POLITECNICO DI TORINO

Department of Electronic and Telecommunications
Master of Science in Electronic engineering

Master's Thesis

Scalability and Virtualization enhancement in Spiking Neural Hardware based on PSOC

Advisors:
Prof. Maurizio Martina
Prof. Jordi Madrenas Boadas
Prof. Mireya Zapata Rodriguez

Candidate:
Roberto Gattuso

Academic year 2019/20

Abstract

This thesis work is focussed on the scalability and virtualization enhancement of *HEENS* architecture, which is a spiking neural hardware emulator. All blocks are described in VHDL, simulated to check their behaviour and finally the whole system is synthesized and implemented on a PSOC, in order to verify time constraints and area occupied.

In particular, all the improvements made concern the processing element array, which represents spiking neurons that have to be emulated. Each processing element consists of an ALU, a register file, a virtualization register, a LFSR and memory blocks containing synaptic parameters and membrane potential values.

Finally the architecture has a new spike distribution structure, making the system more scalable, and virtualization is introduced, in order to extend the array without requiring more resources, thus saving space on the board. Exploiting at most the available area, it has been possible to simulate a 13x13 array with problem-free timings.

Acknowledgments

This thesis work would never have existed without the support of many people who have been present during this beautiful journey abroad. First of all, I have to thank Prof. Jordi Madrenas for having accepted me as a graduate student while I was still in Italy and for being present during these months, guiding this work, always ready to help me to find a solution to every problem, and Mireya Zapata, always available to collaborate despite the difficulties of distance, giving an huge support for FPGA test. I also want to thank Prof. Maurizio Martina, my advisor in Italy, who has had to monitor my work and despite the distance has always wanted to receive updates and give advice about them. I do not forget all the important people who have been close to me during these months, giving me help each in its own way: Martina, Francesca, Vittoria, Salvo; my quiet flatmates of Barcelona, Lorenza and Elisa, and those of Turin, Antonio and Totò; all the Erasmus friends met during this journey; the laboratory guys, Josep Angel, Diana, Josep Maria. I can not mention everyone, but thank you all for being part of my life.

Last but not least, I dedicate this work to my family, without whose moral support, their trust in me and their efforts, I would never have arrived where I am now. Thanks for existing.

Abbreviations

AER	Address Event Representation
ALU	Arithmetic and Logic Unit
BRAM	Block Random Access Memory
EPSP	Excitatory Post-Synaptic Potential
HEENS	Hardware Emulator of Evolved Neural System
IF	Integrate and Fire
IPh	Initialization Phase
IPSP	Inhibitory Post-Synaptic Potential
ISA	Instruction Set Architecture
IZ	Izhikevich
LFSR	Linear-Feedback Shift Register
LIF	Leaky Integrate and Fire
LIFO	Last In First Out
LSB	Less Significant Bit
LUT	Lookup Table
ND	Neural processing Device
MC	Master Chip
MSB	Most Significant Bit
PE	Processing Element
PSP	Post-Synaptic Potential
SIMD	Single Instruction Multiple Data
SNAVA	Spiking Neural-Networks Architecture for Versatile Applications
SNN	Spiking Neural Network
VHDL	Very High speed integrated circuits Hardware Description Language

Table of contents

List of figures

List of tables

Chapter 1

Introduction

The study of human brain has been the subject of multiple investigations in order to understand how it works and being able to imitate its behaviour.
Reverse engineering, that generally is the process by which a man-made object is de-constructed to reveal its designs, architecture, or to extract knowledge from the object, applied to reproducing human intelligence, has led to a revolution of science in several fields with a wide range of applications such as speech recognition, robotics, devices smart, navigation, vision, etc. [5]. The growing interest in having efficient platforms that exhibit massive parallelism, energy efficiency, scalability with efficient connectivity and plasticity [1] has made nowadays simulation of complex biological neural systems a trending research area; moreover neuroscientists consider it an important tool in understanding the structure and dynamics of the human brain [6]. Since human brain holds all these characteristics, therefore its unparalleled performance has encouraged many engineers to develop bio-inspired simulators that mimic part of the human brain functionality. There are various approaches related to this field that concentrate their efforts in reaching a trade-off between realism, scalability, speed, flexibility etc.

From these assumptions, several neural network models were born over the years, such as Spiking Neuron Networks (SNNs), that belong to the 3rd generation of neural networks. This thesis project starts from the *HEENS* ("Hardware Emulator of Evolvable Neural Systems") architecture, that has been developed by the formerly Advanced Hardware Architecture group of the Department of Electronics Engineering of Universitat Politècnica de Catalunya (UPC); currently the architecture is kept on by new Integrated Smart Sensors and Health Technologies (ISSET) group.

Digital hardware is designed for simulating SNN. This work, in particular, is focussed on the hardware part of the HEENS processing element array, making improvements in terms of scalability, operation speed and resource occupancy. The first goal consists in trying to increase the array without violating time constraints: pipeline stages are introduced with the purpose to allow the system to support a

greater number of neurons while preserving the operation speed. Another goal is to support the virtualization of PE up to seven neurons in addition of the main layer: this means that, without introducing new PEs, it is possible to emulate them, increasing the number of neurons to be simulated without using new hardware resources.
Moreover, the architecture is synthesized and implemented on the Xilinx Zynq-7000 SoC ZC706 board, which is an MPSoC device chosen to enhance the system configurability and monitoring, since it integrates both processor and FPGA architectures into a single device. The array size is extended as much as possible to test if time constraints are violated when the area occupancy on the board is close to 100%, so verifying if the introduced pipeline stages have reduced the critical path enough to preserve the correct behaviour of the system.

Chapter 2

State of Art

The human brain study has been the subject of multiple investigations in order to understand how it works and being able to imitate its behaviour. The state of art investigation is of priority importance to understand firstly which are nowadays the most important models that describe a biological neural network and then which are the most significant architectures able to emulate it.

Therefore, this chapter is an introduction step into spiking neural networks world, which belongs to the last generation of neural networks: basic biological notions will be introduced, followed by a description of the most relevant neural models, concluding with a reference to the different variants of implementations that modern architectures adopt.

2.1 Biological neurons network

A biological neural system consists of millions of highly integrated neurons with multiple dynamic functions operating in coordination with each other.

The original work of McCulloch & Pitts in 1943 proposed a neural network model based on simplified "binary" neurons, where a single neuron is represented by a state that can be either *active* or *not active*, and at each neural computation step the state of of the neuron under examination is determined by calculating the weighted sum of the states of all the afferent neurons [1].

$$y = \sum_i x_i w_i \qquad y = \begin{cases} 1 & \text{if } x_i w_i > \theta \\ 0 & \text{if } x_i w_i \leq \theta \end{cases} \tag{2.1}$$

Connections between neurons are mapped from neuron N_i to neuron N_j) and each of them has specific a weight (w_{ij}). If the weighted sum of the states x_i of all the neurons N_i connected to a neuron N_j exceeds the characteristic threshold θ of N_j, the state of N_j is set to active, otherwise it is not. This first model, called "threshold neuron", is the simplest one and it was replaced by more realistic models based on linear or non-linear threshold functions (respectively "saturation" and "sigmoidal" neuron) as shown in fig.2.1.

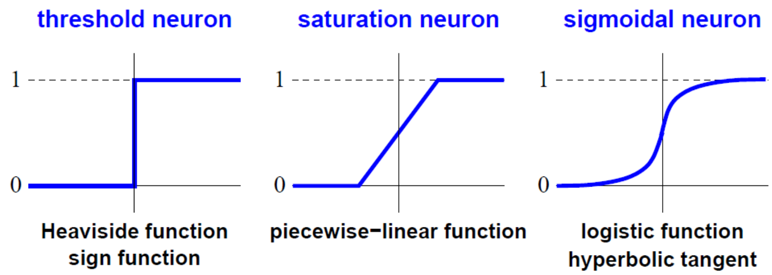


Figure 2.1. Neuron models based on the dot product computation [1].

Separately, neurobiological research has greatly progressed and, regarding the study of cognitive processing, individual spikes time is intended as the means of communication and neural computation. The current consent agrees that cognitive processes are most likely based on the activation of transient assemblies of neurons although the underlying mechanisms are not yet understood well [1].

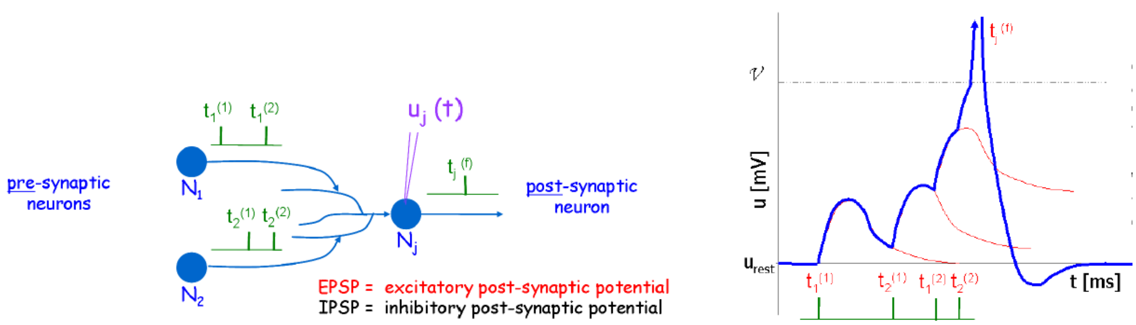


Figure 2.2. A model of spiking neuron: EPSP (red curve) or IPSP; they are all added (blue line) [1].

Fig.2.2 shows a model of spiking neuron: in the left picture each pre-synaptic spike generates an excitatory post-synaptic potential (EPSP) or an inhibitory post-synaptic potential (IPSP) in case of negative weight. A post-synaptic neuron N_j fires a spike whenever the weighted sum of incoming EPSPs generated by its pre-synaptic neurons reaches a given threshold ν. Instead the right graphic shows how the membrane potential of N_j varies through time, under the action of four incoming spikes. Entering into detail, fig.2.3 shows an example of a basic neuron network. Most biological neurons rely on pulses as an important part of information transmission from one neuron to another one and in a rough and non-exhaustive outline, a neuron can generate an action potential (the spike) at the soma, the cell body of the neuron. Then, this brief electric pulse (with 1 or $2ms$ of duration) travels through the neuron's axon arriving to the receiving end of target neurons, the dendrites. In the final part of the axon, synapses connect one neuron to another one, and at the arrival of each individual spike, the synapses may release neurotransmitters along the synaptic cleft. These neurotransmitters are taken up by the neuron at the receiving end, modifying the state of that post-synaptic neuron, in particular the *membrane potential*. The transient impact a spike has on the neuron's membrane potential is generally referred to as the post-synaptic potential (PSP), and it can either inhibit the future firing (IPSP) or excite the neuron, making it more likely to fire (EPSP). Depending on the neuron, and the specific type of connection, a PSP may directly influence the membrane potential for anywhere between tens of microseconds and hundreds of milliseconds [1].

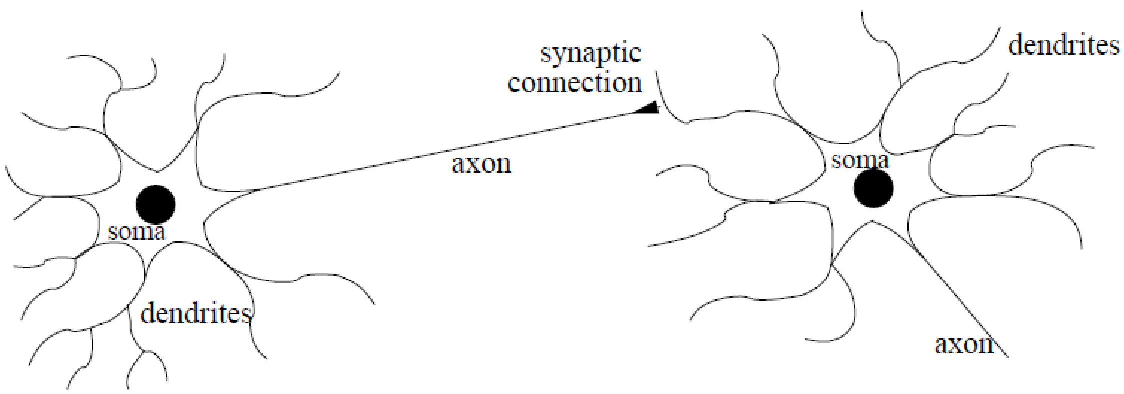


Figure 2.3. Structure of a neuron network.

2.2 Spiking Neuron Networks

Spiking Neuron Networks (SNNs) are often referred to as the 3rd generation of neural networks. They derive their strength and interest from an accurate modelling of synaptic interactions between neurons, taking into account the time of spike firing [1]. One example of such coding that easily compares to traditional neural coding, is temporal coding, a straightforward method for translating a vector of real numbers into a spike train. The basic idea is biologically well-founded: the more intensive the input is, the earlier the spike transmission will be. Hence a network of spiking neurons can be designed with n input neurons N_i whose firing times are determined through some external mechanism. The network is fed by successive n-dimensional input analog patterns x = $(x_1,...,x_n)$ that are inside a bounded interval T_{in} and

translated into spike trains through successive temporal windows. In each time window, a pattern x is temporally coded by one spike emission of neuron N_i at time t_i = T_{in} - x_i, for all i. This temporal coding, with other assumptions, shows that any traditional neural network can be emulated by an SNN. [1]
The first difficult task is to define the model of spiking neurons and synaptic plasticity, as there exist numerous variants already. A spiking neuron model accounts for the impact of impinging *action potentials* (spikes) on the targeted neuron in terms of the internal state of the neuron, as well as how this state relates to the spikes the neuron fires.

2.2.1 Leaky Integrate-and-Fire model

Leaky Integrate-and-Fire (LIF) is a neuron model much more computationally tractable that other ones, as the Hodgkin-Huxley from which it derived from.

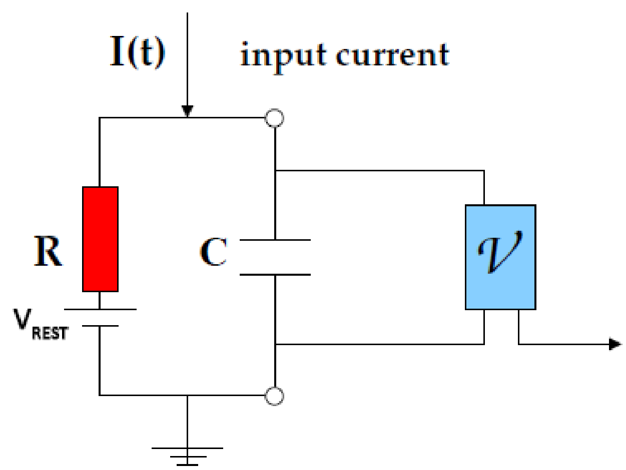


Figure 2.4. The Integrate-and-Fire electrical model.

Fig.2.4 represents this simple model, where ν is the threshold voltage that has to be overcome to generate a spike. This simplification has to be able to approximate the real behaviour shown in fig.2.2.Compared to the Hodgkin-Huxley model, the most important simplification in the LIF neuron implies that the shape of the action potentials is neglected and every spike is considered as a uniform event defined only by the time of its appearance. It can be modelled as a RC circuit and a comparator. The neuron receives a train of impulses coming from pre-synaptic neurons, represented as an input current $I(n)$ charging the capacitor. The mathematical model of the membrane potential $u(t)$ in the LIF is described by a single first-order linear differential equation [1]:

$$C\frac{\mathrm{d}}{\mathrm{d}t}\big(u(t)\big) = -\frac{1}{R}\cdot(u(t) - u_{rest}) + I(t) \tag{2.2}$$

$$\tau_m\frac{\mathrm{d}}{\mathrm{d}t}\big(u(t)\big) = (u_{rest} - u(t)) + RI(t) \tag{2.3}$$

where τ_m = RC is taken as the time constant of the neuron membrane, modelling the voltage leakage. Additionally, the firing time t^f of the neuron is defined by a threshold crossing equation $u(t^f) = \theta$, under the condition $u(t^f) > 0$. Immediately after t^f, the potential is reset to a given value u_{rest} called resting potential.

2.2.2 Izhikevich's neuron model

In the class of spiking neurons defined by differential equations, the two-dimensional Izhikevich (IZ) neuron model is a good compromise between biophysical plausibility and computational cost [1]. It is defined by the coupled equations:

$$\frac{\mathrm{d}}{\mathrm{d}t}\big(u(t)\big) = 0.04u(t)^2+5u(t)+140-w(t)+I(t) \qquad \frac{\mathrm{d}}{\mathrm{d}t}\big(w(t)\big) = a(bu(t)-w(t)) \tag{2.4}$$

with after-spike resetting: if $u \leq \theta$ then $u \leftarrow c$ and $w \leftarrow w+d$. This neuron model is capable to reproducing many different firing behaviours that can occur in biological spiking neurons.

2.2.3 Synaptic plasticity and STDP

In all the neurons models, most of the parameters are constant values and specific to each neuron. The exception are synaptic connections that are the basis of adaptation and learning. *Synaptic plasticity* refers to the adjustments and even formation or removal of synapses between neurons in the brain [1]. From the biological point, it is possible to summarize changes of synaptic weights in two types: those with effects lasting several hours are referred as Long Term Potentiation (LTP) or Long Term Depression (LTD), if the weight values are gotten stronger or weaker; instead, those weight changes in the second or minute time scale are denoted as Short Term Potentiation (STP) and Short Term Depression (STD). One important finding that is receiving increasing attention is Spike-Timing Dependent Plasticity (STDP), which is a form of synaptic plasticity sensitive to the precise timing of spike firing relative to impinging pre-synaptic spike times. A basic computational rule has come out: a maximal increase of synaptic weight occurs on a connection when the pre-synaptic neuron fires a short time before the post-synaptic one, whereas a late pre-synaptic spike, just after the post-synaptic firing, leads to decrease the weight [1]. If both pre and post spikes are too temporally distant, the weight remains unchanged. For computational purposes, STDP is most commonly modelled in SNNs using temporal windows for controlling the weight LTP and LTD that are derived from neurobiological experiments. Different shapes of STDP windows have been used in recent literature: They are smooth versions of the shapes represented by polygons in fig.2.5.

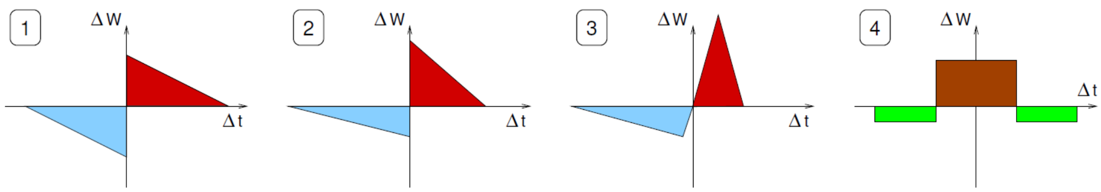


Figure 2.5. Various shapes of STDP windows with LTP in blue and LTD in red for excitatory connections (windows 1 to 3). Standard Hebbian rule (window 4) with brown LTP and green LTD are usually applied to inhibitory connections [1].

$X - axis$ is the spike timing, which is the difference $\Delta t = t_{post} - t_{pre}$ of firing times between the pre and post synaptic neurons. The synaptic change ΔW on $Y - axis$ operates on the weight update. For excitatory synapses, the weight w_{ij} is increased when the pre-synaptic spike is supposed to have a causal influence on the post-synaptic spike, i.e. when $\Delta t > 0$ and close to zero (pictures 1-3 in fig.2.5) and decreased otherwise [1]. For inhibitory synaptic connections, it is common to use a standard Hebbian rule, just strengthening the weight when the pre and post synaptic spikes occur close in time, regardless of the sign of time difference $t_{post} - t_{pre}$.

2.2.4 Polychronization

Nowadays, a growing empirical evidence is that neurons can generate spike-timing patterns with millisecond temporal precision: patterns can be found in the firing sequences of single neurons or in the relative timing of spikes of multiple neurons forming a functional neuronal group [7].
Indeed, if two or more neurons have a common post-synaptic target and fire synchronously, then their spikes arrive to the target at the same time, thereby evoking potent post-synaptic responses. Considering axonal conduction delays are negligible or equal, if neurons fire asynchronously, their spikes arrive to the post-synaptic target at different times evoking possibly only weak or no response.

Nevertheless, depending on the type and location of the neurons, axonal conduction delays could be relevant: for this reason, *Polychronization* takes into account this aspect. Since the firings of neurons are not synchronous, but time-locked

to each other, it is possible to refer to such groups as *polychronous*, where *poly* ($\pi o \lambda \upsilon \varsigma$) means many and *chronous*($\chi \rho o \nu o \varsigma$) stands for time or clock in Greek. Moreover, *Polychrony* should be distinguished from asynchrony, since the latter does not imply reproducible time-locking pattern, but usually describes noisy random non-synchronous events.
Finally, whenever the neurons do fire with the spike-timing pattern determined by the connectivity and delays, the group is *activated* and the corresponding neurons *polychronize*.

2.2.5 Computational power of neurons and networks

Information processing in spiking neuron networks is based on the precise timing of spike emissions (pulse coding) rather than the average numbers of spikes in a given time window (rate coding) [1].
In particular, SNNs add a new dimension, the temporal axis, to the representation capacity and to the processing abilities of neural networks. It is possible to describe different approaches to establish the computational power and the complexity of SNNs thinking on how to exploit these properties. In 1997, Maass [1] proposed to classify neural networks models as follows:

- *1st generation*: Networks based on McCulloch & Pitts' neurons as computational units, i.e. threshold gates, with only digital outputs.
- *2nd generation*: Networks based on computational units that apply an activation function with a continuous set of possible output values, such as sigmoid or polynomial or exponential functions (e.g. MLP, RBF networks). The real-valued outputs of such networks can be interpreted as firing rates of natural neurons.
- *3rd generation*: Networks which employ spiking neurons as computational units, taking into account the precise firing times of neurons for information coding.

This thesis work is based on the hardware design of the *HEENS* architecture, that emulates neural networks models belonging to the 3rd generation.

2.3 SNN Architectures

Nowadays, simulation of complex biological neural systems is a trending research area. Many implementation of SNN models are adopted, each one with its specific advantages and disadvantages; it is possible to report some of these different approaches.

Analog implementations exploit transistor's sub-threshold range operations to create compact and high-speed processing neural simulators: *BrainScaleS* is one of the most prominent projects in full custom analog design to simulate exponential integrate-and-fire neurons. These implementations offer extremely low area and energy consumption for very large-scale networks. [8] However, they are difficult to program and to scale, and they have high manufacturing costs and they require time to be designed and to be tuned. Therefore, the full custom analog implementations could be useful for those applications where the behaviour of the SNN is very well defined and characterized [6].

In contrast to analog-based solutions, most digital implementations are less expensive and more flexible. Currently many of them use general-purpose multiprocessors, Graphical Processing Units (GPUs) or FPGAs. These digital architectures offer wide range of flexibility and reconfigurability to process large scale SNN models at high speed [6].
One of the recently highlighted multiprocessor-based SNN simulators is *TrueNorth*, which implements LIF neurons with high number of synapses without plasticity [9]. *SpiNNaker* is another well-known multiprocessor-based SNN simulator: its programmable feature allows SpiNNaker to support different SNN models at the cost of highly complex processing cores [10].
GPU cards can provide a powerful solution when highly parallel computing is required, so several advanced GPU-based simulators have been proposed during the last ten years: one of them is *NEST* (Neural Simulation Tool) that supports several neural and synaptic models. Its low degree of biophysical detail has been a critical issue [11].
FPGA-based SNN simulation has been proposed and implemented in several works [6]. Zamarreno-Ramos proposed a scalable–reconfigurable neuromorphic Address Event Representation (AER) configured as 2D mesh. They claim that the proposed architecture is capable of managing spike traffic using routing approaches in a single or multiple FPGAs. The architecture simulates simple IF neurons to perform the convolution operation that is used in image processing (character recognition), but neurons do not involve plasticity [12].

A recent innovation in the field of SNN is *Loihi*, a neuromorphic chip fabricated in Intel's 14-nm process, having a total of 130,000 artificial neurons and 130 million synapses. It integrates a wide range of novel features for state-of-the-art modelling of SNN in silicon: hierarchical connectivity, dendritic compartments, synaptic delays, and, most importantly, programmable synaptic learning rules [13]. The chip is implemented as a many-core mesh, each of which houses a learning engine that can support a variety of machine learning models.

The FPGA digital implementations trade off model flexible and high speed processing. Instead, the GPU and general purpose multiprocessor approaches seem to have the flexibility to implement several SNN models and the scalability to implement fairly large-scale networks. The problem is that all these implementations rely on a general purpose Instruction Set Architecture (ISA) and on chip communication to simulate SNN: evidently, there would be some performance loss and power consumption because of certain functionalities that are useless for SNN simulation. Finally, *Spiking Neural Networks for Versatile Applications* (SNAVA) is an example of special purpose architecture that could not have this kind of performance loss and power consumption: it is the predecessor of *HEENS*, the architecture which this thesis work is focussed on and whose features are described in chapter 3. The ISA of SNAVA has been tailored to SNN simulation to get the best out of the utilized hardware. SNAVA is a scalable and programmable parallel architecture that supports real-time, large-scale, multi-model SNN computation[6]. This parallel architecture is implemented in modern FPGAs devices to provide high performance execution and flexibility to support large-scale SNN models. Flexibility is defined in terms of programmability, which allows easy synapse and neuron implementation. This has been achieved by using a special-purpose Processing Elements for computing SNNs, and analyzing and customizing the instruction set according to the processing needs to achieve maximum performance with minimum resources. Its architecture is mainly composed of an array of SIMD (Single Instruction Multiple Data) units, a single control unit and a number of communication units that support software to configure and monitor the system in real time (1 ms time step simulation). In addition, SNAVA is scalable architecture and has the flexibility to be implemented into several chips forming a system in any topology of user's choice.

Chapter 3

Architecture Review

This project starts from the *HEENS* (Hardware Emulator of Evolvable Neural Systems) architecture, that has been developed by the Advanced Hardware Architecture group of the Department of Electronics Engineering of Universitat Politècnica de Catalunya (UPC). This architecture is an evolution of a previous one, called *SNAVA* ("Spiking Neural-Networks Architecture for Versatile")[6], and it works with a more efficient Processing Element array in terms of functionality and resource occupancy. HEENS is more versatile and it allows to load and run via software different models of neurons and change their synaptic interconnection dynamically without resynthesizing the project. Moreover, the versatility of this architecture allows to easily resize the array (number of PE and all the necessary connections for proper operation) using only two parameters: the number of rows and columns.

The work of this thesis project is focussed on the hardware part of the PE array, in particular on the improvement of the architecture in terms of scalability, frequency and resource occupancy, trying to increase the size of array by acting on these aspects. In this chapter there is explained the starting point of this thesis work, focussing in particular on the PE array design.

3.1 HEENS architecture

HEENS is an architecture designed for multi-FPGA implementations for SNN emulation in real time, which has been designed to support evolution network with a high degree of configurability, with a spike communication scheme called AER ("Address Event Representation"), that can be hierarchically extended. It allows several chips to be interconnected in a ring topology (Fig. 3.1) in a Master/Slave and point-to-point hybrid communication scheme. Chip Master (MC) takes control of the network to configure the ring and the neuronal application in all the nodes and it controls the dynamic on-line reconfiguration of each node[14].

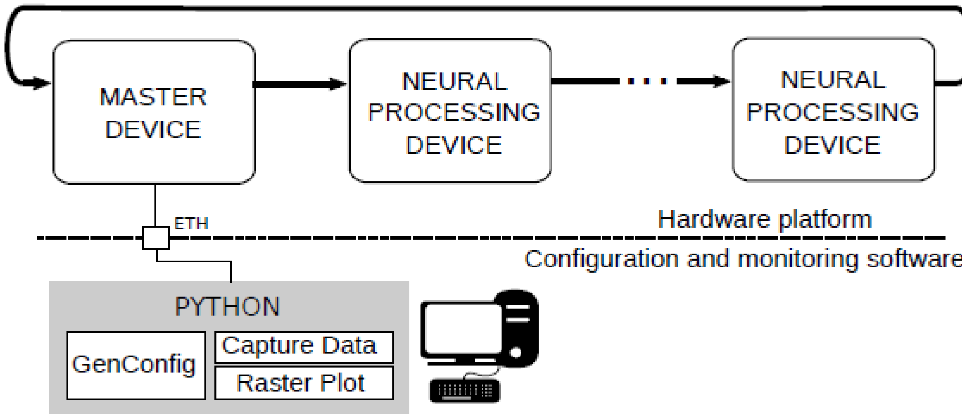


Figure 3.1. HEENS architecture consisting of a Master Chip (MC) and Neural processing devices (NDs) connected in a ring [2].

HEENS bases its processing sequence by emulating the biological behavior of neurons, for which it divides its processing into three phases of operation (as shown in Fig. 3.2)[14]:

- Initialization Phase (IPh): the ring that makes up the multi-chip platform is configured, so it is required to identify each node (ID) and the ring size.

- Execution Phase (EPh): this phase has biological correspondence with the soma. Here the neuronal algorithm is processed by calculating and updating the state variables. Each neuron uses individual parameters for its processing. The start and end of this phase is marked by an internal control signal called *eo_exec*.

- Distribution Phase (DPh): it manages the propagation of neurotransmitters through the synapses. Spikes obtained in the EPh are propagated and delivered to the destination neurons located in the same or in different NDs.

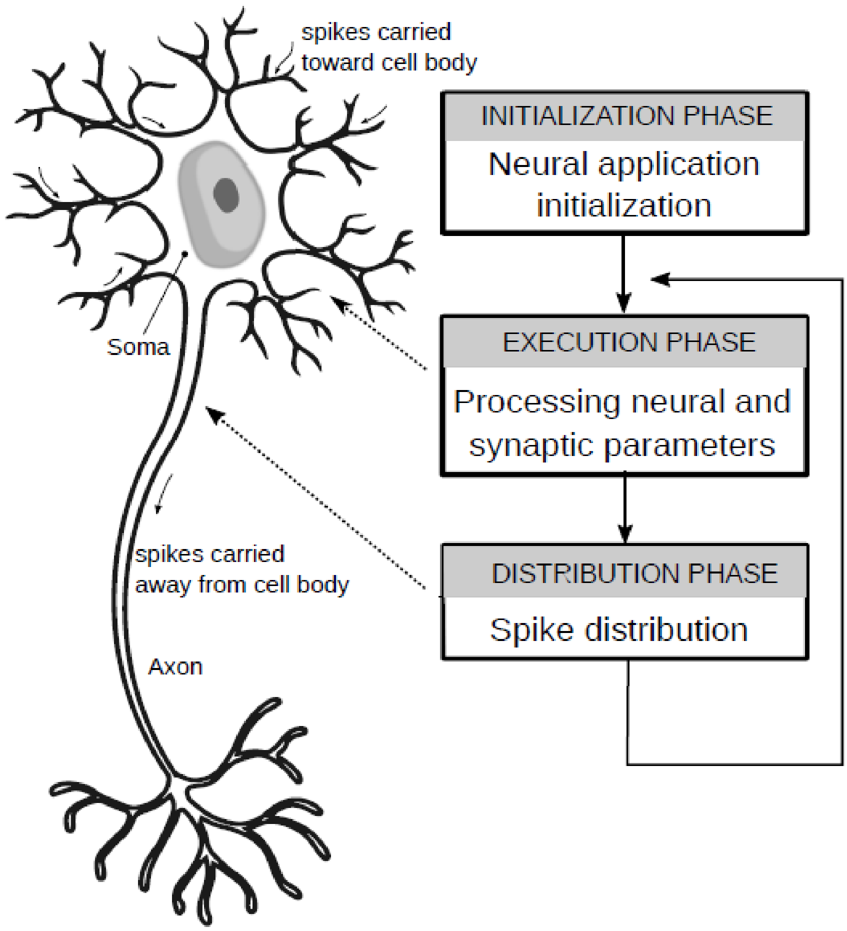


Figure 3.2. HEENS processing stages[2].

The multiprocessor architecture is illustrated in Fig. 3.3 and it corresponds to a ND of the neural network. This performs the processing of neuronal activity programmed by the user and it uses a Single Instruction Multiple Data (SIMD) computation scheme with a single control unit to achieve data level parallelism in the PE array. In fact, each processing elements disposes local memories, holding the modelling parameters of each specific neuron: this means each processor has to run the same set of operations with different local parameters along, allowing the use of a single programmable control unit. This technique is suitable for the implementation of neural networks because it reduces area costs and offers high computational performance, reducing dramatically the resource requirements compared to GPUs or architectures that makes use of complex general purpose cores.

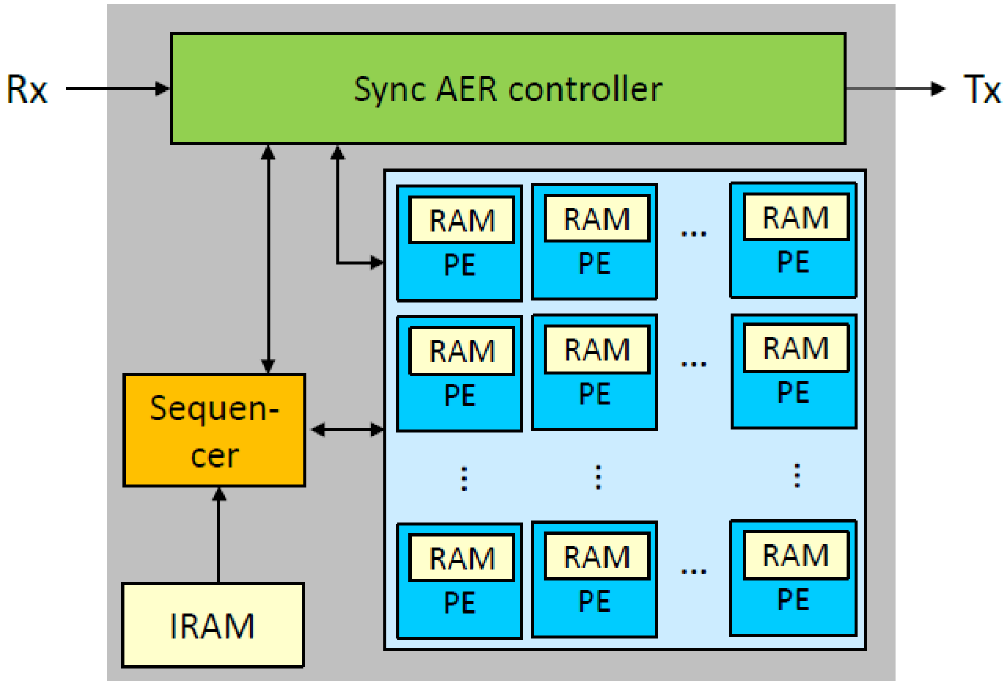


Figure 3.3. Block diagram of HEENS multiprocessor architecture[3].

The main blocks that comprise it and described below are:

- Communication buses: the address and data buses that allow the flow of information to the array are multiplexed between those that transmit configuration packets and those that deliver execution informations corresponding to the opcodes dispatched by the Sequencer and spike events delivered by the controller. Selection of these is given by the internal signal called *config* that is activated depending on the phase that is being processed.

- Control Unit (Sequencer and Instruction-RAM): HEENS-MP is a Harvard type architecture. The instructions are read from a single memory (IRAM). Each PE has its local data memory (SNRAM) where it stores the synaptic and neuronal parameters of each neuron. Sequencer enables the distribution signal *en_spike* after the Execution Phase (EPh), during which it sends instructions to all processors at the same time and all of them calculate the membrane potential in function of the input spikes, synapses and initial configuration. If a neuron spikes, this information is stored in the *spike_out* register, that is located inside the PE. It has a size equal to the chosen virtualization level.

- AER-SRT Controller: it brings the capability of interconnect more FPGAs in a synchronous ring topology (SRT) to increase the number of neurons emulated. Once the algorithm has been processed in the PEs, a row sweep is performed to obtain the post-synaptic spikes triggered by the neurons when the membrane voltage has exceeded the threshold voltage. The spikes generated at each execution cycle are encoded in address events and stored in a FIFO waiting to be transmitted during the DPh.

- PE array: it is the matrix of PE that composes a Neural processing device. More details are explained in section 3.2.

This thesis work is focussed on the PE array block.

3.2 Processing Element array

The array shown in Fig. 3.3 is a 2D array of PEs: its size is parametrizable according to the numbers of rows and columns of the array and the virtual level, which allows to make a 3D array if the level is higher than 0. Due to the hierarchical and modular nature of HEENS, the PEs process two types of spikes: local (generated in the same ND) and global (those coming from others NDs or from the MC).

In Fig. 3.4 the block diagram of PE is shown; the internal components correspond to an Harvard type architecture and the main units that constitute it are [14]:

- ALU: it supports 16-bit fixed point arithmetic and logical operations. *Opcode* is the 6-bit control signal used to choose witch operation ALU has to do. The Carry (C) and Zero (Z) status flags are accessible to the user.

- Register File (R0-R7): a bank of 8 general purpose 16-bit registers that interacts directly with the ALU through the accumulator register R0. Moreover, there is also a shadow registers bank (SR0-SR7), whose access is managed by the Sequencer, used to extend the space of storage of neuronal or synaptic variables. Through the *reg_code* control signal it is possible to manage *data_in* inputs (eight 16 bit inputs) in different way. Tab. 3.1 shows the possible controls of the register file:

Regcode	Function
000	R_x <= Data_in_x
001	reset if EN_x is asserted
010	set if EN_x is asserted
011	swap with respective shadow register if EN_x is asserted
100	$SR_x <= R_x$ if the register is enabled
101	$R_x <= SR_x$ if the register is enabled

Table 3.1. Control values of the Register Bank.

where R_x and SR_x are respectively the register and the shadow register in position x, EN is an 8-bit control signal that enables each register.

- Synaptic/Neural Memory (SNRAM): it is the data memory in charge of storing neuronal and synaptic parameters, seeds of LFSR block and any other data of the neurons processed by the PE in each virtual level.

- Local and Global memories (BRAM): this block is composed by a block of local memory (where the interconnectivity between local neurons is modelled) and an associative scheme that allows the decoding of global connections. The global memory block models the interconnection between neurons of different NDs: each PE has $s_G - 1$ global synapses processed at the main level (VIRT = 0) to emulate a hierarchical connection characterized by sparse connectivity between clusters.

- Virtualization (VIRT): this block emulates more than one neuron per PE per execution cycle. The virtual level is the parameter that defines n pipelined virtualization levels plus a main level, which is VIRT = 0. implementing virtualization is one of the targets of this work.

- Pseudo Random Generator (LFSR): the seed of this 64-bit register is defined by software with the SEED instruction, and it is stored in the SNRAM memory. This register allows generating uncorrelated noise for each PE.

- Freeze LIFO: FREEZE instructions are linked to the Carry and Zero flags and they use a LIFO for their execution, to stack up to eight levels for nested conditions.

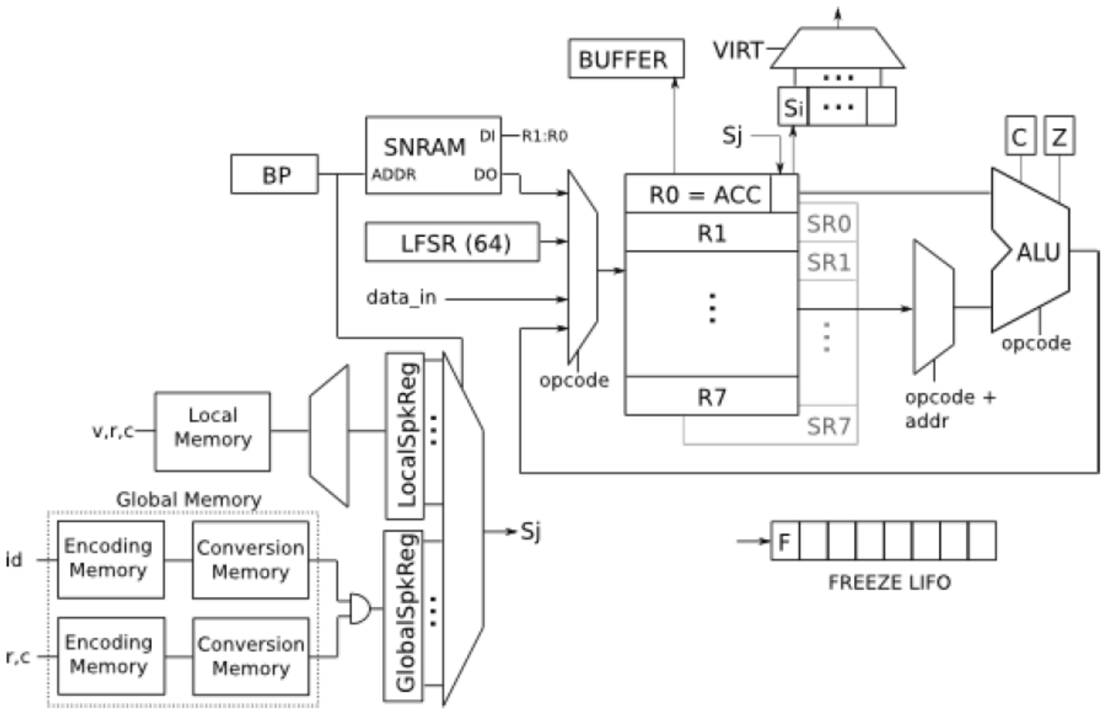


Figure 3.4. Processing Element block diagram [4].

3.3 IF Assembly Code

HEENS architecture is able to emulate the leaky Integrate and Fire (IF) algorithm, which is one of the most used models for emulation SNN. The idea of this model consists on computing he membrane potential as a function of the receiver spikes through the predefined synapses. The assembler code (see Appendix B.1) runs the IF algorithm: it takes into account specific key values of membrane potential, taken from different biological studies in literature. These values are shown in the table 3.2:

Membrane Potential	**Hexadecimal value**	**Potential value** $[10^{-5}V]$
V_{REST}	FFFFE4A8	-7000
V_{THRES}	FFFFEA84	-5500
V_{DEPOL}	FFFFE0C0	-8000
V_{ACT}	00001771	+1000

Table 3.2. Membrane potential key values.

where:

- V_{REST} is the Resting potential, equal to -70 mV;
- V_{THRES} is the Threshold voltage, equal to -55 mV;
- V_{DEPOL} is the Depolarization voltage, equal to -80 mV;
- V_{ACT} is the Action potential, equal to 10 mV.

When the algorithm starts, after random initialization, using the subroutine *LOAD_NEURON* (Appendix B.1), membrane potential of each PE is stored in their respective R2 register and the synapse weight into R1. Then, with the *MEMBRANE_DECAY* subroutine, the membrane potential decay is calculated using a time constant τ_{decay}, which is less but close to 1. In fact, storing the current membrane potential in R2 into the accumulator (MOVA R2) and resting potential in R4 (LDALL R4, V_{REST}), the algorithms does the following computation:

$$V_{mem}^{'} = (V_{mem} - V_{REST}) \cdot \tau_{decay} + V_{REST} \tag{3.1}$$

where $V_{mem}^{'}$ is the new membrane potential after decay.

Every time a neuron receives a spike through one of its synapses, its V_{mem} increases or decreases according to the specific excitatory or inhibitory synapse weight. For this reason, it is necessary a loop (*LOOP tot_synapses*) in which all synapses of each neuron are read in order to detect possible incoming spikes. This loop implements the following calculation to obtain the new membrane potential:

$$V''_{mem} = V'_{mem} + \sum_{k=0}^{n-1} s_k \cdot w_k \tag{3.2}$$

where n is the total number of synapses, w_k is the synapse weight of the k-th synapse and s_k is a value equal to 1 or 0, respectively if a spike impinges that synapse or not. w_k is stored into R1 register and the spike s_k into the LSB of the accumulator (*LOADSP*). Each time a sum is computed, the accumulator is reset (*RST ACC*), the pointer *BP* of the SNRAM, where all the synaptic, neural and some common parameters are loaded, is increased and new parameters, that will be used for the next cycle, are stored inside its respective registers R0 and R1.

Subsequently, after having calculated the new V''_{mem} for each neuron, through the *DETECT_SPIKE* subroutine the algorithm detects during each execution phase if the membrane potential of some neurons has exceeded the threshold voltage, thus producing post-synaptic spikes. This value is stored in R0 (LDALL ACC, V_{THRES}) and it is compared with the current membrane potential in R2, doing a subtraction and checking the Carry Flag of ALU: if it is positive, a spike is generated. In this way, the IF model behaviour is reproduced. Once the spike is detected, the spiking neuron has to be discharged, returning to the initial value of its membrane potential V_{mem} equal to V_{REST} (LDALL ACC, V_{REST}).

Finally, *STORE_NEURON* subroutine is executed: the SNRAM is uploaded storing R1 and the final membrane potential (STORESP) and the pointer *BP* is increased. Now execution phase ends, spikes are distributed (SPKDIS) and the loop can start again.

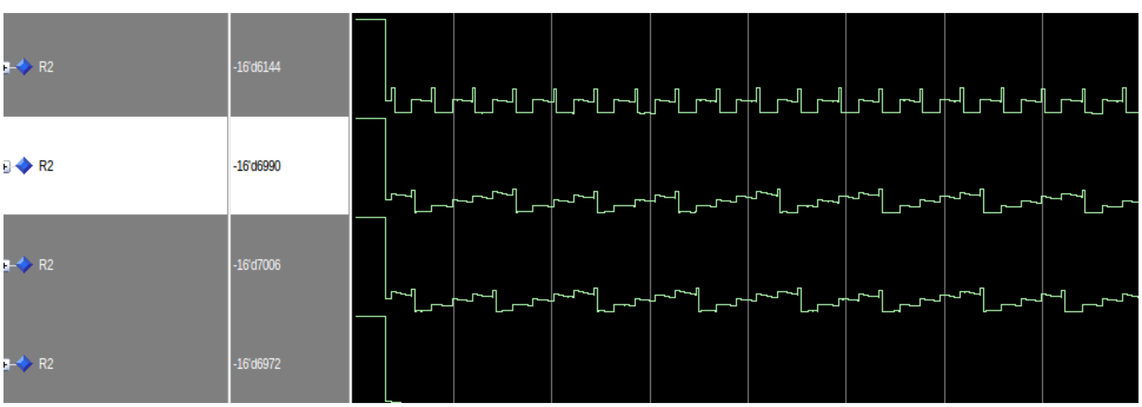


Figure 3.5. Simulation of the three neuron's membrane potentials trend as a function of time done with *QuestaSim* software.

The *HEENS* architecture has been described in synthesizable VHDL. In Fig.3.5

it is showed the R2 register values of four PEs using the analog format of *QuestaSim* software. Each time the membrane potential V_{mem} of a neuron exceeds the threshold voltage, a spike is generated and V_{mem} resets itself, returning to have a value equal to about the resting potential V_{REST}. Moreover, every time a neuron receives a stimulus through one of its synapses, V_{mem} increases and it decays continuously towards V_{REST} with the τ_{decay} constant.

HEENS architecture allows also to change easily the topology of the model: interconnections between neurons (i.e. PE) are described using netlist files, as ones reported in Appendix C and whose explanation is reported in the following chapter, and stored into associative memories. Moreover, it is possible to set the initial membrane potential V_{mem} for each neuron: in this way it is feasible to decide if a neuron, at the beginning, has a spike and any kind of possible spike paths can be described.

Chapter 4

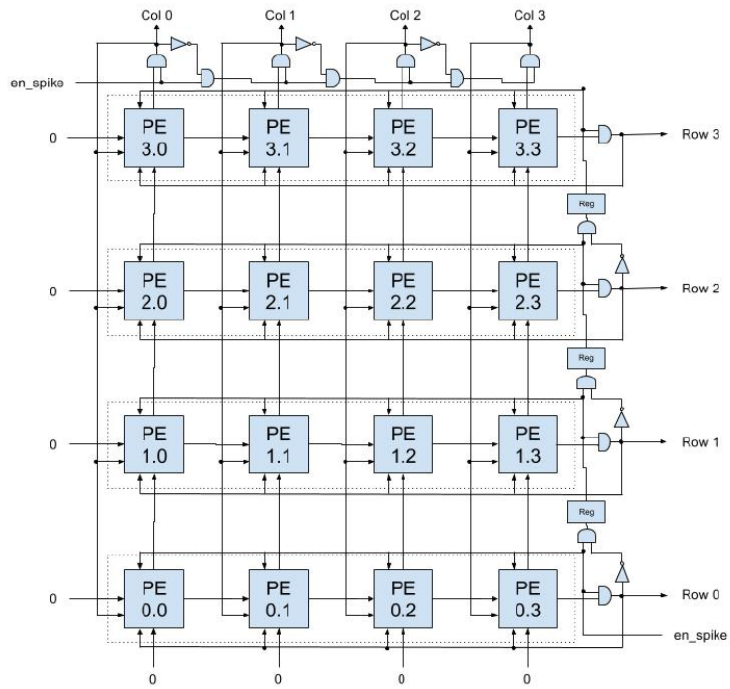
Pipelining and Extension of the Multiprocessor Array

In this chapter there are introduced the first improvements of the architecture in terms of time constraints and area occupancy: designing a new path of the spike distribution, the goal is to extend the array as much as possible without having problems of negative slacks. Simulations are done with *QuestaSim* software, using as assembler code the one in Appedinx B.1. Regarding the initial membrane potential, for all the following simulations it is assigned a V_{mem} initiated at -60 mV (10 mV above resting potential) to all PE except PE[0;0], which is initiated at -40 mV: in this way the PE[0;0] will fire, because its values is above V_{THRES} = -55 mV, and any neuron having a synaptic connection with it will fire too as consequence, due to their membrane potential which is slightly below the threshold.

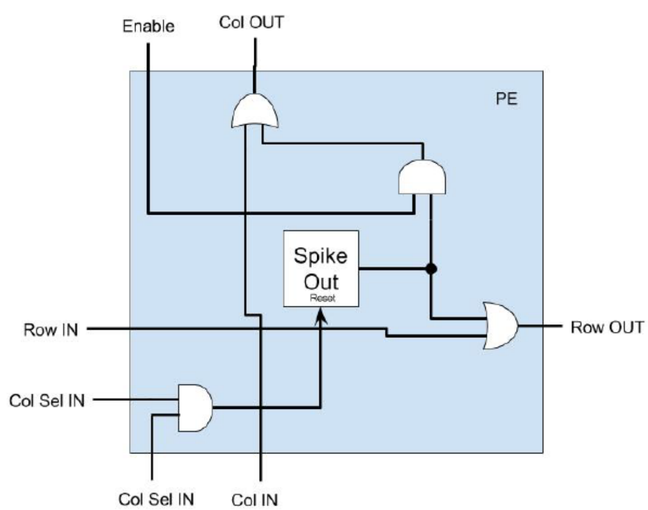
4.1 Spike pipeline

The first goal of this project is to modify the current architecture in order to make a new version of the multiprocessor array in which the critical path during the Distribution Phase (DPh) of spikes is reduced.

The previous architecture is shown in Fig. 4.1.



(a) *Multiprocessor array*



(b) *Processing element*

Figure 4.1. Previous multiprocessor array and PE block diagrams [4].

In this architecture, spike distribution follows a combinational path with priority: spikes generated by each PE are processed and forwarded to the neighbour processors. Actually, processor in position [0,0] (row 0 and column 0) has the highest priority, so if it produces a spike, it will be the first to leave the array and be resetted. Then, PE in position [0,1], having the second highest priority, continues to transmit spikes when all the previous ones were clean and so on. This cycle does not end until the sequencer disables distribution signal *en_spike*. To reduce the combinational circuit, pipeline register are inserted at the end of each row.

The weak point of this architecture is that, if the size of the array increased, the critical path would be too long and it could violate timing constraints. Since it is not optimal to reduce clock frequency, which is set at 125MHz, because the other parts of the system have not problems, the proposed solution is to introduce pipeline stages in order to keep constant the clock frequency. The new architecture designed is shown in Fig. 4.2: the priority scheme is replaced by a simpler architecture, where several signals involved before in the priority scheme are deleted. Now the distribution phase is managed shifting spikes that may be present inside PEs.

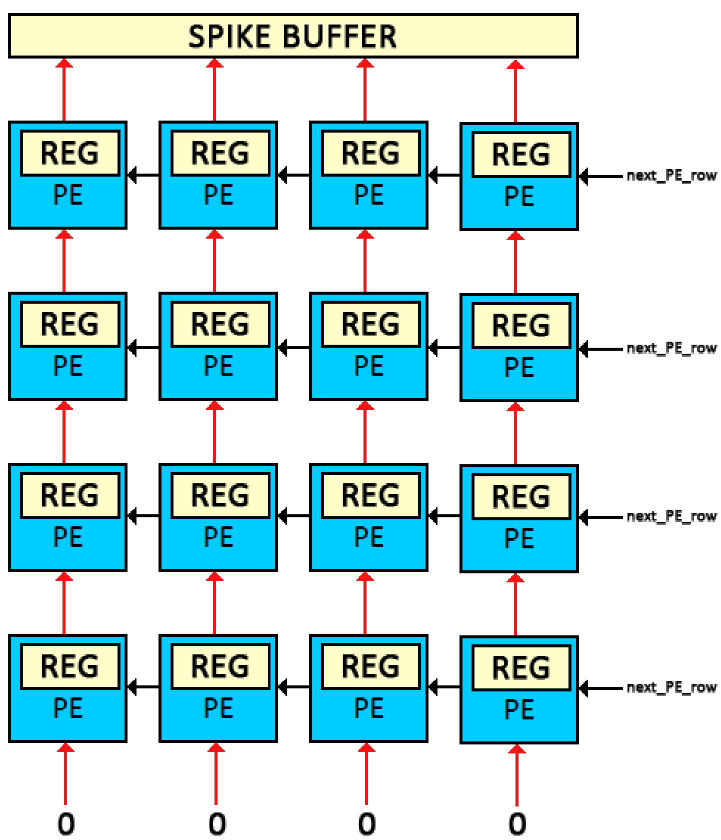


Figure 4.2. Block diagram of the new pipelined architecture of the multiprocessor array. REG corresponds to the spike_out register.

When distribution starts, the highest row is stored in the spike buffer and each one below it scrolls in the corresponding row above. Spike buffer could contain at most a number of spikes equal to the column number (i.e. if all the PEs of the highest row have a spike, the buffer will contain all 1s), so the block requires to transmit all spikes of the row in question as many clock cycles as the number of PE owning a spike. To do this, spike buffer checks the position of 1s, then it transmits the first one detected and it resets that position. To transmit a spike, the PE array block (see appendix D.3 for the VHDL description) enables the *spike_valid* signal and it loads *col_sp* and *row_sp* signals, which are the position of the spike to be distributed. When the buffer has only zeros (which is both the case where no PE has a spike or all the spikes of the row have been transmitted), the signal *next_PE_row* is enabled and it allows to shift the spike lines. A down-counter, called *PE_array_count* with initial equal to the row number, is used to know which row is loaded into the buffer and to end the transmission when it reaches the zero.

The propagation of spikes during the distribution phase is done generating all PE addresses that have a spike. 3 signals are involved to generate the address:

- *spike_valid*: it is a control signal asserted when it is detected a 1 into the row loaded in the buffer. If there is a spike, that position is resetted and the address is generated in order to distribute it.
- *col_sp*: it is the column address of the PE that has fired a spike. When *spike_valid* is asserted, it means that a spike is detected into the buffer. Therefore, the column address of that spike is loaded in order to transmit it.
- *row_sp*: it is the row address of the PE that has fired a spike. When *spike_valid* is asserted, this signal is equal to *PE_array_count*, that is the down-counter position taking into account which row is loaded in the buffer and that is controlled in order to check if there are 1s.

Finally, distribution phase terminates when *PE_array_count* ends its count (it reaches the 0) and all the spikes in row n.0 are transmitted. When these two conditions happen, the control signal called *eo_spike* is asserted and sent to the sequencer.

Regarding the architecture of PEs, the *spike_out* register, used to store the possible spike during the execution phase, is used as pipeline register to store spikes of the rows below. *Spike_out* has a size equal to the virtualization level, so if there is a virtualized array, this shift will involve it too. Now each processing element has only 3 signals used to manage the distribution phase (see appendix D.1 for the VHDL description):

- *en_spike_tx*: it is the control signal referred to the *next_PE_row* signal in the *PE array* block. When a shift of the spikes is required, this signal is asserted and the *spike_out* register is uploaded with the spike of the corresponding PE present in the row below.

- *spike_out*: it is the spike output signal. When *en_spike_tx* is enabled, the spike of the PE is sent to the PE above using this signal.
- *spike_in*: it is the spike input signal. When *en_spike_tx* is enabled, *spike_out* register is updated with this signal, that is the spike of the PE below.

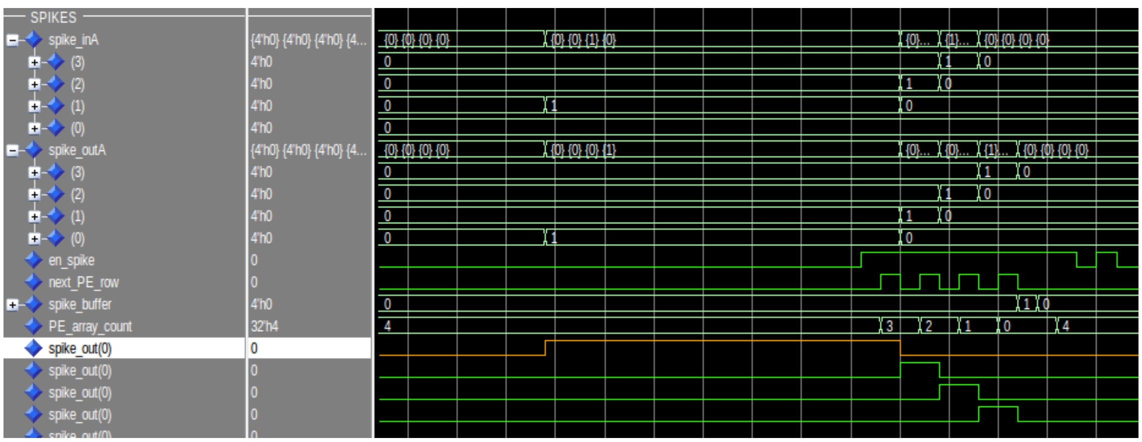


Figure 4.3. Simulation with QuestaSim of a spiking PE (orange line).

In Fig. 4.3 it is illustrated a simulation of a 4x4 array without virtualization: it means that VIRT=0, so the only position of *spike_out* register to take into account is number 0. In particular it is shown the distribution phase of a spiking neuron: in fact the neuron in position [0,0] contains a spike (orange *spike_out(0)* signal) and, when *en_spike* is asserted, distribution starts. *Next_PE_row* is cycled 4 times because there are 4 shifts to do, so the counter resets in 4 cycles and it ends the distribution phase. Spike shifting is well displayed in the other *spike_out(0)* signals, which correspond to the PE in position [1,0], [2,0] and [3,0]: therefore spike [0,0] shifts along its column, until reaching the *spike_buffer*, which assumes a value equal to "0001" (1 in unsigned format on QuestaSim) at the 4^{th} clock cycle, meaning that the PE in column 0 has a spike.

In this particular case, only the neuron in position [0,0] is spiking: if there were other PEs containing a spike, distribution could last longer.

4.2 Array Extension

After that the problem related to the potentially too long combinatorial path has been solved, it is possible to extend the size of the PE array without encountering problems of set-up time. Simulation done so far are made with an 4x4 array, so the next goal is to increase the size solving possible bugs that could happen, since the architecture was tested to work with a 4x4 array.

Once the number of rows and columns is increased, it is important to create and compile a netlist file that describes interconnections between neurons: actually, implementing simple paths permits easily to verify the correct behaviour of the architecture. In appendix C there are reported the netlists used to do the verifications. The format used to describe nets is shown in table 4.1.

Source			**Destination**			
virtual_layer	row	column	row	column	synapse_number	synapse_weight

Table 4.1. Netlist format.

At this stage, virtualization is not yet implemented, so the virtual_layer is always set to 0 and there is not a virtual_layer destination, that has to be implemented when virtualization will be introduced. In addition to the location (row and column) of the source neuron, to the location of the destination one and to the synapse_number, that is the specific synapse of the destination neuron that links both ones, it is possible to decide the synaptic_weight. This field of the table is a decimal value that has to be written into R1 and R2 registers. Considering that weight value has to be into R1 and R0 and R0 are 16-bit registers, its corresponding binary value has to be left-shifted of 16 positions. In this way, the 16 less significant bits will be stored into the accumulator and the 16 MSBs, representing the synapse weight, will be stored into R0.

It is possible to create any kind of networks, linking one neuron to more than one, using different synapses. The first approach is to create a ring oscillator network, as the one shown in Fig. 4.4.

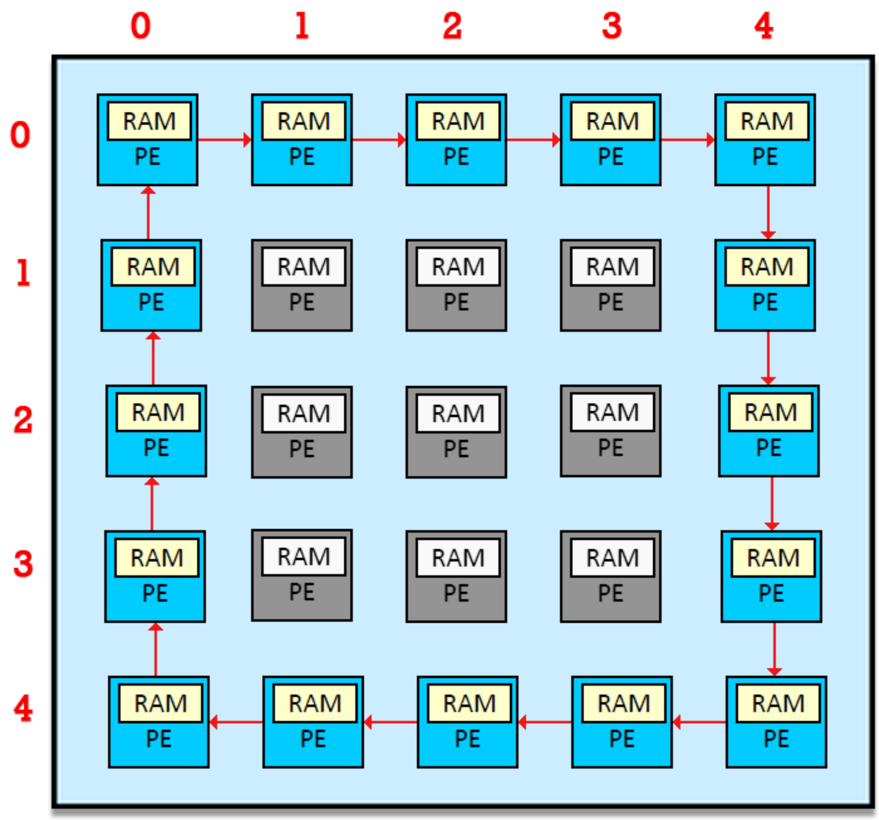


Figure 4.4. Block diagram of 5x5 ring oscillator network.

Using a ring oscillator as network is a good choice since there are few interconnections, so there are less signal to check and the distance between source and destination is the minimum one. It is important to configure the membrane potential in such a way that, if a neuron is stimulated by a single spike, it fires. If this condition is not satisfied, it is impossible to check if the spread of spike is correct, since if a neuron of the network receives a spike but it has not the sufficient membrane potential to fire, it will not produce a spike to send to the next neuron.

The first approach is to extend a bit the array to 5x5 PEs, to see if there are problems increasing the size more than 4x4. To display better the spike evolution, a signal called *spike_displ* is used instead of *spike_out*: actually, *spike_out* is used also as pipeline register during the distribution phase, so it displays also the spike shifting, and it does not make readable the spike evolution. Therefore, *spike_displ* has only visual utility, but it is not useful for the architecture functioning.
Simulation results are shown in Fig. 4.5. The distribution of spikes works as expected: pulses move along the first row and then they spread into the last column

and so on, closing the circle. It is possible to see that the distribution phase works well: the shorter impulses represent spike shifting along the respective column.

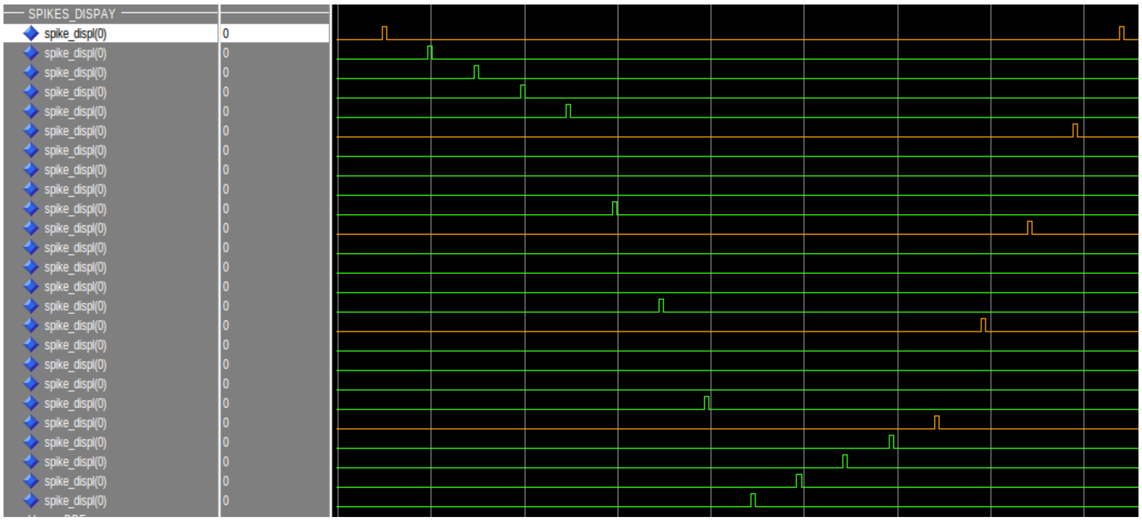


Figure 4.5. Simulation of 5x5 array using a ring oscillator configuration done with *QuestaSim* software. Row n.0 is in orange.

Then, the PE array size is extended to 10x10. In Fig. 4.6 it is shown the simulation result using a ring oscillator configuration as before.

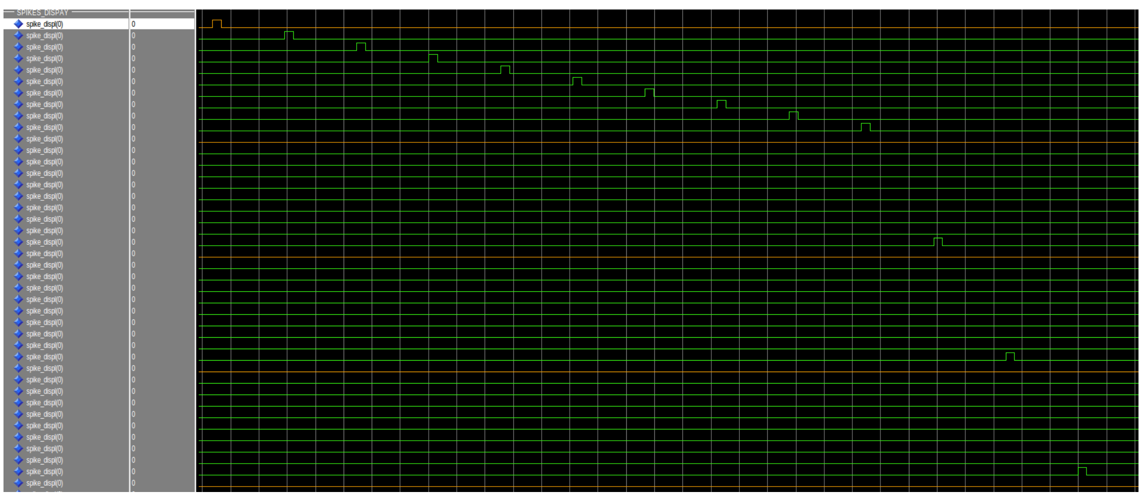


Figure 4.6. Simulation of 10x10 array using a ring oscillator configuration done with *QuestaSim* software. Row n.0 is in orange.

An other test is to check if the system supports an array size with a number of columns other than the number of rows. In Fig. 4.7 it is shown the simulation of a 9x7 array, which therefore has an odd number of columns and rows.

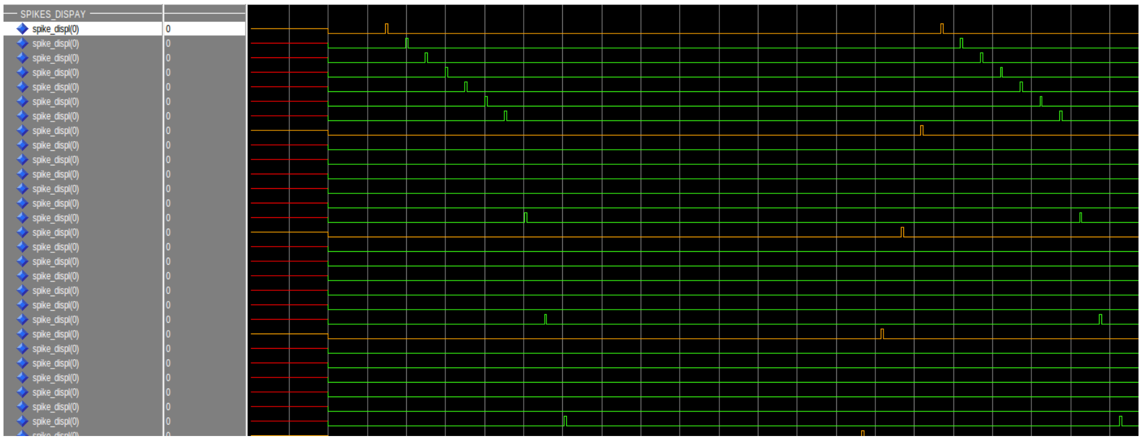


Figure 4.7. Simulation of 9x7 array using a ring oscillator configuration done with *QuestaSim* software. Row n.0 is in orange.

Finally, the PE array size is extended to 16x16, that is the possible maximum size reachable in this project, since the architecture is optimized to support this size at most . In Fig. 4.8 it is shown the simulation result.

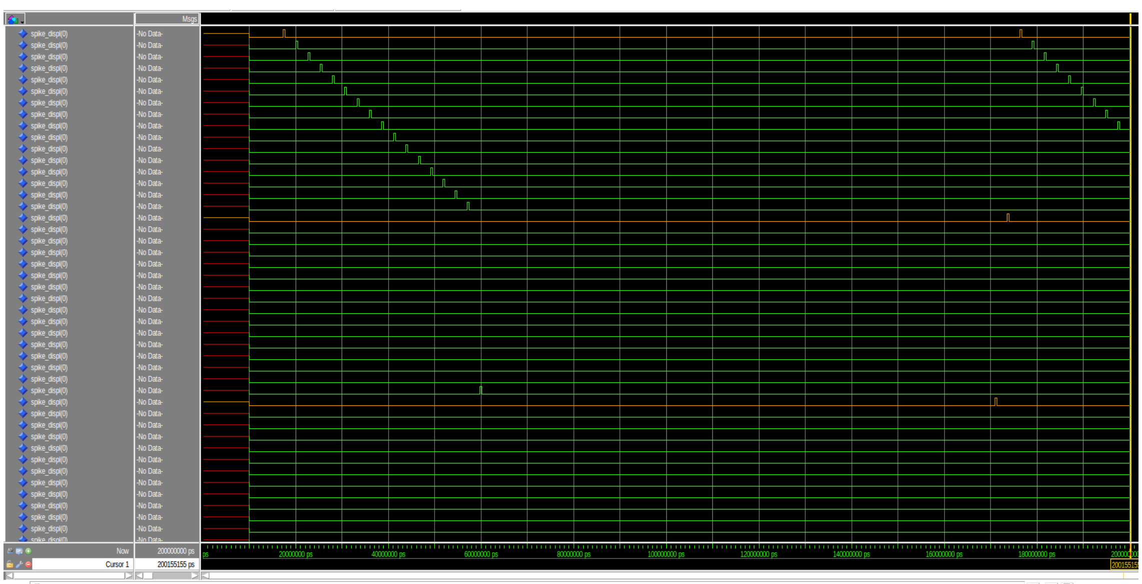


Figure 4.8. Simulation of 16x16 array using a ring oscillator configuration done with *QuestaSim* software. Row n.0 is in orange.

Since it is proved that the architecture works with ring oscillator network, the last test is to verify that it continues to work if the number of nets is increased. In Fig. 4.9 it is shown the netlist in appendix C.3. The path described, called "snake" for its shape, contains a lot of interconnections; moreover, distance between source-destination neurons is still the minimum one, so the simulation remains easily readable.

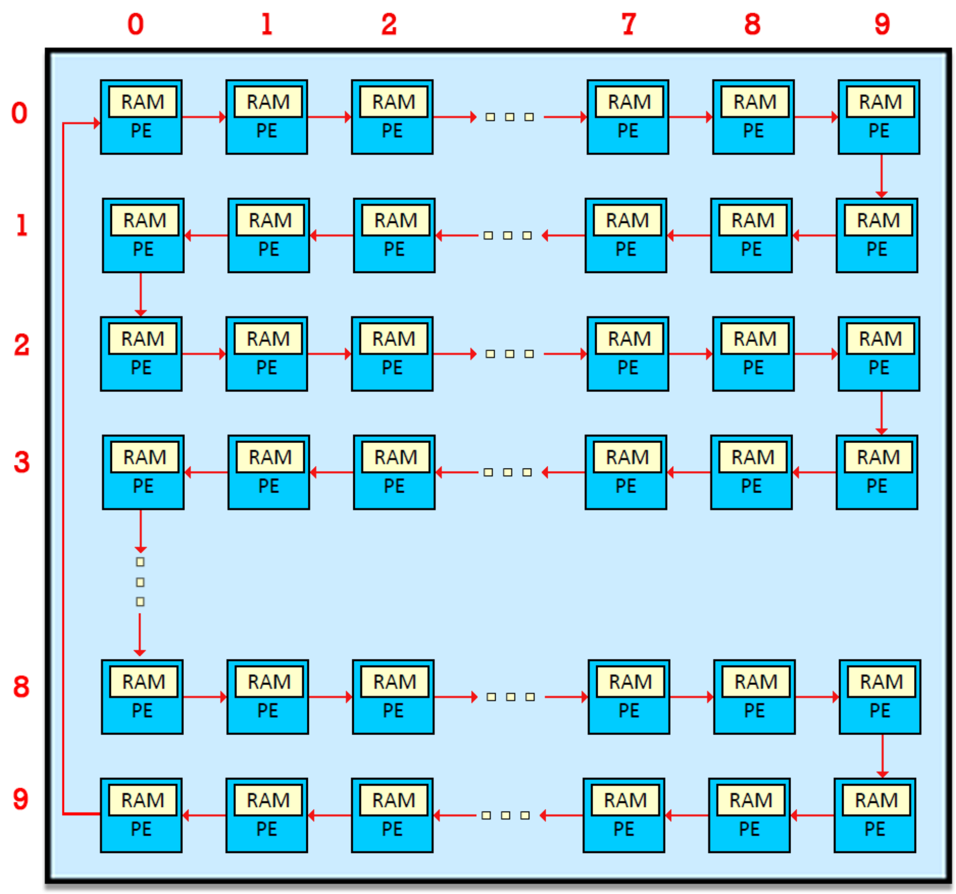


Figure 4.9. Block diagram of 10x10 ”snake path” network.

In Fig. 4.10 it is illustrate the snake path simulation. Also in this more complicate test, the distribution of spikes works as expected: pulses move along the first row and that they spread to the second one and so on. Also the distribution phase has not problems: even if a lot spikes need to be distributed, there are not time violations, so ti means that it is not necessary to modify the architecture speed (clock frequency is equal to $125MHz$).

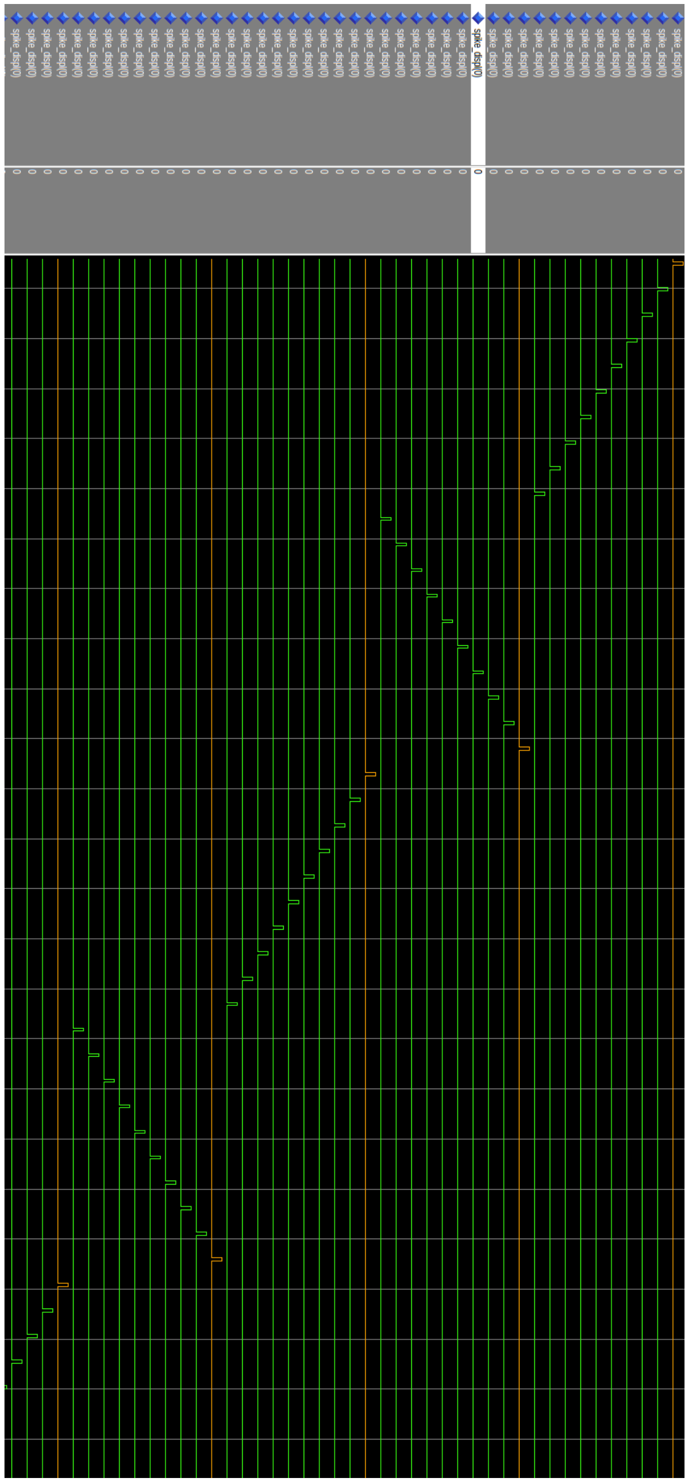


Figure 4.10. Simulation of 10x10 array using a "snake path" configuration done with *QuestaSim* software. Row n.0 is in orange.

4.3 Synthesis and Implementation

After simulating the new spike pipeline, the system has to be tested on a board. Originally, the system was implemented on a KC705 development board that contains a Xilinx XC7K325T Kintex 7 FPGA. The design has been migrated to an MPSoC device to enhance the system configurability and monitoring, thus design synthesis and implementation are done with *Vivado* software on Xilinx Zynq-7000 SoC ZC706, that is shown in Fig.4.11: the devlopment board contains a SoC FPGA device, that integrates both processor and FPGA architectures into a single device. Consequently, it can provide higher integration, lower power, smaller board size and higher bandwidth communication between the processor and FPGA: in particular, the Zynq®-7000 SoC family integrates the software programmability of an ARM®-based processor with the hardware programmability of an FPGA.

Figure 4.11. Xilinx Zynq-7000 SoC ZC706.

First step is the synthesis: 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, which are not detected in the simulation with *QuestaSim*. Also the utilization of the FPGA and its resources (Block RAM, Registers, LUTs...) are computed.

After the synthesis, the implementation has been done: it is a time-consuming process, especially if the array has high dimension, because the software generates all the components to be placed on the FPGA, with all the connections, taking into account the distance between registers and calculating the worst delay to determine if there are time problems.

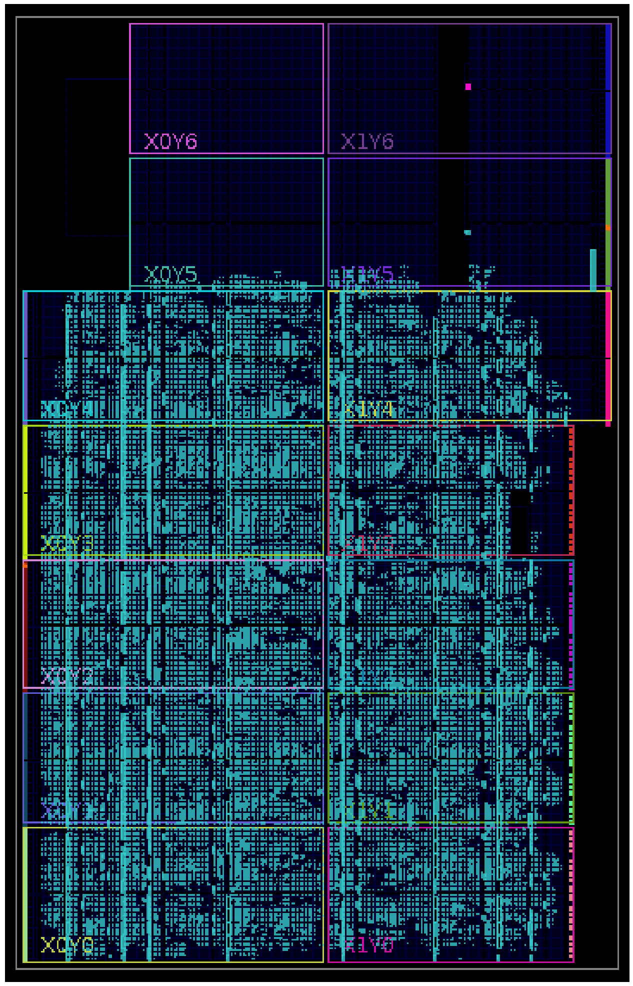


Figure 4.12. Floorplanning of a 10x10 array done with *Vivado* software.

In Fig.4.12 it is shown in blue the area occupancy of the 10x10 multiprocessor array: it is not the maximum possible one, because there is unoccupied space, so it means that on this board it is possible to go beyond 100 PEs.
Regarding the resource occupancy, in Fig.4.13 it is shown which are the components of the whole device that use more resources. Rather than logic, BRAMs represent the bottle neck of *HEENS*: with a 10x10 multiprocessor array, they fill 60% of available space, so this means that with this board it is possible to reach an array size equal to 12x12 or 13x13 in the best case. In these forecasts, the virtualization implementation is not considered: theoretically, it will not have much effect on area problems, because the number of PEs will not change and only few control signals and components (such as counters and registers) are planned to be implemented.

Power report, shown in Fig.4.14, also confirms that BRAMs are the most critical component, due to they are the biggest source of dynamic power consumption. Of course these figures are just presented as an orientation, since power estimation is highly dependent on the resource operation.

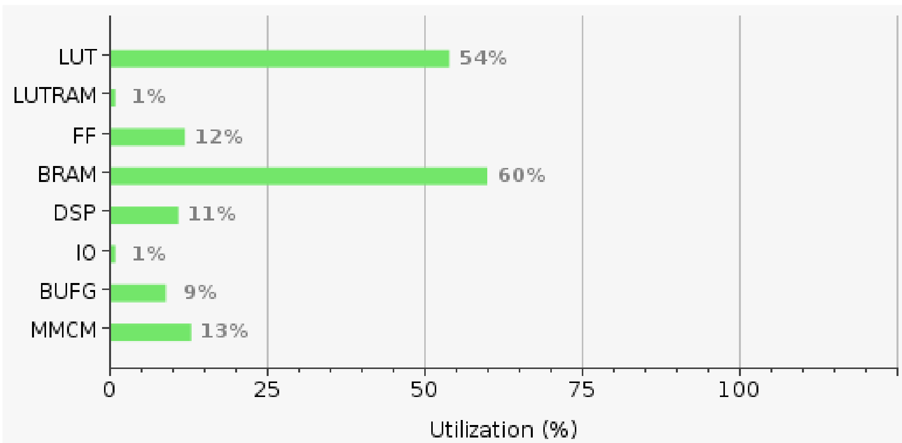


Figure 4.13. Area utilization of the whole architecture with 10x10 multiprocessor array done with *Vivado* software.

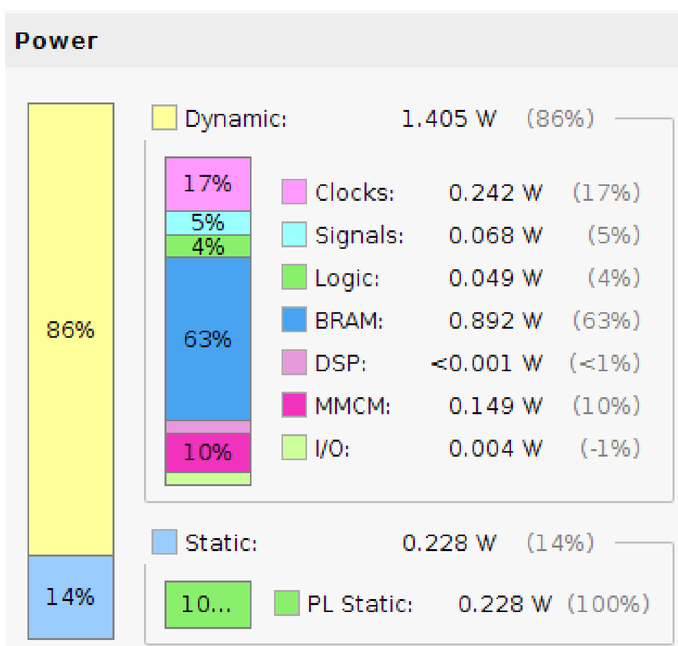


Figure 4.14. Power report of 10x10 array done with *Vivado* software.

Finally, the time report gives informations about time constraints. Fig.4.15

shows both clock frequencies adopted: the first one (125 MHz) is the proper clock used by the architecture; second one (200 MHz) is used, instead, by the AER-SRT controller.

Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
clk_in1_p	{0.000 2.500}	5.000	200.000
clk_out1_clk_wiz_0	{0.000 4.000}	8.000	125.000
clkfbout_clk_wiz_0	{0.000 2.500}	5.000	200.000
dbg_hub/inst/BSCANID.u_xsdbm_id/SWITC...	{0.000 16.500}	33.000	30.303

Figure 4.15. Clock summary provided by *Vivado* software.

Moreover, table 4.2 shows the slack values obtained during the synthesis. Slack is defined as the difference between actual or achieved time and the desired time for a timing path: it determines if the design is properly working at the specified frequency. Setup Slack is defined as:

$$SetupSlack = DataRequiredTime - DataArrivalTime \tag{4.1}$$

where *"Data Required Time"* is the time taken for the clock to traverse through clock path and *"Data Arrival Time"* is the time required for data to travel through data path.
A positive setup slack, as the one obtained, means design is working at the specified frequency and it has some more margin as well. Since speed is not a goal of this architecture, it is preferred to keep this margin in order to do not have possible problems in future implementations that could increase the critical path.

Type	**Worst Slack**	**Total Violation**	**Failing Endpoints**	**Total Endpoints**
Setup	0.190 ns	0.000 ns	0	162938
Hold	0.054 ns	0.000 ns	0	162938
Pulse Width	3.232 ns	0.000 ns	0	55359

Table 4.2. Report timing summary provided by *Vivado* software.

These time results, that confirm there are not any time violations, testify that pipeline stages are able to solve timing problems which the system was affected from. Actually, in Fig.4.16 there is shown the time report of the original architecture, without pipeline stages.

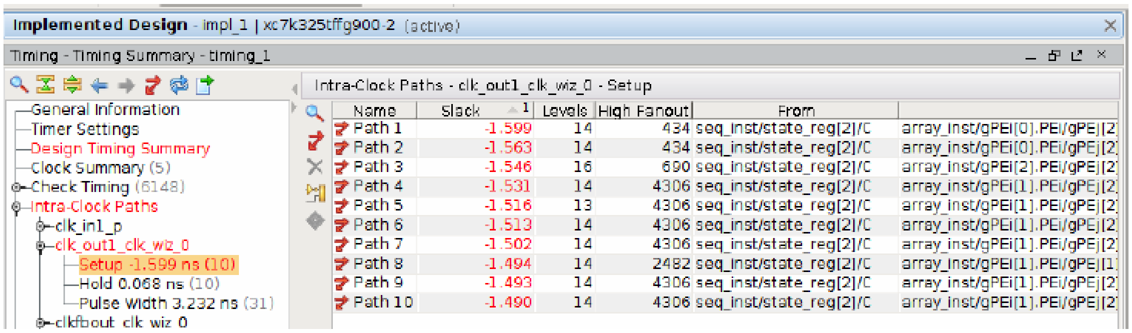


Figure 4.16. Timing report of the architecture implementing a 10x10 multiprocessor array without pipeline stages [4].

The architecture has a 10x10 PE array, as the one tested in this chapter, and the setup slack results, whose worst case is equal to -1.599 ns, demonstrate how increasing the size of the array the previous structure was not able to guarantee the right functioning of the architecture.

Finally, regarding the multiprocessor array, it is possible to find out which is the critical path, or rather the signal passing through the cell (the array) with the longest path. This signal is shown in Fig.4.17 and it is the *config* signal, which goes from *AER_OneBoard* to the *Config_Register* of 6_{th} PE and which has a setup slack equal 0.190 ns. To calculate which is the longest path, software has to take into account both combinational logic that signals could cross and the effective distance on the chip, that is given after the implementation. Therefore, in Fig.4.18 it is illustrated, as a white arrow, the path of this *config* signal along the device.

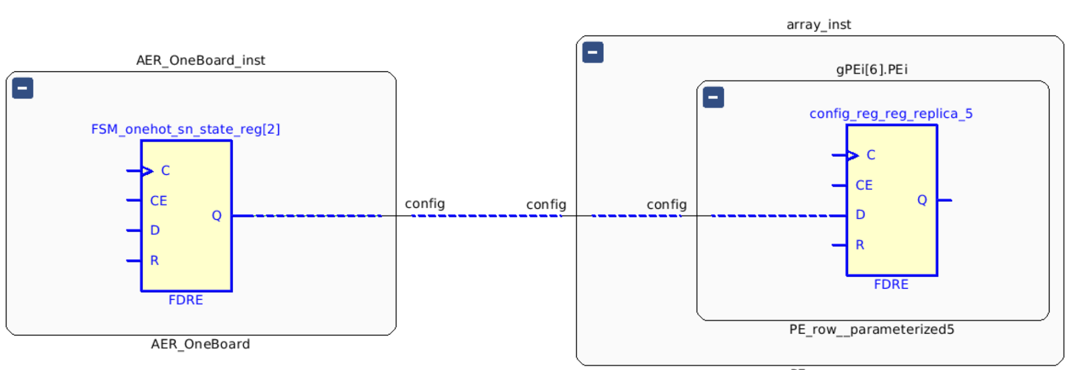


Figure 4.17. Schematic of the critical path provided by *Vivado* software.

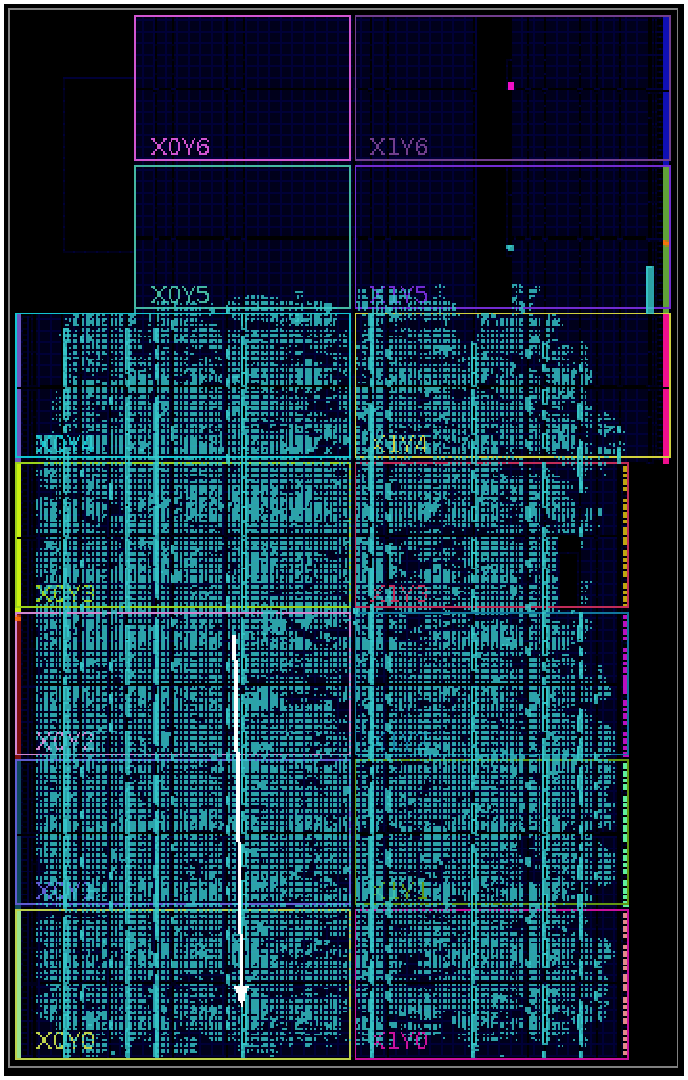


Figure 4.18. Device view of the critical path provided by *Vivado* software.

Chapter 5

Virtualization

In this chapter it is reported the implementation of one of this project goals, which is to support the virtualization of PE up to seven neurons in addition of the main layer: this means that, without introducing new PEs, it is possible to emulate them, increasing the number of neurons without using new hardware resources. Eight virtualization levels are supported, thus for instance, if we have a 10x10 array, this architecture will be able to emulate up to 800 neurons. Also, each PE supports up to 127 local synapses and 32 global synapses (coming form external multiprocessors).

5.1 IF Assembly Code supporting virtualization

Introducing virtualization support means that the assembly code has to consider also virtual PEs, therefore the main idea is to repeat and adapt all the steps described in the original assembly code (see Chapter 3.3), in order to initialize and process also virtual neurons. The resulting assembly code is reported in Appendix B.2 and it is illustrated in Fig.5.1.

When the algorithm starts, a virtual operation initialization is done in order to define the chosen number of virtual layers: with LAYERV instruction the *virtual_layers* value is taken, then with LDALL it is loaded into ACC and finally, using a special move instruction SPMOV, the value in ACC is loaded into th VIRT register of the sequencer. Moreover, this instruction resets the current virtual layer register to be processed to 0. Clearly, the current layer has a range of values going from 0 to *virtual_layers-1*.
Then, after pseudo-random seed initialization, all the steps done in the previous code (Appendix B.1) are mainly repeated as many times as the number of virtual layers, with some specific changes. Therefore, a neuron *LOOP* is called for virtual operations and it will be repeated *virtual_layers* times. During this loop, *LOAD_NEURON* and *MEMBRANE_DECAY* subroutines, whose contents are described in Chapter

3.3, are executed. Then, the address in the SNRAM is read, where there are all synaptic parameters for current virtual layer: to point the right memory location, the address is computed as the sum of the starting address plus the current virtual layer (READMPV SYN_ADDR0). In this way, all the addresses from SYN_ADDR0 to SYN_ADDR7 will be processed during the loop.
Finally, synaptic loop is executed as many times as the number of synapses assigned to the current virtual layer; inside this loop, *SYNAPSE_CALC* subroutine calculates the new membrane potential of the specific processed neuron.
After having checked if a spike is generated (*DETECT_SPIKE* subroutine) and having uploaded the SNRAM, storing the final membrane potential (STORESP) and increasing the pointer *BP*, with INCV instruction the current virtual layer to be processed is increased, in order to start again the neuron *LOOP* for the next virtual layer.

Once all virtual layers are processed, execution phase ends, spikes are distributed (SPKDIS) and the execution loop can start again (*GOTO EXEC_LOOP*).

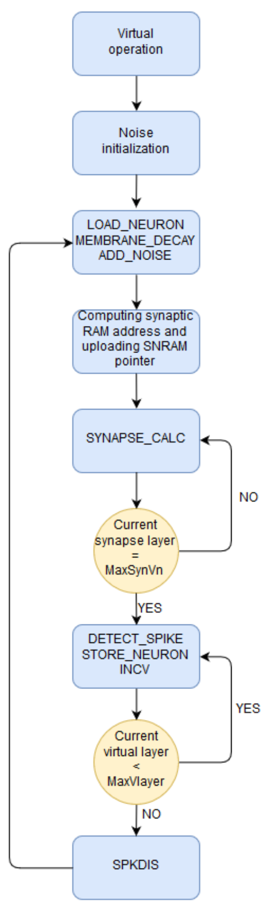


Figure 5.1. Flowchart of IF Assembly Code supporting virtualization.

5.1.1 Memory Interface

An important aspect to take into account is how to manage memory in order to store connectivities, synaptic and neural parameters. Since this work of thesis is focussed on the improvement ans simulation of the PE array, this description will not take into account global memories, since they describe connectivities between processing elements belonging to different boards, which is out of this work scope.

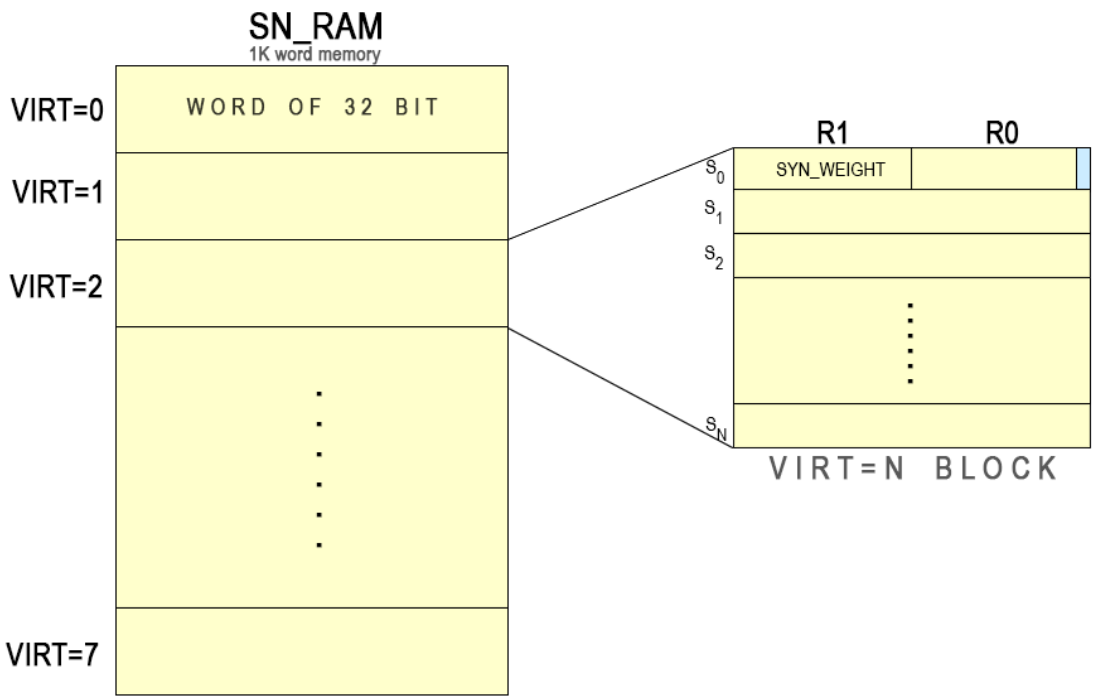


Figure 5.2. SNRAM mapping.

Each processing element has a SNRAM block, which is 1K word memory, where a word is composed by 32 bit, containing synaptic data. Fig.5.2 shows how the SNRAM is mapped.
The memory is divided in a number of blocks equal to *VIRT*, which is the number of virtual layers the system is using. Inside each block, there is a number of lines N equal to the synapses which link the neuron in the virtual layer *VIRT* to other neurons in the same board. Each line contains a 32 bit word that could be divided into two fields: the 16 MSBs are loaded in the R1 register and they contains the synaptic weight of that specific connection; the 16 LSBs are load into R0, whose less significant bit is set to 1 of there a firing spike along that connection. The remaining bits of R0 can be used, for instance, for extra features, such as adaptation

parameters if STDP learning is to be embedded.
Referring to the assembly code in Appendix B.2, that corresponds to a proof-of-concept 4x4 PE network, each neuron has 16 local synapses and it is proposed to assign two synapses for each virtual layer. For this reason, *SYN_ADDR0*, the first address of Synaptic parameters in SNRAM for *VIRT*=0, is equal to the position 0 of the SNRAM, *SYN_ADDR1*, the first address of Synaptic parameters in SNRAM for *VIRT*=1, is equal to "00000002" and so on. Moreover, at the bottom of the SNRAM, neural parameters are stored, referred to the main neuron and the seven ones virtualized. The mapping for these parameters starts from *NEU_ADDR0*="000003E3", the first address of Neural parameters for V=0, to *NEU_ADDR7*="000003EA", the first address of Neural parameters for V=7.

When an incoming spike (pre-synaptic spike) comes from a neuron represented by another virtual layer of the same PE or from another PE, the synaptic weight stored in the SNRAM is used to compute the new membrane potential. The local memory is used, instead, to describe the interconnects that neuron owns.

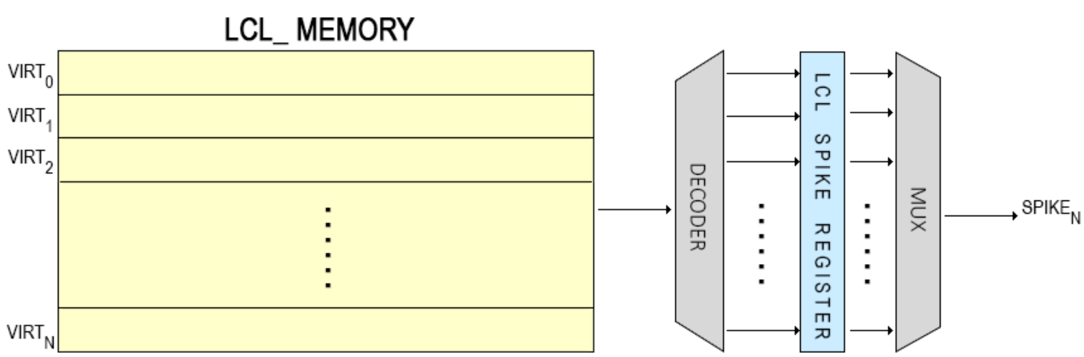


Figure 5.3. Local memory decoding.

Fig.5.3 shows how the local memory is decoded to record if there is an incoming spike. Actually, the local memory describe the interconnections that the PE has with all the other ones in the array: in Fig.5.4 it is shown in more detail how it is mapped.
Also in this case, it is divided in a number of blocks equal to the virtual layers. Each block contains a number of lines equal to 256, which is the maximum number of PEs that the array can support (16x16). Inside each line there is stored synapse number, so the specific synapse linking the neuron in the virtual layer *VIRT* to one of the 256 PEs.

When a pre-synaptic spike arrives, its address is used to take the synapses number from the specific line in the local memory, then this value is decoded and used to set to 1 the corresponding location in the local spike register. Therefore, this register owns as many 1s as the number of incoming spikes. Then, during execution phase, local spike register is read to check if there are pre-synaptic spikes: if $SPIKE_N$ is equal to 1, the corresponding synaptic weight is taken from the SNRAM to compute the new membrane potential.

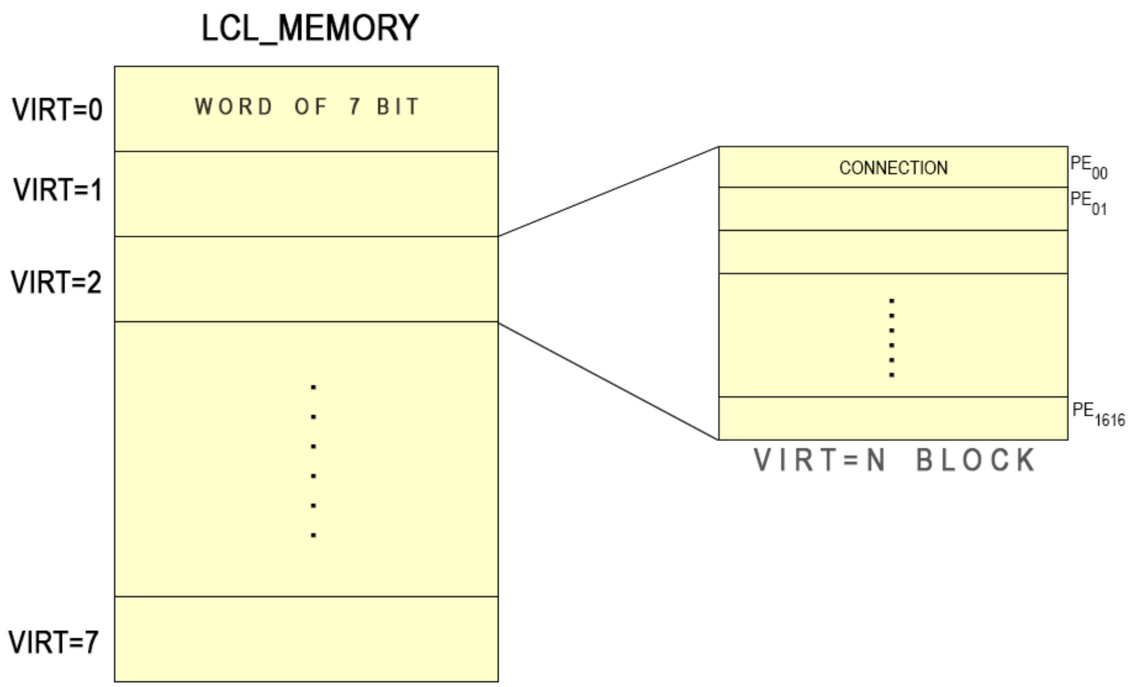


Figure 5.4. Local memory mapping.

5.2 Virtualization Design

In order to allow the architecture to support virtualization, the first change consists in modifying the netlist format to make interconnections between virtualized PEs. To perform it, it is necessary to have, in both source and destination address, a field that includes the virtual layer which the source/destination PE belongs to. The new netlist is shown in Tab.5.1:

Source				**Destination**					**Synaptic Data**
ChipID	vlayer	row	col	ChipID	vlayer	row	col	synapse_number	R1&R0

Table 5.1. New netlist format supporting virtualization.

where

- *vlayer* stands for the virtual virtual of the specific PE;
- *ChipID* is the board number in which there is the PE considered;
- *synapse_number* is the number of the one of 16 possible synapses that connects source PE with the destination one;
- *R0* and *R1* are the first two registers of the register bank. The value assigned on the synaptic data field is then split in order to load the 16 MSBs on R1 and the 16 LSBs on R0.

Subsequently, PE (Appendix D.1) and PE_array (Appendix D.3) architectures are modified in order to support virtualization.
The proposed architecture of the PE is shown in Fig.5.5. Now the 8-bit *spike_out* shift register can be used in all its depth to store informations about virtual layers spikes: in the previous architecture, in fact, only *spike_out[0]* was used, since virtualization was not supported.

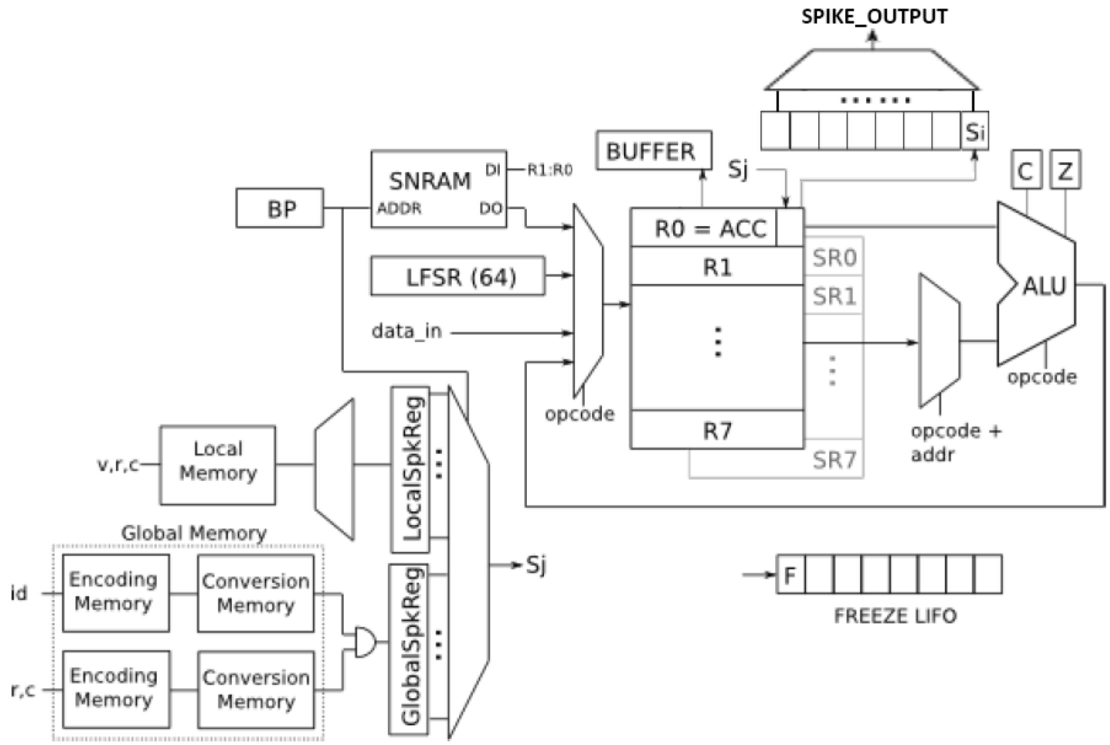


Figure 5.5. Block diagram of *HEENS* processing element supporting virtualization.

Fig.5.6 shows more in detail which are the changes made. As before, when the signal *load_sp* is asserted (when *STOREPS* instruction is executed), if the PE in the main level (VIRT=0) produces a spike, the *spike_out[0]* is set. Now, due to the system has virtualized PEs producing spikes, a left shift is done every time there is a *STOREPS* instruction and then *spike_out[0]* is set to one if a spike is produced. These shifts are repeated as many times as the virtual layer number, starting with layer 0 spike storage and proceeding, at most, to layer 7: in this way, at the end, all the possible spikes are stored along the shift register and the layer 0 spike is stored at the position of the register equal to the number of virtual layers.

Regarding the distribution phase, it is important to know the exact number of virtual layers the system is using. To compute it, a counter in the sequencer is introduced, counting the effective number and transmitting it to PEs with the *vlayers* input signal. So, when a spike has to be distributed, as before *en_spike_tx* is asserted and *spike_out[vlayers]* will be transmitted (before *spike_out[0]*).
Actually, due to the spike_out left shifting, if there is a number of virtual layers equal to *vlayers*, the spike of the main layer is stored in position *vlayers*, the one of 1st layer in position *vlayers-1* and so on. For this reason, *vlayers* is used as index to select the correct spike in the shift register. Moreover, a new signal called *next_virt*

is introduced: it comes from the PE_array and it is asserted each time all the spikes in the j^{th} virtual layer are distributed, so the system is ready to check and, in case, distribute the spikes in the $(j+1)^{th}$ virtual layer. So, when *next_virt* is equal to one, a left-shift is done and *spike_out[0]* is reset. Proceeding in this way brings two benefits:

- left shifting every time a spike is distributed permits to have, each time, the correct next spike to transmit in position *spike_out[vlayers]*; in this way, the index is fixed and no down-counting has to be implemented;
- at the end of distribution phase, the register is reset, so it is ready to store new possible spikes for the next execution phase.

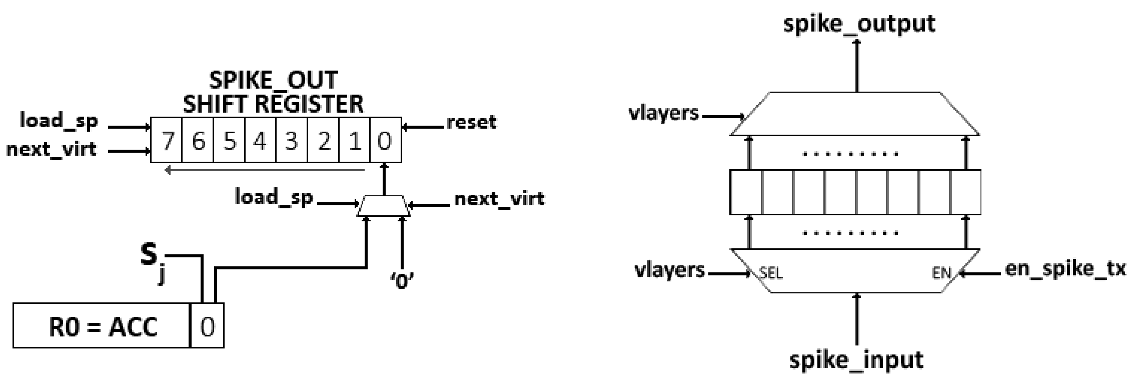


Figure 5.6. Focus on PE blocks that manage distribution phase and support virtualization.

All the others features introduced in Chapter 4 remain valid, becoming compatible with virtualization. The distribution phase of the j^{th} virtual layer uses the same propagation through pipeline stages as before: the whole array shifts spikes of that layer using *spike_input* and *spike_output* signals as before; then, when all the spikes of j^{th} virtual layer spikes are transmitted, the distribution starts again its routine for the $j+1^{th}$ layer.

Finally, in addition to the row and column addresses (*row_sp* and *col_sp*), the address of the PE distributing a spike has to be completed with *virt_sp*, which is the field of the PE address informing about its virtual layer. Therefore, in PE_array it is introduced a virtual layer counter that is incremented each time the spike_out register shifts, so when *shift_sp* is asserted.

5.2.1 Simulations

In order to verify that the design behaves as expected and check if the new netlist format describes in the right way paths including virtual layers, a "band pass filter" path (Appendix C.4) showed in Fig.5.7 is made up to be simulated. The software used to simulate is *QuestaSim* and the assembly code adopted is the one in Appendix B.2.

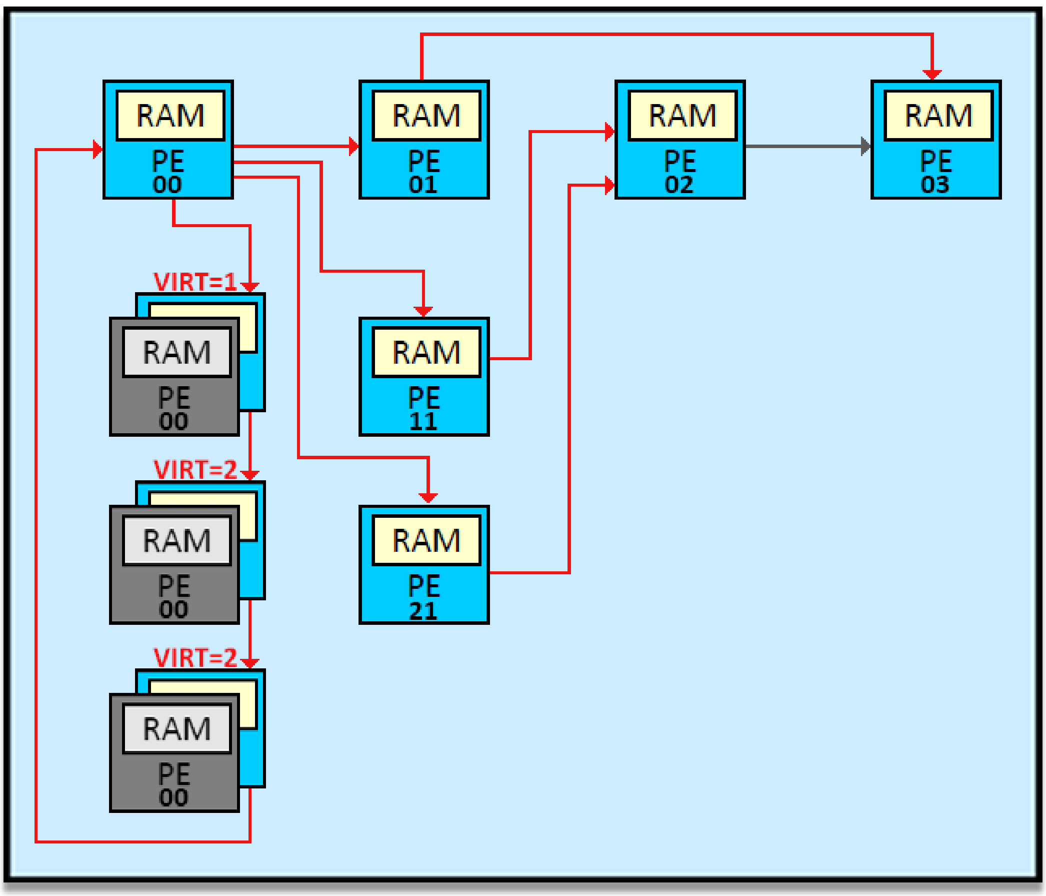


Figure 5.7. Block diagram of 4x4 "band pass filter path" network, including a "ring oscillator" path between virtual layers of PE[0;0]. Red and grey arrows indicate respectively synapses with a positive or negative weights.

Therefore, a 4x4 multiprocessor array is simulated, whose processing element in position [0;0] has all the virtual layer linked between them self, creating a ring oscillator path. This kind of path is used to easily check if the virtualization works. To better display all the PEs with their respective virtual layers, *spike_displ*, already adopted in previous simulations, is modified in order to consider virtual layers: now it is a 3D array containing all the spikes calculated during the execution phase.

In Fig.5.8 it is shown the simulation results of the proposed ring oscillator. All the four neurons simulated, belonging to the processing element in position [0;0], receive a spike coming from the previous virtual layer and they fire as well, producing a post-synaptic spike direct to the next layer.

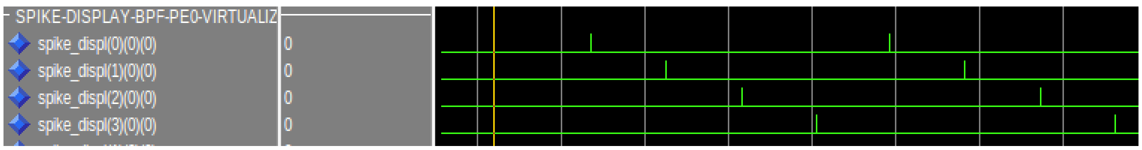


Figure 5.8. Simulation, done with *Questasim* software, of a ring oscillator between virtual layers of PE[0;0].

In particular, the main signals involved to make the architecture compatible for the virtualization are shown in Fig.5.9. During the distribution phase, when *en_si* is asserted, *next_virt* is cycled as many times as $vlayers\_count + 1$ (8) and each time it is asserted, *next_PE_row* is equal to 1 four times, as the number of rows used.

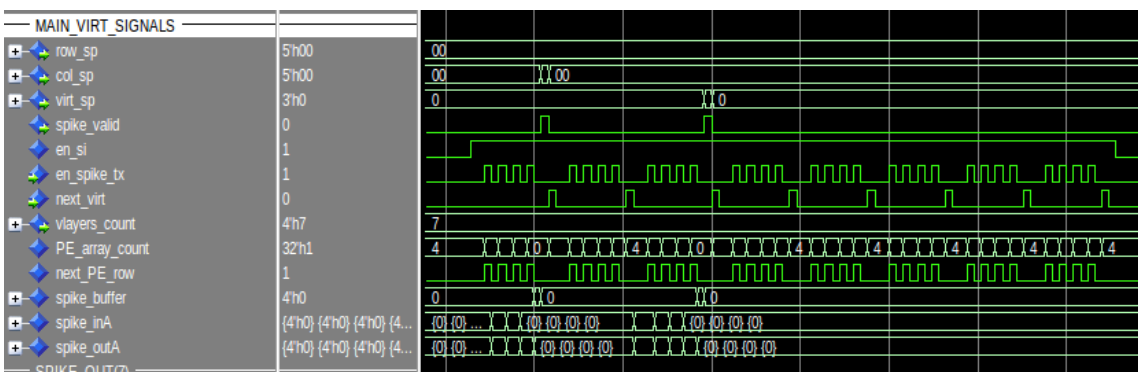


Figure 5.9. Simulation, done with *Questasim* software, of all control signals involved into virtualization.

Enlarging the simulation, as shown in Fig.5.10, it is possible to check that the new address format works. Actually, both PE[0;1] and PE[0;0] have post-synaptic spike: the first one produces a spike from the virtual layer 0; the second one from the virtual layer 1. Each time a spike has to be distributed, *spike_valid* is asserted and *row_sp*, *col_sp* and *virt_sp* are upload.

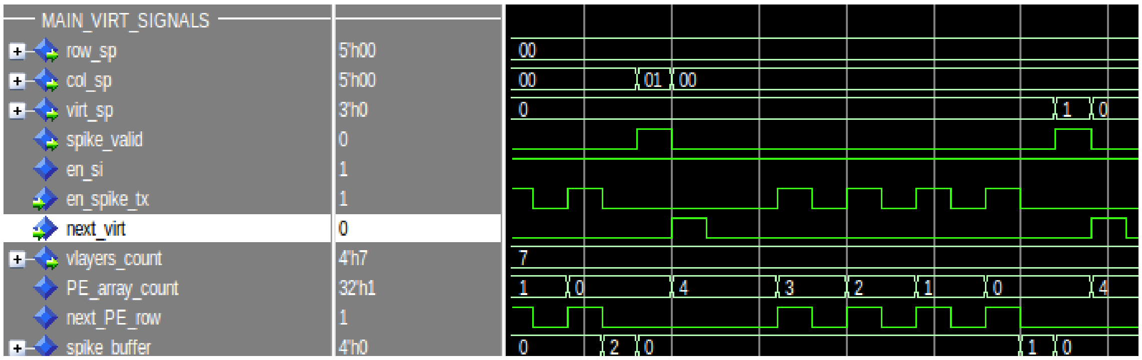


Figure 5.10. Simulation, done with *Questasim* software, of the distribution of two spikes coming from different virtual layers.

From simulation results, the introduction of virtualization does not affect the functioning of the other changes made previously, as pipeline stages and the new distribution structure. Since this simulation include both virtual and non-virtual synaptic connections and it shows clearly that interactions between neurons in different virtual layers works as expected, it is possible to confirm that the architecture is supporting virtualization, so it has to be synthesized and implemented on an FPGA to check if also time constrains remain inviolate and the area occupancy is not significantly increased.

5.3 Synthesis and Implementation

After simulating the new architecture supporting virtualization, the system has to be tested on a board, using *Vivado* software, to check how much the new implementations have impacted on area occupancy and operating time. In order to make significant comparisons with the previous architecture, the system is again implemented on Xilinx Zynq-7000 SoC ZC706 board. Since in chapter 4.3 it is estimated a maximum array size equal to 12x12, the first test is done with this dimension. Firstly, during the synthesis, *Vivado* computes the utilization of the FPGA and its resources and all the constrains as a function of the board used. Then, during implementation, the software generates all the components to be placed on the FPGA, with all the connections, taking into account the distance between registers and calculating the worst delay to determine if there are time problems.

In Fig.5.11 is it shown the obtained area utilization results: as expected, BRAM occupancy is 20% bigger than the one using a 10x10 multiprocessor array (see Fig.4.13). The relevant aspect is that this increase comes from the larger array used and it is not caused by the virtualization support: actually, the amount of hardware introduced is independent from the array size and it consists in few control signals and small blocks, which do not impact a lot on the board occupancy.

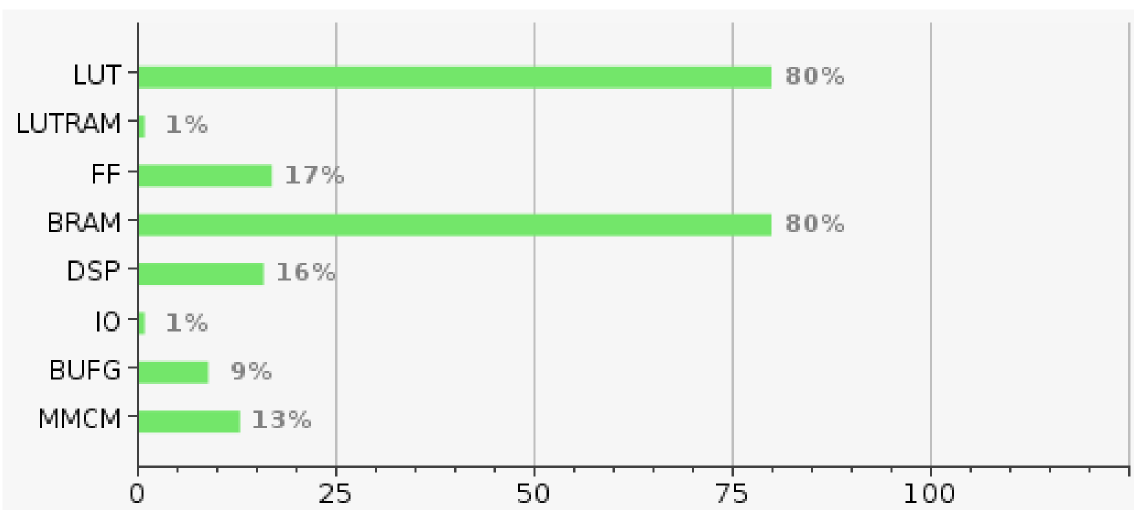


Figure 5.11. Area utilization of the architecture supporting virtualization with 12x12 multiprocessor array done with *Vivado* software.

These results confirm that it is still possible to extend the array more, although it has to be checked if it realizable without having timing violations. In Fig.5.12 it is displayed the floorplanning of the architecture implemented: the black areas clearly show the unoccupied space on the chip that hypothetically permit to extend the array.

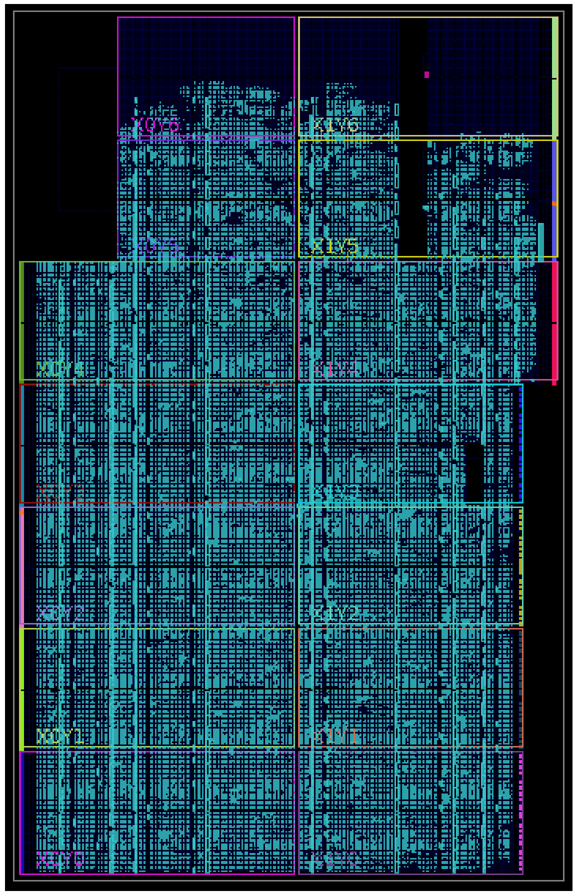


Figure 5.12. Floorplanning of a 12x12 array supporting virtualization done with *Vivado* software.

Moreover, the report timing shown in Table 5.2 confirms that the architecture do not violate time constraints because all slacks are positive, meaning that the 12x12 multiprocessor array works with the correct behaviour.

Type	**Worst Slack**	**Total Violation**	**Failing Endpoints**	**Total Endpoints**
Setup	0.017 *ns*	0.000 *ns*	0	231080
Hold	0.024 *ns*	0.000 *ns*	0	231080
Pulse Width	3.232 *ns*	0.000 *ns*	0	77032

Table 5.2. Report timing summary of a 12x12 array supporting virtualization provided by *Vivado* software.

With the purpose to extend the array as much as possible, the architecture is tested again on the same board with a 13x13 PE array.
In Fig.5.13 it is reported the area utilization provided by *Vivado* software: this time, LUT and BRAM exceed the 90% of the available space. These results confirm that, if there are not time violations, this is the maximum achievable size which can be implemented on this board.

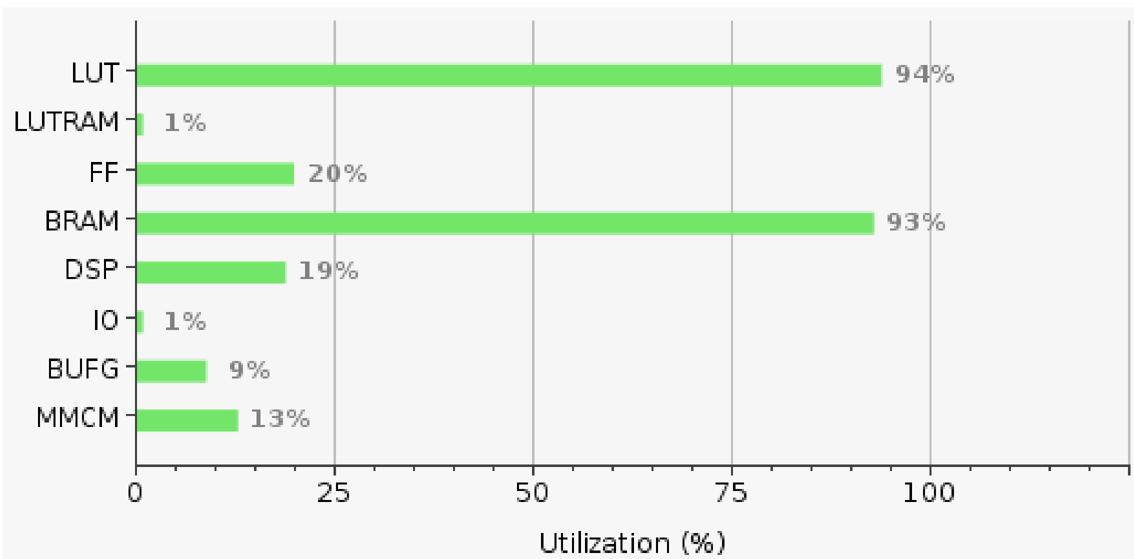


Figure 5.13. Area utilization of the architecture supporting virtualization with 13x13 multiprocessor array done with *Vivado* software.

Furthermore, the floorplanning in Fig.5.14 shows the occupancy of the FPGA, in blue color. Only few small strips near corners and a little area in the centre are free, which are dedicated to another unused functionality.

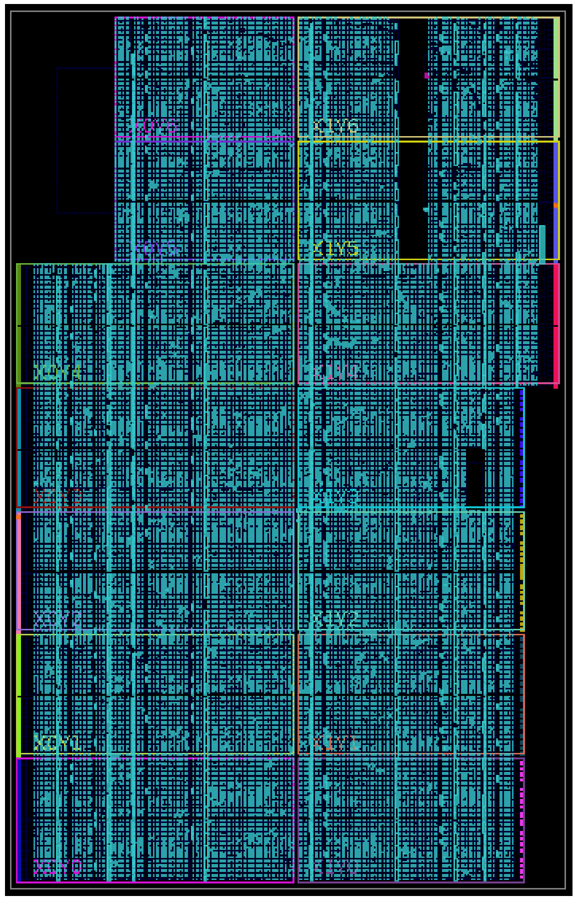


Figure 5.14. Floorplanning of a 13x13 array supporting virtualization done with *Vivado* software.

Finally, timing report in Table 5.3 obtained during the synthesis confirms that the pipeline stages introduced are able to ensure the proper functioning of the architecture, since all slacks are positive. This means that, at the end, it was possible to achieve a number of PE equal to 1352, thanks to the implementation of virtual layers.

Type	**Worst Slack**	**Total Violation**	**Failing Endpoints**	**Total Endpoints**
Setup	0.091 *ns*	0.000 *ns*	0	271182
Hold	0.034 *ns*	0.000 *ns*	0	271182
Pulse Width	3.232 *ns*	0.000 *ns*	0	90411

Table 5.3. Report timing summary of a 13x13 array supporting virtualization provided by *Vivado* software.

Chapter 6

Conclusions and future development

HEENS is an architecture designed for multi-FPGA implementations for SNN emulation in real time and it represents a proposing system, among the ones that use a neural network belonging to the 3^{rd} generation, able to guarantee high level of configurability, scalability and multi-model.

This thesis work has the main goal to enhance system scalability, operation speed and resource occupancy, improving some weaknesses of the system and to introduce virtualization, a feature that permits to increase the number of emulated neurons without impacting the area utilization on the FPGA. Therefore, the architecture is synthesized and implemented on the Xilinx Zynq-7000 SoC ZC706 board, in order to test it and to verify if the design works in the proper way.

In particular, pipeline stages are introduced in order to reduced the critical path: having a pipelined array of PEs, in fact, permits to extend it keeping fixed the operating frequency. Tests on board demonstrate how the system is now capable to support arrays up to 13x13 without timing violations, thus taking a step forward compared to before.

Another bottleneck of the architecture was the area occupancy on the FPGA: extending the array means to introduce new processing elements, each one designed using a big amount of hardware; therefore the array extension was conditioned also by these area problems. For this reason, this work proposes, as solution, the introduction of virtualized PEs, which are a replica of the original ones but they do not require more hardware resources, since they are the same PE with new loaded synaptic parameters. At the end, it was possible to achieve seven virtualized levels, plus the main one, and to extend the array up to 13x13, permitting the simulation of 1352 neurons.

The natural continuation of this thesis work, which is focussed on the hardware enhancement of the architecture, is the development of a software environment for virtual layer memory mapping. Actually, during this work, the memory interface has been mapped manually, assigning addresses of the neural and synaptic parameters of the SNRAM though the assembly code. Moreover, also the generation of interconnections could be managed via software: abandoning the netlist format so far used to make room for a software management that automatically maps the local and global memories, without going through a netlist text file.

Appendix A

Instruction Set

The instruction set, also called ISA (instruction set architecture), provides commands to the processor, to tell what it needs to do. An opcode (operation code) is associated with each instruction of ISA, to specify the operation to be performed. There are different categories of opcode, depending on the purpose of the instruction: arithmetic operations, movements, conditional freeze, flags and others. Instead of binary code, opcode is identified by a name, which coupling to the function which it represents, in order to have an higher lever of reconfigurability, since if the binary value changes, these is nothing to change in the VHDL code. Opcode name 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 corresponding binary number, composed by 6 bits, and it is sent with other data through the communication bus.

	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)
48	RST_SEQ	110000	Jumps to RESET state
49	-	-	-
50	LAYERV	110010	VLAYERS <= n; CURR_VLAYER <= 0; defines number of virtual layers (currently 0 <= n <= 7)
51	GOTO	110011	PC <= addr
52	SHLAN	110100	ACC <= ACC << n, (1 <= n <= 8), Arithmetic shift
53	SHRAN	110101	ACC <= ACC >> n, (1 <= n <= 8), Arithmetic shift
54	LOADBP	110110	BP <= DMEM Loads PE BRAM pointer.
55	BITSET	110111	ACC(n) <= 1
56	BITCLR	111000	ACC(n) <= 0
57	SPMOV	111001	Special MOVE. n = 0: VIRT <= ACC;
58	INCV	111010	VLAYER <= VLAYER + 1
59	READMPV	111011	DMEM <= BRAM(address + VLAYER)
60	MOVSR	111100	shadow_reg <= reg

Appendix B

Assembler Code

B.1 IF

```
; Integrate and fire.
; DEFAULT operation without virtual layers

; Network definitions

define virtual_layers 0 ; Up to 7
define gsynapses 0 ; Up to 32 global synapses
define lsynapses  15 ;99  Due to the local RAM encoding, synapse 0 cannot be used (corresponds to no synapse code)
define tot_synapses 15 ;131

.DATA

; Membrane potential parameters common to all neurons
VREST="FFFFE4A8" ; Resting potential -70 mV = -7000 in tens of of uV
VTHRES="FFFFEA84"   ; Threshold voltage -55 mV = -5500
VDEPOL="FFFFE0C0"   ; Depolarization voltage -80 mV = -8000
VACT = "00001771"   ; Action potential +10 mV = +1000
;
;
; Neural and Synaptic RAM addresses
SEEDH_ADDR = "00000200" ; Address of noise seed in NSBRAM
SEEDL_ADDR = "00000201" ;
NEU_ADDR="00000100"   ; First address of Neural parameters in NSBRAM
SYN_ADDR="00000000"   ; First address of Synaptic parameters in NSBRAM.
;
; General constants
THAU_MEM="0000799A"   ; Membrane time constant decay (inverse value). To be tuned. Thau = 20
NOISE_MSK="0000001F"  ; Noise mask. To be tuned

; Constants for debug
JUMP_MV = "00000100"     ; Jump 2.56 mV on spike
LFSR_VAL= "0000AAAA"
LFSR_VAL2= "00005555"

INIT_VAL ="FFFFE890"   ; Vmem initiated at -60 mV, 10 mV above rest potential

.CODE
;
GOTO MAIN    ; Jump to main program
;
; **************************** PROCEDURES BEGIN ****************************
;
.RANDOM_INIT      ; Uses R0 and R1
    LOADBP SEEDH_ADDR
    LOADSN
    SEED                    ; High seed
    LOADBP SEEDL_ADDR
    LOADSN
    SEED                    ; Low seed
RET
;
.LOAD_NEURON      ; Uses R0, R1, R2 and R3
    READMPV NEU_ADDR ; Address of real neuron + virt (valid also for non-virtual)
    LOADBP ; NSBRAM pointer to currently processed neuron
    LOADSN        ; Load Neural parameters from NSBRAM to R1 & ACC
    MOVR R2       ; Move Vmem from ACC to R2
RET
;
.MEMBRANE_DECAY  ; Uses R0, R4
    MOVA R2
    LDALL R4 VREST
    SUB R4
    LDALL R1, THAU_MEM
    MULS R1       ; Calculate decay
    SHLAN 1       ; Shift one bit left because we multiply by n-1 bits (positive value in 2's complement)
    ADD R4
    MOVR R2       ; Back to R2 where membrane potential is stored
RET
;
.ADD_NOISE ; Uses R0, R2 and R5
    RANDON                    ; LFSR ON
    LLFSR ; Noise to ACC
    MOVR R5
    LDALL ACC, NOISE_MSK
    AND R5
    SHRN 1
    RANDOFF                    ; LFSR OFF. Arbitrarily here
    FREEZENC
        MOVR R5
        RST ACC
        SUB R5       ; Generate signed noise without the negative bias of two's complement
    UNFREEZE
```

```
     MOVSR ACC                                             ;    TO MONITOR THE NOISE
     ADD R2   ; Add to Vmem
     MOVR R2 ; Back to R2
RET
;
.DETECT_SPIKE    ; Uses R0 and R2
     LDALL ACC, VTHRES
     SUB R2       ; Compare Vth - Vmem
     SHLN 1       ;subtraction sign to C flag
     RST ACC
     FREEZENC ; If positive, spike
         SET ACC
         LDALL R2 VREST ; Vmem to resting potential
     UNFREEZE
     STOREPS      ; Push spikes
RET
;
.STORE_NEURON ; uses R0 and R1
     MOVA R2      ; Move Vmem from R2 to ACC
     READMPV NEU_ADDR ; Address of real neuron + virt (valid also for non-virtual)
     LOADBP ; NSBRAM pointer to currently processed neuron
     STORESP  ; Store Vmem to NSBRAM
RET
; **************************** MAIN PROGRAMME BEGIN ************************
.MAIN

; Initial instructions
GOSUB RANDOM_INIT    ; For noise initialization

.EXEC_LOOP ; Execution loop

;%VIRT LOOP VLAYERS ; Neuron loop for virtual operation
GOSUB LOAD_NEURON
GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
GOSUB ADD_NOISE
LOADBP SYN_ADDR ; Initial position for addresses
    LOOP tot_synapses  ; synaptic loop
        LOADSP        ; Load Synaptic parameters and spike to R1 & ACC
        SHRN 1        ; Move spike to flag C
        FREEZENC
             MOVA R1 ; Synaptic parameter to ACC
            ADD R2
            MOVR R2   ; Save Neural parameter in R2
        UNFREEZE
        RST ACC
        STORESP       ; Stores synaptic parameter and increases BP for next synapse
processing
    ENDL
; Compare and eventually spike
GOSUB DETECT_SPIKE
GOSUB STORE_NEURON
NOP  ; Empty pipeline wait NOPs
NOP
NOP
SPKDIS ; Distribute spikes
GOTO EXEC_LOOP ; Execution loop
```

B.2 IF_VIRT

```
; GOTO CODE
;
; Integrate and fire. Both non-virtual and virtual

; Network definitions
define virtual_layers 7 ; from 0 up to 7 (1 to 8 layers)
define gsynapses 2 ; Up to 32 global synapses
define lsynapses  15 ;99  Due to the local RAM encoding, synapse 0 cannot be used (corresponds to no synapse code)
define tot_synapses 15 ;131

.DATA

; Virtual layers

V0 = "00000001" ; Number of assigned synapses (s-1) to the main layer
V1 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 1
V2 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 2
V3 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 3
V4 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 4
V5 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 5
V6 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 6
V7 = "00000001" ; Number of assigned synapses (s-1) to virtual layer 7
VLAYERS="00000007" ; Number of virtual layers(n-1).

; Membrane potential parameters common to all neurons
VREST="FFFFE4A8" ; Resting potential -70 mV = -7000 in tens of of uV
VTHRES="FFFFEA84"   ; Threshold voltage -55 mV = -5500
VDEPOL="FFFFE0C0"   ; Depolarization voltage -80 mV = -8000
VACT = "00001771"   ; Action potential +10 mV = +1000
;
;
; Neural and Synaptic RAM addresses
SYN_ADDR0="00000000"   ; First address of Synaptic parameters in SNRAM for V = 0.
SYN_ADDR1="00000002"   ; First address of Synaptic parameters in SNRAM for V = 1.
SYN_ADDR2="00000004"   ; First address of Synaptic parameters in SNRAM for V = 2.
SYN_ADDR3="00000006"   ; First address of Synaptic parameters in SNRAM for V = 3.
SYN_ADDR4="00000008"   ; First address of Synaptic parameters in SNRAM for V = 4.
SYN_ADDR5="0000000A"   ; First address of Synaptic parameters in SNRAM for V = 5.
SYN_ADDR6="0000000C"   ; First address of Synaptic parameters in SNRAM for V = 6.
SYN_ADDR7="0000000E"   ; First address of Synaptic parameters in SNRAM for V = 7.
GSYN_ADDR="00000064"   ; First address of Global Synaptic parameters in SNRAM.
NEU_ADDR0="000003E3"   ; First address of Neural parameters in SNRAM (995) for V = 0.
NEU_ADDR1="000003E4"   ; First address of Neural parameters in SNRAM (996) for V = 1.
NEU_ADDR2="000003E5"   ; First address of Neural parameters in SNRAM (997) for V = 2.
NEU_ADDR3="000003E6"   ; First address of Neural parameters in SNRAM (998) for V = 3.
NEU_ADDR4="000003E7"   ; First address of Neural parameters in SNRAM (999) for V = 4.
NEU_ADDR5="000003E8"   ; First address of Neural parameters in SNRAM (1000) for V = 5.
NEU_ADDR6="000003E9"   ; First address of Neural parameters in SNRAM (1001) for V = 6.
NEU_ADDR7="000003EA"   ; First address of Neural parameters in SNRAM (1002) for V = 7.

SEEDH_ADDR = "000003FD" ; Address of noise seed in SNRAM
SEEDL_ADDR = "000003FE" ;
PEID  = "000003FF" ; Address of  PE Identifier number

; General constants
;THAU_MEM="00007F00"   ; Membrane time constant decay (inverse value). To be tuned
THAU_MEM="0000799A"   ; Membrane time constant decay (inverse value). To be tuned. Thau = 20
NOISE_MSK="0000001F"  ; Noise mask. To be tuned

; Constants for debug
JUMP_MV = "00000100"     ; Jump 2.56 mV on spike
LFSR_VAL= "0000AAAA"
LFSR_VAL2= "00005555"


.CODE
;
GOTO MAIN   ; Jump to main program
;
; *************************** PROCEDURES BEGIN ***************************
;
.RANDOM_INIT     ; Uses R0 and R1
    LOADBP SEEDH_ADDR
    LOADSN
    SEED                   ; High seed
    LOADBP SEEDL_ADDR
    LOADSN
    SEED                   ; Low seed
RET
;
.LOAD_NEURON     ; Uses R0, R1, R2 and R3
    READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
    LOADBP ; SNRAM pointer to currently processed neuron
```

```
    LOADSN       ; Load Neural parameters from SNRAM to R1 & ACC
    MOVR R2      ; Move Vmem from ACC to R2
    MARK
RET
;
.MEMBRANE_DECAY  ; Uses R0, R4
    MOVA R2                                  ;     TEMPORARY WHILE MULS has problems. REWRITE when it works
    LDALL R4 VREST
    SUB R4
    LDALL R1, THAU_MEM
    MULS R1      ; Calculate decay
    SHLAN 1
    ADD R4
    MOVR R2      ; Back to R2 where membrane potential is stored
RET
;
.ADD_NOISE ; Uses R0, R2 and R5
    RANDON                 ; LFSR ON
    LLFSR ; Noise to ACC
    MOVR R5
    LDALL ACC, NOISE_MSK
    AND R5
    SHRN 1
    RANDOFF                 ; LFSR OFF. Arbitrarily here
    FREEZENC
        MOVR R5
        RST ACC
        SUB R5       ; Generate signed noise without the negative bias of two's complement
    UNFREEZE
    MOVSR ACC                                          ;    TO MONITOR THE NOISE
    ADD R2   ; Add to Vmem
    MOVR R2 ; Back to R2
RET
;
.SYNAPSE_CALC
    LOADSP       ; Load Synaptic parameters and spike to R1 & ACC
    SHRN 1       ; Move spike to flag C
    FREEZENC
        MOVA R1 ; Synaptic parameter to ACC
        ADD R2
        MOVR R2  ; Save Neural parameter in R2
    UNFREEZE
    RST ACC
    STORESP      ; Stores synaptic parameter and increases BP for next synapse processing
RET
;
.DETECT_SPIKE    ; Uses R0 and R2
    LDALL ACC, VTHRES
    SUB R2       ; Compare Vth - Vmem
    SHLN 1       ;subtraction sign to C flag
    RST ACC
    FREEZENC ; If positive, spike
        SET ACC
        LDALL R2 VREST ; Vmem to resting potential
    UNFREEZE
    STOREPS      ; Push spikes
RET
;
.STORE_NEURON ; uses R0 and R1
    MOVA R2      ; Move Vmem from R2 to ACC
    READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
    LOADBP ; SNRAM pointer to currently processed neuron
    STORESP  ; Store Vmem to SNRAM
RET
;
; **************************** PROCEDURES END ******************************

; ************************* MAIN PROGRAMME BEGIN ***************************
.MAIN

; Virtual operation init
LAYERV virtual_layers ; Init sequencer vlayers. It is 0 for non-virtual operation
LDALL ACC, VLAYERS ; Load defined virtual layers to PE array
SPMOV 0  ; VIRT <= ACC

; Initial instructions
GOSUB RANDOM_INIT    ; For noise initialization

.EXEC_LOOP ; Execution loop

LOOP virtual_layers ; Neuron loop for virtual operation
    NOP   ;to prevent pipeline error
    GOSUB LOAD_NEURON
```

```
    GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
    GOSUB ADD_NOISE
    READMPV SYN_ADDR0
    LOADBP
        LOOPV V0  ; synaptic loop. Reads number of current-layer synapses
            NOP   ;to prevent pipeline error
            GOSUB SYNAPSE_CALC
        ENDL
    ; Compare and eventually spike
    GOSUB DETECT_SPIKE
    GOSUB STORE_NEURON
    INCV
ENDL
.FINISH
NOP  ; Empty pipeline wait NOPs
NOP
NOP
SPKDIS ; Distribute spikes
GOTO EXEC_LOOP ; Execution loop
```

Appendix C

Netlist

C.1 netlist_ringosc5x5.txt

0 0 0 0 1 1 131072000
0 0 1 0 2 1 131072000
0 0 2 0 3 1 131072000
0 0 3 0 4 1 131072000
0 0 4 1 4 1 131072000
0 1 4 2 4 1 131072000
0 2 4 3 4 1 131072000
0 3 4 4 4 1 131072000
0 4 4 4 3 1 131072000
0 4 3 4 2 1 131072000
0 4 2 4 1 1 131072000
0 4 1 4 0 1 131072000
0 4 0 3 0 1 131072000
0 3 0 2 0 1 131072000
0 2 0 1 0 1 131072000
0 1 0 0 0 1 131072000

C.2 netlist_ringosc10x10.txt

0 0 0 0 1 1 131072000
0 0 1 0 2 1 131072000
0 0 2 0 3 1 131072000
0 0 3 0 4 1 131072000
0 0 4 0 5 1 131072000

0 0 5 0 6 1 131072000
0 0 6 0 7 1 131072000
0 0 7 0 8 1 131072000
0 0 8 0 9 1 131072000
0 0 9 1 9 1 131072000
0 1 9 2 9 1 131072000
0 2 9 3 9 1 131072000
0 3 9 4 9 1 131072000
0 4 9 5 9 1 131072000
0 5 9 6 9 1 131072000
0 6 9 7 9 1 131072000
0 7 9 8 9 1 131072000
0 8 9 9 9 1 131072000
0 9 9 9 8 1 131072000
0 9 8 9 7 1 131072000
0 9 7 9 6 1 131072000
0 9 6 9 5 1 131072000
0 9 5 9 4 1 131072000
0 9 4 9 3 1 131072000
0 9 3 9 2 1 131072000
0 9 2 9 1 1 131072000
0 9 1 9 0 1 131072000
0 9 0 8 0 1 131072000
0 8 0 7 0 1 131072000
0 7 0 6 0 1 131072000
0 6 0 5 0 1 131072000
0 5 0 4 0 1 131072000
0 4 0 3 0 1 131072000
0 3 0 2 0 1 131072000
0 2 0 1 0 1 131072000
0 1 0 0 0 1 131072000

C.3 netlist_snake10x10.txt

0 0 0 0 1 1 131072000
0 0 1 0 2 1 131072000
0 0 2 0 3 1 131072000
0 0 3 0 4 1 131072000
0 0 4 0 5 1 131072000

0 0 5 0 6 1 131072000
0 0 6 0 7 1 131072000
0 0 7 0 8 1 131072000
0 0 8 0 9 1 131072000
0 0 9 1 9 1 131072000
0 1 9 1 8 1 131072000
0 1 8 1 7 1 131072000
0 1 7 1 6 1 131072000
0 1 6 1 5 1 131072000
0 1 5 1 4 1 131072000
0 1 4 1 3 1 131072000
0 1 3 1 2 1 131072000
0 1 2 1 1 1 131072000
0 1 1 1 0 1 131072000
0 1 0 2 0 1 131072000
0 2 0 2 1 1 131072000
0 2 1 2 2 1 131072000
0 2 2 2 3 1 131072000
0 2 3 2 4 1 131072000
0 2 4 2 5 1 131072000
0 2 5 2 6 1 131072000
0 2 6 2 7 1 131072000
0 2 7 2 8 1 131072000
0 2 8 2 9 1 131072000
0 2 9 3 9 1 131072000
0 3 9 3 8 1 131072000
0 3 8 3 7 1 131072000
0 3 7 3 6 1 131072000
0 3 6 3 5 1 131072000
0 3 5 3 4 1 131072000
0 3 4 3 3 1 131072000
0 3 3 3 2 1 131072000
0 3 2 3 1 1 131072000
0 3 1 3 0 1 131072000
0 3 0 4 0 1 131072000
0 4 0 4 1 1 131072000
0 4 1 4 2 1 131072000
0 4 2 4 3 1 131072000
0 4 3 4 4 1 131072000
0 4 4 4 5 1 131072000
0 4 5 4 6 1 131072000

0 4 6 4 7 1 131072000
0 4 7 4 8 1 131072000
0 4 8 4 9 1 131072000
0 4 9 5 9 1 131072000
0 5 9 5 8 1 131072000
0 5 8 5 7 1 131072000
0 5 7 5 6 1 131072000
0 5 6 5 5 1 131072000
0 5 5 5 4 1 131072000
0 5 4 5 3 1 131072000
0 5 3 5 2 1 131072000
0 5 2 5 1 1 131072000
0 5 1 5 0 1 131072000
0 5 0 6 0 1 131072000
0 6 0 6 1 1 131072000
0 6 1 6 2 1 131072000
0 6 2 6 3 1 131072000
0 6 3 6 4 1 131072000
0 6 4 6 5 1 131072000
0 6 5 6 6 1 131072000
0 6 6 6 7 1 131072000
0 6 7 6 8 1 131072000
0 6 8 6 9 1 131072000
0 6 9 7 9 1 131072000
0 7 9 7 8 1 131072000
0 7 8 7 7 1 131072000
0 7 7 7 6 1 131072000
0 7 6 7 5 1 131072000
0 7 5 7 4 1 131072000
0 7 4 7 3 1 131072000
0 7 3 7 2 1 131072000
0 7 2 7 1 1 131072000
0 7 1 7 0 1 131072000
0 7 0 8 0 1 131072000
0 8 0 8 1 1 131072000
0 8 1 8 2 1 131072000
0 8 2 8 3 1 131072000
0 8 3 8 4 1 131072000
0 8 4 8 5 1 131072000
0 8 5 8 6 1 131072000
0 8 6 8 7 1 131072000

0 8 7 8 8 1 131072000
0 8 8 8 9 1 131072000
0 8 9 9 9 1 131072000
0 9 9 9 8 1 131072000
0 9 8 9 7 1 131072000
0 9 7 9 6 1 131072000
0 9 6 9 5 1 131072000
0 9 5 9 4 1 131072000
0 9 4 9 3 1 131072000
0 9 3 9 2 1 131072000
0 9 2 9 1 1 131072000
0 9 1 9 0 1 131072000
0 9 0 0 0 1 131072000

C.4 netlist_BPF_VIRT.txt

0 0 0 0 0 0 0 1 1 65536000
0 0 0 0 0 0 1 1 1 32768000
0 0 0 0 0 0 2 1 1 32768000
0 0 0 1 0 0 0 3 1 65536000
0 0 1 1 0 0 0 2 1 32768000
0 0 2 1 0 0 0 2 2 32768000
0 0 0 2 0 0 0 3 2 -65536000
0 0 0 0 0 0 1 0 1 163840000
0 0 1 0 0 0 2 0 1 163840000
0 0 2 0 0 0 3 0 1 163840000
0 0 3 0 0 0 0 0 1 163840000

Appendix D

VHDL listing

D.1 PE.vhd

```
----------------------------------------------------------------------------------
-- Project Name:  HEENS
-- Design Name:   PE.vhd
-- Version: 3.9
-- Date: 19/07/2019
--
-- Creator: Roberto Gattuso
-- Company: Universitat Politecnica de Catalunya (UPC)
--
--
-- Description:
--
-- Spike pipeline and virtualization support are introduced
----------------------------------------------------------------------------------
library IEEE;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_MISC.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.log_pkg.all;
use work.SNN_pkg.all;


entity PE is
    generic(
        row_number   : integer;
        col_number   : integer;
        LoadInitFile : integer
    );
    port(
        clk          : in  std_logic;
```

```
        reset       : in  std_logic;
        reset_spike : in  std_logic;
        next_virt   : in  std_logic;
        vlayers     : in  std_logic_vector(vlayer_bits-1 downto 0);
        BRAMD_seq   : in  std_logic_vector(31 downto 0); -- bram_seq  -> BRAMD_seq
        BRAMA_spike : in  std_logic_vector(17 downto 0); -- addr_bram -> BRAMA_spike
        config      : in  std_logic;                     -- Selection signal
        AM_on       : in  std_logic;
        sp_IntExt   : in  std_logic;
        en_x        : in  std_logic;
        en_y        : in  std_logic;
        en_spike_tx : in  std_logic;
        spike_input : in  std_logic;
        spike_output: out std_logic

    );
end PE;

architecture behavioral of PE is

    component SynapNeuralMemory is
    generic(
        row_number   : integer;
        col_number   : integer;
        LoadInitFile: integer
    );
    port(
        clka  : in  std_logic;
        ena   : in STD_LOGIC;
        wea   : in  std_logic_vector(0 downto 0);
        addra : in  std_logic_vector(9 downto 0);
        dina  : in  std_logic_vector(31 downto 0);
        douta : out std_logic_vector(31 downto 0)
    );
    end component;

    component BRAM_Array_Memories is
    generic(
        row_number   : integer;
        col_number   : integer;
        LoadInitFile: integer
    );
    port(
        clk         : in  std_logic;
        reset       : in  std_logic;
        en          : in  std_logic;
        Sp_IntExt   : in  std_logic;
        BRAMA_spike : in  std_logic_vector(AER_RX_WIDTH - 1 downto 0);
        BRAMD_seq   : in  std_logic_vector(GLOBAL_SYN - 1 downto 0);
```

```
        Config         : in   std_logic;
        AM_on          : in   std_logic;
        reset_spikeReg   : in   std_logic;
        GblSpike       : out std_logic_vector(GLOBAL_SYN - 1 downto 0);
        LclSpike       : out std_logic_vector((size_x * size_y) - 2 downto 0)
    );
    end component;

    component REG is
    port(
        clk        : in   std_logic;
        reset      : in   std_logic;
        regcode    : in   std_logic_vector(2 downto 0);
        en         : in   std_logic_vector(7 downto 0);
        data_in0   : in   std_logic_vector(15 downto 0);
        data_in1   : in   std_logic_vector(15 downto 0);
        data_in2   : in   std_logic_vector(15 downto 0);
        data_in3   : in   std_logic_vector(15 downto 0);
        data_in4   : in   std_logic_vector(15 downto 0);
        data_in5   : in   std_logic_vector(15 downto 0);
        data_in6   : in   std_logic_vector(15 downto 0);
        data_in7   : in   std_logic_vector(15 downto 0);
        data_out0  : out  std_logic_vector(15 downto 0);
        data_out1  : out  std_logic_vector(15 downto 0);
        data_out2  : out  std_logic_vector(15 downto 0);
        data_out3  : out  std_logic_vector(15 downto 0);
        data_out4  : out  std_logic_vector(15 downto 0);
        data_out5  : out  std_logic_vector(15 downto 0);
        data_out6  : out  std_logic_vector(15 downto 0);
        data_out7  : out  std_logic_vector(15 downto 0)
    );
    end component;

    component ALU is
    port(
        clk       : in   std_logic;
        reset     : in   std_logic;
        InA       : in   std_logic_vector(15 downto 0);
        InB       : in   std_logic_vector(15 downto 0);
        OP_CODE   : in   std_logic_vector(5 downto 0);
        OutCarry  : out  std_logic;
        OutZero   : out  std_logic;
        OutSolve  : out  std_logic_vector(31 downto 0)
    );
    end component;

-- Operation Signals
    signal data_in    : std_logic_vector(15 downto 0);
    signal addr_reg   : std_logic_vector(2 downto 0);
```

```
    signal addr_reg2  : std_logic_vector(3 downto 0);       -- JM
    signal opcode     : std_logic_vector(5 downto 0);
    signal PE_en      : std_logic;

 -- Register Bank Signals
    signal regcode    : std_logic_vector(2 downto 0);
    signal REG_en     : std_logic_vector(7 downto 0);
    signal en_addr    : std_logic_vector(7 downto 0);
    signal en_op      : std_logic_vector(7 downto 0);
    signal data_in0   : std_logic_vector(15 downto 0);
    signal data_in1   : std_logic_vector(15 downto 0);
    signal data_in2   : std_logic_vector(15 downto 0);
    signal data_in3   : std_logic_vector(15 downto 0);
    signal data_in4   : std_logic_vector(15 downto 0);
    signal data_in5   : std_logic_vector(15 downto 0);
    signal data_in6   : std_logic_vector(15 downto 0);
    signal data_in7   : std_logic_vector(15 downto 0);
    signal data_out0  : std_logic_vector(15 downto 0);
    signal data_out1  : std_logic_vector(15 downto 0);
    signal data_out2  : std_logic_vector(15 downto 0);
    signal data_out3  : std_logic_vector(15 downto 0);
    signal data_out4  : std_logic_vector(15 downto 0);
    signal data_out5  : std_logic_vector(15 downto 0);
    signal data_out6  : std_logic_vector(15 downto 0);
    signal data_out7  : std_logic_vector(15 downto 0);

 -- SynapNeuralMemory Signals
    signal SynNeuMem_WE          : std_logic_vector(0 downto 0);
    signal SynNeuMem_WE_aux      : std_logic_vector(0 downto 0);
    signal SynNeuMem_Addr        : std_logic_vector(9 downto 0);
    signal SynNeuMem_Addr_aux    : std_logic_vector(9 downto 0);
    signal SynNeuMem_DataIN      : std_logic_vector(31 downto 0);
    signal SynNeuMem_DataIN_aux  : std_logic_vector(31 downto 0);
    signal SynNeuMem_DataOUT     : std_logic_vector(31 downto 0);

 -- LFSR Signals
    signal LFSR       : std_logic_vector(63 downto 0);
    signal random_en  : std_logic;

 -- ALU Signals
    signal ALU_A          : std_logic_vector(15 downto 0);
    signal ALU_B          : std_logic_vector(15 downto 0);
--  signal OP_CODE        : std_logic_vector(5 downto 0);
    signal ALU_Carry_aux  : std_logic;
    signal ALU_Carry      : std_logic;
    signal ALU_Zero_aux   : std_logic;
    signal ALU_Zero       : std_logic;
    signal ALU_Solve      : std_logic_vector(31 downto 0);
    signal B_aux          : std_logic_vector(15 downto 0);
```

```
    signal B_aux2          : std_logic_vector(15 downto 0);
    signal B_aux3          : std_logic_vector(15 downto 0); -- JM
    signal flag_MUL        : std_logic;
    signal no_MUL          : std_logic;
    signal signal_MUL      : std_logic;

 -- Freeze Signals
    signal freeze_reg : std_logic_vector(7 downto 0);
    signal no_freeze  : std_logic;

 -- Spike Signals
    signal spike_local  : std_logic_vector((size_x * size_y) - 2 downto 0);
    signal spike_global : std_logic_vector(GLOBAL_SYN - 1 downto 0);
    signal spike        : std_logic;
    signal spike_out    : std_logic_vector(max_neuron_v downto 0);
    signal spike_aux    : std_logic_vector(2**log2_ceil((size_x * size_y) + GLOBAL_SYN) -
    signal load_sp      : std_logic;
    signal spike_a      : std_logic;
    signal virt_num     : integer range 0 to max_neuron_v := 0; -- JM

begin

    data_in   <= BRAMD_seq(27 downto 12);
    addr_reg  <= BRAMD_seq(8 downto 6);
    addr_reg2 <= BRAMD_seq(9 downto 6);-- JM
    opcode    <= BRAMD_seq(5 downto 0) when config = '0' else
                 (others => '0');

    random_enable_process : process
    begin
        wait until clk'event and clk = '1';
        if ( (reset = '1') OR (opcode = RANDOFF) ) then
            random_en <= '0';
        elsif (opcode = RANDON) then
            random_en <= '1';
        end if;
    end process;


    LFSR_process : process
    begin
        wait until clk'event and clk = '1';
        if reset = '1' then
            LFSR <= X"0000000000000001";
        elsif (opcode = SEED) then   --JM
            LFSR(63 downto 32) <= LFSR(31 downto 0);
            LFSR(31 downto 0)  <= data_out1 & data_out0;
        elsif ( (opcode = RANDON) OR (random_en = '1') ) then
            LFSR(63)            <= LFSR(0);
```

```
            LFSR(62 downto 4)  <= LFSR(63 downto 5);
            LFSR(3)            <= LFSR(4) xor LFSR(0);
            LFSR(2)            <= LFSR(3) xor LFSR(0);
            LFSR(1)            <= LFSR(2);
            LFSR(0)            <= LFSR(1) xor LFSR(0);
        end if;
    end process;

    BRAM_write : process
    begin
        wait until clk'event and clk = '1';
        if reset = '1' then
            SynNeuMem_Addr_aux <= (others => '0');
        elsif opcode = STORESP then
            SynNeuMem_Addr_aux <= SynNeuMem_Addr_aux + 1;
        elsif opcode = LOADBP then
            SynNeuMem_Addr_aux <= data_in(9 downto 0);
        end if;
    end process;

    no_freeze <= '1' when ((freeze_reg = x"00") AND (config = '0')) else
                 '0';

    PE_en <= '1' when config = '0' else
             (en_x AND en_y);

    CFLAG_process : process -- carry flag
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            ALU_Carry <= '0';
        elsif (no_freeze = '1') AND ( (opcode = SHLN) OR (opcode = SHLAN) OR (opcode = SH
            if (ALU_Carry_aux = '0') then
                ALU_Carry <= '0';
            elsif (ALU_Carry_aux = '1') then
                ALU_Carry <= '1';
            end if;
        end if;
    end process;

    ZFLAG_process : process -- zero flag
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            ALU_Zero <= '0';
      elsif (no_freeze = '1') AND ( (opcode = SETZ) OR (opcode = RST) OR (opcode = CLRZ)
            if (ALU_Zero_aux = '0') then
                ALU_Zero <= '0';
            elsif (ALU_Zero_aux = '1') then
```

```
            ALU_Zero <= '1';
          end if;
        end if;
    end process;

    FREEZE_process : process
    begin
        wait until clk'event and clk = '1';
        if reset = '1' then
            freeze_reg <= (others => '0');
        else
            if (opcode = FREEZEC) then
                if (ALU_Carry = '1') then
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '1';
                else
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '0';
                end if;
            elsif (opcode = FREEZENC) then
                if (ALU_Carry = '0') then
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '1';
                else
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '0';
                end if;
            elsif (opcode = FREEZEZ) then
                if (ALU_Zero = '1') then
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '1';
                else
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '0';
                end if;
            elsif (opcode = FREEZENZ) then
                if (ALU_Zero = '0') then
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '1';
                else
                    freeze_reg(7 downto 0) <= freeze_reg(6 downto 0) & '0';
                end if;
            elsif (opcode = OP_UNFREEZE) then -- UNFREEZE: opcode not valid
                freeze_reg(7 downto 0) <= '0' & freeze_reg(7 downto 1);
            end if;
        end if;
    end process;

-- Max virtualization number by software
    Max_virtualizations : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            virt_num <= 0;
        elsif opcode = SPMOV then
            virt_num <= to_integer( unsigned( data_out0 ) );
```

```
        end if;
    end process;


    SPIKE_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            spike_out(max_neuron_v downto 0) <= (others => '0');
        elsif load_sp = '1' then
            spike_out((max_neuron_v) downto 1) <= spike_out((max_neuron_v - 1) downto 0)
            spike_out(0) <= data_out0(0);
        elsif en_spike_tx = '1' then
            spike_out( to_integer(unsigned(vlayers)) ) <= spike_input;
        elsif next_virt = '1' then
            spike_out((max_neuron_v) downto 1) <= spike_out((max_neuron_v - 1) downto 0)
            spike_out(0) <= '0';

        end if;
    end process;


-- LOAD Spike enable
    with opcode select
    load_sp <= '1' when STOREPS,
               '0' when others;


    spike_output <= spike_out( to_integer(unsigned(vlayers)) );


    with opcode select
    regcode <= "001" when RST,
               "010" when SET,
               "011" when SWAPS,
               "100" when MOVSR,
               "101" when MOVRS,
               "000" when others; -- External write case

-- Register Enable vector
    with opcode select
    en_op <= en_addr when LDALL | MOVR | SWAPS | RST | SET | MOVRS | MOVSR, -- JM fix --
             "00000001" when LLFSR | SHLN | SHRN | RTL | RTR | INC | DEC | OP_ADD | OP_SU
             "00000011" when LOADSP | LOADSN | MUL,
             "00000000" when others;

-- Register enable vector by address
    with addr_reg select
    en_addr <= "00000001" when "000",
```

```
                "00000010" when "001",
                "00000100" when "010",
                "00001000" when "011",
                "00010000" when "100",
                "00100000" when "101",
                "01000000" when "110",
                "10000000" when "111",
                "00000000" when others;

-- Auxiliary ALU_B operand
   with addr_reg select
   B_aux <= data_out0 when "000",
            data_out1 when "001",
            data_out2 when "010",
            data_out3 when "011",
            data_out4 when "100",
            data_out5 when "101",
            data_out6 when "110",
            data_out7 when "111",
            X"0000"   when others;

   with addr_reg select
   B_aux2 <= X"0000" when "000",
             X"0001" when "001",
             X"0002" when "010",
             X"0003" when "011",
             X"0004" when "100",
             X"0005" when "101",
             X"0006" when "110",
             X"0007" when "111",
             X"0000" when others;

      B_aux3(3 downto 0)  <= addr_reg2;
      B_aux3(15 downto 4) <= (others => '0');

-- ALU_B operand of ALU.
   with opcode select
     ALU_B <= B_aux   when INC | DEC | OP_ADD | OP_SUB | MUL | MULS | OP_AND | OP_OR |
              B_aux2 when SHLN | SHRN | RTL | RTR |SHLAN | SHRAN,
              B_aux3 when BITSET | BITCLR,
              X"0000" when others;


   spike_aux(spike_aux'length - 1 downto spike_global'length + spike_local'length) <= (
   spike_aux(spike_global'length + spike_local'length - 1 downto 0) <= spike_global & s

   spike <= spike_aux(to_integer(unsigned(SynNeuMem_Addr(log2_ceil((size_x * size_y) + (
```

```
-- Write R0 (ACC)
   with opcode select
   data_in0 <= data_in when LDALL,
               LFSR(15 downto 0) when LLFSR,
               ( SynNeuMem_DataOUT(15 downto 1) & spike_a ) when LOADSP | LOADSN,
               x"0000" when RST,
               x"FFFF" when SET,
               B_aux when MOVA,
               data_out0 when MOVR,
               ALU_Solve(15 downto 0) when others;


   with opcode select
   spike_a <= spike when LOADSP,
              SynNeuMem_DataOUT(0) when others;

-- Write R1
   with opcode select
   data_in1 <= data_in when LDALL,
               SynNeuMem_DataOUT(31 downto 16) when LOADSP | LOADSN,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               ALU_Solve(31 downto 16) when others;

-- Write R2
   with opcode select
   data_in2 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;

-- Write R3
   with opcode select
   data_in3 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;

-- Write R4
   with opcode select
   data_in4 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;
```

```
-- Write R5
   with opcode select
   data_in5 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;

-- Write R6
   with opcode select
   data_in6 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;

-- Write R7
   with opcode select
   data_in7 <= data_in when LDALL,
               x"0000" when RST,
               x"FFFF" when SET,
               data_out0 when MOVR,
               x"0000" when others;

-- ALU result
   ALU_A <= data_out0;


   with opcode select
   SynNeuMem_WE <= b"1"  when STORESP,
          SynNeuMem_WE_aux when others;
   SynNeuMem_WE_aux(0) <= '1' when BRAMA_spike(16 downto 13) = "0111" and config = '1'
                     else '0';

   SynNeuMem_DataIN_aux <= data_out1 & data_out0;

   signal_MUL <= '1' when ((opcode = MUL) OR (opcode = MULS)) else
                 '0';

   no_MUL <= '1' when ((flag_MUL = '0') AND (signal_MUL = '1')) else
             '0';

   flag_MUL_process : process
   begin
       wait until clk'event and clk = '1';
       if (reset = '1') then
           flag_MUL <= '0';
       elsif (no_MUL = '1') then
           flag_MUL <= '1';
```

```
        else
            flag_MUL <= '0';
        end if;
    end process;

--  Block register enables when Freeze
    REG_en <= en_op when (no_freeze = '1' AND (no_MUL = '0')) else
              "00000000";

--  Register port map
    REG_inst: REG
    port map(
        clk       => clk,
        reset     => reset,
        regcode   => regcode,
        en        => REG_en,
        data_in0  => data_in0,
        data_in1  => data_in1,
        data_in2  => data_in2,
        data_in3  => data_in3,
        data_in4  => data_in4,
        data_in5  => data_in5,
        data_in6  => data_in6,
        data_in7  => data_in7,
        data_out0 => data_out0,
        data_out1 => data_out1,
        data_out2 => data_out2,
        data_out3 => data_out3,
        data_out4 => data_out4,
        data_out5 => data_out5,
        data_out6 => data_out6,
        data_out7 => data_out7
    );


        SynNeuMem_Addr <= BRAMA_spike(9 downto 0) when config = '1' else
                          SynNeuMem_Addr_aux;
        SynNeuMem_DataIN <= BRAMD_seq when config = '1' else
                                      SynNeuMem_DataIN_aux;

--  BRAM synaptic port map v2
    SynapNeuralMemory_inst: SynapNeuralMemory
    generic map(row_number,
                                col_number,
                                LoadInitFile
                                )
    port map(
        clka     => clk,
        wea      => SynNeuMem_WE,
```

```
        ena       => PE_en,
        addra     => SynNeuMem_Addr,
        dina      => SynNeuMem_DataIN,
        douta     => SynNeuMem_DataOUT
    );

 -- ALU port map
    ALU_inst: ALU
    port map(
        clk          => clk,
        reset        => reset,
        InA          => ALU_A,
        InB          => ALU_B,
        OP_CODE      => opcode,
        OutCarry     => ALU_Carry_aux,
        OutZero      => ALU_Zero_aux,
        OutSolve     => ALU_Solve
    );

    BRAM_Array_Memories_inst: BRAM_Array_Memories
    generic map(row_number,
                                    col_number,
                                    LoadInitFile
    )
    port map(
        Config           => config,          -- MUX signal
        BRAMA_spike      => BRAMA_spike,     -- addr_bram MUX spike_in
        BRAMD_seq        => BRAMD_seq,       -- data_bram MUX data_seq
        AM_on            => AM_on,           -- inputs del array
        en               => PE_en,           -- inputs del array
        reset            => reset,
        reset_spikeReg => reset_spike,
        Sp_IntExt        => sp_IntExt,       -- inputs del array
        clk              => clk,
        GblSpike         => spike_global,
        LclSpike         => spike_local
    );

end behavioral;
```

D.2 PE_row.vhd

```
——————————————————————————————————————————————————————————————————————
-- Project Name:  HEENS
-- Design Name:   PE_row.vhd
-- Version: 4
-- Date: 19/07/2019
--
-- Creator: Roberto Gattuso
-- Company: Universitat Politecnica de Catalunya (UPC)
--
--
-- Description:
--
-- Spike pipeline and virtualization support are introduced
——————————————————————————————————————————————————————————————————————
library IEEE;
use IEEE.std_logic_1164.all;
use work.log_pkg.all;
use work.SNN_pkg.all;
use ieee.numeric_std.all;

entity PE_row is
        generic(
             row_number: integer;
             LoadInitFile: integer
             );
    port (
           clk         : in  std_logic;
           reset       : in  std_logic;
           reset_spike : in  std_logic;
           next_virt   : in  std_logic;
           vlayers     : in  std_logic_vector(vlayer_bits-1 downto 0);
           BRAMD_seq   : in  std_logic_vector(31 downto 0); -- bram_seq
-> BRAMD_seq
           BRAMA_spike : in  std_logic_vector(17 downto 0); -- addr_bram -> BRAMA_spike
           config      : in  std_logic;                     -- Selection signal
--
           AM_on       : in  std_logic;
           sp_IntExt   : in  std_logic;
           enable_x    : in  std_logic_vector(size_x_1 downto 0); --
           enable_y    : in  std_logic;
           next_PE_row : in  std_logic;
           spike_inR   : in std_logic_vector(size_x_1 downto 0);
           spike_outR  : out std_logic_vector(size_x_1 downto 0)
           );
end PE_row;

architecture connection of PE_row is
```

```
    component PE is  -- Process Element cell
    generic(
            row_number   : integer;
            col_number   : integer;
            LoadInitFile: integer
            );
    port (            -- Ports of Process Element
          clk          : in  std_logic;
          reset        : in  std_logic;
          reset_spike  : in  std_logic;
          next_virt    : in  std_logic;
          vlayers      : in  std_logic_vector(vlayer_bits-1 downto 0);
          BRAMD_seq    : in  std_logic_vector(31 downto 0);
          BRAMA_spike  : in  std_logic_vector(17 downto 0);
          config       : in  std_logic;
          AM_on        : in  std_logic;
          sp_IntExt    : in  std_logic;
          en_x         : in  std_logic;
          en_y         : in  std_logic;
          en_spike_tx  : in  std_logic;
          spike_input  : in  std_logic;
          spike_output: out std_logic
          );
    end component;

 -- Registered signals
    signal BRAMD_seq_reg    : std_logic_vector(31 downto 0);
    signal BRAMA_spike_reg  : std_logic_vector(17 downto 0);
    signal reset_reg        : std_logic;
    signal reset_spike_reg  :  std_logic;
    signal config_reg       :  std_logic;
    signal AM_on_reg        :  std_logic;
    signal sp_IntExt_reg    :  std_logic;
    signal enable_x_reg     :  std_logic_vector(size_x_1 downto 0);
    signal enable_y_reg     :  std_logic;

begin

 -- Array of PE port map generation
    gPEj: for j in 0 to size_x_1 generate      -- columns
        PEj: entity work.PE
        generic map(row_number,
                    j,
                    LoadInitFile
                           )
        port map(
                 clk,
                 reset,
```

```
                reset_spike_reg ,
                next_virt ,
                vlayers ,
                BRAMD_seq_reg ,              -- bram_seq
                BRAMA_spike_reg ,            -- addr_bram
                config_reg ,                 -- config
                AM_on_reg ,                  -- AIR flag
                sp_IntExt_reg ,              -- sp_IntExt
                enable_x_reg(j) ,            -- PE_col_enable
                enable_y_reg ,               -- PE_row_enable
                next_PE_row ,
                spike_inR(j) ,               -- spike_input
                spike_outR(j)                -- spike_output
                );
end generate gPEj;                          -- columns

reset_process : process
begin
    wait until clk'event and clk = '1';
        reset_reg <= reset;
end process;

reset_spike_process : process
begin
    wait until clk'event and clk = '1';
    if (reset = '1') then
        reset_spike_reg <= '0';
    else
        reset_spike_reg <= reset_spike;
    end if;
end process;

BRAMD_seq_process : process
begin
    wait until clk'event and clk = '1';
    if (reset = '1') then
        BRAMD_seq_reg <= (others => '0');
    else
        BRAMD_seq_reg <= BRAMD_seq;
    end if;
end process;

BRAMA_spike_process : process
begin
    wait until clk'event and clk = '1';
    if (reset = '1') then
        BRAMA_spike_reg <= (others => '0');
    else
        BRAMA_spike_reg <= BRAMA_spike;
```

```
        end if;
    end process;

    config_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            config_reg <= '0';
        else
            config_reg <= config;
        end if;
    end process;

    AM_on_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            AM_on_reg <= '0';
        else
            AM_on_reg <= AM_on;
        end if;
    end process;

    sp_IntExt_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            sp_IntExt_reg <= '0';
        else
            sp_IntExt_reg <= sp_IntExt;
        end if;
    end process;

    enable_x_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            enable_x_reg <= (others => '0');
        else
            enable_x_reg <= enable_x;
        end if;
    end process;

    enable_y_process : process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            enable_y_reg <= '0';
        else
```

```
            enable_y_reg <= enable_y;
        end if;
    end process;

end architecture connection; -- of PE_row
```

D.3 PE_array.vhd

```
----------------------------------------------------------------------------------
-- Project Name:  HEENS
-- Design Name:   PE_array.vhd
-- Version: 3.4
-- Date: 19/07/2019
--
-- Creator: Roberto Gattuso
-- Company: Universitat Politecnica de Catalunya (UPC)
--
--
-- Description:
--
-- Spike pipeline and virtualization support are introduced
----------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.log_pkg.all;
use work.SNN_pkg.all;
use ieee.numeric_std.all;

entity PE_array is
        generic(
              LoadInitFile: integer
              );
    port (
            en_si          : in  std_logic;
            clk            : in  std_logic;
            reset          : in  std_logic;
            reset_spike    : in  std_logic;
            row_pe         : in  std_logic_vector(4 downto 0);  -- Row PE enable
            col_pe         : in  std_logic_vector(4 downto 0);  -- Col PE enable
            BRAMD_seq      : in  std_logic_vector(31 downto 0); -- bram_seq
-> BRAMD_seq
            BRAMA_spike    : in  std_logic_vector(17 downto 0); -- addr_bram -> BRAMA_spike
            config         : in  std_logic;                     -- Selection signal
            AM_on          : in  std_logic;
            sp_IntExt      : in  std_logic;  -- Enable signal for BRAM memories
            vlayers_count  : in  std_logic_vector(vlayer_bits-1 downto 0);
            eo_spike       : out std_logic;
            spike_valid    : out std_logic;
            row_sp         : out std_logic_vector(4 downto 0); -- Row spike addr
            col_sp         : out std_logic_vector(4 downto 0);  -- Col spike addr
            virt_sp        : out std_logic_vector(2 downto 0)
            );
end PE_array;

architecture interconnect of PE_array is
```

```
    component PE_row is  -- Process Elements Row
        generic(
            row_number: integer;
            LoadInitFile: integer
            );
    port (-- Ports de la fila de Process Elements
          clk         : in  std_logic;
          reset       : in  std_logic;
          reset_spike : in  std_logic;
          next_virt   : in  std_logic;
          vlayers     : in  std_logic_vector(vlayer_bits-1 downto 0);
          BRAMD_seq   : in  std_logic_vector(31 downto 0); -- bram_seq
-> BRAMD_seq
          BRAMA_spike : in  std_logic_vector(17 downto 0); -- addr_bram -> BRAMA_spike
          config      : in  std_logic;                     -- Selection signal
--
          AM_on       : in  std_logic;
          sp_IntExt   : in  std_logic;
          enable_x    : in  std_logic_vector(size_x_1 downto 0); --
          enable_y    : in  std_logic;
          next_PE_row : in  std_logic;
          spike_inR   : in std_logic_vector(size_x_1 downto 0);
          spike_outR  : out std_logic_vector(size_x_1 downto 0)
          );
    end component;


    type     arrC is array(size_y_1 downto 0) of std_logic_vector(size_x_1 downto 0); --
    type     arrV is array(max_neuron_v downto 0) of arrC;
    signal   zero           : std_logic_vector(size_x_1 downto 0) := (others => '0');
    signal   row_aux        : integer range 0 to size_y;
    signal   col_aux        : integer range 0 to size_x_1;
    signal   virt_aux       : integer range 0 to (max_neuron_v+1);
    signal   en_col         : std_logic_vector(size_x_1 downto 0);
    signal   en_row         : std_logic_vector(size_y_1 downto 0);
    signal   no_spike       : std_logic;
    signal   en_spike       : std_logic;
    signal   sp_valid       : std_logic;
    signal   shift_sp       : std_logic;
    signal   spike_displ    : arrV;
    signal   PE_array_count : integer range 0 to size_y;
    signal   spike_buffer   : std_logic_vector(size_x_1 downto 0);
    signal   next_PE_row    : std_logic;
    signal   spike_inA      : arrC;
    signal   spike_outA     : arrC;


begin -- interconnect of net_addr
```

```
-- Matrix of PE_row (Arrays of PE) port map generation
   gPEi: for i in 0 to size_y_1 generate      -- rows
        PEi: entity work.PE_row
        generic map (i,
                LoadInitFile)
        port map(
                clk,
                reset,
                reset_spike,
                shift_sp,
                vlayers_count,
                BRAMD_seq,                          -- bram_seq
                BRAMA_spike,                        -- addr_bram
                config,                             -- Valor a 0 per poder provar codi
                AM_on,                              -- AM_on
                sp_IntExt,                          -- sp_IntExt
                en_col(size_x_1 downto 0),
                en_row(i),
                next_PE_row,
                spike_inA(i)(size_x_1 downto 0),
                spike_outA(i)(size_x_1 downto 0)
                );
   end generate gPEi;

-- 1st Row start vector
   gspike_in0: for j in 0 to size_x_1 generate
        spike_in0: spike_inA(0)(j) <= '0';
   end generate gspike_in0;

-- Array Interconnect
   gspikeA_coli: for i in 0 to size_x_1 generate
        gspikeA_rowj: for j in 0 to (size_y_1 - 1) generate
            spikeAj: spike_inA(j+1)(i) <= spike_outA(j)(i);
        end generate gspikeA_rowj;
   end generate gspikeA_coli;

--PE_Array buffer
   spike_buffer_process: process
   variable temp: integer range 0 to size_x_1;
   begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            PE_array_count <= size_y;
            next_PE_row <= '0';
            no_spike <= '0';
            spike_buffer <= (others=> '0');
            sp_valid <= '0';
        elsif (en_spike = '1' and en_si = '1' and shift_sp ='0') then
```

```
                if (next_PE_row = '1') then
                    next_PE_row <= '0';
                    spike_buffer <= spike_outA(size_y_1); -- Buffer shift
                elsif (spike_buffer = zero) then
                    sp_valid <= '0';
                    if (PE_array_count = 0) then
                        no_spike <= '1';
                        PE_array_count <= size_y;
                    else
                        PE_array_count <= PE_array_count - 1;
                        next_PE_row <= '1';  -- PE array shift
                    end if;
                else
                    for i in 0 to size_x_1 loop
                        if (spike_buffer(i) = '1') then
                            temp := i;
                            spike_buffer(i) <='0';
                            sp_valid <= '1';
                            exit;
                        else
                            sp_valid <= '0';
                        end if;
                    end loop;
                end if;
            else
                no_spike <= '0';
            end if;
            col_aux <= temp;
            row_aux <= PE_array_count;
    end process;

-- Construction of spike vector
    col_sp <= std_logic_vector(to_unsigned(col_aux,5)) when sp_valid = '1' else -- NEXT
              ( others => '0' );
    row_sp <= std_logic_vector(to_unsigned(row_aux,5)) when sp_valid = '1' else -- NEXT
              ( others => '0' );
    virt_sp <= std_logic_vector(to_unsigned(virt_aux,3)) when sp_valid = '1' else
              ( others => '0' );
    spike_valid <= sp_valid;

     -- Spike display
    spike_displ_process:process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1')  then
            spike_displ<= ((others=> (others=> (others=>'0'))));
        elsif sp_valid = '1' then
            spike_displ(virt_aux)(row_aux)(col_aux) <='1';
        else
```

```
            spike_displ<= ((others=> (others=> (others=>'0'))));
        end if;
    end process;


 -- End_flag of pulse draining process
    shift_sp <= no_spike AND en_spike AND en_si;
    eo_spike <= shift_sp;

 -- Spike enable register
    SPIKE_enable: process
    begin
        wait until clk'event and clk = '1';
        if ( (reset = '1') or (shift_sp = '1') or (en_si = '0')) then
            en_spike <= '0';
        elsif (en_si = '1') then
            en_spike <= '1';
        end if;
    end process;

     -- Virtual layer counter
    VIRT_counter: process
    begin
        wait until clk'event and clk = '1';
        if (reset = '1') then
            virt_aux <= 0;
        elsif (en_si = '0') then
                virt_aux <= 0;
        elsif (shift_sp = '1') then
                virt_aux <= virt_aux + 1;
        end if;
    end process;

     -- Write enable single PE process
    PE_row_enabling: process (row_pe)
    begin
        en_row <= (others => '0'); -- default
        en_row( to_integer( unsigned( row_pe ) ) ) <= '1';
    end process;

    PE_col_enabling: process (col_pe)
    begin
        en_col <= (others => '0'); -- default
        en_col( to_integer( unsigned( col_pe ) ) ) <= '1';
    end process;
end architecture interconnect; -- of PE_array
```

Bibliography

[1] H. Paugam-Moisy, S. Bohte, “Computing with Spiking Neuron Networks” in *Handbook of Natural Computing*, pp. 335–376, 2012.

[2] M. Zapata, U. K. Balaji, J. Madrenas, in “PSoC-Based Real-Time Data Acquisition for a Scalable Spiking Neural Network Hardware Architecture” October 2018.

[3] J. Madrenas, in “HEENS private document” February 2018.

[4] S. J. Moreno, in “Implementation of a multiprocessor array for spiking neural network emulation on FPGA” January 2017.

[5] J. Schemmel, J. Fieres, K. Meie, “Wafer-Scale Integration of Analog Neural Networks” in *IEEE International Joint Conference on Neural Networks*, 2008.

[6] A. Sripad, G. Sanchez, M. Zapata, V. P. andTaho Dorta, S. Cambria, A. Marti, K. Krishnamourthy, J. Madrenas, “SNAVA - A real-time multi-FPGA multi-model spiking neural network simulation architecture” in *Neural Networks*, v. 97, pp. 28–45, January 2018.

[7] E. M. Izhikevich, “Polychronization: Computation With Spikes” in *Neural Computation*, June 2005.

[8] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, S. Millner, “A wafer-scale neuromorphic hardware system for large-scale neural modeling” in *In Proceedings of 2010 IEEE international symposium on circuits and systems (ISCAS)*, pp. 1947–1950, 2010.

[9] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy1, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, D. S. Modha, “A million spiking-neuron integrated circuit with a scalable communication network and interface” in *Science*, v. 345 (6197), pp. 668–673, 2014.

[10] X. Jin, M. Lujan, L. A. Plana, S. Davies, S. Temple, S. B. Furber, “Modeling spiking neural networks on SpiNNaker” in *Computing in Science and Engineering*, v. 12(5), pp. 91–97, 2010.

[11] H. E. Plesser, M. Diesmann, M.-O. Gewaltig, A. Morrison, “NEST: The Neural Simulation Tool” in *Encyclopedia of Computational Neuroscience*, pp. 1–4,

2013.

[12] A. Linares-Barranco, R. Paz-Vicente, F. Gomez-Rodriguez, Jimenez, M. Rivas, G. Jimenez, A. Civit, “On the AER convolution processors for FPGA” in *Proceedings of 2010 IEEE international symposium on circuits and systems*, pp. 4237–4240, 2010.

[13] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, H. Wang, “Loihi: a Neuromorphic Manycore Processor with On-Chip Learning” in *IEEE Micro*, v. 38 (1), pp. 82–99, 2018.

[14] M. Zapata, in “Arquitectura Escalable SIMD con Conectividad Jerarquica y Reconfigurable para la Emulacion de SNN” September 2017.