The Iglesias and Villa model was proposed during the development of PERPLEXUS project. The proposed target in case of this project includes the implementation of 300-synapses and 100-neurons in a single Ubichip, and 1 ms time step resolution by simulating this SNN model. The time required to execute these many number of synapses and neurons is 13 ms approximately for every simulation cycle (see Table 2.3 in Chapter 2). Therefore, the simulation of the Iglesias and Villa in real time is not achieved (1 ms step time resolution).

The required execution time for a single simulation step, by considering 100 neurons with 100 synapses per neuron, is about 3.64 ms. In this case the number of synapses has reduced (from 300 to 100) the step resolution is not achieved. The execution time E_{TU} was calculated by using the expression (5.4). The expression (5.3) allows the calculation the number of clock cycles N_{TU} required to simulate the Iglesias and Villa model in a single step simulation in Ubichip.

$$N_{TU} = 1909 + 10 \times N + 1392 \times S + 4 \times N \times S \tag{5.3}$$

Where: the N is the number of neurons and S is the number of synapses. The time execution E_{TU} is obtained as follows:

$$E_{TU} = N_{TU} * T_{clk} \tag{5.4}$$

Where: the N_{TU} is the number of clock cycles to simulate the algorithm in a single step simulation and T_{clk} is the time of the clock of the system. In the case of Ubichip, the clock system works at 50 MHz.

In SNAVA, the maximum number of neurons, which can be implemented in a single FPGA, is 100 neurons (1 layer of virtualization) and the maximum number of synapses is 100 synapses per neuron (100 synapses per processor). The required execution time E_{TS} for a single step simulation is 102.86 µs. The Execution time E_{TS} is calculated by the expression 5.5.

$$E_{TS} = N_{TD} * T_{clk} \tag{5.5}$$

Where N_{TD} is the number of clock cycles, and T_{clk} is the time clock of the system. The calculation of the execution time E_{TS} of current prototype is particularized at 125 MHz clock or 8 ns period. This is because the modules of SNAVA, which involve the Configurable Processing Elements, the Ethernet module and the AER system, work at 125 MHz. It is important to be note that the calculation of the processing time in SNAVA considers the display time assuming the post-synaptic spike and the membrane voltage are send to the monitor for its visualization. To calculate the number of clock cycles N_{TD} (5.2) the following values were taken: P = 100 PEs, BS = 32, B = 8, N_v = 1 and ND = 1.

The step time simulation required in Ubichip to perform the Iglesias and Villa model is 35 times lower than SNAVA. An important improvement in the processing time was achieved by implementing the distributed memory system, the multipliers, and increasing the frequency clock of the system in SNAVA. These improvements were already mentioned above. In order to clarify the real improvement achieved in SNAVA when compared with Ubichip by neglecting the increment in the value of the clock system, the following comparison was carried out by calculating the improvement ratio in term of clock cycles instead of doing with execution time cycle.

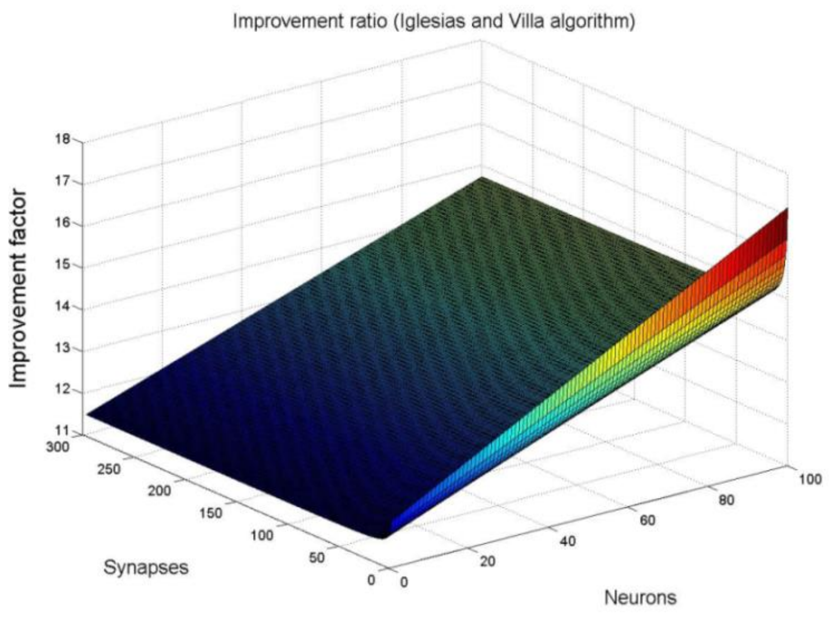


Figure 5.16: Improvement ratio for SNAVA against Ubichip simulating Iglesias and Villa model.

Figure 5.16 shows the improvement factor I_F which was obtained by dividing the number of clock cycles required to perform the Iglesias and Villa model in Ubichip and the number of clock cycles required to simulate the same model in SNAVA. The equation 5.6 allows the calculation of this improvement factor:

$$I_F = \frac{N_{TU}}{N_{TD}} \tag{5.6}$$

where: N_{TU} is the number of clock cycles required to perform the Iglesias and Villa model in Ubichip and N_{TD} is the number of clock cycles to perform the same model in SNAVA. The following values were taking into account to calculate the N_{TD}: P = 100 PEs, BS = 32, B = 8, N_v = 1 and ND = 1. ND implies the visualization of the post-synaptic spike and the membrane voltage on the monitor. N_{TD} was calculated by using the expression (5.2).

The improvement ratio between Ubichip and SNAVA was calculated in terms of clock cycles, considering up to 100 neurons and up to 300 synapses per neuron. For SNAVA only one level of virtual layer was taken into account for consistency in the comparison. As it can be observed in Fig. 5.16, the improvement factor is significant, but it is important to note that Ubichip was designed under restrictions in the area consumption, resulting in a low-performance implementation. The purpose of the Fig. 5.16 is only to show that enhancement factor increases with increase in the number of neurons and synapses taken into consideration. As it can be observed from Fig. 5.16, the improvement factor gradually increases linearly from 100 neurons – 100 synapses to 100 neurons – 300 synapses. In the current version of SNAVA is possible to support 100 neurons with 100 synapses. In the case of using a bigger FPGA the number of synapses can be incremented in SNAVA in order to allocate 300 synapses per neuron. As it was mentioned above the synapses were implemented in LUTs. The implementation of the 300 synapses is feasible for instance in the Virtex 7 FPGA (series XC7VX980T), which contains three times more number of LUTs when compared with the Kintex 7 FPGA (series XC7K325T), the available device for the developed prototypes.

5.4.2 Izhikevich model execution analysis

This section shows the performance evaluation by emulating the selected Izhikevich model [5] by simulating 100 neurons and 100 synapses per neuron. The description of the Izhikevich model is provided in Chapter 2. The algorithm, which has been written in assembler code for SNAVA, is provided in Annexure C. The number of clock cycles for each subroutine and also the whole algorithm in terms of the number of neurons and number of synapses per neuron has been reported below.

The required number of cycles to execute each subroutine in phase 1 is indicated in Table 5.6. The encoding of subroutines contained in the synapse loop is shown in Table 5.7.

Table 5.6: Main loop subroutine encoding and execution number of clock cycles

Symbol	Subroutine	Clock cycles*
T I	Thalamic Input	28
S U	Spike Update	23
M P	Membrane Potential	19
S E	Spike Enable	5
C S	Cycle per each synapse	(15) ·S
M V	Membrane Value	70
R V	Recovery Value	14

* The number of clock cycles depends on the number of synapses (S) and number of virtual layers (N_v)

Table 5.7: Synapse loop routine encoding

Symbol	Subroutine	Clock cycles *
S L	Synapse Load	1
S W	Synaptic weight	9
S S	Synapse Save	5

Equation 5.7 calculates the number of clock cycles. This was obtained by adding all contributions of Table 5.6.The growth in the number of clock cycles depends on two variables, the number of virtual layers and the number of synapses.

$$N_T = 159 \cdot N_v + 15 \cdot S \tag{5.7}$$

Where: N_T is the number of clock cycles, N_v is the number of virtual layers and, S is the number of synapses per processor

The scanning time to monitor the parameters of the model is not considered in previous equation (5.7). The number of clock cycles to compute the algorithm in phase one is calculated as follows:

$$N_{TD} = 159 \cdot N_v + 15 \cdot S + P\left(\frac{BS}{B}\right)\left(S \cdot SD + N_v \cdot ND\right) \tag{5.8}$$

Where: N_{TD} is the number of clock cycles, N_v is the number of virtual layers, S is the number of synapses per processor, P is the number of Processing Elements (PEs), B is the Ethernet bus width, SD is the number of synapse parameters to display, ND is the number of neuronal parameters to display and BS is the Buffer size.

1000 neurons with 100 synapses per neuron is the target network size proposed by Izhikevich [5], with a time step resolution of 1 ms. The execution time obtained is around of 2.1 ms to simulate one step of the Izhikevich model in Ubichip by simulating 100 neurons – 100 synapses per neuron, while the execution time to execute the same model with the same number of neurons and synapses in SNAVA is around of 1.72 µs. In both the cases, the spike distribution was calculated under worst-case assumption that all neurons would fire at every simulation cycle. This improvement guarantees the time step resolution for the simulation of Izhikevich under 1 ms, which is highly required for the simulation of this model [5]. Therefore, 1000 neurons can be emulated in SNAVA by using 10 FPGA boards, by keeping the same execution time (1.72 µs) in all FPGAs. This is because all FPGAs work in parallel. The spike transmission time would however increase.

The calculation of the execution time to perform the Izhikevich model in Ubichip was carried out as follows:

The execution time E_{TU} was calculated by using the expression 5.10. The expression (5.9) allows the calculation the number of clock cycles N_{TU} required to simulate the Iglesias and Villa model in a single step simulation in Ubichip.

$$N_{TU} = 11492 + 10xN + 128xS + 8xNxS \tag{5.9}$$

Where: the N is the number of neurons and S is the number of synapses. The time execution E_{TU} is obtained as follows:

$$E_{TU} = N_{TU} * T_{clk} \tag{5.10}$$

Where: the N_{TU} is the number of clock cycles to simulate the algorithm in a single step simulation and T_{clk} is the time of the clock of the system. In the case of Ubichip, the clock system works at 50 MHz.

The execution time E_{TS}, which is required to perform the Izhikevich model in SNAVA, is calculated by the expression 5.11,

$$E_{TS} = N_{TD} * T_{clk} \tag{5.11}$$

Where N_{TD} is the number of clock cycles, and T_{clk} is the time clock of the system. The calculation of the execution time E_{TS} of current prototype is particularized at 125 MHz clock or 8 ns period. N_{TD} is calculated by using the expression (5.8). The following values were taken: P = 100 processors, BS = 32, B = 4, N_v = 1 and ND = 1 to calculate the number of clock cycles N_{TD}. ND implies the visualization of the post-synaptic spike and the membrane voltage.

$$I_{F=\frac{N_{TU}}{N_{TD}}} \tag{5.12}$$

Although it has achieved a great improvement in terms of speed of processing, the increment in the system clock in SNAVA has increased the performance when compared with Ubichip, as it has been demonstrated in the previous analysis. However, a significant improvement has been achieved by modifying the structure of Ubichip. The improvement factor I_F is carried out by considering the number of clocks. Therefore, the clock of the system is not considered in the calculation. Fig. 5.17 shows the improvement factor in terms of clock cycles by considering 100 neurons and 100 synapses in both architectures. The calculation of this I_F was done by using the expression (5.12). The improvement would be around 6 times of the target criteria which is around 100 neurons with 100 synapses for each neuron, with one virtual layer for the case of SNAVA. This improvement is mainly due to the single cycle instructions for fetching and saving the synapse parameters and the hardware multiplier that simplifies the multiplication operations in the algorithm.

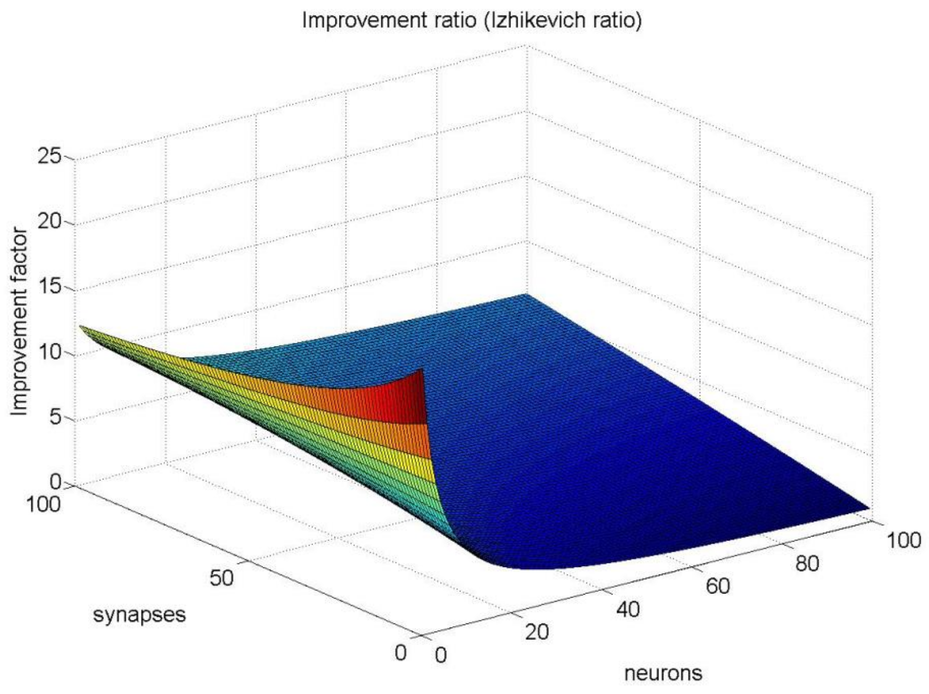


Figure 5.17: Improvement ratio for SNAVA against Ubichip simulating Izhikevich model

5.4.3 Leaky integrate-and-fire model execution analysis

The leaky integrate-and-fire model was presented in Chapter 4. The algorithm, which has been written in assembler code for SNAVA, is provided in Annexure C. The LIF model has been implemented on Ubichip and SNAVA in order to be used in the applications that involve processing of sensory information (see Chapter 6). The algorithm consists of 5 subroutines dedicated to compute the neural parameters and a loop to calculate synaptic parameters. The required number of cycles to execute each subroutine in phase 1 is indicated in Table 5.8. The encoding of subroutines contained in the synapse loop is shown in Table 5.9.

Table 5.8: Main loop subroutine encoding and execution number of clock cycles

Symbol	Subroutine	Clock cycles*
M P	Membrane Potential	26
C S	Cycle per each synapse	(21) ·S
S U	Spike update	34
R F	Refractory period	5
N S	Neuron save	15
S E	Spike Enable	5

* The number of clock cycles depends on the number of synapses (S) and number of virtual layers (N_v)

Table 5.9: Synapse loop routine encoding

Symbol	Subroutine	Clock cycles *
S L	Synapse Load	1
S W	Synaptic weight	19
S S	Synapse Save	1

The expression to calculate the total number of clocks to execute the algorithm in one step emulation is obtained by the addition of all contributions from Table 5.8:

$$N_T = 85 \times N_v + 21 \times S \tag{5.13}$$

Where: N_T is the number of clock cycles, N_v is the number of virtual layers and, S is the number of synapses per processor

The previous expression does not take into account the number of clock cycles to observe the activity of the network (neural variables or synaptic variables) through the monitor. The expression 5.14 adds the delay in order to complete the calculation:

$$N_{TD} = 85 \times N_v + 30 \times S + P\left(\frac{BS}{B}\right)(S \cdot SD + N_v \cdot ND) \tag{5.14}$$

Where: N_{TD} is the number of clock cycles, N_v is the number of virtual layers, S is the number of synapses per processor, P is the number of Processing Elements (PEs), B is the Ethernet bus width, SD is the number of synapse parameters to display, ND is the number of neuronal parameters to display and BS is the Buffer size.

The execution time required to perform the LIF model in SNAVA is around of 27 μs, while the execution time required for the same in Ubichip is around 5.14 ms in a single simulation step, considering the Perplexus target (which is 100 neurons and 300 synapses per neuron). The target of 1 ms time step resolution in Ubichip is not achieved. Despite the complexity of the SNN model to be simulated in Ubichip has been reduced by means of simulating one of the simplest SNN model, which does not demand high computation. In the case of reducing the number of synapses per neuron from 300 to 100, the execution time to perform the LIF model in Ubichip is 1.73 ms by implementing this amount of synapses in Ubichip is closer to the target of 1 millisecond step time resolution.

The calculation of the execution time to perform the LIF model in Ubichip was carried out as follows:

The execution time E_{TU} was calculated by using the expression 5.16. The expression (5.15) allows the calculation the number of clock cycles N_{TU} required to simulate the Iglesias and Villa model in a single step simulation in Ubichip.

$$N_{TU} = 260 + 16xN + 51xS + 8xNxS \tag{5.15}$$

Where: the N is the number of neurons and S is the number of synapses. The time execution E_{TU} is obtained as follows:

$$\mathrm{E_{TU}} = \mathrm{N_{TU}} * \mathrm{T_{clk}} \tag{5.16}$$

Where: the N_{TU} is the number of clock cycles to simulate the algorithm in a single step simulation and T_{clk} is the time of the clock of the system. In the case of Ubichip, the clock system works at 50 MHz.

The execution time E_{TS}, which is required to perform the LIF model in SNAVA, is calculated by the expression 5.17,

$$\mathrm{E_{TS}} = \mathrm{N_{TD}} * \mathrm{T_{clk}} \tag{5.17}$$

Where N_{TD} is the number of clock cycles, and T_{clk} is the time clock of the system. The calculation of the execution time E_{TS} of current prototype is done at 125 MHz clock or 8 ns period. N_{TD} is calculated by using the expression (5.14). The following values were taken: P = 100 processors, BS = 32, B = 4, N_v = 1 and ND = 1 to calculate the number of clock cycles N_{TD}. ND implies the visualization of the post-synaptic spike and the membrane voltage.

The increment in the value of the clock system in SNAVA allows to easy achieving the simulation of LIF in real time (1 ms step time resolution), however, SNAVA has important improvements with respect to the Ubichip architecture. The improvement factor clarifies the impact of these improvements implemented in SNAVA. This improvement factor I_F is obtained by dividing the number of clock cycles required to perform LIF in Ubichip and the number of cycles to perform the same model in SNAVA by using the expression (5.18):

$$I_F = \frac{N_{TU}}{N_{TD}} \tag{5.18}$$

Fig. 5.18 shows the improvement ratio achieved in SNAVA by simulating the LIF model in comparison with Ubichip. Evidently, the LIF model is one of the simplest SNN models which require less number of instructions when compared to the Hodgkin-Huxley model [6]. This model describes the neural dynamics in detail.

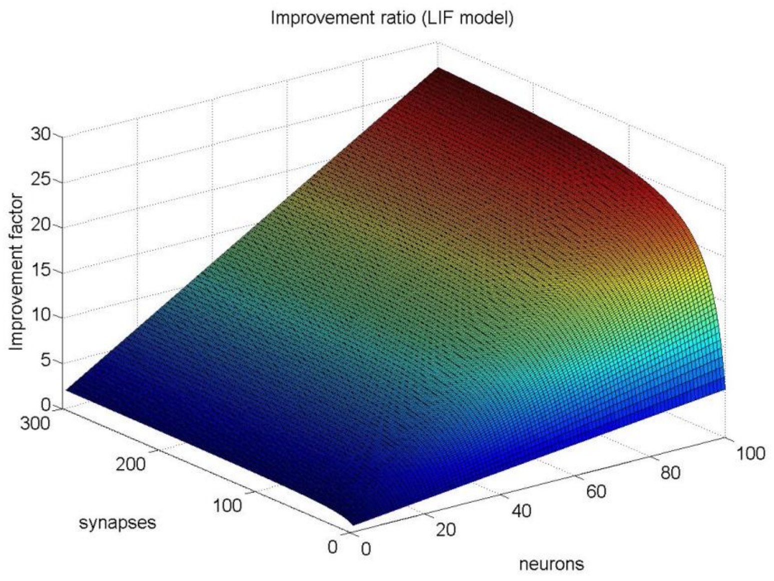


Figure 5.18: Improvement ratio for SNAVA against Ubichip simulating LIF model.

5.3.4 Processing time and distribution time for any SNN model

Three spiking neuron models were implemented in SNAVA, Iglesias and Villa model, Izhikevich model, and Leaky integrate-and-fire model. However, this architecture was designed to support any arbitrary SNN model under the condition that the communication between neurons is through spikes. In fact, the SNN models can be simulated with different levels of abstraction, however the mechanism to perform their variables follow the almost the same pattern of processing. The equation 5.19 generalizes the calculation of clock cycles for any SNN model implemented on SNAVA. The equation is defined by constants K_1 and K_2, the number of virtual layers and synapses per processor.

$$N_T = K_1 \cdot N_v + K_2 \cdot S \tag{5.19}$$

The general equation that calculates the number of clock cycles for processing phase and the monitor delay is expressed by 5.20.

$$N_{TD} = K_1 \cdot N_v + K_2 \cdot S + P\left(\frac{BS}{B}\right)\left(S \cdot SD + N_v \cdot ND\right) \tag{5.20}$$

Where: N_{TD} is the number of clock cycles, N_v is the number of virtual layers, S is the number of synapses per processor, P is the number of Processing Elements (PEs), B is the Ethernet bus width, SD is the number of synapse parameters to display, ND is the number of neuronal parameters to display and BS is the Buffer size.

The value of constants K_1 and K_2 for each SNN model are shown in Table 5.10. In order to calculate the total time required in one emulation step (processing phase + distribution phase) equation 5.21 must be added to equation 5.19 or 5.20 in each case, where the clock system is 125 MHz in the current prototype. The required number of clock cycles to execute the distribution phase N_{TD} depends on the number of neurons N_F that fire at every emulation step and the number of SNAVA chips N_{CHIPS}.

$$N_{TD} = N_F \cdot N_{CHIPS} \tag{5.21}$$

Table 5.10: Value of constants for three SNN models implanted on SNAVA

SNN model	K_1	K_2
Leaky integrate-and-fire	85	30
Izhikevich	159	15
Iglesias and Villa	152	122

5.5 Comparison with other architectures

Any architecture that intends to support the simulation of large scale neural networks must guarantee three aspects in order to achieve the maximum performance. These include low power consumption, less area consumption and faster processing time. At the present, only a few digital architectures are trying to cover these three aspects in order to achieve good performance besides offering great flexibility. Many modern digital systems like Graphics Processing Units (GPUs), multiprocessors and FPGAs provide a platform for designing highly parallel systems which are suitable for the emulation of SNN models. Evidently, these architectures are designed for general purposes making them less efficient in terms of area and power consumption in comparison with the dedicated hardware. This section is vital because it is necessary to show the impact of our contribution in the emulation of large scale SNN in the world of neuromorphic circuits, especially in the digital domain. The digital domain is chosen in order to be consistent. Therefore, the objective of the following analysis is to find the main advantages and drawbacks in emulating large-scale SNN models on SNAVA, and make a comparative study with the existing architectures which claim to emulate Large-scale SNN models efficiently. This takes into account numerous leading architectures reported in the literature.

5.5.1 Implementations on Multiprocessor

One of the representative SNN emulators based on multiprocessors is the custom SpiNNaker machine [8] which can emulate a large number spiking neural networks using a custom ASIC with asynchronous interconnection. This architecture is a reprogrammable platform to emulate the spiking neural networks. The Table 5.11 summarizes technical specifications of SNAVA architecture and SpiNNaker.

Table 5.11: Technical specifications of the SNAVA and other existing multicores implementations

Project Reference	This work for single FPGA	SpiNNaker
Neuron model	Izhikevich Iglesias and Villa Leaky-integrate-and-fire	Leaky-integrate-and-fire Izhikevich
Number of neurons	200	20,000
Number of synapses	9900	2,000,000
Scalability	yes	yes
Flexibility	yes	yes
Migration	yes	no
Processing step	1 ms	1 ms
Hardware	Kirtex-7	ARM 968E
Technology	28 nm	90 nm
Number of cores	100 processing elements	18 ARM9 processing
Frequency operation	125 MHz	200 MHz
Format representation	Fixed precision	Fixed precision
Number of bits	16 bits	32 bits

- Discussion of multiprocessor implementation

This customized architecture promises to be a powerful platform in the simulation of large-scale of SNN models. SpiNNaker can support up to 20000 neurons and 2000000 of synapses per Chip, as shown in Table 5.11. Evidently, the number of neurons and synapses per neurons supported in SpiNNaker are much larger when compared with SNAVA. However, there are some aspects to discuss about this architecture. One of them is regarding the communication system. They assume that the network will not be saturated and there is no mechanism of congestion. This mechanism of congestion is vital when the activity of large networks becomes more active. Another aspect is regarding the memory system of spiNNaker. A large amount of data is transferred from the external memory to the processors. The high memory-bandwidth data interfaces compensate the negative effect of transfer of data from the external memory to the processors.

5.5.2 Implementations on GPU

There have been efforts to develop SNN emulators using Graphical Processing Units. The developers of these devices argue that there can be an efficient design platform for parallel computing, due to its inherent parallelization. It is important to note that these devices were designed especially for parallel processing of graphics. However, few works have confirmed the advantages of using these devices as SNN emulators. Two of the representative works were proposed by [10] and [11]. The technical specifications of these works are summarized in the Table 5.12.

Table 5.12: Technical specifications of the SNAVA and other existing GPU implementation

Project Reference	This work for single FPGA	Nageswaran	Arista
Neuron model	Izhikevich Iglesias and Villa Leaky-integrate-and-fire	Leaky-integrate-and-fire Izhikevich	Izhikevich
Number of neurons	200	100000	7000
Number of synapses	9900	10000000	7000000
Scalability	yes	yes	yes
Flexibility	yes	no	yes
Migration	yes	no	yes
Processing step	1 ms	1 ms	1 ms
Hardware	Kirtex-7	NVIDIA GTX280	NVIDIA TESLA C2050
Technology	28 nm	65 nm	40 nm
Number of cores	100 processing elements	228 scalar processor (SISD)	448 CUDA cores
Area	(mm^2)	ND	ND
Frequency operation	125 MHz	1.2 GHz	1.15 GHz
Format representation	Fixed precision	Float point	Float point
Number of bits	16 bits	128 bits	128 ts

- Discussion of GPUs implementations

The two SNN emulators presented in this work are based on GPUs, which can compute a large number of neurons and synapses by using processors operating around of 10 times more higher frequency than SNAVA, as it can be observed from Table 5.12. The higher processing speed compensates the negative effect produced by the utilization of multi-threads. This computational technique reduces the performance of SNN calculation when a large amount of data must load to processors from the GPU memory or vice versa when the data are stored from processors to memory. The bottlenecks that are present in this type of systems are due to their memory access system, programmability and the limited memory bandwidth.

Nageswaran implementation [10] presents the development of large scale SNN model on GPUs, in particular the proposed by Izhikevich taking into account the STDP rule. There are many reasons which make difficult to map this learning rule in this architecture or even in any architecture. One of them is given by the required high memory bandwidth to store recordings of future events. This was discussed in Chapter 2. Therefore, this makes any architecture expensive in terms of hardware resources. Although this architecture is capable of simulating large scale spike neurons network with process of learning, there are several negative aspects to be discussed here. Most of them are associated to the structure of the system, so that there is depletion in the throughput of SNN computation. One of them is related to the optimization of the parallel execution, namely the number of threads that are limited in number so that not all cores are used. Another important aspect to be discussed here is the process of serialization of the instructions in a stream of multiprocessors. This implies that a greater number of clock cycles are required by executing an instruction in different processors. As mentioned above, the plasticity implemented in the synapses increases the complexity of control of threads.

The SNN emulator proposed by Arista [11] offers scalability in the definition of the number of synapses and neurons. This proposal intends to create a generic SNN implementation to be executed in any GPU device. But this has a cost, which is clearly observed in Table 5.12 when comparing with the capacity of the system for supporting a large scale SNN between this work and Nageswaran work. This system can support fourteen times less number of neurons and hundred times less number of synapses and has two times more number of available cores. This is because the technique proposed by Arista make regular the multithread execution, namely the same number of threads executes the same number of blocks. For that reason the complexity of the threads are reduced and this general implementation can be used in any GPU.

5.5.3 Implementations on FPGA

For several years there has been an evolution of astounding FPGA devices in the industry of programmable devices. The integration of multiple cores into these devices has eased the implementation of complex systems within a short span of time in comparison to the time requirements of an ASIC system. Many applications has been developed using these FPGAs due to its high configurability besides its guarantee towards a very high performance. In particular, neuromorphic systems are being implemented on FPGAs in order to support large scale SNN emulations taking the advantage of the available resources creating configurable architectures with high performance or high communication system. This can be clearly observed by analysing two of the most representative FPGA implementations. Cassidy [12] has proposed a system of supporting one million neurons in real time and another system is named Bluehive [7] which can support 64k neurons with 64 million synapses per each FPGA. The Bluehive project is composed by 64 FPGAs while the architecture of Cassidy is limited to be implemented in a single FPGA. The technical specifications of these works are summarized in the Table 5.13.

Table 5.13: Technical specifications of the SNAVA and other existing FPGA implementation

Project Reference	This work for single FPGA	Bluehive	Cassidy
Neuron model	Izhikevich Iglesias and Villa Leaky-integrate-and-fire	Izhikevich	Leaky-integrate-and-fire
Number of neurons	200	64000	1000000
Number of synapses	9900	64000000	1000000
Scalability	yes	yes	yes
Flexibility	yes	no	no
Migration	yes	yes	yes
Processing step	1 ms	1 ms	10 ms
Hardware	Kirtex-7 407600 Flip-flops 203800 LUTs	Altera Startix IV 1459200 Flips-flops 182400 LUTs	Virtex 5 SX240T 149760 Flips-flops 149760 LUTs
Technology	28 nm	40 nm	65 nm
Frequency operation	125 MHz	200 MHz	200 MHz
Format representation	Fixed precision	Fixed precision	Fixed precision
Number of bits	16 bits	16 bits	16 bits

- Discussion of FPGAs implementation

The earlier works indicated above intend to offer a platform to emulate large scale number of neurons where their efforts were focused to optimize the SNN processing by making emphasis in the processing system or communication system forgetting that both aspects are important to be considered when it requires the design and implementation of an architecture to simulate a large number of neurons efficiently.

The Bluehive system implements thousands of neurons and synapses by exploiting the communication system to the maximum, based on high serial speed links taking the risk that the system could be saturated when the activity of large scale SNN network will be increased, and the logic in order to prevent congestion is not guaranteed in this work. Another aspect to be analysed is the memory access when large number of parameters are to be processed, this reduces the processing time and the parallel architecture is not exploited to the maximum.

The work that implements one million of neurons in a single-FPGA is proposed by Cassidy. This system utilizes a large state cache and also consumes more time to implement such network. The important aspect that has to be discussed in this work is the implementation of more number of neurons but the limiting the number of synapses. This is because of large interconnections involved in the network produces major consumption of power, area resources in any architecture.

The mechanism implemented in both architectures to process huge amount of neurons are based on fixed pipelines stages, which reduce the capacity of the system for supporting different SNN models. These architectures were designed for simulating specific simple SNN models which does not implement the plasticity of the synapses which plays the major role to carry out the learning process. The whole architecture must be redesigned and implemented again for any small modification to support this type of biological mechanisms. Therefore, both the systems are suitable if the SNN model is fixed.

5.5.4 General discussion

As mentioned in the introduction of this section, there are three aspects to be evaluated on the neuromorphic systems which include the performance in power consumption, area consumption and processing time. SNAVA is compared with the previously discussed architectures which were implemented on multiprocessors, GPU devices and FPGAs. Not all architectures discuss about the power or area consumption which is considered to be a vital factor. This section presents the comparison between SNAVA and another approaches regarding on these two factors. Make this comparison is not a simple task due to the several factors to be into account, for instance the structure of the architecture, the number of neurons and synapses supported in each one, the technology of the devices, etc., for consistency in the comparison. Therefore, the proposed criteria to carry out the estimation of power and area consumption have been defined in the evaluation of a single processor. The Single processor of all previous architectures, which are analysed in this section, tends to implement the multiplexing time technique. The multiplex time technique increases the capacity of the systems to support large number of neurons, at the cost of time and memory bandwidth.

Table 5.14: Comparison of SNAVA performance against other existing approaches

Projects	Power consumption per processor	Area consumption per processor	Processing step time	Number of neurons per processor and synapses per neuron	SNN model
This work	4 m W*	994 Flip-Flops 1416 LUTs*	1ms	2 50	-Izhikevich -Iglesias and Villa -Leaky-integrate-and-fire
Spinnaker[8]	41.1 mW 35 mW	N/D	1ms	250 250	Leaky-integrate-and-fire Izhikevich
Bluehive[7]	N/D	18240 Flip-Flops	1ms	1000 1000	Izhikevich
Nageswaran[10]	N/D	N/D	1ms	400 1000	Leaky-integrate-and-fire Izhikevich
Arista[11]	N/D	N/D	1ms	15 1000	Izhikevich
Cassidy[12]	N/D	N/D	10ms	N/D 1	Leaky-integrate-and-fire

*This results have been obtained from the VIVADO tool report. This tool has many advanced strategies for synthesis and implementation to optimize the area and power consumption.

The estimation of the power consumption of Spinnaker system is presented in [13]. This architecture is composed of 48 SpiNNaker chips, and each chip contains 18 ARM cores, where each ARM core can support a population of around 250 neurons with 250 synapses per neuron. According to the data presented considering the evaluation of power performance, the average power consumption of each chip is 0.74 Watts. This means that every ARM consumes around 41.1 mW by supporting 250 neurons and 250 synapses per neuron in the case of Leaky-integrate-and-fire model simulation. And the power consumption required to simulate the Izhikevich model is around 35 mW per ARM core.

The estimation of the area consumption of Bluehive system was analysed in this thesis taking into account the technology of the FPGA in terms of number of Flip-Flops and LUTs. As it was indicating in the Table 5.14, this system was implemented on Altera Startix IV board, which contains a FPGA with 1459200 Flips-flops 182400 LUTs. The structure of their architecture was analysed in order to estimate the area consumption. The first assumption is related to the number of processing nodes required to emulate 64000 neurons. According to the information provided in this work every node has four Processing Engines (PEs) and every PE can emulate up to 1000 neurons. Therefore, the total number of nodes calculated is 16 this is obtained by using the equation 5.22.

$$T_{neurons} = N_{p\_nodes} \cdot N_{p\_engines} \cdot N_{neurons\_x\_engine} \tag{5.22}$$

The second assumption is related the use of registers to implement 64 processing engines. By looking at the structure of the architecture, the processing nodes are mainly used to execute operations like multiplications, subtractions and shift operations. Therefore, the largest amount of area consumption in this architecture could be associated to the Flip-flops consumption. This can be clearly observed in the specifications that the architecture is implemented on a FPGA with a million of Flip-Flops and reduced number of LUTs. They also indicated that the architecture was proposed to exploit the parallel computation and every node can be replicated homogenously when a large number of nodes are required. This feature facilitates the evaluation of the area consumption per each node. If the percentage of the area required for the implementation of Bluehive architecture is taken into account, then around 80% of the total number of flip flops (1459200) which is 1167360 are required. Then, the number of Flip-Flops to implement a single PE is around 18,240 Flip-Flops, which are calculated by the equation 5.23.

$$PE_{FF} = \frac{1167360_{\text{Flip -Flops}}}{64_{PE}} = 18{,}240 \text{ Flip-Flops} \tag{5.23}$$

The result obtained in this comparison reveals that the power consumption for every processor on SNAVA is 10.25 times lower than SpiNNaker, and the consumption of Flip-Flops is 18 times lower than Bluehive processor, taking into account the data of Table 5.14. The evaluation is done by taking hundred processors with 2 layers of virtualization. Evidently, the number of neurons and synapses in our work is lower. This is due to the strategy implemented in this work which uses a bank of registers per virtual layer. Hence there will be a rise in the power consumption as well as area consumption when the number of virtual layers is increased. Making the assumption of using bigger FPGA, and doing an extrapolation to verify the feasibility to continue working on the same manner, namely a bank of registers emulates a virtual layer. For instance, if 100 neurons and 100 synapses per neuron are emulated in a single processing element, it will require around 0.4 mW, 99400 Flip-Flops and 141600, where the capacity of the Kintex-7 FPGA is 407600 Flip-flops, and 203800 LUTs. Only one processor requires around 24 % of Flip-Flops and 69.4 % of LUTS of the total. Thus, a change in the strategy will increase the potential of our project in order to emulate more neurons and synapses by maintaining the area resources and power consumption down. This is possible if the neural values are stored in BRAMs as it has been done for the synaptic parameters. The current version of the system show some advantages compared to another approaches which are listed below:

1. Congestion

The problem of congestion of data during the process of communication in SNAVA is resolved with the help of the synchronous AER system along with the implementation of parallel and serial strategies in the distribution of spikes through the network. The mechanism behind this is that, every postsynaptic spike is transmitted serially to all CAMs via the AER bus, which reads the spikes to generate matches within a single clock cycle. It should be noted that the mechanism of congestion is not considered in other architectures. The saturation of the system will be produced when a large-scale neural activity is increased.

2. Time resolution

Almost all architectures follow the same trend in the processing of large number of synapses, except for the case of Cassidy where the number of synapses are limited. The number of instructions required to execute the SNN model varies in accordance with the presence of the presynaptic spikes. Therefore, the processor requires more or less instructions to update the neural values. These architectures are efficient in terms of number of instructions, however the time-step simulation (1 ms) is not guaranteed. Two aspects are guaranteed in the emulation of the SNN models in SNAVA. The first is the step time resolution (1 ms) and second is the constant power consumption. This is possible because the processing elements (PEs) execute the same number of neurons and synapses at every step simulation irrespective of whether there is or not pre-synaptic spike.

5.6 Improvements in SNAVA: SNAVA+

Starting from the analysis of the results of processing timing, area and power consumption of SNAVA some bottlenecks were identified regarding area consumption. Some changes and optimizations were proposed in previous section in order to improve this factor in the SNAVA architecture, with the ultimate aim of occupying less resource, mainly due to LUTs consumption of the FPGA, and therefore be able to increase the number of neurons and synapses that can be emulated. From this work is then born SNAVA+, the new version of SNAVA.

5.6.1 Brief description of SNAVA+ architecture

This short section only aims to briefly present in a purely descriptive way the architectural changes that have resulted from SNAVA to SNAVA+. For further detailed information about SNAVA+ can be found in “SNAVA+: a large-scale spiking neural network emulation architecture” [14].

SNAVA+ intends to exploit the neural multiplexing-time processing or best known as virtualization at the maximum. This is because the changes done in SNAVA+ consists of the change the hardware storing devices for the neural parameters to support more neurons when compared with SNAVA. In SNAVA, several neurons can be implemented in a single Processing Element by using the bank of registers. The PE is the arithmetic unit which is mainly in charge of performing neural and synaptic variable processing.

Each Processing Element has 7 banks of shadow registers which correspond to the 7 layers of virtualization. The implementation of these banks of registers generates a negative impact in the area consumption of the FPGA. The positive benefit of this approach is that the neural parameters that can be accessed in a single clock which helps to increase the processing speed of the SNN algorithms. In SNAVA+ architecture, the virtualization is made by using the Block Random-Access Memory (BRAMs) available on the FPGA. Whereby for each neuron is allocated with n words of memory which contains the neuronal parameters for that specific neuron (n is programmable by the user according to the model to be emulated and can vary from 1 to a maximum of 8 words per neuron) as shown in Fig. 5.19. Therefore, the area consumption is decreased, however the processing time is increased since access to neuronal BRAM memory requires two clock cycles to read a single memory position. In SNAVA, the maximum number of virtual layers per each processor which can be implemented in the target FPGA with 100 processors and 100 synapses per processor is 2. The number of neurons that can be emulated by using a single BRAM is shown in 5.23. This is because, each Block of RAM in the FPGA consists of 32 bits of data and 10 bits of address and since every neuron can have maximum 8 BRAM words.

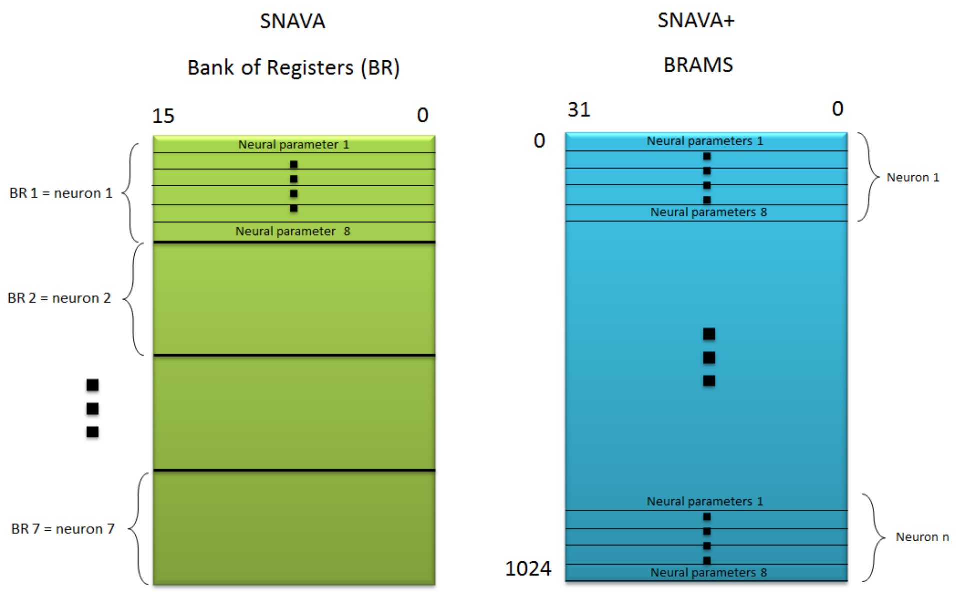


Figure 5.19: The allocation of neural parameters on SNAVA and SNAVA+

$$\text{Maximum number of neurons} = \frac{2^{10}}{8} = 128 \tag{5.24}$$

An important decrement in the consumption of registers and LUTs has been achieved in SNAVA+ by removing the shadow registers in SNAVA. Other profit is regarding the number of synapses supported in

SNAVA. Besides, a significant increment in the number of synapses per neuron is obtained. This will be illustrated in detail in the following sections of this chapter.

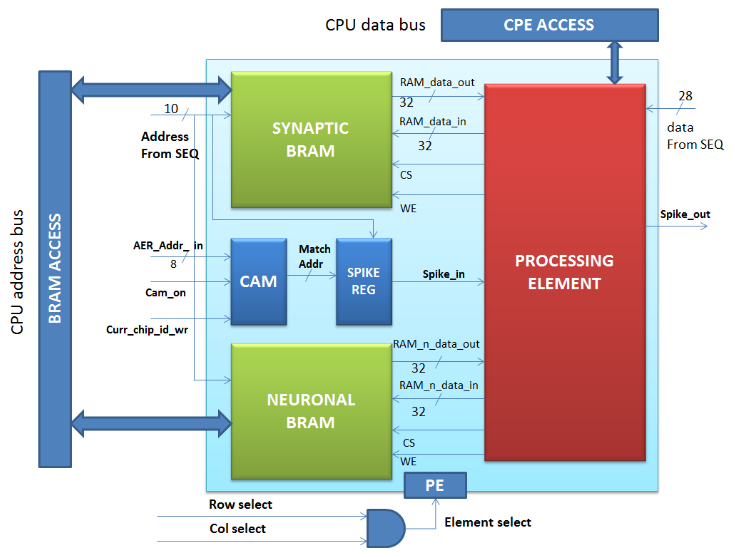


Figure 5.20: Structure of the Configurable Processing Element in SNAVA+

Figure 5.20 illustrates the new structure of the architecture of the Configurable Processing Element which includes the neuronal BRAM. The shadow registers have been removed from the Processing Element as shown in Fig. 5.21. The implementation of the neuronal BRAMS in the structure of SNAVA+ has consequently required several further changes and improvements. The details of the changes made on SNAVA to generate SNAVA+ are provided in the Master thesis of Mr. Vito Pirrone [14].

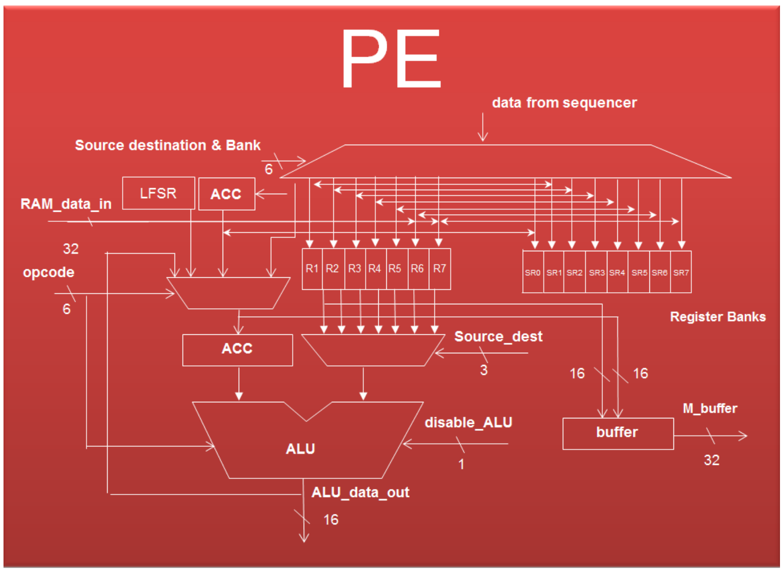


Figure 5.21 Processing Element data path

5.6.2 Implementation and performance

In this section, the post-synthesis results of SNAVA+ are shown and are compared with those of SNAVA, firstly in terms of area (resource occupation of FPGA) and secondly in terms of power consumption.

5.6.2.1 Area consumption

Table 5.15 shows the comparison between SNAVA and SNAVA+ in terms of area consumption. The comparison is made considering the same conditions for both the architectures, so it means the same array dimension (same number of processing element of the array) and the same number of synapses. Regarding the number of neurons, for SNAVA+ it should be considered that for any number of virtualized neurons (from 1 to 128) the same area is occupied on the FPGA, because regardless of the number of neurons a BRAM 1024-byte x 32-bit is synthetized for each processing element of the array.

Table 5.15: Area occupation of SNAVA+ with different numbers of synapses per Processing Element

Resource	SNAVA 10x10 2 levels of virtualization (99 synapses per PE)	SNAVA+ 10x10 n levels of virtualization* (99 synapses per PE)	Available
Flip-Flops	99487 – 24%	77444 – 19%	407600
LUTs	171291 – 84%	134400 – 66%	203800
BRAMs	135 – 15%	213 – 24%	890

Note: *n can be from 1 to 128

An important reduction of hardware resources can be obtained by the implementation of the BRAMs to store the neural parameters instead of using the bank of registers used for the same, as can be observed from the Table 5.15. The percentage in the consumption of Flip-Flops is reduced around of 5%. This is because the number of bank registers and instructions were removed from the original Processing Element. A detailed study was carried out to implement the instructions that are used with more frequency in the description of the SNN algorithms implemented in SNAVA+ [14]. As a consequence of this study, several instructions were removed. The best contribution to the new strategy is given to the consumption of the LUTs and the consumption of the available resources in the FPGA. Around of 18% in the consumption of the LUTs is a gain. This implies that a greater number of synapses can be implemented. Evidently, the increment of the BRAMs is visible, but the consumption is not significantly when compared the consumption of BRAMs in SNAVA.

Table 5.16 shows the area occupation of SNAVA+ with different numbers of synapses available per each Processing Element of the array. Regarding the number of neurons, for SNAVA+ it should be considered that for any number of virtualized neurons (from 1 to 128) the same area is occupied on the FPGA, because regardless of the number of neurons a BRAM 1024-byte x 32-bit is synthetized for each processing element of the array. Note: These results are obtained considering a chip ID of 4 bits, so considering a maximum number of 16 interconnected boards

Table 5.16: Area occupation of SNAVA+ with different numbers of synapses per Processing Element

Resource	SNAVA+ 10x10 n levels of virtualization* (50 synapses per PE)	SNAVA+ 10x10 n levels of virtualization* (100 synapses per PE)	SNAVA+ 10x10 n levels of virtualization* (200 synapses per PE)	Available
Flip-Flops	65216 – 16%	77444 – 19%	97824 – 24%	407600
LUTs	128394 – 63%	134508 – 66%	148774 – 73%	203800
BRAMs	213 – 24%	213 – 24%	213 – 24%	890

Note: *n can be from 1 to 128

As can be observed in Table 5.16, the percentage in the consumption in the number of LUTs and registers is increasing by 3% in the case of 100 synapses per Processing Element with respect to the implementation of 50 synapses per PE. If the percentage of consumption is increased by 3% for every 50 synapses per PE, the possible maximum number synapses that could be implemented in every PE is 500 by consuming around 90% of the total number of LUTs available in the FPGA. However, the time required to synthesize the design in VIVADO increases as shown in Table 5.17. This has prevented to implement so far prototypes with larger number of synapses.

Table 5.17: Synthesis time for the implementation for different number of neurons

SNAVA+ 10x10 n levels of virtualization* (50 synapses per PE)	SNAVA+ 10x10 n levels of virtualization* (100 synapses per PE)	SNAVA+ 10x10 n levels of virtualization* (200 synapses per PE)
4 hours	17 hours	62 hours

5.6.2.2 Power consumption

The total power consumption of SNAVA+ project is around 1.216 Watts, which can be calculated by adding the static power and dynamic power. As it can be observed from Fig. 5.22, the static power is around 0.186 W while the dynamic power is around 1.216 W. Regarding the static power would be constant for all the array sizes and dynamic power, which varies based on the design. The results of power consumption reported in this section were obtained by enabling the settings on VIVADO to optimize the power on the design. These options are:

1. power opt design
2. post placed power opt design

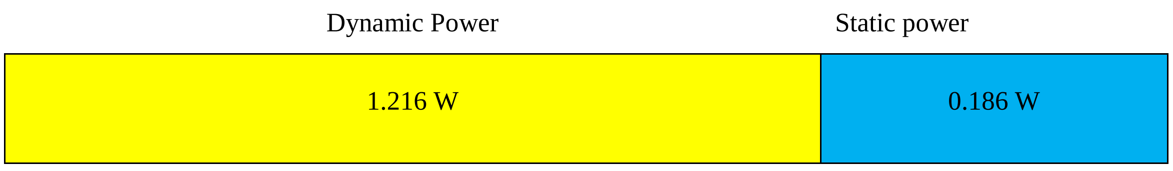


Figure 5.22: Power consumption of SNAVA+ with 10x10 PE array size and 99 synapses per PE unit

Table 5.18 shows the comparison between SNAVA and SNAVA+ in terms of power consumption. The comparison is obviously made considering the same conditions for both the architectures. So, it means the same array dimension (same number of processing element of the array) and the same number of synapses. Regarding the number of neurons, for SNAVA+ it should be considered that for any number of virtualized neurons (from 1 to 128) the same area is occupied on the FPGA, because regardless of the number of neurons a BRAM 1024-byte x 32-bit is synthetized for each processing element of the array.

Table 5.18: Power consumption of SNAVA and SNAVA +

Resource	SNAVA 10x10 1 levels of virtualization (99 synapses per PE)	SNAVA+ 10x10 10 levels of virtualization (99 synapses per PE)
SNAVA	0.649 W	0.922 W
Ethernet controller	0.039 W	0.043 W
AER controller	0.249 W	0.251 W
TOTAL	0.931 W	1.216 W

As it can be observed from Table 5.19, there is an increment in the power consumption of SNAVA+ when compared with SNAVA by an amount of 285 mW. The module of SNAVA in SNAVA+ project, which contains the configurable processing elements, the sequencer, and other components (for further details see [14]) is contributing around of 273 mW more than the SNAVA project. It should be noted that the sequencer and the other components maintain the same consumption for both versions (SNAVA and SNAVA+). Therefore, the bigger consumption is generated by the Configurable PE unit. Table 5.20 shows the consumption of each module of a single Configurable PE in SNAVA module. Evidently, the integration of the BRAM block to store the neural parameters in SNAVA+ is contributing to the total of 2 mW of power per each Configurable Processing Element.

Table 5.19: Power consumption of a single Configurable PE in SNAVA and SNAVA +

Resource	SNAVA 10x10 1 levels of virtualization (99 synapses per PE)	SNAVA+ 10x10 10 levels of virtualization* (99 synapses per PE)
CAM	0.001 W	0.001 W
Spike register	0.0001 W	0.0001 W
Processing element	0.0039 W	0.003 W
neuronal BRAM	-	0.002 W
synaptic BRAM	0.001 W	0.002 W
TOTAL	0.006 W	0.0081 W

Note: *n can be from 1 to 128

Table 5.20: Power consumption of a single PE in SNAVA and SNAVA +

Resource	SNAVA+ 10x10 10 levels of virtualization* (50 synapses per PE)	SNAVA+ 10x10 10 levels of virtualization* (100 synapses per PE)	SNAVA+ 10x10 10 levels of virtualization* (200 synapses per PE)
CAM	0.001 W	0.001 W	0.001 W
Spike register	0.0001 W	0.0001 W	0.0001 W
Processing element	0.003 W	0.003 W	0.003 W
BRAM neuronal	0.002 W	0.002 W	0.002 W
BRAM synaptic	0.002 W	0.002 W	0.002 W
TOTAL	0.0081 W	0.0081 W	0.0081 W

Table 5.20 shows the power consumption of SNAVA+ with different numbers of available synapses and virtualization level of 10 per each Processing Element of the array. These results are obtained by considering the chip ID to be 4 bits, so that the maximum number of boards that can be connected is 16.

5.6.3 Performance evaluation – Leaky integrate-and-fire model

The computation of the LIF model in SNAVA+ is carried out through two operational phases like in SNAVA. The model was described in Chapter 4. The processing phase (phase 1) is in charge of computing the neural and synaptic parameters, and the second phase is responsible of making the spike distribution through the SNN network. The following performance study is dedicated to analyse the processing speed. This is because the new strategy in the mapping of the neural variables affects the performance of the computation of the LIF model in SNAVA+.

Table 5.21: Neuronal loop subroutines

NEURONAL LOOP

Symbol	Subroutine	Clock cycles
L N P	Load neuronal parameters	2·N*
S N P	Save neuronal parameters	2·N*
M P	Membrane Potential	39
C S	Cycles per each synapse	(43) ·S
S U	Spike update	48
R F	Refractory period	5
N S	Neuron save	24
S E	Spikes enable	6

Note: *N is the number of words in the neuronal BRAM assigned to each neuron in order to store the neuronal parameters

The LIF algorithm consists of 7 subroutines dedicated to compute neural parameters and a loop to calculate synaptic parameters. The required number of cycles to execute each subroutine in phase 1 is indicated in Table 5.21. The encoding of subroutines contained in the synapse loop is shown in Table 5.22.

Table 5.22: Synaptic loop subroutines

SYNAPTIC LOOP

Symbol	Subroutine	Clock cycles
S L	Synapse Load	3
S W	Synaptic weight	21
S S	Synapse Save	1

Hence it can be formulated an equation that relates the number of execution cycles with the number of emulated neurons and synapses by adding the contribution of each subroutine to the total delay. There are two equations to compute the number of clock cycles to emulate the LIF model. The first equation (5.25), calculate the number of clocks without considering the delay to display the neuronal parameters in the monitor, and equation (5.26) considers the monitor delay produced by the visualization of the parameters.

Without parameters display:

$$N_T = 4 \cdot N_{BWN} \cdot N_v + 122 \times N_v + 43 \times S \tag{5.25}$$

With parameters display:

$$N_{TD} = 4 \cdot N_{BWN} \cdot N_v + 122 \times N_v + 43 \times S + P\left(\frac{BS}{B}\right)\left(S \cdot SD + N_v \cdot ND\right) \tag{5.26}$$

where: N_T and N_{TD} are the total number of clock cycles for the case of discarding the display of the parameters in the monitor, and considering the motorization time, respectively, N_{BWN} is the number of neuronal BRAM words per each neuron, N_v is the number of virtualization, S is the number of synapses per each Processing Element of the array, P is the number of Processing Elements, B is the Ethernet Bus width, SD is the number of synapse parameters to be displayed, ND is Number of neuronal parameters to be displayed, and BS is the Buffer size.

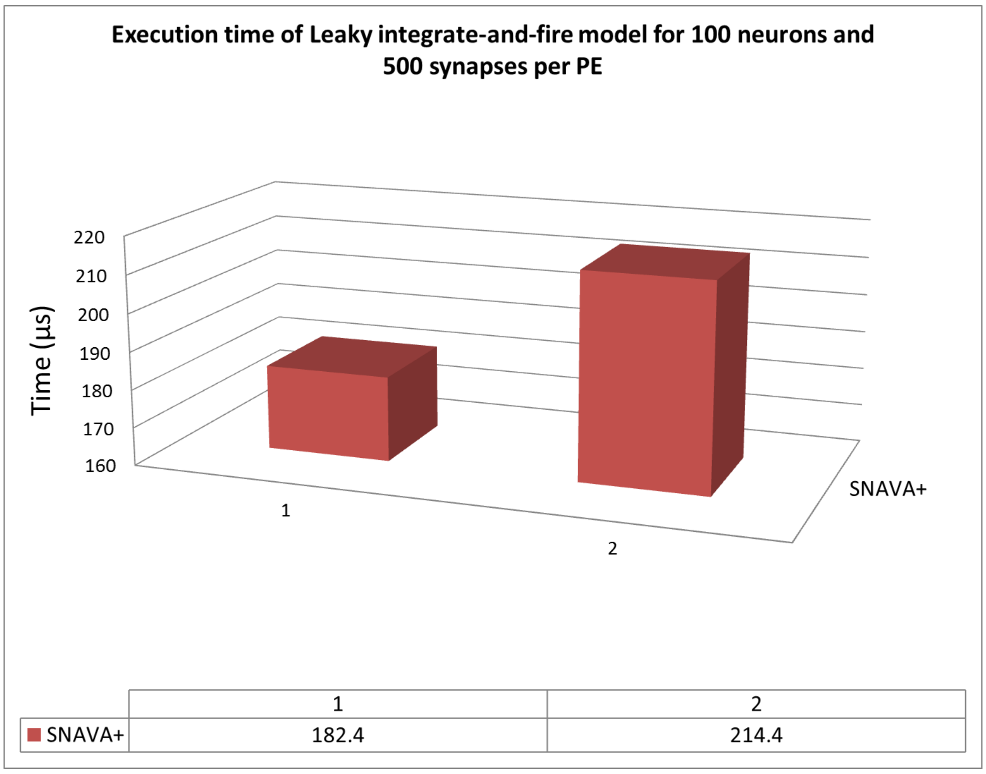
The execution time required emulating the LIF model in SNAVA and SNAVA+ is calculated, by considering 200 neurons and 50 synapses per neuron, and the results are shown in Table 5.23. In the case of SNAVA, equations 5.13 and 5.14 calculate the number of clock cycles for a single step simulation of the LIF algorithm. The delay generated by the display of the parameters is not considered in the equation 5.5, while the number of clock cycles required to display the neural parameters as a Postsynaptic spike, and the membrane voltage are considered in equation 5.14. In the case of SNAVA+, the equations 5.25 and 5.26 are used to calculate the number of clock cycles for the simulation of the LIF algorithm taken into account the same conditions from the equations 5.14 and 5.26, for consistency in the comparison between SNAVA and SNAVA+. SNAVA and SNAVA+ work under the frequency of 125 MHz which is the same clock frequency used by the communication interfaces such as Ethernet communication system and AER system, in order to avoid problems of synchronization.

The distinctive difference between equations 5.13, 5.14 and 5.25, 5.26, is the inclusion of a new variable N_{BWN}, which considers the number of BRAM words to store the neural parameters for each neuron in the case of SNAVA+. In the presented results, the number of neural BRAM words for each neuron is equal to 2. Since, it is sufficient to use two words of memory for storing the neural parameters of the LIF model. In the case of considering the implementation of the Iglesias and Villa model or Izhikevich in SNAVA+, the number of words to store the neural parameters will change according to the requirements of each model. The following values were taken for the equations 5.13, 5.14, 5.25, and 5.26: P = 100 processors, BS = 32, B = 4, N_v = 2 and ND = 1 to calculate the number of clock cycles N_{TD}. ND implies the visualization of the post-synaptic spike and the membrane voltage.

Table 5.23 shows the execution time required to compute the LIF model in SNAVA and SNAVA+ respectively. The calculation of the execution time takes into account the maximum number of virtual neurons and synapses per neuron (200 neurons and 50 synapses per neuron) supported by SNAVA. The execution time in SNAVA+ is evaluated considering the same number of neurons and synapses in order to compare these architectures under the same conditions. As can be seen from Table 5.23, SNAVA+ has a penalization of about 6 µs with respect to SNAVA in both the cases, without and with display.

Table 5.23: The execution time in SNAVA and the execution time in SNAVA+ (200 neurons and 50 synapses)

SNAVA		SNAVA+	
Without Display delay	With display delay	Without Display delay	With display delay
13.36 µs	16.56 µs	19.28 µs	22.12 µs



	1	2
🟥 SNAVA+	182.4	214.4

Figure 5.23: 1) Execution time for the simulation of 100 LIF neurons and 500 synapses without considering the delay produced for the visualization of the parameters in the monitor, 2) the execution time taking into account the display delay

The proposed target in the current version of the system called SNAVA+ is to implement 10 neurons and 500 synapses per PE. Therefore, the number of virtual neurons is 10 and every virtual layer has 50 synapses. The proposed target takes into account the results from the previous area consumption analysis. Figure 5.23 points out the execution time for the simulation of LIF model in SNAVA and SNAVA+ taking into account the proposed target. The bar charts reveal the loss of performance time in SNAVA+ when the parameters are sent to the monitor for visualization. The execution time to perform the LIF model in SNAVA is around 184 µs without displaying the parameters on the monitor and around 214.4 µs by considering the delay on the monitor. In both cases the execution of 100 neurons and 50 synapses per neuron is considered (see Fig. 5.23). The time step resolution is less than 1 ms for both cases, even though the delay is considered for the visualization of the parameters on the monitor. An important improvement is achieved in terms of area consumption by decreasing the processing speed by using two clock cycles to access to the neural BRAM. So this leads to an overhead which does not degrade the performance in a striking and significant way. So in the face of positive results in terms of area and power consumption, a slight loss of time performance time can be certainly considered a good trade off.

5.6.4 Comparison with other approaches

The digital SNN emulators presented and studied in Section 5.5 intends to perform large-scale SNN models. Several aspects were discussed regarding the flexibility, high processing performance, low power and area consumption in order to obtain an efficient SNN emulator. Some of these architectures try to take into account some of the factors mentioned above. However, there are important trade-offs which makes it impossible to offer an efficient architecture to emulate this type of SNN models. One of them is related to the area consumption and flexibility. SNAVA architecture intends to offer an emulator which supports large-scale SNN models by making equilibrium between these two factors. SNAVA offers the possibility to emulate different SNN models with different levels of computational complexity, from the simple Leaky integrate and fire model to the complex Iglesias and Villa model [4] or Izhikevich[5], and keeping the area consumption low. The results obtained in Section 5.5 reveals that SNAVA demands large area consumption in the implementation of the neurons. SNAVA+ is created to abate this consumption and generate a possibility to increase the number of neurons to be emulated. A significant improvement is obtained related to the increment in the number of neurons supported in SNAVA+ at the cost of time processing. The results of the performance evaluation of SNAVA+ indicate that the processing speed is minimally decreased.

SNAVA+ is compared with other approaches in order to clarify the contribution of this work. It is important to be noted that not all architectures, which were presented in Section 5.5, give the details of the power and area consumption of their implementations. Two of the most relevant works are compared with SNAVA+. The first work is implemented in a multiprocessor architecture called SpiNNaker [8]. The second work is implemented in a 64 FPGA boards [7]. In the comparison, the first architecture is selected to be compared with SNAVA+ due to the relevance of this work. Because this work can emulate a very large scale SNN models in a customized architecture. The second work is compared with SNAVA+ in order to make it clear that even though the FPGA has the main feature called configurability. This work does not exploit these advantages and they create a customized architecture for a specific SNN model.

Table 5.24: Comparison of SNAVA performance against other existing approaches

Projects	Power consumption per processor	Area consumption per processor	Processing step time	Number of neurons per processor and synapses per neuron	SNN model
SNAVA	0.004 W*	994 Flip-Flops 1416 LUTs*	1ms	2 50	-Izhikevich -Iglesias and Villa -Leaky-integrate-and-fire
SNAVA+	0.0039 W	774 Flip-Flops 61 LUTs*	1ms	10 20	-Leaky-integrate-and-fire
Spinnaker[8]	41.1 mW 35 mW	N/D	1ms	250 250	Leaky-integrate-and-fire Izhikevich
Bluehive[7]	N/D	18240 Flip-Flops	1ms	1000 1000	Izhikevich

*This results have been obtained from the VIVADO tool report. This tool has many advanced strategies for synthesis and implementation to optimize the area and power consumption.

Taking the data from Table 5.24, the SpiNNaker processor consumes around 41.1 mW and 35 mW for the emulation of LIF model and Izhikevich model respectively. Every processor can emulate up to 250 neurons and 250 synapses per neuron, while SNAVA+ consumes around of 3.9 mW per processor by simulating different SNN algorithms. Every processor can emulate up to 10 neurons and 20 synapses per neuron. In fact, SNAVA+ can emulate up to 128 neurons but the number of synapses per neuron is limited to 1.Therefore, SNAVA+ consumes thirteen times lesser than SpiNNaker processor. SpiNNaker can be more efficient than SNAVA+, respect to the synaptic calculation, because SpiNNaker process the synaptic parameters when there is a presynaptic spike while SNAVA+ processes all synapses irrespective of the presynaptic spike. Therefore, the regularity of the execution of SNAVA+ guarantees the power consumption. Hence, there is no significant rise in the power consumption when the SNN is more active. But, SpiNNaker cannot guarantee this important factor.

The area consumption in Bluehive was calculated in Section 5.5.4. According to this data, every processing unit consumes around of 18240 Flip-Flops, and it can emulate up to 1000 Izhikevich neurons with 1000 synapses per neuron. SNAVA+ consumes around of 774 Flip-Flops to emulate 10 neurons and 61 LUTs to implement 50 synapses per processing element. The consumption of SNAVA+ is 23 times lower than the Bluehive with respect to the consumption of Flip-Flops. This comparison is based on the consumption of the Flip-Flops because the strategy followed in Bluehive involves the implementation of simple processors which can support the Izhikevich neurons. The processors of Bluehive are in charge of performing the neural parameters and synaptic parameters of a neuron only when there is a spike to be processed by this neuron, so that, the data to update this neuron are transferred from external memory to this processor. Another issue to be discussed is the strategy followed to implement the role of the synapses in Bluehive, which differs from the SNAVA+. This is because Bluehive makes a customized connectivity configuration. Therefore, the routers require a simple logic based on LUTs. While SNAVA+ offers the possibility to create any topology and the neurons can have different number of synapses.

5.6 Conclusions

The flexibility is one the main features of SNAVA and SNAVA+ in terms of easy programmability of SNN models. The emulation of three SNN models with different levels of computational complexity has been demonstrated. Several ideas were taken into account to design an efficient architecture in terms of processing speed and distribution time. Therefore, the bottleneck is now not in the algorithm execution. However, the study of the performance of the implementation indicates that there are some remaining aspects regarding the area consumption which were discussed in this chapter. The strategy of time-multiplexing neural computation implemented in this work to emulate more neurons by using the bank of registers of CPE has produced large area consumption on the FPGA. If the register banks are being replaced by the BRAM, the amount of neurons will be increased by wasting lesser amount of area resources. The feasibility to implement this idea has been demonstrated by implementing SNAVA+. Here, greater number of neurons is supported by keeping the area consumption low. Another bottleneck is generated when large number of the synapses is implemented on the CAMs. This strategy allows the encoding of the spikes in parallel to a single pulse, but the area consumption becomes relevant when the number of neurons and synapses are increased. The current strategy to decode the spikes is efficient in terms of processing time by spending a lot of hardware resources. A possible solution to this problem is to change the implementation of CAM to BRAMs. The first benefit of applying this idea is to make the connections programmable. Currently, the synapses are recorded in LUTs, and each time the connectivity has to be changed the whole architecture has to be re-synthesized. Some of these modifications have been applied on SNAVA in order to improve its remaining weak points, so as to create a SNN emulator with high performance and low area consumption.

References

[1] E. Sanchez, A. Perez-Uribe, A. Upegui, Y. Thoma, J. M. Moreno, A. Napieralski, et al., “PERPLEXUS: Pervasive Computing Framework for Modeling Complex Virtually-Unbounded Systems”, in Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on, pp. 587-591, 2007.

[2] Athul Sripad, “SNAVA: A Generic Threshold-Based-SNN Emulation Solution”, Master Thesis, Universitat Politècnica de Catalunya, September 2013.

[3] Taho Dorta Pérez, “AER-RT: Interfaz de Red con Topología en Anillo para SNN Multi-FPGA”, Master Thesis, Universitat Politècnica de Catalunya, July 2013.

[4] J. Iglesias, J. Eriksson, F. Grize, M. Tomassini, and A. E. P. Villa, “Dynamics of pruning in simulated large-scale spiking neural networks”, Biosystems, vol. 79, 2005.

[5] E. M. Izhikevich, "Simple model of spiking neurons”, Neural Networks, IEEE Transactions on, vol. 14, pp. 1569-1572, 2003.

[6] A.L. Hodgkin, A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerves”, Journal of Physiology 117, pp 500–544, 1952.

[7] Moore, S.W.; Fox, P.J.; Marsh, S.J.T.; Markettos, A.T.; Mujumdar, A., Bluehive, “A field-programable custom computing machine for extreme-scale real-time neural network simulation”, Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, pp. 133-140, 2012.

[8] Khan MM, Lester DR, Plana LA, Rast A, Jin X, Painkras E, Furber SB, “SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor”, Neural Networks, 2008 IJCNN 2008 (IEEE World Congress on Computational Intelligence) IEEE International Joint Conference on, pp 2849-2856, 2008.

[9] Nageswaran, J.M., et al. “Computing spike-based convolutions on GPUs”, in Circuits and Systems, 2009, ISCAS 2009, IEEE International Symposium on, pp. 1917-1920, 2009.

[10] A. Arista-Jalife and R. A. Vazquez, "Implementation of configurable and multipurpose spiking neural networks on GPUs", in Neural Networks (IJCNN), The 2012 International Joint Conference on, , pp. 1-8, 2012.

[11] Cassidy A, Andreou AG, Georgiou J (2011), “Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis”, Information Sciences and Systems (CISS), 45th Annual Conference on, pp 1-6, 2011.

[12] E. Stromatias, F. Galluppi, C. Patterson, and S. Furber, “Power analysis of large-scale, real-time neural networks on SpiNNaker”, in Neural Networks (IJCNN), The 2013 International Joint Conference on, pp. 1-8, 2013.

[13] Vito pirrone, “SNAVA+: a large-scale spiking neural network emulation architecture”, Master Thesis, Universitat Politècnica de Catalunya, June 2014.

Proof-of-concept application on SNAVA

6

6.1 Introduction

This chapter shows an engineering application on SNAVA as a proof of concept in order to demonstrate the potential of SNAVA for processing sensors. This application involves detection of frequency and amplitude of a sinusoidal signal by means of the proposed bio-inspired system. A previous version of this application was developed on the Ubichip architecture in Chapter 4. There are several aspects, which were analysed and discussed in order to successfully detect the frequency of a sinusoidal signal by means of a Spiking Neural Network. The proposed system in this work uses the SNAVA architecture in order to extend the previous application. Some changes were done in order to detect both frequency and amplitude from the original version [1]. SNAVA offers several advantages in comparison with the Ubichip architecture. One of these advantages is related to high processing speed and spike distribution. This feature of SNAVA makes it feasible for the implementation of applications which involve processing time-varying signals, where the real time execution (1 ms time step resolution) of the emulator is required. In the available KC705 (based on Xilinx Kintex7) FPGA development board. SNAVA can emulate 200 neurons and 50 synapses per neuron much lower 1 ms time step resolution. Considering that the neuron can be modelled as Leaky Integrate-and-Fire (LIF) model, Izhikevich [2] or Iglesias and Villa model [3]. Operations such as integration and differentiation are some which are being done in circuits which work with time dependent signals. The various processes happening in the nervous system are based on these time dependent functions. One example of these functions is the process carried out in the ear, which responses to derivative signals. The work reported in [4], mimics these types of functions to carry out the audio signal processing by means of LIF model. The range of frequencies for this application is defined from 200 Hz and 7000 Hz. These types of functions also can be applied on SNAVA to perform various applications associated with the feature extraction of the sound. The current ongoing application is inspired on the function of the cochlea [5]. Preliminary results are presented in the Annexure D.

6.2 Bio-inspired system description

A bio-inspired system is proposed in this work in order to achieve the detection of the amplitude and frequency of a sinusoidal signal. Regarding the detection of the amplitude of the signal, the peak amplitude of this signal is estimated by means of the Spiking Neural Networks, while the frequency is detected by the bank of filters.

The proposed bio-inspired system is composed of three modules, which are:

1. Bandpass filter banks
2. Spike coders
3. Digital Multi Processor (DMP)/ Amplitude Classifier system

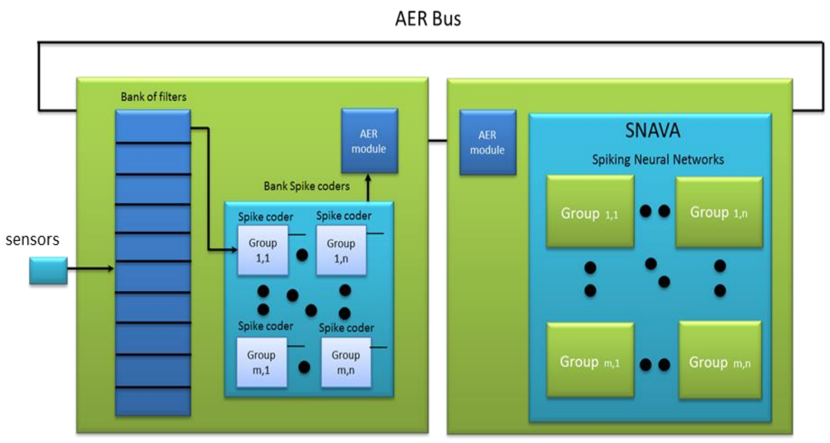


Figure 6.1: Amplitude and Frequency classification by means of bio-inspired system

The functionality of the bio-inspired system is as follows: the bio-inspired system takes information from the external sources by using sensors (MEMS, microphone, etc.). The output signal from the sensors is fed to the Bank of filters block. Here, each band-pass filter bank is centered to a particular frequency. Every spike coder processes the input signal, which is provided by the corresponding bank filter, translating the continuous signals into equivalent spikes. In other words there is a one-to-one connection between the each channel of the filter bank and the spike coder. Hence the output spikes from every spike coder block correspond to a certain frequency, as shown in Fig. 6.1. The spike coder is working on the same principle of operation (delta modulation) as encoders that were used in the previous work [1]. The spikes generated by the spike coders are sent to the Spiking Neural Network through the Address Event Representation AER bus [6]. The SNN network is divided into several groups of neurons, and each group of neuron indicates the amplitude of the input signal by processing the spikes received from its spike coder and the frequency of the signal is detected mainly by the filter bank. The firing of a particular group of neurons indicates which channel of the filter bank is active.

In the previous application, the time-to-space technique was proposed in order to process higher input frequencies by the Ubichip. This technique implies that the spikes, which are generated by the digital coders, are stored in a FIFO, while Ubichip is in its processing phase. These spikes are only transmitted to the Ubichip, when Ubichip is in its phase of distribution. As it can be observed in Fig. 6.2, the density of the train of spikes is variable. Indeed the input signal is asynchronous with respect to the phases of operation (processing and distribution) of Ubichip. This produces that the number of spikes stored in the FIFO varies according to the factors mentioned above for every simulation cycle. A new method to solve the problem is proposed in this work in order to have the correct number of spikes every emulation cycle independently of the density of the spikes and the synchronism between the processing phase and the input signal.

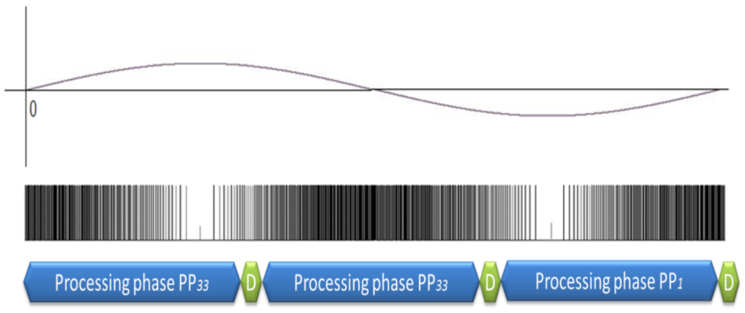


Figure 6.2: Phases of operation of Ubichip and spikes generated by the digital coders

The new method consists of sending the spikes to the SNN network only when the input signals have been processed by the spike coders for the first quarter after detecting the zero crossing, as shown in Fig. 6.3. In other words, the input signal will be processed by the spike coders, while SNAVA is in its distribution phase. When the spike coders finish processing their input frequencies, these spikes will be transferred to the SNAVA to resume its processing phase. Therefore, SNAVA can process a single sample of each coder every processing cycle. Only the first quarter is processed by the SNN network in order to determine the peak amplitude of the input signal.

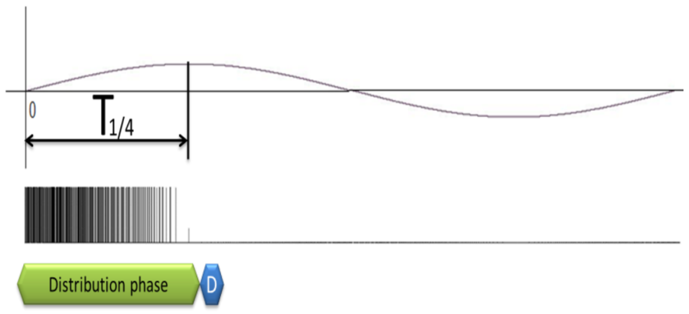


Figure 6.3: Phases of operation of SNAVA with respect to the input signal

The spike coder generates an enable_tx signal to indicate to the AER module that it has processed the first quarter of the signal after zero crossing detection, as shown in Fig. 6.4. Therefore, the spikes can be transferred to the amplitude classifier. The AER module can transfer these spikes to the amplitude classifier only when the SNAVA is in its distribution phase. After SNAVA finishes the processing of these spikes, it enters in its distribution phase. During this phase, it will wait for processing the next spikes.

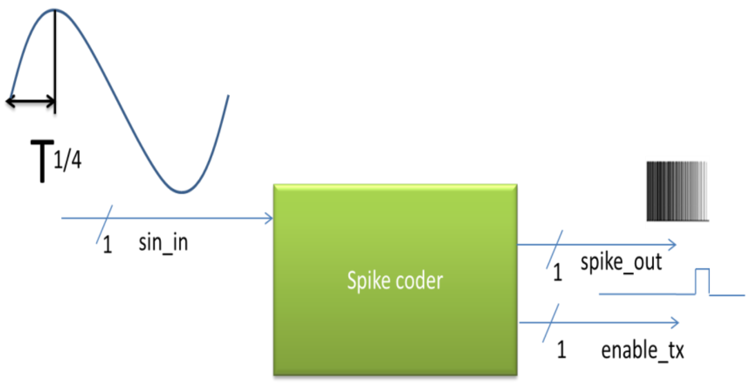


Figure 6.4: Spike coder

6.3 Amplitude detection

The SNN network consists of several groups of neurons which are in charge of detecting the amplitude of the signal. Each group of neurons is connected to a spike coder block that corresponds to a certain center frequency Fc. This group of neurons can be allocated as an array in SNAVA where each array element represents the group of neurons. This is feasible because of scalability of the SNAVA architecture. In this application, the degree to which SNAVA has to be scaled will be a function of the number of frequencies, as shown in Fig. 6.1.

The SNN network intends to indicate the amplitude of the input signal. It is important to be noted that the current version of the bio-inspired system only includes the spike generator and the SNN network to detect the amplitude at a certain frequency. Since the design, debugging and testing of the bank of filters demands more time it increases the overall development time of the bio-inspired system. Therefore, it has been considered that the signal fed has already been filtered and the spike coder generates spikes that have to be sent to the SNN network as in Fig. 6.3. As it can be observed from Fig. 6.3, a spike train is generated by the digital spike coders as in [7] the encoding a sinusoidal waveform. The VHDL code is provided in the Annexure E. In [7] the analog coders generate spikes with sign. The negative spikes are translated to positive to be processed by the DMP. This is because the DMP was designed to process positive spikes. In order to determine the peak amplitude of the signal filtered by the filter bank only the first quarter of the signal is considered (i.e. the first 25% of the signal) after detecting the zero crossing by the spike coders. This is shown in Fig. 6.3. Therefore, only the spikes generated by digital coders for this time are sent to the SNN network to indicate the peak amplitude of the signal after the detecting the zero crossing. The digital spike coder generates spike trains where the density of the spike train is in a function of slope of the input signal. So that the information regarding the amplitude is given by the number of spikes generated by the spike coder, while the time spacing between the spikes which were generated by the spike coders, is the information about the frequency of the signal. Since this information is indicated by the bank of filters, the space between spikes are not considered. However, the samples are preserved since these samples give information about the amplitude of the signal. To make sure there is no loss of samples, every sample is being mapped to a particular neuron and this is done till the end of one quarter cycle of the sinusoidal wave. Once the first quarter cycle is processed, the samples preserved in a FIFO, are immediately sent to the neurons of the amplitude detector block, which is shown in Fig 6.5. Here, the first sample is mapped to Neuron 1 and the next sample to Neuron 2 and so does the pattern continues till the end of the first quarter cycle. The number of samples that are being preserved solely depend on the number of neurons that are available in the input layer (layer 1) as it is shown in Fig 6.5.

6.3.1 SNN model description

The spiking neural model used in this application is modified from the original model proposed by Iglesias and Villa [3], which was already described in Chapter 2, in order to obtain a simple LIF model. There are some reasons to use this model in this application. One of these reasons is that the LIF model is suitable for applications which involve processing of time-varying signals [2, 8]. The model proposed by Iglesias and Villa models the neuron as LIF model and describes the mechanism of plasticity in the synapses. In this application, the plasticity of the synapses are not used.

6.3.1.1 Membrane Potential

$$\mathrm{V_i(t+1) = V_{rest} + \big(1 - S_i(t)\big)\big((V_i(t) - V_{rest})\big)k_{mem} + \sum \omega_{ji}(t)} \qquad (6.1)$$

Where:

$V_i\left(t+1\right)$	refers to the membrane potential of the neuron
$\mathrm{V_{rest}}$	corresponds to the value of the membrane resting potential
$\mathrm{S_i(t)}$	is the spike generated by the neuron
$\mathrm{k_{mem} = e^{\frac{-1}{\tau_{mem}}}}$	is the time constant associated to the current of leakage of the neurons

$\omega_{ji}(t)$ is the postsynaptic potential (excitatory or inhibitory)

The generation of the spike in the neuron $S_i(t)$ is a function of the membrane potential $V_i(t)$ and a threshold potential θ, such that $S_i(t)=H(V_i(t)-\theta)$, where H is the Heaviside function, H(x)=0: x<0, H(x)=1:x>0.

6.3.1.2 Synaptic strength

The post-synaptic potentials $\omega_{ji}(t)$ is a function of the state of the pre-synaptic unit S_j, post-synaptic potential in the synapses $P_{1,2}(t)$, where the postsynaptic potential $P_{1,2}(t)$ is fixed for the particular type of synapse (excitatory or inhibitory). The post-synaptic potential is expressed by the following equation:

$$\omega_{ji}(t+1) = S_j(t) \cdot P_{1,2}(t) \tag{6.2}$$

In the previous application [1], which was developed to detect the frequency level, the value of the synaptic and neural parameters of the SNN model was chosen artificially. Some of these values do not correspond to real time values with which the biological neuron operates. For this application, the values, which are not the same as proposed in the original model, are the post-synaptic potentials P_1 and P_2. Where the postsynaptic potential $P_{1,2}(t)$ is fixed for the particular type of synapse (excitatory or inhibitory). From Table 6.1 one can see that the values of P_1 and P_2 was selected very high, around of 10 times more than the original values, since the number of synapses per neuron and the number of neurons are very small (100 neurons and 8 synapses per neuron) in comparison with the values being proposed for the simulation of 10000 neurons with 300 synapses per neuron).

Table 6.1: Parameter list of the main variables used for leaky integrate-and-fire neurons

Variable	Original values	Modified values	Hexadecimal representation for the modified values	Short description
P_1	0.84 mV	10 mV	03E8	Excitatory Post synaptic potential
P_2	-1.40 mV	-20 mV	F830	Inhibitory Post synaptic potential
V_{rest}	-78 mV	-78 mV	E188	Membrane resting potential
θ_i	-40 mV	-40 mV	E1BA	Membrane threshold potential
$t_{refract}$	3 ms	3 ms	0003	Absolute refractory period
τ_{mem}	15 ms	15 ms	EF7D	Membrane time constant

The decimal values of the synaptic parameters and neural parameters were converted to hexadecimal values by using the expression (6.3), every increment in the value of the synaptic and neural parameters is represented by 1 μV.

$$\text{Hexadecimal} = (\text{V} * 100)_{16} \tag{6.3}$$

$$\text{Hexadecimal value} = ((\text{P}_1)_{10} * 100_{10})_{16} = 3\text{E}8_{16}$$

Where: V is the synaptic or neural value in decimal to be converted in hexadecimal value. For instance, the value of excitatory post synaptic potential P_1 (10 mV) is converted in hexadecimal by using the expression (6.3), its the corresponding value in hexadecimal is $3E8_{16}$.

The expression 6.4 is used for calculating the time constant associated to the current of leakage of the neurons

$$\text{k}_{\text{mem}} = (2^{\text{N}} - 1) * \text{e}^{-\frac{1}{\tau_{\text{mem}}}} \tag{6.4}$$

$$\text{k}_{\text{mem}} = \left((2^{16} - 1) * \text{e}^{-\frac{1}{15}}\right)_{16} = \text{EF7D}_{16}$$

Where: N the maximum number of bits to represent the time constant, and τ_{mem} is the membrane time constant.

6.3.2 Amplitude classifier

The selected topology consists of 50 neurons distributed in two layers per frequency band. The description of the topology was explained in Chapter 4. The same mechanism is used in this application. The only difference that it is applied to detect the amplitude whereas the same topology used in Chapter 4 detects the frequency of the input signal. This topology was proposed in order to process the spikes that are mapped from time to space by using the time-to-space converter (see Chapter 4). The number of spikes is a function of the value of amplitude of the input signal, while the information of the frequency is given by the inter-spike time. Therefore, the variation in the frequency allows to store different number of spikes, while the DMP (Ubichip) is in its processing phase. So that different levels of frequencies can be detected. The value of the amplitude is fixed to a certain value. In SNAVA, the change in the amplitude allows to have different number of spikes, which are stored in the FIFO of the time to space converter, by fixing the value of the frequency. By applying these spikes to the SNN topology, different levels of amplitude can be detected.

The number of bands to be implemented in SNAVA depends mainly on the capacity of the FPGA and the number of levels of amplitude to be detected by means of the SNN network. The current implementation of SNAVA has the capacity of 2 frequency bands by using a single virtual layer. The input layer has 40 neurons which are responsible to detect the input spikes and the second layer has 10 neurons which indicate the amplitude level.

6.4 Experimental results

The results presented in this section show the performance of the current version of the bio-inspired system which only includes the spike generator and the SNN network to detect the amplitude at a certain frequency. The filter bank has not been implemented in the current version of the bio-inspired system due to time constraints. The above work would be carried out in the future. Therefore, it has been considered that the signal fed has already been filtered as shown in Fig. 6.4. In this experiment, the signal provided by the function generator is fed to the Analog to Digital Converter (ADC). The ADC module is embedded in the AMS101 Evaluation Card [9]. This ADC is in charge of translating the continuous signals into equivalent digital words of 16 bits and the rate of conversion is 1 Msamples/s. All these modules have been implemented on two Xilinx KC705 development kits. The ADC was connected to one of these boards, as shown in Fig. 6.5. The implementation of the coders was done in a single board. This is because they can act as the input of the system network. The SNN network is implemented in another board in order to exploit the maximum capacity of the FPGA to support maximum number of neurons. These two boards are connected a ring topology [6] through the AER in order to distribute the spikes through the network. The chip id 1 indicates the board in which the spike coders have been implemented and chip id 2 is the board in which the SNN network is implemented as shown in Fig. 6.5.

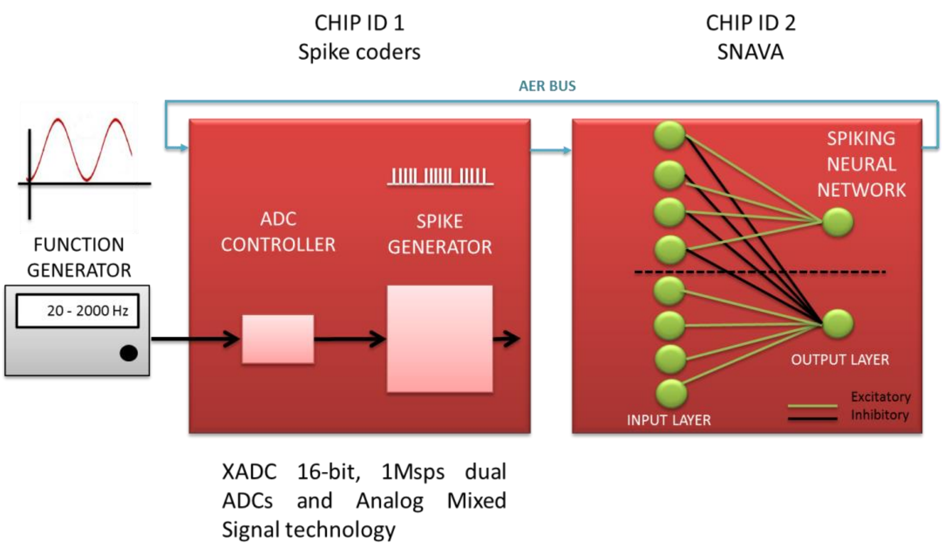


Figure 6.5: Implementation of the bio-inspired system

Our experiments were done by analysing three frequencies, which were generated by the function generator, with different amplitudes ranging from 0 to 1 Volt. The three frequencies proposed are 20 Hz, 200 Hz, 2000 Hz with five different voltages each one 100 mV, 300 mV, 500 mV, 700 m V, and 900 mV.

As mentioned earlier, every spike coder is connected to one group of neurons. These spike coders were programed to perform the detection of zero crossing. Besides, these coders generate ‘n’ number of spikes for every 100% of the amplitude of sinusoidal signal. However, in this application only the 25% signal is processed by the digital coders in order to achieve the detection of the amplitude by means of SNN (see Fig. 6.3). As it was mentioned above, the SNN network consists of 50 neurons in each group. This group has two layers; the first layer (40 neurons) is in charge of processing the spikes produced by the spike coders by using the time to space technique. Therefore, it is necessary to calculate the number of spikes produced by the spike coder in the first quarter of the signal in order to allocate 40 spikes in 40 neuron input layer, to indicate the maximum amplitude voltage which corresponds to 1 V. The 10 neurons output layer indicate the level of the amplitude, each neuron indicates an increment of 100 mV. The spikes are produced by the digital coder are in the same way that the spikes produced by the APP (see Chapter 4). The calculation of the maximum inter-spike time $T_{spikemax}$ indicates the time between spikes. The number of spikes produced by the first quarter is in function of the time $T_{1/4}$ and the $T_{spikemax}$ as is indicating by the equation 6.5. The number of spikes for the first quarter is calculated as follows:

$$\text{Number of spikes}_{1/4} = \frac{\mathrm{T}_{1/4}}{\mathrm{T}_{\text{spike(max)}}} \tag{6.5}$$

The process to obtain the $T_{spikemax}$ was described in Chapter 4, so that substituting the following values: A = 1 V, N_B = 8 bits, and the input frequency f_{in} = 20 Hz in the equations 4.4 and 4.5, the $T_{spikemax}$ is calculated.

$$\text{Number of spikes}_{1/4} = \frac{0.0125\text{s}}{62.41\mu\text{s}} = 200 \text{ spikes}$$

The maximum number of neurons in the input layer is 40. Therefore, every 5 samples are processed by the spike coders in order to obtain 40 spikes when the amplitude is 1 volt, where these samples are provided by the ADC in every 1 µs. The procedure to calculate the maximum number of spikes which are generated by the first quarter was applied to 3 different frequencies, and tabulated in Table 6.2.This table summarizes the values obtained taking into account the same data except for the value of the frequency of the signal f_{in}.

Table 6.2: Number of samples for three different frequencies in the first quarter of the signal

Frequencies f_{in}	$T_{1/4}$	$T_{spikemax}$	Number of spikes
20 Hz	12.5 ms	62.41 µs	200
200 Hz	1.25 ms	6.241 µs	200
2000 Hz	0.125 ms	0.6241 µs	200

According to the SNN topology proposed in this work, every block of neurons is composed by two layers, the first layer contains 4 neurons and the second layer contains 1 output. The output of each neuron of the second layer indicates an increment of 100 mV. Table 6.3 shows the number of input spikes which are required to be mapped in the input layer, in order to detect the increment of 100 mV in each output neuron. The allocation of the input spikes in the input layer is based on the time to space translation. This technique was explained in Chapter 4.

Table 6.3: Input layer neurons to detect the peak of amplitude from 0 V to 1 V

Voltage	Number of input spikes	Voltage	Number of spikes
1000 mV	40	500 mV	20
900 mV	36	400 mV	16
800 mV	32	300 mV	12
700 mV	28	200 mV	8
600 mV	24	100 mV	4

The number of spikes produced by the spike generator, by considering the range from (20 Hz to 2000 Hz), reveals that the value of the frequency of the input signal does not have any incidence in the calculation of the number of spikes for different frequencies as shown in Table 6.4

As it mentioned in the beginning of this section, 3 frequencies have been used to test the bio-inspired system (see Fig. 6.6). Every frequency was tested by using one group of neurons from the available two. This is because the system has only one ADC, which is connected to the spike coders. Every signal was introduced to the system for 300 cycles of simulation. Every cycle of processing phase in SNAVA lasts for 8.68 µs by executing the leaky integrate-and-fire model. This time was calculated by using the expression 6.6.

$$N_T = 85 \times N_v + 30 \times S + P\left(\frac{BS}{B}\right)\left(S \cdot SD + N_v \cdot ND\right) \tag{6.6}$$

Where: N_{TD} is the number of clock cycles, N_v is the number of virtual layers, S is the number of synapses per processor, P is the number of Processing Elements (PEs), B is the Ethernet bus width, SD is the number of synapse parameters to display, ND is the number of neuronal parameters to display and BS is the Buffer size. The following values were considered in the calculation of a single processing phase in SNAVA. Nv =1, S= 8, P = 100, BS = 32, B = 4, SD = 0, and ND = 1. The value of ND implies the sending of the neural parameters to be displayed on the monitor. In this case, the membrane voltage and the post-synaptic spike can be observed. This expression gives the number of clock cycles to execute the LIF model, so that this value has multiplied by the value of the clock system which is 125MHz.

Table 6.4: Number of spikes produced by the spike coder for different frequencies (20Hz-2000Hz) @ 1Volt

f (Hz)	$f_{spike(max)}$ (Hz)	$T_{spike(max)}$ (s)	$T_{1/4}$ (s)	Number of spikes
20	16028.57	6.23E-05	0.0125	200.3
30	24042.85	4.15E-05	0.0083	200.3
40	32057.14	3.11-05	0.0062	200.3
50	40071.42	2.49E-05	0.005	200.3
60	48085.71	2.07E-05	0.0041	200.3
70	56100	1.78E-05	0.0035	200.3
80	64114.28	1.55E-05	0.0031	200.3
90	72128.57	1.38E-05	0.0027	200.3
100	80142.85	1.24E-05	0.0025	200.3
200	160285.71	6.23E-06	0.0012	200.3
300	240428.57	4.15E-06	0.00083	200.3
400	320571.42	3.11E-06	0.00062	200.3
500	400714.28	2.49E-06	0.0005	200.3
600	480857.14	2.07E-06	0.00041	200.3
700	561000	1.78E-06	0.00035	200.3
800	641142.85	1.55E-06	0.00031	200.3
900	721285.71	1.38E-06	0.00027	200.3
1000	801428.57	1.24E-06	0.00025	200.3
2000	1602857.14	6.23E-06	0.0005	200.3

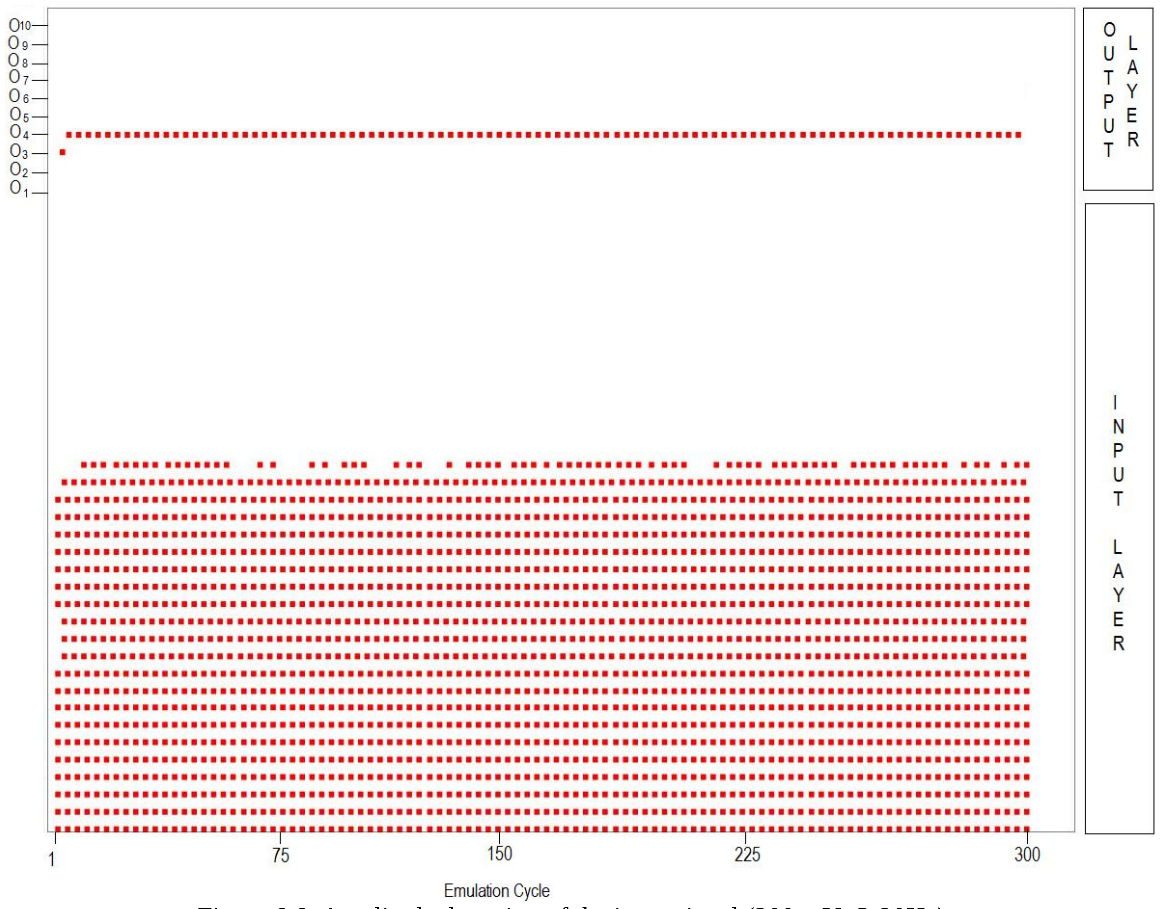


Figure 6.6: Amplitude detection of the input signal (300 mV @ 20Hz)

Figure 6.6 shows the output of the group of neurons by enabling the frequency of 20 Hz and the amplitude of 300 mV. As it can be observed from the Fig. 6.6, the output neuron 4 is indicating that the amplitude is around 300 mV. But theoretically, neuron 3 must indicate the amplitude value as 300mV. One of the possible reasons to justify this deviation in the neuron number is due to noise existing in the conversion of the ADC of the input signal at low frequencies (20-200Hz). The maximum value of the noise at low frequencies is around of 40 mV, and the value of the threshold of the spike coder is 30mV. A similar test was being done and for certain cases the results were coinciding with the theoretical values and in some it was not. The summary of the series of experiments the network underwent has been tabulated in Table 6.5.

Table 6.5: Amplitude and frequency detection for 3 different frequencies
{This summarises the results obtained by the detection of 3 different frequencies (2 Hz, 200 Hz, and 2000 Hz) with 5 different voltages each one (100 mV, 300 mV, 500 mV, 700 m V, and 900 mV).}

Frequency	Amplitude	Output neuron
20 Hz	100 mV	1
	300 mV	4
	500 mV	5
	700 mV	7
	900 mV	9
200 Hz	100 mV	1
	300 mV	4
	500 mV	6
	700 mV	8
	900 mV	9
2000 Hz	100 mV	1
	300 mV	3
	500 mV	5
	700 mV	7
	900 mV	9

6.5 Conclusion

The bio-inspired system for processing the information from the multiple-input sensor is presented in this work. The amplitude and frequency detection application was implemented by using this bio-inspired system. The proposed system extracts the frequency of the input signals by using a bank of band-pass filters. Therefore, the information of the amplitude is given by the number of spikes that are generated by the coders in the first quarter cycle of the signal after the detection of zero crossing. The detection of the amplitude of the input signal was achieved successfully.

The results presented in this chapter were obtained by running the simulation for 300 cycles and keeping the same input signal during the whole simulation. A future testbench will include the evaluation of the system on real time emulation, where the amplitude will change during the simulation in order to observe the transitions in the change of the amplitude. The current version of the software which allows the displaying of the neural activity suffers for some limitations, in particular to display the simulation of the SNN on real time (1 ms step time resolution).

References

[1] Sanchez, G.; Koickal, T.J.; Sripad, T.A.A.; Gouveia, L.C.; Hamilton, A.; Madrenas, J., "Spike-based analog-digital neuromorphic information processing system for sensor applications," Circuits and Systems (ISCAS), 2013 IEEE International Symposium on, pp. 1624-1627, 19-23 May 2013.

[2] E. M. Izhikevich, "Simple model of spiking neurons," Neural Networks, IEEE Transactions on, vol. 14, pp. 1569-1572, 2003

[3] J. Iglesias, J. Eriksson, F. Grize, M. Tomassini, and A. E. P. Villa, "Dynamics of pruning in simulated large-scale spiking neural networks," Biosystems, vol. 79, 2005.

[4] Wysoski, Simei Gomes and Benuskova, Lubica and Kasabov, Nikola, "Evolving spiking neural networks for audiovisual information processing," Neural Networks, pp. 819-835, 2010.

[5] Smith, L.S.; Fraser, D.S., "Robust sound onset detection using leaky integrate-and-fire neurons with depressing synapses", Neural Networks, IEEE Transactions on, vol. 15, no.5, pp. 1125-1134, Sept.2004.

[6] Taho Dorta Pérez, "AER-RT: Interfaz de Red con Topología en Anillo para SNN Multi-FPGA," Master Thesis, Universitat Politècnica de Catalunya, July 2013.

[7] Gouveia, L.C., T.J. Koickal, and A. Hamilton, "An asynchronous spike event coding scheme for programmable analog array," in Circuits and Systems, 2011, ISCAS 2011, IEEE International Symposium on, pp. 791-799, 2011.

[8] A. van Schaik, E. Fragniere, and E. Vittoz, "A silicon model of amplitude modulation detection in the auditory brainstem," In M.C. Mozer et al., editor, Advances in Neural Information Processing Systems 9, MIT Press, pp. 741-747, 1997.

[9] http://www.xilinx.com/products/boards-and-kits/DK-K7-EMBD-G.htm

Conclusion and ongoing work

7

This chapter presents the conclusion as well as the ongoing work. The conclusion gives details about the observations, the original contributions and issues in the development of the bio-inspired system to emulate large-scale SNN models efficiently. The ongoing work is dedicated to the development of a sound application implementation which based on the onset detection. This application is inspired by the functionality of the cochlea. The preliminary results of this implementation are provided in the Annexure B.

7.1 Conclusion

In this thesis, several techniques and methodologies were used for the development of a bio-inspired configurable system for efficient emulation of Large-scale SNN models in FPGA devices. The term efficient refers mainly to the factors to be considered in the Large-scale SNN emulation in compact digital devices. These factors are high processing speed, high interconnectivity, low area and power consumption. In addition to these factors, the bio-inspired system was designed to take maximum advantage of configurability a main feature offered by FPGAs . Therefore, this bio-inspired architecture intends to offer an interesting development tool to support different SNN models by exploiting the re-configurability of the FPGA and minimizing implementation time.

The methodology proposed in this work consisted of two phases in order to achieve the configurable SNN emulator. The first phase was dedicated to the debugging and commissioning of the complex prototype called Ubichip which was proposed in the PERPLEXUS project [1]. These tasks have consumed a significant amount of time and effort because large number of errors were detected and solved. During this period of time, several SNN models were simulated in Ubichip in order to verify its performance. Also, the development of a data interface was done in order to provide sensory information to the Ubichip from an analogue bioinspired pre-processor. This gave an opportunity for the implementation of an application which involves the detection of frequency of a sinusoidal signal by means of spiking neurons. The value of the neural and synaptic parameters involved in the SNN algorithm used in this application was analysed and fixed to determine the right measures to assess that the network activity corresponded to desired behaviour in the application. Many tests were carried out to achieve successful implementation. The configurability of Ubichip has helped in the easy implementation of different algorithms in relatively short time. The second phase was dedicated to the development of the SNAVA SNN emulator and also to the development of proof-of-concept sensing applications. The use of modern FPGAs requires the knowledge of new software tools which synthesize and implement the design in an optimized manner. Several technical issues regarding the implementation of SNAVA in Kintex-7 FPGA were attended. These include the timing delays. These important delays must be taken into account in the design of any digital system that is implemented in new FPGA technologies, because they work at high frequency clocks.

From this point, several original contributions have been made. Firstly, the performance of the Ubichip was quantified and several ideas were obtained from these results. The design of SNAVA takes into account the bottlenecks of Ubichip. Thereby, the contribution is given by the optimal resource allocation in the FPGA. One of these resource allocation is regarding the memory system. Another important contribution is the development of customized instructions in order to increase the performance of the processing system. Therefore, SNAVA provides several features in order to implement different SNN model for different applications:

- Multi-model support
- Different number of synapses per neuron
- Connection between neurons is point to multi-point
- Virtual neuron support

The SNAVA architecture has been compared with other state-of-the-art SNN digital emulators. A distinguishing factor of our architecture is the time-driven processing, in contrast with event-driven processing of most digital implementations. Although event-driven may be more efficient in processing time, it splits from the real-time biological principle as it is very inconvenient (and virtually impossible) to model local noise sources, as this would require continuous event processing. The good point, and also the limitation of the time-driven architectures is that they can faithfully emulate analogue neurons (artificial or biological) provided the processing time step is kept small enough. In this direction, the fact of using simple and time-multiplexed processing elements confers a distinguishing advantage compared with other approaches for network scalability.

7.2 Future work

The available SNAVA prototype is being tested to support various neural models. Hence SNAVA is now being used for the development of the onset detection system proposed in [2]. The onset detection system is an application that determines the sudden changes or the beginning of an acoustic signal. In other words, the functionality of the human middle ear i.e. the cochlea is tried to be imitated using the SNAVA architecture. This application was initially done in the simulation level using MATLAB by the Prof. Leslie Smith team from University of Sterling, Scotland. The future work aims to the implementation of the same application using a hardware platform. The preliminary results are obtained and discussed in Annexure B of this thesis.

There is a remaining point that has to be resolved is the area consumption in SNAVA+. This consumption is due to the implementation of large number of synapses. The current approach requires a large amount of hardware resources. Therefore, there must change in the strategy in order to avoid unnecessary consumption of hardware. The possible method to save the area is to build a system with routers. The current approach in the connection is point to multipoint. Therefore, the flexibility is being reduced because these routers make the connections point to point.

Also the modelling the delay in the dendrites and the axon is a remaining work. Many implementations do not take them into account either, but this is very important aspect since in the biological neuron such delays present in the dendrites and axon are supposed to play a relevant role. Hence in order to realize a more realistic application in SNAVA this modification has to be done which will also improve the performance of SNAVA.

References

[1] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, and J. Madrenas, "The Perplexus bio-inspired reconfigurable circuit", in Adaptive Hardware and Systems, AHS 2007, Second NASA/ESA Conference on, pp. 600-605, 2007.

[2] M. J. Newton and L. Smith, "A neurally inspired musical instrument classification system based upon the sound onset", J. Acoust. Soc. Am. 131, pp. 4785–4798, 2012.

Set of instructions of SNAVA

A.1 Introduction

This annexure describes the operation of the SNAVA sequencer, the format and classification of its instructions, the algorithm structure and finally the mapping of the registers of this sequencer.

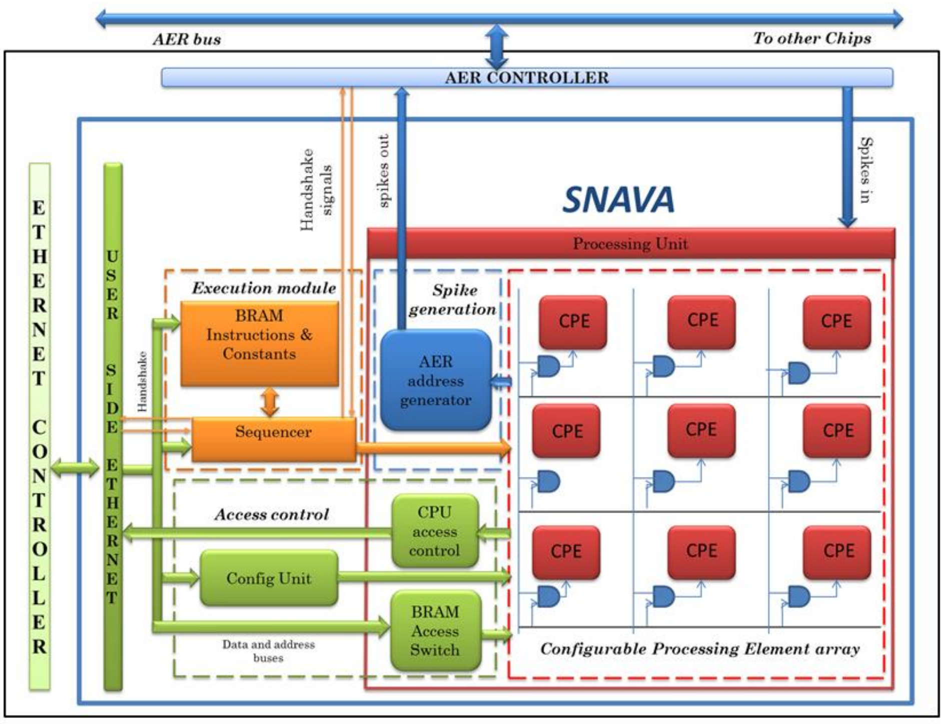


Figure A.1: Architectural Overview of SNAVA

The sequencer controls the program flow of computing the Spiking Neural Network algorithms. Several algorithms can be performed by SNAVA, under the condition that the communication between neurons is represented by spikes. This is possible because the instructions to execute the algorithm are stored in the Block of RAM. Only changing the memory program different SNN algorithms can be simulated. This represents the main feature of SNAVA called programming flexibility. The sequencer read the instructions from this BRAM and decoded it to indicate the operation to be executed by the ALU of the Processing Element (PE) when required. As it can be observed in Figure A.1, the sequencer is a single block that is external to all Configurable Processing Elements. This block is responsible for fetching and decoding the instructions stored in a block of RAM, broadcasting the instruction to be executed by the PE array, executing the instructions specific for the sequencer itself, and provides synapse count to the spike register and synaptic BRAM of the PE to deliver the right data to the PE. The details of these functions will be explained in the next sections.

A.2 Operation of the Sequencer

The sequencer provides two phases of operation, the processing phase (phase 1) and the spike communication phase (phase 2). In phase 1, is used for the general parallel execution in the PE array. The sequencer executes instructions and controls the execution of the PE array. After every cycle of synapse and neuron dynamics computation, the generated output spikes have to be transmitted by the AER module to all the neurons inputs by means of the AER bus [1]. During phase 2 the sequencer stop its operation, only the sequencer resumes its processing phase when the AER module finish its spike communication distribution.

In Figure A.2, a time diagram shows the handshake signals. The default sequencer operation is in processing phase, so after the system reset, it operates in phase 1. When the sequencer executes a SpikeDis instruction, it enters the halt state, and sets the eo_exec signal, indicating the phase 2 starts. The AER module perform the distribution of the spikes through the AER bus, when this module finishes it sets the ei_exec signal. This signal is read by the sequencer to resume its processing phase.

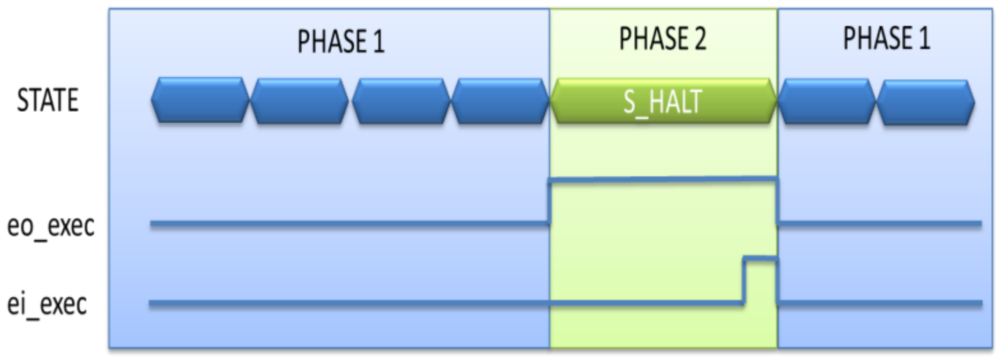


Figure A.2: Handshake sequencer signals for change of the operation mode

A.3 Instruction classification and formats

The sequencer instruction set includes the ALU-related and the control-flow instructions. These two main classes of instructions are distinguished depending on their purpose:

a) ***Flow control***: These instructions determine the program sequence and they are executed by the sequencer. Some of these instructions are transmitted to the PE array, and every PE has the logic to perform customized instructions in combination with the operation of the sequencer in order to increase the processing speed in the computation of the SNN algorithms.

b) ***Data processing***: These are ALU-related instructions to be executed by the PEs. These instructions are broadcasted to the PE array.

- Set of instructions of the sequencer – Flow control

Within each class, the instructions that require the same control line can be joined in groups, as shown in Table A.1. There are 9 groups of instructions related to the operations carried out in the ALU of the PE, and 10 group of instructions related to the sequencer operation in order to control de flow of the processing of the SNN algorithms.

Table A.1: Sequencer instruction group A/S: ALU-related/Sequencer

Instruction group	Class	Description
NOP	A	No operation
REGISTERS	A	ALU register operation
ARITHMETIC	A	ALU arithmetic operation
LOGIC	A	ALU logic operation
MOVEMENT	A	ALU register transfer
CONDITIONAL	A	Disable or enable of ALU registers
FLAGS	A	Flag set or reset
RANDON	A	Pseudorandom number operation
STORESP	A	Data load from the synaptic BRAM to the ALU registers
STOREPS	A	Data store from the ALU registers to the synaptic BRAM
STOREB	A	Data transfer from the buffer to the CPU monitor through the Ethernet bus
LOOPS	S	Start loop. Counter setting of the synapses

LOOPN	S	Start loop. Counter setting of the virtual layers
ENDL	S	End loop. Counter decreasing for LOOPS and LOOPN
GOTO	S	Unconditional jump instruction
GOTONL	S	Conditional jump instruction.
RET	S	Return instruction
HALT	S	Halt instruction
SPKDIS	S	SPKDIS instruction for the AER module
READMP	S	Read memory pointer from the BRAM instruction
RST_SEQ	S	Reset sequencer

Almost all the instructions executed by the sequencer require a single cycle except for the multiplication operation which requires 2 cycles. In certain instructions like the GOTO, RET, LOOPS, LOOPN, ENDL and READMP the pipeline has to be broken as the data to be distributed is also to be fetched from the BRAM. Only 2 clock cycles are required to resume the pipeline operation.

Figure A.3 shows the format of the instructions. These instructions have been defined according the function that performs. As it can be observed from Figure A.3, only three of the instructions require 2 byte, the remaining instructions require 11 bits.

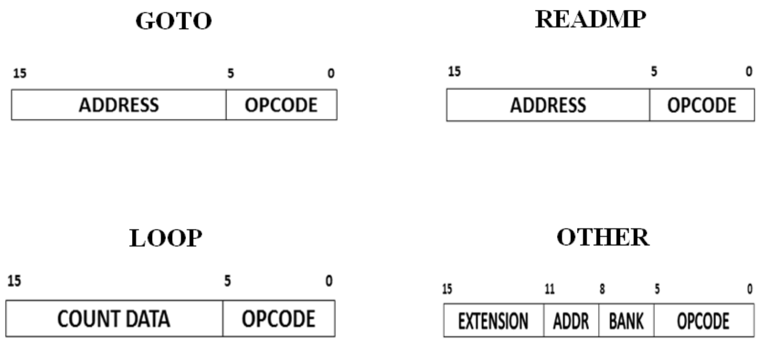


Figure A.3: Instruction formats. GOTO, READMP, LOOP and other instructions

A brief explanation about the instructions is provided below

- **GOTO:** the field of the address indicates the position of the BRAM to be read by the sequencer when is required. The length defined in this architecture is the 10 bits.
- **READMP:** the field of the address refers to the memory position of the constants which are allocated in the BRAM instructions.
- **LOOP:** the LOOP instruction includes LOOPS and LOOPN, the function of each loop will be explained in the following section.
- **OTHER:** The format of the default instructions linked with the sequencer only need the specification of the opcode, while the ALU instructions require the position of the bank of registers (from 1 to 7) to execute the instruction. The details of the format of the ALU instructions will be explained in the next section.

- *PE instructions – Data processing*

Some instructions of SNAVA were designed in order to increase the processing speed in the algorithm execution. As it was indicated above, almost all of the following instruction requires a single clock cycle to be executed, except for the multiplication. This instruction requires 2 clock cycles. Table **A.2** shows the instructions which are processed by the ALU.

- **NOP:** it does not perform any operation.
- **LLFSR reg:** loads in the selected active register the 16 MSB of the 64-bit Galois Pseudo random number generator
- **LOADSP:** loads the synaptic parameters of the current virtual layer from the synaptic BRAM to the active register bank.
- **STOREB:** the monitor buffer contains the values of the register accumulator and the register 1 of each PE. All the monitor buffers of the array are read by the Ethernet user side when the sequencer executes the STOREB instruction. The STOREB instruction is used only when the user wants to send parameters to be displayed on the monitor. These parameters are send to an external CPU (external interface) via Ethernet.
- **STORESP:** saves the synaptic parameters of the current neuron from the active register bank to the synaptic BRAM.
- **RST reg:** resets to 0 the content of the selected active register
- **SET reg:** sets to 1 the content of the selected active register
- **SHLN n:** operates a left shift of n positions on the accumulator. The last shifted bit is saved in the carry out register
- **SHRN n:** operates a right shift of n positions on the accumulator. The last shifted bit is saved in the carry out register

Annexure A Set of instructions of SNAVA

- **RTL:** operates a left shift of 1 position on the accumulator. The last shifted bit is saved in the carry out register
- **RTR:** operates a right shift of 1 position on the accumulator. The last shifted bit is saved in the carry out register
- **INC:** increment 1 to the accumulator.
- **DEC:** subtracts 1 to the accumulator.
- **NEG reg:** copy in the accumulator the content of selected register, changing its sign.
- **ADD reg:** sums the selected register with the accumulator. The result is stored in the accumulator.
- **SUB reg:** subtracts the value contained in the selected register from the value contained in the accumulator. The result is stored in the accumulator.
- **MUL reg:** multiplies the selected register with the accumulator. The 16 bit result out from the actual 32 bit result is stored in the accumulator. The result is represented by unsigned format.
- **UNMUL reg:** multiplies the selected register with the accumulator. The 16 bit saturated and signed result is stored in the accumulator.
- **AND reg:** performs a logical AND between the selected register and the accumulator. The result is stored in the accumulator.
- **OR reg:** performs a logical OR between the selected register and the accumulator. The result is stored in the accumulator.
- **INV reg:** performs a logical NOT on the selected register and the stores the result in the accumulator.
- **XOR reg:** performs a logical XOR between the selected register and the accumulator. The result is stored in the accumulator.
- **MOVA reg:** moves the content of the selected register in the accumulator.
- **MOVR reg:** moves the content of the accumulator in the selected register.
- **SWAPS reg, n:** swaps the contents of the selected shadow register and the corresponding active register. This instruction has two parameters:
    - reg: is the number of register to swap with its corresponding active register.
    - n: is the number of register bank from which it is wanted to select the register (reg) to swap.
- **FREEZEC:** disables the registers of the ALU if the carry out is 1.
- **FREEZENC:** disables the registers of the ALU if the carry out is 0.

Annexure A Set of instructions of SNAVA

- **FREEZEZ:** disables the registers of the ALU if the zero flag signal is 1.
- **FREEZENZ:** disables the registers of the ALU if the zero flag signal is 0.
- **SETZ:** sets to 1 the zero flag.
- **SETC:** sets to 1 the carry flag (carry out).
- **CLRZ:** clear to 0 the carry flag (carry out).
- **CLRC:** clear to 0 the carry flag (carry out).
- **RANDON:** enables the LFSR. The LFSR register is the source of the LLFSR operation.
- **RANDON1:** enables the LFSR. The LFSR register is the source of the LLFSR operation.
- **RANDOFF:** disables the LFSR.

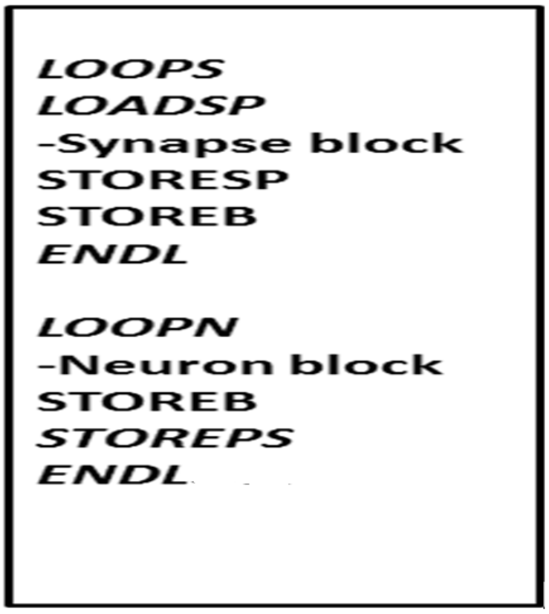
Table A.2: CPE Instruction with opcode

Instruction	Group	Format	Opcode	Description
NOP	NOP	NOP	0	No operation
LDALL	LOADALL	LDALL reg	1	reg <= BRAM sequencer(constants)
LLFSR	LLFSR	LLFSR reg	10	reg <= LFSR register (63 downto 48)
LOADSP	LOADSP	LOADSP	11	reg <= BRAM & spike_register (synapse parameters)
STOREB	STOREB	STOREB	100	Monitor BUFFER <= acc
STORESP	STORESP	STORESP	101	BRAM <= reg
STOREPS	STOREPS	STOREPS	110	AER_FIFO <= pre-synaptic (Si)
RST	REGISTERS	RST reg	111	reg <= (others=>'0')
SET	REGISTERS	SET reg	1000	reg <= (others=>'1')
SHLN	REGISTERS	SHLN n	1001	ACC <= ACC (n) <<, (1 < n < 8), (n = number of positions)
SHRN	REGISTERS	SHRN n	1010	ACC <= ACC (n) >>, (1 < n < 8), (n = number of positions)
RTL	REGISTERS	RTL	1011	ACC <= ACC <<, carry = ACC(msb)
RTR	REGISTERS	RTR	1100	ACC <= ACC >>, carry = ACC(lsb)
INC	REGISTERS	INC	1101	ACC <= ACC + 1
DEC	REGISTERS	DEC	1110	ACC <= ACC – 1
NEG	ARITHMETIC	NEG reg	1111	ACC <= 0 – reg

ADD	ARITHMETIC	ADD reg	10000	ACC <= ACC + reg (Saturation)
SUB	ARITHMETIC	SUB reg	10001	ACC <= ACC – reg (Saturation)
MUL	ARITHMETIC	MUL reg	10010	ACC <= ACC * reg (Saturation)
UNMUL	ARITHMETIC	UNMUL reg	10011	ACC <= ACC * reg (unsigned)
AND	LOGIC	AND reg	10100	ACC <= ACC AND reg
OR	LOGIC	OR reg	10101	ACC <= ACC OR reg
INV	LOGIC	INV reg	10110	ACC <= INV OR reg
XOR	LOGIC	XOR reg	10111	ACC <= ACC XOR reg
MOVA	MOVEMENT	MOVA reg	11000	ACC <= reg
MOVR	MOVEMENT	MOVR reg	11001	reg <= ACC
SWAPS	MOVEMENT	SWAPS reg, n	11010	reg ↔ shadow_reg, 1<n<7, n = number of shadow banks levels
FREEZEC	CONDITIONAL	FREEZEC	100001	Disable the registers of the ALUs if C=1
FREEZENC	CONDITIONAL	FREEZENC	100010	Disable the registers of the ALUs if C=0
FREEZEZ	CONDITIONAL	FREEZEZ	100011	Disable the registers of the ALUs if Z=1
FREEZENZ	CONDITIONAL	FREEZENZ	100100	Disable the registers of the ALUs if Z=0
UNFREEZE	CONDITIONAL	UNFREEZE	100101	Enables the registers of the ALUs
SETZ	FLAGS	SETZ	100111	Sets the zero flag: Z <= 1
SETC	FLAGS	SETC	101000	Sets the carry flag: C <= 1
CLRZ	FLAGS	CLRZ	101001	Clears the zero flag: Z <= 0
CLRC	FLAGS	CLRC	101010	Clears the zero flag: C <= 0
RANDON	RANDON	RANDON	101011	random_en <= 1; LFSR becomes source register for LLFSR
RANDON1	RANDON	RANDON	101100	random_en <= 1; LFSR_STEP<=1; LFSR becomes source register for LLFSR
RANDOFF	RANDON	RANDON	101101	random_en <= 0; LFSR_STEP <=0; LFSR disabled

A.4 Algorithm structure

Every SNN algorithm to be implemented in SNAVA, it must contain in its code the neuronal LOOP (LOOPN) and the synaptic LOOP (LOOPS). The LOOPS and LOOPN are the two instructions that are executed by the ALU and the sequencer in order to perform synaptic parameters and neural parameters respectively. Therefore, the synapses and virtual neurons are carried out serially in SNAVA. Figure A.4 shows a generic code structure of SNN algorithms to be emulated in SNAVA. The details of each Loop and the instructions are provided in the following paragraphs.



```
LOOPS
LOADSP
-Synapse block
STORESP
STOREB
ENDL

LOOPN
-Neuron block
STOREB
STOREPS
ENDL
```

Figure A.4: Code structure of a typical SNN emulation in SNAVA

- *Structure of the synaptic loop*

As can be observed from Figure A.4, the LOOPS contain the instructions LOADSP and STORESP. These instructions were designed in order to load the synaptic parameters from the synaptic BRAM to the active registers by using a single clock cycle or vice versa.

- LOADSP & STORESP: these instructions load the synaptic parameters in the current virtual layer from the synaptic BRAM to the active register bank (LOADSP) and the reverse operation (STORESP). The synaptic BRAM is wired to the active register of every PEs as shown in Figure A.5.
- STOREB: the monitor buffer contains the values of the register accumulator and the register 1 of each PE. All the monitor buffers of the array are read by the Ethernet user side when the sequencer executes the STOREB instruction. The STOREB instruction is used only when the user wants to send parameters to be displayed on the monitor. These parameters are send to an external CPU (external interface) via Ethernet. Figure A.8 shows the structure of the monitor buffers which are allocated in each Processing Element and the interface with the Ethernet controller.

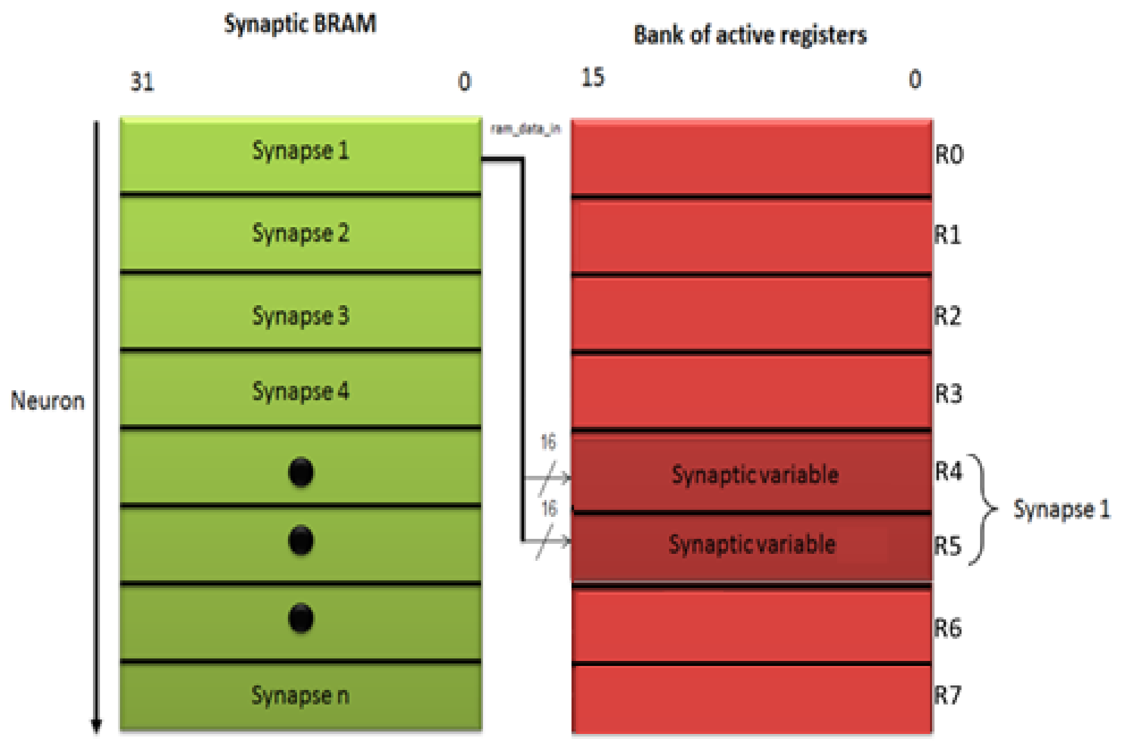


Figure A.5: Synaptic BRAM wired to active registers

- ***Structure of the neuronal loop***

The instructions contained inside of the LOOPN instruction are repeated in equal number of times as the number of virtual layers was defined. The execution of the LOOPN instruction by the sequencer indicates the index of the virtual neuron to the ALU. Therefore, the neural values, which are stored in the bank of shadow registers, can be transferred from the shadow registers to the active registers in a single clock cycle, or vice versa, by executing the SWAP instruction. The index of the loop indicates which bank of shadow register must be transferred in order to calculate the neural parameters by the ALU. The load operation is illustrated in Fig. A.6, while the store operation is illustrated in Fig. A.7.

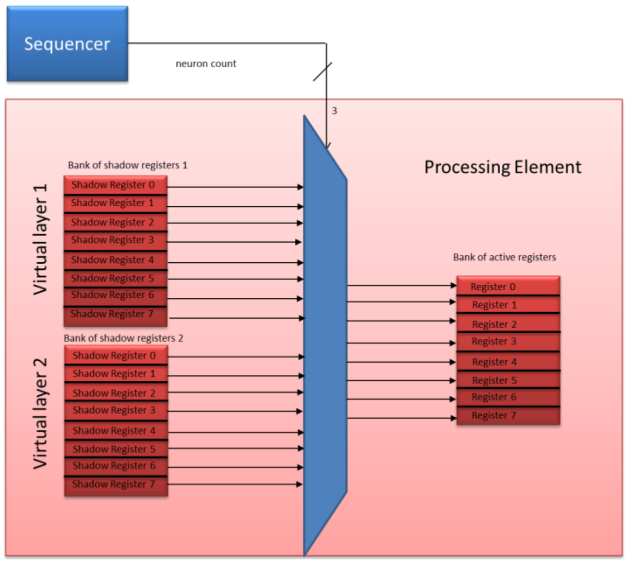


Figure A.6: Neural multiplexing – load operation

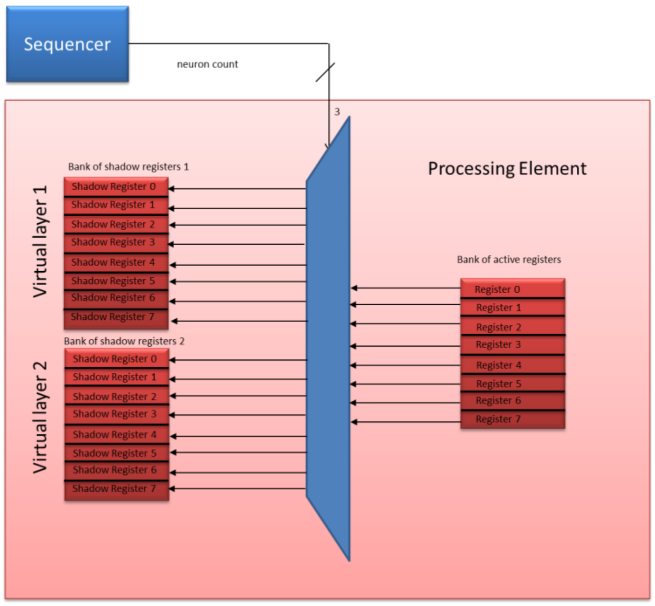


Figure A.7: Neural multiplexing – store operation

There are some fixed instructions that the neuronal loop should contain:

- **STOREB**: the monitor buffer contains the values of the register accumulator and the register 1 of each PE. All the monitor buffers of the array are read by the Ethernet user side when the sequencer executes the STOREB instruction. The STOREB instruction is used only when the user wants to send parameters to be displayed on the monitor. These parameters are send to an external CPU (external interface) via Ethernet. Figure A.8 shows the structure of the monitor buffers which are allocated in each Processing Element and the interface with the Ethernet controller.

- STOREPS: the sequencer enables the operation of the AER address Generator (see Fig. A.10) in order to send the spikes, which are generated by each virtul layer, to the AER module by executing the instruction STOREPS, as shown in Fig. A.4. Every virtual layer, which corresponds to one neuron, is processed serially by means of the LOOPN cycle (see Fig. A.4). Therefore, the spikes are read by the address generator only when the STOREPS instruction is executed by the sequencer, and the neural parameters of one virtual layer have been calculated. The AER address generator generates the AER address for each virtual layer by taking into account the format that is illustrated in Fig. A.9. The AER address generator indicates the position of the neuron, which has fired, this position involves its position in the array of the PEs (row,colum) and its virtual layer (depth). The index of the virtual layer is provided by the sequencer to the AER address generator through the signal depth. The AER address generator write these spikes in the FIFO memory of the AER control unit, as shown in Fig. A.10.

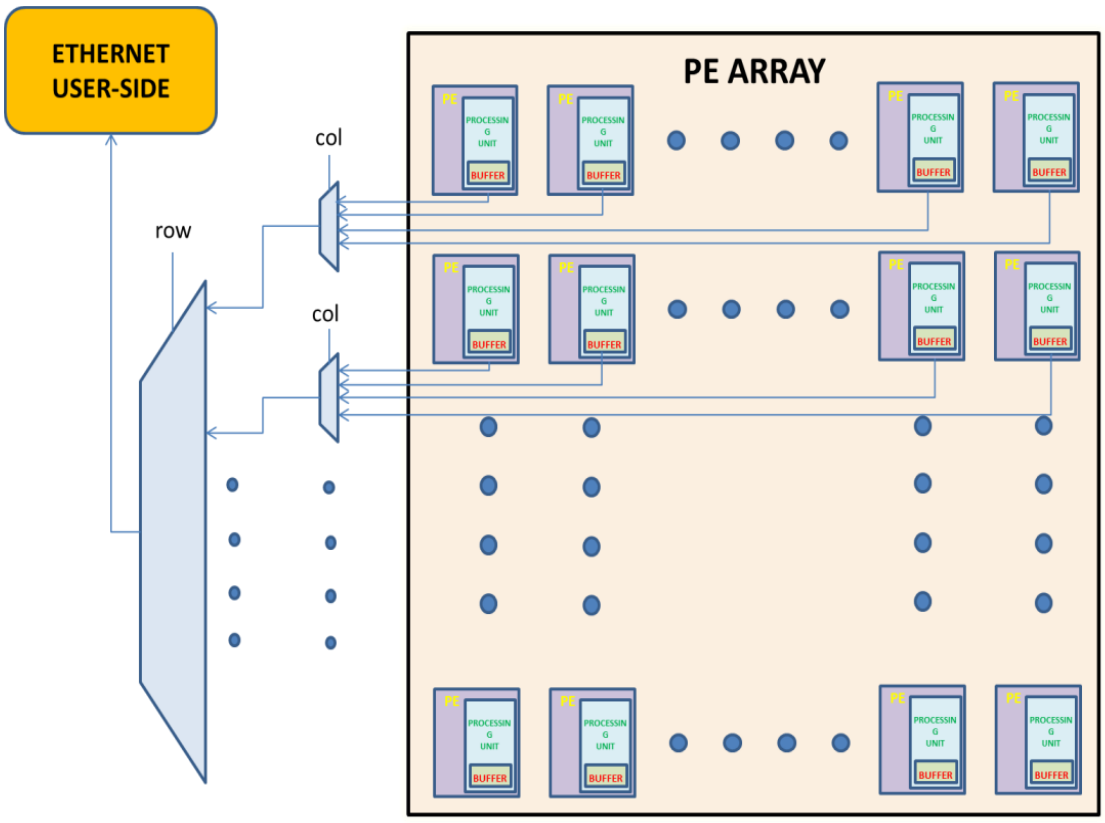


Figure A.8: Monitor buffer structure of SNAVA

ROW	COLUM	DEPTH
4 bits	**4 bits**	**3 bits**

Figure A.9: Format of the AER address per each neuron

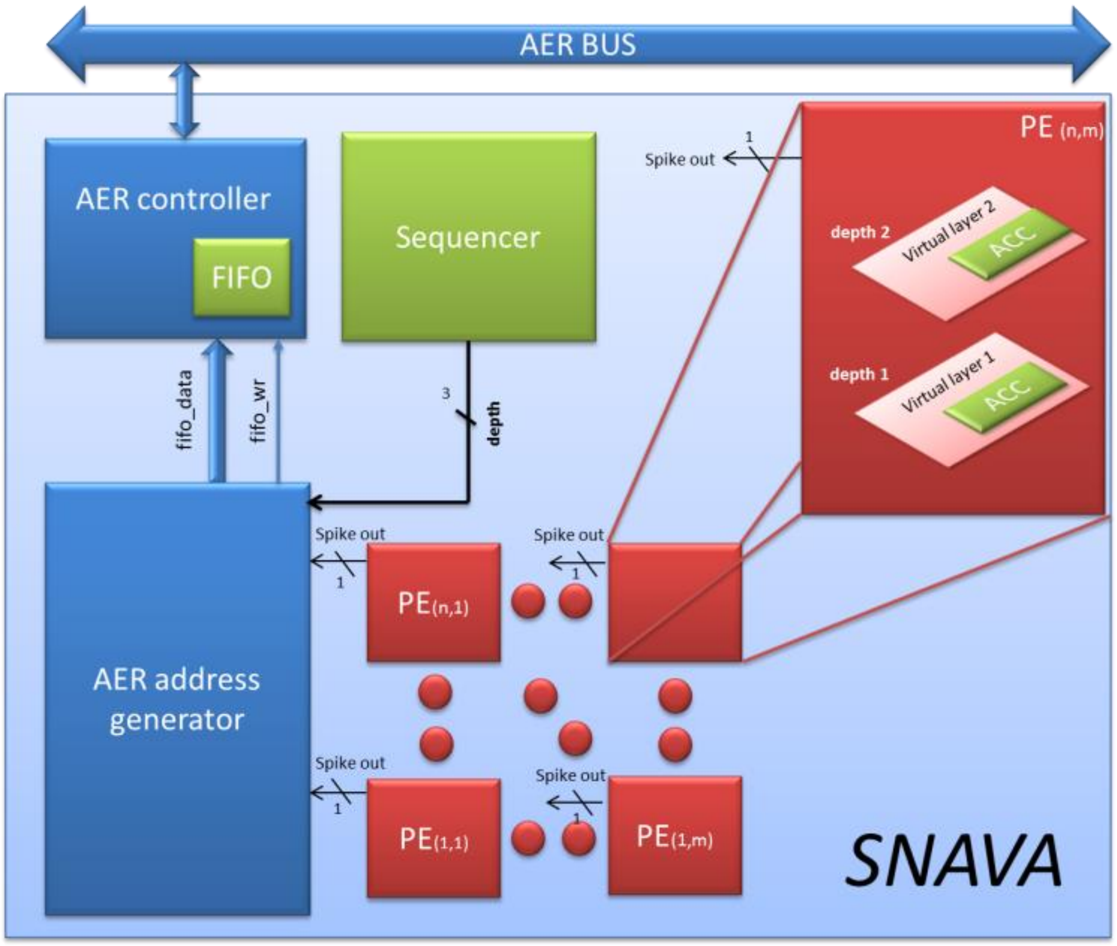


Figure A.10: Scheme of reading of the spikes per each PE by the AER address generator

A.5 Architecture functional details

The SNAVA architecture is defined as Harvard machine, so that the instructions and data can be simultaneously accessed from each memory which saves bus cycles, because of this has allowed implementing the pipeline of a single stage mechanism in the sequencer in order to increase the processing speed in performing the SNN algorithms. The data is stored in two hardware components which are: Blocks of Random Access Memory (BRAM) and bank of registers. The BRAMs store the parameters of the synapses, while the bank of registers is used to store the neural parameters. The BRAM of the sequencer is dedicated to store the instruction memory pointer IMEMP, constants of the algorithm and the instructions and as shown in Fig. A.11.

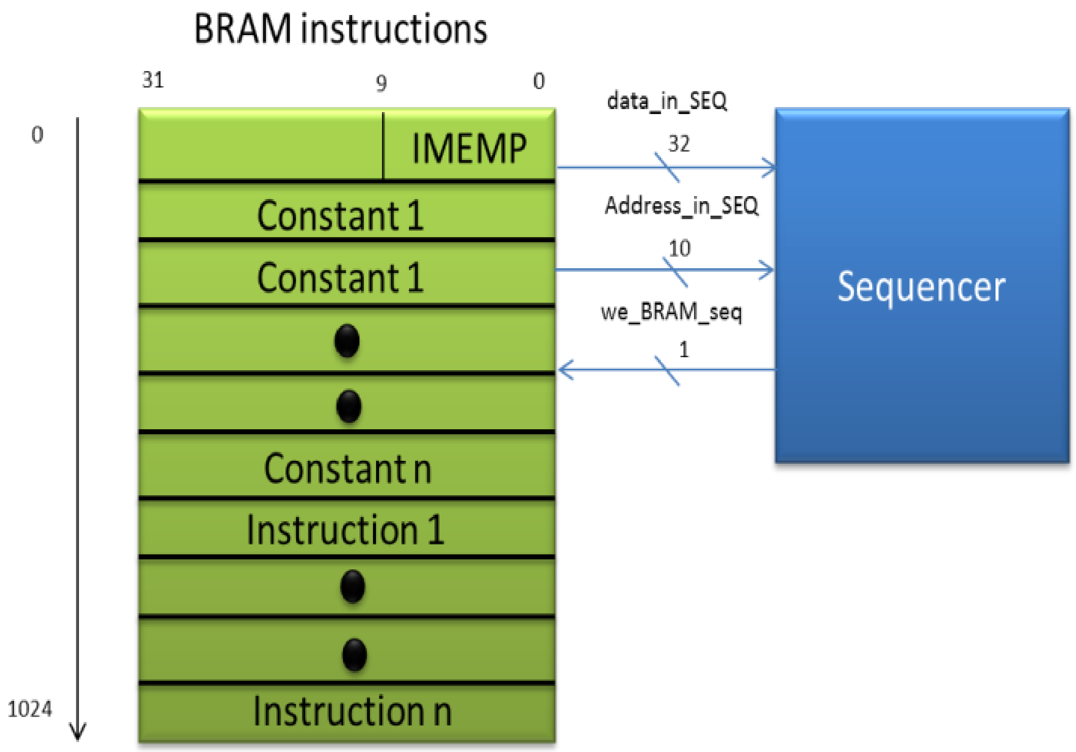


Figure A.11: Instructions and constants BRAM memory map

The sequencer module contains all the necessary logic to control the flow of the process in the simulation of the Spiking Neural Network algorithms. The data path of the sequencer consists of a program counter (PC) that point to the memory position of the next instruction. The length of the memory program is limited to 1024 bytes due to 10-bit addressing of the PC. This component also has the IMEMP register which indicates the position of the first instruction of the program. The DMEM register points the position of the constants to be loaded to the registers of the PE array. Three Last-In First-Out (LIFO) stacks are provided to keep track of iterations in the LOOPS and LOOPN instructions. The LIFO depth by default is 8, so eight nesting levels are supported. Nesting requires storing the PC contents, the current iteration number and the loop limit in the stacks depicted in Fig. A.12 (PC_LIFO, LOOP_LIFO, and LOOP_LIFO2 blocks).

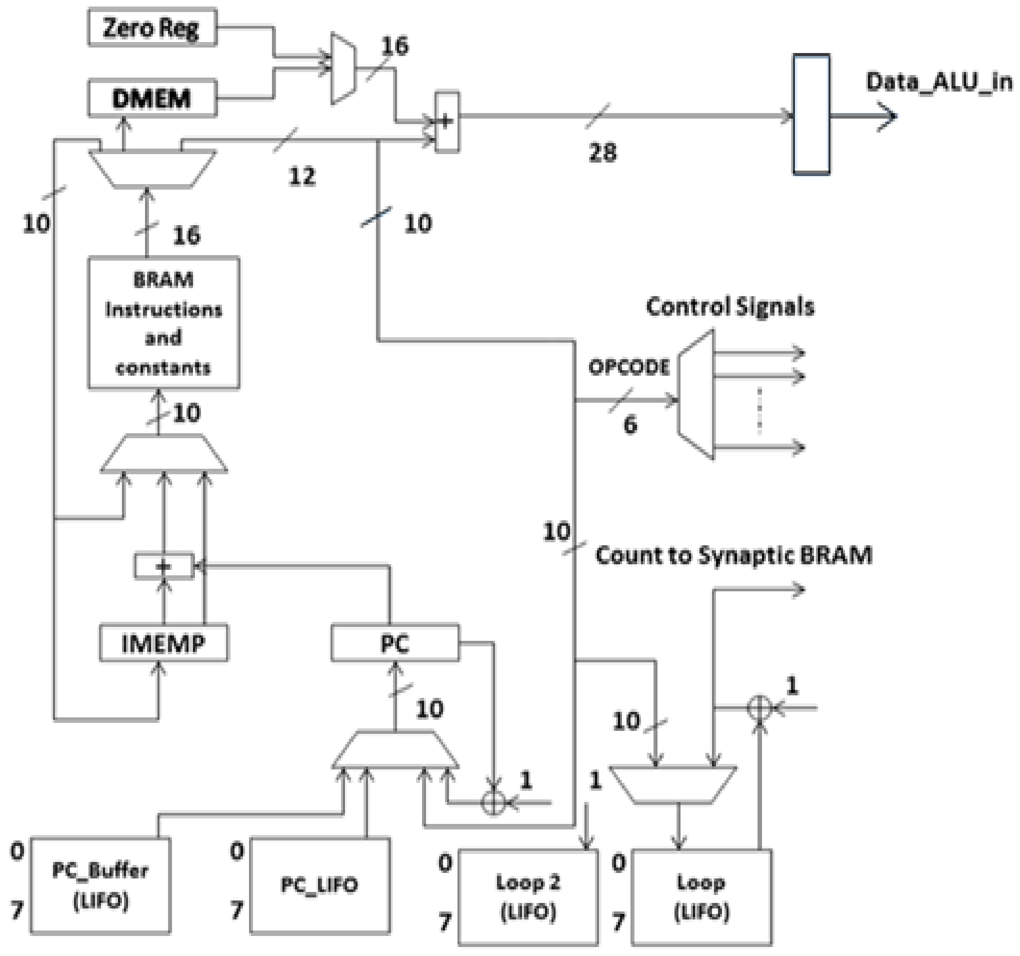


Figure A.12: Sequencer datapath

A Moore-type Finite State Machine was designed in VHDL in order to control the sequencer datapath. The FSM state diagram is shown in Fig. A.13. FSM works as follows: a synchronous reset set the reset state, so that all registers of the sequencer are set to 0. The initial state is reached after the reset state, where the first position of the BRAM instruction is read and loaded into the instruction address pointer IMEMP. Therefore, the program starts from the location pointed by the IMEMP. The next state is FECH. In this state the BRAM is read using the IMEMP as base address so the first instruction is read and decoded by the multiplexor called opcode. This multiplexor indicates the instruction. The PC is incremented to point the next instruction. This process is carried out simultaneously. While the instruction is decoded the next instruction is read by increment the PC counter. This mechanism allows the pipeline execution, only in the instruction needs two clock cycles the pipeline process must broke. These instructions involve the READMP, ENDL, GOTO and RET.

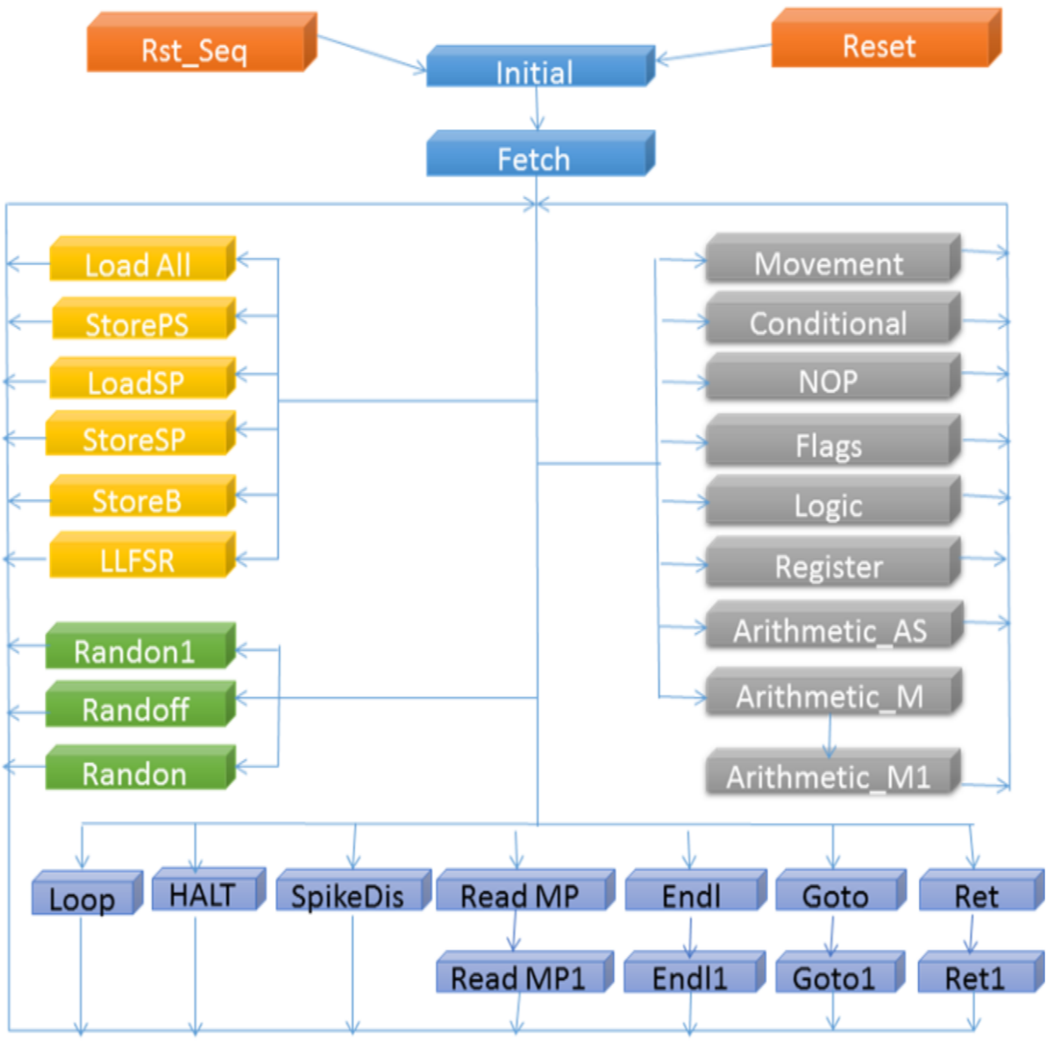


Figure A.13: Sequencer state machine

A.6 Sequencer register mapping

In this section the list of the internal sequencer registers has been provided. The mapping is shown in Table A.3.

Table A.3: Registers of the sequencer

REGISTER	CPU_ADDRESS(6:0)	DESCRIPTION	DATA IN
seq_enable_register	0110011	Loads enable flag from external CPU	seq_enable_flag <= CPU_data_in(0)

Annexure A Set of instructions of SNAVA

escape_flag	0000010	Loads escape flag from external CPU (to escape from S_HALT)	escape <= CPU_data_in(0)
Int_aux_flag	0101101	Loads int_aux flag from external CPU	int_aux <= CPU_data_in(0)
counter_N_L	0000101	Load in the no learning counter coming from the external CPU	counter_N_L<= CPU_data_in
simulation_steps_halt	0000011	Loads the number of simulation steps	simulation_step_halt<= CPU_data_in (15 downto 0)
PC_register	0000000	Lods the progrm counter from the external CPU	PC <= CPU_data_in(pc_length - 1 downto 0)
IMEM_P_register	0000100	Loads the pointer to BRAM instruction bank (IMEM_P) from the external CPU	IMEM_P <= CPU_data_in(pc_length - 1 downto 0)
DMEM_register	0000010	Loads the pointer to SRAM data banks (DMEM) from the external CPU	DMEM <= CPU_data_in
PC_BUFFER(0)	0011000	Loads PC_BUFFER(0) from the external CPU	PC_BUFFER(0) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(1)	0011010	Loads PC_BUFFER(1) from the external CPU	PC_BUFFER(1) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(2)	0011100	Loads PC_BUFFER(2) from the external CPU	PC_BUFFER(2) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(3)	0011110	Loads PC_BUFFER(3) from the external CPU	PC_BUFFER(3) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(4)	0100000	Loads PC_BUFFER(4) from the external CPU	PC_BUFFER(4) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(5)	0100010	Loads PC_BUFFER(5) from the external CPU	PC_BUFFER(5) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(6)	0100100	Loads PC_BUFFER(6) from the external CPU	PC_BUFFER(6) <= CPU_data_in (pc_length - 1 downto 0)
PC_BUFFER(7)	0100110	Loads PC_BUFFER(7) from the external CPU	PC_BUFFER(7) <= CPU_data_in (pc_length - 1 downto 0)
PC_LIFO(0)	0001000	Loads PC_LIFO(0) from the external CPU	PC_LIFO(0)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(1)	0001010	Loads PC_LIFO(1) from the external CPU	PC_LIFO(1)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(2)	0001100	Loads PC_LIFO(2) from the external CPU	PC_LIFO(2)<= CPU_data_in(pc_length - 1 downto 0)

PC_LIFO(3)	0001110	Loads PC_LIFO(3) from the external CPU	PC_LIFO(3)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(4)	0010000	Loads PC_LIFO(4) from the external CPU	PC_LIFO(4)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(5)	0010010	Loads PC_LIFO(5) from the external CPU	PC_LIFO(5)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(6)	0010100	Loads PC_LIFO(6) from the external CPU	PC_LIFO(6)<= CPU_data_in(pc_length - 1 downto 0)
PC_LIFO(7)	0010110	Loads PC_LIFO(7) from the external CPU	PC_LIFO(7)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(0)	1000000	Loads LOOP_LIFO(0) from the external CPU	LOOP_LIFO(0)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(1)	1000010	Loads LOOP_LIFO(1) from the external CPU	LOOP_LIFO(1)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(2)	1000100	Loads LOOP_LIFO(2) from the external CPU	LOOP_LIFO(2)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(3)	1000110	Loads LOOP_LIFO(3) from the external CPU	LOOP_LIFO(3)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(4)	1001000	Loads LOOP_LIFO(4) from the external CPU	LOOP_LIFO(4)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(5)	1001010	Loads LOOP_LIFO(5) from the external CPU	LOOP_LIFO(5)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(6)	1001100	Loads LOOP_LIFO(6) from the external CPU	LOOP_LIFO(6)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO(7)	1001110	Loads LOOP_LIFO(7) from the external CPU	LOOP_LIFO(7)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(0)	1010000	Loads LOOP_LIFO2(0) from the external CPU	LOOP_LIFO2(0)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(1)	1010010	Loads LOOP_LIFO2(1) from the external CPU	LOOP_LIFO2(1)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(2)	1010100	Loads LOOP_LIFO2(2) from the external CPU	LOOP_LIFO2(2)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(3)	1010110	Loads LOOP_LIFO2(3) from the external CPU	LOOP_LIFO2(3)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(4)	1011000	Loads LOOP_LIFO2(4) from the external CPU	LOOP_LIFO2(4)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(5)	1011010	Loads LOOP_LIFO2(5)	LOOP_LIFO2(5)<=

		from the external CPU	CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(6)	1011100	Loads LOOP_LIFO2(6) from the external CPU	LOOP_LIFO2(6)<= CPU_data_in(pc_length - 1 downto 0)
LOOP_LIFO2(7)	1011110	Loads LOOP_LIFO2(7) from the external CPU	LOOP_LIFO2(7)<= CPU_data_in(pc_length - 1 downto 0)

Annexure A Set of instructions of SNAVA

Register mapping in SNAVA for CPU access

B

B.1 Register mapping

B.1 Register mapping

This annexure presents the registers contained in every module in SNAVA in order to be accessed from the external CPU. The access control is the unit which in charge of providing access to the Processing Element array, sequencer, synaptic BRAMs, BRAM instructions, as shown in Fig. 5.1.

All components mentioned above are accessed by the user side of the Ethernet module. This interface establishes the communication between SNAVA and external CPU. The module is responsible for the initialization of the system and the process of debugging. There are two buses of 32 bits each one to interface to the User side protocol and the SNAVA in order to initialise the components of SNAVA or in the case of debugging the SNAVA system. Therefore, two words of 32 bits must send to SNAVA from the external CPU in order to access to SNAVA registers. The first 32 bits word indicates the address of the component to be accessed and the second word has the information to be loaded in the register or memory which is indicated by the address. The Tx and the Rx modules of Ethernet user side have an Ethernet bus of 8 bits to carried out the communication between SNAVA and the external CPU. Therefore, 4 clock cycles are required to receive the 4 bytes of words, which are a part of the address and another 4 bytes words, which are a part of the data. The Ethernet user side has a buffer in the TX module in order to store the data, which are received by the external CPU, and the Ethernet user side enables the data to be read by SNAVA only when these 8 bytes are have been received.

Table B.1: Format of the bus of the address

31	30	29	28	27-26	25-16	15-13	12-8	7-3	2-0
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Annexure B Register mapping in SNAVA for CPU access

The 32 bit word address is composed by 10 fields:

Field:	Description
1 = WR_SNAVA	1 indicates the process of write in the SNAVA components
2 = RD_SNAVA	1 indicates the process of read the SNAVA components
3 = BRAM ACCESS	1 enables the access to the BRAMs
4 = BRAM SELECT	0 indicates the access to the instruction BRAM 1 indicates the access to the synaptic BRAM
5 = EXT	These 3 bits are not used in the version of SNAVA. It can be used in the case of extending the system, in particular the address of instruction BRAM. The current length is 10, so that adding these three bits, the address can be increased to 13.
6 = ADDRESS	Address to be accessed in the case of BRAMs/ (18-16) level of shadow in case of CPE/ address of register in case of others
7 = LAYER	The module to be accessed as is illustrated:

LAYER			**Component**	
0	0	0	No used	FREE
0	0	1	REGISTERS	PE (Active and shadow registers)
0	1	0	LFSR	PE
0	1	1	REGISTERS	CPE -> CAM
1	0	0	REGISTERS	TX_MODULE
1	0	1	REGISTERS	SEQUENCER
1	1	0	REGISTERS	AER
1	1	1	REGISTERS	CONF. UNIT.

8 = COLUMN	Indicated the column address of the PE to be accessed

9 = ROW	Indicated the row address of the PE to be accessed
10 = ADDRESS REGISTERS	Address of the register to be accessed in PE (active register 0 to active register 7 or shadow register 0 to shadow register 7)

- REGISTER BANK

The register bank can be access under the following conditions:

- CPU_address(15 downto 13) = "001"
- CPU_address(7 downto 3) = row (from 1 to 16)
- CPU_address(12 downto 8) = col (from 1 to 16)
- The signal bank_address is composed by CPU_address(18 downto 16) & CPU_address(2 downto 0)

Table B.2: Register mapping in the PE

REGISTER	BANK_ADDRESS	DESCRIPTION	DATA IN
active_registers(0)	000000	Loads the active register from the external CPU	active_registers(0)<= CPU_data_in(15 downto 0)
active_registers(1)	000001	Loads the active register from the external CPU	active_registers(1)<= CPU_data_in(15 downto 0)
active_registers(2)	000010	Loads the active register from the external CPU	active_registers(2)<= CPU_data_in(15 downto 0)
active_registers(3)	000011	Loads the active register from the external CPU	active_registers(3)<= CPU_data_in(15 downto 0)
active_registers(4)	000100	Loads the active register from the external CPU	active_registers(4)<= CPU_data_in(15 downto 0)
active_registers(5)	000101	Loads the active register from the external CPU	active_registers(5)<= CPU_data_in(15 downto 0)
active_registers(6)	0000110	Loads the active register from the external CPU	active_registers(6)<= CPU_data_in(15 downto 0)

active_registers(7)	0000111	Loads the active register from the external CPU	active_registers(7)<= CPU_data_in(15 downto 0)
shadow_registers1(0)	0001000	Loads the shadow register1 from the external CPU	shadow_registers1(0)<= CPU_data_in(15 downto 0)
shadow_registers1(1)	0001001	Loads the shadow register1 from the external CPU	shadow_registers1(1)<= CPU_data_in(15 downto 0)
shadow_registers1(2)	0001010	Loads the shadow register1 from the external CPU	shadow_registers1(2)<= CPU_data_in(15 downto 0)
shadow_registers1(3)	0001011	Loads the shadow register1 from the external CPU	shadow_registers1(3)<= CPU_data_in(15 downto 0)
shadow_registers1(4)	0001100	Loads the shadow register1 from the external CPU	shadow_registers1(4)<= CPU_data_in(15 downto 0)
shadow_registers1(5)	0001101	Loads the shadow register1 from the external CPU	shadow_registers1(5)<= CPU_data_in(15 downto 0)
shadow_registers1(6)	0001110	Loads the shadow register1 from the external CPU	shadow_registers1(6)<= CPU_data_in(15 downto 0)
shadow_registers1(7)	0001111	Loads the shadow register1 from the external CPU	shadow_registers1(7)<= CPU_data_in(15 downto 0)

- LFSR REGISTERS

The the 64-bit Galois LFSR (pseudo-random number generator) can be access under the following conditions:

CPU_address(15 downto 13) = "010"

CPU_address(7 downto 3) = row (from 1 to 16)

CPU_address(12 downto 8) = col (from 1 to 16)

The address of the specific LFSR register to be accessed is placed in CPU_address (2 downto 0)

Table B.3: LFSR Register mapping in the PE

REGISTER	CPU_address(2 : 0)	DESCRIPTION	DATA IN
LFSR(63 :48)	000	Loads LFSR value (63:48) from exteral CPU	LFSR(63downto48)<= CPU_data_in(15 downto 0)
LFSR(47 :32)	001	Loads LFSR value	LFSR(47downto32)<=

		(47:32) from external CPU	CPU_data_in(15 downto 0)
LFSR(31:16)	010	Loads LFSR value (31:16) from external CPU	LFSR(31downto16)<= CPU_data_in(15 downto 0)
LFSR(15 :0)	011	Loads LFSR value (15:0) from external CPU	LFSR(15downto0)<= CPU_data_in(15 downto 0)
LFSR_en	100	Loads the LFSR enable from external CPU	LFSR_en <= CPU_data_in(0)
LFSR_step_flag	101	Loads from external CPU the LFSR flag that permits only LFSR updating when it is read	LFSR_step <= CPU_data_in(1)

- SYNAPTIC BRAM

The synaptic BRAM can be access under the following conditions:

The access to the PE BRAMs is possible by setting:

- CPU_address(29) = ‘1’
- CPU_address(28) = ‘1’ → synaptic BRAM or CPU_address(27) = ‘1’ → neuronal BRAM

- ETHERNET USER SIDE

The Ethernet user side tx module can be access under the following conditions:

CPU_address (15 downto 13) = "100",

Table B.4: Ethernet user side register mapping

REGISTER	CPU_ADDRESS(6:0)	DESCRIPTION	DATA IN
length _ena_bus	000001	Configure tx length of ethernet	length_ena<= CPU_data_in(1 downto 0)
overhead_count_max	000010	Set the value of the maximum delay between two consecutive packets	overhead_count_max<= CPU_data_in(13 downto 0)

- AER CONTROL

The AER control module can be access under the following conditions:

CPU_address(15 downto 13) = "110"

Table B.5: AER control register mapping

REGISTER	CPU_ADDRESS(6:0)	DESCRIPTION	DATA IN
chip id register	000001	Set the board ID	chip_id_reg <= CPU_data_in (CHIP_ID_WIDTH - 1 downto 0)
boards	000010	Set the number of the boards interconnected with the AER	boards <= CPU_data_in(6 downto 0)

- CONFIG UNIT

The config unit module can be access under the following conditions:

CPU_address (15 downto 13) = "111"

Table B.6: Config unit register mapping

REGISTER	CPU_address(7:3)	DATA IN	DATA OUT
config_done_int	000	config_done_int<= CPU_data_in(0)	CPU_data_out(0)<= config_done_int
clk_mode_register	001	clk_mode <= CPU_data_in(0)	CPU_data_out(0)<= clk_mode
dec_clk_counter	010	dec_clk_counter<= CPU_data_in	CPU_data_out<= dec_clk_counter
sna_size register	011	ONLY READ REGISTER	CPU_data_out <= SNA_size
inc_clk_counter(15: 0)	100	inc_clk_counter(15 downto 0) <= CPU_data_in	CPU_data_out<= inc_clk_counter(15 downto 0)
inc_clk_counter(31: 16)	101	inc_clk_counter(31 downto 16) <= CPU_data_in	CPU_data_out<= inc_clk_counter(31 downto 16)
inc_clk_counter(47: 32)	110	inc_clk_counter(47 downto 32) <= CPU_data_in	CPU_data_out<= inc_clk_counter (47 downto 32)
contr_reset	111	contr_reset_reg <= CPU_data_in (0)	CPU_data_out(0)<= contr_reset_reg

Assembly codes

This annexure presents the assembler codes of three SNN models: Iglesias and Villa model, Izhikevich model, and Leaky integrate-and-fire model. These models were implemented in Ubichip, and SNAVA in order to study the performance of these architectures. The Leaky integrate-and-fire model was used to carry out the application developed in this work (Chapter 6). The Leaky integrate-and-fire model was implemented in SNAVA+ in order to evaluate the performance of this architecture (see Chapter 5).

C.1 Assembler code of Iglesias and villa algorithm – Ubichip

```
AMAX="00000003"
DACT1="0000FFFA"
DACT2="0000012C"
DBACK="0000FAEE"
DMEM1="0000EF7D"
DMEM2="0000EF7D"
DSYN1="0000F9AE"
DSYN2="0000F9AE"
LMAX="00003FFF"

MMAX="00000666"
POT1="000003E8"
POT2="0000FFB0"
PROB="00001FFF"
SEED="A553A75A,A554A75A"
THETA1="0000F060"
THETA2="0000F060"
VREST1="0000E188"
VREST2="0000E188"
UNO="00000001"
MASC="00000003"
MASK1="0000E000"
MASK2="0000C000"

.CODE

; ---------------------------------- INIT   VARIABLES  -----------------------------
LDALL R4,PROB
MOVA R4
SETMP SEED
READMP
RANDINI
RANDON
```

```
LOAD R1
RANDOFF
AND R1
MOVR R1
SWAP R1      ;SR1 <-- activation probability
; ------------------------------------------------------------------------------------------

GOTO MAIN

; ****************************** PROCEDURES BEGIN ***************************
; ------------------------------  NEURON LOAD --------------------
.NEURON_LOAD
SWAP  R6
LOAD  R6,NEU-2                                   ;SR6 <-- Vi
SWAP  R6
SWAP  R0
LOAD  R0,NEU-3                                   ;SR0 <-- SUM_WEIGHTS
SWAP  R0
; ------------------------------ Neuron Type + Si ----------------------
LOAD  R2,NEU-1                                    ;R2  <-- Mi + Neuron Type + Si
MOVA  R2
LDALL R3,MASC
AND   R3
SWAP  R5
MOVR  R5                                         ;SR5 <-- Neuron Type + Si
SWAP  R5
; ------------------------------------- Mi ---------------------------------
MOVA  R2
SHR
SHR
SWAP  R4
MOVR  R4                                          ;SR4 <-- Mi
SWAP  R4
;------------------------------ Tref + exponential ---------------------
LDALL R3,MASK1   ;MASK1="0000E000"
SWAP  R5
MOVA  R5
SWAP  R5
SHR
SHR
FREEZENC
             LDALL R3,MASK2                                     ;MASK2="0000C000"
UNFREEZE
LOAD  R1,NEU-4                                   ;R1  <-- Tref + exponential
INV   R3                                         ;MASK1 --> 1FFF ; MASK2 --> 3FFF
AND   R1
MOVR  R7                                                     ;R7  <-- 1FFF
SWAP  R7                                         ;SR7 <-- exponential
MOVA  R1
AND   R3                                         ;MASK1 = E000 ; MASK2 = C000
MOVR  R7                                         ;R7  <-- Tref

RET

; ------------------------------ MEMBRANE VALUE ------------------------
.MEMBRANE_VALUE

RST  R1
RST  R2
SWAP R5                                          ;SR5 --> NEURON TYPE + Si
LDALL R3,DMEM1                                   ;R3  <-- DECAY DONATOR 1
LDALL R4,VREST1                                  ;R4  <-- Vres1
MOVA R5
SHR
SHR                                              ;IF NEURON TYPE = TYPE_II (CONDITIONAL LOAD)
FREEZENC
             LDALL R3,DMEM2                      ;R3  <-- DECAY DONATOR 2
             LDALL R4,VREST2                     ;R4  <-- Vres2
UNFREEZE
;---------------------- R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) -----------------
MOVA R5
SHR
```

```
FREEZEC              ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
          SWAP R6                         ;SR6 <-- Vi
          MOVA R6                                  ;R0  <-- Vi
          SUB R4                          ;R0  <-- Vi - Vres
          MOVR R2                         ;R2  <--(Vi(t)-Vres)
          GOTO DECAY                      ;R2  =  (Vi(t)-Vres), R3 = DECAY DONATOR (1 or 2)
                                          ;R2  <--(Vi(t)-Vres) * (Kmem)
UNFREEZE
MOVA R5
SHR
FREEZENC                                  ;IF (Si = 1) THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
   RST  R2                                ;R2  <-- ((0)*(Vi(t)-Vres)*(Kmem)
UNFREEZE
;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS ----------
LDALL R4,VREST1                           ;R4  <-- Vres1
MOVA R5
SHR
SHR
FREEZENC                                  ;IF NEURON TYPE = TYPE_II (CONDITIONAL LOAD)
          LDALL R4,VREST2                 ;R4  <-- Vres2
UNFREEZE
MOVA R4                                   ;R0  <-- Vres1 or Vres2
ADD  R2                                   ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
MOVR R2                                   ;R2  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
SWAP R0                                   ;R0  <-- SUM_WEIGHTS
ADD  R2                                   ;R0     <--    (Vres1    or    Vres2)    +    (1-Si(t))*(Vi(t)-Vres)*(Kmem)    +
SUM_WEIGHTS
MOVR R6                                   ;SR6 <-- Vi
SWAP R5                                   ;SR5 <-- NEURON TYPE + Si
RST R0                                    ;SUM_WEIGHTS = 0
SWAP R0                                   ;SR0 <-- SUM_WEIGHTS
RET
; --------------------------------------------------------------------------------

; -------------------------- SYNAPSE LOAD ----------------------------
.SYNAPSE_LOAD
; --------------------------------------------------------------------------------
; -------------------------------------- SP1 ----------------------------------
; --------------------------------------------------------------------------------
; ----------------------------  Mj + Synapse Type + Sj -----------------
LDALL R1,MASC
SETC
SETMP SYN-0                                ;LOOP INDEX
READMP 1
LOAD R2                                    ;R2  <-- Mj + Synapse Type + Sj
; -------------------------------  Synapse Type + Sj -----------------------
MOVA R2
AND  R1
MOVR R6                                   ;R6  <-- Synapse Type + Sj
; ------------------------------------ Mj ----------------------------------------
MOVA R2
SHR
SHR
MOVR R5                                   ;R5  <-- Mj
; --------------------------------------------------------------------------------
; -------------------------------------- SP2 -----------------------------------
; --------------------------------------------------------------------------------
; ----------------------------------- Lji + Aji ---------------------------------
LOAD R2                                   ;R2  <-- Lji + Aji
; --------------------------------------- Aji -----------------------------------
SWAP R3
MOVA R2
AND  R1
RST  R1
MOVR R3
SWAP R3                                   ;SR3  <-- Aji
; --------------------------------------- Lji -----------------------------------
MOVA R2
SWAP R2
SHR
SHR
MOVR R2                                   ;SR2  <-- Lji
```

```
SWAP R2
RET
; --------------------------------------------------------------------------------
; ----------------------------- SYNAPTIC WEIGHT --------------------
.SYNAPTIC_WEIGHT
RST R1
MOVA R6
SHR
FREEZENC                                          ;IF (Sj = 1) THEN R0 <-- wji = Sj * Aji * P

        LDALL R4,POT1                             ;R4  <--  POT1
        MOVA R6
        SHR
        SHR
        FREEZENC
                LDALL R4,POT2
        UNFREEZE
;----------------------------------------- Aji * P ---------------------------------
        MOVFS R3
        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                MOVR R1
        UNFREEZE

        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                ADD R4
                MOVR R1
        UNFREEZE

        MOVFS R3
        MOVA R3
        SHR
        FREEZENC
                SHR
                FREEZENC
                        MOVA R1
                        ADD  R4
                        MOVR R1
                UNFREEZE
        UNFREEZE
UNFREEZE
SWAP R0
ADD R1                                            ; SR0 <-- wji = Sj * Aji * P
SWAP R0
RET
; ---------------------------------------------------------------------------
; ---------------------------- REAL_VALUE_VARIABLE ----------------
.REAL_VALUE_VARIABLE
;-------------Lji(t+1)  = Lji(t) * Kact + Si(t) * Mj - Sj(t) * Mi(t)-------
LDALL R3,DACT1                                    ;R3  <-- DACT1
MOVA R6
SHR
SHR
FREEZENC
        LDALL R3,DACT2                            ;R3  <-- DACT2
UNFREEZE
MOVFS R2                                          ;R2  <-- Lji
;--------------------------------- Lji(t) * Kact 1 or 2 ------------------------
GOTO DECAY                                        ;R2 <-- Lji(t) * Kact 1 or 2
SWAP R5
MOVA R5
SWAP R5
SHR                                               ;R5  <-- Si
```

```
FREEZENC
            MOVA R2
            ADD R5                                       ;R0  <-- Mj
            MOVR R2                                      ;R2  <-- (Lji(t) * Kact) + (Si(t) * Mj)
UNFREEZE
MOVA R6
SHR                                                     ;R6  <-- Sj
FREEZENC
            MOVA R2
            SWAP R4
            SUB R4                                       ;R0  <-- Mi
            SWAP R4
            MOVR R2                                      ;R2  <-- (Lji(t) * Kact) + (Si(t) * Mj) - Sj(t) * Mi(t)
UNFREEZE
MOVTS R2                                                ;SR2 --> Lji
RET
; ------------------------------------------------------------------------------------
; ------------------------- ACTIVATION_VARIABLE --------------------
.ACTIVATION_VARIABLE
LDALL R1,UNO
SWAP R3
MOVA R3
FREEZEZ                                                  ; if (Aji =/ 0) then

            LDALL R0,LMAX                                ; R0 <--- Lmax
            SWAP R2                                      ; R2 <-- SR2  <-- Lji
            SUB  R2                                     ; Lmax - Lji
            SHL
            FREEZENC
                        MOVA R3                                      ; ACC <-- R3 <-- Aji
                        ADD R1                          ; Aji + 1
                        MOVR R3                         ; Aji --> SR3
                        LDALL R0,AMAX
                        SUB R3                          ; R0 <--  Amax - Aji
                        SHL
                        FREEZENC                        ; Aji - R1 = 0
                                    LDALL R3,AMAX
                        UNFREEZE
                        LDALL R0,LMAX    ; Lji=Lmax/2
                        SHR
                        MOVR R2
            UNFREEZE
                                                       ; else if (Lji < Lmin)
            MOVA R2                                                  ; Lji --> ACC, Lmin=0
            SHL
            FREEZENC                                    ; (Lji-Lmin)
                        MOVA R3
                        SUB R1                          ; Aji-1
                        MOVR R3                         ; Aji --> R3
                        LDALL R0,LMAX                   ; Lji=Lmax/2
                        SHR
                        MOVR R2
            UNFREEZE
UNFREEZE
MOVA R3
FREEZENZ                                                ;IF CONNECTION IS INACTIVE
            RST R2
UNFREEZE
SWAP R3
SWAP R2
RET
; ------------------------------------------------------------------------------------------
; --------------- MEMORY_OF_LAST_PRESYNAPTIC_SPIKE ---------
.MEMORY_OF_LAST_PRESYNAPTIC_SPIKE
; --------  Mj(t+1) = (Sj(t) * Mmax) + (1 - Sj(t)) * Mj(t) * Ksyn -----------
;--------------------- R2 <-- (1 - Sj(t)) * Mj(t) * Ksyn ---------------------------
LDALL R3,DSYN1                                          ; R3 <-- Ksyn1
MOVA R6                                                             ; R6 <-- Synapse Type + Sj
SHR
SHR
FREEZENC
            LDALL R3,DSYN2                              ; R3 <-- Ksyn2
```

```
UNFREEZE
MOVA R5                                         ; R5 <-- Mj
MOVR R2                               ; R2 <-- Mj
          GOTO DECAY                  ; R2 <-- (1 - Sj(t)) * Mj * Ksyn1 or Ksyn2
MOVA R6                                         ; R6 <-- Synapse Type + Sj
SHR
FREEZENC                              ;IF Sj(t) = 1 THEN R2 <-- Mmax
          LDALL R0,MMAX
          MOVR R2                     ; R2 <-- Mmax
UNFREEZE
MOVA R2
MOVR R5                                ; R5 <-- (Sj(t) * Mmax) + (1 - Sj(t)) * Mj(t) * Ksyn
RET
; ------------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE -------------------------
.SYNAPSE_SAVE

SETMP SYN-0                           ;LOAD LOOP INDEX!
READMP 1       ;READMPX
;  ---------------------------- MJ+SI+TYPE --------------------------
MOVA R6                                ;R6 <--- S type + Sj
SHR
SHL
MOVR R6
MOVA R5                                         ;<--MJ
SHL
SHL
ADD R6                                ;+TYPE+SJ
MOVR R3                               ;composed DATA
RST R0
SHR
STNC R3 ;SAVE DATA
; -----------------------------------  LJI+AJI  --------------------------------
SWAP R2
MOVA R2                                         ;<--LJI
SWAP R2
SHL
SHL
SWAP R3
ADD R3                                ;+AJI
SWAP R3
MOVR R3                               ;composed DATA
RST R0
SHR
STNC R3                               ;SAVE DATA
RET
; -----------------------------------------------------------------------------------
; ---------- MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE ---------
.MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE
LDALL R3,DSYN1                  ;TYPE=1
SWAP R5
MOVA R5
SHR
SHR                                   ;--> TYPE
FREEZENC
          LDALL R3,DSYN2              ;TYPE=2
UNFREEZE
SWAP R4                               ;R2=MI
MOVA R4
SWAP R4
MOVR R2
GOTO DECAY                            ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY
MOVA R5
SWAP R5
SHR                                   ;-->SI
FREEZENC
          LDALL R0,MMAX
          MOVR R2                     ;OVERWRITE DECAY RESULT
UNFREEZE
MOVA R2
SWAP R4
MOVR R4                               ;RES IN SR4
```

```
SWAP R4
RET
; --------------------------------------------------------------------------------
; ---------------------------- SPIKE UPDATE -----------------------------
.SPIKE_UPDATE
LDALL R3,THETA1                         ;R3  <-- THETA1 = "0000F060"
SWAP R5                                 ;SR5 <-- Neuron Type + Si
MOVA R5
SHR
SHR
FREEZENC
        LDALL R3,THETA2                 ;R3  <-- THETA2 = "0000F060"
UNFREEZE

MOVA R5
SHR
SHL
MOVR R5                                 ;R5  <-- Neuron Type + 0
SWAP R6                                 ;SR6 <--  Vi
MOVA R6                                 ;R0  <--  Vi
SUB  R3                                 ;R0  <--  Vi - (THETA1 or THETA2)
SWAP R6

FREEZENC
        MOVA R7                                 ;R0  <-- refractary period
        SHL
        FREEZEC
                MOVA  R5
                LDALL R3,UNO
                ADD   R3
                MOVR  R5                ;R5  <-- Neuron Type + 1
                SET   R7                ;R7  <-- activation of refractory time
        UNFREEZE
UNFREEZE
SWAP R5
RET
; ----------------------------------------------------------------------------
; ------------------------ BACKGROUND_ACTIVITY-------------
.BACKGROUND_ACTIVITY
SWAP R7                                 ; SR7 <-- exponential
MOVA R7
SWAP R7
MOVR R2                                 ; R2  <-- exponential
LDALL R3,DBACK                          ; R3  <-- DBACK = "0000FEB9"
        GOTO DECAY                      ; R2  <-- DBACK * exponential
SWAP R1                                         ; R1  <-- activation probability
LDALL R4,PROB                           ; R4  <-- PROB = "00001FFF"
MOVA R4                                         ; R0  <-- PROB
SUB R2                                  ; R0  <-- PROB - (DBACK * exponential)
RANDON
CLRC
SUB R1                                  ; (PROB - (DBACK * exponential)) - Activation probability
FREEZENC                                ; If ((PROB - (DBACK * exponential)) > Activation probability) then
        LOAD R1                         ; R1  <-- new activation probability
        RANDOFF
        MOVA R4                         ; R0  <-- PROB = "00001FFF"
        AND R1                          ; R0  <-- PROB = "00001FFF" AND new activation probability
        MOVR R1                         ;/ R1  <-- PROB = "00001FFF" AND new activation probability
        MOVA R4                                 ; R0  <-- PROB = "00001FFF"
        MOVR R2                         ;/ R2  <-- PROB = "00001FFF"
        MOVA R7                                 ; R0  <-- Tref
        SHL
        FREEZEC                         ; IF  ( C = 1 ) THEN Tref
                SWAP R5                 ; SR5 <-- Neuron Type + Si
                MOVA R5
                SHR
                SHL         ; SR5 <-- Neuron Type + Si = 0
                LDALL R3,UNO
                ADD R3        ; SR5 <-- Neuron Type + Si = 1
                MOVR R5
                SWAP R5
                SET R7                  ; R7  <-- activation of refractory time
        UNFREEZE
UNFREEZE
SWAP R1                                        ; SR1 <-- Activation probability
MOVA R2
SWAP R7
MOVR R7
SWAP R7                                        ; SR7 <-- exponential
RET
; ------------------------------------------------------------------------
; ------------------------------- REFRACTORY P -----------------
.REFRACTORY_P
MOVA R7
SHL                                            ; -1ms
MOVR R7
RET
; ------------------------------------------------------------------------
; ------------------------------- NEURON SAVE ------------------
.NEURON_SAVE
SWAP R4          ;R4  <-- Mi
SWAP R5                                ;R5  <-- Neuron Type + Si
SWAP R6
RST R3
MOVA R4
SHL
SHL
ADD R5
MOVR R3          ;R3  <-- Mi + Neuron Type + Si
;--------------------- INDIVIDUAL DATA STORE ---------------
RST R0
SHR
STNC R3,NEU-1                ;SRAM  <-- Mi + Neuron Type + Si
RST R0
SHR
STNC R6,NEU-2                ;SRAM  <-- Vi
SWAP R0
CLRC
STNC R0,NEU-3                ;SRAM  <-- SUM_WEIGHTS
SWAP R0
LDALL R3,              ;MASK1 = "0000E000"
MOVA R5
SWAP R5
SHR
SHR
FREEZENC
        LDALL R3,MASK2  ;MASK2 = "00008000"
UNFREEZE
MOVA R7                                        ;ACC   <-- Tref
AND R3
SWAP R7                                ;R7    <-- exponential
OR R7
SWAP R7
CLRC
STNC R0,NEU-4                ;SRAM  <-- Tref + exponential
RET
; ------------------------------------------------------------------------
;------------------------ENABLE SPIKES PROPAGATION----
.SPIKES_ENABLE
SWAP R5                      ; ACC <== Spikes
MOVA R5
SWAP R5
SETC
SETMP SYN-0                  ; Point to Sj
READMP
RET
;------------------------------------------------------------------------
; --------------------------- EXPONENTIAL DECAY ----------
.DECAY
RST R1
MOVA R2
MOVR R4
SHL
FREEZENC
```

```
        RST R0
        SUB R2
        MOVR R2
UNFREEZE
LOOP 15
        MOVA R2
        SHL
        MOVR R2
        FREEZENC
                MOVA R1
                ADD R3
                MOVR R1
        UNFREEZE
        MOVA R3
        SHR
        MOVR R3
ENDL
MOVA R1
SHR
MOVR R1
MOVA R4
SHL
FREEZENC
        RST R0
        SUB R1
        MOVR R1
UNFREEZE
MOVA R1
MOVR R2
RST R1
RET
; ---------------------------------------------------------------------------
; ******************************** PROCEDURES END *****************************

; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN
GOTO NEURON_LOAD
GOTO MEMBRANE_VALUE
LOOP synapses
        GOTO SYNAPSE_LOAD
        GOTO SYNAPTIC_WEIGHT
        GOTO REAL_VALUE_VARIABLE
        GOTO ACTIVATION_VARIABLE
        GOTO MEMORY_OF_LAST_PRESYNAPTIC_SPIKE
        GOTO SYNAPSE_SAVE
ENDL
GOTO MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE
GOTO SPIKE_UPDATE
GOTO BACKGROUND_ACTIVITY
GOTO REFRACTORY_P
GOTO NEURON_SAVE
GOTO SPIKES_ENABLE
STOP
HALT
GOTO MAIN
; **************************** MAIN PROGRAMME END *************************
```

C.2 Assembler code of Izhikevich algorithm – Ubichip

```
CTEIN="00001400"
CTEAE="0000000A"        ;a=  0.04   excitatory = E
CTEAE1="00000005"       ;0.02 * v
CTEAI1="00000033"       ;a=  0.2    inhibitory = E
CTEAI="0000001A"        ;a=  0.1    inhibitory = I
CTESE="00000600"        ;s=  6
CTESI="0000FB00"        ;s= -5
CTEB="00000033"         ;b=  0.2
CTECE="0000BF00"        ;c=-65
CTECI="0000F300"        ;c=-13
CTEDE="00000800"        ;d=  8
```

```
CTEDI="00000200"            ;d=  2
CTECU="00000400"            ;cte=4
CTECIN="00000500"           ;cte=5
CTEIN="00001400"            ;I= 20
CTEZE="00000000"            ;CTE=0
CTE30="00001E00"            ;Vmax = 30
CTE25="00000280"            ;2.5
CTE70="00004600"            ;70
CTE05="00000080"            ;0.5
CTE12="0000FECD"            ;-1.2
CUNO="00000001"             ;CTE=1
CTE095="000000F4"           ;0.95
CTE090="000000E7"           ;0.90
CTEBT="00008000"
UNO="00000001"
DOS="00000002"
TRES="00000003"
UNO1="00000001"
DOS2="00000002"
TRES3="00000003"
CUATRO4="00000004"
CARRY="00000004"
CCARRY="00000003"
CTE1000="000003E8"          ;CTE=1000
CTE128N="00008000"          ;CTE=-128
CTE127P="00007FFF"          ;CTE=127,9961
CTE20="00000014"             ;CTE=20
CTEP01="00000003"           ;CTE=0.01

.CODE
;--------------------------INIT SOME VARIABLES-----------------------------
SETMP SEED
READMP
RANDINI
;-------------------------------------------------------------------------
GOTO MAIN
;------------------------- INICIALIZATION I -------------------------------
.INICIALIZATION_I
RANDON1
LDALL R1
RANDOFF
RST R2
SWAP R2
RST R2
LDALL R2,MASCP                      ;MASCP="00000003"
MOVA R1
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
SHR
AND R2
MOVR R1
LOAD R3,ID
MOVA R3
SUB  R1
FREEZENZ
        LDALL R2,CTEIN              ;R2     <--    I = 20 ONLY FOR ONE PROCESSOR
        SWAP R2
UNFREEZE
RET
;--------------------------------------------------------------------------------
;--------------------------------------------------------------------------------
.SPIKE_UPDATE
  SWAP R1                                   ; It has to be deleted the previous spike
  MOVA R1
  SHR
  SHL
  MOVR R1
  SWAP R1                          ; It has deleted the previous spike
  SWAP R7
  MOVA R7                         ; ACC <-- v
  MOVR R4                                          ; R4  <-- v
  SWAP R7
  SHL
FREEZENC
      LDALL R4,CTESE            ; It has assigned a positive value under 30 because it has verified that is lower than 0
UNFREEZE
  MOVA   R4
  LDALL  R3,CTE30                ; R3  <-- 30
  SUB    R3                      ; v - 30
FREEZENC                         ; if v > 30
             SWAP  R1
             MOVA  R1
             SHR
             SHL
             LDALL R3,CUNO
             ADD   R3
             MOVR  R1
             SWAP  R1            ; SR1 <---  counter = 1000 + Carrypa + Carrypb + Neuron type + Si <-- 1
UNFREEZE
RET
;------------------------------------------------------------------------------------------------
; ---------------------- UPDATE MEMBRANE VALUE ----------------------------
.UPDATE_MEMBRANE_VALUE
SWAP R1                    ; R1 contains the current spike generated by research the threshold potential to be distribuited to anothers
neurons
RST R0
SHR
STNC R1,NEU-1                   ; for drawing the spikes over raster plot
SWAP R1
SWAP R1                          ;  SR1    <--  counter = 1000 + Carrypa + Carrypb + Neuron type + Si
MOVA R1
SWAP R1
SHR
SHR
FREEZENC                        ; NEURON TYPE EXCITATORY = 1
             LDALL  R2,CTEDE   ; Conditional store  d = 8 constant (excitatory)
UNFREEZE
FREEZEC                          ; NEURON TYPE INHIBITORY = 0
             LDALL  R2,CTEDI   ; Conditional store  d = 2 constant (inhibitory)
UNFREEZE
SWAP  R7
MOVA  R7                                      ; ACC <-- v
MOVR  R4                                      ; R4  <-- v
SWAP  R7
SHL
 FREEZENC
  LDALL R4,CTESE                  ; It has assigned a positive value under 30 because it has verified that is lower than 0
 UNFREEZE
MOVA  R4
LDALL R3,CTE30                                ; R3  <-- 30
SUB   R3                                      ; v - 30
FREEZENC                                      ; v = -65 , u = u + d, STDP = 0.1, only if v >= 30

             SWAP  R7                         ; R7  <-- v
             LDALL R7,CTECE                                ; v   <-- -65
             SWAP  R7                         ; SR7 <-- v = -65

             SWAP  R6
             MOVA  R6                         ; ACC <-- u
   ADD   R2                                   ; ACC <-- ACC + d
             MOVR  R6                         ; R6  <-- u + d
   SWAP  R6                                   ; SR6 <-- u = u + d

UNFREEZE
RET
;--------------------------------------------------------------------------------
;---------------------------------------------------------------------------------
.SPIKES_ENABLE
```

```
LOAD  R6,NEU-12          ; R6 <-- u
SWAP  R6               ; SWAP R6 <-- u
SWAP R1
MOVA R1
SWAP R1
SETC
SETMP SYN-0; Point to Sj, indicate to memory pointer the beginning of the distribution per each synapse of each neuron
READMP
RET
;--------------------------------------------------------------------------------
;**************************************************** Sj ************************************************
; ---------------------------- SYNAPSE LOAD ------------------------------
.SYNAPSE_LOAD1
SETMP SYN-0                                          ;LOAD LOOP INDEX!
READMP 1                                             ;READMPX
SWAP  R3              ;SR3  <-- s
SWAP  R4              ;SR4  <-- sd
MOVA  R5
SWAP  R0              ; respaldo de R5 en SR0 <-- Neuron ID + Synapse ID
RET
;------------------------------------------------------------------------------
;------------------------------------------------------------------------------
.SYNAPSE_SAVE1
SETMP SYN-0
READMP 1
MOVA R7
SHR
SHL
MOVR R7
RST R0
SHR
STNC R7 ;R7 <-- Sj
RET
;------------------------------------------------------------------------------
;----------------------------MEMBRANE VALUE---------------------
.MEMBRANE_VALUE
RST    R0
SWAP   R0
RST    R0
SWAP   R7                 ;SR7 ----> R7
MOVA   R7                 ;ACC <---- v <---- R7
SWAP   R7
MOVR   R2                 ;R2 <-- v
LDALL  R3,CTEAE            ;R3 <-- CTEAE="00000005" = 0.04
       GOTO MULTIPLICATION    ;R6 contains the result = 0.04*v
MOVA   R6
MOVR   R3                         ;R3 <-- 0.04*v
LDALL  R2,CTECIN           ;R2 <-- 5
       GOTO SUMA            ;R4 <-- 0.04*v + 5
MOVA   R4
SWAP   R0                 ;SR0 <-- 0.04*v + 5
SWAP   R7
MOVA   R7
SWAP   R7
MOVR   R2                 ;R2 <----- v
SWAP   R0
MOVR   R3                 ;R3 <----- 0.04*v + 5
SWAP   R0
       GOTO MULTIPLICATION    ;R6 <----- (0.04*v + 5)*v
LDALL  R2,CTE127P
MOVA   R6
MOVR   R3
       GOTO SUMA            ;R4 <-- (0.04*v + 5)*v + 140
MOVA   R4
MOVR   R3                         ;R3  <-- (0.04*v + 5)*v + 140
SWAP   R6                 ;SR6 <-- u
RST    R0
SUB    R6
MOVR   R2                 ;R2  <--- - u
SWAP   R6                 ;ACC <---- (0.04*v + 5)*v + 140 - u
       GOTO SUMA
MOVA    R4                ;R4 <---- (0.04*v + 5)*v + 140 - u
```

```
MOVR    R3                  ;R3 <---- (0.04*v + 5)*v + 140 - u
SWAP    R2
MOVA    R2
SWAP    R2
MOVR    R2                  ;R2 <---- I
       GOTO SUMA            ;R4 <---- (0.04*v + 5)*v + 140 - u + I
MOVA  R4
MOVR  R2                    ;R2 <---- (0.04*v + 5)*v + 140 - u + I
LDALL R3,CTE05              ;R3 <----  0.5
       GOTO MULTIPLICATION    ;R6 <----  0.5 * ((0.04*v + 5)*v + 140 - u + I)

MOVA   R6
MOVR   R2                          ;R2 <----  0.5 * ((0.04*v + 5)*v + 140 - u + I)
SWAP   R7
MOVA   R7
MOVR   R3
       GOTO   SUMA                ;R4  <---- v + 0.5 * ((0.04*v + 5)*v + 140 - u + I)
MOVA   R4
MOVR   R7                   ;R7 <---- v + 0.5 * ((0.04*v + 5)*v + 140 - u + I)
SWAP   R7
RET
;---------------------------- COMPUTATION OF RECOVERY VALUE ------------------------
.RECOVERY_VALUE
;----------------------------------u = u + a(0.2*v-u)-------------------------------
LDALL R2,CTEB               ;R2 <---- 0.2 = b
SWAP R7
MOVA R7
MOVR R3                     ;R3 <---- v
SWAP R7
   GOTO MULTIPLICATION       ;R6 <----- 0.2*v
MOVA R6
MOVR R2                     ;R2 <----- 0.2*v

SWAP   R6
RST    R0
SUB    R6
MOVR   R3                   ;R3  <--- - u
SWAP   R6
   GOTO SUMA                ;R4 <--- 0.2*v - u
MOVA R4
MOVR R3                ;R3 <--- 0.2*v - u
SWAP R1
MOVA R1
SWAP R1
SHR
SHR
FREEZENC               ; NEURON TYPE EXCITATORY = 1
              LDALL  R2,CTEAE1   ; Conditional store to a =  0.02 constant (excitatory)
UNFREEZE
FREEZEC                          ; NEURON TYPE INHIBITORY = 0
              LDALL  R2,CTEAI    ; Conditional store to a =  0.1 constant (inhibitory)
UNFREEZE
   GOTO MULTIPLICATION       ;R6 <----- a*(0.2*v - u)
MOVA R6
MOVR R2                    ;R2 <----  a*(0.2*v - u)

SWAP R6                    ;R6 <---  u
MOVA R6
SWAP R6
MOVR R3                    ;R3 <---  u
   GOTO SUMA               ;R4 <---- u + a*(0.2*v - u)
MOVA R4
MOVR R6
SWAP R6
RET
;--------------------------------------------------------------------------------
;--------------------------------------------------------------------------------
.NEURON_SAVE
SWAP R7        ;R7  <-- Vi
MOVA R7
MOVR R2
RST R0
```

```
SHR
STNC R2,NEU-2
SWAP R7
RET
;-----------------------------------------------------------------------------------
.SUMA
;##################################################################################
RST  R4
;------------------------------------- PRIMER BLOQUE ---------------------------------
;------------------------------- DOS NUMEROS POSITIVOS ------------------------------
MOVA R3
SHL
FREEZEC
  MOVA R2
  SHL
    FREEZEC
       MOVA R3
                    ADD  R2
                    MOVR R4          ;R4 CONTIENE EL VALOR DE LA SUMA
                    SHL
                     FREEZENC
                     LDALL R4,CTE127P
        UNFREEZE
    UNFREEZE
UNFREEZE
;-------------------------------------------------------------------------------------
;------------------------------------- SEGUNDO BLOQUE ---------------------------------
;-------------------------------------- R2 ES NEGATIVO  -------------------------------
MOVA R3 ;B
SHL
 FREEZEC
    MOVA R2 ;A  NEGATIVO
    SHL
    FREEZENC
      RST     R0
      SUB     R2
      MOVR    R2
      MOVA    R2
                   SUB     R3
      FREEZENC
                     MOVR    R2
                                   RST     R0
                                   SUB     R2
                     MOVR    R4
      UNFREEZE
        MOVA    R3
                     SUB     R2
      FREEZENC
                                   MOVR    R4
      UNFREEZE
    UNFREEZE
 UNFREEZE
;-------------------------------------------------------------------------------------
;------------------------------------- TERCER BLOQUE ---------------------------------
;-------------------------------------- R3 ES NEGATIVO  -------------------------------
MOVA R3 ;B NEGATIVO
SHL
 FREEZENC
    MOVA R2 ;A
    SHL
    FREEZEC
      RST     R0
      SUB     R3
      MOVR    R3       ;VALOR POSITIVO DE R3

      MOVA    R2
                   SUB     R3
      FREEZENC
                     MOVR  R4
      UNFREEZE
        MOVA    R3
                     SUB     R2
```

```
     FREEZENC
       MOVR    R2
                              RST     R0
                              SUB     R2
                              MOVR    R4
     UNFREEZE
   UNFREEZE
UNFREEZE
;-------------------------------------------------------------------------------------
;------------------------------------  CUARTO BLOQUE ---------------------------------
;------------------------------------- AMBOS NEGATIVOS  ------------------------------
   MOVA R3
   SHL
   FREEZENC
       MOVA R2
       SHL
       FREEZENC
           RST     R0
           SUB     R3
           MOVR    R3       ;VALOR POSITIVO DE R3
           RST     R0
           SUB     R2
           MOVR    R2       ;VALOR POSITIVO DE R2
           MOVA    R3
                  ADD     R2
                                          MOVR    R3
                            RST     R0
                                          SUB     R3
                                          MOVR    R4       ;R4 CONTIENE EL VALOR DE LA SUMA TOTAL
                  MOVA    R3       ;R3 CONTIENE EL RESPALDO VALOR DE LA SUMA
                  SHL
                      FREEZENC
                          LDALL R4,CTE128N
               UNFREEZE
       UNFREEZE
                 UNFREEZE
RET
;-----------------------------------------------------------------------------------
;###################################################################################
.MULTIPLICATION
;------------------------------DETECCION DEL SIGNO DE R2-----------------------------
RST  R6
MOVA    R2
MOVR    R4                ;R4 MANTIENE EL VALOR DE M1

SHL
FREEZENC
RST     R0
SUB     R2
MOVR    R2
UNFREEZE
;------------------------------DETECCION DEL SIGNO DE R3-----------------------------
MOVA    R3
MOVR    R7                ;R7 MANTIENE EL VALOR DE M2

SHL
FREEZENC
RST     R0
SUB     R3
MOVR    R3
UNFREEZE
;-------------------------------Calculo de la parte baja----------------------------
       MOVA R2
       MOVR R5
   LOOP 15
      MOVA R3
      SHR
      MOVR R3
       FREEZENC
             CLRC
             MOVA R6
             ADD  R5
```

```
            MOVR R6
            FREEZENC
              SWAP R1
              MOVA R1
              LDALL R2,CARRY
              OR R2
              MOVR R1
              SWAP R1
            UNFREEZE
       UNFREEZE
      MOVA R5
      SHL
      MOVR R5            ;R5 <-- M1 * M2
   ENDL
RST R0
RST R1
RST R2
RST R3
RST R5
;------------------------------DETECCION DEL SIGNO DE R2----------------------------
MOVA    R4
MOVR    R2
SHL
FREEZENC
RST     R0
SUB     R4
MOVR    R2
UNFREEZE
;------------------------------DETECCION DEL SIGNO DE R3-----------------------------
MOVA    R7
MOVR    R3
SHL
FREEZENC
RST     R0
SUB     R7
MOVR    R3
UNFREEZE
MOVA    R7
MOVR    R1       ;AHORA EL VALOR DE R7 ESTA EN R
;----------------------------Calculo de la parte alta--------------------
      RST  R7
      MOVA R2
      MOVR R5

   LOOP 15
      MOVA R3
      SHL
      MOVR R3
                FREEZENC
           MOVA R7
           ADD  R5
           MOVR R7
      UNFREEZE
                MOVA R5
      SHR
      MOVR R5
   ENDL
;--------------------------------------------------------------------------
SWAP R1
MOVA R1
SWAP R1
SHR
SHR
ADD R7
SHR
MOVR R7
MOVA R6
SHR
SHR
SHR
SHR
SHR
```

```
SHR
SHR
SHR
MOVR R6         ;MOVIMIENTO HACIA LA DERECHA PARA OBTENER EL VALOR DE LA PARTE FRACCIONARIA
MOVA R7
SHL
SHL
SHL
SHL
SHL
SHL
SHL
SHL
MOVR R7         ;MOVIMIENTO HACIA LA DERECHA PARA OBTENER EL VALOR DE LA PARTE ENTERA
MOVA R7
XOR  R6        ;UNION DE LAS DOS PARTES
MOVR R6
MOVR R5            ; R5 TIENE EL VALOR DE R6

;------------------------CALCULO FINAL DEL SIGNO DEL PRODUCTO---------------
MOVA    R1
LDALL   R2,CTEBT
AND     R2
MOVR    R2
MOVA    R4
LDALL   R3,CTEBT
AND     R3
MOVR    R3
MOVA    R2
XOR     R3
MOVR    R3            ; R3 CONTIENE EL BIT DE SIGNO DEL PRODUCTO FINAL
SHL                  ; IF  + or - the final sign is negative
FREEZENC
   RST  R0
   SUB  R6
   MOVR R6
   MOVA R5
   SHL
   FREEZENC
    LDALL R6,CTE128N     ;SE CARGA EL MAXIMO NUMERO NEGATIVO REPRESENTADO EN 7 BITS DE LA PARTE
ENTERA
   UNFREEZE
UNFREEZE
MOVA     R3
SHL
FREEZEC
    MOVA R5
    SHL
    FREEZENC
        LDALL R6,CTE127P     ;SE CARGA EL MAXIMO NUMERO POSITIVO REPRESENTADO EN 7 BITS DE LA PARTE
ENTERA
  UNFREEZE
UNFREEZE
SWAP R1
MOVA R1
LDALL R2,CCARRY
AND R2
MOVR R1
SWAP R1
RET
;############################################################################

; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN

                GOTO INICIALIZATION_I
                GOTO SPIKE_UPDATE              ;OUT SPIKE Si
                GOTO UPDATE_MEMBRANE_VALUE
                GOTO SPIKES_ENABLE
                STOP               ;AER/CAM UPDATE OF SPIKES
                LOOP synapses                  ; spikes Sj
                    GOTO SYNAPSE_LOAD1
                    GOTO SYNAPTIC_WEIGHT_PRE
                    GOTO SYNAPSE_SAVE1
    ENDL
                GOTO MEMBRANE_VALUE
                GOTO MEMBRANE_VALUE
                GOTO RECOVERY_VALUE
                GOTO NEURON_SAVE

GOTO MAIN
; **************************** MAIN PROGRAMME END ***************************
```

C.3 Assembler code of Leaky integrate-and-fire algorithm – Ubichip

```
THETA1="0000F060"
THETA2="0000F060"
POT1="000003E8"
POT2="0000FFB0"
VREST1="0000E188"
VREST2="0000E188"
UNO="00000001"
MASC="00000003"
MASK1="0000E000"
MASK2="0000C000"

.CODE

GOTO MAIN

; ---------------------------------------------------------------------------
; ***************************** PROCEDURES BEGIN ***************************
; ------------------------------  NEURON LOAD --------------------
.NEURON_LOAD
SWAP  R6
LOAD  R6,NEU-2                  ;SR6 <-- Vi
SWAP  R6
SWAP  R0
LOAD  R0,NEU-3                  ;SR0 <-- SUM_WEIGHTS
SWAP  R0
; ------------------------------ Neuron Type + Si ----------------------
LOAD  R2,NEU-1      ;R2  <-- Mi + Neuron Type + Si
MOVA  R2
LDALL R3,MASC
AND   R3
SWAP  R5
MOVR  R5                          ;SR5 <-- Neuron Type + Si
SWAP  R5
; ------------------------------------- Mi ----------------------------------
MOVA  R2
SHR
SHR
SWAP  R4
MOVR  R4                                   ;SR4 <-- Mi
SWAP  R4
;------------------------------ Tref + exponential ----------------------
LDALL R3,MASK1      ;MASK1="0000E000"
SWAP  R5
MOVA  R5
SWAP  R5
SHR
SHR
FREEZENC
            LDALL R3,MASK2   ;MASK2="0000C000"
UNFREEZE
LOAD  R1,NEU-4                  ;R1  <-- Tref + exponential
INV   R3                         ;MASK1 --> 1FFF ; MASK2 --> 3FFF
AND   R1
```

```
MOVR  R7                          ;R7  <-- 1FFF
SWAP  R7                          ;SR7 <-- exponential
MOVA  R1
AND   R3                          ;MASK1 = E000 ; MASK2 = C000
MOVR  R7                          ;R7  <-- Tref
RET
; ---------------------------------------------------------------------------
; -------------------------- MEMBRANE VALUE --------------------
.MEMBRANE_VALUE
RST  R1
RST  R2
SWAP R5                             ;SR5 --> NEURON TYPE + Si
LDALL R3,DMEM1                ;R3  <-- DECAY DONATOR 1
LDALL R4,VREST1               ;R4  <-- Vres1
MOVA R5
SHR
SHR                     ;IF NEURON TYPE = TYPE_II (CONDITIONAL LOAD)
FREEZENC
              LDALL R3,DMEM2 ;R3  <-- DECAY DONATOR 2
              LDALL R4,VREST2 ;R4  <-- Vres2
UNFREEZE
;----------------------- R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) -----------------
MOVA R5
SHR
FREEZEC                 ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
              SWAP R6                         ;SR6 <-- Vi
              MOVA R6                                       ;R0  <-- Vi
              SUB R4                          ;R0  <-- Vi - Vres
              MOVR R2                         ;R2  <--(Vi(t)-Vres)
              GOTO DECAY                      ;R2  =  (Vi(t)-Vres), R3 = DECAY DONATOR (1 or 2)
                                              ;R2  <--(Vi(t)-Vres) * (Kmem)
UNFREEZE
MOVA R5
SHR
FREEZENC                ;IF (Si = 1) THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
   RST  R2              ;R2  <-- ((0)*(Vi(t)-Vres)*(Kmem)
UNFREEZE
;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS ----------
LDALL R4,VREST1               ;R4  <-- Vres1
MOVA R5
SHR
SHR
                              ;IF NEURON TYPE = TYPE_II (CONDITIONAL LOAD)
FREEZENC
              LDALL R4,VREST2 ;R4  <-- Vres2
UNFREEZE
MOVA R4                       ;R0  <-- Vres1 or Vres2
ADD  R2                       ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
MOVR R2                       ;R2  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
SWAP R0                       ;R0  <-- SUM_WEIGHTS
ADD  R2                       ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
MOVR R6                       ;R6  <-- Vi = (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
SWAP R6                       ;SR6 <-- Vi
SWAP R5                       ;SR5 <-- NEURON TYPE + Si
RST R0                        ;SUM_WEIGHTS = 0
SWAP R0                       ;SR0 <-- SUM_WEIGHTS
RET
; -----------------------------------------------------------------------------------------
; ---------------------------------- SYNAPSE LOAD -------------------------------
.SYNAPSE_LOAD
; ------------------------------------------------------------------------------
; -------------------------------------- SP1 -----------------------------------
; ------------------------------------------------------------------------------
; ----------------------------  Synapse Type + Sj -------------------------
LDALL R1,MASC
SETC
SETMP SYN-0                                             ;LOOP INDEX
READMP 1
LOAD R2                                       ;R2  <-- Synapse Type + Sj
; -------------------------------  Synapse Type + Sj ----------------------------
MOVA R2
AND  R1
```

```
MOVR R6                                ;R6  <-- Synapse Type + Sj
MOVA R2
SHR
SHR
MOVR R5
; ---------------------------------------------------------------------------------------------
; --------------------------------- SYNAPTIC WEIGHT ------------------------------
.SYNAPTIC_WEIGHT
RST R1
MOVA R6
SHR
FREEZENC                    ;IF (Sj = 1) THEN R0 <-- wji = Sj * Aji * P
        LDALL R4,POT1       ;R4  <--  POT1
        MOVA R6
        SHR
        SHR
        FREEZENC
                LDALL R4,POT2
        UNFREEZE
;----------------------------------------- Aji * P ---------------------------------
        MOVFS R3
        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                MOVR R1
        UNFREEZE
        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                ADD R4
                MOVR R1
        UNFREEZE
        MOVFS R3
        MOVA R3
        SHR
        FREEZENC
                SHR
                FREEZENC
                        MOVA R1
                        ADD  R4
                        MOVR R1
                UNFREEZE
        UNFREEZE
UNFREEZE
SWAP R0
ADD R1                                 ; SR0 <-- wji = Sj * Aji * P
SWAP R0
RET
; ---------------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE -------------------------------
.SYNAPSE_SAVE
SETMP SYN-0                            ;LOAD LOOP INDEX!
READMP 1       ;READMPX
; ************************** 1. MJ+SI+TYPE ********************************
MOVA R6     ;R6 <--- S type + Sj
SHR
SHL
MOVR R6
MOVA R5                                ;<--MJ
SHL
SHL
ADD R6                      ;+TYPE+SJ
MOVR R3                     ;composed DATA
RST R0
SHR
STNC R3 ;SAVE DATA
```

```
; ***************************** 2. LJI+AJI *********************************
SWAP R2
MOVA R2                                 ;<--LJI
SWAP R2
SHL
SHL
SWAP R3
ADD R3                          ;+AJI
SWAP R3
MOVR R3                         ;composed DATA
RST R0
SHR
STNC R3                 ;SAVE DATA
RET
; ---------------------------------------------------------------------------------
; ---------------------------- SPIKE UPDATE --------------------------------
.SPIKE_UPDATE
LDALL R3,THETA1                   ;R3  <-- THETA1 = "0000F060"
SWAP R5                                             ;SR5 <-- Neuron Type + Si
MOVA R5
SHR
SHR
FREEZENC
        LDALL R3,THETA2     ;R3  <-- THETA2 = "0000F060"
UNFREEZE
MOVA R5
SHR
SHL
MOVR R5                                             ;R5  <-- Neuron Type + 0
SWAP R6              ;SR6 <--  Vi
MOVA R6               ;R0  <--  Vi
SUB  R3                                             ;R0  <--  Vi - (THETA1 or THETA2)
SWAP R6
FREEZENC
        MOVA R7                                             ;R0  <-- refractary period
        SHL
        FREEZEC
                MOVA  R5
                LDALL R3,UNO
                ADD   R3
                MOVR  R5       ;R5  <-- Neuron Type + 1
                SET   R7                ;R7  <-- activation of refractory time
        UNFREEZE
UNFREEZE
SWAP R5
RET
; ---------------------------------------------------------------------------------
; ------------------------------- REFRACTORY P ---------------------------
.REFRACTORY_P
MOVA R7
SHL                                                 ; -1ms
MOVR R7
RET
; ---------------------------------------------------------------------------------
; ------------------------------- NEURON SAVE ----------------------------
.NEURON_SAVE
SWAP R4           ;R4  <-- Mi
SWAP R5                                 ;R5  <-- Neuron Type + Si
SWAP R6
RST R3
MOVA R4
SHL
SHL
ADD R5
MOVR R3           ;R3  <-- Mi + Neuron Type + Si
;-------------------------- INDIVIDUAL DATA STORE ------------------
RST R0
SHR
STNC R3,NEU-1                   ;SRAM  <-- Neuron Type + Si
RST R0
SHR
STNC R6,NEU-2                   ;SRAM  <-- Vi
```

```
SWAP R0
CLRC
STNC R0,NEU-3                     ;SRAM  <-- SUM_WEIGHTS
SWAP R0
LDALL R3,                 ;MASK1 = "0000E000"
MOVA R5
SWAP R5
SHR
SHR
FREEZENC
        LDALL R3,MASK2  ;MASK2 = "00008000"
UNFREEZE
MOVA R7                                              ;ACC   <-- Tref
AND R3
SWAP R7                                      ;R7    <-- exponential
OR R7
SWAP R7
CLRC
STNC R0,NEU-4                     ;SRAM  <-- Tref + exponential
RET
; ------------------------------------------------------------------------------------------
;-------------------------ENABLE SPIKES PROPAGATION-----------------------
.SPIKES_ENABLE
SWAP R5                           ; ACC <== Spikes
MOVA R5
SWAP R5
SETC
SETMP SYN-0                       ; Point to Sj
READMP
RET
;------------------------------------------------------------------------------------------
; ---------------------------- EXPONENTIAL DECAY ----------------------------
.DECAY
RST R1
MOVA R2
MOVR R4
SHL
FREEZENC
        RST R0
        SUB R2
        MOVR R2
UNFREEZE
LOOP 15
        MOVA R2
        SHL
        MOVR R2
        FREEZENC
                MOVA R1
                ADD R3
                MOVR R1
        UNFREEZE
        MOVA R3
        SHR
        MOVR R3
ENDL
MOVA R1
SHR
MOVR R1
MOVA R4
SHL
FREEZENC
        RST R0
        SUB R1
        MOVR R1
UNFREEZE
MOVA R1
MOVR R2
RST R1
RET
; --------------------------------------------------------------------------
; **************************** PROCEDURES END ******************************
; **************************** MAIN PROGRAMME BEGIN **********************
```

```
.MAIN
GOTO NEURON_LOAD
GOTO MEMBRANE_VALUE
LOOP synapses
        GOTO SYNAPSE_LOAD
        GOTO SYNAPTIC_WEIGHT
        GOTO SYNAPSE_SAVE
ENDL
GOTO SPIKE_UPDATE
GOTO REFRACTORY_P
GOTO NEURON_SAVE
GOTO SPIKES_ENABLE
STOP
HALT
GOTO MAIN
; **************************** MAIN PROGRAMME END **************************
```

C.4 Assembler code of Iglesias and villa algorithm – SNAVA

```
AMAX="00000003"
DACT1="0000FFFA"
DACT2="0000FFFA"
DBACK="0000E7A3"
DMEM1="0000EF7D"
DMEM2="0000EF7D"
DSYN1="0000F9AE"
DSYN2="0000F9AE"
LMAX="00003FFF"
MMAX="00000666"
POT1="000003E8"
POT2="0000FFB0"
PROB="00001FFF"
THETA1="0000F060"
THETA2="0000F060"
VREST1="0000E188"
VREST2="0000E188"
UNO="00000001"
DOS="00000002"
CTETP="0000F448"
CTE1="00000007"

.CODE

LOOPN neurons_virtualized
; --------------------------INIT SOME VARIABLES----------------------------
LDALL R2,PROB
MOVA R2
RANDON
LLFSR R1
RANDOFF
AND R1
MOVR  R5
SWAPS R5                                   ;SR5_2 <-- activation probability
; ------------------------------------------------------------------------
ENDL

GOTO MAIN
; ------------------------------------------------------------------------
; ***************************** PROCEDURES BEGIN **************************
; ------------------------------- MEMBRANE VALUE -------------------------------
.MEMBRANE_VALUE
;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS ----------
```

```
 LDALL R4,DMEM1                                    ;R4    <-- DECAY DONATOR 1
 LDALL R5,VREST1                                   ;R5    <-- Vres1
            SWAPS R0                                  ;R0    <-- SR0_2 = Nt + Si
            MOVR  R3           ;R3    <-- Nt + Si
            SWAPS R0           ;SR0_2 <-- R0 = Nt + Si
            MOVA  R3
            SHRN DOS
            FREEZENC           ;IF NEURON TYPE = II (CONDITIONAL LOAD)
                        LDALL R4,DMEM2                         ;R4  <-- DECAY DONATOR 2
                        LDALL R5,VREST2                        ;R5  <-- Vres2
            UNFREEZE
;----------------------- R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) -----------------
            MOVA R3               ;R0  <-- R3 = Nt + Si
            RTR
            FREEZEC            ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
                        SWAPS R1                               ;R1  <-- SR1_2 = Vi
                        MOVA  R1                                           ;R0  <-- R1 = Vi
                        SUB   R5                               ;R0  <-- Vi - Vres
                        UNMUL   R4                                ;R0  <--(Vi(t)-Vres) * (Kmem)
                        MOVR  R2           ;R2  <--(Vi(t)-Vres) * (Kmem)
            UNFREEZE
            MOVA R3
            RTR
            FREEZENC           ;IF (Si = 1) THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
                        RST  R2               ;R2  <-- ((0)*(Vi(t)-Vres)*(Kmem)
            UNFREEZE
MOVA R2             ;R0  <-- (Vi(t)-Vres)*(Kmem)
ADD  R5             ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
SWAPS R2      ;R2  <-- SR2_2 = SUM_WEIGHTS
ADD  R2                         ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
MOVR R1          ;R1  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
SWAPS R1                 ;SR1_2  <-- R1 = Vi
RST R2                          ;SUM_WEIGHTS <-- 0
SWAPS R2      ;SR2_2 <-- R2 = SUM_WEIGHTS
RET
; -------------------------------------------------------------------------------
; ---------------------------------- SYNAPSE LOAD -------------------------------
.SYNAPSE_LOAD
LOADSP
      ;R4 <-- St + Sj
                    ;R5 <-- Aj
                    ;R6 <-- Lji
                    ;R7 <-- Mj
RET
; -------------------------------------------------------------------------------------------
; --------------------------------- SYNAPTIC WEIGHT ------------------------------
.SYNAPTIC_WEIGHT

MOVA R4          ; R0 <-- St + Sj
RTR
FREEZENC  ;IF (Sj = 1) THEN R0 <-- wji = Aji * P
            LDALL R1,POT1      ; R1  <--  POT1
            MOVA  R4              ; R0 <-- St + Sj
            SHRN  DOS
                        FREEZENC
                        LDALL R1,POT2
                        UNFREEZE
            MOVA R1        ;R0  <-- POT1 or POT2
            MUL  R5        ;R0  <-- wji = Aji * P
   SWAPS R2   ;R2  <-- SR2_2 = sumW
            ADD  R2                ;SR0 <-- wji = Sj * Aji * P
            MOVR R2        ;R2  <-- wji = Sj * Aji * P
   SWAPS R2   ;SR2_2 <-- R2  = sumW
UNFREEZE
RET
;-------------------------------------------------------------------------------------------
;---------------------------- REAL_VALUE_VARIABLE --------------------------
 .REAL_VALUE_VARIABLE
;---------------Lji(t+1)  = Lji(t) * Kact + Si(t) * Mj - Sj(t) * Mi(t)-------
LDALL R1,DACT1                          ;R1  <-- DACT1
MOVA  R4                          ;R0  <-- St + Sj
SHRN  DOS
```

```
FREEZENC        ;St = 1 = inhibitory synapse
            LDALL R1,DACT2                  ;R1  <-- DACT2
UNFREEZE
MOVA  R6        ;R0 <-- R6 = Lji
UNMUL   R1        ;R0 <-- Lji(t) * Kact
MOVR  R6        ;R6 <-- Lji

SWAPS R0    ;R0 <-- SR0_2 = St + Si
MOVR  R2
SWAPS R0    ;R0 <-- SR0_2 = St + Si
MOVA  R2
RTR
FREEZENC        ;IF Si = 1 THEN
            MOVA R7           ;R0  <-- R7 = Mj
            ADD  R6                             ;R0  <-- (Lji(t) * Kact) + (Si(t) * Mj)
            MOVR R6           ;R6  <-- (Lji(t) * Kact) + (Si(t) * Mj)
UNFREEZE

MOVA R4         ;R4  <-- St + Sj
RTR
FREEZENC        ;IF Sj = 1 THEN
            SWAPS R3       ;R3  <-- SR3_2 = Mi
            MOVA  R6          ;R0  <-- Mi
            SUB   R3                          ;R0  <--(Lji(t) * Kact) + (Si(t) * Mj) - Sj(t) * Mi(t)
            SWAPS R3       ;R3  <-- SR3_2 = Mi
            MOVR  R6          ;R6  <--(Lji(t) * Kact) + (Si(t) * Mj) - Sj(t) * Mi(t)
UNFREEZE
RET
; ------------------------------------------------------------------------------
; ------------------------------- ACTIVATION_VARIABLE --------------------------
.ACTIVATION_VARIABLE
MOVA R5              ;R0 <-- R5 = Aji
            FREEZEZ                ; IF (Aji =/ 0) THEN
                        LDALL R0,LMAX                  ; R0 <--- Lmax
                        SUB   R6          ; Lmax - Lji
                        RTL
                        FREEZENC
                                    MOVA R5                                ; R0 <-- R5 <-- Aji
                                    INC                                    ; R0 <-- Aji + 1
                                    MOVR R5                             ; R5 <-- Aji

                                    LDALL R0,AMAX
                                    SUB   R5                ; R0 <--  Amax - Aji
                                    RTL
                                    FREEZENC
                                                LDALL R5,AMAX
                                    UNFREEZE
                                    LDALL R0,LMAX    ; Lji=Lmax/2
                                    RTR
                                    MOVR R6
                        UNFREEZE
                                                                                ;else if (Lji < Lmin)
                        MOVA R6                                                 ; Lji --> ACC, Lmin=0
                        RTL
                        FREEZENC                                ; (Lji-Lmin)
                                    MOVA R5        ; R0 <-- R5 = Aji
                                    DEC                                    ; Aji-1
                                    MOVR R5                             ; Aji --> R5
                                    LDALL R0,LMAX    ; Lji=Lmax/2
                                    RTR
                                    MOVR R6
                        UNFREEZE
            UNFREEZE

MOVA R5
FREEZENZ                                          ;IF CONNECTION IS INACTIVE
            RST R6
UNFREEZE
RET
; ---------------------------------------------------------------------------------------
; ------------------------------- MEMORY_OF_LAST_PRESYNAPTIC_SPIKE ----------------------
.MEMORY_OF_LAST_PRESYNAPTIC_SPIKE
```

```
; ----------------- Mj(t+1) = (Sj(t) * Mmax) + (1 - Sj(t)) * Mj(t) * Ksyn -----------------
;--------------------------- R2 <-- (1 - Sj(t)) * Mj(t) * Ksyn ----------------------------
LDALL R1,DSYN1                    ; R1 <-- Ksyn1
MOVA R4                                                  ; R0 <-- R4 = Synapse Type + Sj
SHRN DOS
FREEZENC
        LDALL R1,DSYN2            ; R1 <-- Ksyn2
UNFREEZE

MOVA R4                                                  ; R0 <-- R4 = Synapse Type + Sj
RTR
FREEZEC    ; IF Sj = 0 THEN
MOVA R7                                                  ; R7 <-- Mj
UNMUL  R1          ; R0 <-- (1 - Sj(t)) * Mj * Ksyn1 or Ksyn2
MOVR R7          ; R3 <-- (1 - Sj(t)) * Mj * Ksyn1 or Ksyn2
UNFREEZE

MOVA R4                                                  ; R0 <-- R4 = Synapse Type + Sj
RTR
FREEZENC    ;IF Sj= 1  THEN R7 <-- Mmax
        LDALL R7,MMAX       ;R2 <-- Mmax
UNFREEZE
;MOVA R7          ; R0 <-- (1 - Sj(t)) * Mj * Ksyn1 or Ksyn2
RET
; -------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE -------------------------------
.SYNAPSE_SAVE
; THE SYNAPTIC PARAMETERS GO TO BUFFER 32 bits
;MOVA R5          ;R5 <-- Aji
MOVA  R6          ;R6 <-- Lji
;RTR
;RTR
RTL
RTL
OR    R5
MOVR  R1          ;R1 <-- Lji + Aji
;MOVA R4          ;R4 <-- St + Sj
MOVA  R7                              ;R0 <-- R7 = Mj
;RTR
;RTR
RTL
RTL
OR    R4                          ;R0 <-- Mj + St + Sj
STOREB
NOP

MOVA R4                           ;R0 <--- R4 = St + Sj   to delete the spike
RTR
RTL
MOVR R4
STORESP
RET
; -----------------------------------------------------------------------------------------------------
; ------------------------ MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE ---------------
.MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE
; ----------------- Mi(t+1) = (Si(t) * Mmax) + (1 - Si(t)) * Mi(t) * Ksyn -----------------


SWAPS R0        ;R0 <-- SR0_2 = St + Si
MOVR  R2        ;R2 <-- St + Si
SWAPS R0        ;SR0_2 <-- R0 = St + Si
MOVA R2
LDALL R1,DSYN1                    ;TYPE=1
SHRN DOS
        FREEZENC
                LDALL R1,DSYN2            ;TYPE=2
        UNFREEZE

SWAPS R3        ;R3 <-- SR3_2 = Mi

MOVA R2          ;R0 <-- St + Si
```

```
RTR
            FREEZEC
                        MOVA   R3           ;R0 <-- Mi
                        UNMUL  R1            ;R0 <-- (1 - Si(t)) * Mi * Ksyn1 or Ksyn2
                        MOVR   R3           ;R4 <-- (1 - Si(t)) * Mi * Ksyn1 or Ksyn2
            UNFREEZE

MOVA R2
RTR
            FREEZENC
                        LDALL R3,MMAX
            UNFREEZE

;MOVA R3
SWAPS R3        ; SR3_2 <-- R3 = Mi
RET
; ---------------------------------------------------------------------------
; --------------------------- SPIKE UPDATE --------------------------------
.SPIKE_UPDATE
SWAPS R0                         ;R0        <-- SR0_2 = Nt + Si
MOVR  R2
LDALL R3,THETA1                   ;R3       <-- THETA1 = "0000F060"
SHRN  DOS
FREEZENC
            LDALL R3,THETA2     ;R3       <-- THETA2 = "0000F060"
UNFREEZE
            MOVA R2
            RTR
            RTL
            MOVR R2              ;R2          <-- Neuron Type + 0 It has been set Si = 0

   SWAPS R1      ;R1 <-- SR1_2 = Vi
            MOVA  R1
   MOVR  R5
            SWAPS R1              ;SR1_2  <-- R1 = Vi
            RTL
                        FREEZEC
                                    LDALL  R5,CTETP      ; It has assigned a positive value under 30 because it has verified that is
lower than
                        UNFREEZE
MOVA  R5             ;R0          <--  Vi
SUB   R3                                     ;R0          <--  Vi - (THETA1 or THETA2)
            FREEZENC
      SWAPS             R4                    ;R4          <-- SR4 = Tref
                        RST                   R0
                        SUB     R4
                        SWAPS   R4
                        FREEZENZ    ; IF  (Z = 1) THEN Tref is setting
                        MOVA  R2
                        INC
                        MOVR  R2
                        LDALL   R4,CTE1   ;CTE1 = 7
                        SWAPS   R4
                        UNFREEZE
            UNFREEZE
MOVA  R2             ;R0    <--  Nt + Si
SWAPS R0                                                ;SR0_2 <-- R0 = Nt + Si
RET
; -----------------------------------------------------------------------------
; ------------------------------ REFRACTORY P --------------------------
.REFRACTORY_P
            SWAPS    R4                       ;R4          <-- SR4 = Tref
            MOVA    R4
            RTR
            MOVR    R4
            SWAPS    R4                       ;SR4         <-- R4 = Tref
RET
; -----------------------------------------------------------------------
.NEURON_SAVE
;SWAPS R0                                     ;Nt + Si
;SWAPS R1         ;Vi
;SWAPS R2         ;sum_W
```

```
;SWAPS R3          ;Mi
SWAPS  R0          ;R0  <-- SR0 = Nt + Si
MOVR   R1          ;R1  <-- R0
SWAPS  R0          ;SR0 <-- R0  = Nt + Si
SWAPS  R3          ;R3 <-- SR3 = Mi
MOVA   R3
;RTR
;RTR
RTL
RTL
OR     R1          ;R0 <-- Mi + Nt + Si
SWAPS  R3          ;R3 <-- SR3 = Mi
SWAPS  R1          ;R1 <-- SR1 = Vi
STOREB
NOP
SWAPS  R1
RST   R1
SWAPS R2           ;R2 <-- SR2 = sum_W
MOVA  R2
SWAPS R2
STOREB
NOP
RET
; ----------------------------- BACKGROUND_ACTIVITY----------------------------
.BACKGROUND_ACTIVITY
SWAPS R6          ; R6  <-- SR6_2 = exponential
MOVA  R6
SWAPS R6          ; SR6_2 <-- R6 = exponential

LDALL R4,PROB                                ; R4  <-- PROB = "00001FFF"
LDALL R3,DBACK         ; R3  <-- DBACK = "00005E2C"
UNMUL   R3                                            ; R0  <-- DBACK * exponential
MOVR  R2               ; R2  <-- DBACK * exponential
SWAPS R5                              ; R5  <-- SR5_2 = activation probability
MOVA R4                                                               ; R0  <-- PROB
SUB R2                ; R0  <-- PROB - (DBACK * exponential)
RANDON
CLRC
SUB R5                                                ; (PROB - (DBACK * exponential)) - Activation probability
FREEZENC                        ;If ((PROB - (DBACK * exponential)) > Activation probability) then
             LLFSR R5                    ; R1  <-- new activation probability
             RANDOFF
             MOVA R4            ; R0  <-- PROB = "00001FFF"
             MOVR R2            ; R2  <-- PROB = "00001FFF"
             AND  R5           ; R0  <-- PROB = "00001FFF" AND new activation probability
             MOVR R5            ; R1  <-- PROB = "00001FFF" AND new activation probability
              FREEZENC                                                ; IF  ( C = 1 ) THEN Tref
      SWAPS             R4                          ;R4          <-- SR4 = Tref
                        RST                         R0
                        SUB     R4
                        SWAPS   R4
                                     FREEZENZ    ; IF  (Z = 1) THEN Tref is setting
                                                 SWAPS   R0                           ; R0 <-- SR0_2 = Neuron Type + Si
                                                 RTR
                                                 RTL
                                                 INC
                                                 SWAPS   R0                           ; R0 <-- SR0_2 = Neuron Type + Si
                                                 LDALL   R4,CTE1   ;CTE1 = 7
                                                 SWAPS   R4
                                     UNFREEZE
              UNFREEZE
UNFREEZE
SWAPS R5                                     ;SR5_2 <-- R5 = activation probability
MOVA  R2
SWAPS R6          ; R6  <-- SR6_2 = exponential
MOVR  R6
SWAPS R6          ; SR6_2 <-- R6 = exponential
RET
; ---------------------------------------------------------------------
;-------------------------ENABLE SPIKES PROPAGATION-----------------------
.SPIKES_ENABLE
SWAPS R0
```

```
MOVR R2                ; R2 <-- St + Si
SWAPS R0
MOVA R2                                                          ; R0 <== Spikes
STOREPS
RET
; -----------------------------------------------------------------------
; **************************** PROCEDURES END *******************************
; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN
LOOPN neurons_virtualized
GOTO MEMBRANE_VALUE
ENDL

LOOPS synapses                ;synaptic loop
            GOTO  SYNAPSE_LOAD
            GOTO  SYNAPTIC_WEIGHT
            GOTOL REAL_VALUE_VARIABLE
            GOTOL ACTIVATION_VARIABLE
            GOTOL MEMORY_OF_LAST_PRESYNAPTIC_SPIKE
            GOTO  SYNAPSE_SAVE
ENDL

LOOPN neurons_virtualized
GOTO MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE
GOTO SPIKE_UPDATE
GOTO BACKGROUND_ACTIVITY
GOTO REFRACTORY_P
GOTO NEURON_SAVE
GOTO SPIKES_ENABLE
ENDL
NOP
SPKDIS
NOP
NOP
GOTO MAIN
; **************************** MAIN PROGRAMME END **************************
```

C.5 Assembler code of Izhikevich algorithm – SNAVA

```
PROB="0000FFFF"  ;0.99
CTE1="00000000"  ;5
CTE2="00000000"  ;2
CTE3="00000000"  ;140
CTE4="00000000"  ;0.04
CTE5="00000000"  ;0.5
CTE6V="00000000" ;6
CTE30V="00000000";30
MAXE="00000000"  ;800

.CODE
; -------------------------INITIALIZATION PHASE----------------------------

GOTO MAIN

.thalamic_input
            LDALL    R0,PROB
            RANDON
            LLFSR   R1                                    ; R1  <-- new probability
            RANDOFF
            LDALL   R2,CTE1                   ; CTE1 = 5
            MOVA    R1
            MUL     R2
            MOVR    R2         ; R2 <-- 5 * new probability
            SWAPS    R0            ;R0  <-- SR0 =  Nt + Si
            MOVR    R3             ;R3  <-- R0  =  Nt + Si
            SWAPS   R0  ;SR0 <-- R0  =  Nt + Si
            MOVA    R3  ;R0  <-- R3  =  Nt + Si
             RTR
            RTR
            FREEZENC            ;1 = inhibitory, 0 = excitatory
                LDALL   R0,CTE2                 ; CTE2 = 2
                MUL     R1
                MOVR    R2  ; R2 <-- 2 * new probability
            UNFREEZE
            MOVA  R2
            MOVR  R1
            SWAPS R1
            RET

.spike_update
            SWAPS    R0        ; Si <-- 0
            RTR
            RTL
            SWAPS   R0
            SWAPS    R2
            MOVA     R2            ; R0  <-- v
            SWAPS    R2
            RTL
                FREEZENC
                    LDALL R4,CTE6V
                UNFREEZE
            MOVA  R4
            LDALL R3,CTE30V       ; R3  <-- 30
            SUB   R3            ; v - 30
            FREEZENC            ; v = c , u = u + d, only if v >= 30
                SWAPS    R0       ; Si <-- 1
                 INC
                SWAPS   R0
            UNFREEZE
            RET

.membrane_potential_update

                SWAPS   R0
                MOVR    R1
                SWAPS   R0
                MOVA    R1
                RTR
                FREEZENC
                    SWAPS    R6                         ; SR6  <-- c
                    MOVA    R6
                    MOVR    R2
                    SWAPS    R2            ; SR2  <-- v = c
                    SWAPS    R6            ; SR6  <-- v = c

                    SWAPS    R7                              ; R7  <-- SR7 = d
                    SWAPS   R3                               ; R3  <-- SR3 = u
                    MOVA    R3                               ; R0  <-- u
                    ADD        R7         ; R0  <-- u + d
                    MOVR    R3
                    SWAPS   R3                               ; R3  <-- SR3, u = u + d
                    SWAPS    R7                              ; R7  <-- SR7 = d
                UNFREEZE
            RET


.SPIKES_ENABLE
                SWAPS R0
                MOVR R2              ; R2 <-- Si
                SWAPS R0
                MOVA R2                                          ; R0 <== Spikes
                STOREPS
RET
.SYNAPSE_LOAD
                LOADSP
                    ;R4 <--
                    ;R5 <--
                    ;R6 <-- Sj
                    ;R7 <-- S
RET
.SYNAPTIC_WEIGHT
```

```
            MOVA   R6   ; R0 <-- Sj
   RTR
   FREEZENC
                        SWAPS  R1              ; R1 <-- SR1 = I
               MOVA   R7           ; R0 <-- R7 = S
                        ADD    R1   ; R0 <-- I= I + S
                        MOVR   R1
                        SWAPS  R1
            UNFREEZE

RET
.SYNAPSE_SAVE

            MOVA R6      ;R0 <--- R4 = St + Sj   to delete the spike
            RTR
            RTL
            MOVR R6
            STORESP
RET
.membrane_potential_calculation

            SWAPS    R1           ;R1 <-- SR1 = I
            SWAPS    R2           ;R2 <-- SR2 = v
            SWAPS   R3            ;R3 <-- SR3 = u
            SWAPS   R4            ;R4 <-- SR4 = a
            SWAPS   R5            ;R5 <-- SR5 = b
            LDALL   R0,CTE3    ;CTE3 = 140
   SUB     R3  ;R0 = 140 - u
            ADD     R1  ;R0 = 140 - u + I
            MOVR    R7  ;R7 = 140 - u + I
            LDALL   R0,CTE1    ;5
            MUL     R2  ;R0 = 5 * v
            ADD     R7  ;R0 = 5 * v + 140 - u + I
            MOVR    R7  ;R7 = 5 * v + 140 - u + I
            LDALL   R0,CTE4 ;0.04
            MUL     R2
            MUL     R2
            ADD     R7  ;R0 = 0.04 * v * v + 5 * v + 140 - u + I
            MOVR    R7  ;R7 = 0.04 * v * v + 5 * v + 140 - u + I
            LDALL   R0,CTE5 ;0.5
            MUL     R7
            ADD     R2
            MOVR    R2            ;R2 = v = v + 0.5 * (0.04 * v * v + 5 * v + 140 - u + I)
            SWAPS    R1           ;R1 <-- SR1 = I
            SWAPS    R2           ;R2 <-- SR2 = v
            SWAPS   R3            ;R3 <-- SR3 = u
            SWAPS   R4            ;R4 <-- SR4 = a
            SWAPS   R5            ;R5 <-- SR5 = b
RET
.recovery_variable_calculation

            SWAPS   R2  ;R2 <-- SR2 = v
            SWAPS   R3            ;R3 <-- SR3 = u
            SWAPS   R4            ;R4 <-- SR4 = a
            SWAPS   R5            ;R5 <-- SR5 = b
            MOVA    R2
            SUB     R3  ; v - u
            MUL     R5  ; b * (v - u)
            MUL     R4  ; a * (b * (v - u))
            ADD     R3
            MOVR    R3  ; u = u + a * (b * (v - u))
            SWAPS   R2  ;R2 <-- SR2 = v
            SWAPS   R3            ;R3 <-- SR3 = u
            SWAPS   R4            ;R4 <-- SR4 = a
            SWAPS   R5            ;R5 <-- SR5 = b

RET
;------------------------------ MAIN -------------------------

.MAIN
            LOOPN neurons_virtualized
```

```
        GOTO thalamic_input
        GOTO spike_update
        GOTO membrane_potential_update
        GOTO SPIKES_ENABLE
        ENDL
        SPKDIS
        NOP
        LOOPS synapses
        GOTO SYNAPSE_LOAD
        GOTO SYNAPTIC_WEIGHT
        GOTO SYNAPSE_SAVE
        ENDL
        LOOPN neurons_virtualized
        GOTO membrane_potential_calculation
        GOTO membrane_potential_calculation
        GOTO recovery_variable_calculation
        ENDL
GOTO MAIN
```

C.6 Assembler code of Leaky integrate-and-fire algorithm – SNAVA

```
DMEM="0000EF7D"
VREST="00008AD0"   ; -300 mV
POT1="00000002"
POT2="0000FFF8"
CTE1="00000007"    ;  "111"
DOS="00000002"     ;  "2"
CTETP="0000F448"

.CODE

GOTO MAIN

; ---------------------------------------------------------------------------
; ****************************** PROCEDURES BEGIN ****************************

; ------------------------------- MEMBRANE VALUE -----------------------------
.MEMBRANE_VALUE
;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS ----------
        LDALL R4,DMEM                       ;R4   <-- Kmem
        LDALL R5,VREST                      ;R5   <-- Vres
        SWAPS R0                              ;R0   <-- SR0 = Si
        MOVR  R3          ;R3   <--  Si
        SWAPS R0          ;SR0   <--  R0 = Si
        MOVA  R3
        RTR
        FREEZEC           ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
                SWAPS   R1                                  ;R1  <-- SR1 = Vi
                MOVA    R1                                  ;R0  <-- R1 = Vi
                SUB     R5                          ;R0  <-- Vi - Vres
                UNMUL   R4                            ;R0  <--(Vi(t)-Vres) * (Kmem)
                MOVR    R2          ;R2  <--(Vi(t)-Vres) * (Kmem)
        UNFREEZE
        MOVA R3
        RTR
        FREEZENC          ;IF (Si = 1) THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
                RST  R2                             ;R2  <-- ((0)*(Vi(t)-Vres)*(Kmem)
        UNFREEZE
        MOVA      R2                        ;R0  <-- (Vi(t)-Vres)*(Kmem)
        ADD       R5                        ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
        SWAPS     R2                        ;R2  <-- SR2 = SUM_WEIGHTS
        ADD       R2                                ;R0    <--   (Vres1   or   Vres2)   +   (1-Si(t))*(Vi(t)-
        Vres)*(Kmem) SUM_WEIGHTS
        MOVR      R1            ;R1  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
        SWAPS     R1                                ;SR1 <-- R1 = Vi
        RST       R2                                ;SUM_WEIGHTS <-- 0
```

```
            SWAPS    R2                                    ;SR2 <-- R2 = SUM_WEIGHTS
RET
;  --------------------------------------------------------------------------------

; ---------------------------------- SYNAPSE LOAD -------------------------------

.SYNAPSE_LOAD
            LOADSP
                                    ;R7 <-- St + Sj
                                    ;R6 <-- Aj
RET
; --------------------------------------------------------------------------------

; --------------------------------- SYNAPTIC WEIGHT ------------------------------
.SYNAPTIC_WEIGHT

MOVA R7           ; R0 <-- St + Sj
RTR
FREEZENC  ;IF (Sj = 1) THEN R0 <-- wji = Aji * P
            LDALL R1,POT1     ; R1  <--  POT1
            MOVA  R7                  ; R0 <-- St + Sj
            SHRN  DOS
                        FREEZENC
                        LDALL R1,POT2
                        UNFREEZE
            MOVA     R1          ;R0  <-- POT1 or POT2
            MUL      R6          ;R0  <-- wj = Aj * P
   SWAPS             R2                   ;R2  <-- SR2 = sumW
            ADD      R2                      ;SR0 <-- wj = Sj * Aj * P
            MOVR     R2          ;R2  <-- wj = Sj * Aj * P
   SWAPS             R2                   ;SR2 <-- R2  = sumW
UNFREEZE
RET
;----------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE -----------------------
.SYNAPSE_SAVE

RST     R1
MOVA     R6
SHLN     DOS
OR                  R7
STOREB      ;buffer <-- R0 = A + St + Sj

MOVA     R7      ;R0 <--- R4 = St + Sj   to delete the spike
RTR
RTL
MOVR     R7
STORESP
RET
; ----------------------------------------------------------------------------
; ---------------------------- SPIKE UPDATE ------------------------
.SPIKE_UPDATE
            SWAPS    R4                   ;R4  <-- SR4 = THETA
            SWAPS    R0                   ;R0  <-- Si <-- '0'
            RTR
            RTL
            SWAPS    R0
   SWAPS R1                               ;R1 <-- SR1_2 = Vi
            MOVA  R1
   MOVR  R5
            SWAPS R1                      ;SR1_2  <-- R1 = Vi
            RTL
                        FREEZEC
                                    LDALL R5,CTETP
                         UNFREEZE
            MOVA     R5           ;ACC      <--  Vi
            SUB      R4                                    ;ACC       <--  Vi - (THETA1 or THETA2)
                        FREEZENC
                                    SWAPS    R3                      ;R3          <-- SR3 = Tref
                                    RST                   R0
                                    SUB     R3
                                    SWAPS   R3
                        FREEZENZ    ; IF  (Z = 1) THEN Tref is setting
                                SWAPS   R3
                                SWAPS   R0
                                RST     R0
                                INC
                                SWAPS   R0
                                LDALL   R3,CTE1   ;CTE1 = 7
                                SWAPS   R3
                        UNFREEZE
                UNFREEZE
        SWAPS    R4                                        ;R4  <-- SR4 = THETA
RET
; ----------------------------------------------------------------------------
; ------------------------------- REFRACTORY P ---------------------------
.REFRACTORY_P
        SWAPS    R3                 ;R3          <-- SR3 = Tref
        MOVA    R3
        RTR
        MOVR    R3
        SWAPS    R3                 ;SR3         <-- R3 = Tref
RET
; -----------------------------------------------------------------------
; ------------------------------- Ethernet TX ---------------------------
.NEURON_SAVE
        ;SR0 <-- Si
        ;SR1 <-- Vi
        ;SR2 <-- Sum_W
        ;SR3 <-- Tref
        ;SR4 <-- theta
        SWAPS R0
        MOVR  R5
        SWAPS R0
        SWAPS R3
        MOVA  R3
        RTL
        OR    R5
        SWAPS R1
        STOREB              ;buffer = R1 + R0 = Vi + Tref + Si
        SWAPS R1
        SWAPS R3
        SWAPS R2
        MOVA  R2
        STOREB      ;buffer = R0 = Sum W
        SWAPS R2
RET
;-------------------------ENABLE SPIKES PROPAGATION-----------------------
.SPIKES_ENABLE
SWAPS    R0
MOVR     R2             ; R2 <-- St + Si
SWAPS    R0
MOVA     R2                                                ; R0 <== Spikes
STOREPS
RET
; ******************************** PROCEDURES END *******************************

; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN

LOOPN neurons_virtualized
GOTO MEMBRANE_VALUE
ENDL
        LOOPS synapses
                GOTO SYNAPSE_LOAD
                GOTO SYNAPTIC_WEIGHT
                GOTO SYNAPSE_SAVE
        ENDL
LOOPN neurons_virtualized
GOTO SPIKE_UPDATE
GOTO REFRACTORY_P
GOTO NEURON_SAVE
GOTO SPIKES_ENABLE
ENDL
```

```
SPKDIS
NOP
NOP
GOTO MAIN
; *************************** MAIN PROGRAMME END *************************
```

C.7 Assembler code of Leaky integrate-and-fire algorithm – SNAVA+

```
DMEM1="0000EF7D"
DMEM2="0000EF7D"
POT1="000003E8"
POT2="0000FFB0"
THETA1="0000E380"
THETA2="0000E380"
VREST1="0000E188"
VREST2="0000E188"
UNO="00000001"
DOS="00000002"
CTETP="0000F448"
CTE1="00000000"

.CODE

GOTO MAIN

.LOAD_NEURAL_PARAMETERS
 NOP
 NOP
 NOP
RET

.SAVE_NEURAL_PARAMETERS
 NOP
 NOP
 NOP
RET

; --------------------------------------------------------------------------
; **************************** PROCEDURES BEGIN **************************
; ------------------------------- MEMBRANE VALUE -------------------------------
.MEMBRANE_VALUE
;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS ----------

 LDALL R4,DMEM1                                   ;R4    <-- DECAY DONATOR 1
 LDALL R5,VREST1                                  ;R5    <-- Vres1
            SWAPS R0                                 ;R0    <-- SR0_2 = Nt + Si
            MOVR  R3              ;R3    <-- Nt + Si
            SWAPS R0              ;SR0_2 <-- R0 = Nt + Si
            MOVA  R3
            SHRN DOS
            FREEZENC              ;IF NEURON TYPE = II (CONDITIONAL LOAD)
                        LDALL R4,DMEM2                            ;R4  <-- DECAY DONATOR 2
                        LDALL R5,VREST2                           ;R5  <-- Vres2
            UNFREEZE
;----------------------- R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) -----------------
            MOVA R3                 ;R0  <-- R3 = Nt + Si
            SHRN UNO
            FREEZEC               ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
                        SWAPS R1                                  ;R1  <-- SR1_2 = Vi
                        MOVA  R1                                              ;R0  <-- R1 = Vi
                        SUB   R5                                  ;R0  <-- Vi - Vres
                        UNMUL   R4                                  ;R0  <--(Vi(t)-Vres) * (Kmem)
                        MOVR  R2              ;R2  <--(Vi(t)-Vres) * (Kmem)
            UNFREEZE

            MOVA R3
            SHRN UNO
            FREEZENC              ;IF (Si = 1) THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
                        RST  R2                   ;R2  <-- ((0)*(Vi(t)-Vres)*(Kmem)
            UNFREEZE
MOVA R2                 ;R0  <-- (Vi(t)-Vres)*(Kmem)
ADD  R5                 ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
SWAPS R2       ;R2  <-- SR2_2 = SUM_WEIGHTS
ADD  R2                                   ;R0  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
MOVR R1            ;R1  <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
SWAPS R1                          ;SR1_2  <-- R1 = Vi
RST R2                                    ;SUM_WEIGHTS <-- 0
SWAPS R2       ;SR2_2 <-- R2 = SUM_WEIGHTS
RET
; -----------------------------------------------------------------------------------------
; ---------------------------------- SYNAPSE LOAD -------------------------------
.SYNAPSE_LOAD
NOP
NOP
NOP
LOADSP
      ;R4 <-- St + Sj
                        ;R5 <-- Aj
                        ;R6 <-- Lji
                        ;R7 <-- Mj
RET
; ----------------------------------------------------------------------------------------
; --------------------------------- SYNAPTIC WEIGHT --------------------------
.SYNAPTIC_WEIGHT

MOVA R4            ; R0 <-- St + Sj
SHRN UNO
FREEZENC  ;IF (Sj = 1) THEN R0 <-- wji = Aji * P
            LDALL R1,POT1      ; R1  <--  POT1
            MOVA  R4                   ; R0 <-- St + Sj
            SHRN  DOS
                        FREEZENC
                        LDALL R1,POT2
                        UNFREEZE
            MOVA R1         ;R0  <-- POT1 or POT2
            MUL  R5         ;R0  <-- wji = Aji * P
   SWAPS R2    ;R2  <-- SR2_2 = sumW
            ADD  R2                    ;SR0 <-- wji = Sj * Aji * P
            MOVR R2         ;R2  <-- wji = Sj * Aji * P
   SWAPS R2    ;SR2_2 <-- R2  = sumW
UNFREEZE
RET
;-------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE -------------------------------
.SYNAPSE_SAVE
; THE SYNAPTIC PARAMETERS GO TO BUFFER 32 bits
;MOVA R5                ;R5 <-- Aji
MOVA  R6                ;R6 <-- Lji
SHLN UNO
SHLN UNO
OR    R5
MOVR  R1                ;R1 <-- Lji + Aji
;MOVA R4                ;R4 <-- St + Sj
MOVA  R7                                      ;R0 <-- R7 = Mj
SHLN UNO
SHLN UNO
OR    R4                                  ;R0 <-- Mj + St + Sj
STOREB
NOP
MOVA R4                                   ;R0 <--- R4 = St + Sj   to delete the spike
SHRN UNO
SHLN UNO
MOVR R4

STORESP
```

```
RET
; --------------------------------------------------------------------------------
; ---------------------------- SPIKE UPDATE --------------------------------
.SPIKE_UPDATE

SWAPS R0                                          ;R0        <-- SR0_2 = Nt + Si
MOVR  R2
LDALL R3,THETA1                 ;R3      <-- THETA1 = "0000F060"
SHRN  DOS
FREEZENC
            LDALL R3,THETA2     ;R3      <-- THETA2 = "0000F060"
UNFREEZE
            MOVA R2
            SHRN UNO
            SHLN UNO
            MOVR R2                                          ;R2        <-- Neuron Type + 0 It has been set Si = 0
   SWAPS R1      ;R1 <-- SR1_2 = Vi
            MOVA  R1
   MOVR  R5
            SWAPS R1            ;SR1_2  <-- R1 = Vi
            SHLN UNO
                        FREEZEC
                                    LDALL  R5,CTETP      ; It has assigned a positive value under 30 because it has verified that is
lower than 0
                        UNFREEZE
MOVA  R5             ;R0        <--  Vi
SUB   R3                                  ;R0        <--  Vi - (THETA1 or THETA2)
            FREEZENC
      SWAPS             R4                  ;R4        <-- SR4 = Tref
                        RST                 R0
                        SUB     R4
                        SWAPS   R4
                        FREEZENZ    ; IF  (Z = 1) THEN Tref is setting
                        LDALL R3,UNO
                        MOVA  R2
                        ADD   R3
                        MOVR  R2
                        LDALL   R4,CTE1   ;CTE1 = 7
                        SWAPS   R4
                        UNFREEZE
            UNFREEZE
MOVA  R2             ;R0   <--  Nt + Si
SWAPS R0                                          ;SR0_2 <-- R0 = Nt + Si
RET
; --------------------------------------------------------------------------------
; ------------------------------- REFRACTORY P ---------------------------
.REFRACTORY_P
            SWAPS    R4                 ;R4        <-- SR4 = Tref
            MOVA    R4
            SHRN UNO
            MOVR    R4
            SWAPS    R4                 ;SR4       <-- R4 = Tref
RET
; ---------------------------------------------------------------------------------

.NEURON_DISPLAY
;SWAPS R0                               ;Nt + Si
;SWAPS R1          ;Vi
;SWAPS R2          ;sum_W
;SWAPS R3          ;Mi
SWAPS  R0          ;R0  <-- SR0 = Nt + Si
MOVR   R1          ;R1  <-- R0
SWAPS  R0          ;SR0 <-- R0  = Nt + Si
SWAPS  R3          ;R3 <-- SR3 = Mi
MOVA   R3
SHLN UNO
SHLN UNO
OR     R1          ;R0 <-- Mi + Nt + Si
SWAPS  R3          ;R3 <-- SR3 = Mi
SWAPS  R1          ;R1 <-- SR1 = Vi
STOREB
NOP
```

```
SWAPS  R1

RST   R1
SWAPS R2           ;R2 <-- SR2 = sum_W
MOVA  R2
SWAPS R2
STOREB
NOP
RET
; ----------------------------- BACKGROUND_ACTIVITY----------------------------
.BACKGROUND_ACTIVITY

SWAPS R3          ; R3  <-- SR3_2 = INITIAL SPIKING
MOVA  R3
SWAPS R3
SHRN UNO
MOVR R3
                    FREEZENC    ; IF  (Z = 1) THEN Tref is setting
                                    SWAPS   R0                     ; R0 <-- SR0_2 = Neuron Type + Si
                                    SHRN    UNO
                                    SHLN    UNO
                                    LDALL R0,UNO
                                    SWAPS   R0                     ; R0 <-- SR0_2 = Neuron Type + Si
                         UNFREEZE
SWAPS R3                            ;SR3_2 <-- R3 = NEW SPIKING REG VALUE
RET
; ---------------------------------------------------------------------------------------------
;-------------------------ENABLE SPIKES PROPAGATION----------------------
.SPIKES_ENABLE
SWAPS R0
MOVR R2             ; R2 <-- St + Si
SWAPS R0
MOVA R2                                                ; R0 <== Spikes
STOREPS
RET
; -------------------------------------------------------------------------------------------
; **************************** PROCEDURES END ******************************

; **************************** MAIN PROGRAMME BEGIN ***********************
.MAIN
LOOPN neurons_virtualized
GOTO LOAD_NEURAL_PARAMETERS
GOTO MEMBRANE_VALUE
GOTO SAVE_NEURAL_PARAMETERS
ENDL

LOOPS synapses              ;synaptic loop
        GOTO  SYNAPSE_LOAD
        GOTO  SYNAPTIC_WEIGHT
        GOTO  SYNAPSE_SAVE
ENDL
LOOPN neurons_virtualized
GOTO LOAD_NEURAL_PARAMETERS
GOTO SPIKE_UPDATE
GOTO BACKGROUND_ACTIVITY
GOTO NEURON_DISPLAY
GOTO SPIKES_ENABLE
GOTO SAVE_NEURAL_PARAMETERS
ENDL
NOP
SPKDIS
NOP
NOP
GOTO MAIN
; **************************** MAIN PROGRAMME END *************************
```

Onset detection

This annexure presents the current ongoing work. This ongoing work is dedicated to the development of an implementation based on the onset detection, which is inspired by the functionality of the cochlea. The preliminary results of this implementation, which have recently been obtained, are provided in this annexure.

D.1 Onset detection

The ear in our human body detects sound from the external environment in an amazing manner and interprets the signals by using signal processing techniques. Onset refers to the starting of any changes in acoustic signals which can be perceived by the ear [1]. Onset detection is the process of detecting the beginning of these changes in the sound that is entering the system. This detection is used in various applications such as speech recognition, music transcription, sound segmentation, lip synchronisation and so on [1, 2]. One of the future works of SNAVA is to imitate the functionality of the ear´s main auditory portion i.e. Onset detection by the Cochlea. This is done by means of the 3 reservoir algorithm using the Leaky Integrate and Fire model of Spiking Neural Networks on SNAVA. The results of this implementation are presented in this section. The intricate details about this implementation can be seen in the thesis of Ms.Sanjana Sekar [3].

D.1.1 Onset detection system description

The diagrammatic representation of the complete implementation of the detection of onset is shown in Fig D.1. Onset detection is made up of four main blocks they are the sensors, filters, spike coders and the spiking neural network. The sensors are those which detect the external parameters and convert into equivalent electrical signal. Here the sound is being converted into equivalent electrical signals. These signals are then passed through the bank of filters. Each filter is centered to a particular frequency and the signals are being filtered accordingly. The response from the filter is then passed through the spike coder in order to generate equivalent spikes for these signals so as to activate the spiking neural network. Finally, these spikes enter the SNN topology and accordingly alter the neural and synaptic parameters of certain neurons so that certain neurons fire indicating the detection of onset in the system. This system was proposed in [1,2], the results are shown in simulation level using MATLAB software. This work concentrates on extending the same project in a real time environment with the help of SNAVA. The schematic of the entire system is shown in Fig D.1.

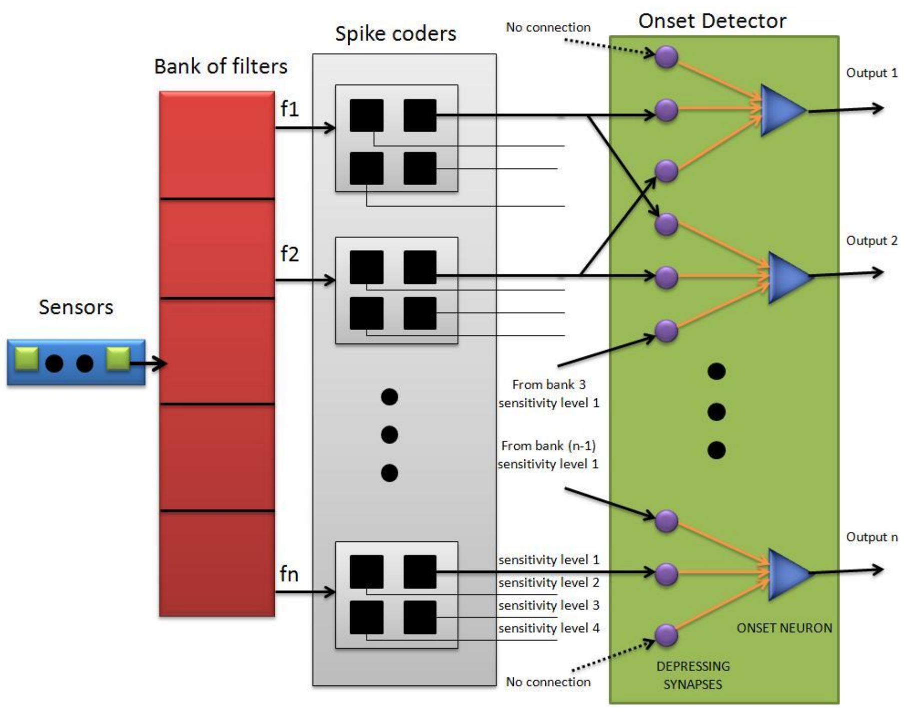


Figure D.1: Schematic of the Bio-inspired system using LIF-Reservoir model for Cochlea, from simplicity of the drawing only one sensitivity level is connected. This Figure is extracted from [1].

D.1.1.1 Filtering

As shown in Fig. D.1, the input sound is first detected using a sensor (Eg. Microphone) and sampled at the rate of a minimum of 16K samples per second and 16 bits linear. These samples are then filtered using a series of Bandpass filters called Gammatone filters [1]. The response of this filter is analogous to the basilar membrane of the cochlea [1].

Each filter is tuned at different centre frequencies F_c (around 15 bands are being used) ranging between 250 Hz to 6 KHz .The filters can be designed by determining the coefficients in MATLAB software (use gammatonefir function from the LTFT toolbox) and getting the IP core from Xilinx filter design tool. The filter delay also referred to as the gammatone delay, is assumed to be proportional to the reciprocal of the bandwidth. The filtered signals are then passed to the Spike coder Module in order to obtain the onset fingerprint.

D.1.1.2 Spike coders

The spike coders are similar to the design proposed by Smith [1]. The design of these coders has been derived from the concept proposed by Ghitza [1]. Here the filtered signal is converted to an equivalent spike which is fed to the SNN. The advantage of spike based representation is that this makes the system to work under a wide dynamic range [1]. For every filter bank there is a corresponding spike coder associated with it. In other words there is a one to one connection between the outputs of the gammatone filter to the input of the spike coders. Each spike coder block consists of one input and certain number of sensitivity levels as outputs. Each coder can have N sensitivity levels. The sensitivity level is the minimum energy level or mean voltage level the signal must have crossed in order to produce a spike. The threshold levels are made to be 3dB or 6 dB away from each other [1]. The equation for the threshold levels is given by:

$$E_i = D^i \cdot E_0 \tag{D.1}$$

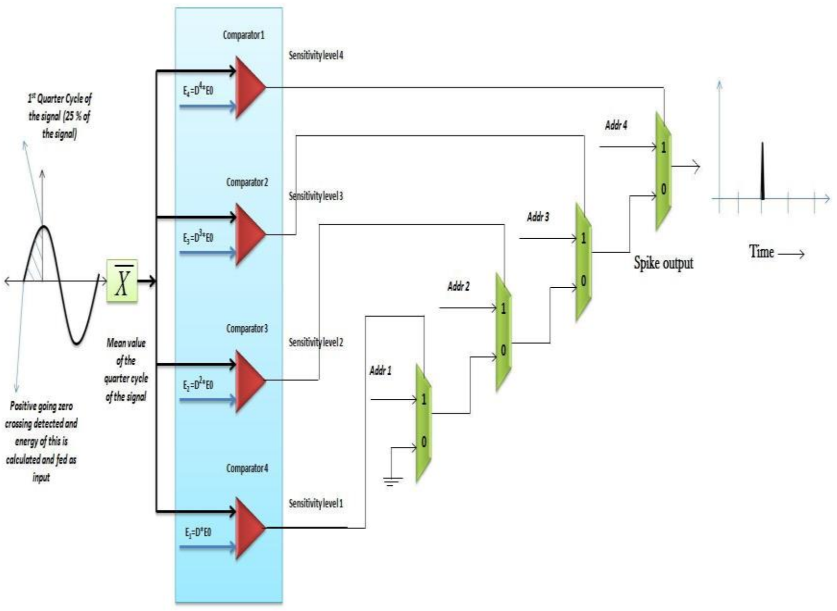


Figure D.2: Schematic of a single spike coder with 4 sensitivity levels. (Each box is a comparator module)[1]

where E_i is the energy associated with the i^{th} band where i=1 to N. E_0 is the minimum mean voltage of the signal and D is the separation between the threshold levels whose value is around 3dB as discussed before Each spike coder consists of N comparators here N will be equal to the number of sensitivity levels of every coder has. . The output from the filter is taken and the positive going zero crossing is being detected. Once the zero crossing is detected the overall mean voltage level for the first quarter cycle is calculated. This energy is compared with the threshold level value as mentioned in equation (D.1). If the value crosses this threshold value an equivalent spike is being generated in the same instance of time. A point to be noted is that the greater the threshold value the lesser would be the sensitivity level. Since sensitivity level describes the maximum ability of the coder to detect the weakest signal possible. When there is a spike generated in a band “m” then there will be spikes generated in all j bands where j ranges between 0 to m. The schematic diagram of a spike coder implemented is shown below in Fig. D.2.

Here each threshold level is D dB more than the previous value. The value of D was taken as 3 dB i.e. 1.414. Upon observing the outputs one can see that all bands before the third band produce spike as the third band is the maximum value the signal crosses the threshold value.

D.1.1.3 Onset detector

The onset detector is designed using the reservoir model. The topology of the network is shown in Fig. D.1. As we can see that each onset neuron has three inputs. One comes from the current frequency band and one from the previous band and the last from the next band. In case of the first and the last onset neuron we can see from the diagram that there are only two inputs. The third one is indicated by a dotted line which represents no connection as there is no previous band or next band respectively [1]. This can be clearly seen in the diagram. Here the synapses are depressing in nature and the onset neuron fires once the membrane potential crosses a particular threshold value. Depending upon the frequency of the signal the threshold value varies from each onset neuron. The firing of a particular neuron indicates the presence of the signal and hence onset is detected. The neural model used in this application is leaky integrate and fire which has already been discussed in detail in chapter 4.

D.1.1.4 Three reservoir model

The onset detection block is being designed as per the 3 reservoir model [1] of the synapses. As the name suggests this model has three interconnected regions/reservoirs consisting of the neurotransmitter which are:

1. M= The presynaptic neurotransmitter reservoir (available)
2. C= The amount of neurotransmitter in the synaptic cleft (in use)
3. R= The amount if neurotransmitter in the process of reuptake (which is used, but not yet available again)

The interconnections of these three regions or reservoirs are done using the assistance of the three differential equations which explicitly describe their connection [1]. These equations are:

$$
\begin{aligned}
dM/dt &= \beta R - gM \\
dC/dt &= gM - \alpha C \\
dR/dt &= \alpha C - \beta R
\end{aligned}
\tag{D.2}
$$

Here α and β are rate constants, and g is a positive quantity when a spike occurs whereas when there is no spike it is considered to be zero. These constants are calculated for each sample period. Neither the losses nor the synthesis of the neurotransmitter are taken into account while deriving these equations and the amount of post-synaptic depolarization is assumed to be directly proportional to C.

The 3 Reservoir model can be visualized as shown in the diagram below (Fig D.3). The shown processes happen in a parallel manner except for the third one which happens only when a spike is encountered. For initial level implementation, the sinusoidal wave from the generator is taken as input. The gammatone filters are replaced by a simple band pass filter with good frequency response. Once the output from the onset detector is available it indicates the beginning of the musical note.

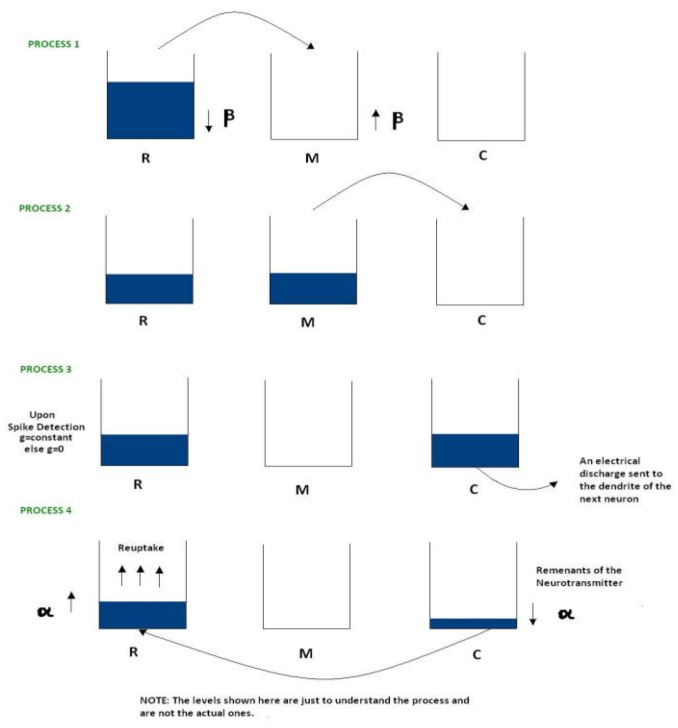

PROCESS 1
β
β
R
M
C
PROCESS 2
R
M
C
PROCESS 3
Upon Spike Detection g=constant else g=0
R
M
C
An electrical discharge sent to the dendrite of the next neuron
PROCESS 4
Reuptake
α
Remenants of the Neurotransmitter
α
R
M
C
NOTE: The levels shown here are just to understand the process and are not the actual ones.

Figure D.3: Diagrammatic Representation of the 3 Reservoir model [1]

In order to implement the above equations, they are integrated and converted into equivalent differential equations [1] which are shown in equation (D.2).

$$M(t+1) = \beta R(t) + \big(1 - (Sj(t) * g)\big)\, M(t)$$

$$C(t+1) = \big((Sj(t) * g)\big)M(t) + (1-\alpha)C(t)$$

$$R(t+1) = (\alpha * C(t)) + (1-\beta)\, R(t)$$

Here the condition for the constant g is included by means of adding $Sj(t)$ to the equation

D.1.2 Experimental results

In the experiments carried out, the filters were not included as it involves large amount of computation which is the current work being done Advanced Hardware Architecture (AHA) research group members from Universitat Politècnica de Catalunya, Barcelona. In order to check the system, an equivalent lookup table was implemented for the first level of testing the system assuming the output from this lookup table is analogous to the filter response. The lookup table consists of several sample values of sinusoidal signals which are stored in ROM memory. These signals are generated according to the frequency and amplitude the user provides. A schematic diagram of the implemented onset detection block is depicted in Fig. D.4 below:

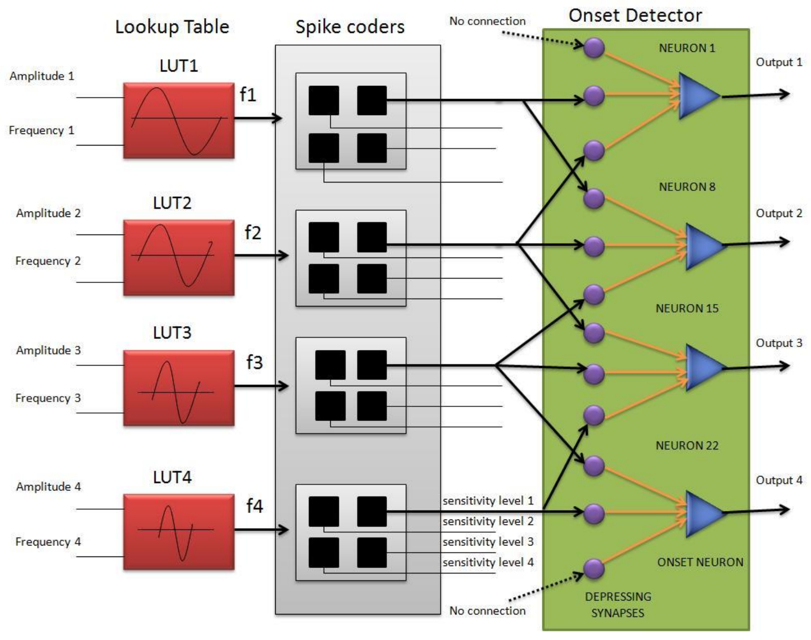


Figure D.4: Onset Detection using Lookup Table as input to the spike coders [1]

The lookup table works in such a way that one unit of signal will be equivalent to 1 volt of sample with frequency of 1 Hz. Based upon the hexadecimal value of amplitude and frequency the user gives an equivalent number of units are being generated [7].

Onset detection was successfully working in the SNAVA architecture and the experiments were done for frequencies in the range of 250 Hz to 6 kHz as proposed in [1]. Here the frequency values were taken into account was 500, 1500, 3500 and 5000 Hz so that they fall into the range proposed as well as they are logarithmic in nature. The detection of 4 frequencies is depicted in the neural raster plot shown in Fig D.5. Here the onsets are detected at the emulation cycles 7 and 8.

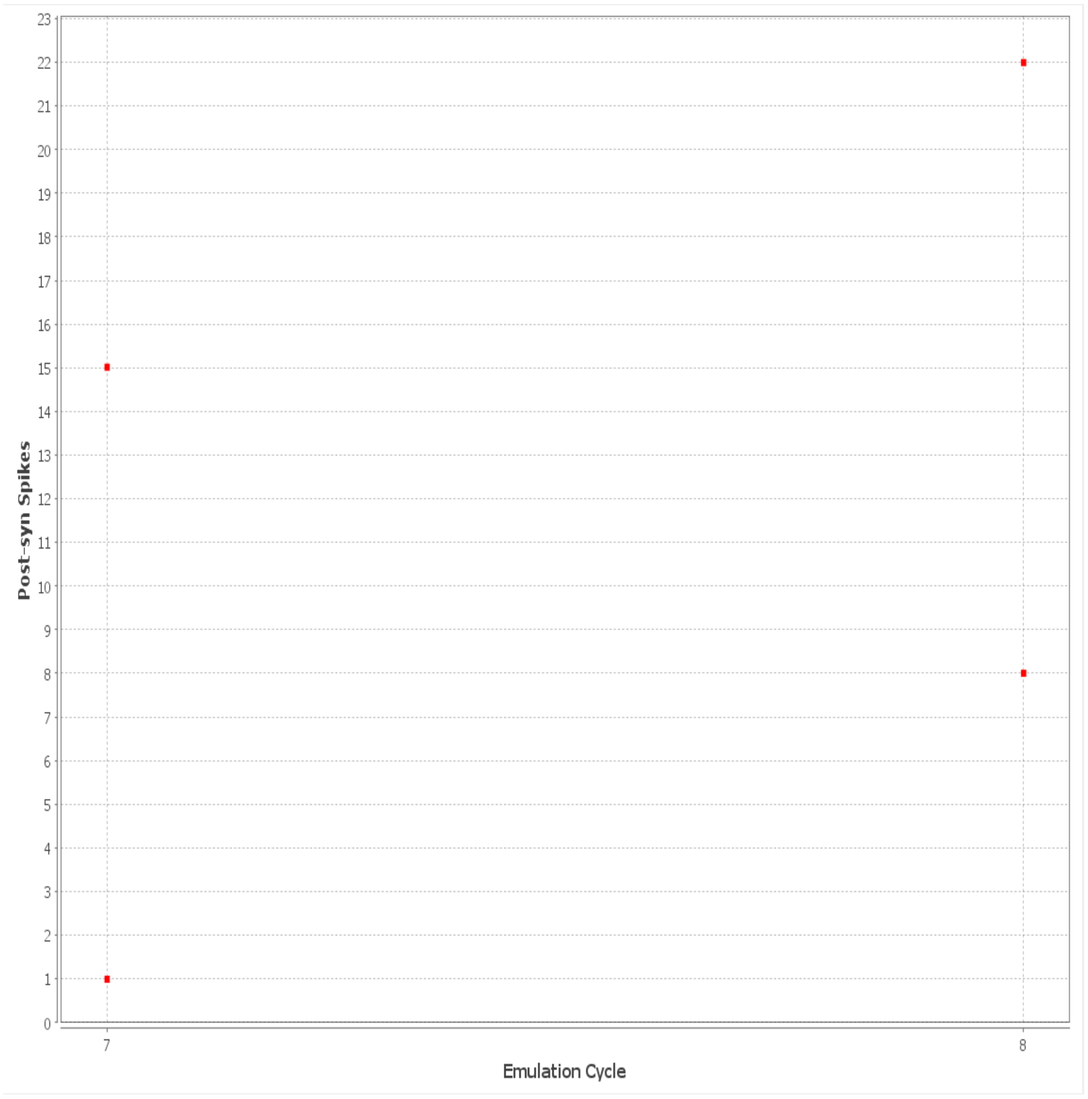


Figure D.5: Raster plot of the Onset Detector block detecting 4 different frequencies by neuron no 1, 8, 15 and 22.

The behavior of each neuron was studied by analyzing the neural waveform of the membrane potential, the post synaptic spike and the sum of weights. To understand the neuron 1 is analyzed here. This is shown in Fig D.6. As can be seen from this figure, the initial membrane potential is around -300mV and it keeps rising for every emulation cycle. The onset is detected during the emulation cycle 11 which is clear from the raster plot. At this moment, a post synaptic spike is being generated which indicates the firing of the onset neuron. This marks the detection of the frequency component.

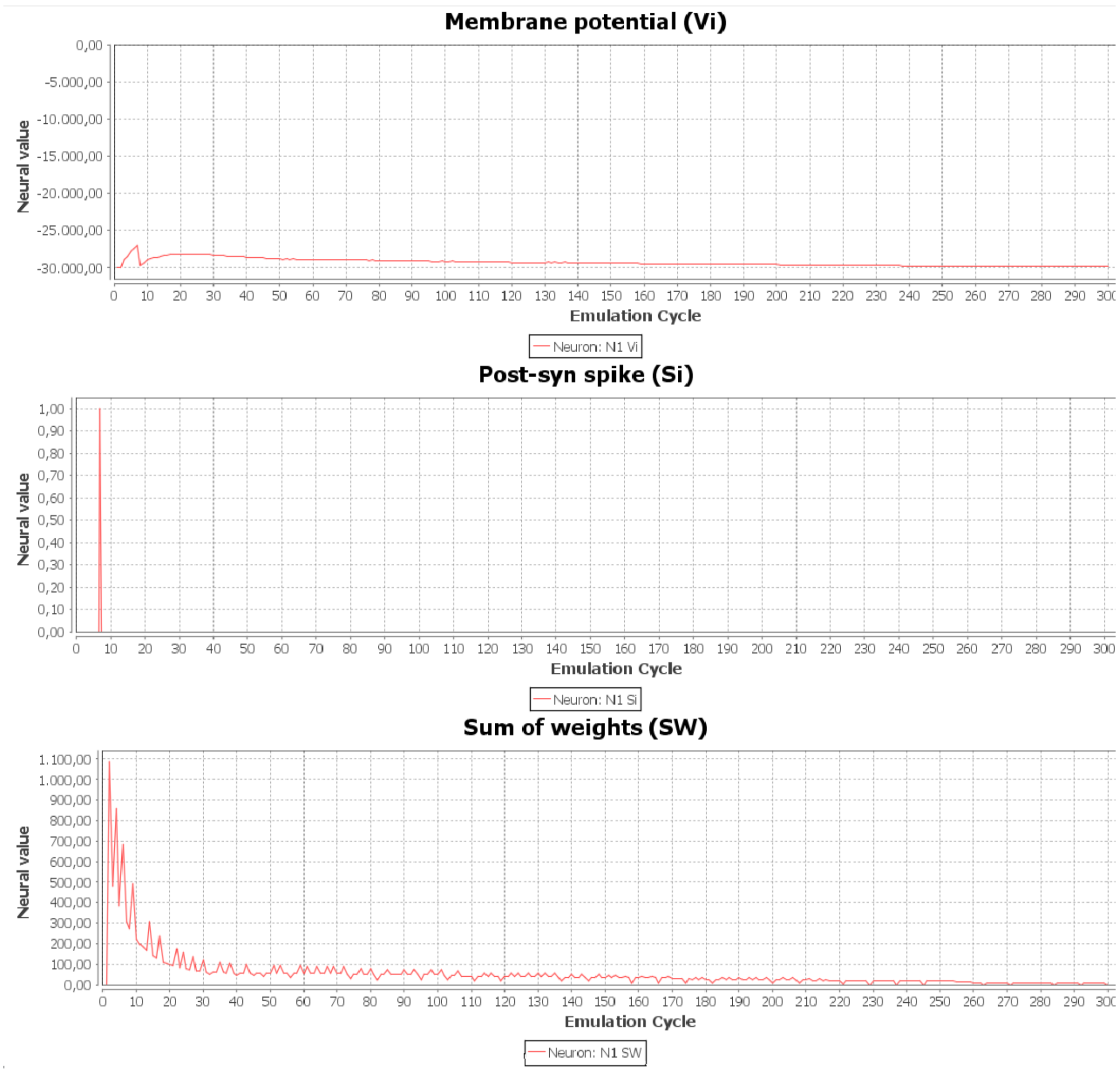


Figure D.6: Neural waveform for Neuron1

By looking at the process happening in the synapse of neuron 1 is similar to the reservoir equations discussed earlier (see Fig. D.7). As can seen from this graph, the behavior of M is almost complementary to the behavior of C. During the presence of the spike the value of M is completely transferred to C. The value of R gradually decreases, but at some cycles this value increases due to the contribution of reuptake process from C to R.

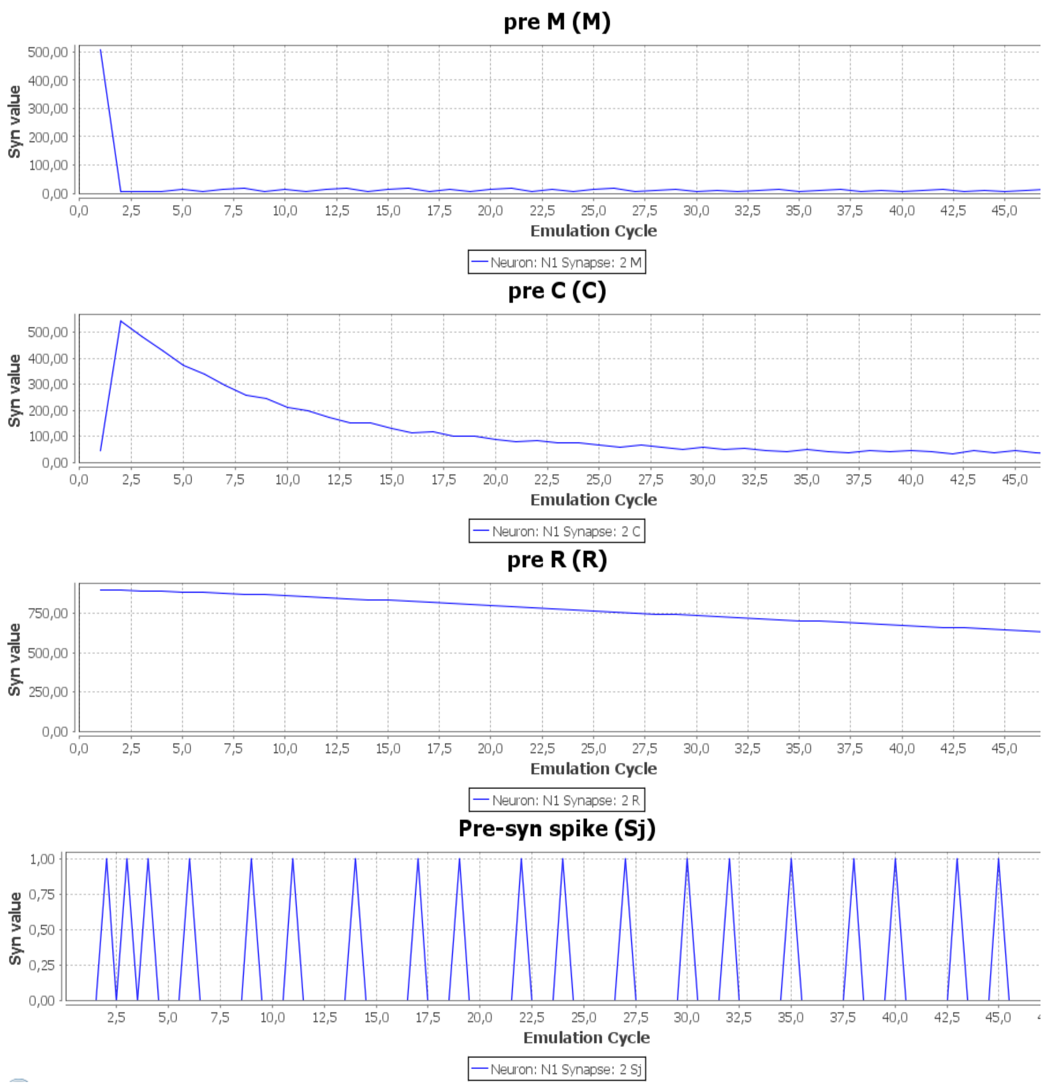


Figure D.7: Synaptic Waveform for Neuron 1

D.1.3 Conclusion

The onset detection was implemented for a single sensitivity level and for 4 different frequencies of which belong to the audible range. The results were analysed and compared with the theoretical explanations given in [1]. This work can be made more real time by replacing the look up tables with gammatone filter bank. The value of the reservoir model parameters were restricted to certain number of bits hence in this application the resource constraints were taken into account thereby reducing the range of values the system can operate on. As a future work one can extend the architecture to support floating point operations as well as increase the amount of resources. Moreover, once the number of frequency bands increase the entire topology needs to be redesigned from the beginning manually, in order to improvise this, an application could be created in order to automatically design the entire topology given the basic amount of information about the topology.

References

[1] Smith, L.S.; Fraser, D.S., "Robust sound onset detection using leaky integrate-and-fire neurons with depressing synapses", Neural Networks, IEEE Transactions on, vol. 15, no.5, pp. 1125-1134, Sept.2004.

[2] M. J. Newton and L. Smith, "A neurally inspired musical instrument classification system based upon the sound onset", J. Acoust. Soc. Am. 131, pp. 4785–4798, 2012.

[3] Sanjana Sekar, "Bio inspired Application Using SNAVA Architecture", Bachelor Thesis, SASTRA University and Universitat Politècnica de Catalunya, April 2014.

Spike coder

This annexure presents the spike coder structure which generates spikes under the same principle of operation like the analogue coders [1]. The VHDL code is provided in this Annexure.

E.1 Spike coder description

Figure E.1 shows the datapath of the spike coder. The spike coder is composed by three components which are: a Flip-Flop, a conditional subtractor, and a comparator. The mechanism of the spike coder is as follows: the spike coder receives the signal, which can be generated artificially by the values stores in LUTs, or the signal is provided by the ADC converter. The conditional subtractor only executes the subtraction operation only when the comparator indicates the difference between the signal sin and sin_in_aux_delayed crosses a certain threshold.

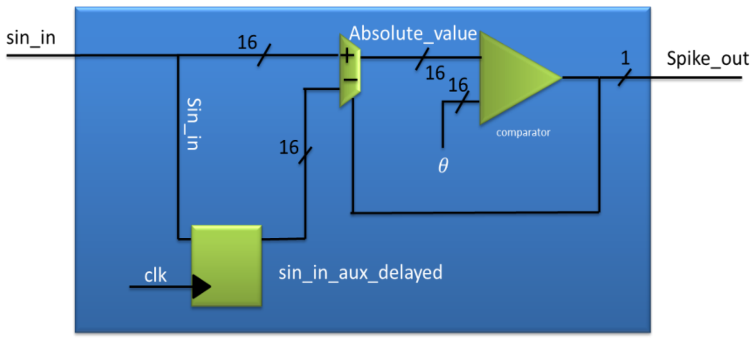


Figure E.1: Spike coder datapath

E.2 VHDL code

```
entity spike_generator is
   Port    (
                                                            clock        : in  STD_LOGIC;
                                                            rst          : in  STD_LOGIC;
                                                            enable       : in  STD_LOGIC;
                                                            sin_in       : in  STD_LOGIC_VECTOR(15 DOWNTO 0);
                                                            Spike_out    : out STD_LOGIC
```

```
                    );
end spike_generator;

architecture Behavioral of spike_generator is



    signal      sin_in_aux_delayed      : std_logic_vector(15 downto 0) := (others => '0');
    constant    threshold               :    std_logic_vector(15    downto    0)    :=
"0000100001110011";
    signal      enable_comparator             : std_logic := '0';
    signal      absolute_value          : std_logic_vector(7 downto 0) := (others => '0');


    begin


 process (clock)
    begin
      if (rising_edge(clock)) then
            if (rst = '1') then
                sin_in_aux_delayed <= (others => '0');
                    elsif enable = '1' and enable_comparator = '1' then
                    sin_in_aux_delayed <= sin_in;
                end if;
      end if;
    end process product_delayed;

 addition: process (clock)
    begin
      if (rising_edge(clock)) then
            if (rst = '1')  or (enable_comparator = '1') then
                            absolute_value <= (others => '0');
                            elsif enable = '1' then
                                absolute_value     <=     conv_std_logic_vector(abs(
(CONV_INTEGER (sin_in))-(CONV_INTEGER (sin_in_aux_delayed)) ),8);
                end if;
          end if;
    end process addition;

    Spike_out <= '1'   when (absolute_value >= threshold)          else '0';

    enable_comparator <= '1'  when  (absolute_value >= threshold) else '0';

end Behavioral;
```

References

[1] Gouveia, L.C., T.J. Koickal, and A. Hamilton, *"An asynchronous spike event coding scheme for programmable analog array"*, in *Circuits and Systems, 2011, ISCAS 2011, IEEE International Symposium on*, pp. 791-799, 2011.

List of publications and conferences

Giovanny Sánchez, Thomas Jacob Koickal, Athul Sripad, Luiz Carlos Gouveia, Alister Hamilton and Jordi Madrenas, Spike-Based Analog-Digital Sensor Information Processing System for Neuromorphic Applications, The IEEE International Symposium on Circuits and Systems (ISCAS), Beijing, China, pp. 1624-1627, 2013. (CONGRESO NOTABLE)

Adam Sokolnicki, **Giovanny Sánchez**, Jordi Madrenas, Manuel Moreno y Bartosz Sakowicz, Graphical representation of data for a multiprocessor array emulating spiking neural networks, Przegląd Elektrotechniczny, vol. R. 88, nr 11a, pp. 332-336, 2012.

Jordi Madrenas, Daniel Fernández, Jordi Cosp, J. Manuel Moreno, Luis Martínez-Alvarado and **Giovanny Sánchez,** bio-inspired sensory integration for environment-perception embedded systems, International Conference on Biomedical Electronics and Devices, Rome, Italy, pp. 260-267, 2011.

Giovanny Sánchez, Jordi Madrenas and J. Manuel Moreno, Performance evaluation and scaling of a multiprocessor architecture emulating complex SNN algorithms, 9th International Conference on Evolvable Systems - From Biology to Hardware, from Biology to Hardware 6th-8th September 2010, St Williams College, York, UK, pp. 145-156, 2010.