UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

IMPLEMENTATION OF A MULTIPROCESSOR ARRAY FOR SPIKING NEURAL NETWORK EMULATION ON FPGA

A Degree Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Sergi Juan Moreno

In partial fulfilment
of the requirements for the degree in
ELECTRONIC SYSTEMS ENGINEERING

Advisors:
Mireya Zapata Rodríguez
Jordi Madrenas Boadas

Barcelona, January 2017

Abstract

During this project I have described and simulated an Array of Multiprocessors to emulate Spiking Neural Networks in a new architecture. Finally I have synthesized and implemented it on a FPGA to check the correct behaviour.

The main described block is the Process Element, which contains several elements that allow the processing of the neuronal algorithm. Inside we find the ALU, which allows to perform calculations, a bank of registers to store values, a LFSR to generate pseudorandom numbers, data memory to store parameters and memories, which allow a dynamic interaction between Processing Elements.

Each element connects within a parameterizable array with the necessary hardware for the complete functionality of all of the instruction set. Finally it has been synthesized and modified to meet the time constraints, achieving to generate an array of 12x12 with problem-free timings.

Resum

Durant aquest projecte he descrit i simulat un Array de Multiprocessadors per emular Spiking Neural Networks en una nova arquitectura. Finalment ho he sintetitzat per implementar-ho sobre una FPGA i comprovar el correcte funcionament.

Els principal bloc descrit és l'Element de Processament, el qual conté diversos elements que permeten processar l'algoritme neuronal. A dins trobem l'ALU, que permet realitzar càlculs, un banc de registres per desar valors, un LFSR per generar números pseudoaleatoris, la memòria de dades per guardar paràmetres i les memòries associatives, que permeten una interconnexió dinàmica entre elements de processament.

Cada element es connecta dins d'un array parametritzable amb el hardware necessari per a la completa funcionalitat del l'arquitectura. S'han realitzat simulacions i depurat d'errors fins aconseguir un correcte funcionament de tot el set d'instruccions. Finalment s'ha sintetitzat i s'ha modificat per complir les restriccions temporals, aconseguint generar un array de 12x12 PEs sense problemes temporals.

Resumen

Durante este proyecto he descrito y simulado un array de Multiprocesadores para emular Spiking Neural Networks en una nueva arquitectura. Finalmente lo he sintetizado para implementarlo sobre una FPGA y comprobar el correcto funcionamiento.

El principal bloque descrito es el Elemento de Procesado, que contiene diversos elementos que permiten procesar el algoritmo neuronal. Dentro encontramos la ALU, que permite realizar cálculos, un banco de registros para guardar valores, un LFSR para generar números pseudoaleatorios, la memoria de datos para guardar parámetros y las memorias asociativas, que permiten una interconexión dinámica entre elementos de procesado.

Cada elemento se conecta dentro de un array parametrizable con el harware necesario para la completa funcionalidad de todo el set de instrucciones. Finalmente se ha sintetizado y se ha modificado para cumplir las restricciones temporales, conseguiendo generar un array de 12x12 sin problemas temporales.

I want to dedicate this project and thank my parents, Linda and Lluis, for their encouragement. They always supported me during this time and gave me forces to continue on.

Acknowledgements

This work would not have been possible without the inestimable and unconditional help of my supervisors: Mireya Zapata and Jordi Madrenas. Their advice and guidance, all the technical support and their immense experience have taught and guided me constantly and faithfully to the end. Thank you very much.

I also want to thank the work done by the group of the DSP-FPGA course, Llorenç Manonelles, Javier Pérez and David Torres. Thanks for choosing final work of that course I decided I would do this project about SNN.

Finally, I want to thank Jordi Vives for encouragement and help me with the document in LaTeX.

Revision history and aproval record

Revision	Date	Purpose
0	19/11/2016	Document creation
1	19/12/2016	Document revision
2	09/01/2017	Document revision
3	13/01/2017	Document revision

Document distribution list

Name	E-mail
Sergi Juan Moreno	sergijuan93@gmail.com
Mireya Zapata Rodríguez	mireya.zapata@upc.edu
Jordi Madrenas Boadas	jordi.madrenas@upc.edu

Written by:		Reviewed and approved by:	
Date	15/01/2017	Date	15/01/2017
Name	Sergi Juan	Name	Mireya Zapata Jordi Madrenas
Position	Project author	Position	Project supervisors

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this project is to develop in VHDL an evolvable, modular and scalable bio-inspired architecture. It will be able to emulate dynamic Spiking Neural Networks (SNN) using an array of Processing Elements (PEs) with a Single Instruction Multiple Data (SIMD) processing scheme. The project is carried out at the Advanced Hardware Architectures (AHA) Research Group of the Electronics Engineering Department.

1.1 Goals of the project

Describe and implement a resizeable array of PE with the capability of computing neural and synaptic algorithms using instructions received from a single Control Unit (Sequencer).

Create the proper interface between the Multiprocessor Array and the Sequencer to allow the correct communication and flow control of the algorithm computed in each PE.

Implement all the new operation codes (opcodes) at the PE and ALU components to achieve a full functionality of all the system.

Include an associative synaptic memory of Block RAMs and a spike register in each processor to optimize area resources, implement the virtualization capability and support dynamic connectivity.

Synthesize the final design and implement it on a Xilinx Kintex-7 FPGA KC705 Evaluation Kit (see figure 1.1).

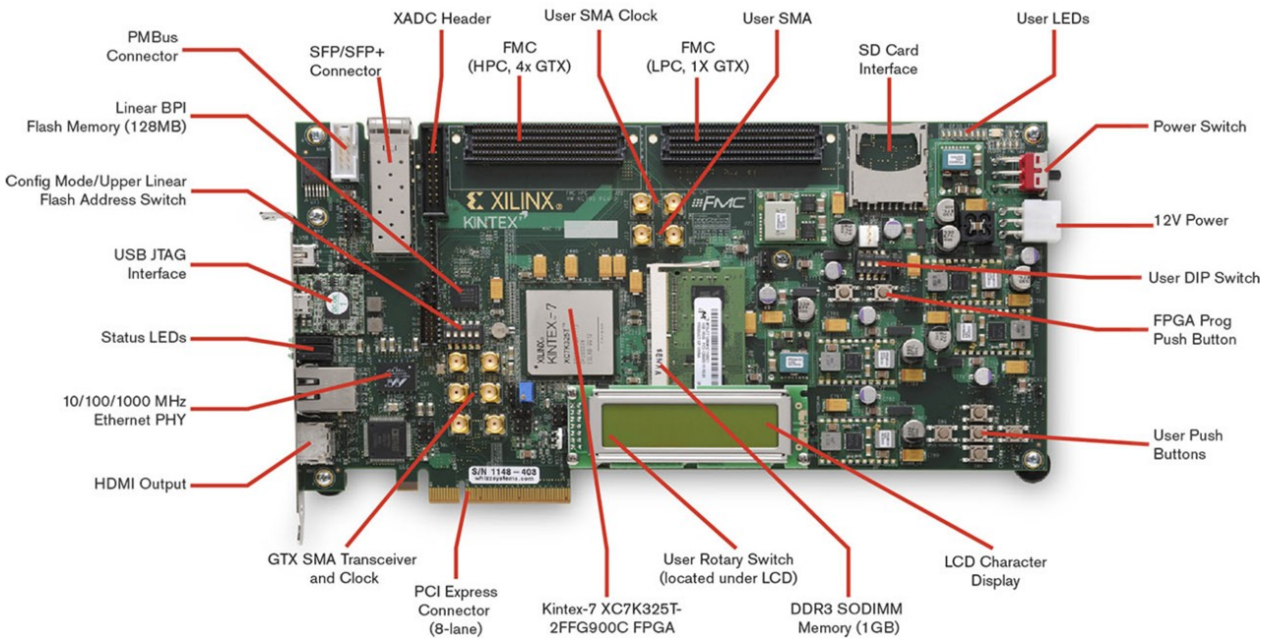


Figure 1.1: Xilinx Kintex-7 FPGA KC705 Evaluation Kit. Source: Xilinx website

1.2 Requirements and specifications of the project

This project starts from a previous architecture called SNAVA, described in XX. In this project, a more efficient Processing Element array in terms of functionality and resource occupancy is pursued. This array forms part of the new architecture called HEENS (”Hardware Emulator of Evolvable Neural Systems”).

The new architecture, Hardware Evolved Emulator Neural System (HEENS), will be versatile and it will allow us to load and run via software different models of neurons and change their synaptic interconnection dynamically without resynthesizing the project.

Resizeable array (number of PE and all the necessary connections for proper operation) based on only two parameters, number of rows and columns, with a range of 1 to 31. Depending on the resources of the FPGA, it will be possible to implement arrays with different sizes.

Full modular architecture allows the virtualization of PE up to seven neurons in addition of the main layer. This means that if we have a 10 x 10 array, this neural network will be able to emulate up to 800 neurons. Also, each PE supports up to 100 local synapses and 32 global synapses (coming form external multiprocessors)

1.3 Background

The study of neural networks and the brain has fascinated many people. Over time models have been developed to simulate the neuronal processes which occur in a biological brain. There have been more and more realistic architectures that help us to understand biological processes, such as make decisions and visual recognition, and facilitate the study of neural computation.

SNAVA is the starting point of the new HEENS architecture. It has been developed in the previous years by the Advanced Hardware Architecture group of the Department of Electronics Engineering. Both supervisors, Mireya Zapata and Jordi Madrenas, gave me the main initial ideas and first versions of the rest of the components to start with the Multiprocessor Array implementation. This fact implies that the coordination between all parts is essential to achieve a functional system.

This new architecture includes some new capabilities of synaptic interconnections, also, a better utilization of FPGA area resources thanks to new Associative Memories developed by my supervisors. The part of the new architecture related with my project started with some preliminary work carried out during the final project of the DSP-FPGA course, in the last semester. In that project some VHDL components began to be developed.

1.4 Work plan

The Multiprocessor Array project consists of two stages: VHDL description and simulation with QuestaSIM and implementation with Vivado. In both parts it is necessary to run simulations and check the correct functioning of the components described. After both parts is delivered the correspondent milestone.

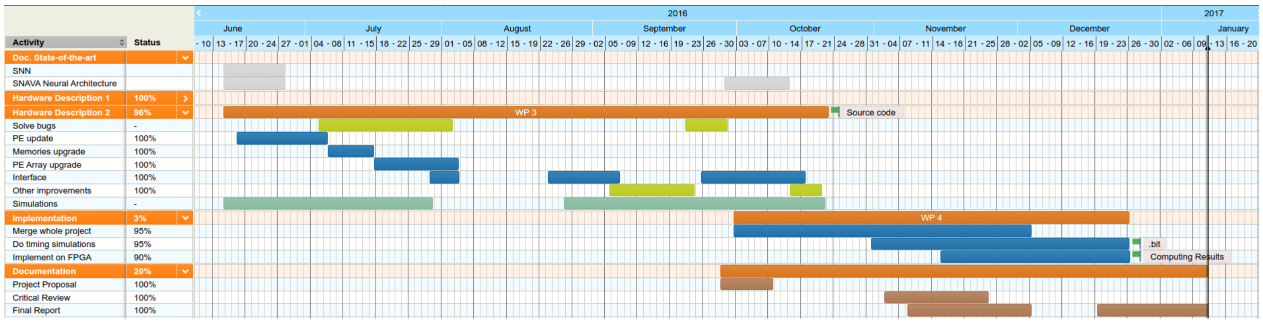
The table 1.1 shows the milestones, with the expected deliverable date and the real date, of the two stages plus the final test with the HEENS architecture loaded on the FPGA. Below are located the breakdown structure with the Work Packages listed (table 1.2) and the Gantt diagram with the main tasks distributed along the semester (figure 1.3).

Short title	Milestone/Deliverable	Exp. date	Real date
VHDL Description	Compiled source code	14/10/2016	04/11/2016
Successful implementation	.bit file	11/11/2016	23/12/2016
Testing with simple Integrate and Fire algorithm	Computing results	05/12/2016	09/01/2017

Table 1.1: Milestones of the project

WP 1	Doc. State-of-the-art
WP 2	Hardware description 1
WP 3	Hardware description 2
WP 4	Implementation
WP 5	Documentation

Figure 1.2: Work Breakdown Structure



Activity	Status
Doc. State-of-the-art	
SNN	
SNAVA Neural Architecture	
Hardware Description 1	100%
Hardware Description 2	96%
Solve bugs	-
PE update	100%
Memories upgrade	100%
PE Array upgrade	100%
Interface	100%
Other improvements	100%
Simulations	-
Implementation	3%
Merge whole project	95%
Do timing simulations	95%
Implement on FPGA	90%
Documentation	20%
Project Proposal	100%
Critical Review	100%
Final Report	100%

Figure 1.3: Gantt diagram

Chapter 2

State of the art

Neural networks are studied since many years ago, not only for understanding the biological functioning of the brain itself and their learning mechanisms, but also to create systems that can recognize objects, take decisions, learn and interact with objects and humans.

2.1 Neural networks

A biological neural network is characterized by millions of cells interconnected between them. Each cell, called neuron, has three differentiate parts (see figure 2.1): dendrites (input connections), a cell body with the nucleus, and the axon (output connections). The electrochemical impulse from one neuron is propagated, through the axon, to more than a hundred of neurons up to ten thousand, by connections called synapses.

An Artificial Neural Network (ANN) is a set of interconnected elements in a similar way as neurons in a biological brain. The study of these architectures allows us to start to understand how brain carry out some process, such as decision, visual recognition or remembering things.

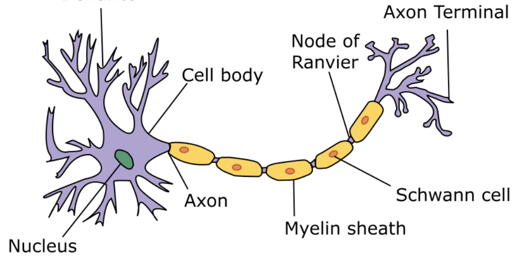


Figure 2.1: Neuron parts. Source: ”Anatomy and Physiology” by the US National Cancer Institute’s Surveillance

We can differentiate three generations of ANNs:

- **First Generation**:
  The first generation is based on McCulloch-Pitts [1] neurons as computational elements. In terms of processing, each element generates only digital outputs from digital inputs also from other elements or from the outside. In this generation are entering threshold circuit models, Boltzmann Machines [2] and Hopfield Nets [3].

- **Second Generation**:
  The second generation is characterized by an activation function used to generate continuous output, therefore, work with analog signals that determine the output value of the subsequent neurons. An example are sigmoid function ($\sigma(y) = \frac{1}{(1+e^{-y})}$) and saturated linear function π ($\pi(y) = y$ for $0 \leq y \leq 1$, $\pi(y) = 0$ for $y < 0$ and $\pi(y) = 1$ for $y > 1$). With thresholds and combinational circuits, these processing units (analog interface) can generate boolean answers and interact with units of the first generation (digital interface).

- **Third Generation**:
  Finally we have the third generation, the Spiking Neural Networks (SNNs), much closer to real biological neurons. This is the model used in this project and it is described in the following. [4]

2.2 Spiking Neural Networks

SNNs are the third generation of Neural Networks, as indicated before. They are characterized by incorporating the time concept in operation model. In contrast to other models, each neuron generates a unique spike (pulse) only when its membrane potential reaches a given threshold and not transmits data in every processing cycle. This feature makes the SNN model more realistic and similar to brain mechanism, allowing us to study better how the learning process works.

2.3 Implementations of SNN

Inside the category of SNN, there are different architectures in order of the number of neurons and synapses, algorithms, and the hardware where these are emulated. There are realistic models to emulate the neuron behaviour that require complex processor units to carry out the algorithms. On the other hand, there are models with a lighter computational load, like

”integrate an fire” model, which can be implemented on small chips with a lot of simple neurons with a fixed number of synapses. An example of each are:

SpiNNaker is a low-power parallel neuromorphic supercomputer. It is developed at the University of Manchester in the UK. The system consists of more than 60 thousand processors with 18 cores each one, reaching more than a million cores in total. Each processor has the capacity to addresses other neurons to form synapses. [5]
Our approach is similar to Spinnaker, although we focus on much more compact processing elements and fast Address Event Representation (AER) for spike transmission.

TrueNorth, launched by IBM in 2011, has a 64x64 array of neurosynaptic cores with 256 neurons each one, reaching a million neurons in total. Every neuron is connected to another 256 neurons. This supports huge networks, but limitations arise in the oversimplified neural model and the connectivity problem.[6]

2.4 Spiking Neural-network Architecture for Versatile Applications

SNAVA is an evolution of a previous architecture, Ubichip, which was developed during the Perplexus project, an IST-FET FP6 European research project. With SNAVA, substantial improvements were achieved, for example, a greater number of PEs, virtualized neurons, or the ability to vary the number of synapses independently for each neuron[1].

This architecture is characterized by a scalable and resizeable implementation which can allow us to emulate Spiking Neural Networks (SNN). Being inspired by the mechanisms of brain biology, SNAVA facilitates an experimental approach to the study of neural dynamics.

It consists of a Multiprocessor Array with Single Instruction Multiple Data (SIMD) units, or Processing Elements (see figure 2.2). The processor was modified from the previous architecture and, among other improvements, was added the capability of emulate more than one neuron thanks to virtualization concept. The array is controlled by the Sequencer, which sends the instructions of the algorithm loaded in the Instruction BRAM. The Address-Event Representation (AER) Module brings the capability of interconnect more FPGAs in a ring topology to increase the number of neurons emulated.

[1]A comparison between the two architectures can be found at the Ph.D. thesis ”Efficient multiprocessing architectures for Spiking Neural Networks emulation devices based on configurable”[7].

The SNAVA architecture had hardware non used from the previous version, this meant that its design was not optimal, occupying more resources and consuming more energy unnecessarily. The instruction set is used to program the neural algorithm by Assembler language. The array, although the dimensions of 10x10 PEs, did not have pipeline registers and this may cause sporadic bugs when signals didn't arrive on time. Synaptic connections were made by means of combinational logic, being forced to maintain a predefined topology. This produced very long synthesis times and inconvenience for connectivity changes.

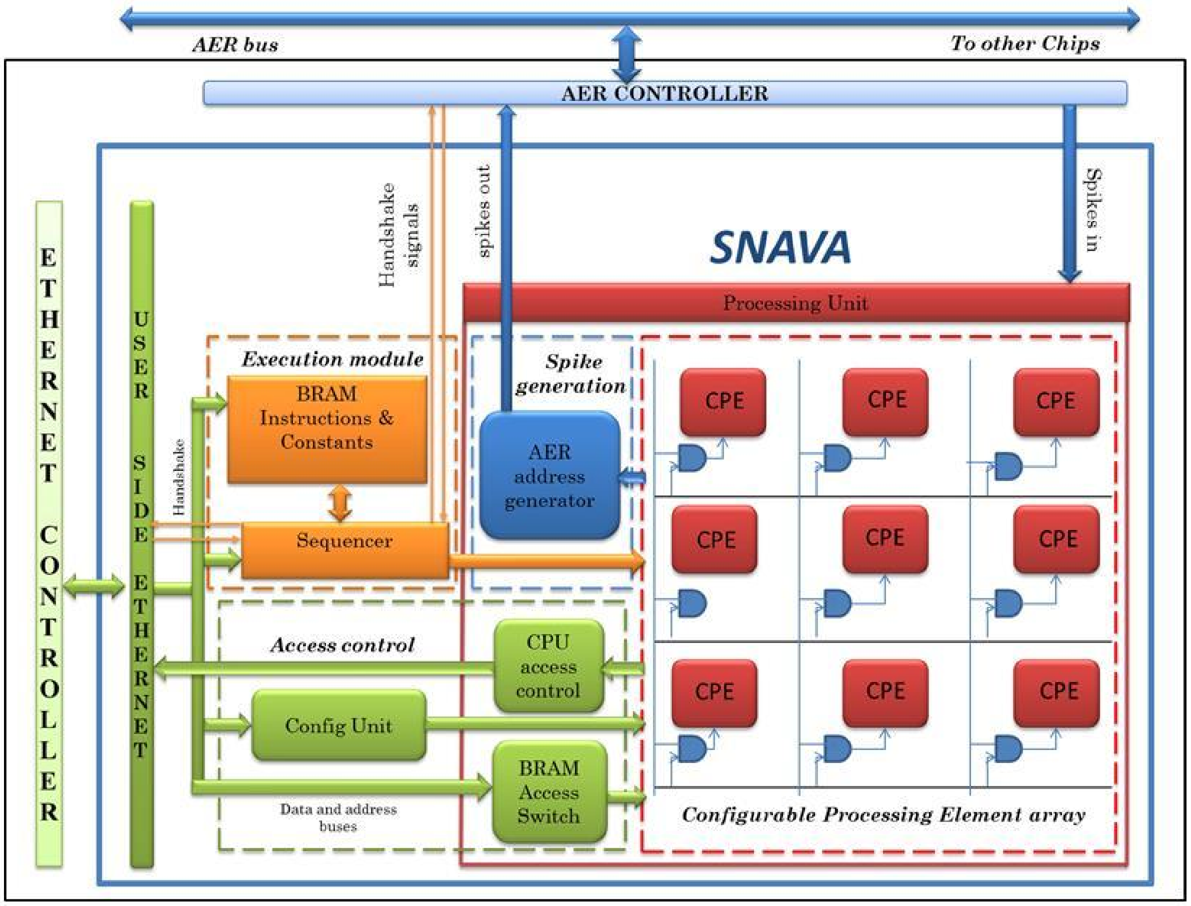


Figure 2.2: SNAVA architecture block diagram [7]

Chapter 3

Project development

This project is a part of the HEENS architecture, which is still being developed during this semester. The Multiprocessor Array project development is divided in two parts. The first one is focused on describing all components in VHDL and simulating the correct behaviour using QuestaSim. The second part is to implement these components doing the synthesis and the implementation using Vivado to obtain the ".bit" file to load and run it on the FPGA.

In contrast to SNAVA, the components of the new implemented architecture had been described again, which is optimized in concordance with the instruction set improved. The Associative Memory [8] allows dynamic synaptic connections and reduces a lot the time of synthesis. In addition, with a similar array dimensions, pipeline registers have been added to prevent slacks of setup (as discussed later).

3.1 VHDL description

The main goal of this part is to describe all components to generate the Multiprocessor Array with PEs and all the other components inside. Also, after each modification, it is necessary to check the correct logical operation of the whole system and test the set of instructions forcing extreme cases to detect possible bugs and unexpected results.

3.1.1 Components provided

The following components, provided by both supervisors, and the Multiprocessor Array implemented in this project (explained in detail at **3.1.2 Described Components** section) conforms the whole HEENS architecture. Each component is an evolution of the previous one, from SNAVA, in order to solve problems of resource utilization, optimize instruction set and improve the system scalability.

Top entity

As its name says, this component includes all the other entities and allows the connection between the whole system and the FPGA.

Sequencer

This is the controller of the whole system. It's in charge to read the algorithm loaded, manage the control signals and syncronize the Multiprocessor Array with the AER System.

Address-Event Representation (AER)

To make the system scalable it is necessary to manage spike communications between FPGA boards. The AER interface allows to transmit this spikes through a ring topology allowing the interconnection of 126 Kintex7 plus a master one, a PicoZed Z030.

Block Memories

In every PE also there are three memories implemented with Block RAMs to save the synapses with other neurons (associative memory) and input spikes (local and global memories). All of them are inside this component and are one of the improvements to achieve a bigger array than the 10x10 of the previous architecture.

3.1.2 Implemented components

In this project the following components are compiled and simulated using QuestaSIM. The waveform generated in each simulation is very useful to check the operation codes and the expected results for all the signals involved in the process. The components described in this project are:

Multiprocessor Array

It consists in n Multiprocessor Rows (each one with m Processing Elements inside). Is described using a *generate loop* with prefixed parameters; by this way the array can be resizeable (until fill all the FPGA) changing only the number of rows and columns from the definition package.

There are two communication buses, one for data and another one for memory address, to transmit data to all units and load configurations. The capability to activate only one PE by enable signals allows to preconfigure parameters of each neuron separately. Also, there are control signals to synchronize all the execution between PEs and the Sequencer. The combinational logic at the end of the rows and columns allows generated spikes to flow out from all the PEs.

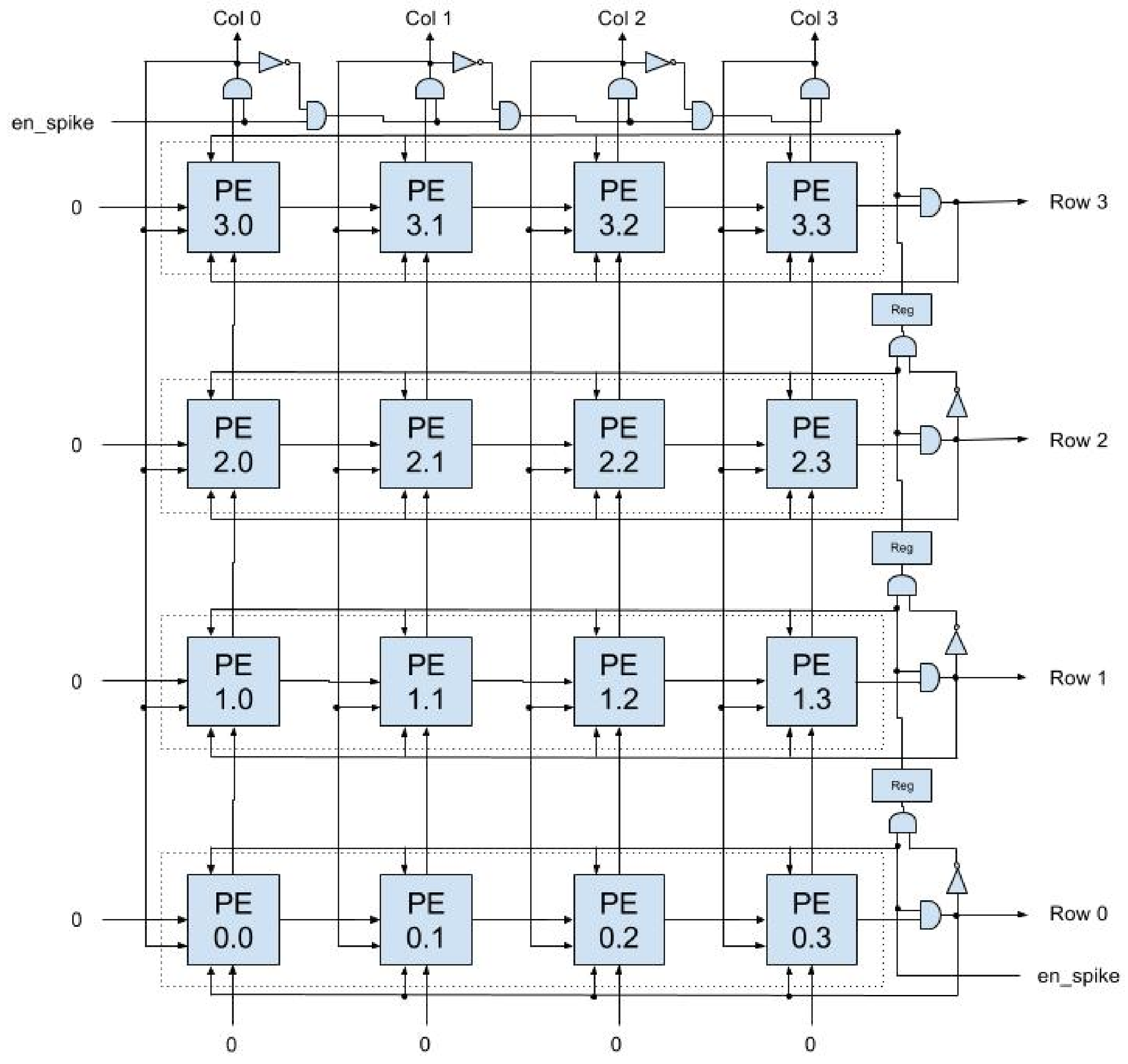


Figure 3.1: Multiprocessor Array block diagram

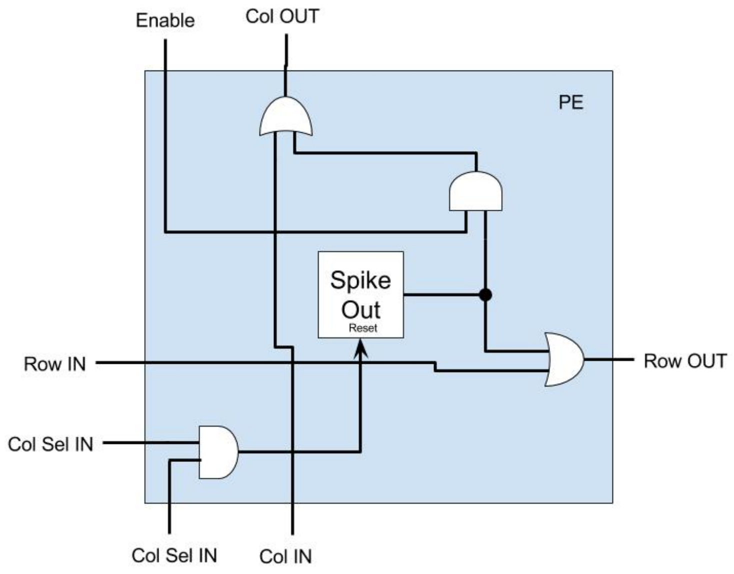


Figure 3.2: Read and Reset spike circuit located inside each PE

As shown in the figure 3.1, the blocks of the Multiprocessor Rows are expanded to see the complete structure with its connections.

The operation of the HEENS architecture is separated into four modes: initialization, configuration, execution and distribution. These modes are alternated throughout the process to perform the neuron network emulation.

Initialization takes place only at the beginning and is controlled by the Address-Event Representation (AER) system. In this mode, AER System sends a ChipID and the RingSize to all FPGAs. First one allows the identification of each node at the transmission bus, and the second value determines the total number of FPGAs connected. Instruction memory is also configured with the algorithm that will be executed at the Multiprocessor Array.

Setup mode is in charge to write the memories located inside the PEs. To do it, an address bus enables each processor individually to load the network topology definition and the neural parameters and synapses of the algorithm to be emulated.

Execution of the neural algorithm starts once the entire array is set. In this mode the Sequencer sends the instructions to all processors at the same time. All of them calculate the membrane potential in function of the input spikes, synapses and initial configuration. After each execution loop, the Sequencer enables the distribution signal. Then, spikes generated by each PE are processed and forwarded to the other processors to start once the loop. The first processor in the first row (row 0 and column 0) is the one with higher priority, so if it have a spike, will be the first to leave the array and reset (combinational circuit at figure 3.2). Then, the following processors of the same row will continue to transmit spikes when all the previous ones were cleaned. Once the entire first row is empty, the next one is enabled and the sequence starts again. This cycle is repeated to empty the multiple layers of virtualization and does not end until the sequencer disable the distribution signal. Figure 3.3 is an example of the complete process of spike distribution. At the simulation there are several signals involved, but the most significant are the addresses of the spikes of rows and columns (blue) signal with spikes codified (green) and control signals (yellow), which allow an ordered output. The last ones enable a row each time and the processors with a spike to be distributed, one by one.

Distribution stage is when the spikes are distributed to the other FPGAs via bus AER, and then, to all the associative memories to emulate the synaptic behaviour of neurons.

Multiprocessor Row

This component implements every row of PEs inside the Array. It is generated, also by *generate loop*, in a new instantiation to make easier the utilization of pipeline registers to manage time slacks afterwards. It also depends on the parameters of the package definitions, allowing to resize the number of columns changing only the correspondent value.

Inside this component there are the m PEs with corresponding connections to configure, send data and read generated spikes. During the implementation process, due to the dimensions and the complexity of a 12x12 array, pipeline registers were added to neutralize the slack time errors (see **3.2.2 Slack time of setup** section).

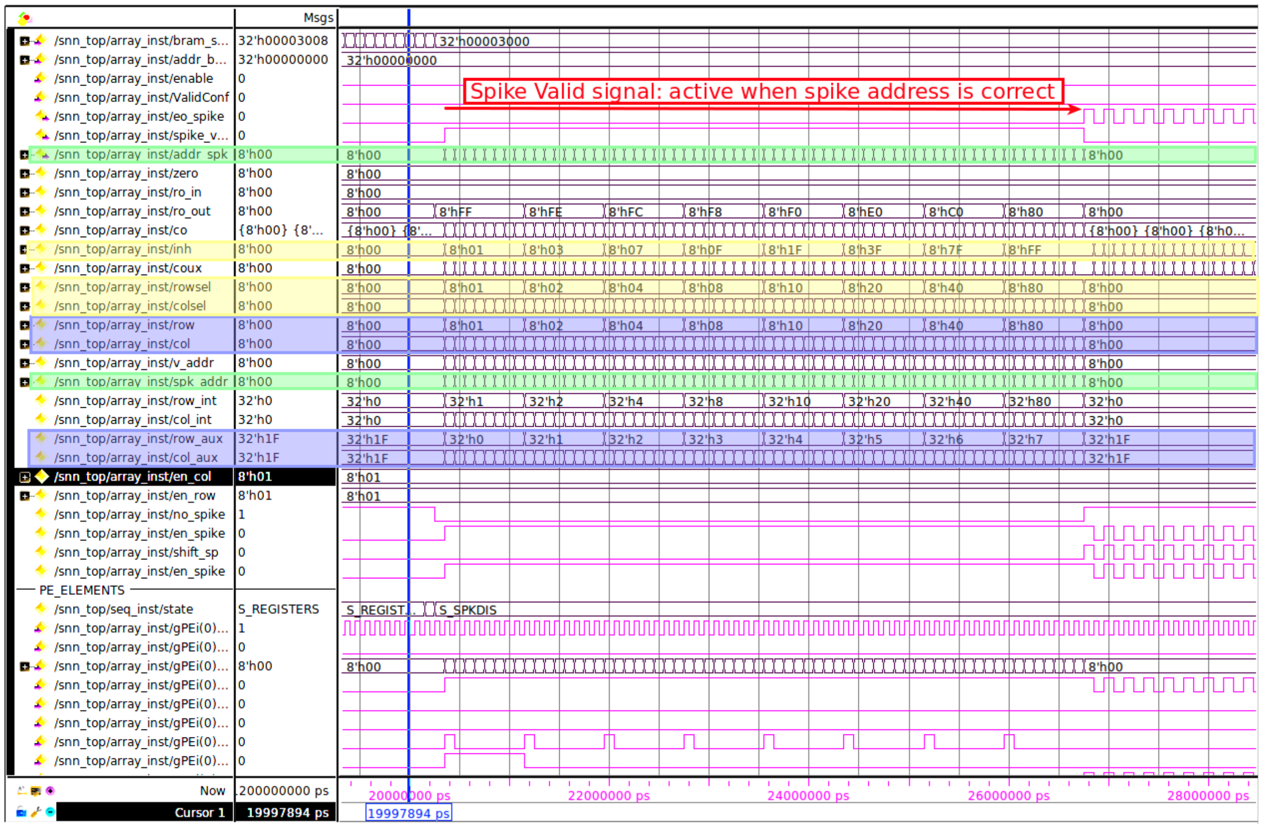

Msgs
Spike Valid signal: active when spike address is correct
/snn_top/array_inst/bram_s... 32'h00003008
/snn_top/array_inst/addr_b... 32'h00000000
/snn_top/array_inst/enable 0
/snn_top/array_inst/ValidConf 0
/snn_top/array_inst/eo_spike 0
/snn_top/array_inst/spike_v... 0
/snn_top/array_inst/addr_spk 8'h00
/snn_top/array_inst/zero 8'h00
/snn_top/array_inst/ro_in 8'h00
/snn_top/array_inst/ro_out 8'h00
/snn_top/array_inst/co {8'h00} {8'...
/snn_top/array_inst/inh 8'h00
/snn_top/array_inst/coux 8'h00
/snn_top/array_inst/rowsel 8'h00
/snn_top/array_inst/colsel 8'h00
/snn_top/array_inst/row 8'h00
/snn_top/array_inst/col 8'h00
/snn_top/array_inst/v_addr 8'h00
/snn_top/array_inst/spk_addr 8'h00
/snn_top/array_inst/row_int 32'h0
/snn_top/array_inst/col_int 32'h0
/snn_top/array_inst/row_aux 32'h1F
/snn_top/array_inst/col_aux 32'h1F
/snn_top/array_inst/en_col 8'h01
/snn_top/array_inst/en_row 8'h01
/snn_top/array_inst/no_spike 1
/snn_top/array_inst/en_spike 0
/snn_top/array_inst/shift_sp 0
/snn_top/array_inst/en_spike 0
PE_ELEMENTS
/snn_top/seq_inst/state S_REGISTERS
/snn_top/array_inst/gPEi(0)... 1
/snn_top/array_inst/gPEi(0)... 0
/snn_top/array_inst/gPEi(0)... 8'h00
/snn_top/array_inst/gPEi(0)... 0
/snn_top/array_inst/gPEi(0)... 0
/snn_top/array_inst/gPEi(0)... 0
/snn_top/array_inst/gPEi(0)... 0
/snn_top/array_inst/gPEi(0)... 0
Now 200000000 ps
Cursor 1 19997894 ps
20000000 ps
22000000 ps
24000000 ps
26000000 ps
28000000 ps
19997894 ps

Figure 3.3: Spike distribution of 8x8 Multiprocessor Array

Processing Element (PE)

Is the basic processing unit of the array. It is responsible for emulating the behaviour of a neuron and, if virtualitation is enabled, the same PE will emulate a maximum of 8 different neurons, each with own parameters and synapses.

As we see at figure 3.4, the internal components correspond to a Harvard type architecture: ALU, Registers, Memory and communication buses. The HEENS architecture is able to run several models of SNN and it have to compute various functions and algorithms needed to achieve an approximation to neurons behaviour.

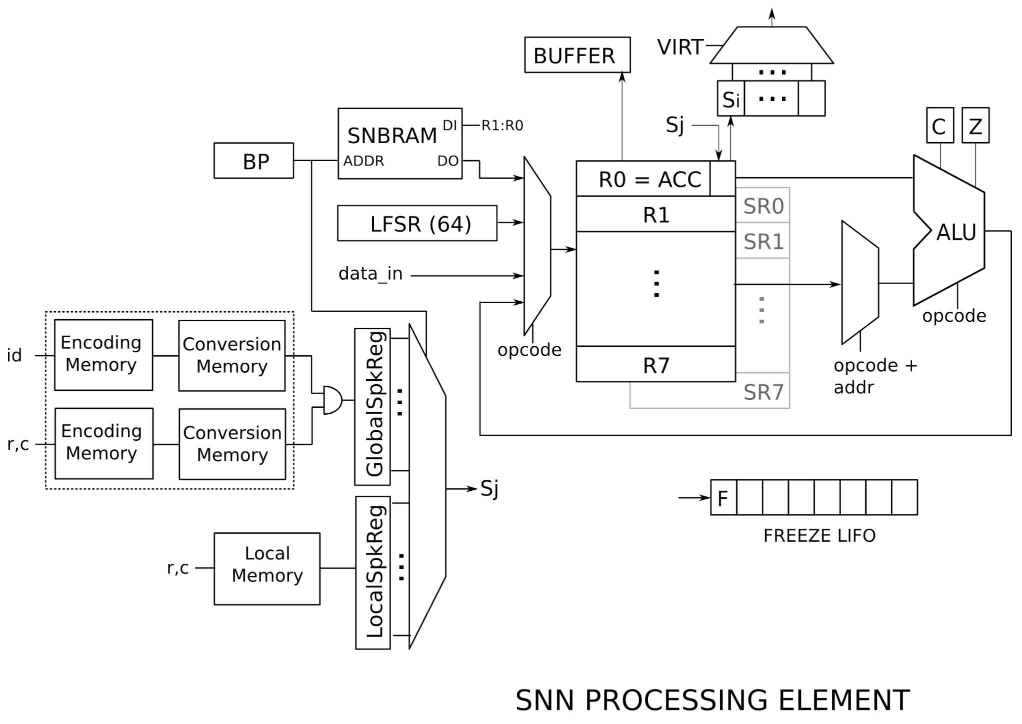


Figure 3.4: Processor Element block diagram

Arithmetic Logic Unit (ALU)

As its name indicates, this component carries out all the arithmetic operations (addition, subtraction and multiplication) and movements (right and left shifting, circular shifting, set to ones or zeros...) required to execute the algorithm.

Thanks to the parallelism allowed by the FPGA, it is possible to describe the component in order to calculate all the operations at the same time and then taking only the requested value by the opcode. This solution is optimal in terms of processing time but it increments considerably the energy consumption and the space used.

The figure 3.5 shows the space occupied (in white color) by all the components of one row of PEs. More specifically, all the connections of the opcode signal inside the ALU's combinational logic.

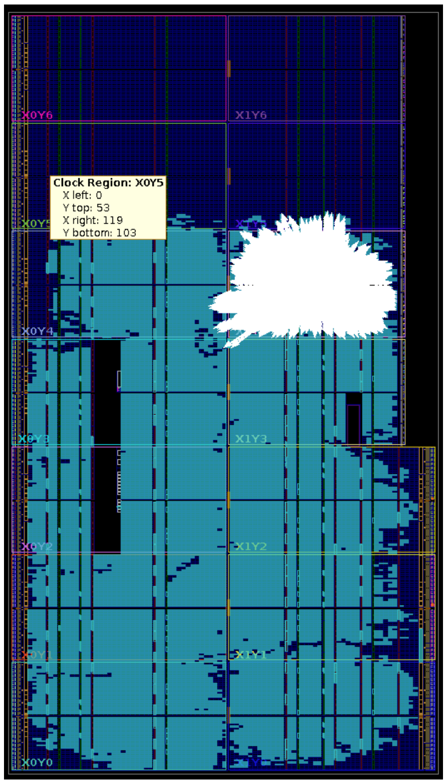


Figure 3.5: Floorplanning of a 10x10 array with an entire row in white color

Block Registers

Is a simple component with 16 registers (8 visible plus 8 shadowed registers) of 16 bits each one. The utility is similar to a cache memory of a personal computer: load and read values in function of the instructions received and the results of the ALU. The fact of having visible and shadowed registers allows us to load important values in the second ones and load the needed value on the correspondent visible registers to use it in a new execution loop.

LFSR

This component is implemented inside the PE. It is in charge of generate pseudorandom numbers by shifting its 64 registers. The functionality is to emulate neural noise.

RAMs

Is used to store the parameters of each emulated neuron and the seed to initialize the LFSR. Also are stored some processed values from the Block Register to load them later.

3.1.3 Simulations

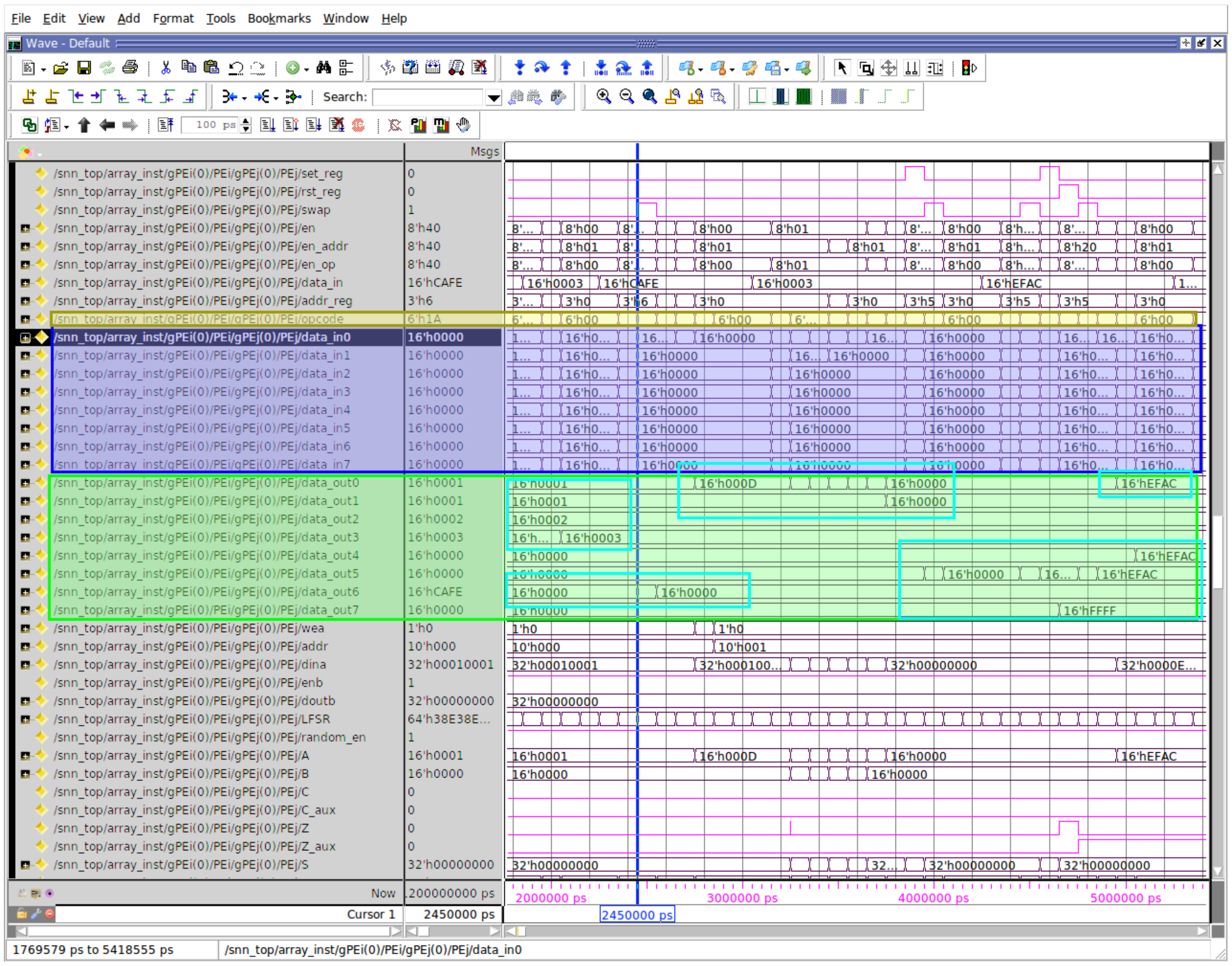
During the description and simulation phase, several changes were made to improve the functionality and solve issues detected. Most of them were related with the arithmetic operations of the ALU, improvements on the operation codes (opcodes) set and the freeze functionality[1].

All the programs executed by the Multiprocessor Array (neural algorithm and test programs) are written in Assembler and compiled to generate the data that will be load an executed (see appendices **B Integrate and fire algorithm** to an example of a simple neural model). Dummy codes can help to debug the components by simulating a controlled behaviour. Once the execution is finished, QuestaSIM can show a waveform diagram that helps to find the primary malfunction. To solve these issues is necessary to modify all the involved functions and adapt the other components to keep the consistency between the interconnected signals.

Registers behaviour test

An example of a waveform obtained by simulating the project with a testing code is the figure 3.6, which shows the value of the register signals after do a simulation. In function of the opcode value (yellow signal), the selected register (green signals) loads a value from inputs (blue signals), reset to 0 or set to 1 the selected bit and swap the value with the same shadow register.

[1]Set of opcodes that provide the capability to exclude neural algorithm sections, allowing the implementation of conditional statements of the emulated model. This is necessary because of the SIMD architecture employed, with many PEs dispatching the same instruction from a single sequencer.

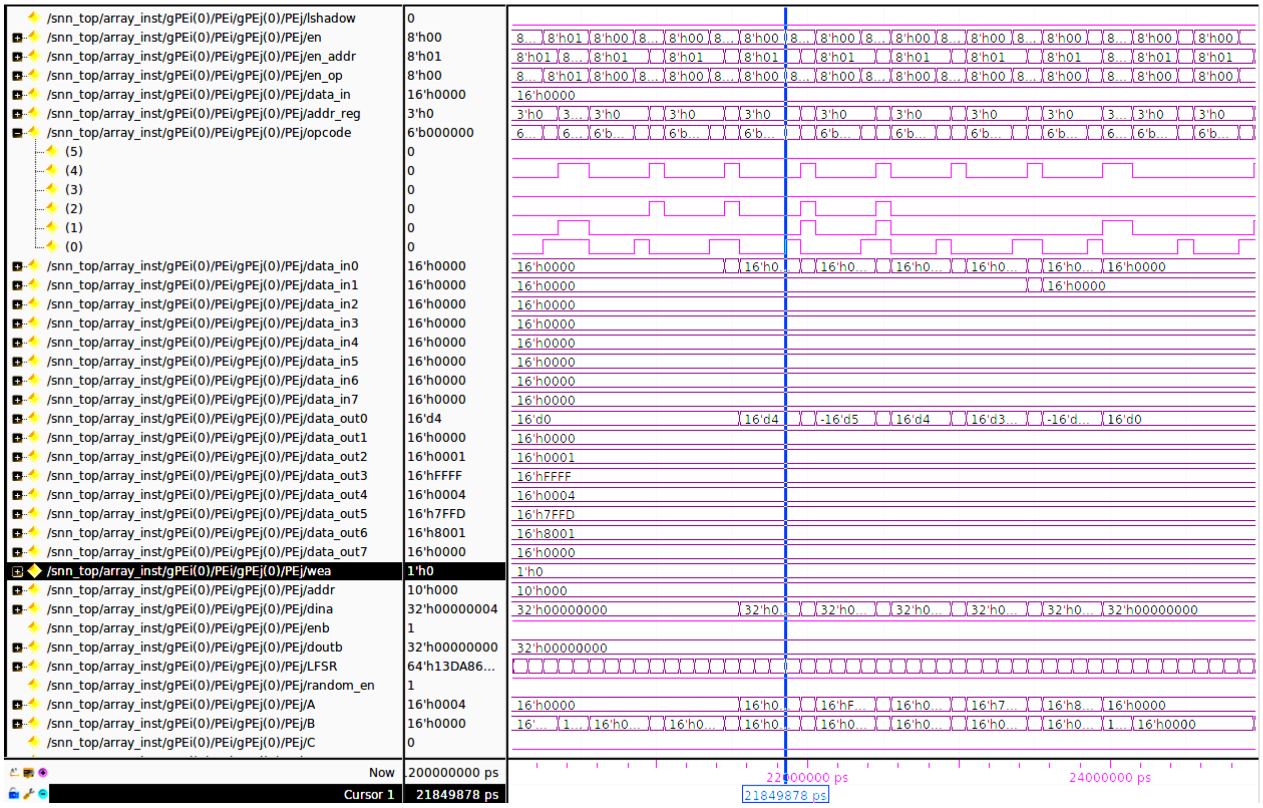


```
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/set_reg      0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/rst_reg      0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/swap         1
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en           8'h40
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en_addr      8'h40
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en_op        8'h40
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in      16'hCAFE
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/addr_reg     3'h6
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/opcode       6'h1A
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in0     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in1     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in2     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in3     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in4     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in5     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in6     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in7     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out0    16'h0001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out1    16'h0001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out2    16'h0002
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out3    16'h0003
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out4    16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out5    16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out6    16'hCAFE
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out7    16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/wea          1'h0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/addr         10'h000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/dina         32'h00010001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/enb          1
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/doutb        32'h00000000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/LFSR         64'h38E38E...
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/random_en    1
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/A            16'h0001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/B            16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/C            0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/C_aux        0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/Z            0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/Z_aux        0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/S            32'h00000000
                                              Now        200000000 ps
                                         Cursor 1        2450000 ps
1769579 ps to 5418555 ps     /snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in0
```

Figure 3.6: Waveform exemple of the Registers simulation executing the opcodes: LDALL, RST, SET and SWAP

ALU behaviour test

The ALU is one the most complex component of the system. It is necessary that always works well even in extreme cases. That is why were created several test codes to verify all operations related to this component (arithmetic, shift registers, comparison...). Figure 3.7 shows a sequence of a simulation using all arithmetic opcodes (ADD, SUB, MUL, MULS, AND, OR, INV and XOR) with different numbers to test the extreme cases.

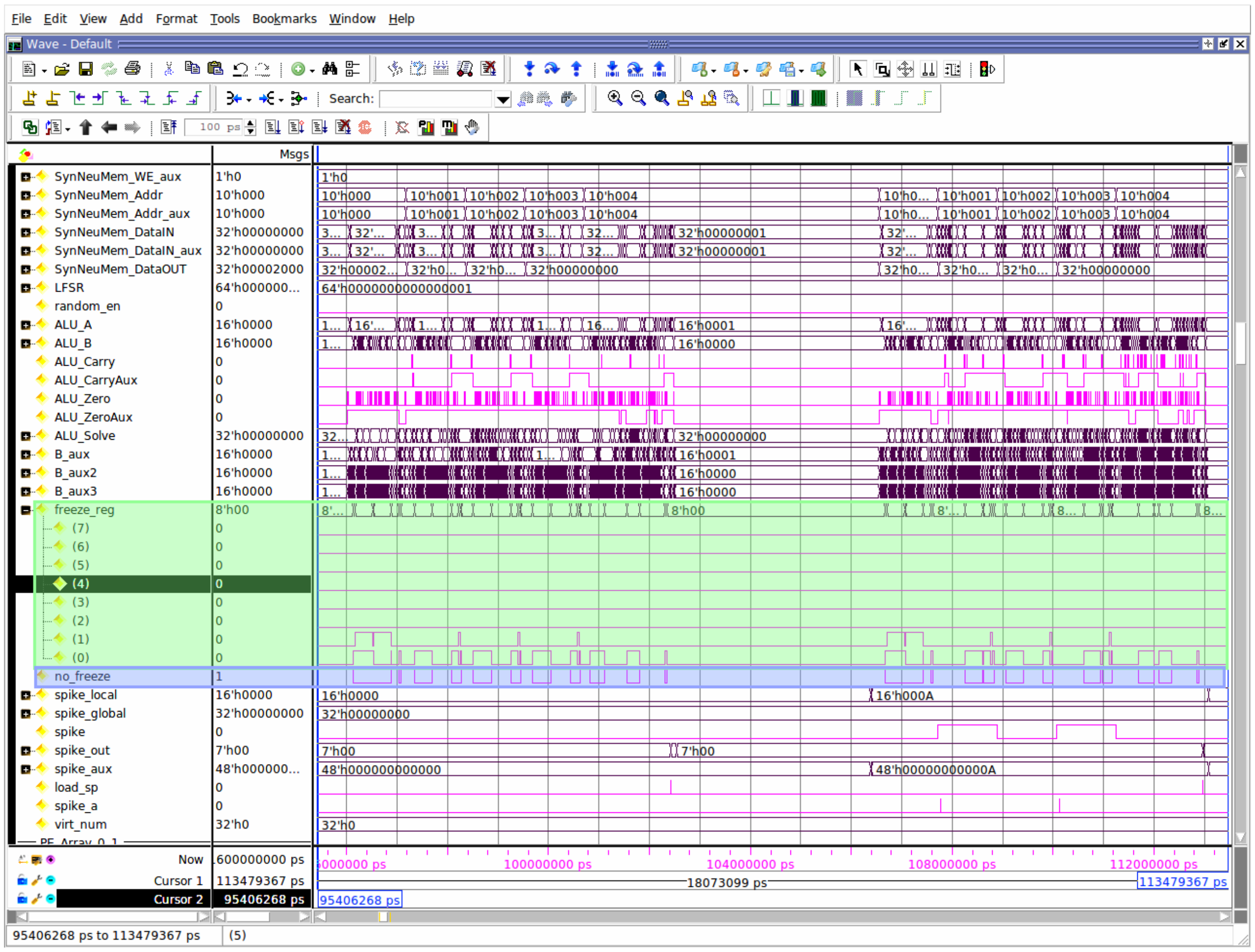


```
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/lshadow      0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en           8'h00
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en_addr      8'h01
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/en_op        8'h00
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in      16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/addr_reg     3'h0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/opcode       6'b000000
    (5)                                                  0
    (4)                                                  0
    (3)                                                  0
    (2)                                                  0
    (1)                                                  0
    (0)                                                  0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in0     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in1     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in2     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in3     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in4     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in5     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in6     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_in7     16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out0    16'd4
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out1    16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out2    16'h0001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out3    16'hFFFF
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out4    16'h0004
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out5    16'h7FFD
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out6    16'h8001
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/data_out7    16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/wea          1'h0
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/addr         10'h000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/dina         32'h00000004
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/enb          1
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/doutb        32'h00000000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/LFSR         64'h13DA86...
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/random_en    1
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/A            16'h0004
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/B            16'h0000
/snn_top/array_inst/gPEi(0)/PEi/gPEj(0)/PEj/C            0
                                              Now        200000000 ps
                                              Cursor 1   21849878 ps
```

Figure 3.7: Arithmetic opcodes test

Freeze functionality test

The operation of the freeze can be seen in figure 3.8. The image shows a detail of the freeze LIFO stack (green signals) and nofreeze binary value (blue signal) that indicates if any freeze condition has been met. The LIFO stack is where are keep all the condition results. If one of them is positive, the code between the "if" and "end if" should not been executed, so is pushed 1 in the stack. The registers are only enabled when there are all zeros in this stack, otherwise they remain disabled. With the opcode "UNFREEZE", the last value is pulled, doing the same as the "end if" in a condition function.



	Msgs
SynNeuMem_WE_aux	1'h0
SynNeuMem_Addr	10'h000
SynNeuMem_Addr_aux	10'h000
SynNeuMem_DataIN	32'h00000000
SynNeuMem_DataIN_aux	32'h00000000
SynNeuMem_DataOUT	32'h00002000
LFSR	64'h000000...
random_en	0
ALU_A	16'h0000
ALU_B	16'h0000
ALU_Carry	0
ALU_CarryAux	0
ALU_Zero	0
ALU_ZeroAux	0
ALU_Solve	32'h00000000
B_aux	16'h0000
B_aux2	16'h0000
B_aux3	16'h0000
freeze_reg	8'h00
(7)	0
(6)	0
(5)	0
(4)	0
(3)	0
(2)	0
(1)	0
(0)	0
no_freeze	1
spike_local	16'h0000
spike_global	32'h00000000
spike	0
spike_out	7'h00
spike_aux	48'h000000...
load_sp	0
spike_a	0
virt_num	32'h0
Now	600000000 ps
Cursor 1	113479367 ps
Cursor 2	95406268 ps

95406268 ps to 113479367 ps (5)

Figure 3.8: Integrate and fire algorithm simulation

3.2 Synthesis and Implementation

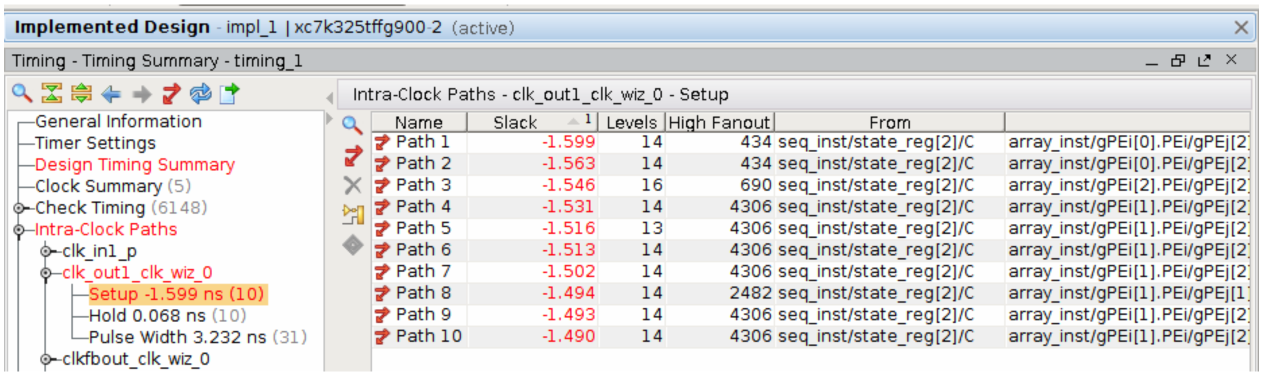
In order to program the FPGA with the .bit file of all the implementation, first of all it is necessary to include a clock block and modify some lines of the top component to prepare the interface between the FPGA and the whole system. Then, the project can be synthesized. During this process, Vivado checks the compliance with all the constrains in function of the board used and other issues related with time execution latches (not detected in the simulation with QuestaSIM). Also the utilization of the FPGA and its resources (Block RAM, Registers, LUTs...) are computed.

The next step is the implementation. This process requires a lot of time because the computer generates all the components to be placed on the FPGA, with all the connections. It takes into account the distance between registers and calculates the worst delay in a signal between two registers to determine if there are time problems. Due to the large amount of time it takes for the synthesis and implementation, the first tests should be done with an array of reduced dimensions. Tests carried out on the FPGA can reveal errors which do not appear in the simulations.

3.2.1 Main problems found

The problems to be solved in this part were related with time constrains. Due to the main characteristic of the VHDL hardware describing language, all the values (signals processed by combinational circuits and saved with registers) have a time dependence. In other words, if a signal (real wire between two registers) is too long across the FPGA or it depends of many cascaded combinational circuits (each logic component have an specific time of response), the time needed by the destination register to load the next value when the clock arrives is too short or negative (slack time of setup[2]), it may cause incorrect data capture and/or metastability and, therefore, computational problems.

Vivado detects critical paths of signals involved in opcode lines from the Sequencer to the memory registers, crossing the ALU and, in special, the multiplier module. Then, it shows the time constrain that is violated and the extra time required, as it is shown at figure 3.9.


Implemented Design - impl_1 | xc7k325tffg900-2 (active)

Timing - Timing Summary - timing_1

- General Information
- Timer Settings
- Design Timing Summary
- Clock Summary (5)
- Check Timing (6148)
- Intra-Clock Paths
  - clk_in1_p
  - clk_out1_clk_wiz_0
    - Setup -1.599 ns (10)
    - Hold 0.068 ns (10)
    - Pulse Width 3.232 ns (31)
  - clkfbout_clk_wiz_0

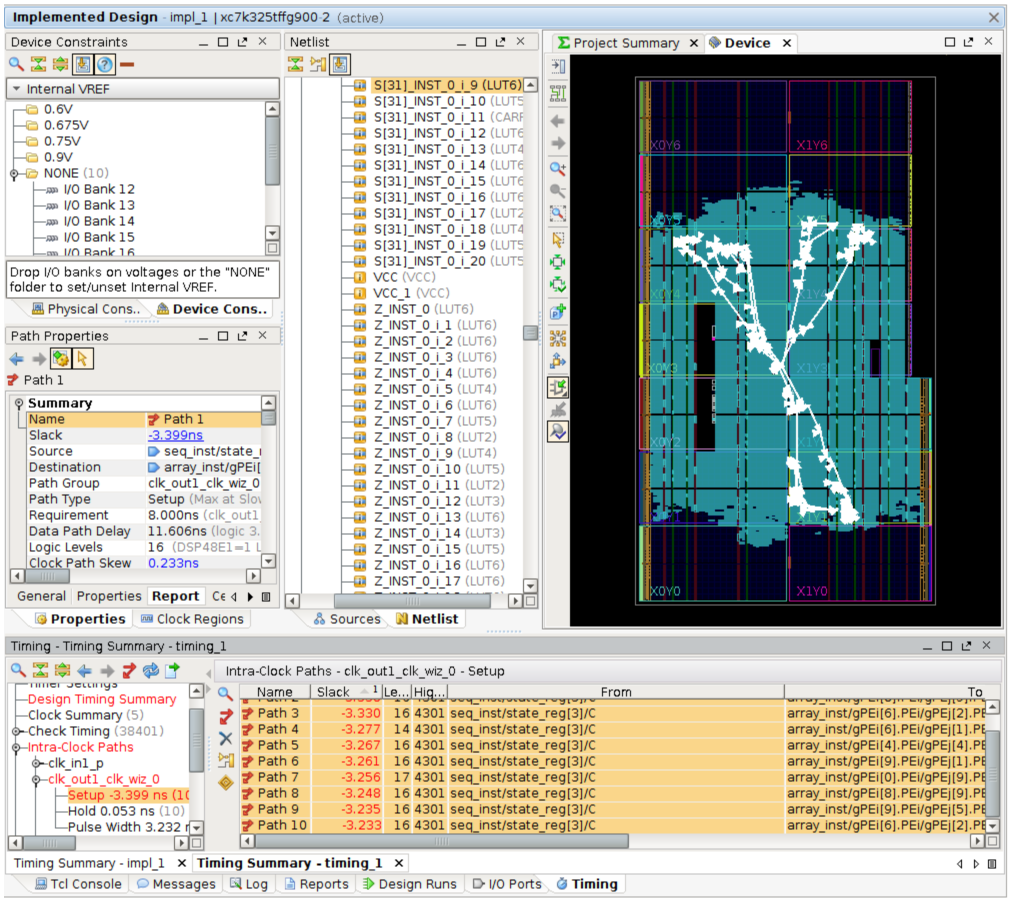
Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup

Name	Slack 1	Levels	High Fanout	From	
Path 1	-1.599	14	434	seq_inst/state_reg[2]/C	array_inst/gPEi[0].PEi/gPEj[2]
Path 2	-1.563	14	434	seq_inst/state_reg[2]/C	array_inst/gPEi[0].PEi/gPEj[2]
Path 3	-1.546	16	690	seq_inst/state_reg[2]/C	array_inst/gPEi[2].PEi/gPEj[2]
Path 4	-1.531	14	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]
Path 5	-1.516	13	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]
Path 6	-1.513	14	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]
Path 7	-1.502	14	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]
Path 8	-1.494	14	2482	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[1]
Path 9	-1.493	14	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]
Path 10	-1.490	14	4306	seq_inst/state_reg[2]/C	array_inst/gPEi[1].PEi/gPEj[2]

Figure 3.9: Screenshot of the faulty paths by a setup slack in a 10x10 array

Using the Floorplanning view, its possible to see the physical paths, affected by slack time, through the FPGA. An example is shown at the figure 3.10. Also, its very helpful to use the critical path option (see figure 3.11) to see all the blocks involved (blue signal).

[2]There are three types of slacks: setup, hold and pulse width. The first occurs when the signal comes too close to clock pulse (or later). The second appears when the signal changes before the period required for charging value. The third slack indicates that the clock waveform meets all the requirements.



Implemented Design - impl_1 | xc7k325tffg900-2 (active)

Device Constraints

Internal VREF
- 0.6V
- 0.675V
- 0.75V
- 0.9V
- NONE (10)
  - I/O Bank 12
  - I/O Bank 13
  - I/O Bank 14
  - I/O Bank 15
  - I/O Bank 16

Drop I/O banks on voltages or the "NONE" folder to set/unset Internal VREF.

Path Properties

Path 1

Summary

Name	Path 1
Slack	-3.399ns
Source	seq_inst/state_
Destination	array_inst/gPEi[
Path Group	clk_out1_clk_wiz_0
Path Type	Setup (Max at Slov
Requirement	8.000ns (clk_out1
Data Path Delay	11.606ns (logic 3.
Logic Levels	16 (DSP48E1=1 L
Clock Path Skew	0.233ns

Netlist

```
S[31]_INST_0_i_9 (LUT6)
S[31]_INST_0_i_10 (LUT5
S[31]_INST_0_i_11 (CARF
S[31]_INST_0_i_12 (LUT6
S[31]_INST_0_i_13 (LUT4
S[31]_INST_0_i_14 (LUT6
S[31]_INST_0_i_15 (LUT6
S[31]_INST_0_i_16 (LUT6
S[31]_INST_0_i_17 (LUT2
S[31]_INST_0_i_18 (LUT4
S[31]_INST_0_i_19 (LUT5
S[31]_INST_0_i_20 (LUT5
VCC (VCC)
VCC_1 (VCC)
Z_INST_0 (LUT6)
Z_INST_0_i_1 (LUT6)
Z_INST_0_i_2 (LUT6)
Z_INST_0_i_3 (LUT6)
Z_INST_0_i_4 (LUT6)
Z_INST_0_i_5 (LUT4)
Z_INST_0_i_6 (LUT6)
Z_INST_0_i_7 (LUT5)
Z_INST_0_i_8 (LUT2)
Z_INST_0_i_9 (LUT4)
Z_INST_0_i_10 (LUT5)
Z_INST_0_i_11 (LUT2)
Z_INST_0_i_12 (LUT3)
Z_INST_0_i_13 (LUT6)
Z_INST_0_i_14 (LUT3)
Z_INST_0_i_15 (LUT5)
Z_INST_0_i_16 (LUT6)
Z_INST_0_i_17 (LUT6)
```

Timing - Timing Summary - timing_1

- Timer Settings
- Design Timing Summary
- Clock Summary (5)
- Check Timing (38401)
- Intra-Clock Paths
  - clk_in1_p
  - clk_out1_clk_wiz_0
    - Setup -3.399 ns (10)
    - Hold 0.053 ns (10)
    - Pulse Width 3.232 n

Intra-Clock Paths - clk_out1_clk_wiz_0 - Setup

Name	Slack	Le...	Hig...	From	To
Path 3	-3.330	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[6].PEi/gPEj[2].PE
Path 4	-3.277	14	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[6].PEi/gPEj[1].PE
Path 5	-3.267	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[4].PEi/gPEj[4].PE
Path 6	-3.261	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[9].PEi/gPEj[1].PE
Path 7	-3.256	17	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[0].PEi/gPEj[9].PE
Path 8	-3.248	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[8].PEi/gPEj[9].PE
Path 9	-3.235	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[9].PEi/gPEj[5].PE
Path 10	-3.233	16	4301	seq_inst/state_reg[3]/C	array_inst/gPEi[6].PEi/gPEj[2].PE

Timing Summary - impl_1 | Timing Summary - timing_1

Tcl Console | Messages | Log | Reports | Design Runs | I/O Ports | Timing

Figure 3.10: Scrheenshot of VIVADO windows with the Report Timing summary report

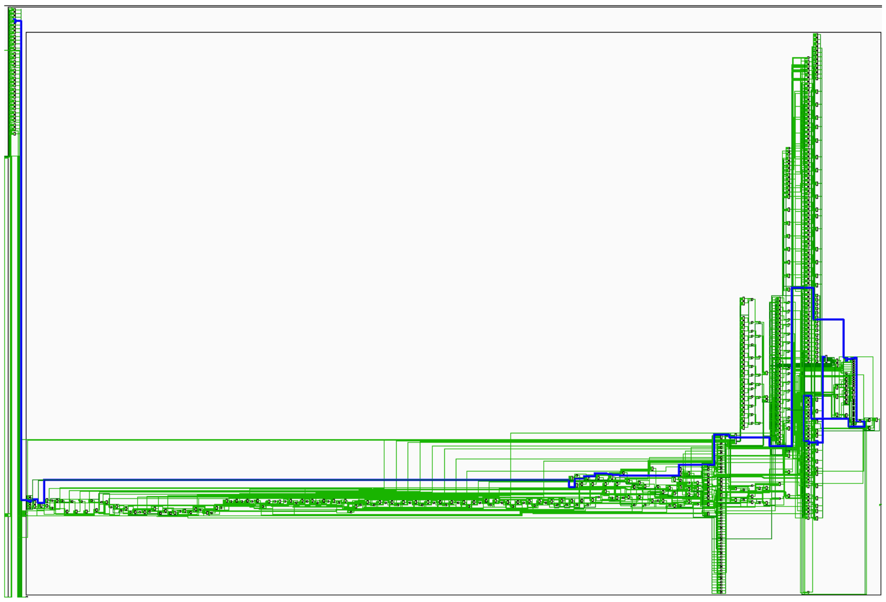


Figure 3.11: Screenshot of a critical path of a 10x10 array

3.2.2 Slack time of setup

The best way to solve problems of timing is using pipeline registers. Reduce the frequency under 125 MHz is not optimal because the rest of the system has not problems and the architecture must work at selected frequency.

But the most important consequence of using pipeline registers inconsistently is the desynchronization between components, in other words, all the pipelined signals will arrive one or more clocks later to the other component (depending on the number of registers added). Because of that, it's necessary to be very careful and apply pipeline registers to all the signals coming from the same component and taking in account the delay generated of all the output signals that depend on the previous ones.

In total, during the implementation, three pipeline points must be added to ensure the constraints compliance of the slack time of setup:

Input signals

All the input signals from the sequencer (opcode, data, address and control signals) have to be pipelined because opcode signal was a very critical path (adding more than a half of a clock period).

This modification causes a general delay of one clock in all the array, but it isn't important if all the input signals are synchronized.

Multiplication opcodes

Both operands of the multiplication unit inside the ALU also need to be pipelined to achieve the minimum time required to carry out the multiplication.

The addition of a register before the multiplier unit is very sensitive because it affects only to the multiplication opcode. The delay causes a desynchronization between operations and it forces to modify the Sequencer behaviour only in the multiplying case.

The solution adopted was to increment one clock the opcode duration of MUL and MULS instructions and modify the enable logic to allow the write process only during the second period clock. This modification can be observed at the simulation of the figure 3.12.

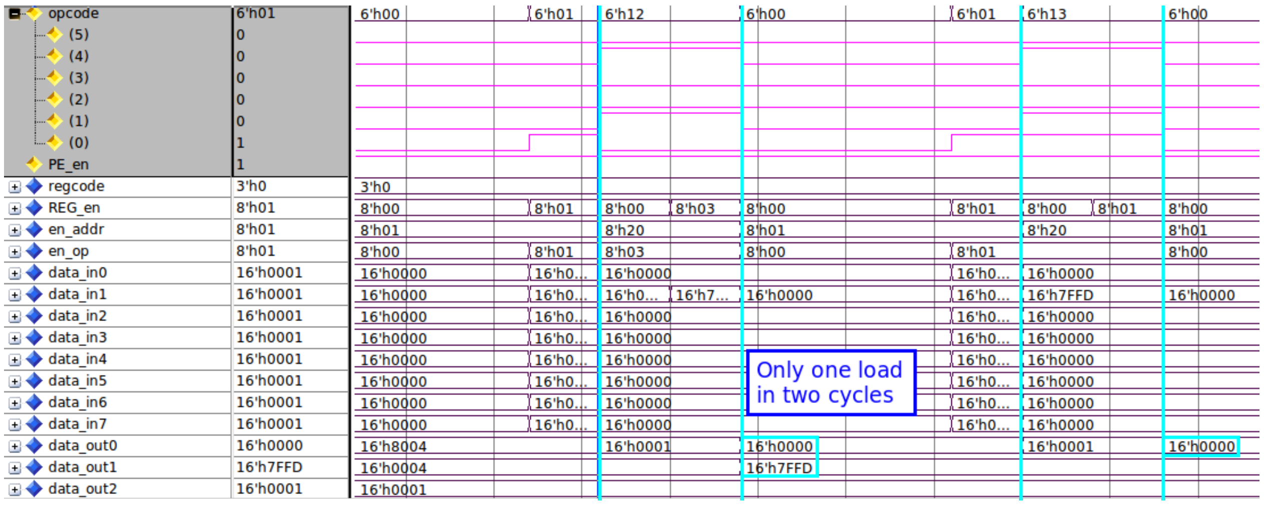


Figure 3.12: Detail of the double clock cycle for execute MUL and MULS opcodes

Spike distribution

During the spike distribution, due to the big size of the array, is necessary to add one clock delay before the activation of the next row. This causes a discontinuity during the spikes distribution that has been solved by modifying the combinational circuit of the spike valid signal. The discontinuity is clearly visible at the simulation waveform of the figure 3.13.

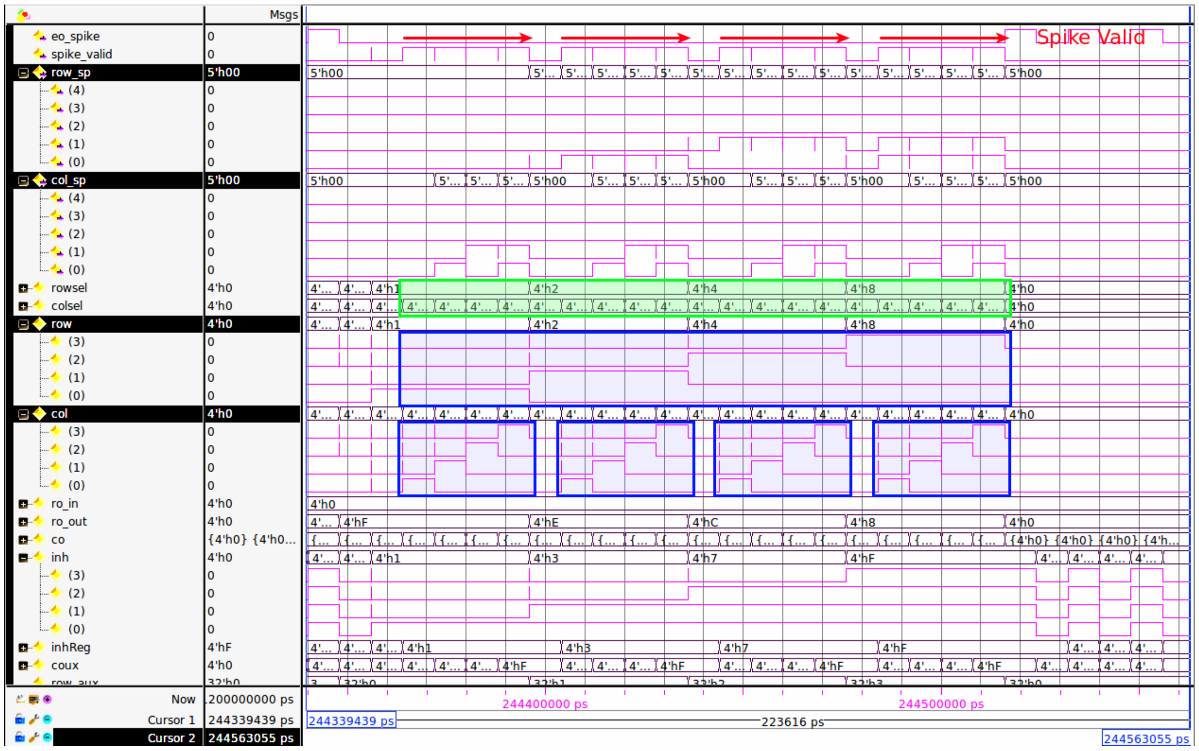


Figure 3.13: Detail of the discontinuous spike distribution and the spike_valid signal

Chapter 4

Results

After the simulations and tests carried out, it is verified that the PEs are working properly, spikes are correctly read from the array and the subsequent feedback to carry out several cycles. Although some components may be more optimized and the system may be tested with various interconnected FPGAs in a ring topology, the Multiprocessor Array of the HEENS architecture meets the specifications described in the introduction.

The new architecture using Block RAMs instead of registers for synaptic interconnects optimizes the utilization of the FPGA resources and allows to implement a bigger Multiprocessor Array. The previous version array achieved a 10x10 PEs and the main limitation was the utilization of registers. With the HEENS architecture using Block RAMs, the array achieve a dimension of 12x12 PEs. Taking in account the maximum virtualization layers (8 neurons by PE), the total number of emulated neurons is 1,152 in each FPGA. Finally, assuming that the AER System supports 126 chips connected in a ring topology, the potential number of neurons reaches 145,152.

Moreover, the utilization of Block RAMs is the new limitation on the used FPGA, reaching almost a 100 % of them. At the table 4.1 is shown the utilization of HEENS architecture in a full FPGA and the number of resources used by one Process Element. Also there is a comparative between SNAVA and HEENS architectures implementing a 10x10 Multiprocessor Array (figure)4.1).

The instruction set has also been modified. The improvement allows to execute faster (in less cycles) some usual actions. This is vital to reduce the among of time of the entire algorithm or program a more extensive one without increasing the duration of the execution.

Resources	**Total**	**Used by array [%]**	**Used by PE**
Slice LUTs	203800	88.60	1,245
Slice Registers	407600	18.78	512
F7 Muxes	101900	7.09	50
F8 Muxes	50950	2.27	8
Slice	50950	96.01	418
LUT as Logic	203800	88.53	1,245
LUT as memory	64000	0.22	0
LUT Flip Flop Pairs	203800	90.16	1,308
Block RAM Tile	445	97.64	3
DSPs	840	17.14	1
Bonded IOB	500	0.60	0
BUFGCTRL	32	12.50	0
MMCME2_ADV	10	10.00	0
BSCANE2	4	25.00	0

Table 4.1: Total of resources, total utilzation by a 12x12 array and utilization by one PE

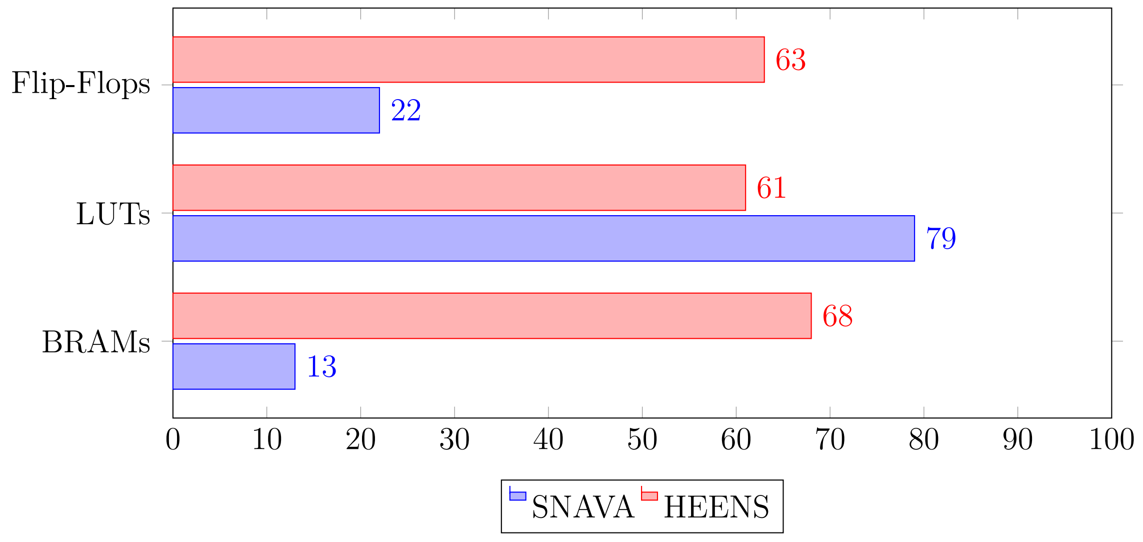


Figure 4.1: Percentage of the utilization between HEENS and SNAVA at 10x10 array

The improvement also reduces in more than 90 % the time required to synthesize and implement all the components. This is because the connections are defined in Block RAMs and is faster to generate huge memory blocks than interconnect a lot of isolated registers to generate only one of them. Before, with the SNAVA architecture, the synthesis and implementation process could take more than 18 hours, but now, using HEENS, is completed in not more than 30 minutes (i.e. 9:33 min. for synthesis and 20:26 min. for implementation), as shown on the table 4.2.

Process	**SNAVA**	**HEENS**
Synthesis	17:35	00:10
Implementation	00:39	00:20
Total time	18:14	00:30

Table 4.2: Comparsion of the utilization between the previous and the current PEs resources

As an interesting fact, the figure 4.2 shows the maximum occupancy of the FPGA, in blue color. Only a few small areas near corners are free, except two central regions, which are dedicated to another unused functionality.

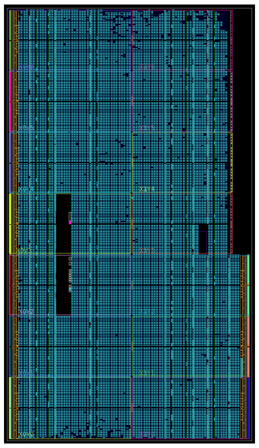


Figure 4.2: Floorplanning of the 12x12 Multiprocessor Array with AER interface

Integrate and fire is one of the most used models for emulation SNN. It is based on the membrane potential computing in function of the receiver spikes through the synapses predefined. Running this algorithm (included at **Appendix B Integrate and Fire**) and showing the membrane potential, at the waveform of the figure 4.4 can be observed a spike generated by a neuron. In this case, the programmed topology is all the neurons spiking to the neuron at position row 0 and column 0, as is shown on image 4.3, so the spike generated is from this one.

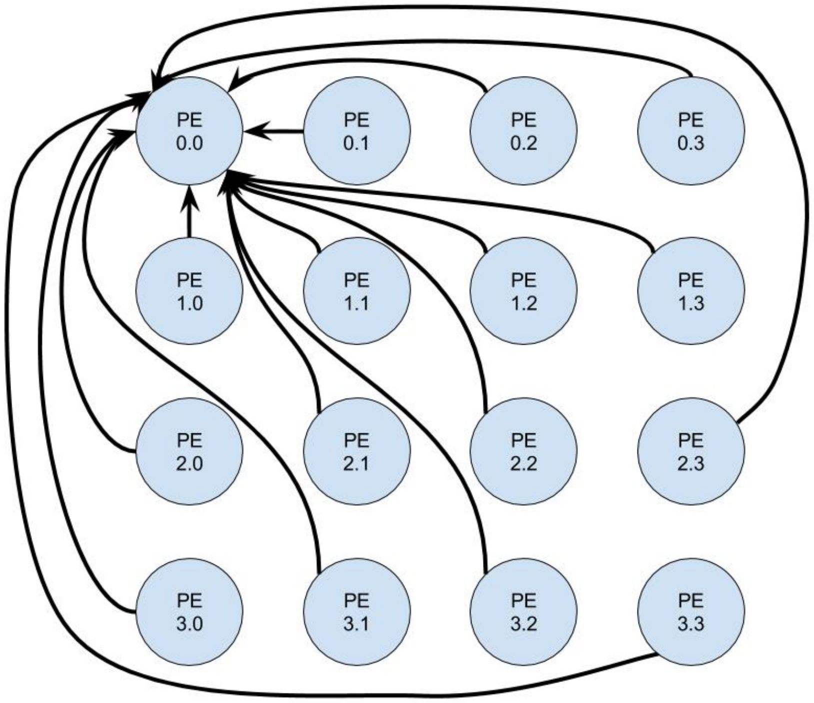


Figure 4.3: Topology of all spiking to one

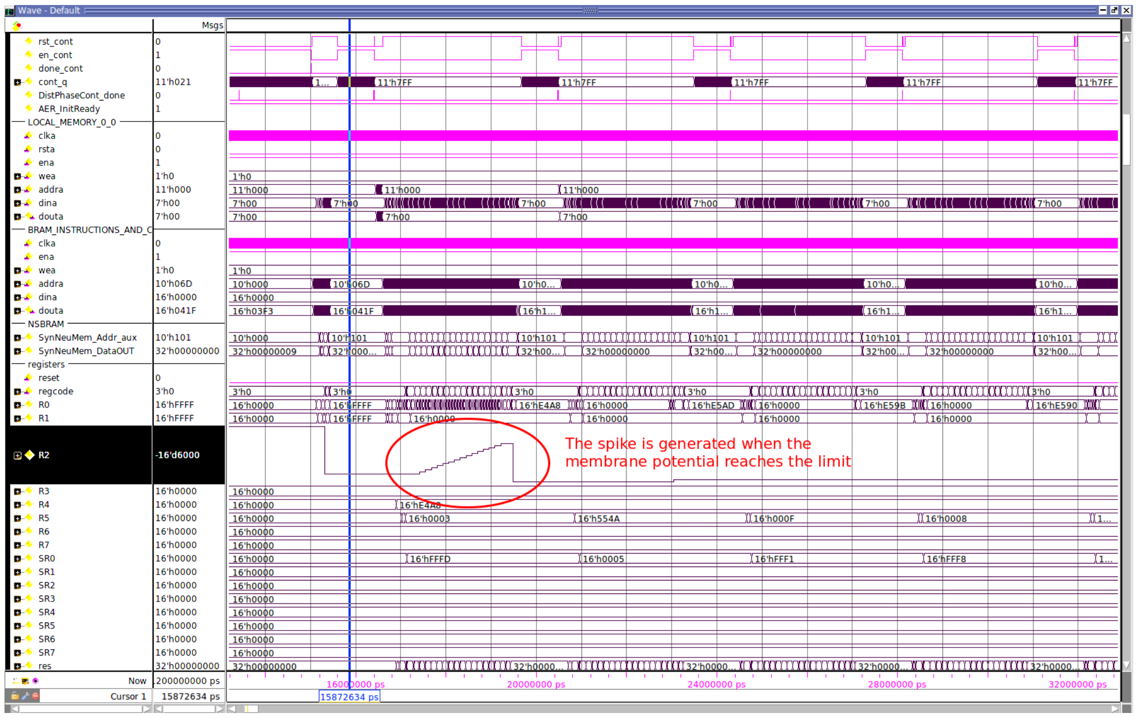


Figure 4.4: Waveform membrane potential when the neuron generates a spike

Chapter 5

Budget

Budget items	**€/item**	**Amortization**	**Time**	**Budget [€]**
Xilinx Kintex 7 KC705	1,600 €	3 years	3 months	133
QuestaSIM University Licence	1,550 €	1 year	6 months	755
Vivado Academic Licence	Free	-	-	0
RRHH	8 €/h	-	720 hours	5,760
			Total	**6,648**

Table 5.1: Project cost

The total budget for this project is **6,648 €**.

Chapter 6

Conclusions and future development

Compared to the previous SNAVA architecture, the HEENS architecture optimizes the FPGA resources and allows to implement more Process Elements in one chip (44 % of improvement). Also reduces the necessary time to implement all the components; that is very important because modifications of the design can be changed and tested faster. With the virtualization layers and the ring topology (outside the work of this scope) thanks to the scalable capability, the number of emulated neurons can be significantly higher than the previous architecture.

To increase the number of neurons, Block RAM memories would be optimized to use less resources by each PE and allow the implementation of more of them. Another improvement is to reform the ALU, because it is an extremely complex component and has many combinational circuits that limit the maximum frequency (increases the slacks time). A more drastic solution is to use another FPGA with more resources.

It very is important to describe all the components optimized to reduce delays between registers and prevent timing slacks, very frequently due to the complexity and longitude of signals between registers, in particular, between each PE and the Sequencer.

Bibliography

[1] Zhang, Ling, and Bo Zhang. ”A geometrical representation of McCulloch-Pitts neural model and its applications.” IEEE Transactions on Neural Networks 10.4 (1999): 925-929.

[2] Hinton, Geoffrey E., and Terrence J. Sejnowski. ”Learning and releaming in Boltzmann machines.” Parallel distributed processing: Explorations in the microstructure of cognition 1 (1986): 282-317.

[3] Floreen, Patrik, and Pekka Orponen. ”On the computational complexity of analyzing Hopfield nets.” Complex Systems 3.6 (1989): 577-587.

[4] Wolfgang Maass, ”Networks of Spiking Neurons: The Third Generation of Neural Networks Models”. Institute for Theoretical Computer Science, Technische Universität Graz, Graz, Austria. 1997.

[5] T. Sharp, F. Galluppi, A. Rast, and S. Furber, “Power-efficient simulation of detailed cortical microcircuits on SpiNNaker”. Journal of Neuroscience Methods, 2012.

[6] Ewan Nurse, Benjamin S. Mashford, Antonio Jimeno Yepes, Isabell Kiral-Kornek, Stefan Harrer, and Dean R. Freestone. 2016. ”Decoding EEG and LFP signals using deep learning: heading TrueNorth”. In Proceedings of the ACM International Conference on Computing Frontiers (CF ’16). ACM, New York, NY, USA, 259-266. DOI: https://doi.org/10.1145/2903150.2903159

[7] Giovanny Sánchez Rivera, “Efficient multiprocessing architectures for Spiking Neural Network emulation based on configurable devices” Ph.D. dissertation. Advanced Hardware Architectures (AHA) Research Group of the Electronics Engineering Department, Universitat Politècnica de Catalunya, Barcelona, Spain, 2014.

[8] Mireya Zapata and Jordi Madrenas, “Compact Associative Memory for AER Spike Decoding in FPGA-Based Evolvable SNN Emulation”. Electronics Engineering Department, Universitat Politècnica de Catalunya, Barcelona, Spain.

Glossary

- AER: Address-Event Representation
- ALU: Arithmetic Logic Unit
- ANN: Artificial Neural Network
- FPGA: Field Programmable Gate Array
- HEENS: Hardware Evolved Emulator Neural System
- LFSR: Linear Feedback Shift Register
- Opcode: Operation code
- PE: Processor Element
- SIMD: Single Instruction Multiple Data
- SNAVA: Spiking Neural-network Architecture for Versatile Applications
- SNN: Spiking Neural Network
- VHDL (VHSIC + HDL): ”Very High Speed Integrated Circuit” + ”Hardware Description Language”

Appendices

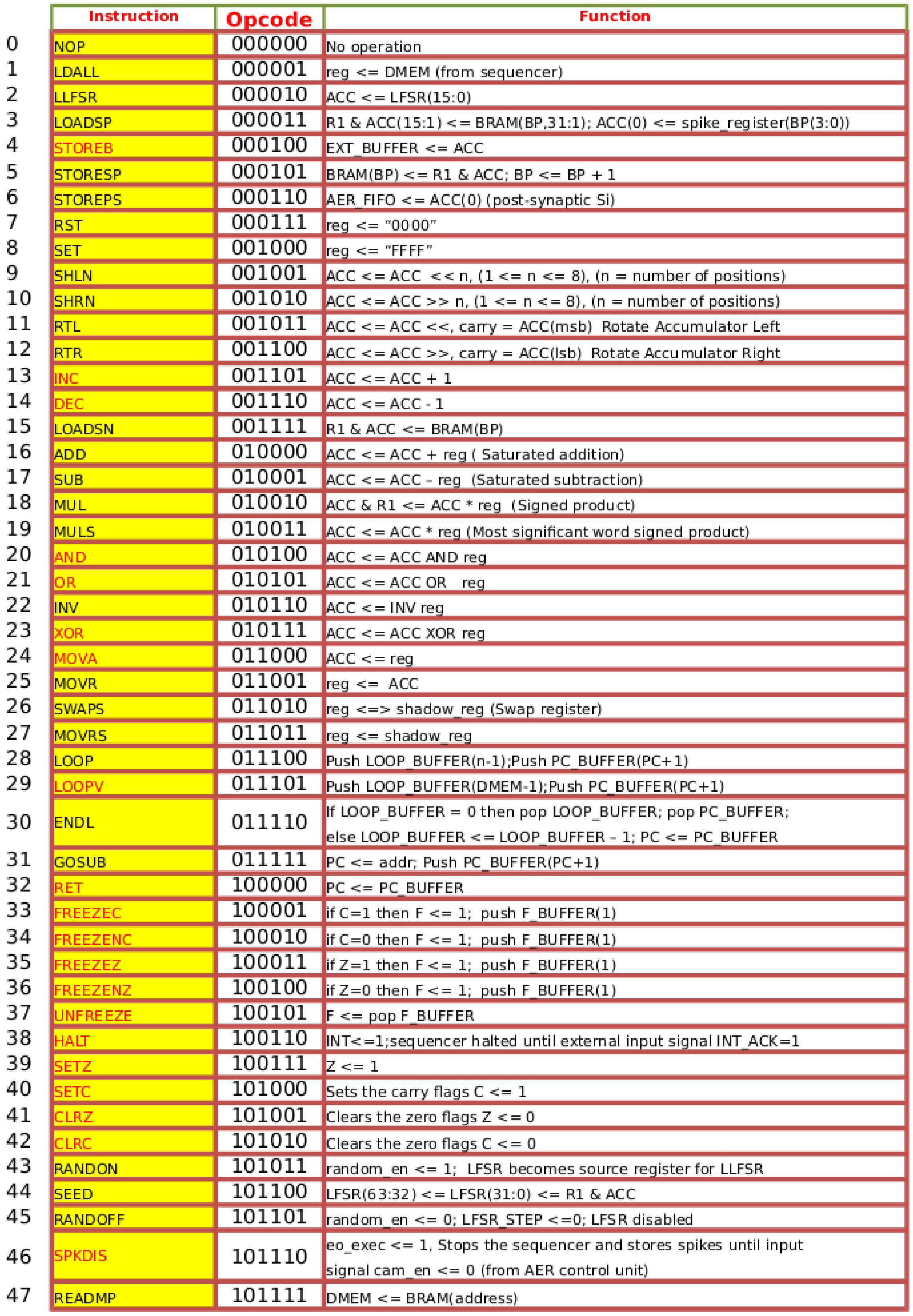
Appendix A

Set of instructions

The opcodes (operation codes) are the instruction set used to carry out the execution of the entire algorithm. There are some categories depending on the purpose of the order, ie, arithmetic, movements, freeze, sequencer, registers, but all of them always are sent to all the components.

The name of the opcodes is used to create the programs in Assembler that will be executed by the Processing Elements. Then, when it is compiled, the name is translated to its binary number of 6 bits and, with other data, is sent through the communication bus.

Also, the name is used instead of the binary code in the VHDL files to avoid having to change the binary code associated with the opcode if it is moved to another position of the table (figure A.1).



	Instruction	Opcode	Function
0	NOP	000000	No operation
1	LDALL	000001	reg <= DMEM (from sequencer)
2	LLFSR	000010	ACC <= LFSR(15:0)
3	LOADSP	000011	R1 & ACC(15:1) <= BRAM(BP,31:1); ACC(0) <= spike_register(BP(3:0))
4	STOREB	000100	EXT_BUFFER <= ACC
5	STORESP	000101	BRAM(BP) <= R1 & ACC; BP <= BP + 1
6	STOREPS	000110	AER_FIFO <= ACC(0) (post-synaptic Si)
7	RST	000111	reg <= "0000"
8	SET	001000	reg <= "FFFF"
9	SHLN	001001	ACC <= ACC << n, (1 <= n <= 8), (n = number of positions)
10	SHRN	001010	ACC <= ACC >> n, (1 <= n <= 8), (n = number of positions)
11	RTL	001011	ACC <= ACC <<, carry = ACC(msb) Rotate Accumulator Left
12	RTR	001100	ACC <= ACC >>, carry = ACC(lsb) Rotate Accumulator Right
13	INC	001101	ACC <= ACC + 1
14	DEC	001110	ACC <= ACC - 1
15	LOADSN	001111	R1 & ACC <= BRAM(BP)
16	ADD	010000	ACC <= ACC + reg (Saturated addition)
17	SUB	010001	ACC <= ACC – reg (Saturated subtraction)
18	MUL	010010	ACC & R1 <= ACC * reg (Signed product)
19	MULS	010011	ACC <= ACC * reg (Most significant word signed product)
20	AND	010100	ACC <= ACC AND reg
21	OR	010101	ACC <= ACC OR reg
22	INV	010110	ACC <= INV reg
23	XOR	010111	ACC <= ACC XOR reg
24	MOVA	011000	ACC <= reg
25	MOVR	011001	reg <= ACC
26	SWAPS	011010	reg <=> shadow_reg (Swap register)
27	MOVRS	011011	reg <= shadow_reg
28	LOOP	011100	Push LOOP_BUFFER(n-1);Push PC_BUFFER(PC+1)
29	LOOPV	011101	Push LOOP_BUFFER(DMEM-1);Push PC_BUFFER(PC+1)
30	ENDL	011110	If LOOP_BUFFER = 0 then pop LOOP_BUFFER; pop PC_BUFFER; else LOOP_BUFFER <= LOOP_BUFFER – 1; PC <= PC_BUFFER
31	GOSUB	011111	PC <= addr; Push PC_BUFFER(PC+1)
32	RET	100000	PC <= PC_BUFFER
33	FREEZEC	100001	if C=1 then F <= 1; push F_BUFFER(1)
34	FREEZENC	100010	if C=0 then F <= 1; push F_BUFFER(1)
35	FREEZEZ	100011	if Z=1 then F <= 1; push F_BUFFER(1)
36	FREEZENZ	100100	if Z=0 then F <= 1; push F_BUFFER(1)
37	UNFREEZE	100101	F <= pop F_BUFFER
38	HALT	100110	INT<=1;sequencer halted until external input signal INT_ACK=1
39	SETZ	100111	Z <= 1
40	SETC	101000	Sets the carry flags C <= 1
41	CLRZ	101001	Clears the zero flags Z <= 0
42	CLRC	101010	Clears the zero flags C <= 0
43	RANDON	101011	random_en <= 1; LFSR becomes source register for LLFSR
44	SEED	101100	LFSR(63:32) <= LFSR(31:0) <= R1 & ACC
45	RANDOFF	101101	random_en <= 0; LFSR_STEP <=0; LFSR disabled
46	SPKDIS	101110	eo_exec <= 1, Stops the sequencer and stores spikes until input signal cam_en <= 0 (from AER control unit)
47	READMP	101111	DMEM <= BRAM(address)

Figure A.1: Set of instuctions (opcodes)

Appendix B

Integrate and fire algorithm

```
;16 Neurons, 16 Synapses
define synapses  15
define neurons_virtualized  0

.DATA

DMEM1="0000EF7D"   ;DMEM
POT1="000003E8"
THETA1="0000E380"  ;THETA
VREST1="0000E188"
CERO = "00000000"
UNO="00000001"
DOS="00000002"
CTE_1="00000007"    ;CTER Refractory time
CTETP="0000F448"

.CODE
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 = Si
    MOVR  R3        ;R3    <-- Si
    SWAPS R0        ;SR0_2 <-- R0 = Si
;-------------------- R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) -----------------
    MOVA R3                     ;R0  <-- R3 = Si
    RTR
    FREEZEC       ;IF (Si = 0) THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
        SWAPS R6    ;R6  <-- SR6_2 = Vi
        MOVA  R6    ;R0  <-- R6 = Vi
        SUB   R5    ;R0  <-- Vi - Vres
        MULS  R4    ;R0  <--(Vi(t)-Vres) * (Kmem)   --MZ
        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  + (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    R6      ;R6  <-- Vres1+(1-Si(t))*(Vi(t)-Vres)*(Kmem)+SUM_WEIGHTS
    SWAPS   R6  ;SR6 <-- R6 = Vi
    RST     R2  ;SUM_WEIGHTS <-- 0
    SWAPS   R2      ;SR2_2 <-- R2 = SUM_WEIGHTS
RET
; --------------------------------------------------------------------------
; ---------------------------- SYNAPSE LOAD -------------------------------

.SYNAPSE_LOAD

LOADSP
        ;ACC <-- BRAM(BP,15:1) & Spike_reg(BP) : Sj
        ;R1 <-- BRAM(BP,31:16) : Wij
RET
; ---------------------------------------------------------------------------
; --------------------------- SYNAPTIC WEIGHT ------------------------------
.SYNAPTIC_WEIGHT

RTR       ; C <-- Sj
FREEZENC  ;IF (Sj = 1) THEN R0 <-- wji = Aji * P
    MOVA     R1  ; R0  <--  Wij
    SWAPS    R2  ;R2  <-- SR2_2 = sumW
    ADD      R2  ;SR0 <-- wji = Sj * P
    MOVR     R2      ;R2  <-- wji = Sj * P
    SWAPS    R2  ;SR2_2 <-- R2  = sumW
UNFREEZE
RET
;----------------------------------------------------------------------------
; --------------------------- SYNAPSE_SAVE ---------------------------------
.SYNAPSE_SAVE
; THE SYNAPTIC PARAMETERS GO TO BUFFER 32 bits

    RST      R0
    STORESP
RET
; ---------------------------------------------------------------------------
; ---------------------------- SPIKE UPDATE --------------------------------
.SPIKE_UPDATE
    RST      R0
;   SWAPS    R0  ;R0     <-- SR0 = Si
;   RTR
;   RTL
    MOVR     R2  ;R2     <-- Si has been reset = 0
    LDALL    R3,THETA1   ;R3  <-- THETA1 = "0000F060" THRESHOLD VOLTAGE
    SWAPS    R6      ;R6 <-- SR6_2 = Vi
    MOVA     R6
    SWAPS    R6  ;SR6_2  <-- R6 = Vi
    SUB      R3  ;R0     <--  Vi - (THETA1)
    RTL      ; Vi -  (THETA1) > 0 ?
    FREEZEC
            SWAPS   R4  ;R4     <-- SR4 = Tref
        RST R0
        XOR     R4
        SWAPS   R4
        FREEZENZ        ; IF  (Z = 1) THEN Tref=0 and Si is set
            RST     R0
            BITSET  CERO
            MOVR    R2
;           LDALL   R3,UNO
;           MOVA    R2
;           ADD     R3
;           MOVR    R2
            LDALL   R4,CTE_1    ;CTE_1 = 7
            SWAPS   R4  ;Load again in SR4 initial refractory time
        UNFREEZE
    UNFREEZE
    MOVA  R2    ;R0    <--  Si
    SWAPS R0    ;SR0_2 <-- R0 = Si
RET
; ---------------------------------------------------------------------------
; ------------------------------- REFRACTORY P ------------------------------
.REFRACTORY_P
    SWAPS   R4  ;R4     <-- SR4 = Tref
    MOVA    R4
    RTR
    MOVR    R4
    SWAPS   R4  ;SR4    <-- R4 = Tref
RET
; ---------------------------------------------------------------------------
; ------------------------ENABLE SPIKES PROPAGATION-------------------------
.SPIKES_ENABLE
    SWAPS R0
    MOVR R2     ; R2 <-- Si
    SWAPS R0
    MOVA R2     ; R0 <== Spikes
    STOREPS
```

```
RET

; --------------------------------------------------------------------------
; **************************** PROCEDURES END ******************************
; ************************** MAIN PROGRAMME BEGIN **************************
.MAIN

LDALL   R6,VREST1 ; Vi<-Vrest
SWAPS   R6

.ALG_LOOP
    GOSUB MEMBRANE_VALUE
    LOOP synapses           ;synaptic loop
        GOSUB  SYNAPSE_LOAD
        GOSUB  SYNAPTIC_WEIGHT
        GOSUB  SYNAPSE_SAVE
    ENDL
    GOSUB SPIKE_UPDATE
    GOSUB REFRACTORY_P
    GOSUB SPIKES_ENABLE
    SPKDIS
GOTO ALG_LOOP
; **************************** MAIN PROGRAMME END *************************
```