POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master's Degree Thesis

Parameter Monitoring and Communication in a Ring-Topology-Based SNN Emulator Hardware

Supervisors

Prof. Jordi MADRENAS BOADAS

Prof. Mireya ZAPATA

Prof. Guido MASERA

Candidate

Corrado BONFANTI

November 2020

Summary

The work of this thesis is focused on the extraction and distribution of internal parameters belonging to the HEENS architecture, which is a scalable Spiking Neural Network emulator. All of the projects are developed by means of the VHDL description, then simulated through the *QuestaSim Advanced Simulator* and finally synthesized and implemented on a Programmable System on Chip (PSOC), in order to verify time constraints and resources exploitation. It has been carried out online with the server provided by the *Universitat Politècnica de Catalunya* (UPC).

In more details, the additions and improvements made to the architecture concern the array of Processing Elements and especially those modules relative to the Address Event Representation over Synchronous Serial Ring Topology, all previously developed by the *Integrated Smart Sensors and Health Technologies* (ISSET) research group from the UPC. The AER-SRT blocks are used to support a serial communication between different FPGA of the network.

After the first monitoring implementation had been designed and verified, the following step of the work was to improve performances, through a better exploitation of the PE-array parallel nature (it is a Single Instruction Multiple Data architecture) and by means of an hardware enhancement, since the first implementation did not represent a critical part from an area occupancy point of view. The multi-board version has been implemented on a 5x5 array configuration, while the single-board has been tested on a 13x13 architecture.

Acknowledgements

All this work took place during a very difficult period for our daily lives.The pandemic caused by the SARS-CoV-2 virus, led to stop my abroad experience a few days after my arrival in Barcelona, where I should have been working these past months at the UPC for my final thesis.

In spite of everything, I managed to remotely work and conclude my university career, with a project that left me more than satisfied. All of this, would never have been possible without the help and support of Professors Jordi Madrenas and Mireya Zapata, my supervisors. They accepted me as a graduate student to work on their projects and they have been so collaborative and precious during this period, despite all the difficulties that came from the distance and the COVID-19 issue. So, they are the first ones I want to thank with all my heart. I want also to thank my Italian advisor from "Politecnico di Torino", Prof. Guido Masera, who monitored my work and helped me during this process.

My whole academic career has been signed by the special support my family gift me, who have firmly believed in me and helped me during all the difficult times I've been through and that were always present to celebrate joyful and remarkable moments. I thank you very much, since you really helped me make it all come true.

As well as my friends, that I have always considered the lifeblood of my way of being and therefore of my successes in recent years. It is not possible to mention everyone, but I want thank each of you for have been close to me and for giving me beautiful moments.

A special thanks also goes to Antonio Caruso, Roberto Gattuso and Luca Valente, my colleagues and friends, whose help and support have been useful for this thesis work.

Table of Contents

List of Tables

List of Figures

Abbreviations

AER

Address-Event Representation

AER-SRT

Address Event Representation over Synchronous Serial Ring Topology

ALU

Arithmetic Logic Unit

ANN

Artificial Neural Network

ASM

Algorithmic State Machine

ChipId

Chip Identifier

CU

Control Unit

FIFO

First In First Out

FPGA

Field Programmable Gate Array

FSM

Finite-State Machine

GPU

Graphical Processing Unit

GUI

Graphical User Interface

HEENS

Hardware Emulator of Evolvable Neural Systems

HMI

Human Machine Interface

IMEM

Instruction Memory

IoT

Internet of things

ISA

Instruction Set Architecture

LFSR

Linear-Feedback Shift Register

LIF

Leaky Integrate-and-Fire

LIFO

Last In First Out

LSB

Less Significant Bit

LTD

Long-term Depression

LTP

Long-term Potentiation

LUT

Look Up Table

MAC

Multiply-Accumulate

MC

Master Chip

mif

Memory Initialization File

MMCM

Mixed-Mode Clock Manager

MSB

Most Significant Bit

MUX

Multiplexer

NC

Neuromorphic Chip

OL

Online learning

PC

Program Counter

PE

Processing Element

PEID

Processing Element Identifier

PSOC

Programmable System on Chip

RAM

Random Access Memory

RF

Register File

SIMD

Single Instruction Multiple Data

SNAVA

Spiking Neural Networks for Versatile Applications

SNN

Spiking Neural Network

SNRAM

Synaptic/Neural Memory

STDP

Spike-Timing-Dependent Plasticity

TD

Timing Diagram

VHDL

Very High Speed Integrated Circuits Hardware Description Language

VIRT

Virtualization

Chapter 1

Introduction

In recent years, Bio-inspired neural networks has been representing a hot topic in research community.

The great interest is motivated by many reasons, one of them is the necessity of emulating and mimic the human brain functionalities in order to better understanding the intrinsic dynamics of it [1]. Indeed, even though modern Von-Neumann machines are capable of very fast and even low-power computations and elaborations, they can achieve poor results in some common tasks that are common to human beings (image recognition, natural language processing, and so on..)[2](p.18).

An important application of Neural Networks has to deal with the Internet of things (IoT). Nowadays, a huge quantity of data is generated by the environment, in particular by sensors, actuators and all devices that interact with the external world and that are connected to modern electronic systems for further elaborations, in order to monitor and to manage all the actions and health related to them.

Such a large amount of data requires fast and efficient computations that are suitable for neural algorithms and relative implementations. A way of achieving these results is through the Online learning (OL)[3], which is a learning algorithm based on an incremental update of the algorithm itself by means of a sequential processing of single samples received at each time instant (instead of chunks of data in *batch learning*). This strategy imposes some timing, memory area and power constraints that can be accomplished by Spiking Neural Network (SNN).

A SNN is a spiking-based neural model, in particular it is a third-generation Neural Network that shows a more realistic biological approach: it is based on information carried out by spikes pattern, exploiting concepts of space and time through neural connectivity and plasticity [3]. Spiking neurons have much more complicated dynamics respect to the other popular model Artificial Neural Network (ANN) and this could make SNN more powerful. One other great advantage of this model, it is its easier and more efficient hardware implementation possibilities, since these could be based on Event-Driven computation and on Address-Event Representation (AER) , differently

from ANN [2](pp.119,173-175). These concepts will be discussed in next paragraphs.

1.1 Motivations and goals

This thesis work is focused on the Hardware Emulator of Evolvable Neural Systems (HEENS) architecture, developed by Integrated Smart Sensors and Health Technologies (ISSET) group of Universitat Politècnica de Catalunya (UPC). This is a multiple FPGA-based architecture designed for emulating SNN, by means of an array of NxN processing elements (N is a value that can be set before either simulation or synthesis), as it will be explained in next sections.

In particular, this work concerns the propagation and distribution of data representing the evolving information of the Neural Network. Indeed, since HEENS is finalized to simulate a SNN, it is strictly necessary to monitor the internal parameters of the network, such as membrane potentials, synaptic weights, in order to keep track of the current state of the neural algorithm.

The precedent version of this architecture is the one reported in [1]: this work included a SNAVA Human Machine Interface (HMI) which is a software created to control and to monitor the execution and the information relative to the proposed network in a graphical way. All data involved in these configuration and monitoring phases, were transmitted by means of G-Ethernet connections. In this particular project, the monitoring information will be collected exploiting the same ring topology communication channel that is being used for the transmission of spikes among all FPGA used in the network. The HMI at the current state has not been developed yet and neither the hardware support to collect those data and to transmit them. So, summarizing, the basic support for this future application was necessary at that initial state of the work.

In order to accomplish this goal, the already developed Address Event Representation over Synchronous Serial Ring Topology (AER-SRT) protocol [4] has been exploited together with the processing hardware, which required a fully understanding of the architecture in order to manipulate, to modify and to create those parts that are involved in the extraction and transmission of these data.

Finally, the architecture proposed and then verified through simulation has been synthesized and implemented on the Xilinx Zynq-7000 SoC ZC706 board, an PSOC device, in order to check if timing constraints had been respected and to inspect the area and power consumption.

1.2 State of art

In the current state of knowledge, it is not possible to say that science perfectly understood how human brain works and how it manages and realizes all of its functionalities.

The major interesting capabilities of a biological neuron network are scalability, high efficiency connectivity, intensive parallelism, which are all skills carried out with an optimized energy consumption. Anyway, reverse engineering has produced interesting results that nowadays find application in many fields, such robotic, image recognition, artificial intelligence. and so on [5].

It is important to understand basic concepts behind the theory that has led to such important results, so in next section a brief overview of the biological functionalities of neurons and their models will be described.

1.2.1 Neuron models

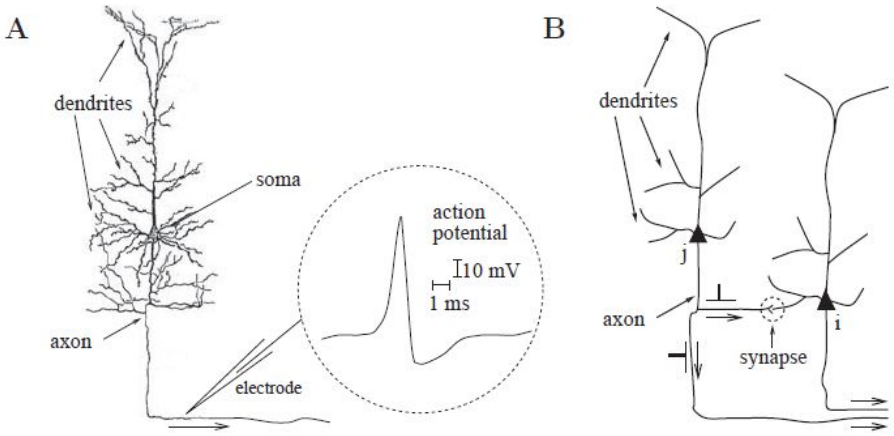


Figure 1.1: A. Single neuron(by Ramo'n y Cajal) and B. Connection and signal transmission between a pre-synaptic and a post-synaptic neuron.[6](p.13)

In Figure 1.1 it is portrayed a representation of a biological neuron: it is composed by a soma, which is like the central processing element of the neuron that generates an output signal in case a particular input threshold is exceeded; then by the dentrites that are in charge of collecting all signals from other neurons and finally by the axon, that transmits all the information outside the neuron [6]. In addition, there is the synapse that is the connection point between the axon of a sending (pre-synaptic) neuron and the dentrites of a receiving (post-synaptic) neuron. The synapse collects the spike information through complex chemical processes and, by means of neurotransmitters, it leads to the flowing of ions current into the cell of the post-synaptic neuron.

That is, an influx of ions inside the cell changes the so called "membrane-potential" of the neuron, which is the potential difference between the interior of the cell and its surroundings [6](p. 12). If this change is positive, the synapse is excitatory, whereas if the change is negative, the synapse is inhibitory.

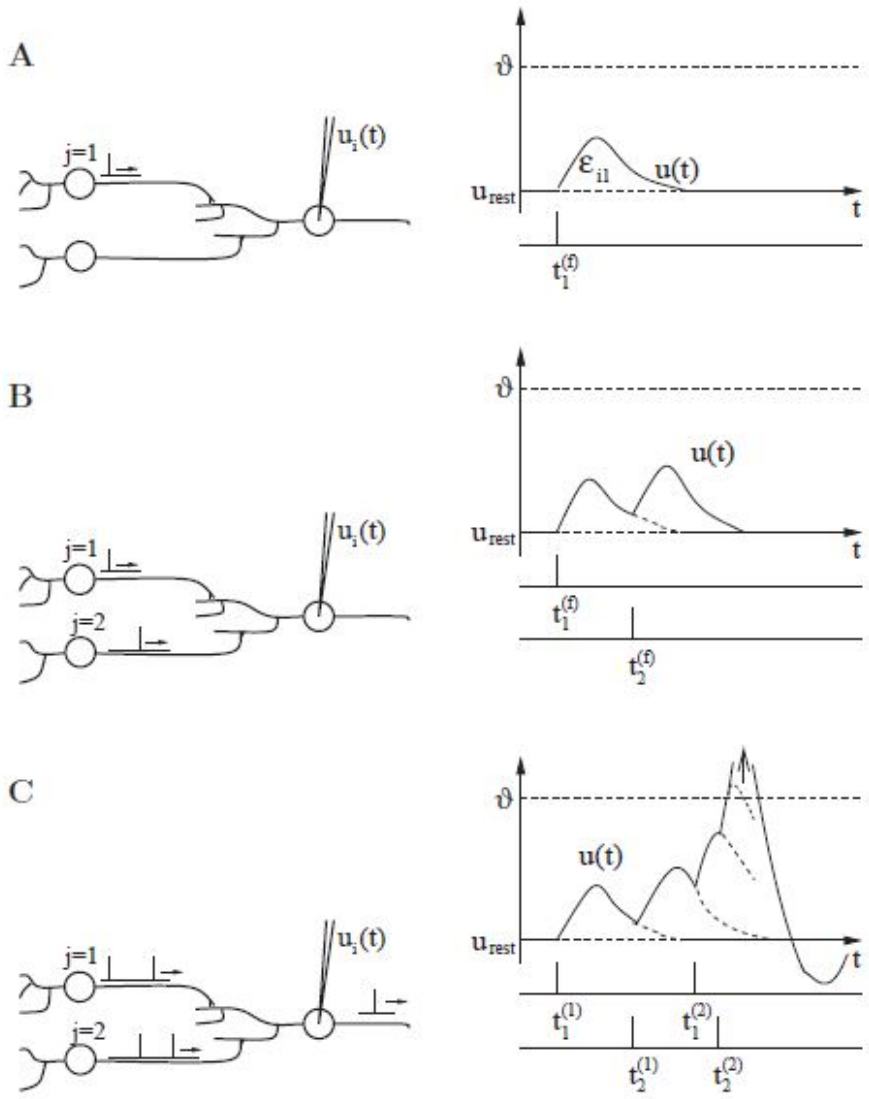


Figure 1.2: A postsynaptic neuron i receives input from two presynaptic neurons j = 1, 2.[6](p.16)

In Figure 1.2 are reported three different possible situations in which a post-synaptic neuron is excited by two pre-synaptic neurons. To better explain the dynamic of these, let's define $u_i(t)$ as the membrane potential of the neuron i and let's consider the equation[6](p.17):

$$u_i(t) = \eta(t - \hat{t}_i) + \sum_j \sum_f \epsilon_{ij}(t - t^{(f)}) + u_{rest} \tag{1.1}$$

In eq.1.1 all $\epsilon_{ij}(t)$ is defined as: $u_i(t) - u_{rest}$, where $u_{rest} \cong -65\,mV$ is the resting potential of the cell (the one the cell usually has if the cell receives no spikes at the input. Then, $\eta(t)$ is the trend of the membrane potential of the neuron i after it fired a spike at time $\hat{t}_i$. So, a post-synaptic neuron receives different spikes from pre-synaptic ones at different time instants $t^{(f)}$, which means that its membrane potential its incremented by an amount equal to the sum of all contributes $\epsilon_{ij}(t - t^{(f)})$ and if it reaches and exceeds the threshold ϑ (usually in the range of 20-30 mV above the resting potential) it fires a spike [6](pp.15,16,17).

Spikes (or action-potentials) are pulses with an amplitude of about 100 mV, that lasts 1-2 ms. After the spike, the membrane-potential goes below the resting potential (hyperpolarization) and it shows a behaviour described by the function $\eta(t)$, for a time lapse called "refractory period". Thus, the neuron has to receive a precise number of pre-synaptic spikes in a specific time window in order to fire a post-synaptic spike. These strong time and space dependencies, are those that give so much computational power to these systems and so to their close imitation represented by SNN [2](p.119).

Before talking about Spiking Neural Network, let's review some of the most important mathematical models of the biological neuron. For this part [3] [2] and the more accurate and detailed [6] had been consulted.

1. Leaky Integrate-and-Fire (LIF)

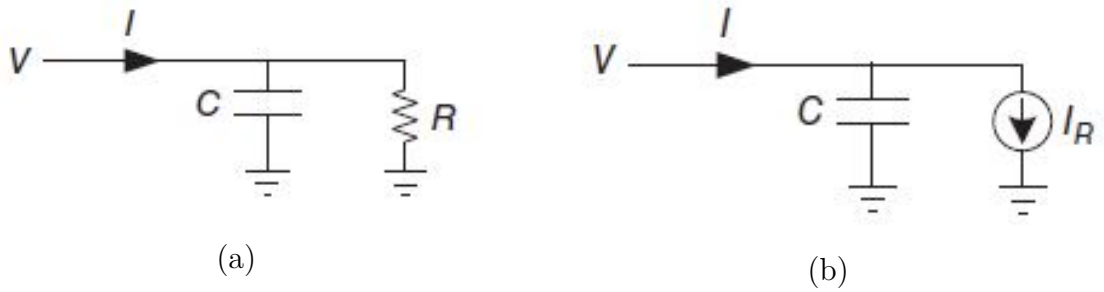


Figure 1.3: (a)The conductance-based LIF model and (b)the current-based LIF model[2](p.120).

This is a simple and a computationally effective model, as well as one of the most popular spiking neuron model used in order to build and to simulate a SNN. The model can be graphically described in Figure 1.3. The equation that describe the model is here reported:

$$C\,\frac{dV}{dt} = I - I_R \tag{1.2}$$

Basically, I is the current injected inside the neuron and I_L is the leakage ions current that leaks through the channels in the cell membrane. This leakages can be described with a conductance-based current with an impedance (R in Figure 1.3(a)), which is more biologically accurate but it strongly depends on membrane potential and that could lead to a much higher computational effort. The other model is depicted in 1.3(b), where the leakage current is approximated with an independent current source and that is a more hardware friendly model, since the current does not depend on the membrane potential.

The injected current I can be determined by the type of synapses used, which can be also in this case current-based synapses and conductance-based synapses. The latter makes the current a function of the post-synaptic potential, while the former doesn't lead to this dependecy[2](p.121).

In [3] it is underlined that in a more general form the LIF can contain a refractory period in which dynamics are stopped for a fixed period. Other equations that describe the conductance-based model are reported below:

$$I(t) = I_R(t) + I_C(t) \tag{1.3}$$

$$\tau_m \frac{du(t)}{dt} = -u(t) + RI(t) \tag{1.4}$$

$$t^f : u(t^f) = \vartheta \tag{1.5}$$

In Eq.1.4 $\tau_m = RC$ is the membrane time constant, while t^f is the firing time in Eq.1.5: after that the membrane-potential is reset to the resting potential [3].

2. Hodgkin–Huxley

The Hodgkin–Huxley model is a more biologically accurate mathematical model, which takes into account the Nerst potential, that is the potential difference between the cell end the extracellular liquid caused by the ion transport through the cell membrane. The model is graphically depicted in Figure 1.4 and it is mathematically described by Eq.(1.6)

$$I = C\,\frac{dV}{dt} + G_{Na}m^3(V - V_{Na}) + G_K n^4(V - V_K) + G_L(V - V_L) \tag{1.6}$$

In this equation, V_{Na}, V_K and V_L are called reverse potentials, while G_{Na}, G_K and G_L are the conductance of the sodium, potassium, and leakage channels respectively. Other variables n, m are gating variables which dynamics are described in [7], [2](pp.136-137) and in [3].

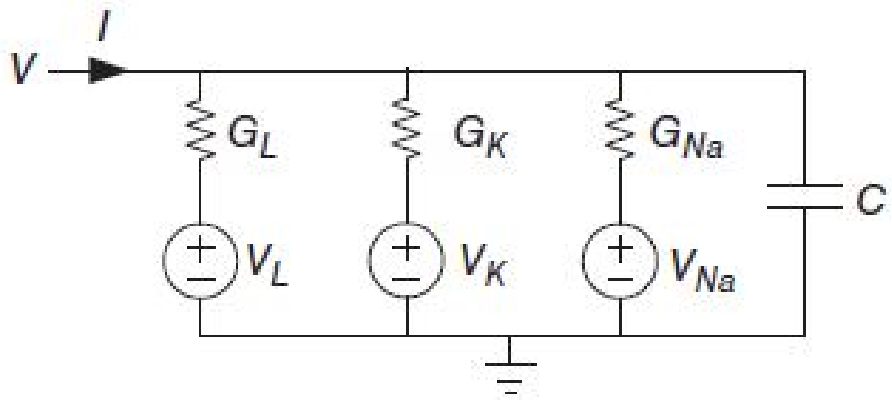


Figure 1.4: The Hodgkin-Huxley model [2](p.120)

This model is able to describe very accurately the dynamics of the neuron, but it requires too high computational efforts [2](137).

3. Izhikevich

   Since the Hodgkin–Huxley model could become prohibitive for many application and that the LIF does not very faithfully mimic the dynamics of neuron (even if it might be very useful in many SNN due to its semplicity)[2](p.121,122), the Izhikevich model[8] claims to be a good intermediate between the biological accuracy of Hodgkin–Huxley and the computational efficiency of LIF models [3]. The model is described by Eqs.1.7 and 1.8.

$$\frac{dV}{dt} = 0.004V^2 + 5V + 140 - U + I \tag{1.7}$$

$$\frac{dU}{dt} = a(bV - U) \tag{1.8}$$

   In this equations, V is the membrane potential while U is the membrane recovery. If $V \geq 30\,mV$, then V is reset to c and U is reset to $U + d$. The variables a, b, c and d are model parameters.

   This model is able to reproduce many phenomena observed in biological neurons with a computational complexity comparable to that of an LIF model[2](p.121).

1.3 Spiking Neural Networks

Spiking Neural Networks have aroused an increasing interest in the research community since they are able to reproduce with a more biological realistic approach the functionalities of a neuron network, due to the exploitation of spikes information and computation. The other popular model ANN is considered a more simplified versions of biological neural networks in terms of structure and function. Furthermore, the simply nature of a SNN based on Leaky Integrate-and-Fire approach, makes them better suited for an hardware implementation [3].

The main differences between a SNN and a ANN can be summarized here[2](p.119): (1)a non spiking network ANN uses real-value activations to convey information, whereas a spiking neuron modulates information on spikes, (2) a ANN does not usually have memory, while a SNN generally does. Finally (3) most SNNs are based on a time-varying nature, while the output generated by ANN generally is not an output of time.

In Figure 1.5, other differences are reported [2](p.124). It is important to underline that in an ANN the outputs from previous layers (pre-synaptic neurons), are real numbers which come out from many combinational logics, like Multiply-Accumulate (MAC). So, it is basically memory less, since the output depends on its particular layer position (in a multi-layer network) and on a particular class that output belongs: if it is activated (by means of its activation function) it means that a particular input category has been presented.

On the other side, in a simple LIF model, output from neurons (spikes) are more like binary vectors that have a spatial distribution but also a temporal distribution, which means that a spiking neuron can be implemented as a Finite-State Machine (FSM), where the output depends on its present inputs, but also on the past history of the input values. A SNN has an inherent memory that leads it to be often trained to learn spatio-temporal patterns.

In Figure 1.6 there is an analogy between biological neurons and artificial spiking neurons on the right, in which it is shown that a post-synaptic spikes is the result of a particular sequence of input spikes that, basing on the weight and on the type of the synapse (inhibitory and excitatory), causes the output neuron to exceed the threshold value.

1.3.1 Data encoding

An important issue in a SNN, is related to proper encoding strategy to apply. Indeed, the information carried by an analog input signal has to be translated in a spatio-temporal pattern of spikes, which is not trivial, since it has to take into account the signal characteristics, in time and in the frequency domain, the presence of noise and how all of these features can be affected by a particular encoding scheme [9].

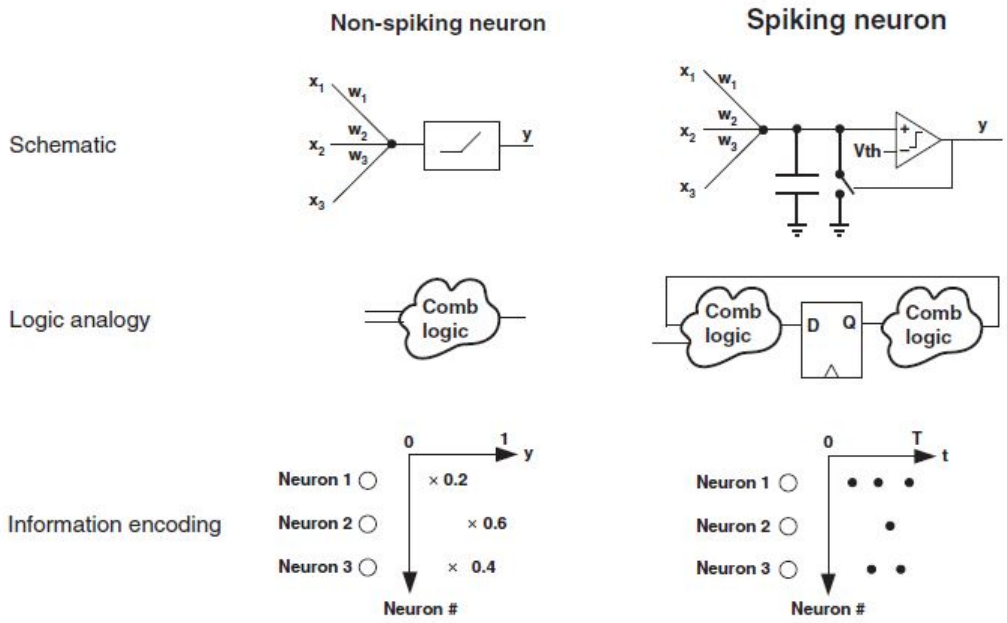


Figure 1.5: Comparison between spiking neurons and non-spiking neurons[2] (p.123)

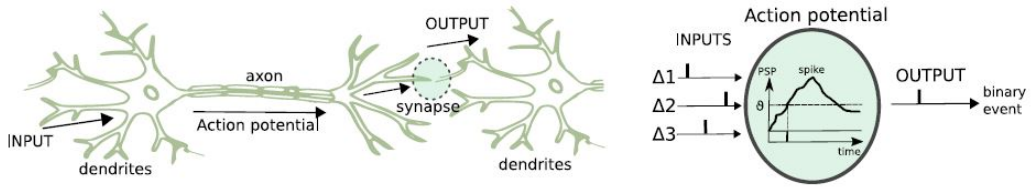


Figure 1.6: Biological neuron and its association with an artificial spiking neuron[6]

There are two main encoding scheme that can be found in literature: $rate-based\,encoding$ and $temporal-encoding$. The former is focused on spiking characteristics in a certain window of time and it can be related to three different notions of mean firing rate, that are: average ("rate as a spike count") over time, average over several repetitions of the experiment ("rate as a spike density") , or average over a population of neurons ("rate as a population activity")[3].

$Temporal-encoding$ extracts information on the exact timing of a spike, which marks a change in the value of the input signal and this is considered to be the realistic biological behavior of neurons [9]. In $temporal-encoding$ there are three main timing information schemes about spikes: "time-to-first-spike" when all timing features are related to the code for the timing of the first spike, then "phase" which is as the first, but with a periodic signal and finally "correlation and synchrony" when a spike code is

based on the reference signals from other neurons[3].

1.3.2 Synaptic Plasticity

Many electrophysiological experiments have proven that the response amplitude of a given post-synaptic neuron is not fixed over the time, but it is conditioned by the input spikes of its pre-synaptic neuron/neurons. Interesting results in [6](p.363) are shown in Figure 1.7.

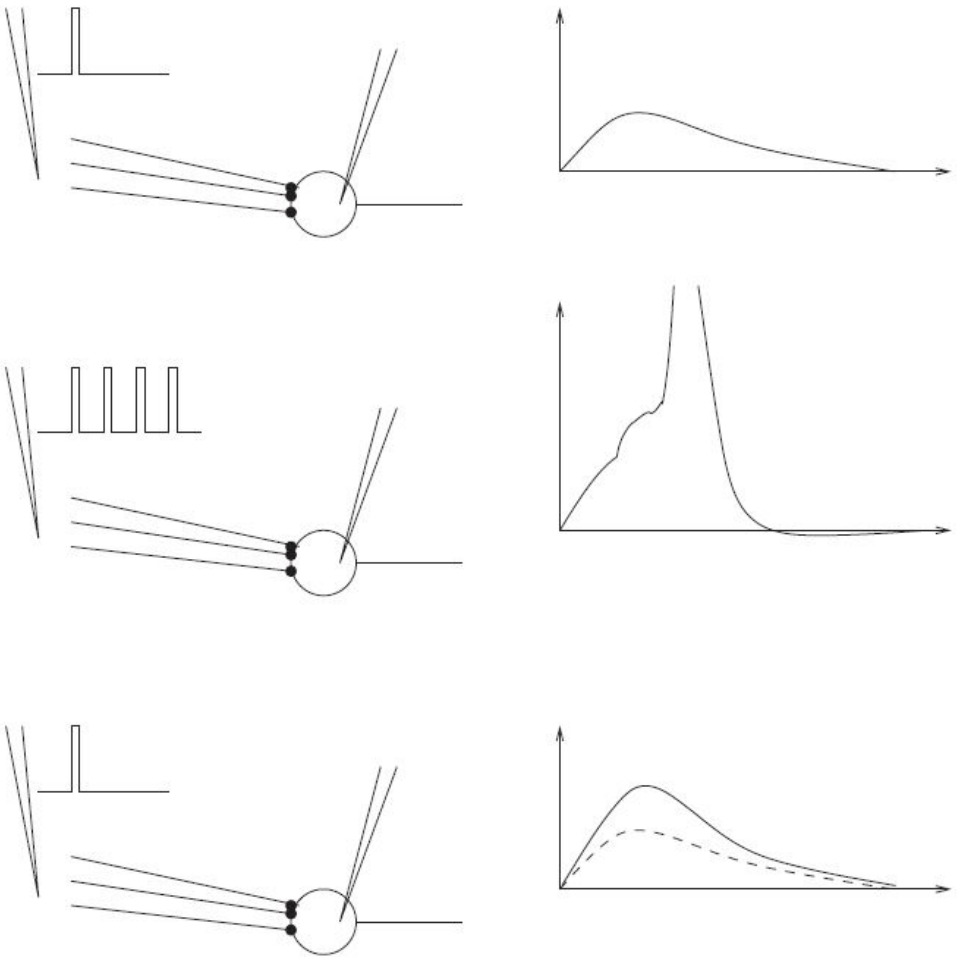


Figure 1.7: Schematic drawing of a paradigm of Long-term Potentiation induction[6](p.363)

The membrane potential of pre-synaptic neuron is stimulated by means of an extracellular electrode, while the post-synaptic one (the output) is measured with

another electrode. What can be deduced is that, after the post-synaptic spike, the strength of the synapse is increased, since the same input stimulation creates greater response in the post-synaptic membrane potential [6](p.363).

The formal theory of neural networks explains that the synapse weight w_{ij} (from neuron i to neuron j) is a parameter that can be set and adjusted in order to optimize the rate of successes of a network, given a particular task. The procedure that leads to adjust the parameters of a network is called *learning rule* and many of them have been proposed, depending on the type of network and on the goal to accomplish [2][3]. The class of learning rules that are base on the correlation between pre- and postsynaptic neurons, is referred as "Hebbian learning", which is inspired by the work of Donald Hebb [10].

The conventional Hebbian learning rule is a correlation-based learning rule that does not explicitly depend on the timings of spikes, while the Spike-Timing-Dependent Plasticity (STDP) is a demonstrated behaviour, which tells that the amount of change in the synapse depends on the relative timings of pre-synaptic and post-synaptic spikes [2](p.128): if the post-synaptic neuron spikes shortly after a pre-synaptic one, the synapse goes through a Long-term Potentiation (LTP), but the increase in the synaptic weight has amplitude that exponentially decreases in function of the difference between the two timing spikes. On the other hand, if the post-synaptic neuron fires before a pre-synaptic spike, the weight is decreased and the synapse experiences a Long-term Depression (LTD), which again is a function of the timing difference between the two neurons. All of that is depicted in Figure 1.8.

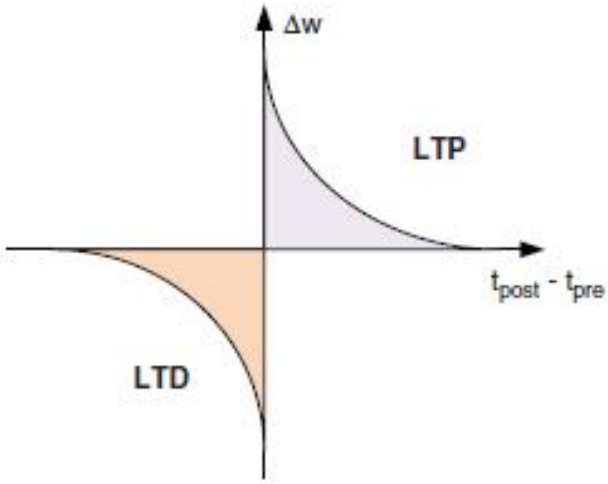


Figure 1.8: Illustration of a typical STDP protocol.[2](p.129)

In the following equations the mathematical approach is described:

$$\Delta w = \sum_n \sum_m K(t^m_{post} - t^n_{pre}) \tag{1.9}$$

$$K(x) = \begin{cases} A_+ \, exp(-x/\tau_+), \; x > 0 \\ A_- \, exp(x/\tau_-), \; x < 0 \end{cases} \tag{1.10}$$

In Eq.1.9, t^m_{post} and t^n_{pre} are the pre-synaptic and pre-synaptic spike timings respectively, while in Eq.1.10 τ_+ and τ_- are used to control the decay of the exponential and $K(x)$ is a kernel function. This is referred as a pair-based STDP rule [2](p.128).

There is another solution that differently from Eq.1.9 imposes a limit on the maximum weight, in which A_+ and A_- are a function of the weight itself. In this case the synapse weight is a function not only of the timing of the spikes, but of the synapses weights too, preventing it from growing without limit [2](p.129) It is important to underline that a STDP protocol determines how synaptic weights should change, based on timing, but how the learning is conducted vary depending on the implementation.

1.3.3 SNN implementations

In recent years, many studies have been conducted on possible hardware implementation of SNNs, since the needing of simulating complex networks. The simulations based on software, which are implemented in Von Neumann machines, hardly meet the biological spiking rate (milli seconds) constraint and furthermore they requires a huge quantity of power [1]. Thus, many application specific implementations have been developed and here some of them are briefly discussed.

"BrainScaleS" is a full custom analog design, specialized in simulating exponential integrate-and-fire neurons [11]. In analog implementations, transistor's sub-threshold range operations are exploited to create compact and high-speed processing neural simulators and they have the advantages of being characterized by extremely low area and energy consumption for very large-scale networks. Anyway, it is hard to program and to scale such architectures and they require long time in order to be designed and to be tuned. Furthermore, they can be utilized in those application where the topology and the task of the SNN are well defined [1].

Digital implementations, differently from analog ones, are less costly and more flexible and are based on general-purpose multiprocessors, Graphical Processing Units (GPU)s or FPGAs [1]. "TrueNorth" implementation[12] utilizes LIF neurons with high number of synapses without plasticity and "SpiNNaker" it's a multiprocessor-based simulator, which can support different SNN models, due to its programmable features, although it requires very high costs in terms of processing cores[1]: it consists of a chip multiprocessor (CMP) and a 128-MB off-die synchronous dynamic random-access memory (SDRAM)[2](p.186-187).

GPU-based architectures can exploit their parallel computation nature in order to provide a powerful implementation, though in complex SNNs, memory management and spike propagation represent an important obstacle for this solution. A popular GPU-based simulator is NEST (Neural Simulation Tool)[13] which is able to support many neural and synaptic models, but it lacks biophysical detail. In [14], NeoCortical Simulator 6 (NCS6) has been developed to take that issue into account. Futhermore, it supports LIF and Izhikevich models and it allows the user to design his/her own interface for other neural models[1].

FPGA-based SNN simulators, have been developed in several works [1]. In [15], a multiple FPGA-based architecture is proposed, where communication is performed by means of high speed serial links available in advanced FPGA boards. This architecture is able to simulate Izhikevich neurons with fixed pipeline stages, which makes it not suitable for supporting different SNN models. Furthermore, this architecture is designed for the simulation of simple and specific SNN models that do not take plasticity of synapses into consideration [1].

Another architecture is reported in [16], where a scalable-reconfigurable neuromorphic device based on an AER in a 2D mesh configuration has been developed. The authors claim that the architecture is capable of managing spike traffic using routing approaches in a single or multiple FPGAs [1]. It can simulate simple LIF model in order to perform the convolution operation used in image processing [17], but it does not involve plasticity.

Loihi is very interesting and recent architecture that has been developed by Intel's Microarchitecture Research Lab[18]. It is fabricated in a 14-nm process and the chip with a die size of 60 mm^2 contains 128 neuromorphic cores, where each core implements 1024 primitive spiking neural units cores. Then, there are x86 cores and in total, it includes 16MB of synaptic memory. Davies et al. claim that Loihi is able to support sparse network compression, core-to-coremulticast, variable synaptic formats, and population-based hierarchical connectivity [2](pp.191-192)[18]. The important feature of it, it's the on-chip learning capability through a microcode-based learning rule engine within each neuron core, which make it able to implement pairwise STDP and other more advanced learning rules [2](p.191).

All FPGA discussed above, trade off model flexibility and high speed processing, while GPU and general purpose processor approaches, should have the flexibility to implement several SNN models, with the capability of implementing fairly large-scale networks. All of these architectures rely on a general purpose Instruction Set Architecture (ISA) and on communication on chip strategy, in order to simulate the SNN. Evidently, in such implementations, there would be some general purpose functionalities that could lead to an unnecessary power and performance losses. The Spiking Neural Networks for Versatile Applications (SNAVA) architecture, which is the previous version of HEENS, is a scalable and programmable solution for real-time multi-model SNN simulation[1]. By means of several SNN models, the ISA of this

implementation has been fitted in order to exploit the proper quantity of hardware truly required for these kind of applications.

The software-hardware codesign, allow the user to design, configure and monitor the network. The hardware is composed by a parallel architecture, that is implemented on modern FPGA devices, in order to ease the programmability and so the simulation of different synapses and neurons topologies. It is composed by a Single Instruction Multiple Data (SIMD) array of Processing Elements, a single control unit and also communication units to support software to configure and monitor the system in real time with a 1 ms time step simulation. The ISA of the PEs has been customized in order to obtain high performance using minimum resources.

1.4 HEENS architecture

In this section, an overview of the Hardware Emulator of Evolvable Neural Systems architecture will be provide, in order to give a general idea of what kind of tasks the hardware is able to perform, considering that all the thesis work has been developed upon this already implemented structure. Then, in the last part of the introduction, the Address Event Representation over Synchronous Serial Ring Topology protocol and its hardware support will be described, since it is a fundamental part too that has been exploited and also modified to carry out all the targets set.

As previously mentioned, the HEENS architecture is a evolution of the previous Spiking Neural Networks for Versatile Applications (SNAVA) implementation[1] and it presents some upgrades respect to the predecessor: it is characterized by a better resource utilization, as reported in [5](p.46) and by an enhanced programmability and scalability capabilities. Indeed, it allows the user to decide the number of PEs in the 2D array and the number of virtual layers, setting before simulation or synthesis few parameters, like *row*, *column* and some others related to the number of layers. Furthermore, the user can totally set the topology of synapses, as will be described later, by means of specific text files that are going to be loaded in the Master Chip (MC) of the network. The HEENS implementation offers a hierarchical communication, that make it capable of synthesize up to 1352 neurons, a number that is more or less 6.76 times greater respect to the number supported by SNAVA[5](p.46)

One of the most interesting thing of this solution, is the capacity of supporting an online dynamic evolution and reconfiguration of the synapses interconnections, which leads to great savings in terms of design time, since there is no longer the needing of a new synthesis every time the configuration needs to be changed in some way.

In Figure 1.9, an example of HEENS network is reported. As it is depicted, there is a Master Chip (MC), that is in charge of communicating with the general purpose processor in order to receive all necessary initialization directives by the user, that will be utilized to configure and in some cases also to reconfigure the other nodes. Then,

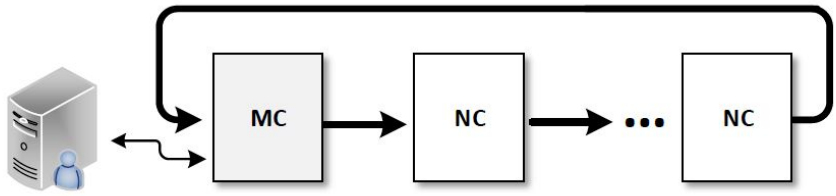


Figure 1.9: HEENS architecture, composed by a Master Chip and n Neuromorphic Chips connected in a ring topology[5](p.30)

during the execution phase, the MC behaves like a regular Neuromorphic Chip (NC), so it will compute the neural algorithm, in order to collect all the spikes from other nodes or from its own PEs, then to update the neural and synaptic parameters (like membrane potential) and finally to distribute post-synaptic spikes, when these latter are present.

1.4.1 Operational stages of HEENS

In Figure 1.10, the state diagram of the different processing phases of HEENS is depicted.

- *Initialization phase (IPh):* In this stage, the MC dynamically assigns the Chip Identifier (ChipId) relative to each node and the ring size to the rest of the network.
- *Configuration phase (CPh):* It is in charge of sending all fundamental data needed for the neural processing, which are synaptic and neural parameters and also the local and global connections mapping, as well as the execution program.
- *Execution phase (EPh):* This is the important stage in which the main biological functionalities of the soma are emulated. Each PE (artificial neuron), computes its state parameters, starting from its individual previous ones and from the input pre-synaptic spikes. This stage starts and finish by means of a control flag *eo_exec*.

  It is important to underline that monitoring operations need to be performed in this very stage: the serial communication bus used for the spike distribution will be exploited also for this task and so it is important to assure that monitoring distribution is completed before starting spikes transmission. This will be taken into account in this project, as it is explained in Chapter 2.
- *Distribution phase (DPh):* It emulates the propagation of spikes and neurotransmitter that happen in a biological neural network, by means of the serial

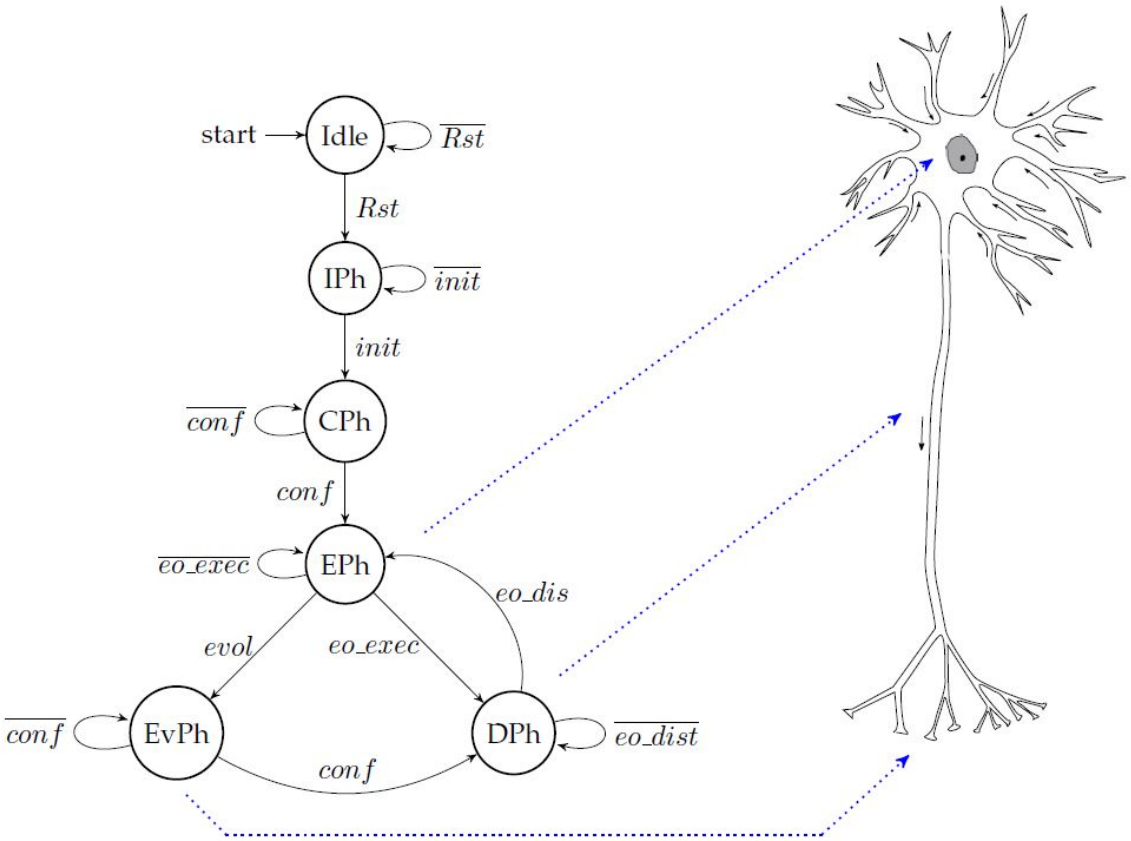


Figure 1.10: Operational stages of HEENS [5](p.31)

communication bus and of the AER-SRT protocol. All the spikes emitted in the execution phase are broadcasted to neurons that belong to the same chip or to another one.

- *Evolution phase (EPh):* This stage happens at the end of each EPh only in the specific case in which a evolution command has been received. In this eventuality, all synaptic connections and weights will be adjusted in each NC, depending on the information sent to the MC.

The basic unit of time utilized is the sum of the execution and distribution phases, after that IPh and CPh has finished.

1.4.2 Multiprocessor structure

In this section a brief discussion on the constituent modules of the architecture will be carried out. The reference architecture schematic of HEENS is reported in Figure 1.11.

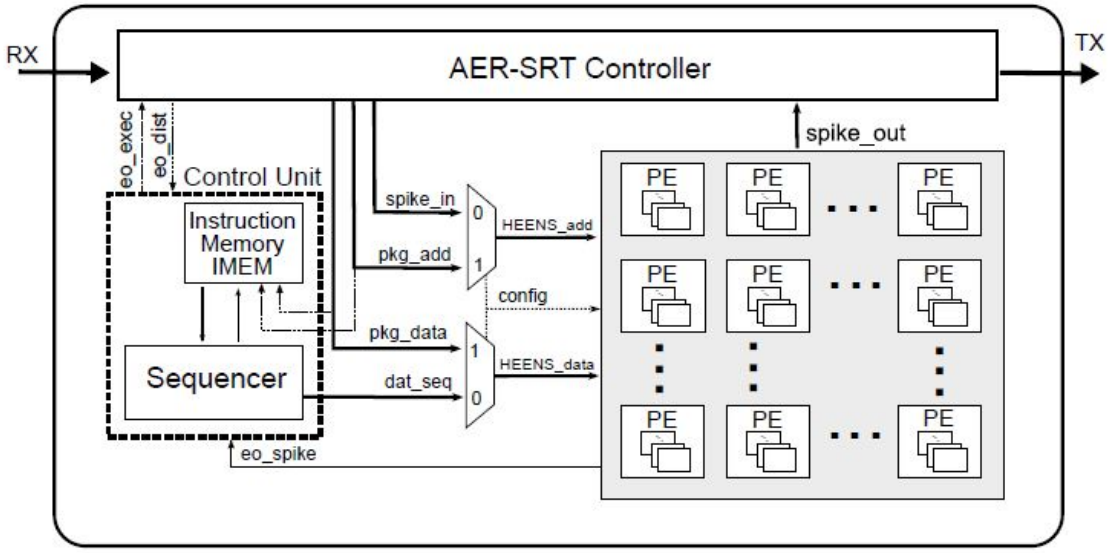


Figure 1.11: Block diagram of HEENS multiprocessor [5](p.33)

The figure shows that the main blocks of the architecture are: the communication buses, the Control Unit (CU), the PEs array and the AER-SRT controller.

- *Communication bus:* the first substantial thing is related to the exchange of information/data between the control unit, AER controller and the array of PEs. The two buses *HEENS_add* and *HEENS_data* are multiplexed between two different kind of addresses and data. The first is related to the configuration phase, when each memory inside the PE-array needs to be initialized and for this there is the needing of the address word *pkg_add*. Since this signal may come from the MC (by means of the RX side of the AER serial bus), it has some fields of bits related for example to the ChipId, to identify the chip in which the configuration data has to go. Other fields are related to the numbers of row, column and virtual layer to be selected in the array, in order to store a data in the local memory of a specific PE(neuron). Finally, depending on the type of configuration data, there are fields which address the right memory among all those present in a PE, as it will be described later. The *pkg_add* and *pkg_data* signals are also used to initialize the Instruction Memory (IMEM), as it is shown in Figure 1.11.

  The other kind of data, is the one related to the opcode of the instruction to be executed *data_seq* (from the sequencer), and the *spike_in* data from the AER-SRT controller, in Figure 1.12, the format of the address is described, which as all the rest of data, needs ID, virtualization, row and column fields. Anyway, these two kind of data are selected by means of the *config* signal, which is set in

the proper processing phase.

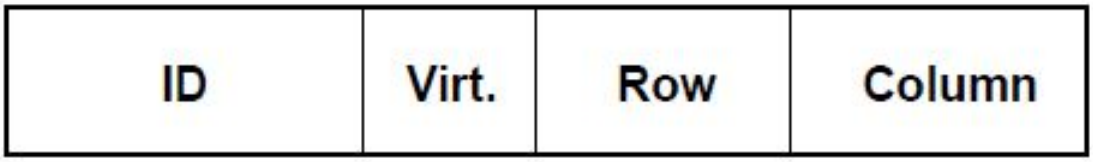


Figure 1.12: Address format for a spike event [5](p.34).

- *PE-array:* the processing units that make up the whole array represent a fundamental part, since they contain the data path for executing the neural and synaptic algorithms and, relatively speaking about this thesis work, for the collection and propagation of monitoring data. So, some details will be provided about the structure of the PE, of which schematic is portrayed in Figure 1.13. It is important to remind that this is a Single Instruction Multiple Data architecture, so it includes only one Control Unit, that is in charge of sending all commands relative to instructions coming from the IMEM

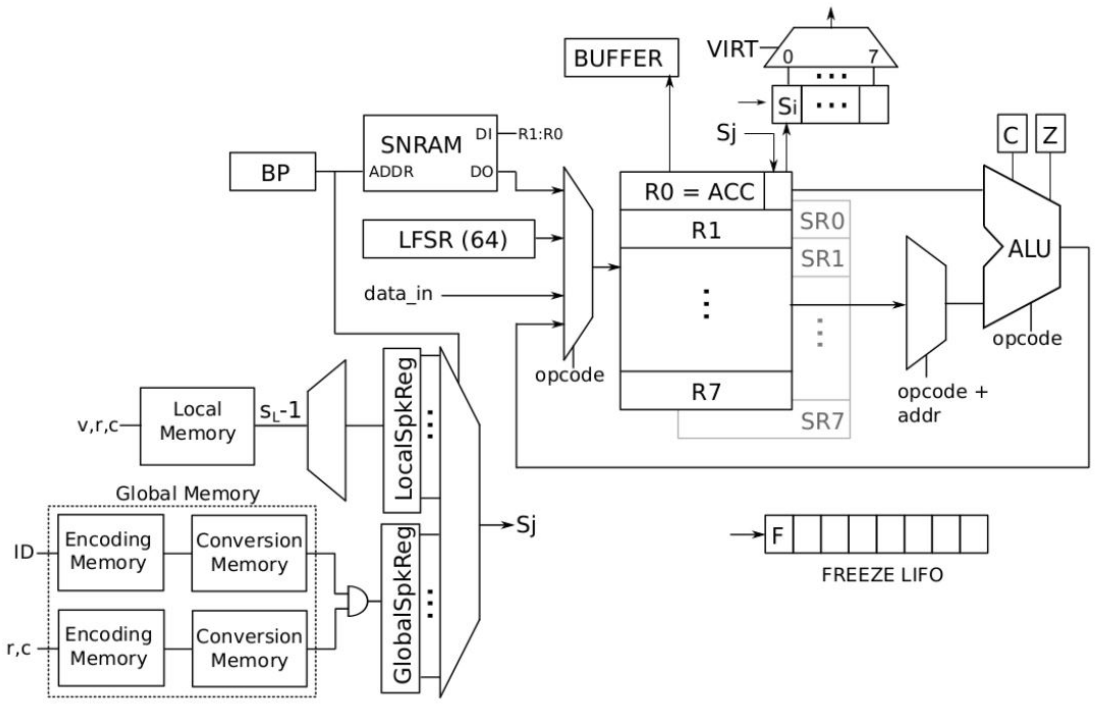


Figure 1.13: Processing Element [19]

– *Arithmetic Logic Unit (ALU)*

  This unit supports 16-bit fixed point arithmetic operations and logic ones too, which are identified by means of the instruction opcode (forwarded to the PE by the sequencer). There are two important bits coming from the ALU, that are the *carry* and *zero* flags, which are fundamental for conditional instruction (*FREEZEC, FREEZENC, FREEZEZ, FREEZENZ*): indeed, since this is a SIMD architecture, it is not possible to handle (at least for simplicity) all the support for the conditional and unconditional branches. What is done here, it's blocking the execution of specific instructions, in case those flags mentioned above are set.

  When a *FREEZE* condition is executed, all registers and ALU flags are disabled and a '1' is pushed in the Last In First Out (LIFO) register of the PE, in order to block all operations and to store the number of times the reactivation has to be performed in case of nested freeze conditions. Then the *UNFREEZE* unblocks everything to let the PE perform other operations.

– *Virtualization (VIRT)*

  Each PE performs a multiplexing operation during its execution phase in order to manage more than one neuron. Indeed, all computations involved in the neural algorithm are repeated for a number of time equal to the number of virtual layers. In 1.13 can be noticed from the output spike MUX, that the maximum number of virtual layers supported at the actual state it's 8. An explanatory image of Virtualization is reported in Figure 1.14.

– *Register File (RF)*

  This is a bank of 16-bit general purpose registers which are used to interact with the SNRAM and the ALU, in order to store the parameters relative to the neuron involved in computation and to provide the right operators to the ALU. There is also a bank of shadow registers, which access is controlled by the sequencer and that are used to enlarge the storage space of the PE. An important register that communicate also with output buffer register, is *R0*, which in this architecture is called "accumulator".

– *Synaptic/Neural Memory (SNRAM)*

  This memory on chip stores all synaptic and neural parameters, the seeds for Linear-Feedback Shift Register (LFSR). All of these data are those related to each virtual neuron of that specific PE.

– *Local and Global memories (BRAM)*

  This block of memory is used to decode the addresses that notify the presence or not of spikes, either from local PEs of the same node/chip (*local memory*), or from an external chip (*global memory*). The latter are processed only by the main virtual level ($VIRT = 0$, called *HUB* neuron), in order to support

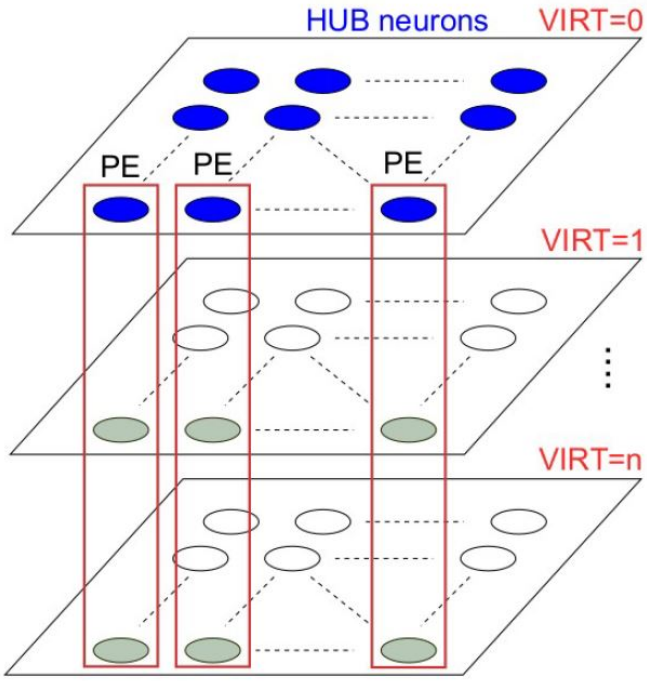


Figure 1.14: Virtualization of PE-array.[19]

a hierarchical communication between clusters. It is important to underline that a number of fixed synapses is assigned to each virtual layer by the user, as it will be described in Section 1.7, so the number of spikes contained in the local and global spike registers in Figure 1.13 are set.

The *local memory* has v, r and c (referring to *virtual, row and column* respectively) as inputs/addresses, then it has an output composed by $s_L - 1$ bits, where s_L is the number of maximum synaptic connections assigned to each PE. A specific bit of the output is set to one if the input correspond to a synaptic connection of that PE/postsynaptic neuron. The size of the memory is $2^{v+r+c} \cdot \log_2(s_L - 1)$.

The *global memory* has a block dedicated to the ID and one for the row and column addresses. This is due to the fact that neurons from different chips can have the same row and column positions. In this case there is also present an encoding scheme, in order to reduce the size of this whole block. The total bits size is equal to $(2^{r+c} + 2^{ID}) \cdot \log_2(s_G) + 2s_G(s_G - 1)$ [5](p.37).

– *Linear-Feedback Shift Register (LFSR)*

It is used to generate uncorrelated noise for each PE. The seeds composed

by 64 bits for this register, are set with the instruction *SEED* and they are stored in the SNRAM memory.

- *Freeze LIFO*
  As mentioned before, this is used for nested conditional instructions and it is linked to the *carry* and *zero* flags coming from the ALU.
- *Monitoring Buffer*
  This is directly linked to the *R0* register and it is used to store the information that the user wants to monitor as will be explained in next chapter. The data is moved from the accumulator to this buffer by means of the *STOREB* instruction and (even if it is not present in Figure 1.13) it can accept monitoring data from lower positioned PEs in the array.

- *Control Unit:* As it has already been explained, the HEENS is a SIMD architecture, which means it is provided with a single *control unit*. Furthermore, this is a Harvard architecture, that leads to a separate Instruction Memory, while the data memories are the SNRAMs inside each PE. The sequencer is the component in charge of sending the proper instruction address, by means of the Program Counter (PC), then of receiving the instruction, from which the opcode and all meaning values (like register number, pointer to the SNRAM memory, number of shift operation to perform, constant and so on..) will be extracted and sent to the PE-array, through the *data_seq* signal in Figure 1.11.

  In Appendix A, the Instruction Set Architecture (ISA) of HEENS is reported. It is possible to notice that it contains different kind of instruction, depending on the function and on the parameter it implements and transmits respectively, like move operations, arithmetic, logic, conditional, store, load and others. Then there are the macros, which identify a set of more instructions that the assembler is able to recognize.

  It is also important to underline that all the execution is divided in four pipeline stages: fetch, decode, execute and write-back. This is a very used technique in order to reduce the clock cycles required to execute an algorithm. With this approach, the whole architecture is able to complete an instruction each clock cycle, except for those that requires one or more cycles in addition, like jumps to subroutines, conditional ones, in some cases arithmetic operations and so on.

1.5 AER-SRT controller

In this section the Address Event Representation over Synchronous Serial Ring Topology controller will be illustrated, since it plays a key role for the work of this thesis. First of all, a brief discussion on the protocol operating mode will be carried on, then the hardware implementation details will be explained.

The AER protocol has become very popular in SNN multi-chip neuromorphic systems, since it is able to solve several problems that had arisen in such kind of implementations: the number of neurons and synapses that can be inserted in a single and specific silicon device is limited, while the use of multi-chip architecture leads to emulate large SNN models. However, this requires high efficient spike distribution capabilities and thus, the intra-chip communication becomes a critical point and it is responsible of scalability degree and efficiency of the whole system [4].

The AER protocol is characterized by the fact that all spike events are assigned to specific time slots, by means of a time multiplexing distribution. This leads to a resolution controlled by the widths of time slots in which the uncertainty can be reduced if these widths are made smaller. Such a solution overcomes problems related to collision of spike distribution events in asynchronous systems and so the information is preserved. Furthermore, the serial solution (AER-SRT), utilizes fewer wires, offers better performance, using point-to-point high-speed differential serial transreceiver at frequencies of Gbps. So, the latency introduced to allow this kind of communication in a pipeline fashion is well compensated by this high throughput and high speed.

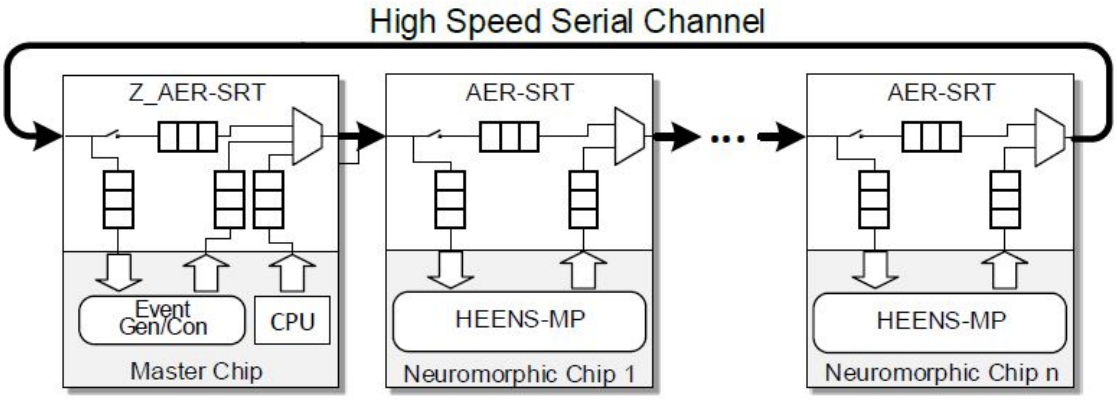


Figure 1.15: AER-SRT communication model [5](p.54)

In Figure 1.15, the structure of the AER-SRT implementation is depicted. The AER controller of the MC is the Z_AER_SRT, while the one related to the NC is the AER_SRT. The Xilinx-Aurora protocol is used to serialize and to deserialize all packets that travel around the ring.

The Master Chip has a key role of in the initialization, configuration and in the dynamic evolution phases of the newtork. In [1], the dynamic reconfiguration represented a problem, since the were no MC that coordinated these phases and so, all the nodes have to perform the evolution in real-time [5](p.54).

1.5.1 Control packets of AER-SRT protocol

The protocol utilized in this work manages two different kind of information, which are *data* and *control* packets. These are distinguished by the MSB of the 16-bit packet: '1' for control and '0' for data. Control packets are formed by a control information, to signal that a specific kind of control is being received, then by data and finally by another control sequence, in order to inform the receiving chip that the packet is finished. In Table

Contol packet	**Function**
IDLE	It keeps the ring active
INIT EOINIT	Initialization phase
CONF EOCONF	Configuration phase
EVOL	Evolution phase
SYNC START FINISH	Distribution phase

Table 1.1: Control packets of AER-SRT protocol

During the execution phase, and *IDLE* packet is transmitted by all chips in order to keep the link active. Now a brief discussion on these phases is made.

- *Initialization packet:* The MC is in charge of dynamically transmitting *ID* and *Ring Size* parameters to each node of the network in this phase. The former is used to signal which chip the information comes from and the latter is necessary for counters inside the AER controllers in each node, in order to perform synchronization tasks.

  The packet starts with a start control information and it is followed by 16-bits information that contains the *ID*: the ChipId of the MC is equal to one, so it will add '1' to this parameter and it will retransmit it to the next node. This arithmetic operation on the ChipId is performed by every following NC before it is retransmitted. After that, the *Ring Size* follows and finally there is the control information that signals the end of the *Initialization phase* (EOINIT).

- *Configuration packet:* Again, the packet starts and finishes with control information to signal the beginning and the end of this particular phase. The packet fulfills the task of network configuration. The following data sequences are composed by all the information utilized to initialize the four memories inside each PEs of each node (Synaptic/Neural Memory, local memories, conversion and codification

blocks) and for this reason, they are preceded by a ChipId and then by addresses to identify the chip and the specific memory to be written respectively, with the related locations too. It is possible also to signal that all data coming need to be written in each node of the network (common mode), by sending the ChipId of the MC.

The evolution packet is very similar, but it can be transmitted by the MC after each execution phase, in order to reconfigure the network (learning phase).

As for the *Initialization packet*, the MC understand that the configuration phase is over when it receives the EOCONF packet, after it has traveled for all nodes of the ring.

- *Distribution packet:* This phase is characterized by an initial SYNC packet, which is necessary to synchronize all the nodes. Indeed, a NC may finish the execution phase later respect to other nodes. So, each node at the end of the execution phase sends a SYNC packet and retransmits the same packet received from previous nodes. There is a counter inside the receiver block that allows each chip to count how many of them have been received and if the number is equal to the *Ring size*, the synchronization is over and the real distribution starts.

  The distribution packet is composed by a START sequence in which it is reported the ChipId from which the following spikes originate. Then all the spikes are sequentially transmitted and travel around the network. Once a NC receives its own spikes (comparing the received ChipId), these latter are discarded and the node sends a FINISH control information. The distribution finishes in the same manner the synchronization phase ends, but this time the FINISH control information is involved.

1.5.2 Master Chip

The structure and properties of the Master Chip will be described in this section. The Neuromorphic Chip has similar characteristic, but it has less functionalities, since it is involved in fewer control operations respect to the MC. The block diagram of the chip is illustrated in Figure 1.16.

- *CPU core:* This is in charge of loading all configuration parameters used for the neural algorithm that will be loaded in the memories of each chip. The user, as it will be explained in Section 1.7, needs to create and, by means of the *CPU*, to transmit all of these data to the MC, which will store them in the *CONFIG_FIFO*.

- *Spike Gen/Consum:* This is the part of the HEENS that receives spikes and applies the neural algorithm to them, as it was described in Section 1.4

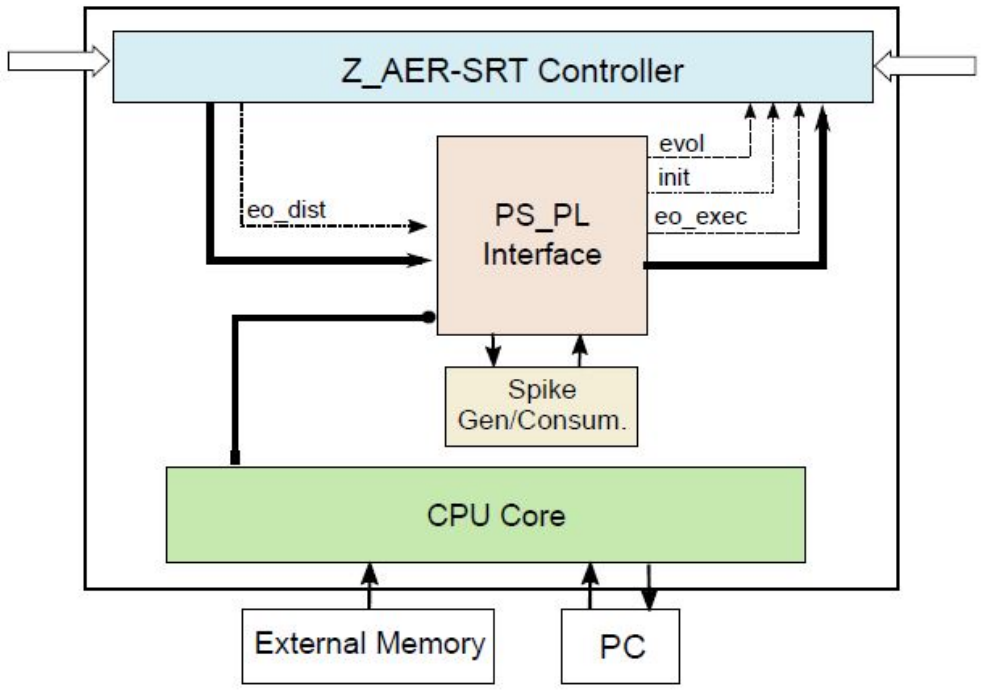


Figure 1.16: Master chip structure [5](p.59)

- *PS/PL interface:* Data coming from the CPU core travel with a different protocol respect to that used by AER-SRT. For this reason, an ARM processor (PS) it is utilized to convert those data in a way they can be transmitted to the serial bus. The other part of programmable logic (PL) it is an interface between the sequencer, PE-array and the AER controller.

- *Z_AER_SRT Controller:* This represents the core of the communication in this protocol. A schematic of it is reported in Figure 1.17.

  1. *Z_AER RX*

     This module has to read all data coming from the Aurora Rx side in order to detect control or data packets, in order to send them in the right FIFOs. It is in charge of setting flags that notify the end of initialization and configuration phases, by means of control packets *EOINIT* and *EOCONF* respectively. Furthermore, thorugh counters inside of it, it signals the end of synchronization and distribution phases, by counting the *SYNC* and *FINISH* packets.

     The last function is related to the spikes: in distribution phase, when it receives a packet, it needs to extract the ChipId and to compare it with its own. If the ChipId belongs to another chip, all the spikes are redirected to

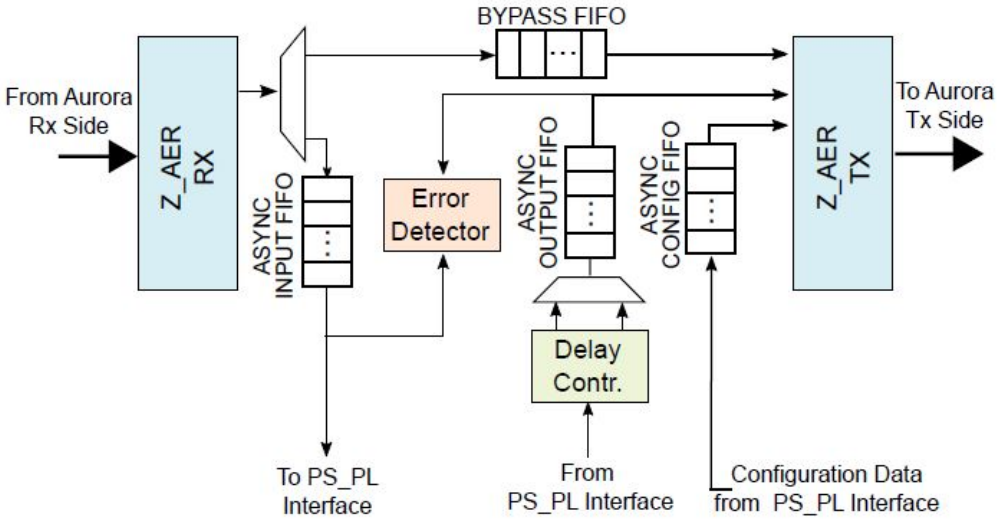


Figure 1.17: Z_AER_SRT controller [5](p.60)

the bypass FIFO, in order to let them keep traveling around the ring. If, on the other hand, the spikes are those it generates, the node can discard them and send a *FINISH* packet.

2. *Z_AER TX*

   The transmission module has been largely used in this thesis work. It is in charge of sending the right data/control sequences to the Aurora TX side. It is composed by a main FSM and by many other state machines utilized to coordinate the transmission of different packets. An explanatory schematic is depicted in Figure 1.18.

   The main controller determines in which phase the transmission is. For example, in the execution phase, the *IDLE* input of the MUX will be selected, or when the RX side will notify the ending of the synchronization phase, the input will be changed from the *SYNC* packet to the *START* one. Then, in the spike distribution phase, after the *START* packet, the output FIFO will be selected and its data will be transmitted as it happens for the configuration phase (although in that situation, the CONFIG FIFO would be involved).

   Anyway the selection of the MUX entries is determined by the main FSM and each data/control packet is managed by specific and dedicated smaller state machines.

3. *Error Detector and Delay Controller*

   Data from output FIFO are transmitted and also copied inside an error FIFO and they reach again the origin (after they have traveled around the

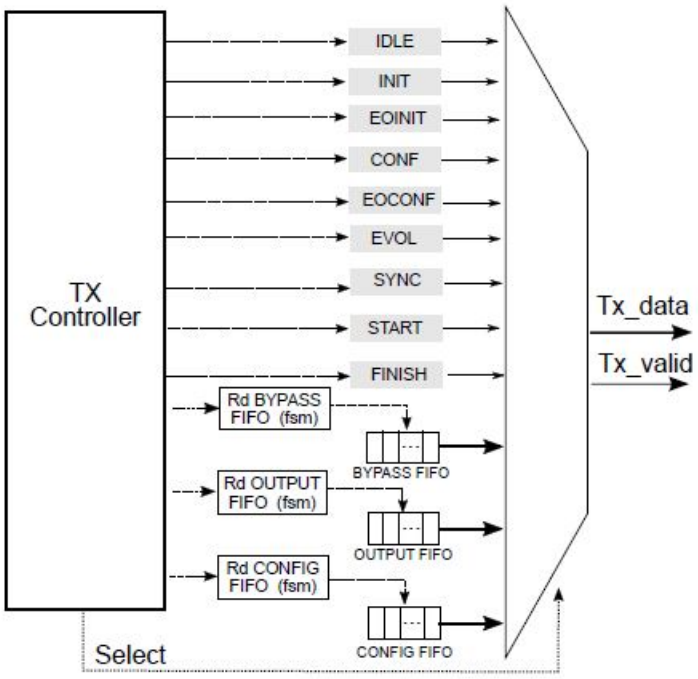


Figure 1.18: Z_AER_TX module [5](p.61)

whole ring) and the RX side signals that, the received data are compared to those previously stored in the error FIFO: if they differ, an error counter is incremented and it will be taken into account by the processor. At the current state of the project, the error cannot be corrected.

The axonal delay controller is in charge of assigning a certain delay to the spike before transmitting its address to the AER-SRT communication bus. Basically, it is composed by a RAM, which stores the delay for each spike and which is addressed by means of row, column and virtualization parameters of each spike. Then, before writing the spike parameters into the output FIFO, the delay is decremented at each clock cycle until it reaches a value of '0'. Only after that, the spikes information can be written into the output FIFO and finally be transmitted.

1.6 Neural algorithm

In Appendix B there is an example of the assembly code that will be used in simulation. In particular Appendix B.1 does not involve virtual layer in the whole execution, while Appendix B.2 does. The current algorithm performs a Leaky Integrate-and-Fire (LIF)

model to emulate the biological neuron and it is divided in four main sections:

1. In the first part all necessary declarations are made.

2. In the *.DATA* neural and synaptic parameters are defined. These includes the number of local synapses assigned to each virtual layer, with their related starting addresses in the SNRAM, in which it is possible to find the synaptic weight related to the synapses of that specific layer. There are specific addresses for global and neural parameters too. For example, in B.1, line 28, *SYN_ADDR0* refers to the address of the first synaptic weight of the main layer V0 and since there are three synapses in this case for each layer, the next pointer position of course will start three position ahead. After that, also the seeds addresses are set. Then there are several constant definitions, but some of them need an explanation since they represent the core of the computation.

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

Table 1.2: Fundamental values of membrane potential

In Table 1.2 are reported respectively: the resting potential, which is the potential to which the membrane decay tends if no spikes occurs, then the threshold potential after which the neuron fires a spike. V_{ACT} is the depolarization potential, while V_{ACT} is the activation level. Another important value that will be used is the Processing Element Identifier (PEID).

3. Then, the *.CODE* part starts. In this section the main core of the algorithm is performed: it is composed by few initial subroutines utilized to initialize some parameters in the array, the initial noise and then the main loop is executed. This latter, is in charge of computing the algorithm for each virtual layer: at the beginning, the neural parameters are loaded, then the membrane decay calculation is performed. This computation is reported in Eq.1.11.

$$V'_{memb} = (V_{memb} - V_{REST})\tau_{dec} + V_{REST} \tag{1.11}$$

In this equation, τ_{dec} is the decay parameter, which is less but close to 1 and it determines how fast the membrane potential reaches the resting value. Another

important equation is the following:

$$V''_{memb} = V'_{memb} + \sum_{k=0}^{n-1} s_k \cdot w_k \tag{1.12}$$

In Eq.1.12 the updating of the membrane potential is performed adding all the contributions coming from the pre-synaptic neurons: the parameter s_k can be either 0 or 1, in case there has been a pre-synaptic spike or not respectively. It is from this equation that the inner loop *LOOPV* comes from. Indeed, the algorithm analyze all the synaptic connections in each iteration of the virtual loop.

At the end of the inner loop, if the spike is detected (through the subroutine *DETECT_SPIKE*), the bit relative to the spike of that specific virtual layer is stored into the Less Significant Bit (LSB) of the accumulator, in order to go then in the output spike buffer that is shown in Figure 1.13. Finally, the values of the current virtual layer are stored with the subroutine *STORE_NEURON* and the next layer is computed.

4. When the main loop ends its execution, the final part begins, in which the spike distribution it's performed. After that, all the computation starts again.

The assembly codes of Appendix B contain a part related to the global spike detection that is commented. Indeed, the true simulation with more than one boards has not been yet implemented for the present architecture version: in this work, only a MC node will be present and it will communicate with itself in a ring topology communication. So, basically, all information will run from the output to the input of the same chip in a loop mode, as it will be described better in next chapters. A first simulation example of the LIF model is reported in Figure 1.19.

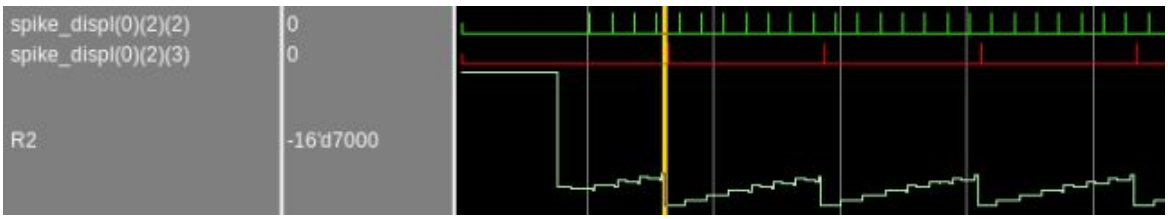


Figure 1.19: Leaky Integrate-and-Fire (LIF) model simulation

In the simulation the register *R2* (where the membrane potential is stored) of a specific neuron is reported: it is highlighted how the neuron fires a spike each time the threshold is reached. After that the membrane potential is reset to V_{REST} value (as shown by the cursor in Figure 1.19) and furthermore each time it doesn't receives an input spike, the value of the potential slowly decreases toward the resting potential, as described in Eq.1.11.

1.7 Design flow

In this section the basic steps by means of which all the simulations have been performed are briefly explained. In order to set up all the support necessary to run the neural algorithm described in Section 1.6, it is important to build all the Memory Initialization Files and to generate the machine code utilized by the CU. The first thing to do, it's choosing the algorithm ASM file to run, like the LIF one of previous section. Then, the user needs to define the netlist of synaptic connections that will be used. In Appendix C three possible configurations are reported. Also it is important to underline that in particular files (*neuron.csv*) it is possible to set the initial values for membrane potentials (in order to make a neuron to spike immediatly, to start the simulation), in which also the correct addresses of the SNRAM need to be set (like in the ASM file of Appendix B).

In Figure 1.20 the operating principle of the netlist file reported in Appendix C.1 is graphically explained, by means of a 4x4 PE-array configuration: there are three important columns related to the numbers of *row*, *column* and *virtual layer* of a presynaptic neuron and at the same row position on the right, it is possible to set the three same parameters of the postsynaptic neuron. So, basically these lines are used to establish the connections between different neurons. Other important parameters are the specific synapses chosen for that particular link (labeled as ph) and the synaptic weight, which can be either positive (excitatory) or negative (inhibitory). In the figure is also described how much the membrane potential needs to be incremented in order to reach the threshold. The behaviour of this configuration proposed is shown in the simulation result reported in Figure 1.21.

In Appendix C.2 and C.3 other two examples that will be used are reported. In the former, an oscillator has been implemented, which involves all PEs/neurons of the 4x4 array: each PE of a row causes the next one (of the same row) to fire and so on until the last neuron of a row has been reached. At that point, the first PE of the next row is lead to fire and the chain continues in this way, until it starts again from the first row and column positions. The example in Appendix C.3 is similar, but this time one neuron in a certain position and that belong to a specific virtual layer excites the neuron in the same array position of the next virtual layer, in order to let it fire.

Therefore, by means of *bash* and *Python* scripts developed by the research group, the *.asm* file is used to create the *.mif* file for the IMEM, while the *neuron.csv* and *netlist.lst* files are converted in *.mif* ones for the local memories (to decode addresses of synapses) and for the SNRAMs (synaptic and neural parameters) of each PE. The *.mif* files contain the memory initial data for the simulation.

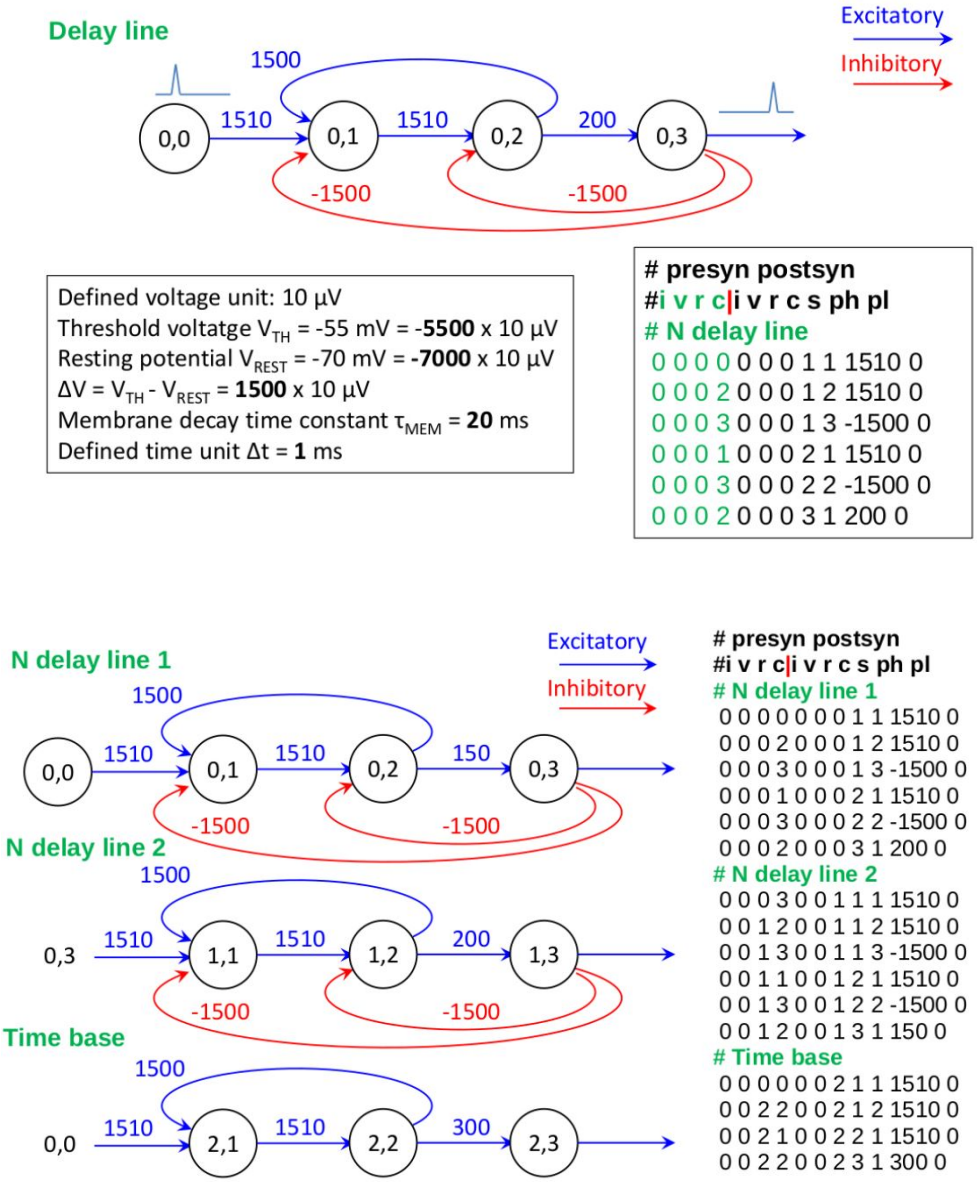


Figure 1.20: Delay line example [19]

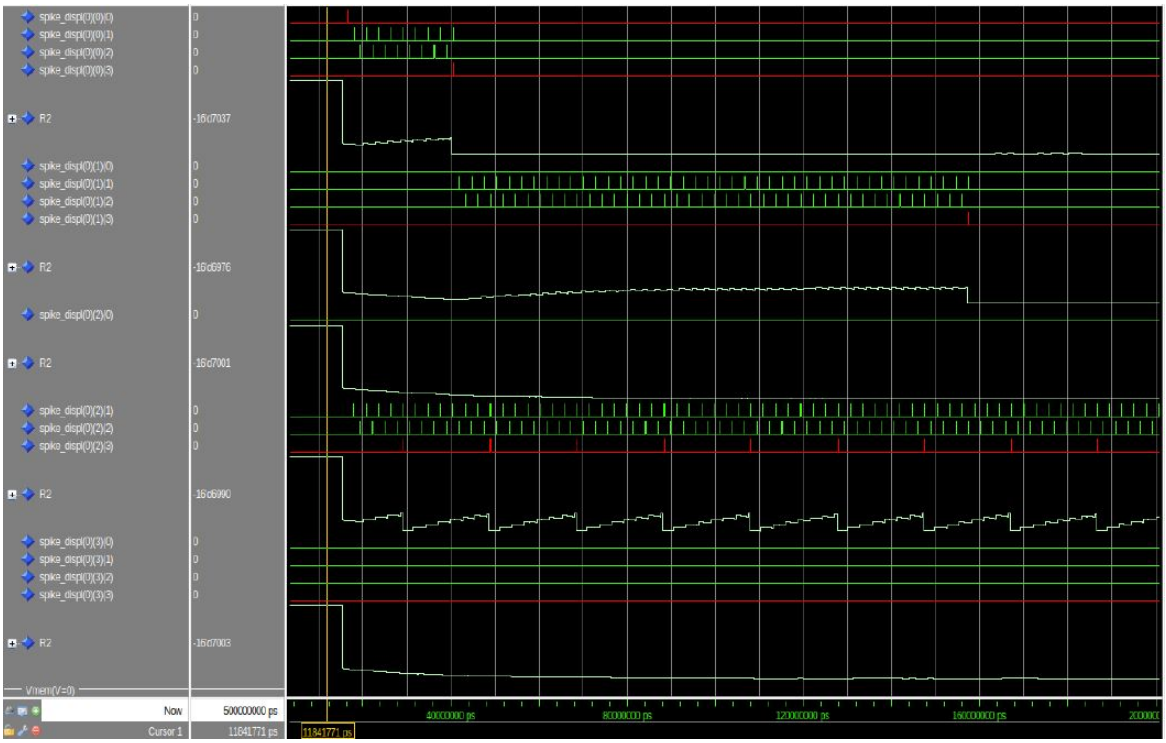


Figure 1.21: Delay line simulation

Chapter 2

Monitoring implementation

In this chapter all hardware support for the monitoring system that has been implemented is reported and explained. First of all, the changes brought to the Control Unit will be described, then a section will be dedicated to the PE-array that is in charge of loading all information required and of propagating them until they reach the transmission modules. Then, specific sections will be devoted to the transmission parts of both single and multi-board versions of the architecture. All changes made to the sequencer are discussed in the AER section, since they are related to this latter.

Finally, the logic synthesis and implementation works and results will be covered and analyzed.

2.1 Software and Algorithm

As it was described in Chapter 1, the ISA of HEENS architecture is reported in Appendix A. Some of these instructions have been exploited and particularly an instruction *STOREB* (coming from the previous SNAVA implementation [1]) had already been created before this work started: this instruction basically tells each PE to perform a special movement, from the accumulator(*R0*) to the output monitoring buffer, which is illustrated in Figure 1.13.

Indeed, the aim of this thesis work is to collect and to transmit a specific information required by the user and this data, related to a particular neuron, is usually stored in the register bank while the neuron is being processed during a loop iteration. So, the first useful action is to move this needed information from a specific register to the accumulator and then from the accumulator to the monitoring output buffer. For this reason, a macro called *MONIT* has been created: as it is shown in Appendix A, this macro must be followed by a register number (that the user wants to monitor) and then it will be decomposed by the assembler compiler in two instructions. The first is a movement operation from the interested register to the accumulator and the second

is the already mentioned above *STOREB* instruction. The research group created an ad-hoc compiler written in Python and a little part of it was modified in this work, in order to accomplish the goal of handling this new macro.

It is also possible to load a value from the internal SNRAM of each PE to the accumulator and then to transfer it up to the monitoring buffer. The procedure in this case it is quite different: first of all, there is a register in the PE that is used as a pointer to the SNRAM (as it is shown in Figure 1.13), so, it is necessary to load in this register the address of the value the user wants to monitor and for this purpose the instruction *LOADBP* can be used, as reported in Appendix A. The addresses values can be taken directly from the sequencer, in which some important constants are stored in order to ease the extraction of recurrently used memory pointers.

Then, *LOADSN* can be exploited to load the pointed location of the SNRAM into the accumulator. As it is described in the Appendix A, this instruction load two values at the same time to *R0*(accumulator) and to *R1*. Indeed, in the subroutine *LOAD_NEURON* of the algorithm (Appendix B.1), *LOADSN* is used to load at the same time neural parameters to *R1* and to the accumulator.

In Appendix B.3, the algorithm used to test the monitoring instruction is reported. As it is shown, a specific address is loaded in the SNRAM pointer, which is the one of the Processing Element Identifier (PEID) data: this will be used for debugging purpose, in order to verify that each PE sends the correct information and to check how it travels through the path that will lead it to the transmission modules. Furthermore, this information is moved from the accumulator to *R3* (by means of a *MOVR* instruction), in order to launch a proper monitoring instruction next, that will involve the register *R3* itself.

It is necessary to underline that, in this case, the monitoring is launched at the end of each virtual neuron loop, in order to not corrupt values of important registers (like *R0*) before the neural algorithm starts. Another way could be to place the monitoring before the core of the algorithm starts.

2.2 PE-array

The strategy applied to propagate the monitoring information is very similar to the one applied for the spike distribution. As it was mentioned in Section 1.4.1, the monitoring operations (propagation through the array, transmission) need to be performed in parallel with the normal execution, so after the *STOREB* instruction is executed, all the information extracted from each PE will be propagated and then transmitted, while the sequencer (and the rest of the array) will continue to perform its normal operations.

2.2.1 Hardware structure

In Figure 2.1, a 4x4 configuration is reported, in order to explain the basic principles of the monitoring propagation through the array: this task is performed by means of pipeline (like spike distribution), which means that only one row will be loaded to the final transmitter module at the top of the array in each clock cycle. In the figure the yellow rectangles represent in a simple way pipeline registers that load and propagate monitoring data.

This choice derives from the fact that the array, at the current implementation, can reach a 13x13 number of PEs and the architecture supports up to 16x16 cores, so loading all monitoring information in one clock cycle would lead to problems for the final hardware implementation, due to routing issues. Furthermore, the final transmission part (AER modules) needs to send one 16-bit data at time and therefore, while a specific row is being transmitted to the serial bus, the other rows can climb the array and finally be loaded for the final transmission.

In order to accomplish this task, a specific FSM in the top module of the PE-array has been created, in order to handle the propagation of monitoring data through the pipeline registers of the array. In this stage, each row of data, composed by $[size\_x] \times [16 - bit]$ monitoring information, is loaded inside a monitoring FIFO (as it is described in Section 2.4): it means that the FSM has to stop the propagation of the other rows until all monitoring data of the top row are loaded inside the FIFO. This strategy has the disadvantage of taking exactly NxN clock cycles before all data are loaded inside the monitoring FIFO: if another *STOREB* is executed immediately after a first one, the sequencer needs to stop the execution of the algorithm, since the new instruction would overwrite the previous monitoring data before they reach the final AER module. Furthermore, this strategy does not fully exploit the greater operative frequency of the AER part, which is double the PE-array clock frequency.

Anyway, implemented in this way, the architecture is a good first attempt, since it has an easy structure and does not require too much hardware and control logic. In order to improve performances, an upgraded version will be discussed in Chapter 3.

The schematic of the monitoring controller in the top module of the PE-array is reported in Figure 2.2.

Some signals of the figure will be explained in next sections. For now, the important flag are the *block_monit* signal, which tells the PE-array controller to stop the propagation of monitoring data, and the *next_row* flag (*en_monit_x* for each PE), which is an output from the controller.

In Figure 2.3 the main parts that are involved in monitoring operations are illustrated as example. In this case, two PEs that belong to the same column but to consecutive rows are reported: each PE receives from the sequencer the *Opcode* field of the instruction (together with other data that are not present in the figure), from which it generates a flag that is used as enable signal for the monitoring buffer. In this phase

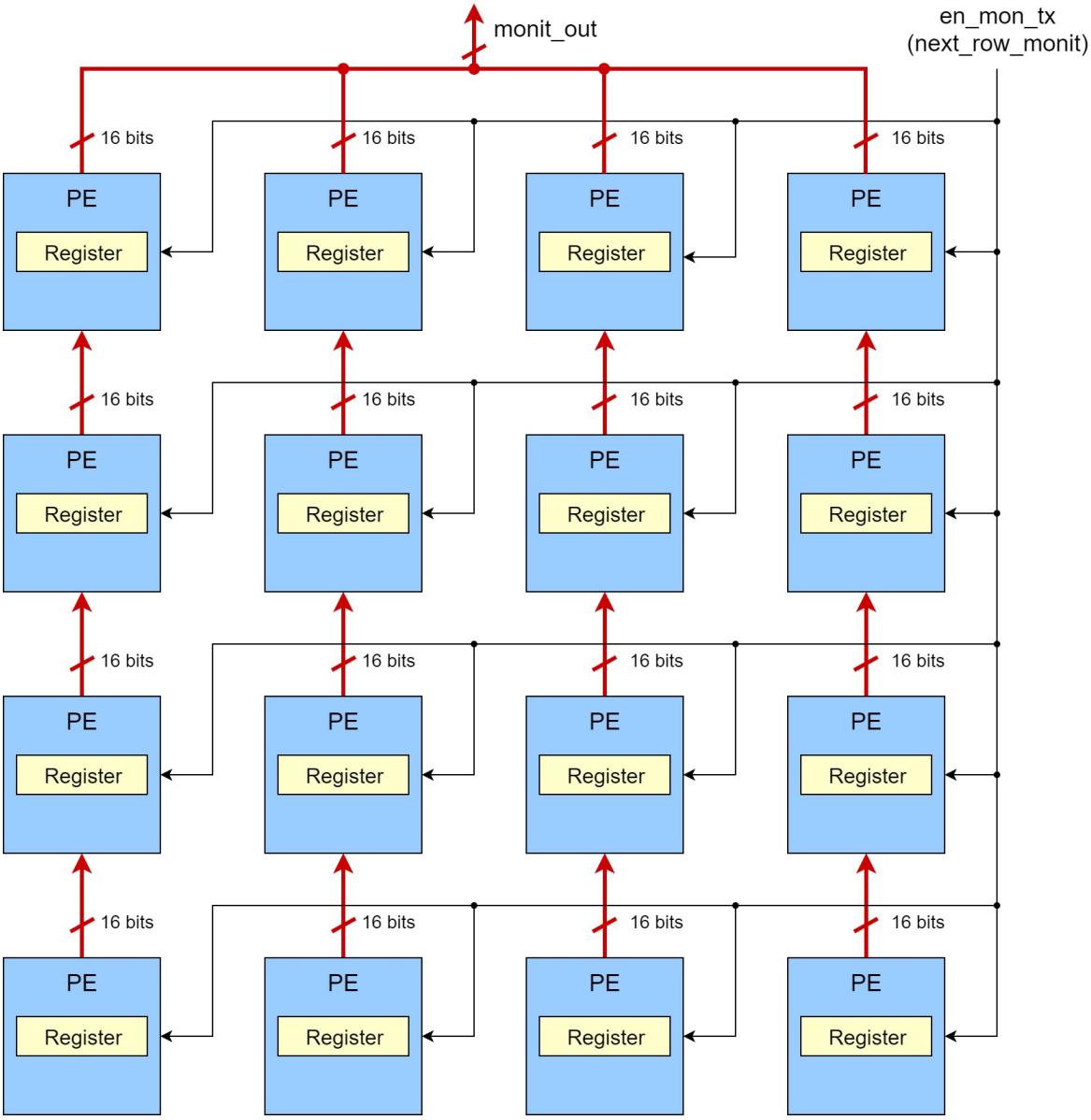


Figure 2.1: Four by four PE-array configuration

the *en_monit_tx* is set to '0', as it will be soon described, so the input of the buffer is taken from the accumulator. Then, when *en_monit_tx* goes to '1', input from the lower PE is loaded and the enable is still high (due to the or logic gate), which means that the propagation is being carried on. A Timing Diagram (TD), that shows all significant phases of the monitoring controller is reported in Figure 2.4.

The signals involved in this diagram are all synchronous, which means that they

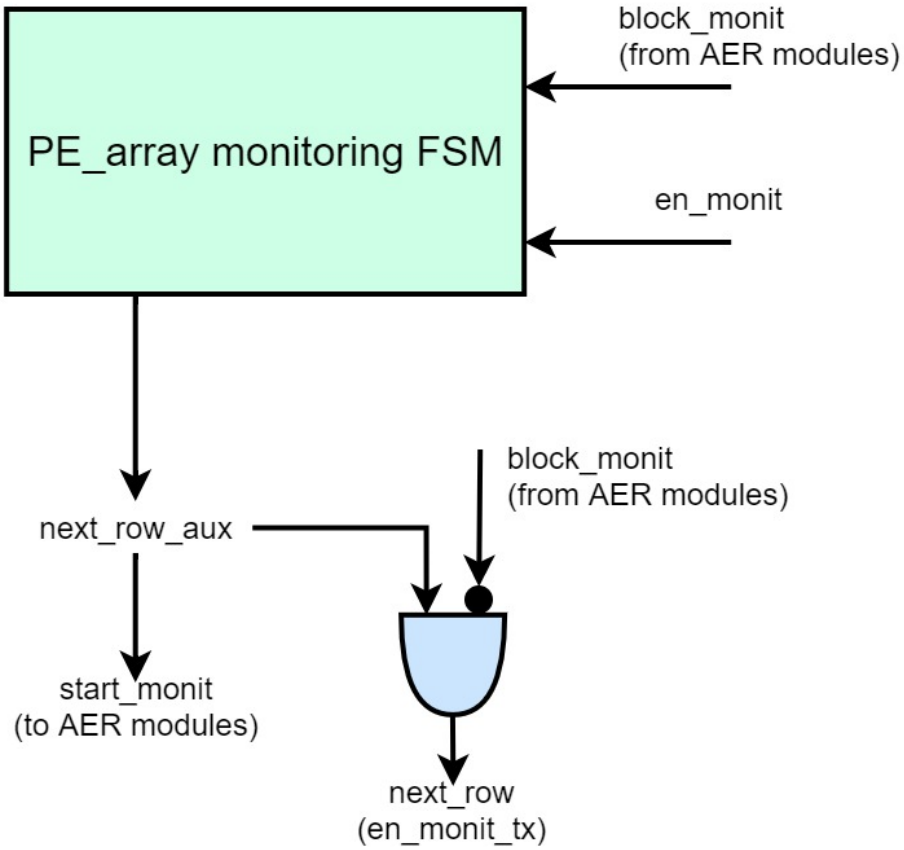


Figure 2.2: Monitoring controller of the PE-array

come from a previous sequential element and maybe some logic, so they change with a variable delay after the rising clock edge. In the diagram this delay is a little fixed time interval for simplicity.

The signal *Pe_count_monit* is a counter, while *monit_out* represents the top row of monitoring data, so the output of the PE-array that will be loaded inside the monitoring FIFO of the AER module. There are some interesting occurrences that is worth mentioning:

1. At the beginning, a reset signal is set and the FSM goes to its *idle* state, in order to initialize some values, like for example the counter to the actual number of rows of the array. Then the state machine remains in that state until *en_monit* is set to '1'. This latter signal comes directly from the sequencer as it will be explained later and it is set in the execution stage of the *STOREB* instruction.

2. Then comes the *start* state: the signal *next_row_monit_aux* goes to '1' and, since the *block_monit* is still not set by the AER module, the *next_row_monit*

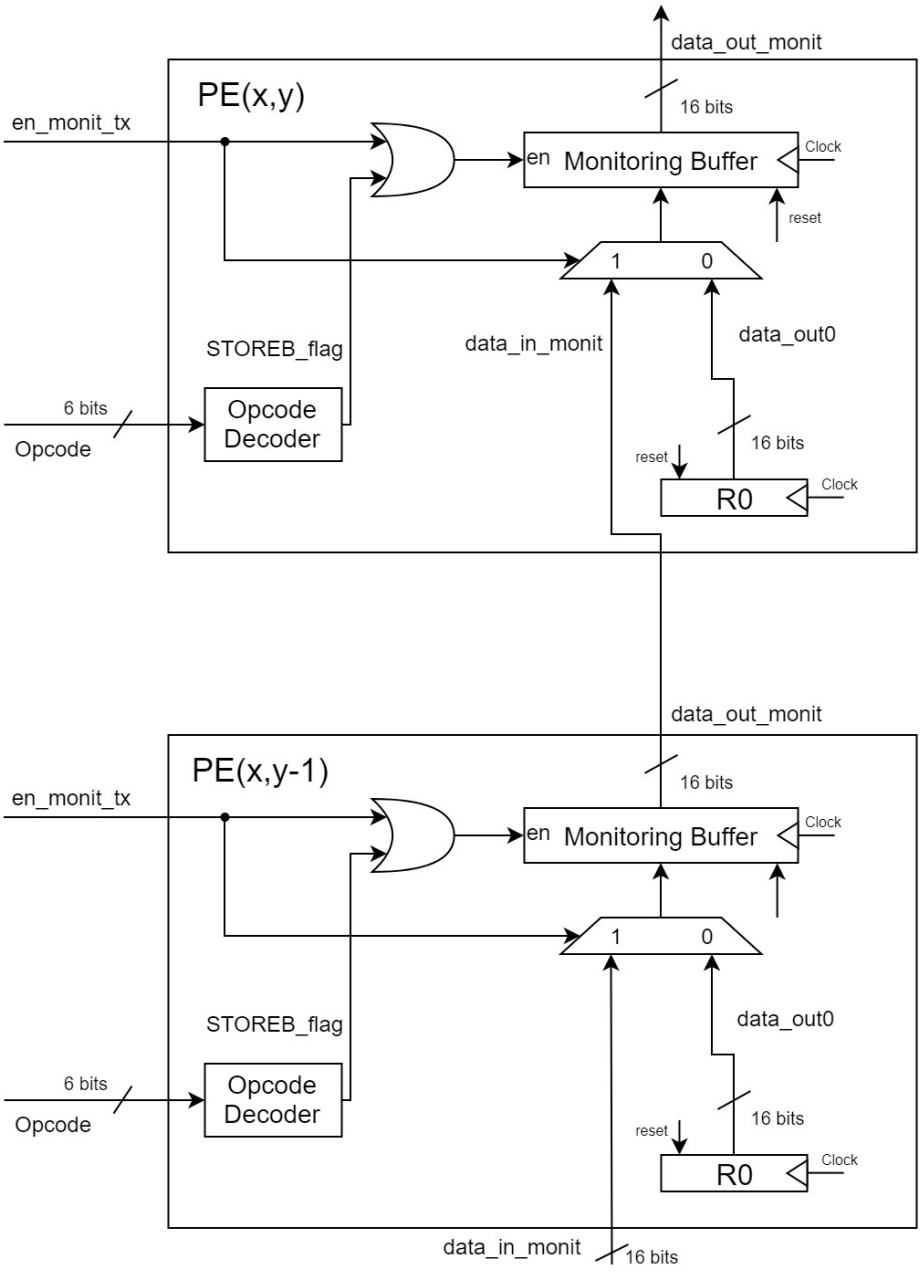


Figure 2.3: Structure dedicated to monitoring operations in the PE

can be set '1'as well. This latter, is the final *en_monit_tx* that allows each PE to load the input monitoring value from the lower PE. So, for example at the beginning the top output row (*monit_out*, Figure 2.1) samples the first row of the array $(y-1)$.

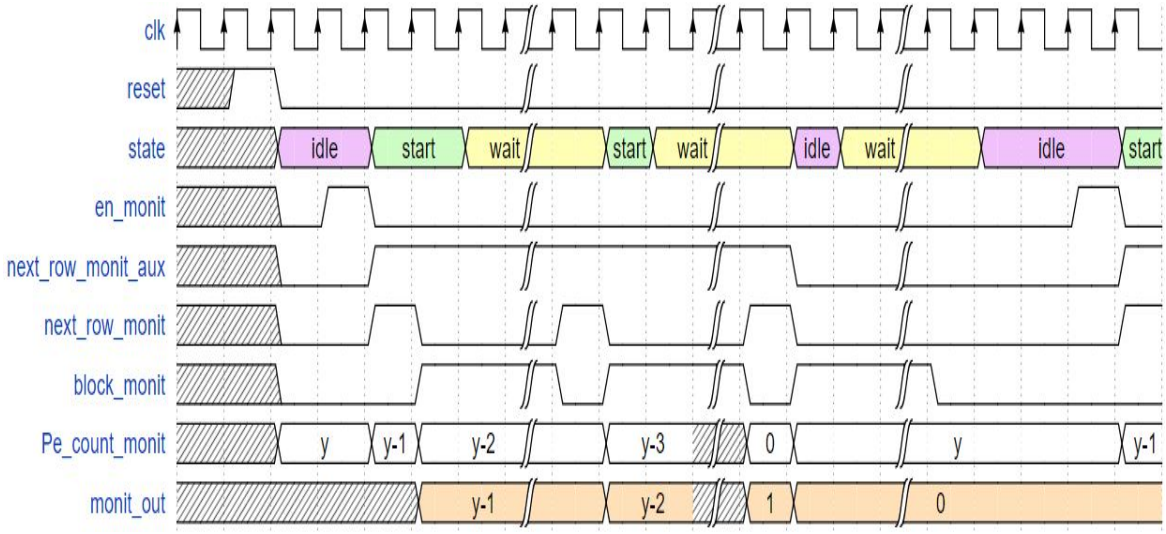


Figure 2.4: Timing Diagram of the PE-array monitoring controller

3. After that, *block_monit* goes to '1' and the FSM proceeds to the yellow coloured state *wait*, which is needed to stop the propagation of the other rows of monitoring data, in order to allow loading of any data belonging to the current output line into the AER FIFO. It is important to say that this state machine presents a *Mealy* behaviour: the output *next_row_monit* is in AND with $\overline{block\_monit}$, since if a new row is loaded at the output of the PE-array, the propagation of the other rows need to be stopped immediately. Therefore, waiting for another state would not respect the correct timing of the algorithm.

   Anyway all signals are synchronous and no chains of *Mealy* FSM are present, therefore no timing problems are generated by the synthesis and implementation tools (Section 2.5).

4. After all data of a row have been loaded inside the monitoring FIFO, the AER module sets *block_monit* to '0' and the propagation can continue.

5. In the blue coloured *wait* state, it is implicitly shown that all rows of data have climbed the array and have been loaded inside the AER storage component. Finally, when the last row needs to be loaded ($Pe\_count\_monit = 0$), the FSM moves to the *idle* state again, in which the counter is reset and signals used to propagate the array are set to '0'. Anyway, the last row still needs to be loaded, so the AER module sends a block signal, in order to prevent the rest of architecture from starting another *STOREB*.

   Finally, if another monitoring instruction is waiting, a specific combination of different flags coming from the AER modules, will unlock the sequencer, which

in turn will set *en_monit* to '1' again. These control actions are shown in next sections. In Figure 2.5 is reported the state diagram of the monitoring FSM just discussed.

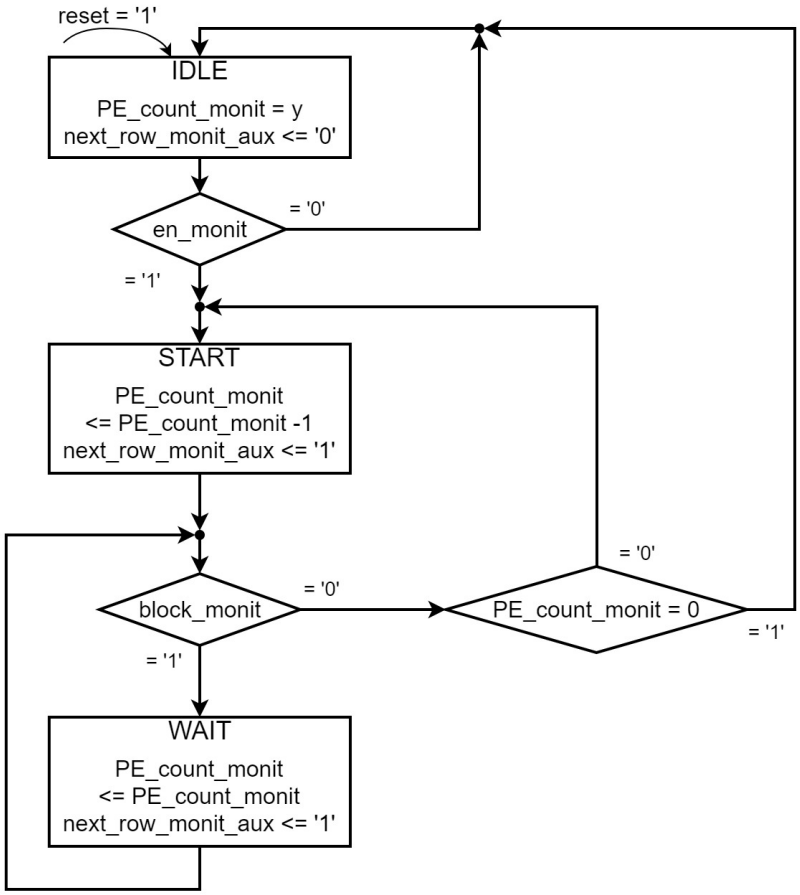


Figure 2.5: Flowchart of the monitoring controller in the PE-array unit

2.2.2 VHDL and Simulation

In Figure 2.6 it is illustrated the structure hierarchy that is used to built the PE-array architecture by means of the Very High Speed Integrated Circuits Hardware Description Language (VHDL).

The source files of the architecture can be found in Appendix D.1, D.2 and D.3. In these sections, not all the lines of the VHDL files are reported, for simplicity and furthermore, it seemed more appropriate to show only the parts that are directly involved in the propagation of the monitoring data.

It is necessary to point out that at the beginning of each source file, two specific

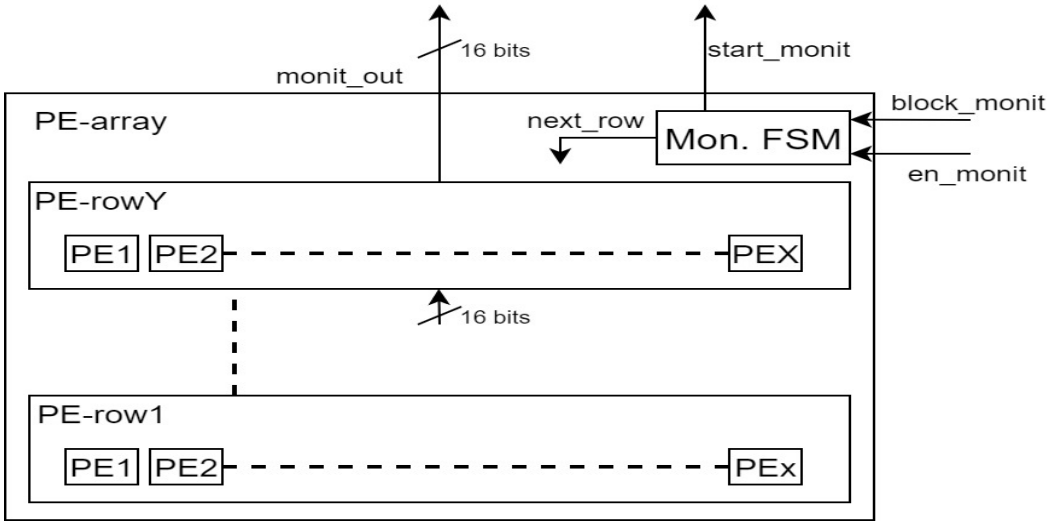


Figure 2.6: VHDL structure of PE-array architecture

packages are loaded: *SNN_pkg* and *log_pkg*, in which there are present important constants/ parameters declarations and definitions and useful functions utilized in all the architecture. For example, in this architecture, the state of the main FSM machine of the sequencer, are referenced by names defined in the *SNN_pkg* (like *STOREB*) and, in this way, if the opcode field of an instruction changes, only the package file needs to be modified.

In Figure 2.7 a simulation performed on a 5x5 array is reported, for which the algorithm of Appendix B.2 with the monitoring instructions shown in B.3 has been launched. As it is shown, the *block_monit* signal for five clock cycles, in order to let the AER module load all monitoring data inside its FIFO and this is done five times, since in this case there are five rows. Since in the algorithm of Appendix B.3 there are two *STOREB* instruction separated by one clock cycle (the *MOVA* instruction of *MONIT* macro), the next rising edge of *en_monit* need to wait NxN clock cycles plus exactly four mores after the first one. The cause of this overhead is better explained in the next section, where the sequencer structure is described.

From the simulation, it is also possible to notice how all PEIDs (the information required by the assembly code) are loaded at the output of the array (*monit_out*). The clock frequency of the HEENS architecture is set to 125 MHz.

2.3 Multi-Board version

In this section the multi-board version will be explored in order to introduce and explain how the transmission of collected monitoring data through the AER-SRT modules

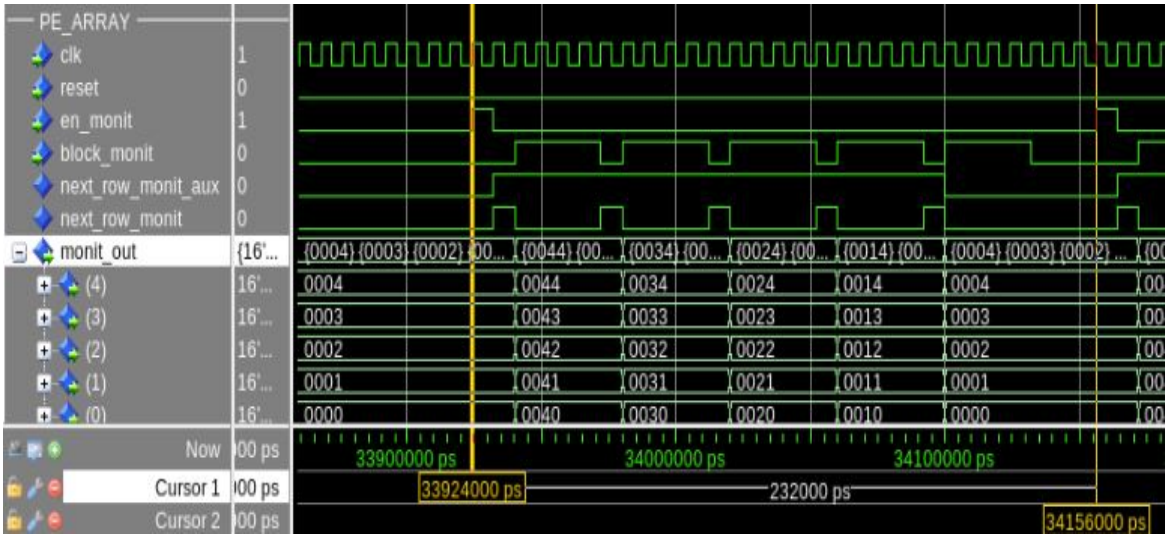


Figure 2.7: Monitoring data propagation through a five by five PE-array

and serial communication bus is handled. A first picture of the VHDL structure is provided in Figure 2.8.

In this graphic representation, it is reported the *SNN_OneBoardTop*, which is the single-board version of the AER part that is described in Section 2.4. Then, for what concerns the multi-board version, there is the top module, *ZynqKintexTop*, which contains both the Master Chip (*Zynq_top* and the regular Neuromorphic Chip (*HEENS_top*). As already mentioned in the previous chapter, in this work only the MC will be eploited, since the regular node has not already been completed by the research group.

The AER-SRT implementation has already been described in Section 1.5 and, from now on, all the changes that have been made on this architecture in order to support monitoring are described, included the sequencer since this part is strictly related to the reception and transmission operations. A simplified structure of the VHDL hierarchy is reported in Figure 2.9: here, it is underlined the clock frequency of the AER part, which is 250 MHz, while the clock of the HEENS part is set to 125 MHz.

2.3.1 Z_AER_interface

This is the PL (programmable logic) part of the PS-PL interface of the whole architecture. As already described in previous chapter, it communicates directly with the CU and the PE-array, in order to synchronize all the operations performed and controlled by the sequencer and transmission parts.

Basically, the interface receives as input all the monitoring data coming from a row of PEs and then it writes these values inside the the monitoring FIFO. This latter is

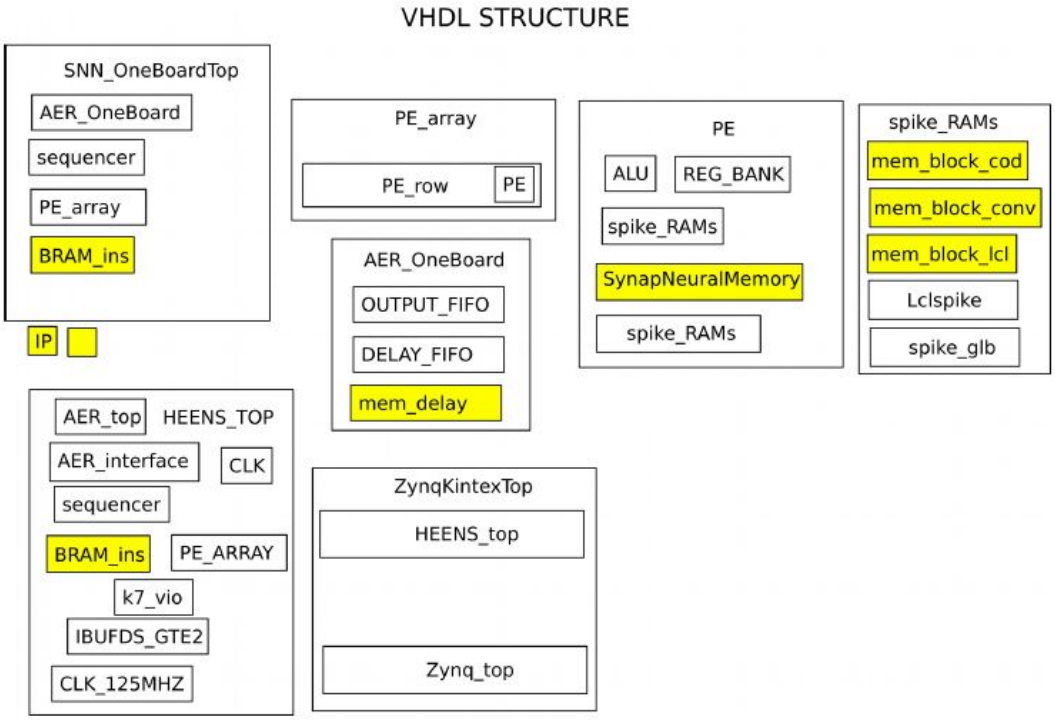


Figure 2.8: VHDL top structure of HEENS and AER architectures [19]

an Aurora IP created for this specific purpose and its characteristic are reported in Table 2.1. As is it shown, there is the AER clock (250 MHz) used to read data from the FIFO, an operation performed by the *Z_AER_TX* module, then the HEENS clock (125 MHz) used to write data inside of it.

Dimension and parallelism
Write Width: 16 bits
Write Depth: 1024 words
Read Width: 16 bits
Clock and Reset
Independent clocks (Block RAM)
Asynchronous reset
Flags
Empty, Almost Empty
Full, Almost Full

Table 2.1: Multi-board Monitoring FIFO IP

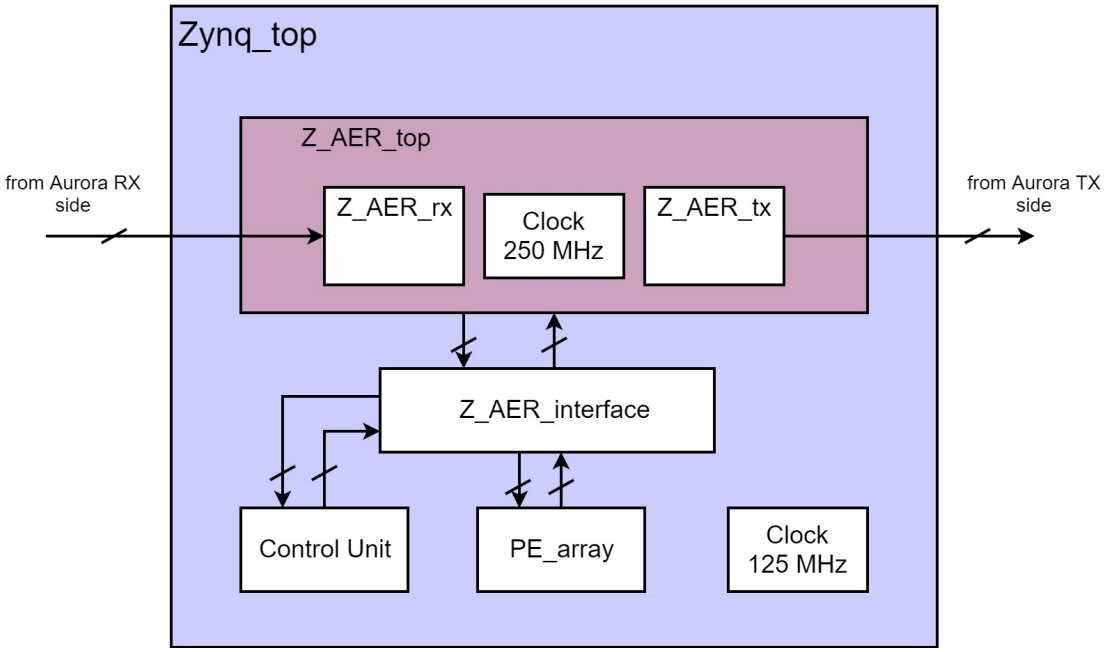


Figure 2.9: Simplified VHDL top structure of the Master Chip

In the interface, a specific controller has been designed to manage the writing into this FIFO and it communicates directly with the FSM in the PE-array, described in the previous section. The block diagram of this structure is reported in Figure 2.10.

The schematic reports only the main parts involved in monitoring operations. It is illustrated the row of monitoring data that goes to a multiplexer inside the interface module, then a specific controller sends a signal *monit_count* (which size depends on the number of data to count and so on the number of PEs in a row), that is in charge of selecting the proper input to load into the monitoring FIFO. There are some signals and data related to the reading of monitoring data from the FIFO, that are handled by the transmission module of AER (*Z_AER_TX*), but this part is discussed in next section.

So, the controller of this module, needs to communicate with the FSM of the PE-array, in order to generate the block and the MUX selection signals. Furthermore, it has to set high the *write enable* flag of the monitoring FIFO, in order to load all data at the input. For these tasks, two dedicated Finite-State Machines have been created and their Timing Diagrams are reported in Figure 2.11.

The first upper TD regards the FSM that is in charge of controlling the blocking signal, in order to load all the monitoring data into the FIFO. It also manages the selection signal of the multiplexer in Figure 2.10, that picks the proper input at the right time. The problem of this operation, could arise if the monitoring FIFO is full, which would mean to wait before writing a new data or loading a new row to the

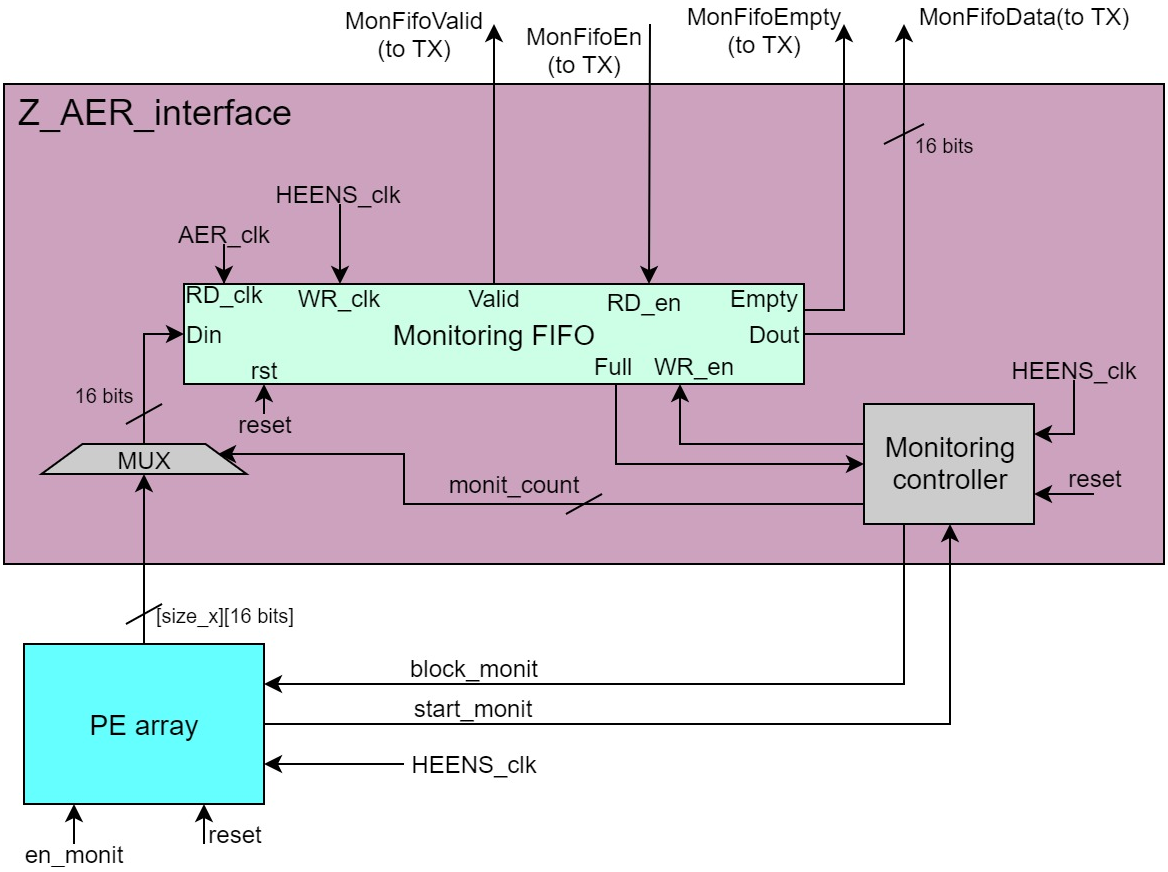


Figure 2.10: Schematic of the *Z_AER_interface* module

output of the PE-array. This particular circumstance is applied in some special cases in the diagram.

The second state machine set to one the write enable signal and it simply has to stop if the *full* flag is high. In this case the *wr_mon_en* signal is in *and* with the negation of the *full* flag for timing causes, which means, again, this is more a *Mealy* like machine, but as was explained in the previous section, it should not create problems.

1. At the beginning a *start_monit* signal (Section 2.2) is set to '1' by the array, which means that new data are coming for the monitoring FIFO, but also the full flag is high, and it maybe comes from the fact that previous data of the monitoring FIFO have not been read and transmitted yet by the TX module. That keeps the FSM stuck in the *full* state until a location gets free and it does not allow to write into the FIFO

   The *monit_block* output signal is in *or* with the internal blocking one generated by the FSM and it prevents the array from loading a new row of monitoring data.

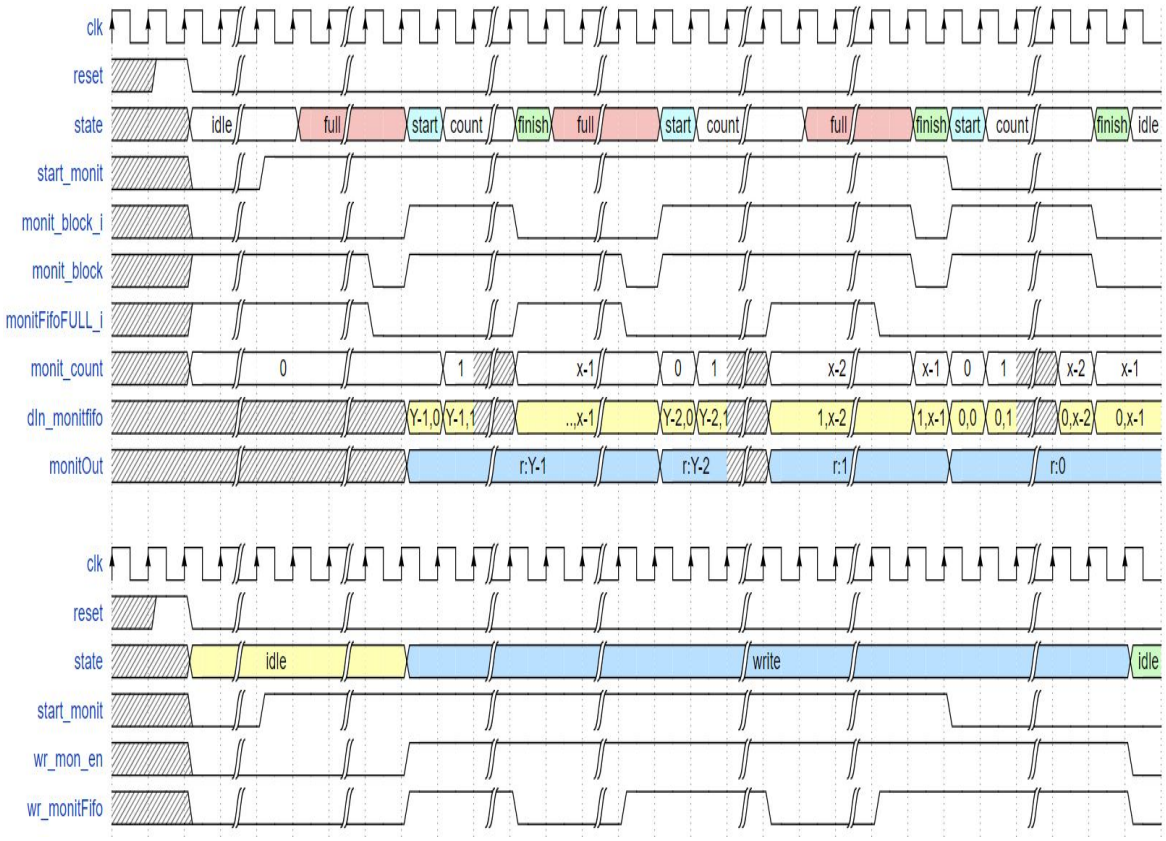


Figure 2.11: Timing Diagrams of the *Z_AER_interface* monitoring controller

2. At a certain clock cycle, the *full* flag goes down and the writing starts: a new row is loaded and the counting signal, that is the selection signal for the MUX, is incremented at each clock cycle. The counter keeps track of what element is being written into the FIFO.

3. Other occurrences of the *full* flag are explored in two interesting cases reported in the diagram, which are the end of a row loading and during the writing of a generic element (in this case the second to last data). What can be noticed from the TD, it's that in every *full* state, the *monit_block_i* signal maintains the last assumed value before entering this particular state: that's choice comes from the necessity of loading or not a new row exactly one clock cycle after the *monitFifoFULL_i* signal goes down (last data to load and a generic one respectively).

4. When the last row has been loaded, the PE-array set to zero the *start_monit* flag and both FSMs remain stuck (by means of the *monit_block_i*) before moving to the *idle* state, since all the elements of the last row obviously need to be loaded into the monitoring FIFO.

The VHDL file related to this module is reported in Appendix D.4. As usual, not all the hardware description of it is included for simplicity.

2.3.2 Z_AER_tx

The transmitter module of the AER architecture is the one in charge of assigning the right time slot to each control and data packets, in order to send it to the Aurora TX part, which in turn will serialize all information and will transmit them by means of the serial communication bus. The mechanism and the main packets were described in Section 1.5.2.

In order to transmit the monitoring information, other control and data packets have been added to the Table 1.1 and these changes are reported in Table 2.2. Similarly to the distribution phase, the monitoring packet includes a start information, to signal a chip that the specific information is coming, then the data and finally the finish control packet to signal the end of this phase.

Contol packet	Function
START MON	Monitoring phase
FINISH MON	

Table 2.2: Monitoring packets of AER-SRT protocol

A picture that illustrates the timing of this packet, is reported in Figure 2.12, where the yellow slots refer to the monitoring information of each PE, of which the relative position in the array is specified.

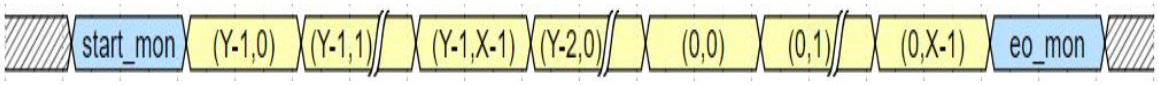


Figure 2.12: Timing of the monitoring distribution

So, the first thing was to add specific packet identifiers in order to build the control packets that will be recognized by the reception part. In Figure 2.13 it is possible to analyze the composition of these packets: the length of transmitted data is 16 bits, so the MSB field signals whether is a control packet (active low) or not, then four bits are dedicated to identify the type of if. There is an unused field that will be exploited in next chapter, while the last part is related to identify from which chip the information comes from (there are 2^7 possible nodes to identify).

It is also important to mention, that the following monitoring data packet does not include (as the other data does) the first *DATA_head* bit (active high): this is due to the fact that all 16 bits of this information are needed, so the MSB cannot be wasted for a control flag. Basically, the RX module of each chip is in charge of realizing that

Start Mon packet:

CNTRL_head (1 bit)	START_MON_head (4 bits)	Unused section (4 bits)	CHID_ID (7 bits)

Finish Mon packet:

CNTRL_head (1 bit)	EO_MON_head (4 bits)	Unused section (4 bits)	CHID_ID (7 bits)

Figure 2.13: Structures of monitoring packets

all information received after the *START_mon* packet are monitoring data, by means of a counter and a special flag. This will be discussed in the next section.

After that, new features have been added to this transmission block. In Figure 1.18, the main structure of it is illustrated and its new characteristics are reported in the block diagram of Figure 2.14.

The main Finite-State Machine activates each smaller FSM dedicated to the particular packet that is being transmitted. In the schematic, all components and signals involved in the monitoring phases are reported. As it is shown, the data FSM interacts with the AER interface, while the main state machine interacts with the sequencer, in order to receive the signal that starts the monitoring phase and to transmit two semaphore flags that will be explained in the dedicated section of the CU.

The outputs of the final MUX are: the data to transmit and a flag (*tx_src_rdy*) that signals the Aurora TX module that the data is ready to be transmitted. The Aurora TX side responds with another flag (*ready_tx*), in order to advice when the bus is able to transmit a data.

The bypass FIFO is also exploited in these operations and to understand why, let's have a look on the specific monitoring section of the main FSM algorithm in Figure 2.15. The flowchart starts with the *IDLE* state, in which the transmitter module waits until the execution phase of the HEENS is over in order to start the synchronization phase (*TX_SYNC_1*). In this state the idle packet is sent to the bus, in order to keep the ring active.

Since the monitoring operations and transmission need to be performed in parallel with the normal execution phase of the main neural algorithm, in the *IDLE* state has been created a branch dedicated to this task, that starts when the *en_monit_in* signal is set to one by the sequencer. The main stages of this process are listed below.

1. The *START_MON* phase starts, in which the enable signal for the FSM of the start packet is set to '1', the proper MUX selection is chosen and also the *monit_busy* signal is set high. This latter is a semaphore for the sequencer, that stops it in case another monitoring instruction is fetched from the IMEM and it is going to be high until the finish packet is transmitted.

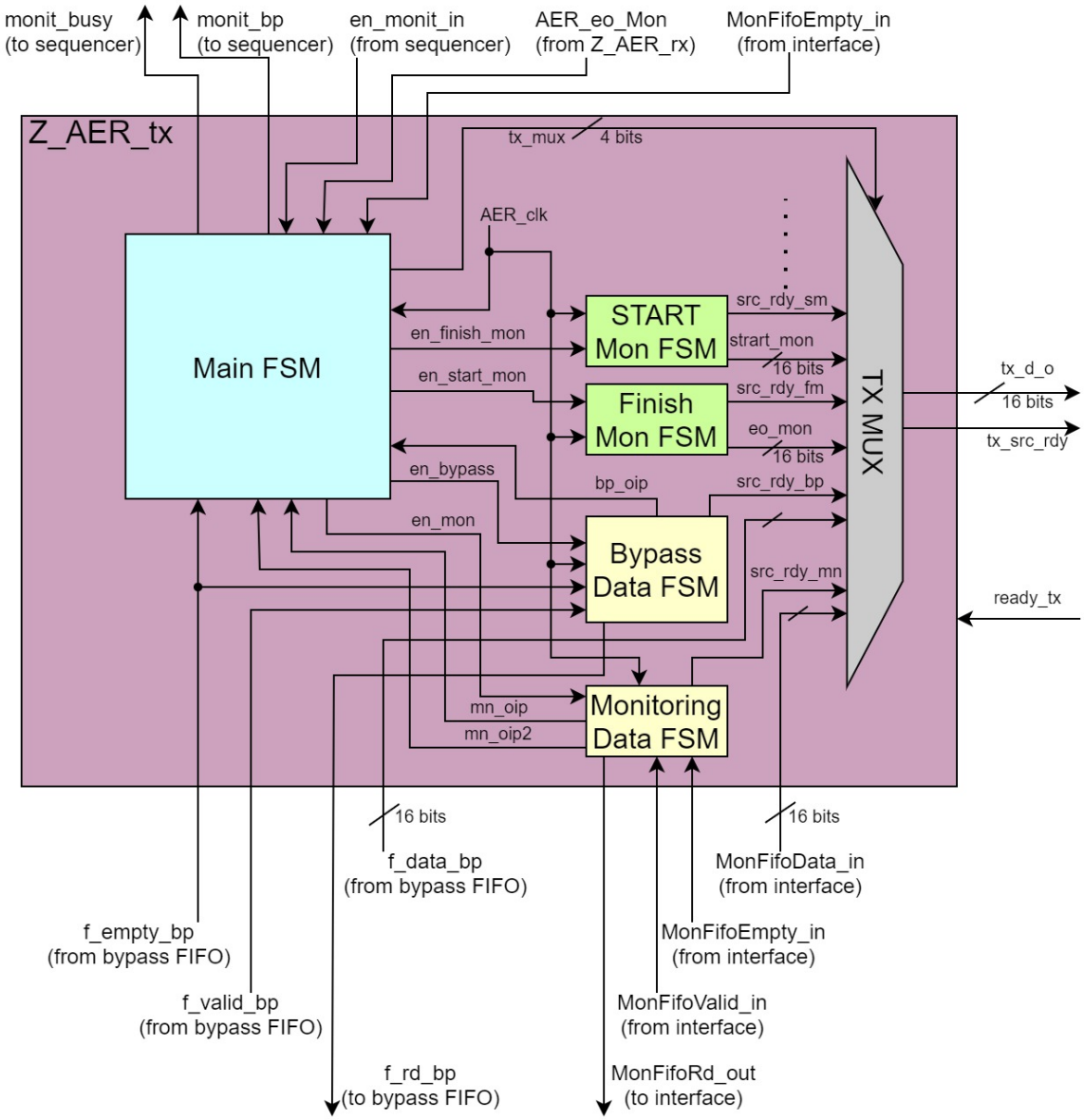


Figure 2.14: Schematic of the *Z_AER_tx* module

For each state there are special flags that signal the end of that specific transmission, like *sm_done* for the start stage. Each FSM generates these signals to let the main state machine move on.

2. Then the monitoring data of the chip are transmitted. Again, the enable flag for the relative FSM is set high and the multiplexer selection signal changes, in order to pick the right inputs. In this case a specific combination of signals and the

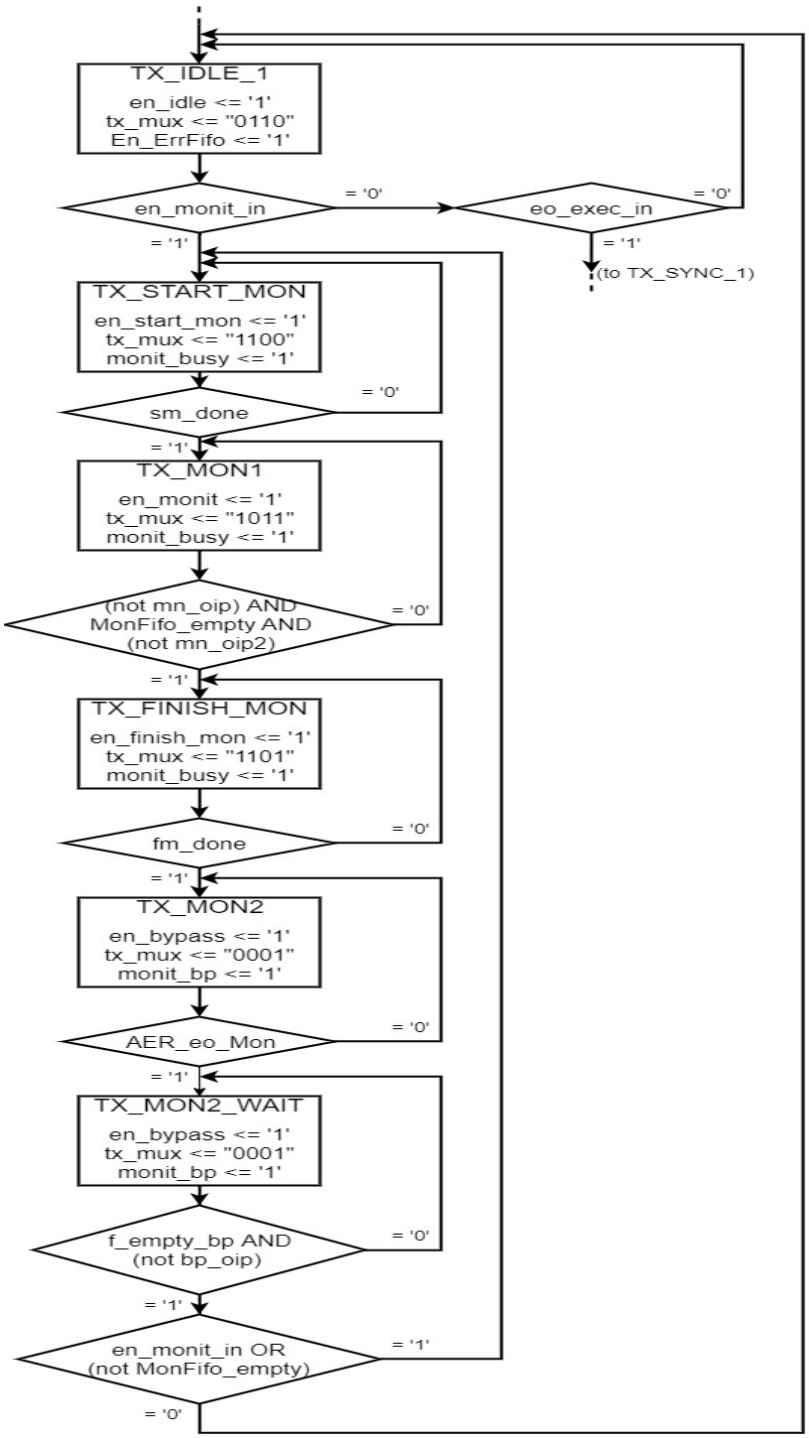


Figure 2.15: Flowchart of the monitoring phase of the main Finite-State Machine in the *Z_AER_tx* module

empty flag of the monitoring FIFO are necessary to finish the transmission.

The final *FINISH_MON* state has the same behaviour of the start state.

3. After that, all monitoring information coming from the other chips need to be retransmitted to the rest of the network and. for this reason, data from bypass FIFO need to be picked and sent to the serial communication bus: the receiver module of AER writes these kind of data to the bypass and counts how many packets have been received. When a fixed number is reached, it will set *AER_eo_Mon* high and the TX main state machine will move to the last state.

4. In the final stage, even if the proper number of monitoring packet have been received, maybe the chip has to retransmit them to the ring (if it's a NC), so the main state machine must wait until the bypass FIFO is empty. This concept will be explored better in the RX module.

   Finally, a last check is done, in order to verify if another monitoring instruction has been launched by the sequencer.

The flowchart of the *START_MON* FSM is reported in Figure 2.16. It is important to underline that in the TX module, the output *tx_src_rdy* signal is active low, and so also the valid signals will have this characteristic.

Anyway, this FSM has a simple behaviour: if the *sm_done* flag is high, then the valid signal is set low, but the final *tx_src_rdy* gets low only if the bus is ready to transmit (*ready_tx* = '1'). If the transmission is successful, the *sm_done* is set high to signal that the start monitoring packet has been transmitted. The *FINISH_MON* state machine works in the same way, so it will not be analyzed.

The FSM that handles the transmission of monitoring data is analyzed through a TD, since it presents some occurrences that need to be studied. The diagram is reported in Figure 2.17 and as usual some interesting eventualities are examined below.

1. In order to start the transmission the FSM must be enabled by the main TX controller, the bus needs to be ready to transmit and the monitoring FIFO has to contain data (*en_monit* = '1', *ready_tx* = '1' and the FIFO empty flag to '0' respectively). In this example, at the first tentative to start, the FIFO is empty because maybe the rest of the architecture has not loaded data into it yet. Then, when it is not empty anymore, the monitoring FIFO is read and transmission starts.

2. Immediately after, an example of the possible behaviour in case an empty flag is raised is depicted. The last data is transmitted correctly, but the FSM moves to an *EMPTY* state and then back to *IDLE*, until the FIFO obtains some new data to send.

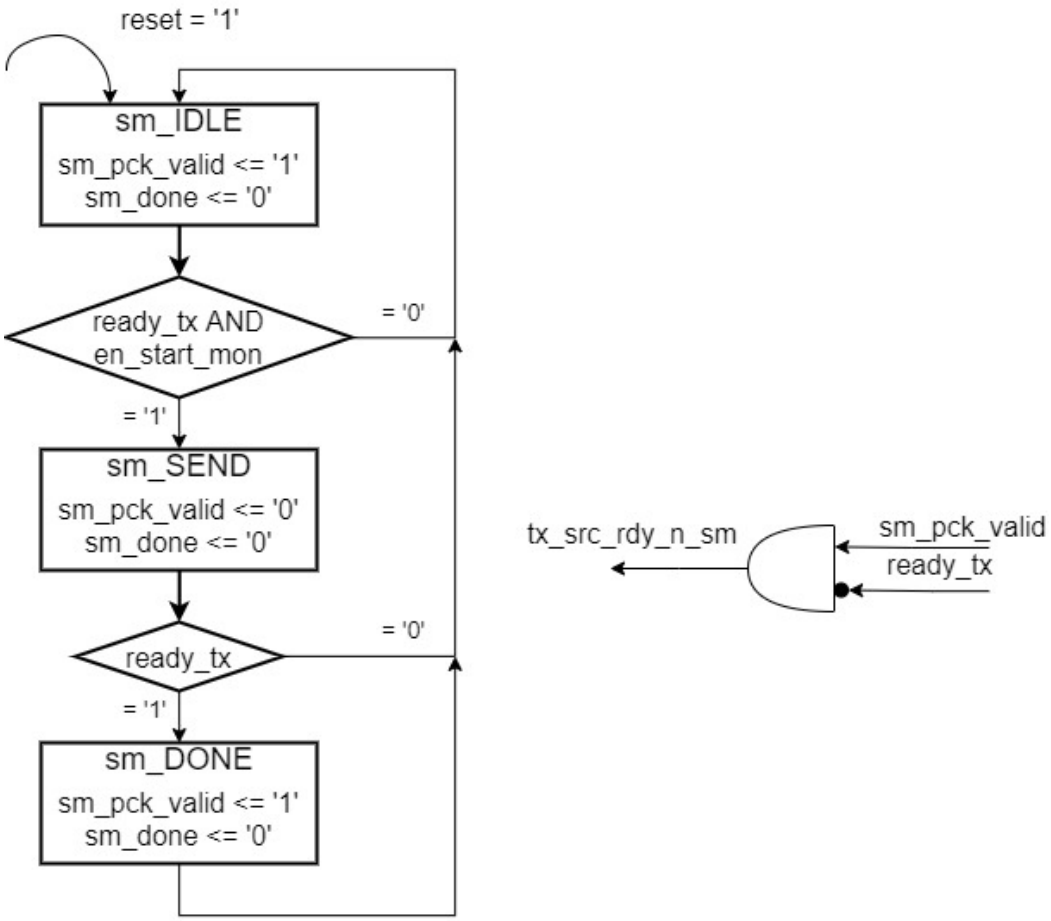


Figure 2.16: Flowchart of the *START_MON* Finite-State Machine in *Z_AER_tx* module

3. Then, another eventuality is examined, that is when the *ready_tx* signal goes down and so the bus is not ready anymore to transmit, in the middle of a monitoring transmission operation: in this case, the *EMPTY* state is reached but then a waiting state starts.

   When the Aurora TX side gets ready again, the last data that was not transmitted because of the bus, is now sent and then the transmission continues normally. The only problem is that the *valid* flag of the monitoring is not zero anymore (active low), since this component asserts this signal only for one clock cycle. So, another signal *wait_docc_mn* is set low to signal that the source data is ready to be transmitted.

4. Finally, when the last data of the last PE is sent, the state machine goes back

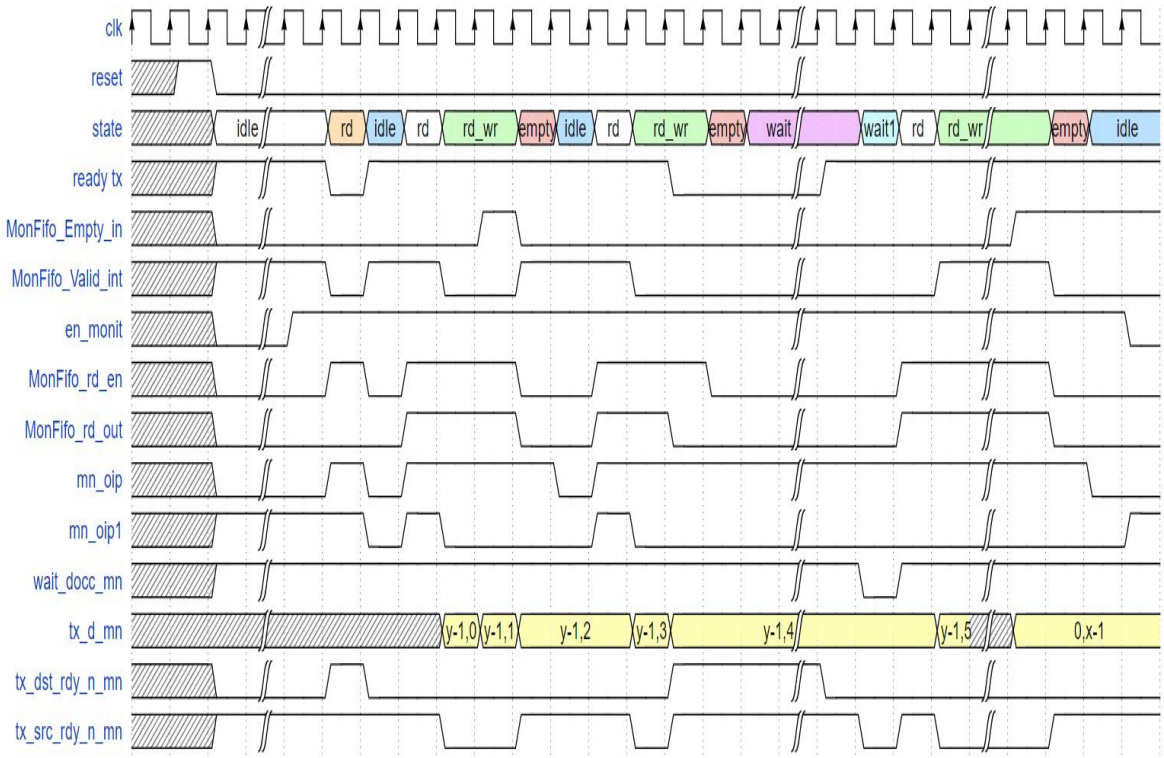


Figure 2.17: Timing Diagram of the monitoring data packet FSM

to *IDLE* state. At this point both signals *mn_oip* and *mn_oip1* are low, the monitoring FIFO is empty, which means that the monitoring controller has finished to transmit and the main state machine can move on.

The signals *mn_oip* and *mn_oip1* need to not let the main TX controller move on the next state (*FINISH_MON*), in case the monitoring FIFO gets temporally empty, or maybe at the beginning if the data have not been loaded yet. Only if the monitoring state machine goes back to the *IDLE* state and the FIFO remains empty for some clock cycles, the main controller is allowed to go on.

A person could argue on the fact that for any reason, the AER interface module could get blocked for a while in the middle of the monitoring transmission and so, the main TX state machine would wrongly move on. That is an almost impossible eventuality since once the loading of monitoring data (performed by the interface module) starts, there should not be interrupts. Anyway, this improbable issue is fixed in the upgraded version of the next chapter.

The *bypass* controller is very similar, so it won't be examined. The monitoring state machine can be found in the Appendix D.5, together with all the other controllers and components that have been described so far regarding the *Z_AER_tx* module. Not

all the controllers of it have been included for simplicity and also because they are not relevant in this explanation.

2.3.3 Z_AER_rx

This is the last modified module of the AER architecture. Before analyzing it, a couple of important issue need to be explained. The only main difference between a Master Chip and a Neuromorphic Chip in the monitoring implementation, regards their actions after this kind of information is received by the relative modules: the MC has to store all these data inside a specific FIFO, called *SinkFIFO*, of which information are summarized in Table 2.3. The MC needs to store as many monitoring packets as there are chips in the ring, considering that it will also receive its own packet that it sent previously.

Dimension and parallelism
Write Width: 16 bits
Write Depth: 1024 words
Read Width: 32 bits
Clock and Reset
Common clock (Block RAM)
Asynchronous reset
Flags
Empty, Almost Empty
Full, Almost Full

Table 2.3: Sink FIFO IP

As is it shown in the table, this specific FIFO is characterized by a 32-bit read width: this component is read by the PS-interface (ARM processor) and this latter works with a parallelism of 32 bits precisely, so this word width is required. Furthermore, in this case a common clock for reading and writing has been chosen, since the PS module is not ready yet and the clock frequency is not determined. Therefore, for now, both operations are performed at the same operative frequency. Anyway, another *SinkFifo* component with two different clocks was created for every eventuality.

The regular NC node has not this latter FIFO: indeed, what it it supposed to do when monitoring data are received, is to load them into the bypass FIFO and to retransmit them to the ring. It has to wait for a number of packets equal to its relative position after the MC. So, for example, the second NC after the master one has to receive and to transmit again two monitoring packets. These concepts are illustrated in Figure 2.18.

In the figure, the red line refers to the links of serial bus, but in this case they

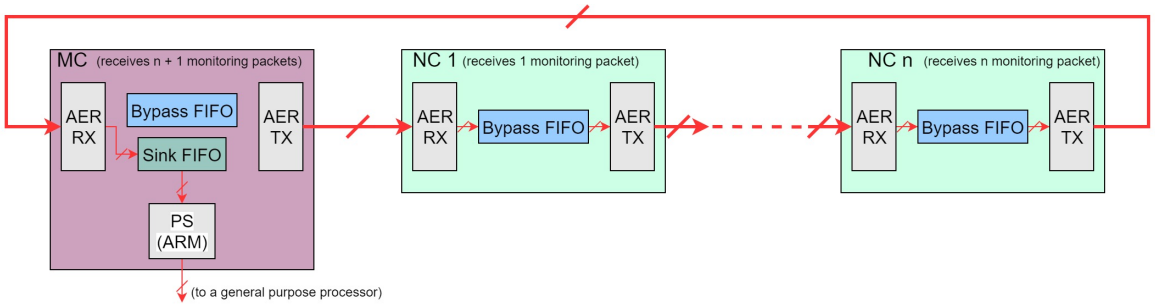


Figure 2.18: Monitoring procedure in a multi-board network topology

are related to the monitoring transmission. Indeed, inside each chip only the path that transfer this latter information is underlined in the schematic. For simplicity, the monitoring FIFO and its relative links are not reported.

It is shown that each NC does not present a *SinkFifo* and it sends the received data directly to the AER transmitter. It is underlined for each chip how many packets it has to wait. The Master Chip (MC) stops for a number of packets equal to the ring size (n+1) and stores directly them in the *SinkFifo*.

This presented so far, it is how normally the network should behave. Since in this work the NC has not been used , only one MC has been adopted and, following the procedure explained above, it should wait only for its own monitoring data and move on. In order to verify the correct behaviour of the regular NC, in this project both properties of the two kind of chips has been implemented in the MC. This feature is shown in Figure 2.19.

The picture shows that MC writes i times the received monitoring packet in the bypass FIFO, in order to emulate the behaviour of the last NC in a $i+1$ network topology. Then, it will write in parallel the data into the *SinkFifo*, which is the regular behaviour of a MC in a network of the same size $(i+1)$. Each time the bypass FIFO receives a packet, it transmit it again to the serial communication, by means of the *TX_MON2* and *TX_MON2_WAIT* states of the TX module in Figure 2.15. So, basically the monitoring data travel in a loop for fixed number of times, that can be determined by setting a parameter i (that is *MON_SIZE*) in the VHDL package.

In Figure 2.20, the block diagram of the receiver is reported, where, as usual, the main blocks involved in monitoring operations are illustrated. As it is shown, the data received by the Aurora core are forwarded to several FIFOs, among which there are the *Bypass* and the *Sink* FIFOs. Two flags are received too from the Aurora RX side: the first, *rx_src_rdy*, indicates when source data is valid and it is used by the decoder.This latter is in charge of detecting all packets used in this protocol. The *channel up* flag reports the status of the channel to the FSMs of the module, like the *ready_tx* of the transmitter side.

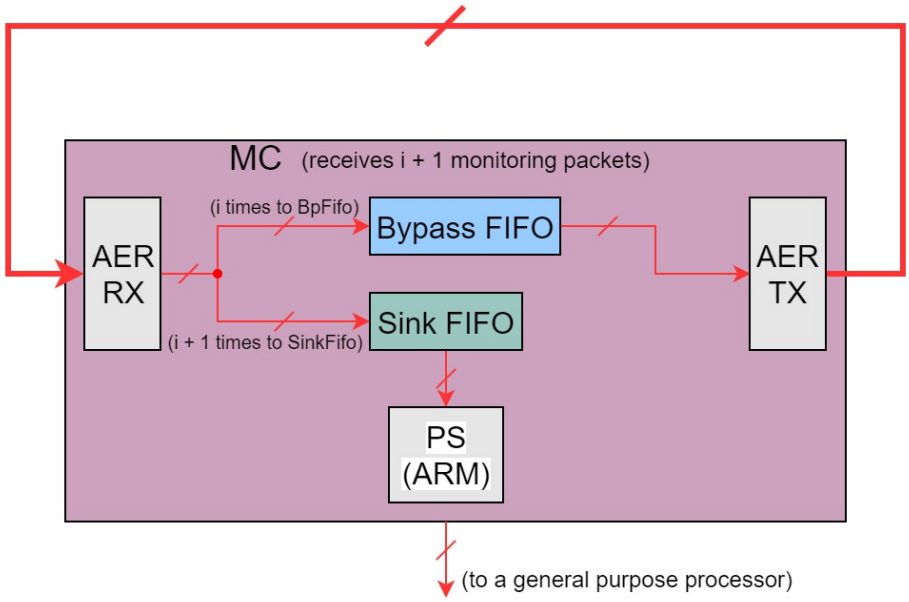


Figure 2.19: Monitoring procedure with a single MC in the ring

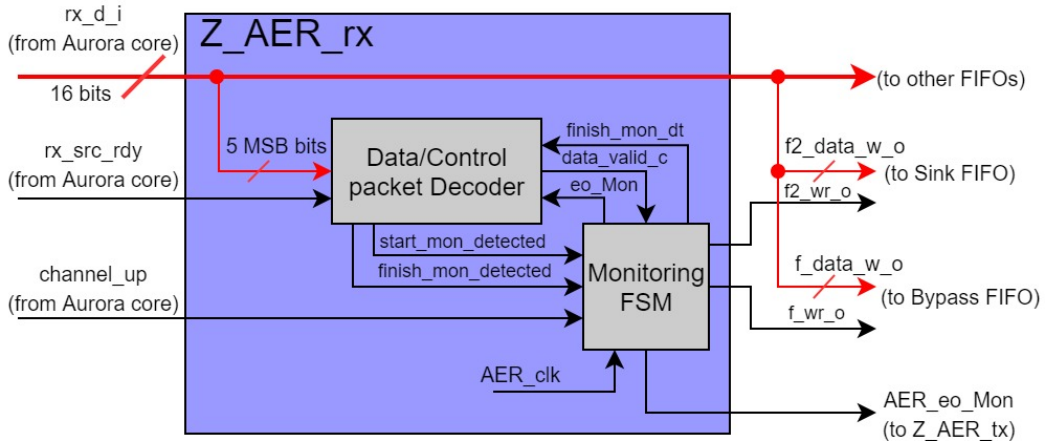


Figure 2.20: Schematic of the *Z_AER_rx* module

Basically this controller has to set in a proper way the write enable flags of *SinkFifo* and of the *BypassFIFO* and it sets also the *AER_eo_Mon* flag to indicate that the correct number of monitoring packets have been received. The flowchart of this controller is reported in Figure 2.21.

Apart from the *IDLE*(*reset*) state, the others are needed to set the output signals in a proper way, in order to write either in both FIFOs (sink and bypass), or only in the *Sink* one or finally in none of them (when the *finish* monitoring packet is detected).

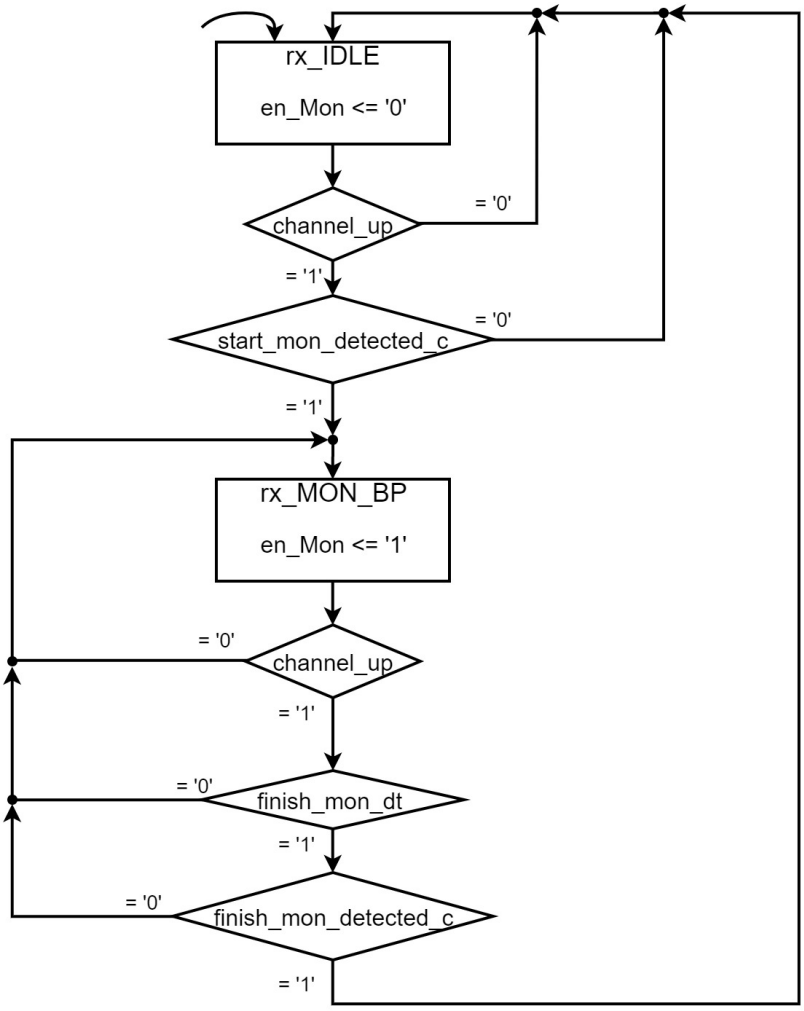


Figure 2.21: Flowchart of the monitoring controller in the *Z_AER_rx* module

The flag *MON_SIZE* is used to distinguish between the first two possible situations and it represents the number of virtual nodes of the ring (the equivalent of parameter i in Figure 2.19). It can be set in the *SNN_pkg* VHDL file. The meaning of these signals is explained in the schematic of Figure 2.22.

Basically, the *f_wr_o* write enable flag of the bypass FIFO is set to '0' if the conditions in the TD are true, which are related to the fact that data must not to be loaded anymore in this latter FIFO. These other input of the *OR* gate in the figure are not reported for simplicity. The only thing to say is that the bypass FIFO is used in other phases too, so the other conditions, set by the creator of the module, were already present and those had not to conflict with monitoring ones.

The *finish_mon_dt* is used to signal when the right number of data have been

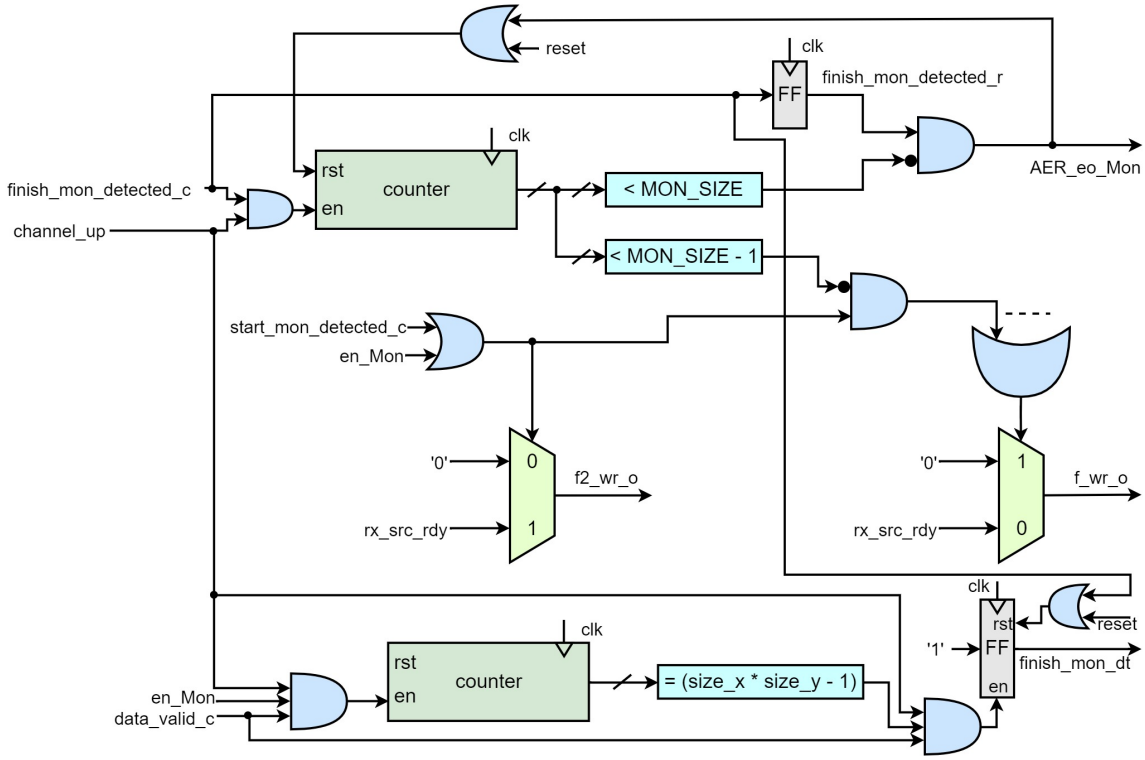


Figure 2.22: Data path of the monitoring controller in the *Z_AER_rx* module

received ($size\_x \cdot size\_y$ data coming from all the PEs of a chip), which is necessary since monitoring data exploits all bits available for the communication, so the receiver cannot distinguish between a control or a data packet. Therefore, after the *START_MON* is detected, the flags *en_Mon* is set to '1', which causes the receiver to treat all following received packets as *data*, regardless of the value the MSB has.

Only when *finish_mon_dt* is high, the *FINISH_MON* packet can be detected and the *en_Mon* and is set to zero again. The signal *data_valid_c* (active high) indicates when the data received is a valid and it has to be high to increment the data counter of to set the *finish_mon_dt* flag. All VHDL source files regarding these components and controllers of the receiver related to monitoring operations are reported in Appendix D.6.

2.3.4 Sequencer

This is the last module that will be discussed in this chapter. As already mentioned in the introduction, HEENS is a Harvard architecture and so a specific Instruction Memory is present in order to store all the instructions needed to perform the algorithm and other important functions. The instruction is performed in four pipelined stages,

which are *FETCH*, *DECODE*, *EXECUTION* and *WRITE BACK*.

In this first version of the monitoring implementation, the sequencer needs to stop the execution of the algorithm in two particular situations because of the monitoring phase: the first one is when a *STOREB* instruction is fetched and either the propagation or distribution of a previous one have not been completed yet. The second time the algorithm is stopped, is when the execution stage (that is performed in parallel to the monitoring operations) ends and the spike distribution phase needs to start. Indeed, since the serial communication bus exploited to distribute data is the same, the architecture is not allowed to start distributing spikes if monitoring data are still travelling around the ring.

In Figure 2.23 a simplified schematic of the CU and the synchronization signals involved in the monitoring operations are presented. Signals from the AER modules have been generated with a higher clock frequency respect to the *HEENS_clk*, so they need to be synchronized. For this task, specific components formed by a cascade of three flip flops are exploited, in order to prevent the signal from going in a *metastable* state and the opposite situation, regarding the *en_monit* signal, is handled as well. The schematic of these components is reported in Figure 2.24.

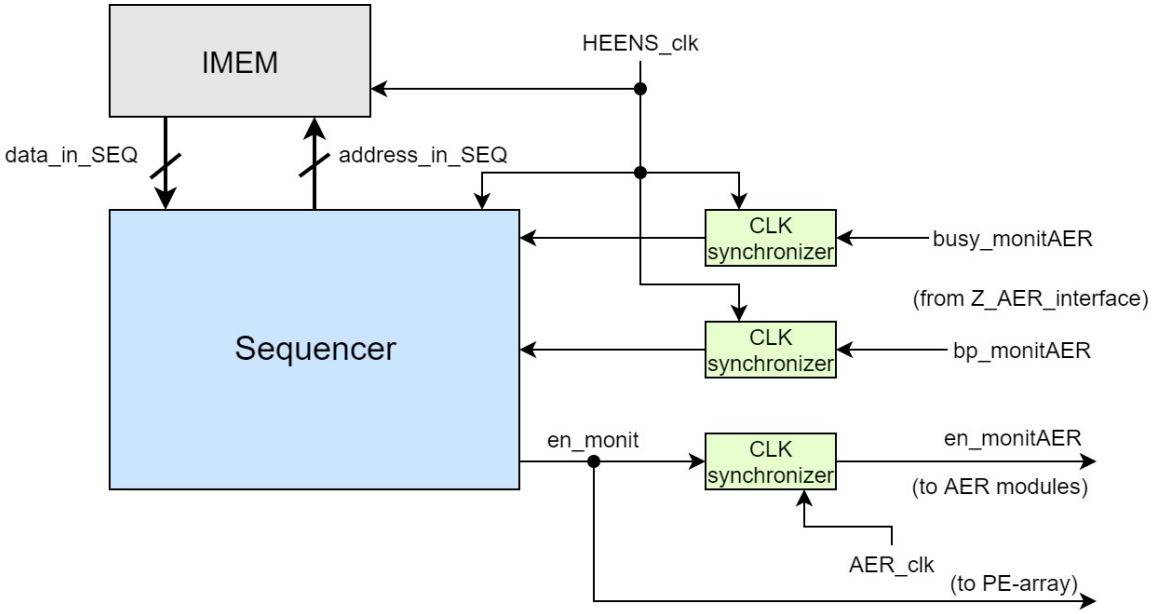


Figure 2.23: Control unit and monitoring signals

The idea behind this synchronization signals, is to emulate the semaphores concept used in operating systems, but in an hardware fashion: basically, lock and unlock actions will be performed by some components in the sequencer, in order to understand if this latter is allowed to move on or stop the execution of the algorithm. This idea is illustrated in Figure 2.25.

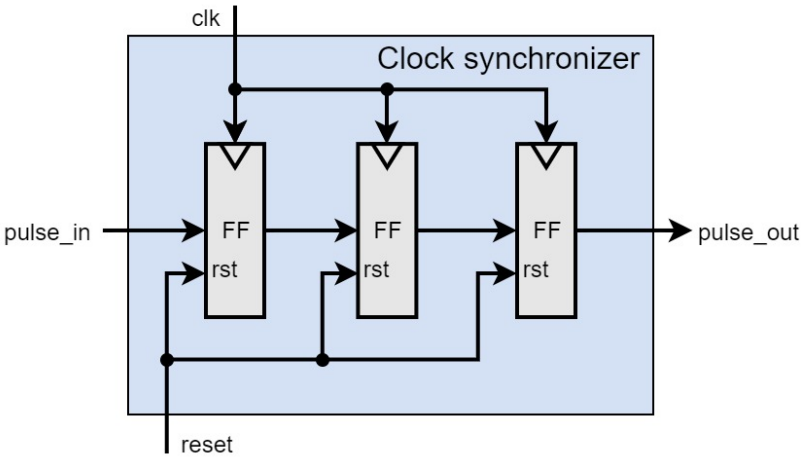


Figure 2.24: Schematic of the *clock synchronizer* component

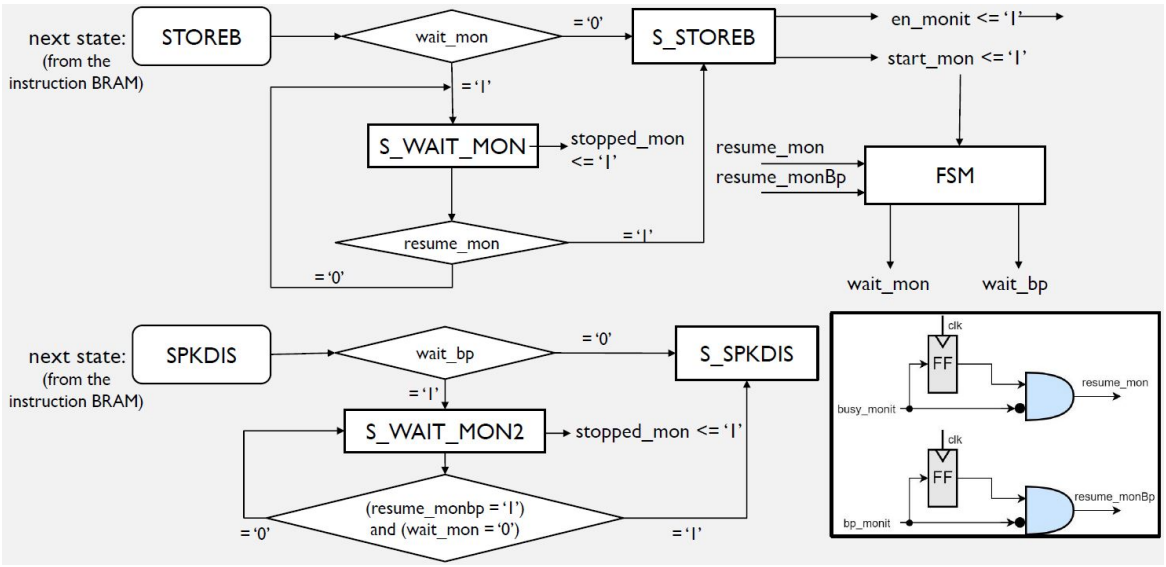


Figure 2.25: Monitoring states of the main FSM in the sequencer unit

Let's analyze the case in which a *STOREB* instruction is fetched. Like a semaphore, a flag *wait_mon* is checked and if it's free (equal to zero), the instruction can be decoded. In the *S_STOREB* state of the main controller in the sequencer, the output *en_monit* is set high and delayed by one clock cycle, in order to make it coincide with the execution phase of the pipeline stages. Then another flag *start_mon* is set to '1' and this latter activates a small FSM, that is in charge of "blocking" the semaphore, by means of setting to one *wait_mon*. Then, if another *STOREB* is fetched from the IMEM, it would find the blocking signal high and thus, the state machine would move

in a waiting state.

The only way to leave this condition, is through the *Z_AER_tx* module, that, after it has transmitted the monitoring data of its own chip, will set the *busy_monit* signal to zero again (Section 2.3.2) and this will let the *resume_mon* flag go to one for one clock cycle, like a pulse. This will free the semaphore, so *wait_mon* will go back to zero and the blocked *STOREB* will proceed to the decode stage.

A similar procedure is performed by the second semaphore, that is in charge of controlling if the algorithm is trying to move towards the spike distribution phase before the whole transmission of monitoring data has been completed. There are only two differences: the first, is related to the fact that the *wait_bp* flag is set automatically to one (which means that this semaphore gets locked), after a *STOREB* instruction is decoded.

The second regards the release of the semaphore when the *wait_mon* flag is checked again, in order to verify if another monitoring instruction was previously launched. Indeed, if all the information of the previous chips have been received (and transmitted again in case of a NC), the AER TX module would lead to the release of the second semaphore and the spike distribution would wrongly start, even if another *STOREB* was launched. Instead, by checking again the first semaphore (*wait_mon*), the sequencer would stop and wait until the newest monitoring distribution ends.

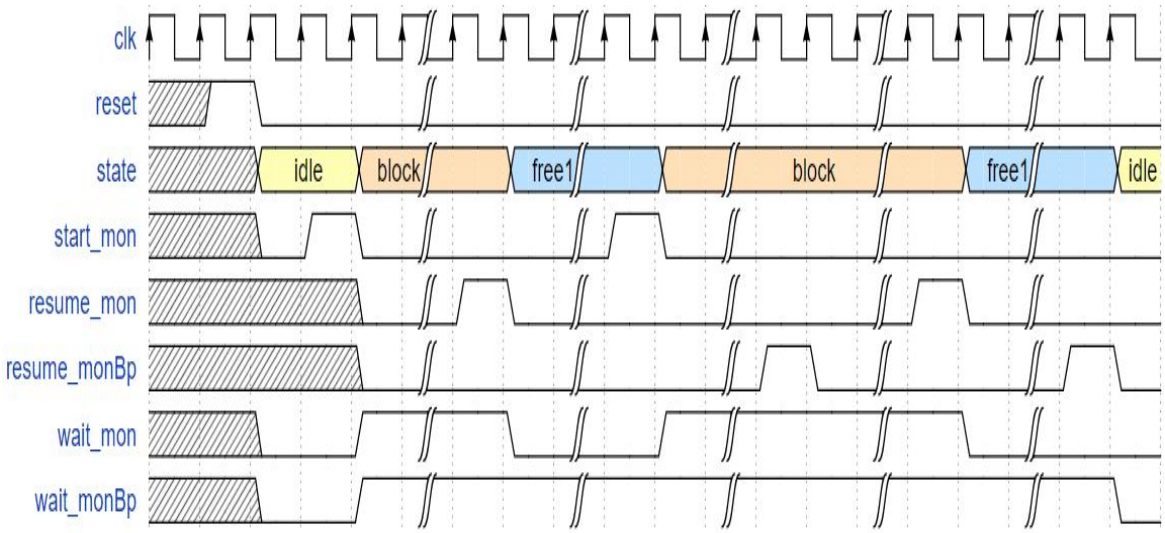


Figure 2.26: Timing Diagram of the semaphores FSM

The TD of the FSM inside the sequencer that manages the two semaphores, is reported in Figure 2.26. When *start_mon* goes to '1', both semaphores get locked. After that, If the *resume_mon* is set high, the first semaphore (regarding the blocking of another *STOREB*) is released. Then, another same instruction is decoded (*start_mon* goes to '1' again) and so both semaphores are locked again.

At this point, even if all first monitoring data has been received and transmitted by the *Sink/Bypass* FIFO (depending if it's a NC or a MC), the *resume_monBp* is ignored, since the FSM is stuck in the *blocked* state again. Then, if *resume_mon* and *resume_monBp* are set to zero in this order, both semaphores can be released and the FSM goes back to the *idle* state.

The VHDL source files of these blocks and components in the sequencer are not reported for simplicity, since, apart from the latter simple controller, all other changes that have been made to support this semaphores procedures (like the two waiting states of the main state machine) are small and punctual, but several and scattered throughout the architecture.

2.3.5 Simulation

In this section, all relevant modules and their actions described so far are verified through simulation. For this task the *QuestaSim Advanced Simulator* is exploited.

In these simulations, the algorithm of Appendix B.2 with the monitoring instructions of B.3 and so the netlist of Appendix C.3 are used. A five by five PE-array is utilized. The behaviour of this configuration is reported in Figure 2.27.

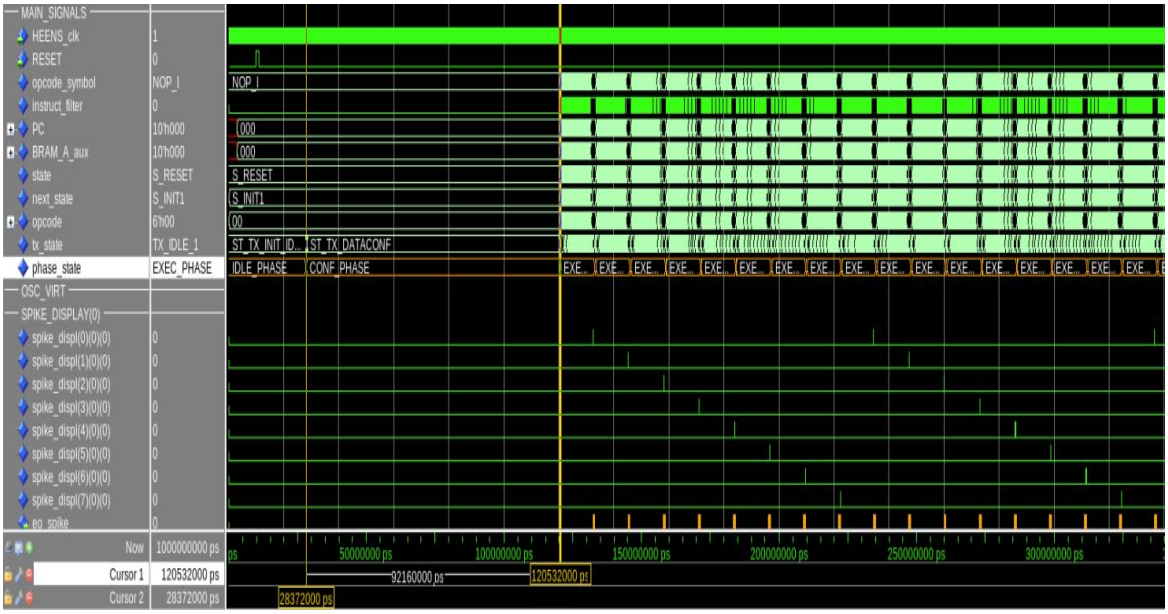


Figure 2.27: Simulation of the LIF algorithm with a 5x5 oscillator configuration and virtualization

As it is underlined by the *phase_state* signal, the *configuration* phase lasts more or less $0.1ms$, during which the AER TX module goes through the *IDLE* and *DATACONF* transmission states. After that, executions stages and distribution phases are performed

by the architecture. The spike display shows the only PE involved in this simulation and the first index on the left, refers to the virtual level. So, as it was explained in Section 1.7, a neuron excites the one of the same PE but that belongs to the following virtual level, which creates this oscillation behavior showed in the snapshot. Basically, the eight neurons associated with the PE at row 0 and column 0 are connected forming a ring oscillator.

- *Monitoring packets*

  In Figure 2.13, the structure of the monitoring control packets is shown and, starting from those, it is possible to predict the precise values they are going to have for this simulation. The *START_MONITORING* is going to assume the hexadecimal value of "5*B*01" (*b*"0101101100000001"), which corresponds to: one MSB for the control head "0", four bits related to the monitoring start packet identifier "1011", then four unused bits set to "0110" and finally the identifier of the chip "0000001" (it is the only chip present in this version).

  Then all the information coming from the PEs are transmitted, which represent the real monitoring data. In the algorithm reported in Appendix B.3, as it was explained in Section 2.1, it is clear that the PEID of each PE is loaded into the accumulator, then moved in the *R3* register, and then two monitoring instruction are performed. So, basically all the PEIDs will be transmitted as monitored information, in order to check the correct behaviour of the system. Following the order of the information transmitted that is illustrated in Figure 2.12, monitoring data will be: *h*"0040", that is the identifier of the PE at the first row (n°4) and first column (n°0), then *h*"0041" that belongs the the first row and second column and so on. The final transmitted data will be *h*"0000", *h*"0001",..., *h*"0004", which represent all the information coming from the first row (n°0).

  After the monitoring data packet, the *FINISH_MON* control information is transmitted. It differs from the start one only for the four bits related to the packet identifier, that in this case are: *b*"1100". So, it is represented by the hexadecimal value of "6301" (*b*"0110001100000001").

- *Z_AER_interface*

  Let's visualize the behaviour of the interface module, in order to verify the algorithms designed.

  In Figure 2.28, a debug signal *MonitFullFifo_deb* has been inserted to check if the *full* condition of the FIFO at the beginning of these stages is correctly handled, since the monitoring FIFO does not get full in these simulations. It can be noticed that the blocking signal is correctly set high each time a new row of monitoring data is loaded from the PE-array and monitoring data are all loaded into the FIFO.

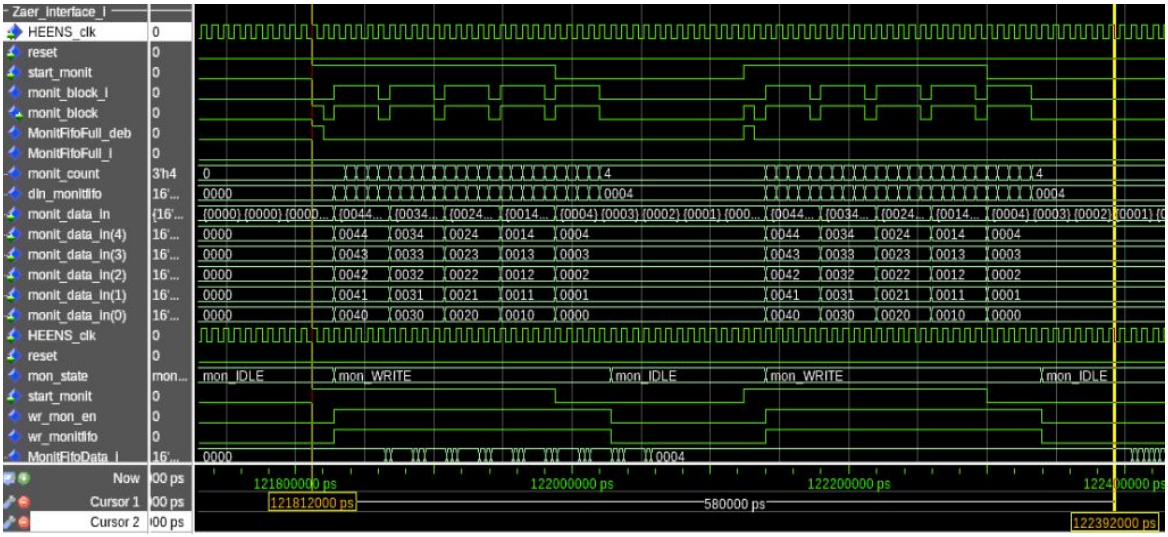


Figure 2.28: Simulation of the *Z_AER_interface*

- *Z_AER_tx*

  In Figure 2.29 relevant monitoring phases of the transmitter module are captured. The important thing to notice is the transmission of the *t_d_mn* data, in the upper part and the states of the three different FSMs involved in the TX unit (related to the *start*, *data* and *finish* monitoring packets). After the *en_monit_in* signal is set to one by the sequencer (it is high for two clock cycles since the AER clock period is the half of to the HEENS one), the start monitoring packet is sent with the expected hexadecimal value of "5B01".

  Then, all the PEIDs are sent (starting from "0040" to "0004"), by means of the monitoring data packet controller and finally the *finish mon* packet is sent ("h6301"). It is important to underline that the output signal *tx_src_rdy_n_o* is in charge of telling the Aurora TX module, when the data is ready to be transmitted. As it is shown in the simulation, this latter is set to zero (it is active low) each time the correct data is transmitted by the AER tx module.

  In Figure 2.30, the final phases of the transmission are reported: for this simulation, the *MON_SIZE* parameter, that indicates basically how many time the monitoring data have to travel in a loop in the AER bus, is set to '2'. Therefore, the transmitter module sends the data of the chip, which are then received and written into both *Sink* and *Bypass* FIFOs. In this way the *Bypass* phase of the monitoring transmission is performed once. Finally, when these information are received for the second time, the receiver writes them into the *Sink* FIFO and the transmission phase ends, by means of the *AER_eo_Mon* that is set to one (for one clock cycle).

  So, what is reported in the figure, represents this very last bypass stage

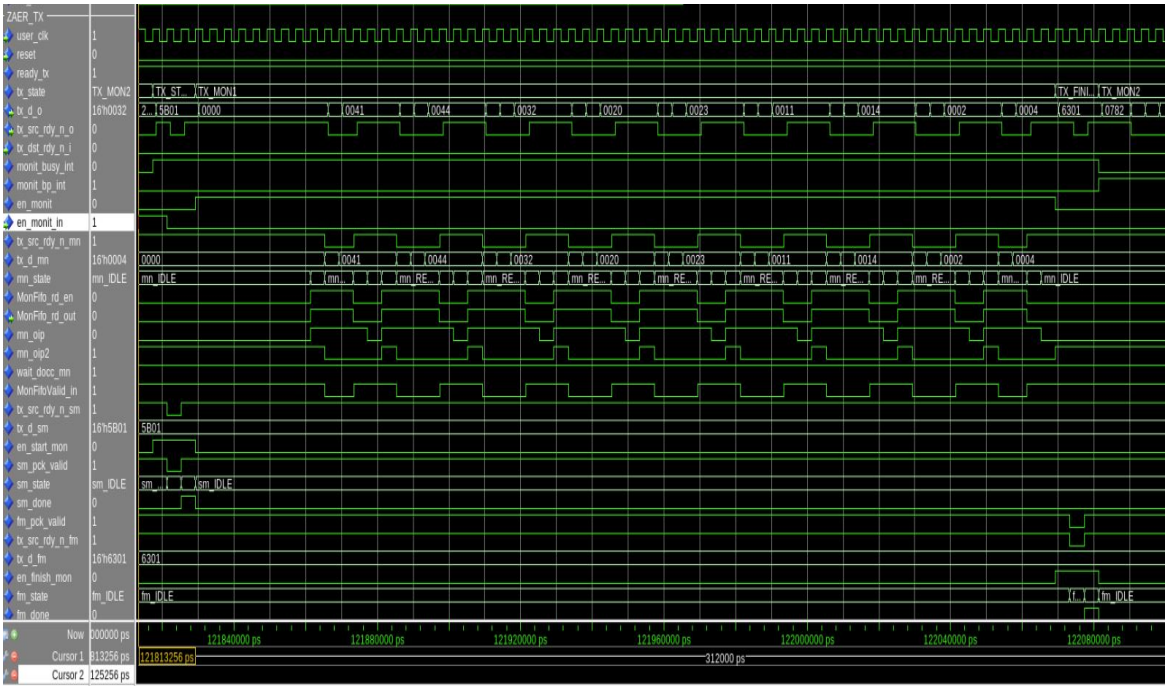


Figure 2.29: Simulation of the *Z_AER_tx*

(*TX_MON2*), in which all data are transmitted again to the ring.

- *Z_AER_rx*

  For the receiving module, a snapshot of the simulation is reported in Figure 2.31. Here, the several packet detection flags, show all the phases of communication: after the first *start_mon_packet*, all data are written in both FIFOs (*f_wr_o* = '1' and *f2_wr_o* = '1' as well). Then, after the first *finish_mon_packet* is received, only the *Sink* FIFO is activated. Finally, after the second finish packet, *AER_eo_Mon* is set to one, as expected.

  It is possible to notice that only when the right number of received data is reached, the *finish_mon_dt* flag is set to '1', in order to prevent the receiver from mistaking a regular monitoring data for the finish control packet. Only after the proper number of data is received, the module is allowed to detect a finish packet (*finish_detect_c* can be set to '1').

  It is also important to underline, that first set of data received in the figure, are not perfectly consecutive: this is due to the fact that in this first implementation, the AER TX module has to wait for the PE-array and the interface units to write data into the monitoring FIFO and they work with the HEENS clock, which has a lower frequency. So, the first transmission and also reception will have some

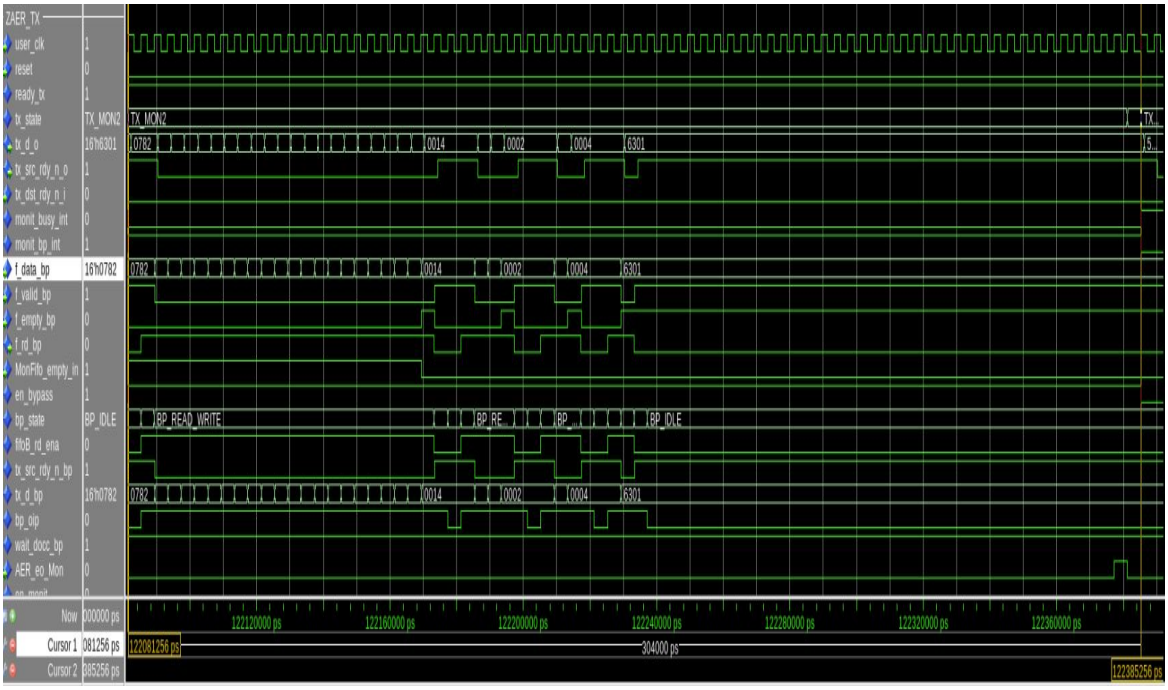


Figure 2.30: Simulation of the *Z_AER_tx* (bypass phase)

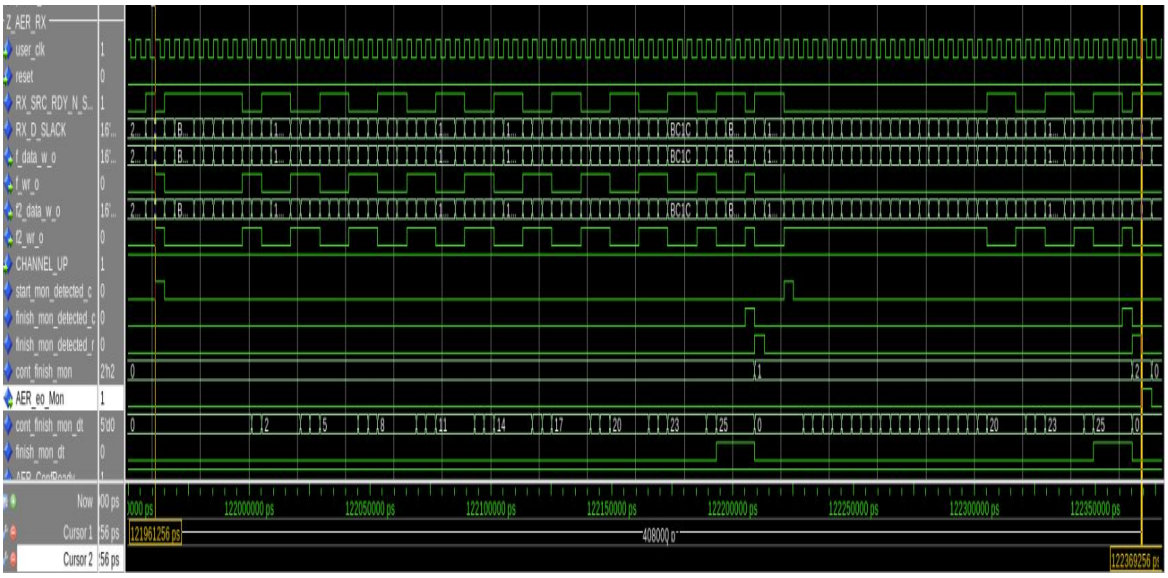


Figure 2.31: Simulation of the *Z_AER_rx*

clock cycles of not valid data. From this first packet received on, these gaps are recovered and indeed the *f2_wr_o* flag, in the second time, is continuously high.

- *Sequencer*

  Finally the sequencer results are reported in Figure 2.32. The figure illustrates the last two monitoring instructions before ending in the spike distribution phase. It can be noticed how the sequencer enters in the first waiting state because of the second *STOREB* instruction and then in the second *S_WAIT_MON2*, before starting the *S_SPKDIS* state, in order to let all monitoring data finish to travel around the ring.

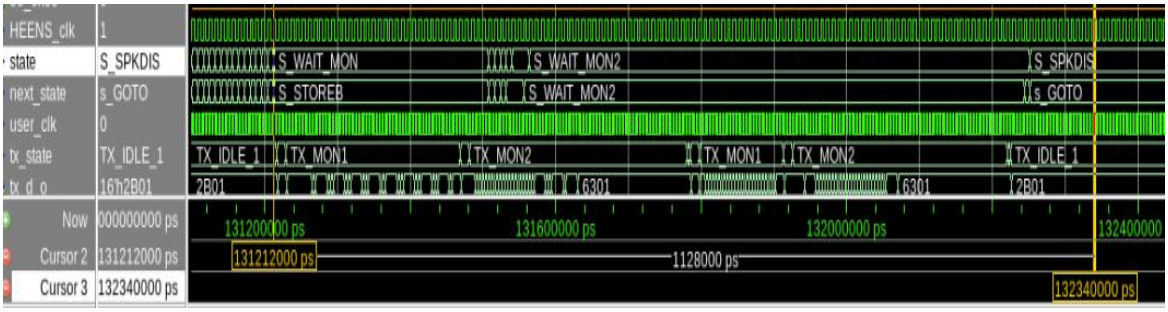


Figure 2.32: Simulation of the sequencer

2.4 Single-Board version

This section regards the single-board implementation. As it could be imagined it is a much simpler version of the HEENS architecture, since it does not include all the AER modules discussed so far. It's purpose, is to verify through the FPGA implementation the functionalities of the neural algorithm without exploiting the AER-SRT protocol.

Indeed, it is composed by an AER single board version unit, which will communicate directly with the ARM processor interface, in order to finally transmit or receive configuration, initialization and monitoring information.

2.4.1 Architecture and main differences

In this version, a simplified architecture is needed to handle both spike and monitoring phases, whose schematic is illustrated in Figure 2.33. As usual only monitoring signals are reported for simplicity.

The *Z_AER_controller* handles all flags and data related to monitoring operations in the multi-board version, while now they are managed by this simpler *AER_SB* unit. All the HEENS side components (sequence, PE-array and so on), remain the same and so the new communication module has to provide the same flags with the correct timing. The other only task, is to receive as usual every row of input monitoring data and to load each 16-bit data of them into the *Sink* FIFO, that will be the only one

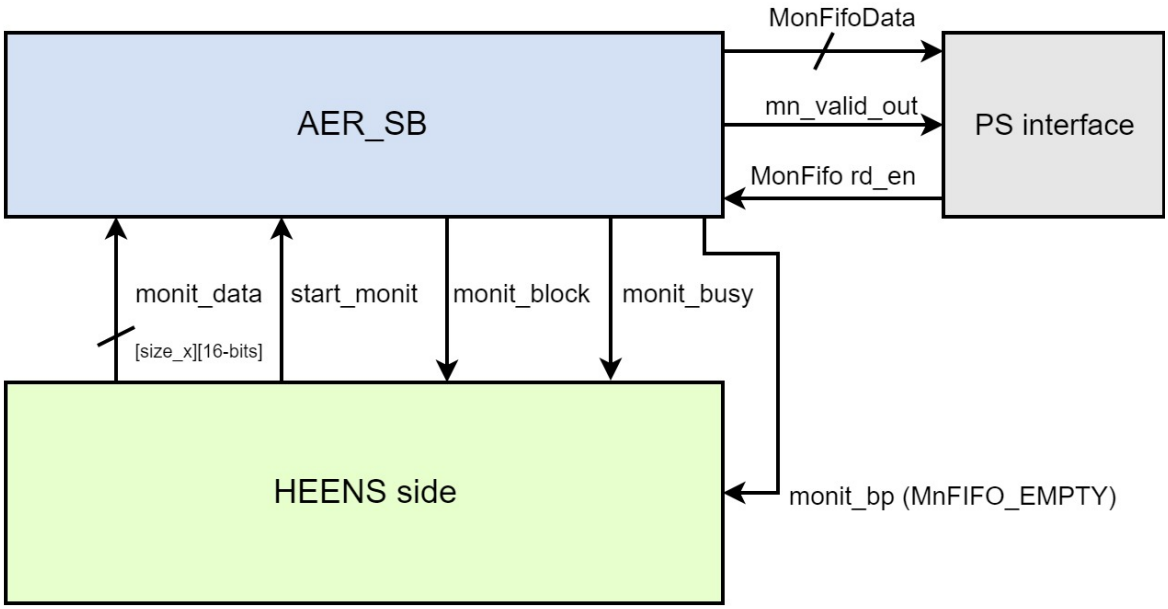


Figure 2.33: Single-board version

present in this version to accumulate information for the PS interface (it is the only monitoring FIFO utilized).

Basically, inside the *AER_SB* unit, there is the same controller present in the *Z_AER_interface* of Section 2.3.1, but this time, until the controller finishes writing all the rows inside the FIFO, the module will set *monit_busy* high, in order to provide the first semaphore signal for the sequencer. The other locking/unlocking signal is simply generated by the *empty* flag of the *Sink* FIFO, in order to signal when all monitoring data have been sent to the PS interface.

2.4.2 PS interface reading operation

The only last issue to discuss in this section, is the few hardware components that have been created to handle the reading operation performed by the ARM processor, so by the external world. In Figure 2.34 the TD of this operation is reported.

In the diagram, *MonFifoRdPS* is the enable signal from the PS interface, while *resume_MonRd* and *cntrl_mn_rd* are control flags. Basically, it is important to check if the FIFO is empty before the read flag signal in a generic situation goes to zero. By doing this, the final output valid flag will be set or not to '1' when the read signal returns to one (which means that PS wants to read again), in order to avoid a wrongly reading of the same data for two consecutive times. The schematic of this hardware is reported in Figure 2.35

The VHDL source file relative to the AER one board module, with only the

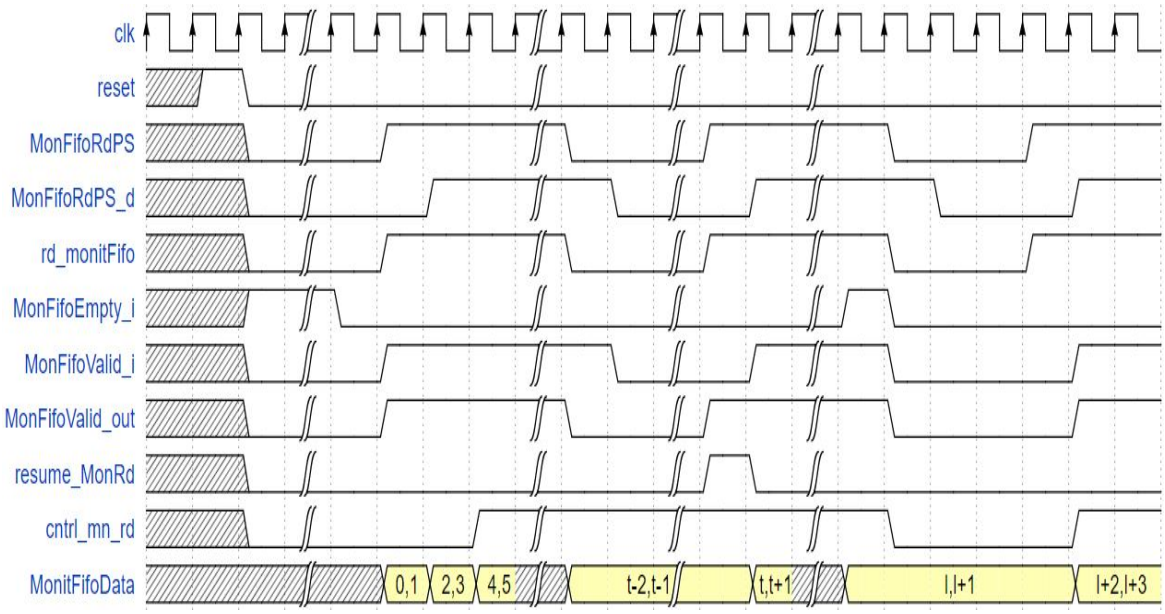


Figure 2.34: Timing Diagram of the PS reading operation

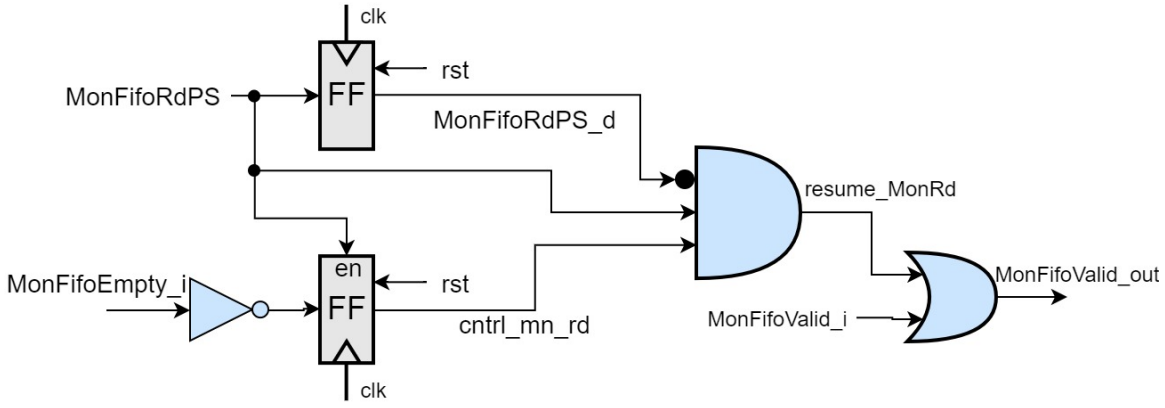


Figure 2.35: Hardware components for PS reading operations

monitoring controller (for simplicity), can be found in Appendix D.7. The simulation that confirms the correct output monitoring data from the *Sink* FIFO is reported in Figure 2.36

In this simulation, a fake read enable signal has been created and a process that regularly set this signal to one and zero for a fixed number of cycles too, in order to verify if the reading process works fine. The first cursor on the left indicates when the FIFO was empty when the read enable flag went to zero, and so later the output valid signal is not set to '1', while the second cursor provides the opposite example. It is shown how data are correctly sent and validated by the *AER_OneBoard* unit. Same

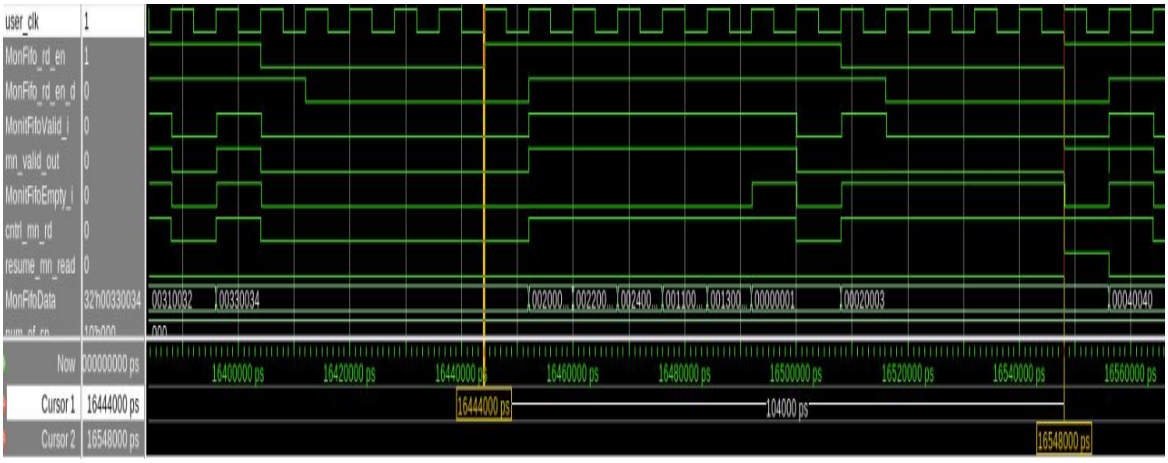


Figure 2.36: Simulation of the reading operations performed by the PS interface

hardware and protocol have been introduced in the Master Chip of the previously mentioned multi-board version, since it is in charge of communicating with the PS interface.

Of course, an empty flag should be brought out, in order to signal the external reading controller that there are data available in the *Sink* FIFO, but this will be done when the PS module will be available.

2.5 Logic Synthesis and Hardware Implementation

After the simulations, the next step of the design is the synthesis and implementation on the already mentioned *Xilinx Zynq-7000 SoC ZC706* board, shown in Figure 2.37.

It is a *System on Chip* device, used to exploit both the software programmability of an ARM-based processor and the hardware programmability of the SoC FPGA, integrated in the same architecture. Indeed, the *Zynq-7000* Soc family, provides the user with both these kind of devices in the same board, in order to have better configurability and monitoring qualities, together with low power, better integration and higher bandwidth characteristics. and furthermore, the size of this kind of board is not too heavy. The tool used for these purposes, is the *Vivado* software tool.

2.5.1 Single-Board

The single board version has been tested first, on a 5x5 array configuration. The first step is the logic synthesis, in which all VHDL source files and their hierarchical connections are compiled by *Vivado* and the timing constraints applied are verified. In Figure 2.38 the clock signals utilized in this project are showed.

Figure 2.37: Xilinx Zynq-7000 SoC ZC706.

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

Figure 2.38: Clock Summary of the single board synthesis and implementation

A period of 5 ns has been applied to the clock input *clk_in_p*, then it's directly linked to a Mixed-Mode Clock Manager to obtain a local generated clock at the desired frequency of 125 MHz for the HEENS architecture. This is the only clock source exploited, since in this version, there is no needing of a dedicated AER clock source. An input *jitter* uncertainty of 0.05 ns has been set on this latter source and finally a *reset* input has been created with a delay of 0.1 ns respect to the clock.

After synthesis, the timing report summary was generated and it is reported in Figure 2.39. It is shown the *setup slack* obtained from the difference between the *data required* time, which is the time that the clock takes to travel all the path to destination sequential element and the *data arrival* time, which is the time required for the data to reach the destination, starting from the instant in which the rising edge of the source clock occurred. The slack is positive, which means that the design is able to work at the desired frequency.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.690 ns	Worst Hold Slack (WHS):	-0.091 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-91.021 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	1006	Number of Failing Endpoints:	0
Total Number of Endpoints:	38064	Total Number of Endpoints:	38064	Total Number of Endpoints:	11853

Figure 2.39: Timing report summary of the single board synthesis

On the contrary, the total *hold slack* is negative, showing that the hold time is not respected: this issue is fixed with a specific option offered by the implementation tool, since the hold requirements can be resolved adding specific logic buffers or gates, in order to delay a possible change of a signal on the data path that links two sequential elements. Nevertheless, this fix has not to deteriorate the *setup* slack, which mast be kept under control.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.005 ns	Worst Hold Slack (WHS):	0.026 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	38204	Total Number of Endpoints:	38204	Total Number of Endpoints:	11923

All user specified timing constraints are met.

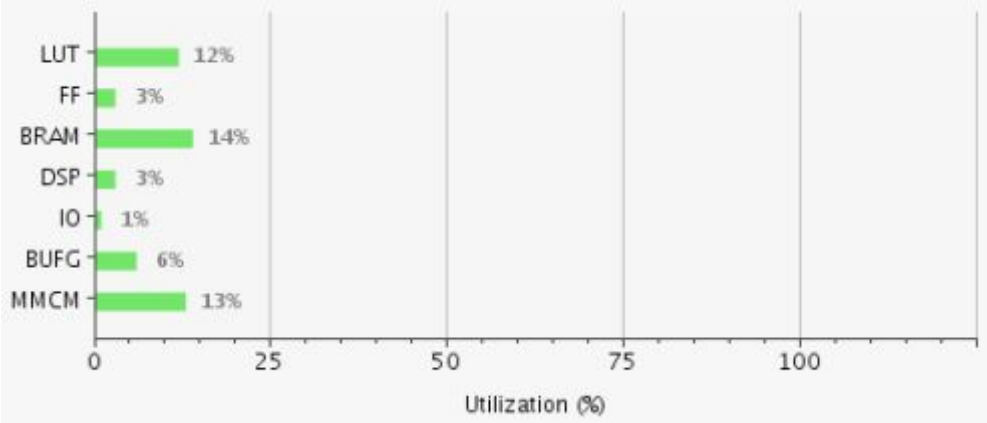
Figure 2.40: Timing report summary of the single board implementation

Then, the implementation has been performed and for this task, the tool had to place all gates on the FPGA, create all connections (*Place and Route*) and it has to apply some optimizations, in order to reduce some path delays or the power consumption of the hardware.

From Figure 2.40 the timing report shows that all *hold* issues have been fixed by the implementation process, while in Figure 2.41, a report regarding the resources utilization is illustrated.

The *BRAM* memories are the most area consuming blocks and it is possible to notice form the figure below, that the *AER_OneBoard* unit, in which most of the hardware added for monitoring operations have been introduced, is a very low critical module from an area occupancy point of view, while the PE-array is the greatest consumer of resources.

In Figure 2.42 the power consumption is reported. The dynamic power is mostly dominated by the *BRAM* memories and the clock manager, since of course it has to propagate the main clock source to all the architecture. This measures are only a rough estimate, since the power consumption is strongly dependent on the activities of



Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (1093 0)	F8 Muxes (54650)	Slice (54650)	LUT as Logic (218600)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSP s (90 0)	Bonded IOB (362)	IBUFD S (348)	BUFGCTRL (32)	MMCME2_ADV (8)
SNN_OneBoardTop	12.25%	2.68%	1.35%	0.18%	14.79%	12.25%	3.05%	14.22%	2....	0.83%	0.2...	6.25%	12.50%
seq_inst (sequencer)	0.17%	0.07%	0.00%	0.00%	0.22%	0.17%	0.08%	0.00%	0....	0.00%	0.0...	0.00%	0.00%
clock_inst (clk_wiz_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.2...	6.25%	12.50%
array_inst (PE_array)	11.92%	2.52%	1.35%	0.18%	14.38%	11.92%	2.90%	13.76%	2....	0.00%	0.0...	0.00%	0.00%
BRAM_seq_inst (BRA...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	0....	0.00%	0.0...	0.00%	0.00%
AER_OneBoard_inst (...	0.15%	0.08%	0.00%	0.00%	0.24%	0.15%	0.06%	0.37%	0....	0.00%	0.0...	0.00%	0.00%

Figure 2.41: Resources utilization of the 5x5 single board implementation.

the resources.

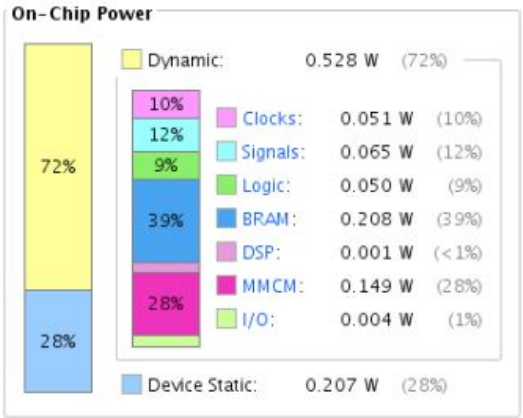


Figure 2.42: Power report summary of the single board implementation.

Finally, in Figure 2.43 the resulted *floorplanning* is showed, from which it is possible

to notice that the area exploited by the design is not critical and that a bigger array configuration can be implemented, as it will be done in the next chapter.

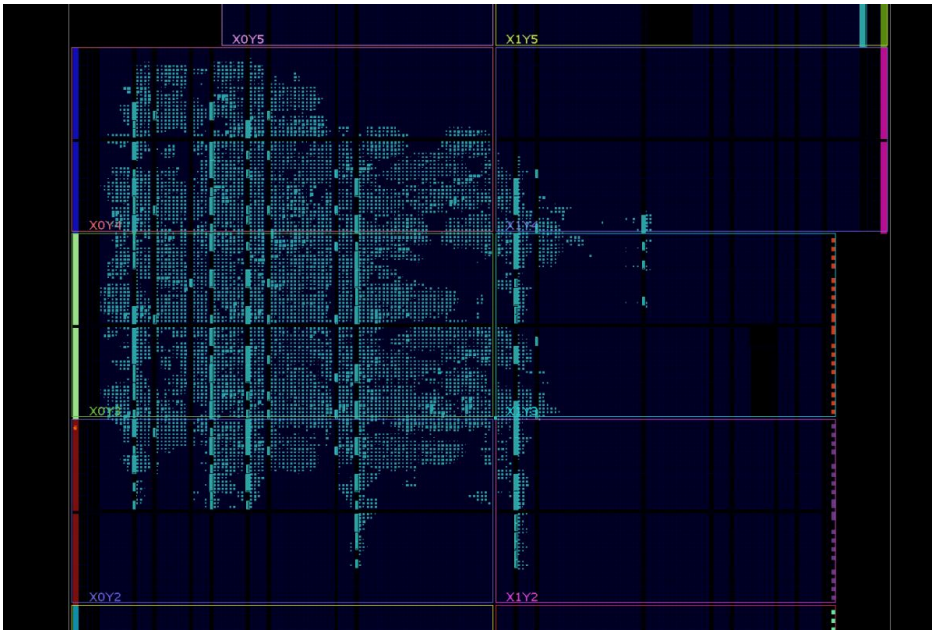


Figure 2.43: Floorplanning of the of the single board, 5x5 array implementation.

2.5.2 Multi-Board

Regarding the multi-board version, the implementation is more complicated: in addition to the HEENS specific hardware, it is also composed by the structure of the *Aurora* core and all the AER modules, to allow the serial communication between different chips. Furthermore, in the single-board version, the mif files were loaded inside the synthesized memories of the project, while in this case (that will be the final version of the whole design), the configuration files are directly loaded, as already mentioned, by the PS interface (ARM processor).

Unfortunately, this latter module is under development by the research group, so accurate results from an implementation point of view cannot be obtained at the current state, especially those regarding the verification of timing constraints: indeed, many control input signals are coming from the PS interface and so many path delay are determined by that. Therefore, the synthesis and implementation operations for this kind of architecture have been principally carried out to analyze the area occupation overhead introduced by the monitoring hardware and to check if at least the new data paths respect the setup time of the project.

In Figure 2.44 the clocks generated by the tool after the synthesis are reported.

Name	Waveform	Period (ns)	Frequency (MHz)
GT_REFCLK1	{0.000 4.000}	8.000	125.000
INIT_CLK_P	{0.000 2.500}	5.000	200.000
clk_out1_clk_wiz_0	{0.000 4.000}	8.000	125.000
clk_out2_clk_wiz_0	{0.000 10.000}	20.000	50.000
clkfbout_clk_wiz_0	{0.000 2.500}	5.000	200.000
example_design_1_i/z_aer_top_i/aurora_mo...	{0.000 2.000}	4.000	250.000

Figure 2.44: Clock Summary of the multi-board synthesis and implementation

It is shown that in this case, two kind of input clock are necessary: the first of 200 MHz is dedicated to the *Aurora* core, while the second is is going to a clock wizard that generates two clocks of 125 MHz and 50 MHz, for the HEENS and for other specialized module of the *Aurora* part respectively. The AER clock characterized by a frequency of 250 MHz is directly generated by the *Aurora* module, which has a specialized block to perform this task.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.046 ns	Worst Hold Slack (WHS):	-3.013 ns	Worst Pulse Width Slack (WPWS):	0.970 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-192.117 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	1440	Number of Failing Endpoints:	0
Total Number of Endpoints:	41393	Total Number of Endpoints:	41313	Total Number of Endpoints:	13678

Figure 2.45: Timing report summary of the multi-board synthesis

The timing report of Figure 2.39 shows again hold issues that can be fixed by the implementation process, while the worst *setup* slack (the critical path) has a very low value of 0.046 ns. This latter, is due to those signals that come from the Z_AER_tx module and so, they are generated with the timing set by the AER clock. Then, as explained in previous sections, those signals are synchronized by means of the component of Figure 2.24, which means a clock domain change. An example of this kind of path is showed in Figure 2.46.

In Figure 2.47 the general resource utilization is reported.

Again, the *BRAM* memories are greatly used and respect with the single-board version, there is an increment of the utilization of LUT and global buffers (BUFG). The latter, are used in clock modules and generator in order to reduce the skew between registers that are physically located large distances apart.

In Figure 2.48, some details of the resources employed in the project are shown: it is possible to notice that all the hardware related to the AER modules is much less then the resources allocated for the array (HEENS) unit. So, it does not represent a critical issue from an area point of view.

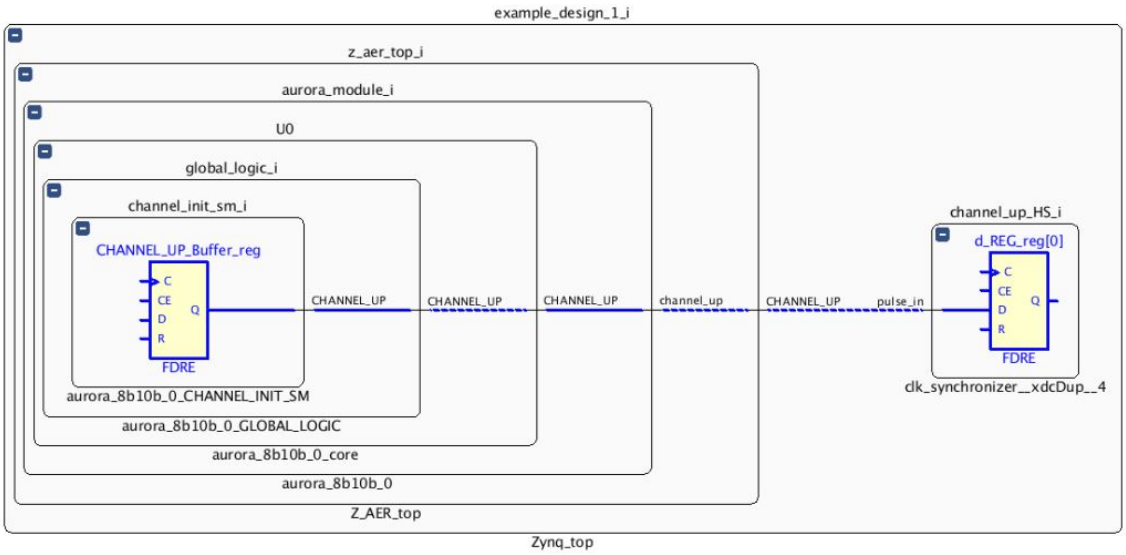


Figure 2.46: One of the critical paths of the multi-board synthesized version

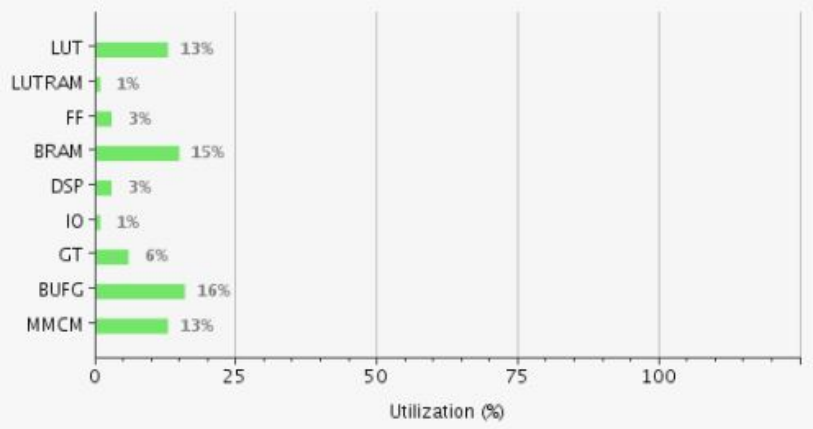


Figure 2.47: Resources utilization summary of the multi-board architecture

The power report of the multi-board implementation is reported in Figure 2.49, in which it is shown that the total power consumed by the architecture (*static* + *dynamic*) is more or less the same respect to the previous single board version. That means all the *Z_AER* modules don't consume an excessive amount of power and the *Aurora* core is probably well optimized for this device. Again, the clock network, composed also by the MMCM, is the biggest responsible for this dynamic power consumption.

Finally the *floorplanning* of the multi-board implementation is showed in Figure 2.50, from which it is possible to notice again a poor exploitation of the whole available area.

Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (109300)	F8 Muxes (54650)	Slice (54650)	LUT as Logic (218600)	LUT as Memory (70400)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSPs (900)	Bonded IOB (362)	Bonded IPADs (50)	IBUFDS (348)	GTXE2_COMMON (4)	GTXE2_CHANNEL (16)
ZynqKintexTop	12.88%	3.08%	1.36%	0.18%	15.83%	12.87%	0.03%	3.33%	14.68%	2....	1.10%	4.00%	0.2...	25.00%	6.25%
clock_inst (clk_wiz_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.2...	0.00%	0.00%
example_design_1_i ...	12.88%	3.08%	1.36%	0.18%	15.83%	12.87%	0.03%	3.33%	14.68%	2....	0.00%	0.00%	0.0...	25.00%	6.25%
array_inst (PE_array)	12.01%	2.53%	1.35%	0.18%	14.29%	12.01%	0.00%	2.90%	13.76%	2....	0.00%	0.00%	0.0...	0.00%	0.00%
bp_monit_HEENS...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
BRAM_seq_inst (B...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
busy_monit_HEEN...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
channel_up_H5_i (...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
clk_syn1 (clk_sync...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
en_monit_AER_i (cl...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
enFIFO_AER_i (clk_...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_config_HEENS...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_dist_HEENS_i (...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_exec_AER_i (cl...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_init_HEENS_i (p...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_tx_data_HEENS...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
MasterTask_i (Mas...	0.04%	<0.01%	0.00%	0.00%	0.08%	0.04%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
PhaseController_i (...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
reset_HEENS_i (clk...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
seq_inst (sequencer)	0.18%	0.07%	0.00%	0.00%	0.26%	0.18%	0.00%	0.08%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
z_aer_top_i (Z_AE...	0.34%	0.24%	<0.01%	0.00%	0.70%	0.33%	0.03%	0.19%	0.18%	0....	0.00%	0.00%	0.0...	25.00%	6.25%
zaer_interface_i (Z...	0.27%	0.22%	0.00%	0.00%	0.58%	0.27%	<0.01%	0.13%	0.64%	0....	0.00%	0.00%	0.0...	0.00%	0.00%

Figure 2.48: Resources utilization details of the multi-board architecture

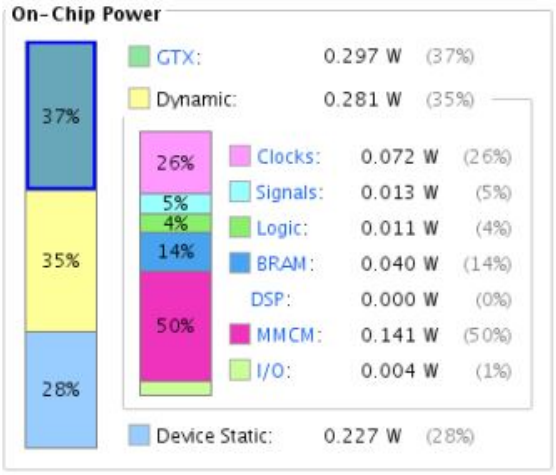


Figure 2.49: Power report summary of the multi-board implementation

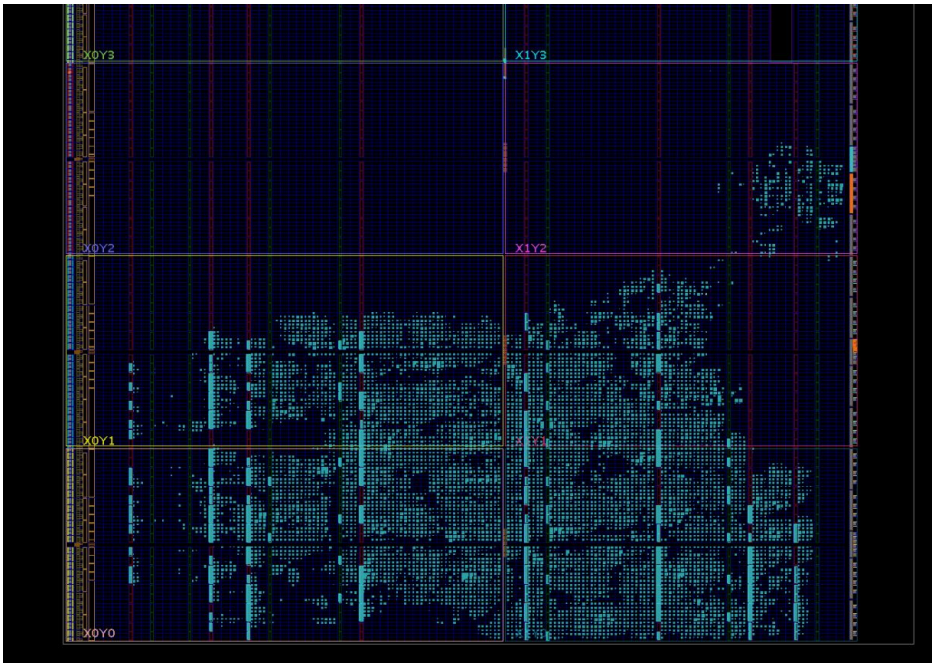


Figure 2.50: Floorplanning of the of the multi board, 5x5 array implementation.

Chapter 3

Performance upgrading

In this chapter, a better solution for monitoring is proposed. The intended task is to better exploit the parallelism of the PE-array and the higher clock frequency of the AER structure, in order to speed-up the propagation and also the transmission of those data.

The main differences are here reported and commented, but the same detailed level of the first chapter is not adopted for simplicity, while performances results and comparison between the first solution are more focused in the chapter.

3.1 Architecture improvements

The significant bottleneck of the previous version, derives from the fact that if a *STOREB* instruction is fetched and the monitoring data of a previous stage are still been propagating through the array or for example, the AER transmitter modules have not finished yet to transmit those data to the ring, the new instruction is not decoded and the sequencer stops the execution of the algorithm, as already explained. This is not an optimal solution, since the monitoring FIFO is loaded only with data of a single *STOREB* instruction and its size is not exploited.

In the previous version the size of the FIFO could be adapted to store only a fixed number of data for each monitoring request (13x13 data of 16 bits maximum allowed at the current state), which would lead to a safe in terms of hardware resources, but as it was explained in the final sections of Chapter 2, FIFOs don't represent a critical issue from an area occupancy point of view.

So, the idea is to block the sequencer only if the PE-array has not finished yet to propagate monitoring data up to the final FIFOs. In this way, even if the transmitter is still sending the information of its own chip through the serial communication bus, another instruction is allowed to starting loading data and the sequencer would not stop.

The other bottleneck is due to the additional waiting that derives from the loading of single 16-bit data into the monitoring FIFO, which does not exploit the parallel nature of the PE-array. The ideal would be to employ one clock cycle at most for each row of monitoring data and load in parallel a number of inputs equal to the number of PEs in a row of the array (*size_x*). These concepts are illustrated in Figure 3.1, where the solution to improve performances is graphically described.

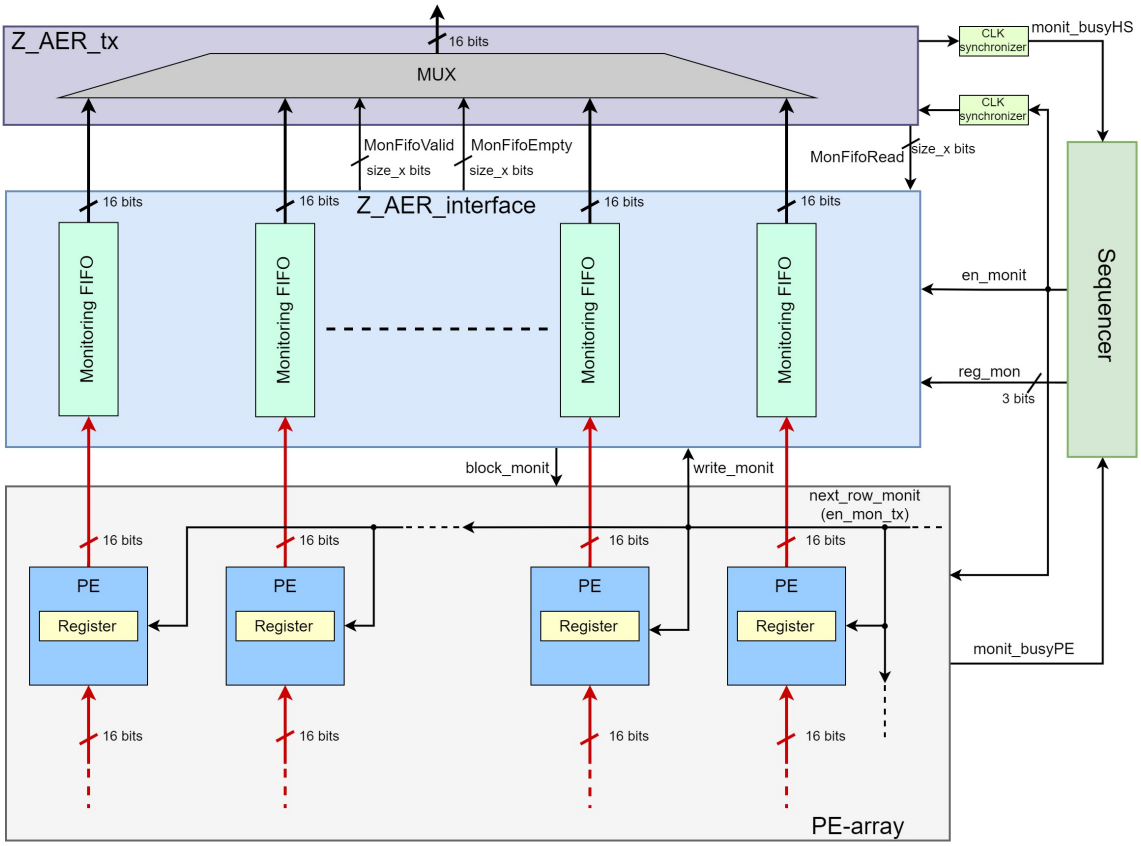


Figure 3.1: Upgraded monitoring architecture

From the schematic, it is clearly shown that no matter how many PEs are present in the array, because each column is getting its own monitoring FIFO, which size this time has been reduced to 256 words each, in order not to add too much area overhead. Anyway, with this configuration applied to the actual maximum PE-array size (13x13 PEs), considering to use also all the eight levels of virtualization, it is possible to store in these FIFOs exactly all data that come from two consecutive monitoring instructions plus three virtual level information of a new one, before filling the FIFOs.

In this architecture, the interface is in charge of writing all data of a row inside all FIFOs in one clock cycle, while the transmission part has to apply a multiplex

operation to their outputs, in order to send the final 16-bit monitoring data to the final TX multiplexer. All changes will be briefly discussed in the next sections, the only module unchanged is the *Z_AER_rx*, which is not reported again for simplicity.

3.1.1 Z_AER_interface & PE-array

The functionalities of the PE-array monitoring controller are similar to the previous version, except for the fact that now it provides the sequencer with the *monit_busy* flag, that is in charge of stopping the execution of the algorithm, in case another *STOREB* instruction is fetched while the previous monitoring data are still being loaded into the FIFOs. The *Z_AER_interface* module now has to write all data inside the FIFOs and stop the propagation in case even just one of them gets full.

Furthermore, in this version, the four unused bits of the *START_MON* packet (Section 2.3.2) are now exploited to send information about which register is being monitored, to ease the classification work on these data performed by the final general purpose unit. So, the *reg_mon* signal (3 bits) is sent to the interface by the sequencer, to keep track of this information. The combined TDs of these two modules are reported in Figure 3.2 and the remarkable changes respect to the basic version are described below.

1. The interface controller now has a state in which the register number information is written into a monitoring FIFO. Basically, in this stage the first data of the first FIFO on the left (arbitrary choice) is loaded with the *reg_mon* data, while in the others all zeros are loaded, since the rest of the row data are not useful. The *din_monitfifo* signal, represents the row and it's marked in the TD with the information that it's carrying (like *monit_out* for the other FSM).

2. This time, after a full condition ends, data are not immediately written into the FIFOs, as well as the other rows are not propagated in the following clock cycle: this, creates a behaviour closer to a *Moore* FSM, even if the two signals, *next_row_monit* and *wr_monitfifo*, that propagates and writes respectively the monitoring row of data, still needs to changes immediately if a full condition happens, in order to respect the right timing.

3. In the PE-array controller, the counter this time is aligned with the effective row outputted by the array. Furthermore, in this version, this controller is in charge of setting the *busy_monitPE* flag (in the previous version it was *busy_monit*),in order to stop the sequencer. It is set to zero when the array output the third last row. This has been done, to release the main CU at the right time in order not to waste clock cycles: indeed, after the sequencer is released, the *STOREB* instruction will be decoded, then executed (*en_monit* is set to one in the diagram) and after that, new monitoring data will be available at the output of each PE.

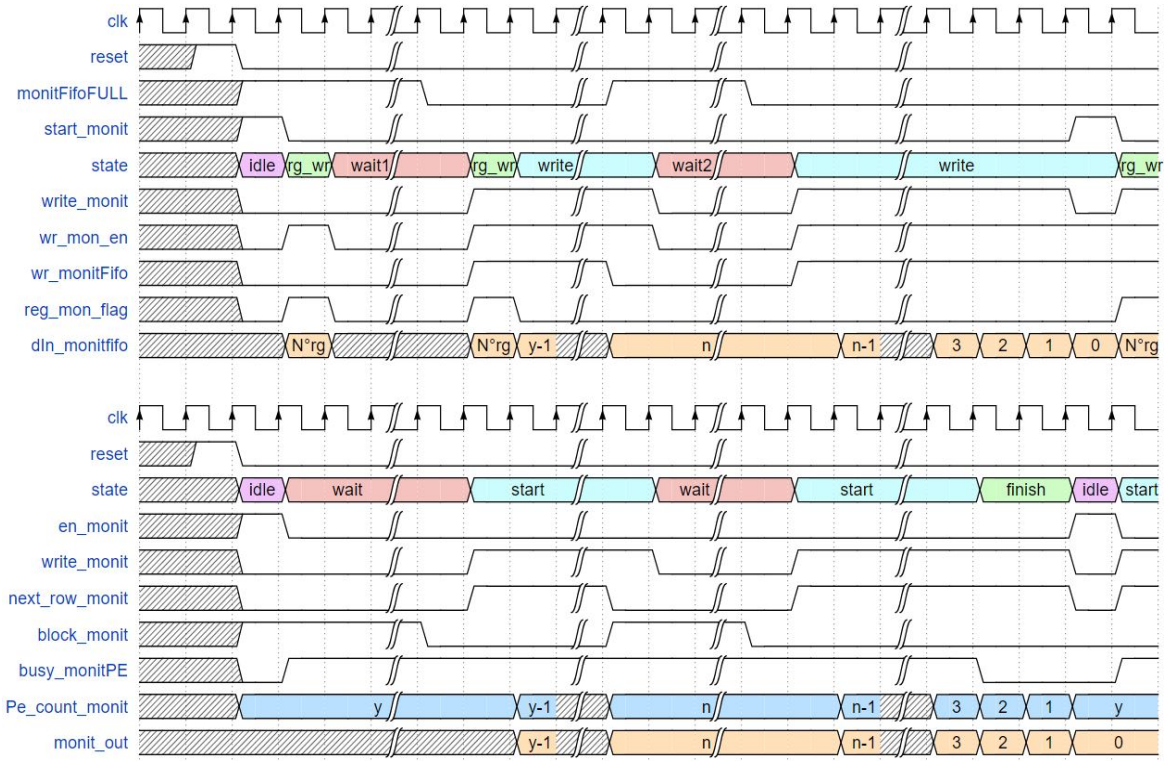


Figure 3.2: Timing Diagrams of the PE-array (down) and *Z_AER_interface* (up) monitoring controllers

Meanwhile, the last rows of the previous instruction are loaded and the new register number too, as it is shown in the upper diagram of Figure 3.2.

These are the main differences between the two modules, in which all necessary components have been designed to follow the behaviour of the TDs. The only upgraded parts of the VHDL source files are reported in Appendix D.8 and D.9, where are reported only the upgraded blocks or components with respect to previous versions.

3.1.2 Z_AER_tx

In this upgraded version, this unit is in charge of multiplexing the outputs of the monitoring FIFOs coming from the *Z_AER_interface* and also to set the proper read signals, to get the correct inputs at the right time. To accomplish this goal, a single FSM that is similar to the basic version has been created to manage the read enables. By means of using a couple of counters, the right inputs (monitoring data, valid and empty flags) are selected and sent to the Aurora TX module as *tx_d_mn* and *tx_src_rdy_mn* (Section 2.3.2).

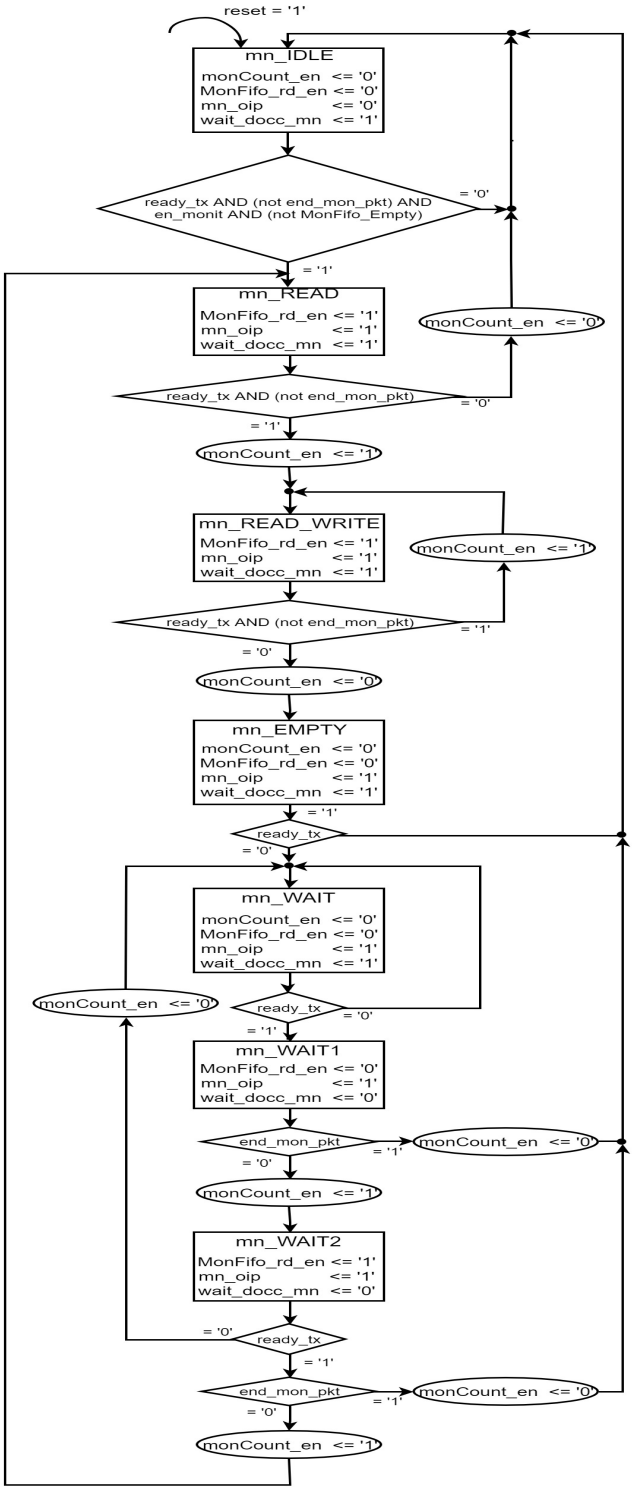


Figure 3.3: Flowchart of the *Z_AER_tx* monitoring controller (second version).

Before that, initially the register number under monitoring is picked and sent by means of the *START_MON* packet, as it was previously mentioned. The new FSM of this latter controller is not discussed for simplicity, but it is straightforward enough to be understood from the VHDL source file in the Appendix.

In Figure 3.3, the flowchart of the algorithm adopted to read and transmit monitoring data is showed. Basically, employing the already mentioned counters, read enable flags are handled one by one: when a data of a FIFO is being transmitted, the read enable flag of the next one is set, in order to have the data ready in the next transmission cycle. For these tasks, the counter enable is managed in a *Mealy* way but, since all signal are synchronized, this does not create problems and it is necessary in order to not switch the output to chose (incrementing the counter) in case the bus is not ready (*ready_tx* = '0') or all data belonging to the actual packet have been transmitted (*end_mon_pkt* = '1').

This time a more reliable method is used to signal the end of the monitoring data transmission of a chip, through the *end_mon_pkt*: this flag, along with the counters, is able to mark when the correct number of data have been transmitted ($size\_x \times size\_y$), so basically emulates what the empty flag does in the basic version.

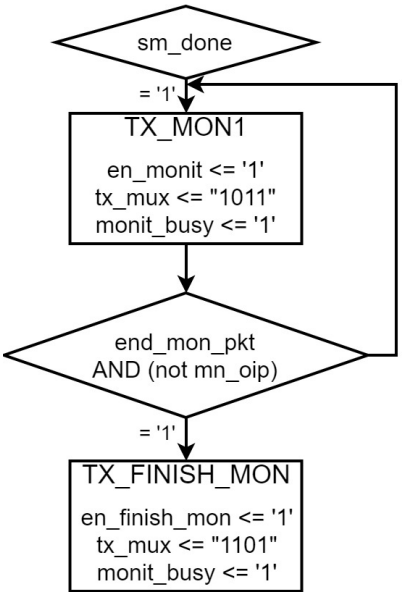


Figure 3.4: Changed monitoring states of the *Z_AER_tx* main FSM

Thanks to that, even if the monitoring FIFO is not empty because a new *STOREB* instruction may have loaded its relative data inside of it, the TX module moves to the next state in which the data of the rest of the network are transmitted or accumulated into the *SinkFIFO* (NC or MC respectively). This new behaviour is showed in the extract of the main state machine in the *Z_AER_tx* unit of Figure 3.4 and it is

necessary to transmit to the ring only data from a specific *STOREB* instruction in the right order and so to avoid a mixing of information that would complicate the work of the final general purpose unit.

The flowchart of Figure 3.3 presents an additional waiting state (*mn_WAIT2*), that is needed when, after a certain period in which the serializer Aurora TX module was not ready to accept new data, the transmission starts again and the previous monitoring data have to be transmitted: the consequence is similar to the previous version, except to the fact that in this case, there are two data to transmit of which the valid signal is not high anymore, which leads to the necessity of the other waiting state (Section 2.3.2, Figure 2.17). The data path of this new controller is reported in Figure 3.5.

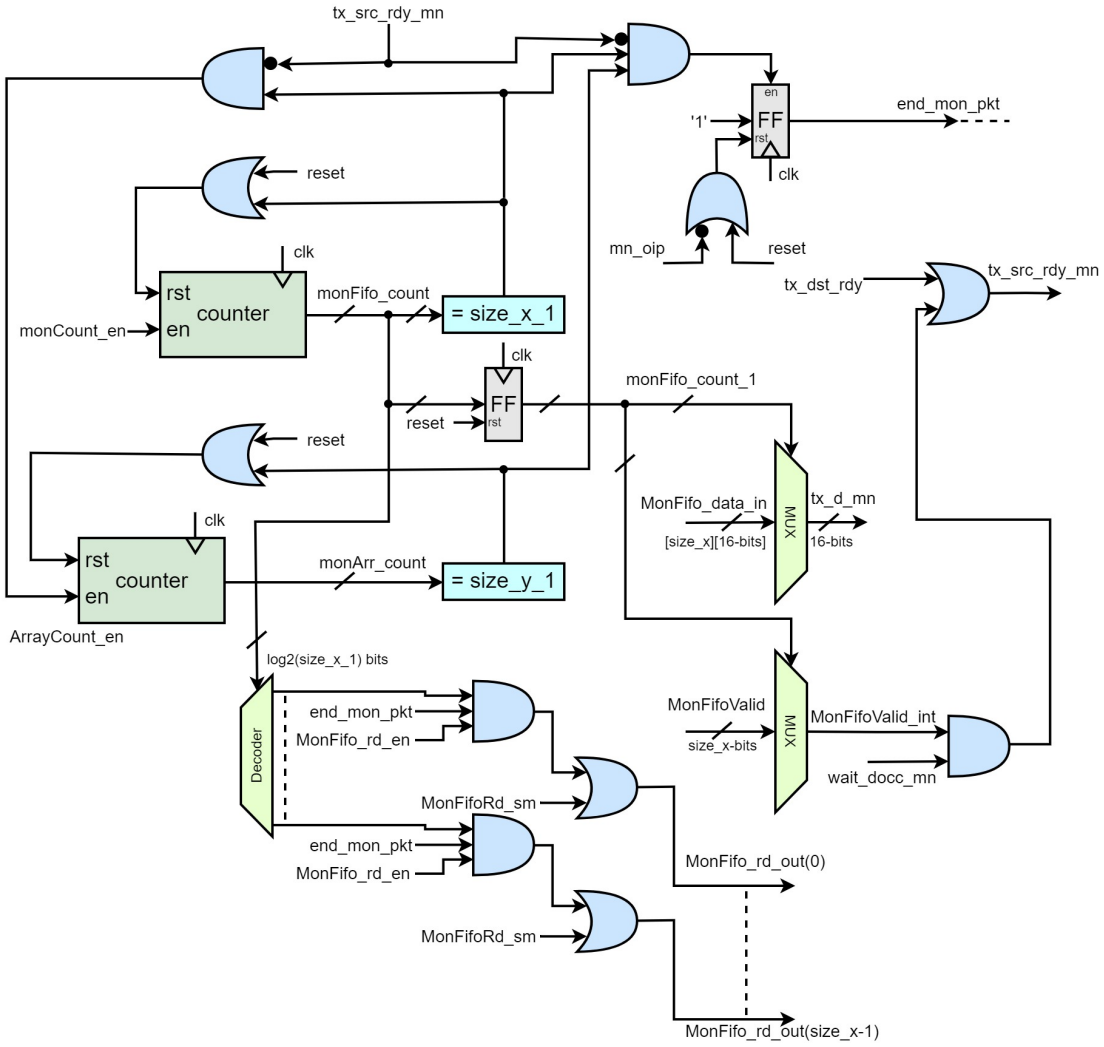


Figure 3.5: Datapath of the *Z_AER_tx* monitoring controller (second version)

It is shown how the read enables are set, in which it must be underlined that the signal *MonFifoRd_sm* (generated by the start monitoring FSM) sets all flags to one, in order to read the register number information and to discard all other zeros loaded in the rest of the first row (Section 3.1.1). As in the previous modules, the VHDL source file containing only the upgraded parts of the transmitter in Appendix D.10.

3.1.3 Sequencer

Very few modifications have been made to the sequencer in this upgraded version. The "semaphores" strategy is still adopted, but this time, as already mentioned, these latter are unlocked in a different way. The new portion of the algorithm of the main FSM in the sequencer is reported in Figure 3.6.

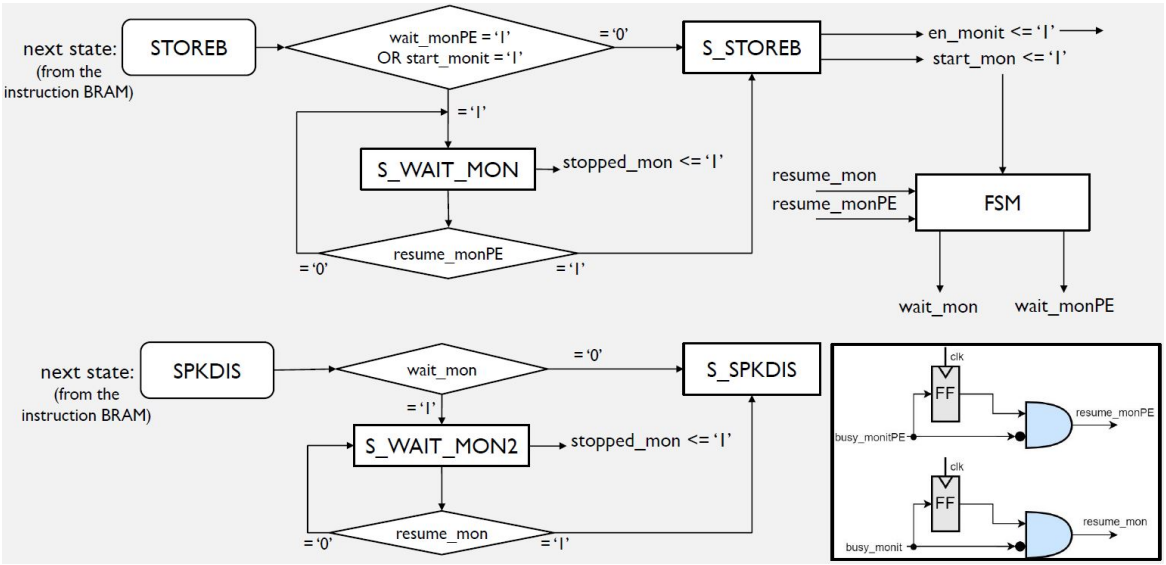


Figure 3.6: Monitoring states of the upgraded main FSM in the sequencer unit

This time, the PE-array is in charge of setting to zero the *busy_monitPE* signal, to unlock the first semaphore (*resume_monPE*). Furthermore, by checking the *start_monit* flag, (that goes to one in the decode stage of a *STOREB* instruction), it is possible to have two *STOREB* in a row. This latter functionality has been added to the FSM of the previous version (Figure 2.25) as well.

In this version, as already explained, the *Z_AER_tx* module is in charge of unlocking the second semaphore and it is worth to mention that it keeps locked the sequencer for all the monitoring transmission and reception phases. So, the main CU will be allowed to enter in the spike distribution stage only when all data from a specific monitoring operation will be transmitted, received, when they come from other chips, and finally sent to the other nodes (NC) or loaded into the *SinkFIFO* (MC).

3.1.4 Simulation

In this section interesting extracts from the simulation are being provided, in order to verify the different behaviour of the upgraded version with respect to the previous one (Section 2.3.5). Again the algorithm proposed in Appendix B.3 is used and in this version the second monitoring instruction is useful to verify if the register number is well propagated, by means of the *START_MON* packet.

- *Z_AER_interface & PE-array*

  The simulations of the two combined controllers belonging to the AER interface and PE-array units, are illustrated in Figure 3.7.

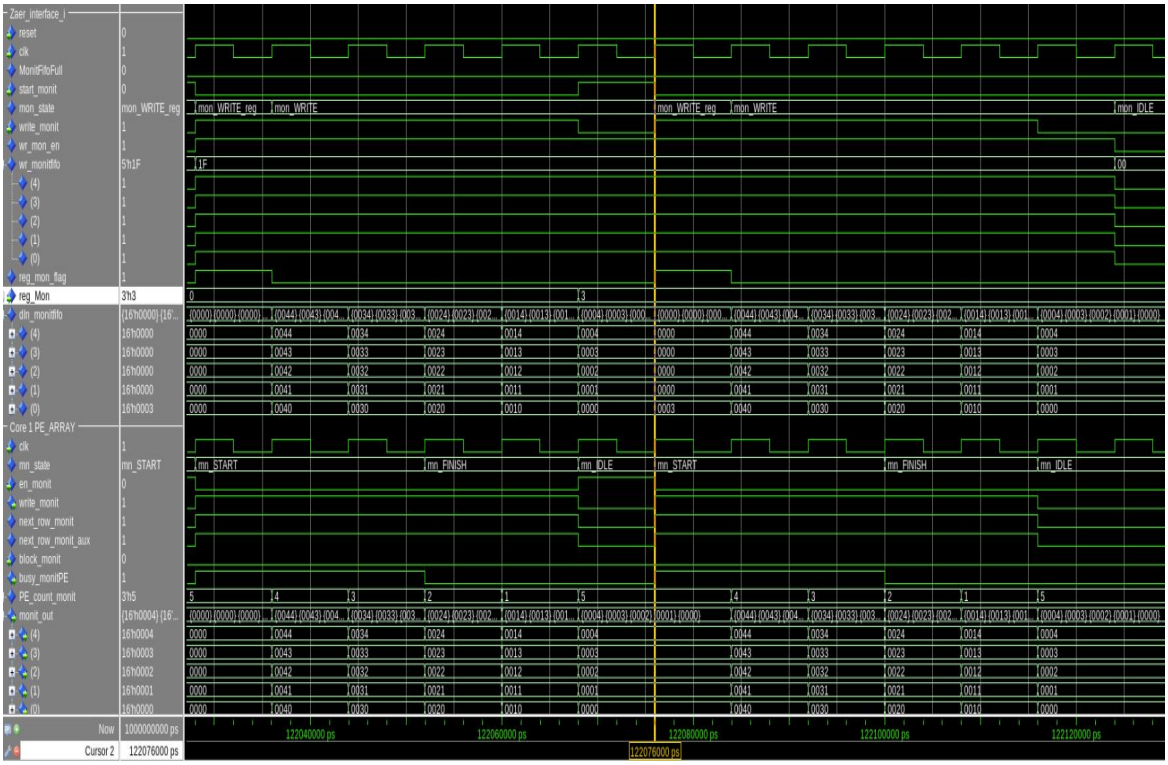


Figure 3.7: Simulation of the monitoring controllers in the *Z_AER_interface* and PE-array modules (second version)

The same signals of the TD in Figure 3.2 are reported in the snapshot and even if some interesting occurrences studied in that diagram are not showed (since the *MonitFifoFull* flag never goes to one), the main behaviour is respected in this picture: it is possible to notice that, this time, in each clock cycle an entire row is loaded by the interface, which was the original goal of this version.

It is also possible to notice the *reg_Mon* signal that carries the information

relative to the register under monitoring, that is equal to 3 the second time the *en_monit* flag goes to '1': indeed, in the assembler code of Appendix B.3 a *MONIT R3* instruction is performed. This latter information is written during the *mon_WRITE_reg* state in the first monitoring FIFO, while the other locations of the same row are loaded with all zeros as expected.

- *Z_AER_tx*

  For this module, two simulation scenes are studied. The first one is reported in Figure 3.8.

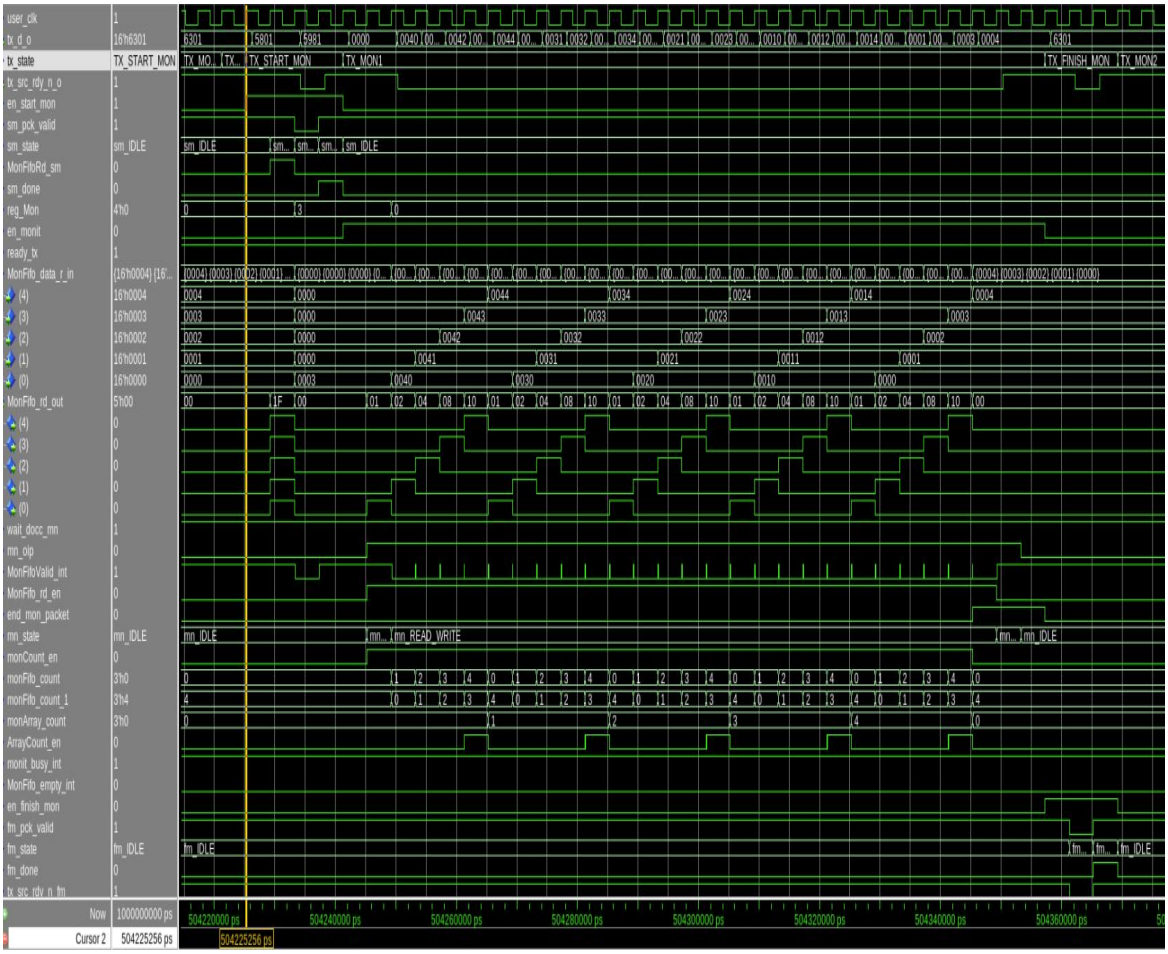


Figure 3.8: First simulation of the monitoring controller in the *Z_AER_tx* module (second version)

The simulation offers a snapshot of the main stages regarding the transmission of the monitoring data that belong to the chip. At the beginning, the *START_MON* packet is sent and the figure shows that it corresponds to the hexadecimal value

"5981" ("0*b*0101100110000001"), while the start monitoring packet showed in Figure 2.29 was "0*x*5*B*01" ("0*b*0101101100000001"): this is due to the four unused bits that, in the previous version, were set to "0*b*0110" and now they represent the current monitored register number, which is the fourth (0*b*0011).

Then the *TX_MON1* phase starts and the screenshot shows how read enables are correctly set in the proper order. Indeed, all data from each row of every FIFO are sequentially read, a part from the first row that is read in one clock cycle, since it contains the register number and all zeros. Again, the data transmitted in these simulations are the PEIDs, that identifies the array position of the PE in the array (*row,column*). After this stage, the *FINISH_MON* packet (unchanged) is sent and finally, the bypass phase starts.

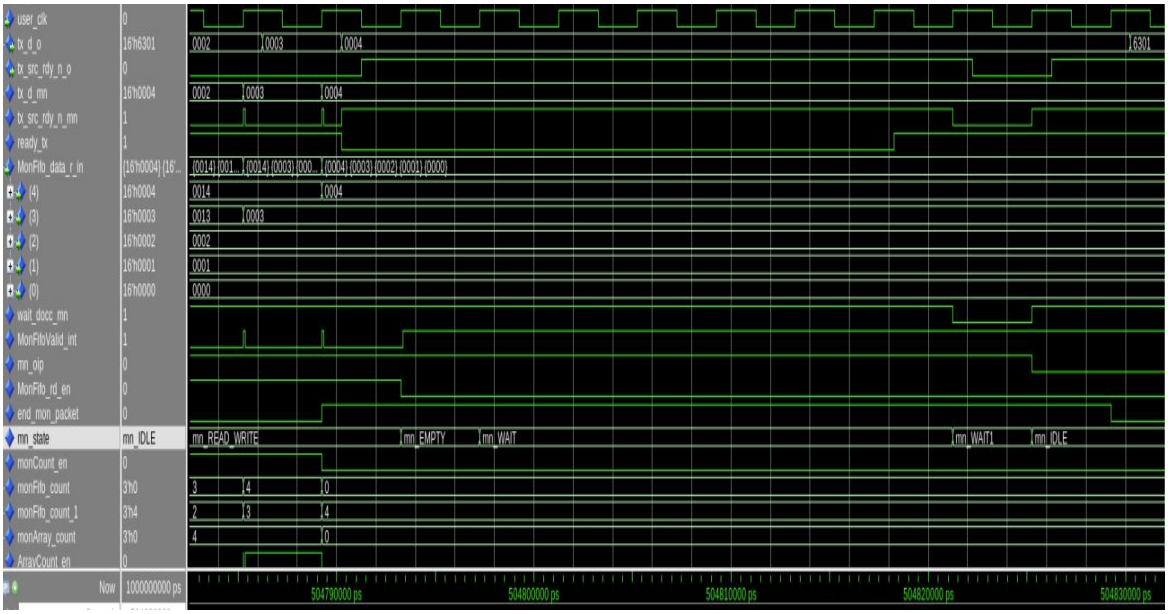


Figure 3.9: Second simulation of the monitoring controller in the *Z_AER_tx* module (second version)

In Figure 3.9 an interesting eventuality has been captured: the *ready_tx* flag, that indicates when the serial transmitter is ready to accept new data, goes to zero exactly when the last data of a packet has to be transmitted (*end_mon_pkt* is set to '1'). This event is correctly handled by the controller that, after the transmission is allowed again, manages to send the last data (in the *mn_WAIT1* state) and then to move in the *IDLE* state.

- *Sequencer*

  The functionalities of the sequencer, as already discussed, has undergone few changes. In Figure 3.10 is depicted an extract of the moment in which the CU is

stopped because of two near *STOREB* instructions. As shown, the next monitoring operation is allowed after five clock cycles, that are due to the propagation of the five rows and an additional cycle in which data are issued by PEs at the beginning.

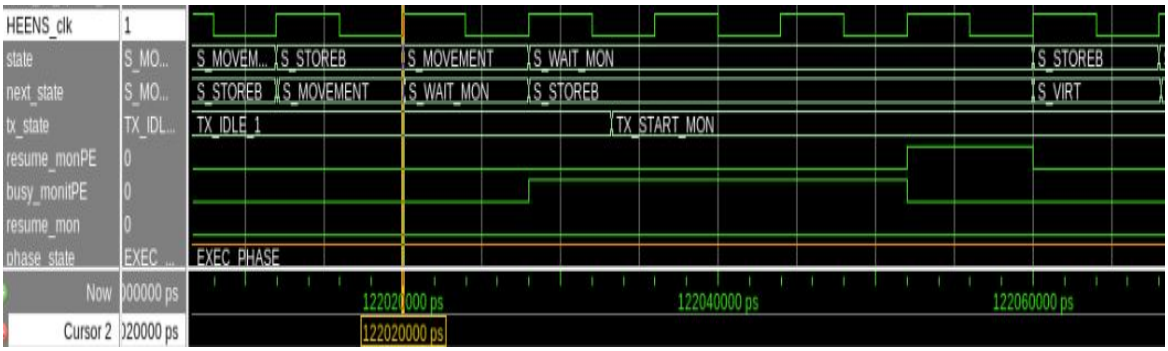


Figure 3.10: First simulation of the waiting phases of the CU (second version)

Instead, the second simulation in Figure 3.11 shows the waiting state of the sequencer, right before the spike distribution phase. It is interesting to notice from the screenshot that the AER transmitter, after it finishes to send monitoring data from the ring, goes back to the *TX_MON1* state, since it has pending data from the last *STOREB* instruction to transmit. Indeed, before unlocking the CU and moving to the *IDLE* state, the TX module checks if the monitoring FIFO is empty or not.

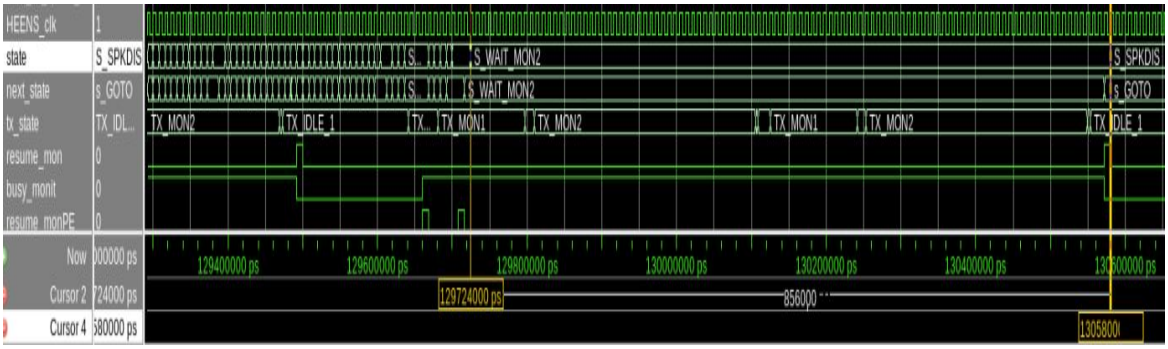


Figure 3.11: Second simulation of the waiting phases of the CU (second version)

3.1.5 Single Board

The architecture developed to the single-board version has been upgraded as well. The challenging part of this variant, was collecting two monitoring data at the time, to transmit them to the PS interface module. Indeed, as already discussed in Section

2.4.2, the data path of the ARM processor works with 32 bits and consequently, all the 16-bit outputs coming from monitoring FIFOs had to be ordered and arranged in a different format. Thus, the monitoring controller that has been designed for the *AER_oneBoard*, is similar to the one in the *Z_AER_tx* described in a previous section, but this time, for the reading operation, two data and two read enable flags are picked and set respectively.

The number of the register monitored is sent to the outside, but all the details of this structure are not reported as the previous modules for simplicity, since this variant does not represent the important core of this work. Indeed, as already said, it is only used to verify in a simple way (without exploiting all the *AER* multi-board modules) the correctness of information transmitted by the HEENS architecture. A simplified schematic of the *AER_OneBoard* unit, is reported in Figure 3.12.

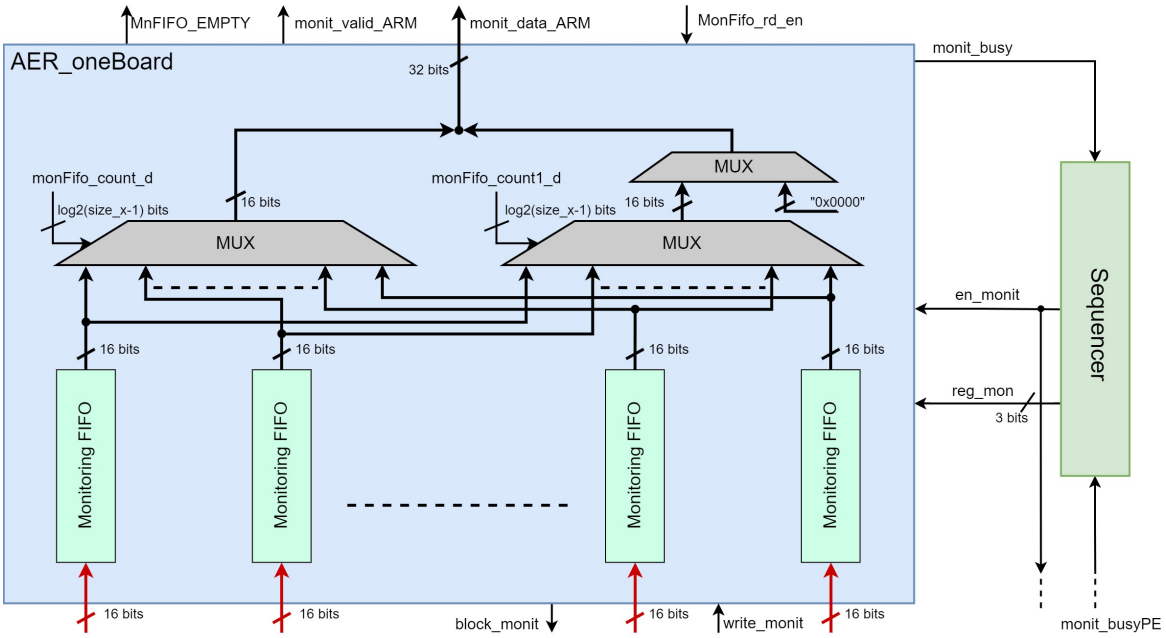


Figure 3.12: Block diagram of the upgraded single-board AER module

This diagram does not illustrate all the data path exploited to allow the transmission of monitoring data stored in the several FIFOs, but it explains the basic principle underlying this mechanism: two data are picked at the time, so two multiplexers are needed and two specific selection signals as well, that are *monFifo_count_d* and *monFifo_count1_d*. Like in the previous version, these latter are generated by two separated counters that, together with the *MonFifo_rd_en* flag (from the ARM processor), are used to select the proper inputs to transmit and they are also exploited to set the read enables for the FIFOs (not reported in the figure).

Another important issue is that in this version the *monit_busyPE* (the first

semaphore) is handled by the PE-array, like in the multi-board architecture, and the *monit_busy* is simply the result of a *NAND* operation between all the empty flags from monitoring FIFOs: indeed, only when all of them are empty (all empty flags equal to one) the transmission is over and the *monit_busy* can return to zero, in order to unlock the sequencer. The monitoring controller used to write into the FIFOs is the same as the one in the *Z_AER_interface* module and it works with the PE-array in the same way.

The part of VHDL source file that describes the monitoring controller of this single board version, is reported in Appendix D.11, in which it is possible to notice that a very similar strategy (FSM) to the upgraded multi-board version has been adopted to handle the transmission, even if it is more complicated, due to the reasons just explained. The other strategy could have been to store all data in a classic *Sink* FIFO (16 bits input, 32 bits output), but it would have required more hardware.

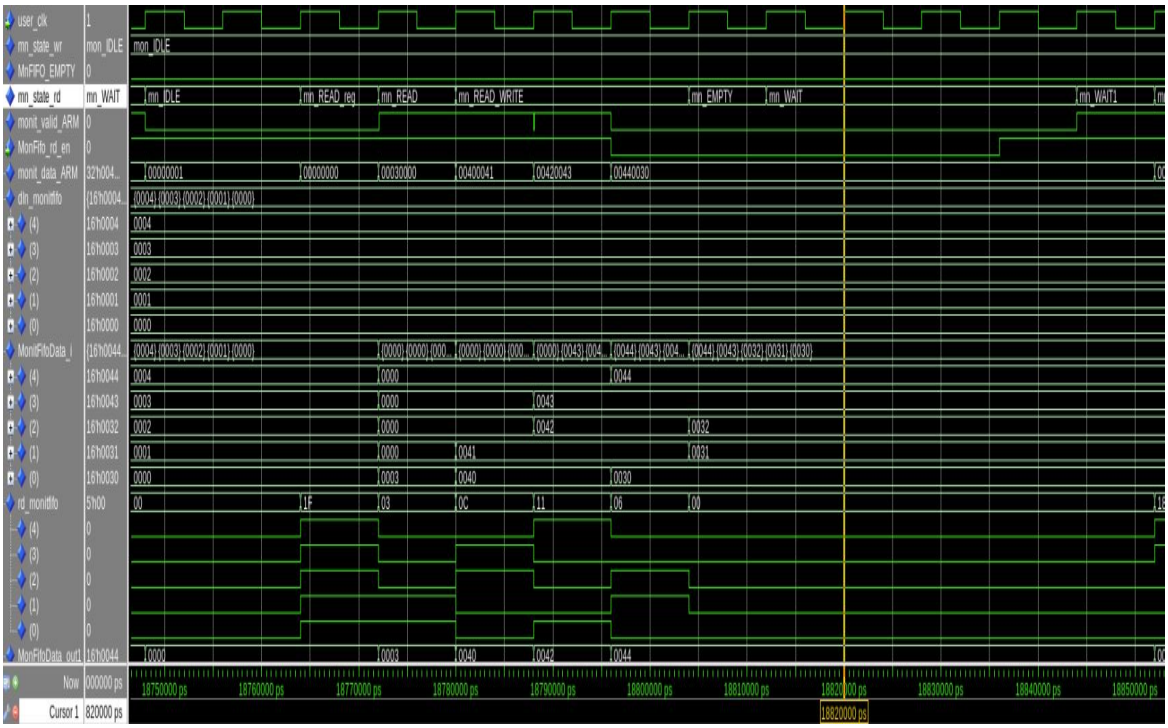


Figure 3.13: Simulation of the monitoring controller in the *AER* single board module (second version)

In Figure 3.13 is reported an interesting frame from the simulation of the upgraded single board operations related to the monitoring stages. Indeed, it is depicted what happens between two monitoring transmissions (related the two instructions of the assembler code in Appendix B.3): first, the *mn_READ_reg* state is performed, in which all read enable flags are set to '1', in order to read the register number. This latter

information is sent together with all zeros (in the multi-board version, it was carried by the *START_MON* packet, Section 3.1.2), as it is shown by the *monit_data_ARM* data.

Then, all read enables and outputs data from monitoring FIFOs are set and selected in pairs, as expected. Again, only for debug purpose, a fake *MonFifo_rd_en* flag is set to '0' and to '1' at regular intervals, to verify the behaviour: after a period in which this latter signal is set to zero, it is shown that the last information is correctly validated (*monit_valid_ARM*, active high) and consequently transmitted by the AER module. So, basically the behaviour of the *SinkFIFO* is correctly reproduced, by means of this more complex controller.

3.2 Logic Synthesis and Hardware Implementation

The synthesis and implementation operations have been performed to the upgraded version on the *Xilinx Zynq-7000 SoC ZC706* board as well. This time, the single-board architecture has been tested with a 13x13 array configuration, to analyze the exploitation of the area of such a large structure, while the multi-board approach has been tested with the same 5x5 configuration, to highlight the differences with respect to the older version of the monitoring implementation.

3.2.1 Multi-Board

Regarding the multi-board synthesis and implementation, apart from the changed modules of the architecture, the main difference is the monitoring FIFO, whose size has been reduce has already mentioned. The clock network and constraints have not been changed. The synthesis at the beginning produced a particular violation related to the *monit_busy* flag, which comes from the *Z_AER_tx* module, described in Section 3.1.2: it is produced by the main FSM of the AER transmitter module and then it is sent to a synchronizer, for the HEENS 125 MHz clock domain. The synthesis tool introduced a particular LUT between these two modules, which is shown in Figure 3.14 and which creates a setup violation reported in Figure 3.15.

This issue has been resolved introducing a register to break this path, which did not create timing problems, since it regards the flag which unlocks the sequencer from a waiting state, as already described. So, the only consequence of this delay is one clock cycle more of waiting before moving to the spike distribution phase.

After the implementation has been performed, only the area and power results are analyzed, since the timing results are not reliable at the current state of the project, as already mentioned in the first chapter. In Figure 3.16 the general resource utilization is reported, which shows that the situation is only slightly changed from the previous version (of Figure 2.48) and the *Z_AER_interface* is the module that has undergone the largest increase in the hardware exploitation.

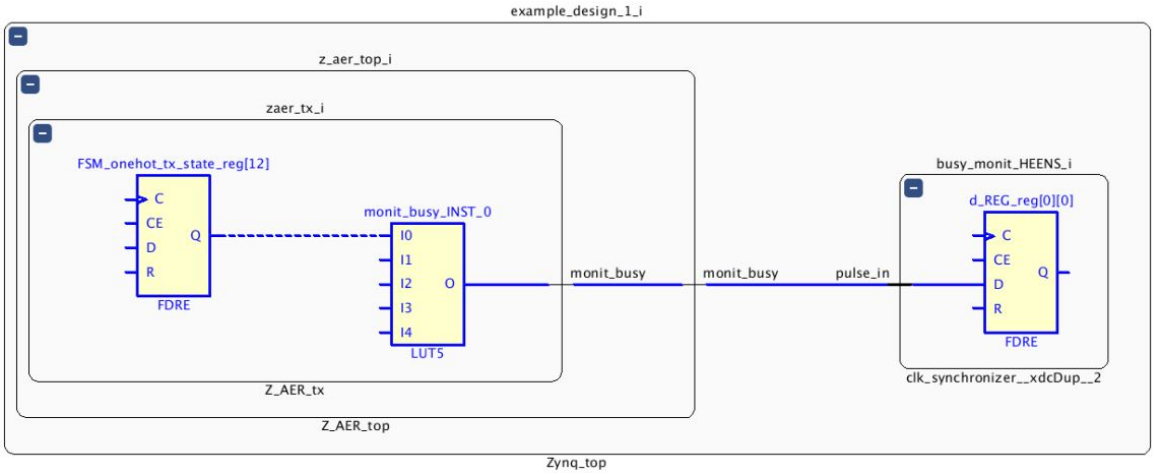


Figure 3.14: Critical path of the 5x5 multi-board array synthesis (second version).

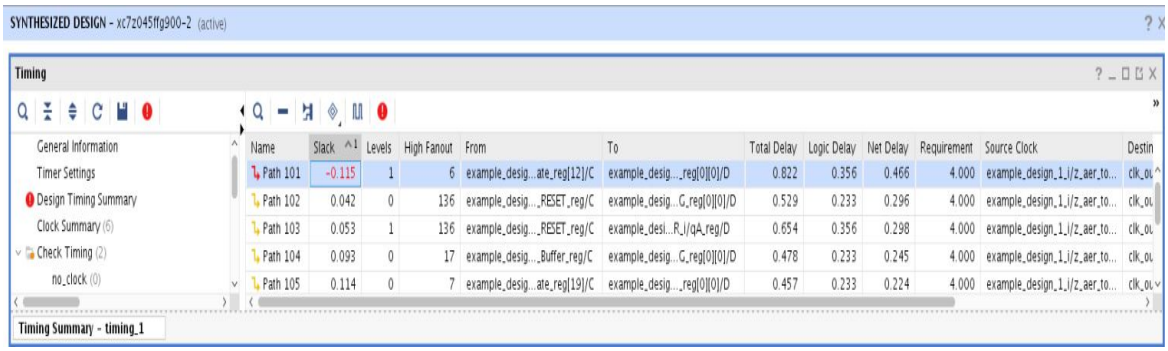


Figure 3.15: Critical path delay of the 5x5 multi-board array synthesis (second version).

The power consumption is reported in Figure 3.17, which shows a 5.3 % increase in dynamic power with respect to the previous version. The *floorplanning* does not show remarkable changes, so it is not reported for simplicity.

3.2.2 Single-Board

For the single-board structure, the results obtained from the synthesis are similar to the previous version, so the implementation outcomes are directly analyzed from now on. Figure 3.18 illustrates the timing report of the final implementation of the single-board version, in which it is shown that the timing constraints are all respected, even if with a minor *setup* slack respect to the previous version.

In Figure 3.19, the resources utilization are reported, from which it can be noticed that this time, the LUTs and *BRAM* memories occupy almost all the available hardware.

Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (10930)	F8 Muxes (54650)	Slice (54650)	LUT as Logic (218600)	LUT as Memory (70400)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSPs (900)	Bonded IOB (362)	Bonded IPADs (50)	IBUFDS (348)	GTXE2_COMMON (4)	GTXE2_CHANNEL (16)
ZynqKintexTop	13.06%	3.22%	1.35%	0.18%	16.11%	13.05%	0.03%	3.42%	15.05%	2....	1.10%	4.00%	0.2...	25.00%	6.25%
clock_inst (clk_wiz_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.2...	0.00%	0.00%
example_design_1_i ...	13.06%	3.22%	1.35%	0.18%	16.10%	13.05%	0.03%	3.42%	15.05%	2....	0.00%	0.00%	0.0...	25.00%	6.25%
array_inst (PE_array)	12.00%	2.53%	1.35%	0.18%	14.27%	12.00%	0.00%	2.90%	13.76%	2....	0.00%	0.00%	0.0...	0.00%	0.00%
BRAM_seq_inst (B...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
busy_monit_HEEN...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
channel_up_HS_i (...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
clk_syn1 (clk_sync...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
en_monit_AER_i (cl...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
enFIFO_AER_i (clk_...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_config_HEENS...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_dist_HEENS_i (...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_exec_AER_i (cl...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_init_HEENS_i (p...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
eo_tx_data_HEENS...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
MasterTask_i (Mas...	0.04%	<0.01%	0.00%	0.00%	0.08%	0.04%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
PhaseController_i (...	<0.01%	<0.01%	0.00%	0.00%	<0.0...	<0.01%	0.00%	<0.01%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
reset_HEENS_i (clk...	0.00%	<0.01%	0.00%	0.00%	<0.0...	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
seq_inst (sequencer)	0.17%	0.07%	<0.01%	0.00%	0.26%	0.17%	0.00%	0.08%	0.00%	0....	0.00%	0.00%	0.0...	0.00%	0.00%
z_aer_top_i (Z_AE...	0.36%	0.24%	<0.01%	0.00%	0.75%	0.35%	0.03%	0.19%	0.18%	0....	0.00%	0.00%	0.0...	25.00%	6.25%
zaer_interface_i (Z...	0.43%	0.35%	0.00%	0.00%	0.88%	0.43%	<0.01%	0.23%	1.01%	0....	0.00%	0.00%	0.0...	0.00%	0.00%

Figure 3.16: Resources utilization summary of the multi-board architecture (second version).

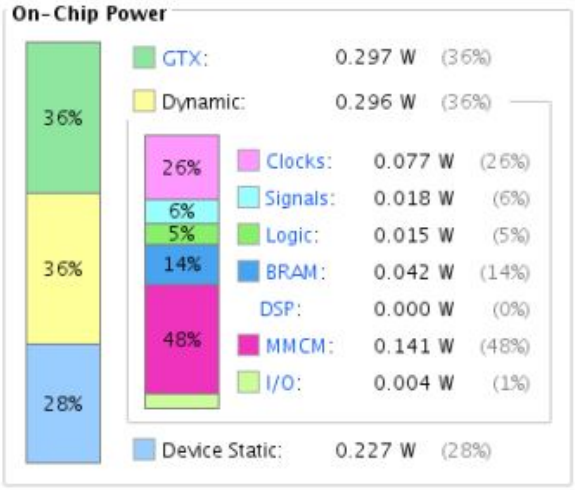


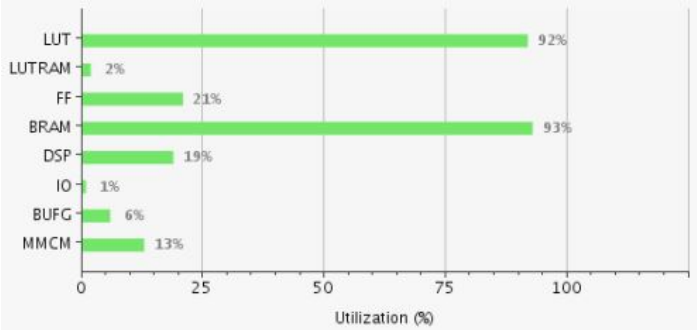
Figure 3.17: Power report summary of the multi-board implementation (second version).

Again, looking at the second detailed resources report, it is possible to see that the array structure is the most consuming module of the architecture, so the overhead introduced by the monitoring implementation (monitoring FIFOs, control logic, multiplexers and so on) doesn't impact so much on the FPGA occupancy. Anyway, it is also clear that it is not possible to further extend the array dimension at the current state of the project.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.039 ns	Worst Hold Slack (WHS):	0.004 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	287020	Total Number of Endpoints:	287020	Total Number of Endpoints:	94490

All user specified timing constraints are met.

Figure 3.18: Timing report summary of the 13x13 single board array implementation (second version).



Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (109300)	F8 Muxes (54650)	Slice (54650)	LUT as Logic (218600)	LUT as Memory (70400)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSPs (900)	Bonded IOB (362)	IBUFDS (348)	BUFGCTRL (32)	MMCME2_ADV (8)
SNN_OneBoardTop	92.10%	21.04%	10.21%	0.96%	97.80%	91.57%	1.63%	27.74%	93.39%	1...	0.83%	0.2...	6.25%	12.50%
AER_OneBoard_inst (...	1.18%	0.30%	0.00%	0.00%	1.74%	0.66%	1.63%	0.32%	0.28%	0....	0.00%	0.0...	0.00%	0.00%
array_inst (PE_array)	90.84%	20.71%	10.20%	0.96%	96.83%	90.84%	0.00%	27.38%	93.03%	1...	0.00%	0.0...	0.00%	0.00%
BRAM_seq_inst (BRA...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	0....	0.00%	0.0...	0.00%	0.00%
clock_inst (clk_wiz_0)	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0....	0.00%	0.2...	6.25%	12.50%
seq_inst (sequencer)	0.06%	0.02%	<0.01%	0.00%	0.12%	0.06%	0.00%	0.01%	0.00%	0....	0.00%	0.0...	0.00%	0.00%

Figure 3.19: Resources utilization of the 13x13 single board array implementation (second version).

In Figure 3.20, the power consumption is explored. It is shown, that the dynamic power is 4.03 times greater, which is mainly due again to the *BRAM* memories and clock network. With this configuration also the dynamic power consumed by the logic and signal interconnection has more than doubled, since of course more LUTs are required and also the interconnections are much longer then the smaller 5x5 configuration.

Finally the *floorplanning* of the 13x13 array is shown in Figure 2.43, in which it is shown that almost all of the available area is exploited by this implementation.

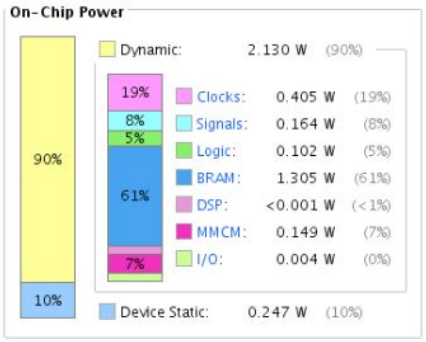


Figure 3.20: Power report summary of the 13x13 single board array implementation (second version).

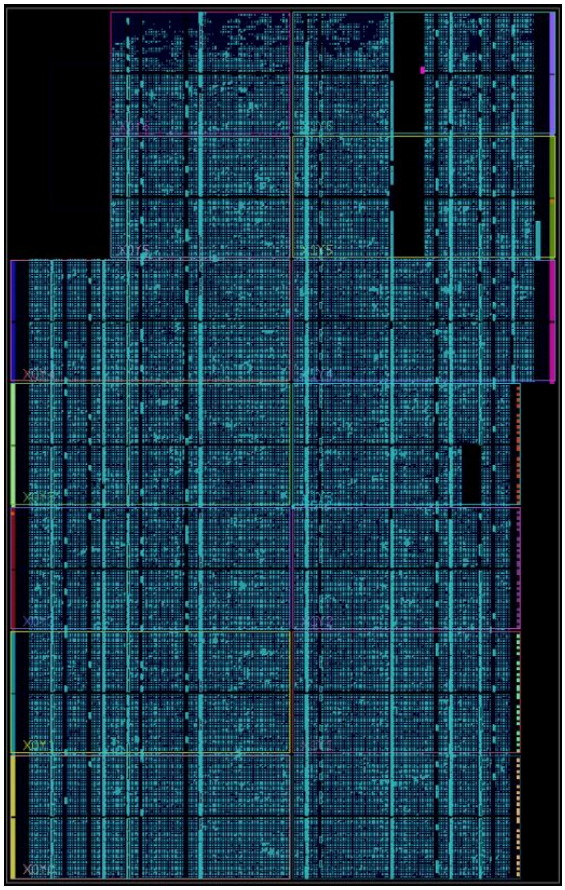


Figure 3.21: Floorplanning of the 13x13 single board array implementation (second version).

Chapter 4

Conclusions

This thesis work is focused on the HEENS architecture, which is an SNN emulator belonging to the 3^{rd} generation and meant to be implemented on a FPGA device, which provides the user with an high level of configurability, scalability and capability of emulating different neural models.

The task of this work, was to design all the hardware support to collect and distribute specific data and parameters inside each PE, in order to monitor the behaviour of the neural algorithm under execution. This is a very important goal, since it is necessary to keep under control the information related to all the neurons of the network, in order to chose or modify a particular model, which can better meet the needs of a particular application. Therefore, after designing and simulating the additional architecture, by means of VHDL and *Questasim Advanced simulator* tool, the project has been synthesized and implemented on the *Xilinx Zynq-7000 SoC ZC706 board*, in order to verify if the structure properly worked.

After the first solution, the hardware for the monitoring operations has been enhanced to increase performance of the procedure, which led to a small overhead in terms of area exploitation and power consumption. In the first version, the amount of clock cycles required to propagate all monitoring data through the array was proportional to $N \times N$, where N is the number of PEs in a row or column of the array. After the upgrading, this time has been reduced to be linearly dependent on just N, which leads to a great speedup. Indeed, the clock frequency of the HEENS architecture is the half respect to the AER part of the system and thus, reducing the overall number of clock cycles required to load data into monitoing FIFO (action performed with the HEENS clock), led to great improvements in the speed of the operation.

The continuation of this work is the development of a software environment to properly organize and eventually display all collected data, which are transmitted to a general processing unit by the PS interface (*ARM* processor inside the FPGA). Indeed, a GUI will be developed by the research group to allow the user to configure and monitor the SNN running on the FPGA.

Another possible upgrading of the architecture designed in this work, could concern a specific filtering of information coming from the array, in case not all PEs (neurons) of it are utilized. Indeed, the current monitoring hardware is going to transmit data coming from all PEs and leave to the future software application the task of selecting the information of a specific neuron chosen by the user. This possible improvement would provide overall faster transmission and additionally require little control logic.

Appendix A

Instruction Set Architecture

	Instruction	Group	Opcode	Function	Format
0	NOP	SEQ	000000	No operation	NOP
1	LDALL	REGISTERS	000001	reg <= DMEM (from sequencer)	LDALL reg ****
2	LLFSR	MOVEMENT	000010	ACC <= LFSR(15:0)	LLFSR
3	LOADSP	LOADSP	000011	R1 & ACC(15:1) <= BRAM(BP,31:1); ACC(0) <= spike_register(BP(3:0))	LOADSP
4	STOREB	STOREB	000100	EXT_BUFFER <= ACC	STOREB
5	STORESP	STORESP	000101	BRAM(BP) <= R1 & ACC; BP <= BP + 1	STORESP
6	STOREPS	STOREPS	000110	AER_FIFO <= ACC(0) (post-synaptic Si)	STOREPS
7	RST	REGISTERS	000111	reg <= "0000"	RST reg
8	SET	REGISTERS	001000	reg <= "FFFF"	SET reg
9	SHLN	REGISTERS	001001	ACC <= ACC << n, (1 <= n <= 8), (n = number of positions)	SHLN n
10	SHRN	REGISTERS	001010	ACC <= ACC >> n, (1 <= n <= 8), (n = number of positions)	SHRN n
11	RTL	REGISTERS	001011	ACC <= ACC <<, carry = ACC(msb) Rotate Accumulator Left	RTL
12	RTR	REGISTERS	001100	ACC <= ACC >>, carry = ACC(lsb) Rotate Accumulator Right	RTR
13	INC	ARITHMETIC	001101	ACC <= ACC + 1	INC
14	DEC	ARITHMETIC	001110	ACC <= ACC - 1	DEC
15	LOADSN	LOADSN	001111	R1 & ACC <= BRAM(BP)	LOADSN
16	ADD	ARITHMETIC	010000	ACC <= ACC + reg (Saturated addition)	ADD reg
17	SUB	ARITHMETIC	010001	ACC <= ACC – reg (Saturated subtraction)	SUB reg
18	MUL	ARITHMETIC	010010	ACC & R1 <= ACC * reg (Signed product)	MUL reg
19	MULS	ARITHMETIC	010011	ACC <= ACC * reg (Most significant word signed product)	MULS reg
20	AND	LOGIC	010100	ACC <= ACC AND reg	AND reg
21	OR	LOGIC	010101	ACC <= ACC OR reg	OR reg
22	INV	LOGIC	010110	ACC <= INV reg	INV reg
23	XOR	LOGIC	010111	ACC <= ACC XOR reg	XOR reg
24	MOVA	MOVEMENT	011000	ACC <= reg	MOVA reg
25	MOVR	MOVEMENT	011001	reg <= ACC	MOVR reg
26	SWAPS	MOVEMENT	011010	reg <=> shadow_reg (Swap register)	SWAPS reg
27	MOVRS	MOVEMENT	011011	reg <= shadow_reg	MOVRS reg
28	LOOP	SEQ	011100	Push LOOP_BUFFER(n-1);Push PC_BUFFER(PC+1)	LOOP n
29	LOOPV	SEQ	011101	Push LOOP_BUFFER(DMEM-1);Push PC_BUFFER(PC+1)	LOOPV ****
30	ENDL	SEQ	011110	If LOOP_BUFFER = 0 then pop LOOP_BUFFER; pop PC_BUFFER; else LOOP_BUFFER <= LOOP_BUFFER – 1; PC <= PC_BUFFER	ENDL
31	GOSUB	SEQ	011111	PC <= addr; Push PC_BUFFER(PC+1)	GOSUB addr
32	RET	SEQ	100000	PC <= PC_BUFFER	RET
33	FREEZEC	CONDITIONAL	100001	if C=1 then F <= 1; push F_BUFFER(1)	FREEZEC
34	FREEZENC	CONDITIONAL	100010	if C=0 then F <= 1; push F_BUFFER(1)	FREEZENC
35	FREEZEZ	CONDITIONAL	100011	if Z=1 then F <= 1; push F_BUFFER(1)	FREEZEZ
36	FREEZENZ	CONDITIONAL	100100	if Z=0 then F <= 1; push F_BUFFER(1)	FREEZENZ
37	UNFREEZE	CONDITIONAL	100101	F <= pop F_BUFFER	UNFREEZE
38	HALT	SEQ	100110	INT<=1;sequencer halted until external input signal INT_ACK=1	HALT
39	SETZ	FLAGS	100111	Z <= 1	SETZ
40	SETC	FLAGS	101000	Sets the carry flags C <= 1	SETC
41	CLRZ	FLAGS	101001	Clears the zero flags Z <= 0	CLRZ
42	CLRC	FLAGS	101010	Clears the zero flags C <= 0	CLRC
43	RANDON	RAND	101011	random_en <= 1; LFSR becomes source register for LLFSR	RANDON
44	SEED	MOVEMENT	101100	LFSR(63:32) <= LFSR(31:0) <= R1 & ACC	SEED
45	RANDOFF	RAND	101101	random_en <= 0; LFSR_STEP <=0; LFSR disabled	RANDOFF
46	SPKDIS	SEQ	101110	eo_exec <= 1, Stops the sequencer and stores spikes until input signal cam_en <= 0 (from AER control unit)	SPKDIS
47	READMP	SEQ	101111	DMEM <= BRAM(address)	READMP addr
48	RST_SEQ	SEQ	110000	Jumps to RESET state	RST_SEQ
49	-	-	110001	-	MONIT reg
50	LAYERV	SEQ	110010	VLAYERS <= n; CURR_VLAYER <= 0; defines number of virtual layers (currently 0 <= n <= 7)	LAYERV n
51	GOTO	SEQ	110011	PC <= addr	GOTO addr
52	SHLAN	REGISTERS	110100	ACC <= ACC << n, (1 <= n <= 8), Arithmetic shift	SHLAN n
53	SHRAN	REGISTERS	110101	ACC <= ACC >> n, (1 <= n <= 8), Arithmetic shift	SHRAN n
54	LOADBP	LOADBP	110110	BP <= DMEM Loads PE BRAM pointer.	LOADBP ****
55	BITSET	REGISTERS	110111	ACC(n) <= 1	BITSET n
56	BITCLR	REGISTERS	111000	ACC(n) <= 0	BITCLR n
57	SPMOV	SPMOV	111001	Special MOVE. n = 0: VIRT <= ACC;	SPMOV n
58	INCV	SEQ	111010	VLAYER <= VLAYER + 1	INCV
59	READMPV	SEQ	111011	DMEM <= BRAM(address + VLAYER)	READMPV addr
60	MOVSR	MOVEMENT	111100	shadow_reg <= reg	MOVSR reg
61	MARK	SEQ	111101	No operation	MARK

*Flags If the given instruction can change the indicated flag
** En
F: Frozen flag. /F= not(F) means unfrozen and the indicated instructions become enabled

*** Z can change only if ACC is set or reset (not in case of other registers)
**** See macros

MACRO INSTRUCTIONS: Conversion into elementary instructions.
It is recommended to use macro instructions instead of the associated simple instructions

1	LDALL			LDALL reg, const: reg <= DMEM(const) (from sequencer)
	Elementary instructions:			NOP
				READMP const
				LDALL reg

	MONIT			MONIT reg: Monit_buffer <= reg
	Elementary instructions:			MOVA reg
				STOREB

29	LOOPV			LOOPV vp: Push LOOP_BUFFER(DMEM(vp)-1);Push PC_BUFFER(PC+1)
	Elementary instructions:			NOP
				READMPV vp
				LOOPV

54	LOADBP			LOADBP bp: BP <= DMEM(bp) Loads PE BRAM pointer.
	Elementary instructions:			NOP
				READMP bp
				LOADBP

Appendix B

Assembler code

B.1 Algorithm with no virtualization

```
breakatwhitespace
define virtual_layers 0 ; From 0 up to 7
define gsynapses  2 ; Up to 32 global synapses

.DATA

; Virtual layers

V0 = "00000002" ; Number of assigned synapses (s-1) to the main layer
V1 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 1
V2 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 2
V3 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 3
V4 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 4
V5 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 5
V6 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 6
V7 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 7
VLAYERS="00000000" ; 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
;
; Synapse parameters common to all neurons come here
; TBD
;
; Neural and Synaptic RAM addresses
SYN_ADDR0="00000000"   ; First address of Synaptic parameters in SNRAM for V = 0.
SYN_ADDR1="00000003"   ; First address of Synaptic parameters in SNRAM for V = 1.
SYN_ADDR2="00000006"   ; First address of Synaptic parameters in SNRAM for V = 2.
SYN_ADDR3="00000009"   ; First address of Synaptic parameters in SNRAM for V = 3.
SYN_ADDR4="0000000C"   ; First address of Synaptic parameters in SNRAM for V = 4.
SYN_ADDR5="0000000F"   ; First address of Synaptic parameters in SNRAM for V = 5.
SYN_ADDR6="00000012"   ; First address of Synaptic parameters in SNRAM for V = 6.
SYN_ADDR7="00000015"   ; 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.
```

```
38 NEU_ADDR1="000003E4"    ; First address of Neural parameters in SNRAM (996) for V = 1.
39 NEU_ADDR2="000003E5"    ; First address of Neural parameters in SNRAM (997) for V = 2.
40 NEU_ADDR3="000003E6"    ; First address of Neural parameters in SNRAM (998) for V = 3.
41 NEU_ADDR4="000003E7"    ; First address of Neural parameters in SNRAM (999) for V = 4.
42 NEU_ADDR5="000003E8"    ; First address of Neural parameters in SNRAM (1000) for V = 5
       .
43 NEU_ADDR6="000003E9"    ; First address of Neural parameters in SNRAM (1001) for V = 6
       .
44 NEU_ADDR7="000003EA"    ; First address of Neural parameters in SNRAM (1002) for V = 7
       .
45
46 SEEDH_ADDR = "000003FD" ; Address of noise seed in SNRAM
47 SEEDL_ADDR = "000003FE" ;
48 PEID   = "000003FF" ; Address of  PE Identifier number
49 ;
50 ; General constants
51 THAU_MEM="0000799A"    ; Membrane time constant decay (inverse value). To be tuned.
       Thau = 20
52 NOISE_MSK="0000001F"   ; Noise mask. To be tuned
53
54 ; Constants for debug
55 JUMP_MV = "00000100"      ; Jump 2.56 mV on spike
56 LFSR_VAL= "0000AAAA"
57 LFSR_VAL2= "00005555"
58
59
60 .CODE
61 ;
62 GOTO MAIN ; Jump to main program
63 ;
64 ; *************************** PROCEDURES BEGIN ***************************
65 ;
66 .RANDOM_INIT     ; Uses R0 and R1
67   LOADBP SEEDH_ADDR
68   LOADSN
69   SEED          ; High seed
70   LOADBP SEEDL_ADDR
71   LOADSN
72   SEED          ; Low seed
73 RET
74 ;
75 .LOAD_NEURON     ; Uses R0, R1, R2 and R3
76   READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
77   LOADBP ; SNRAM pointer to currently processed neuron
78   LOADSN     ; Load Neural parameters from SNRAM to R1 & ACC
79   MOVR R2    ; Move Vmem from ACC to R2
80   MARK
81 RET
82 ;
83 .MEMBRANE_DECAY ; Uses R0, R4
84   MOVA R2                                    ;     TEMPORARY WHILE MULS has
       problems. REWRITE when it works
85   LDALL R4 VREST
86   SUB R4
87   LDALL R1, THAU_MEM
88   MULS R1    ; Calculate decay
89   SHLAN 1
90   ADD R4
91   MOVR R2    ; Back to R2 where membrane potential is stored
92 RET
93 ;
```

```
94 .ADD_NOISE ; Uses R0, R2 and R5
95   RANDON                    ; LFSR ON
96   LLFSR ; Noise to ACC
97   MOVR R5
98   LDALL ACC, NOISE_MSK
99   AND R5
100  SHRN 1
101  RANDOFF                   ; LFSR OFF. Arbitrarily here
102  FREEZENC
103       MOVR R5
104       RST ACC
105       SUB R5
106  ; Generate signed noise without the negative bias of two's complement
107  UNFREEZE
108    MOVSR ACC                                                ;     TO MONITOR THE
       NOISE
109  ADD R2  ; Add to Vmem
110  MOVR R2 ; Back to R2
111 RET
112 ;
113 .SYNAPSE_CALC
114     LOADSP     ; Load Synaptic parameters and spike to R1 & ACC
115     SHRN 1     ; Move spike to flag C
116     FREEZENC
117       MOVA R1 ; Synaptic parameter to ACC
118         ADD R2
119         MOVR R2 ; Save Neural parameter in R2
120     UNFREEZE
121     RST ACC
122     STORESP    ; Stores synaptic parameter and increases BP for next synapse
       processing
123 RET
124 ;
125 .DETECT_SPIKE ; Uses R0 and R2
126   LDALL ACC, VTHRES
127   SUB R2     ; Compare Vth - Vmem
128   SHLN 1     ;subtraction sign to C flag
129   RST ACC
130   FREEZENC  ; If positive, spike
131     SET ACC
132     LDALL R2 VREST ; Vmem to resting potential
133   UNFREEZE
134   STOREPS    ; Push spikes
135 RET
136 ;
137 .STORE_NEURON ; uses R0 and R1
138   MOVA R2    ; Move Vmem from R2 to ACC
139   READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
140   LOADBP ; SNRAM pointer to currently processed neuron
141   STORESP  ; Store Vmem to SNRAM
142 RET
143 ;
144 ; ************************** PROCEDURES END ****************************
145
146 ; ************************** MAIN PROGRAMME BEGIN ***********************
147 .MAIN
148 ;
149
150 ; Virtual operation init
151 LAYERV virtual_layers ; Init sequencer vlayers. It is 0 for non-virtual operation
152 LDALL ACC, VLAYERS ; Load defined virtual layers to PE array
```

```
153 SPMOV 0 ; VIRT <= ACC
154
155 ; Initial instructions
156 GOSUB RANDOM_INIT ; For noise initialization
157
158 .EXEC_LOOP ; Execution loop
159
160 ; ---------------------- UNCOMMENT AND CHECK FOR GLOBAL SYNAPSES
161 ; LAYER 0 NEURON
162 ; Global synapses (layer 0)
163 ; GOSUB LOAD_NEURON
164 ; GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
165 ; GOSUB ADD_NOISE
166 ;
167 ; LOADBP GSYN_ADDR
168 ; LOOP gsynapses
169 ;       NOP
170 ;       GOSUB SYNAPSE_CALC
171 ; ENDL
172 ; End of global synapses
173 ; ---------------------- END UNCOMMENT AND CHECK FOR GLOBAL SYNAPSES
174 ;
175 LOOP virtual_layers ; Neuron loop for virtual operation
176     NOP    ;to prevent pipeline error
177     GOSUB LOAD_NEURON
178     GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
179     GOSUB ADD_NOISE
180     READMPV SYN_ADDR0
181     LOADBP
182       LOOPV V0  ; synaptic loop. Reads number of current-layer synapses
183             NOP    ;to prevent pipeline error
184             GOSUB SYNAPSE_CALC
185         ENDL
186     ; Compare and eventually spike
187     GOSUB DETECT_SPIKE
188     GOSUB STORE_NEURON
189     INCV
190     ENDL
191 .FINISH
192 NOP  ; Empty pipeline wait NOPs
193 NOP
194 NOP
195 SPKDIS ; Distribute spikes
196 GOTO EXEC_LOOP ; Execution loop
```

B.2 Algorithm with virtualization

```
breakatwhitespace
1  define virtual_layers 7 ; from 0 up to 7 (1 to 8 layers)
2  define gsynapses  2 ; Up to 32 global synapses
3
4  .DATA
5
6  ; Virtual layers
7
8  V0 = "00000002" ; Number of assigned synapses (s-1) to the main layer
9  V1 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 1
10 V2 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 2
```

```
11 V3 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 3
12 V4 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 4
13 V5 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 5
14 V6 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 6
15 V7 = "00000002" ; Number of assigned synapses (s-1) to virtual layer 7
16 VLAYERS="00000007" ; Number of virtual layers (n-1).
17 ;  VLAYERS="00000000" ; Number of virtual layers (n-1).
18
19 ; Membrane potential parameters common to all neurons
20 VREST="FFFFE4A8"  ; Resting potential -70 mV = -7000 in tens of of uV
21 VTHRES="FFFFEA84"   ; Threshold voltage -55 mV = -5500
22 VDEPOL="FFFFE0C0"   ; Depolarization voltage -80 mV = -8000
23 VACT = "00001771"   ; Action potential +10 mV = +1000
24 ;
25 ; Synapse parameters common to all neurons come here
26 ; TBD
27 ;
28 ; Neural and Synaptic RAM addresses
29 SYN_ADDR0="00000000"   ; First address of Synaptic parameters in SNRAM for V = 0.
30 SYN_ADDR1="00000003"   ; First address of Synaptic parameters in SNRAM for V = 1.
31 SYN_ADDR2="00000006"   ; First address of Synaptic parameters in SNRAM for V = 2.
32 SYN_ADDR3="00000009"   ; First address of Synaptic parameters in SNRAM for V = 3.
33 SYN_ADDR4="0000000C"   ; First address of Synaptic parameters in SNRAM for V = 4.
34 SYN_ADDR5="0000000F"   ; First address of Synaptic parameters in SNRAM for V = 5.
35 SYN_ADDR6="00000012"   ; First address of Synaptic parameters in SNRAM for V = 6.
36 SYN_ADDR7="00000015"   ; First address of Synaptic parameters in SNRAM for V = 7.
37
38 GSYN_ADDR="00000064"   ; First address of Global Synaptic parameters in SNRAM.
39 NEU_ADDR0="000003E3"   ; First address of Neural parameters in SNRAM (995) for V = 0.
40 NEU_ADDR1="000003E4"   ; First address of Neural parameters in SNRAM (996) for V = 1.
41 NEU_ADDR2="000003E5"   ; First address of Neural parameters in SNRAM (997) for V = 2.
42 NEU_ADDR3="000003E6"   ; First address of Neural parameters in SNRAM (998) for V = 3.
43 NEU_ADDR4="000003E7"   ; First address of Neural parameters in SNRAM (999) for V = 4.
44 NEU_ADDR5="000003E8"   ; First address of Neural parameters in SNRAM (1000) for V = 5
       .
45 NEU_ADDR6="000003E9"   ; First address of Neural parameters in SNRAM (1001) for V = 6
       .
46 NEU_ADDR7="000003EA"   ; First address of Neural parameters in SNRAM (1002) for V = 7
       .
47
48 SEEDH_ADDR = "000003FD" ; Address of noise seed in SNRAM
49 SEEDL_ADDR = "000003FE" ;
50 PEID  = "000003FF" ; Address of  PE Identifier number
51 ;
52 ; General constants
53 ;THAU_MEM="00007F00"   ; Membrane time constant decay (inverse value). To be tuned
54 THAU_MEM="0000799A"   ; Membrane time constant decay (inverse value). To be tuned.
       Thau = 20
55 NOISE_MSK="0000001F"  ; Noise mask. To be tuned
56
57 ; Constants for debug
58 JUMP_MV = "00000100"     ; Jump 2.56 mV on spike
59 LFSR_VAL= "0000AAAA"
60 LFSR_VAL2= "00005555"
61
62
63 .CODE
64 ;
65 GOTO MAIN ; Jump to main program
66 ;
67 ; ************************** PROCEDURES BEGIN **************************
```

```
68 ;
69 .RANDOM_INIT     ; Uses R0 and R1
70   LOADBP SEEDH_ADDR
71   LOADSN
72   SEED          ; High seed
73   LOADBP SEEDL_ADDR
74   LOADSN
75   SEED          ; Low seed
76 RET
77 ;
78 .LOAD_NEURON     ; Uses R0, R1, R2 and R3
79   READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
80   LOADBP ; SNRAM pointer to currently processed neuron
81   LOADSN     ; Load Neural parameters from SNRAM to R1 & ACC
82   MOVR R2    ; Move Vmem from ACC to R2
83   MARK
84 RET
85 ;
86 .MEMBRANE_DECAY ; Uses R0, R4
87   MOVA R2                                          ;     TEMPORARY WHILE MULS has
       problems. REWRITE when it works
88   LDALL R4 VREST
89   SUB R4
90   LDALL R1, THAU_MEM
91   MULS R1    ; Calculate decay
92   SHLAN 1
93   ADD R4
94   MOVR R2    ; Back to R2 where membrane potential is stored
95 RET
96 ;
97 .ADD_NOISE ; Uses R0, R2 and R5
98   RANDON                    ; LFSR ON
99   LLFSR ; Noise to ACC
100  MOVR R5
101  LDALL ACC, NOISE_MSK
102  AND R5
103  SHRN 1
104  RANDOFF                    ; LFSR OFF. Arbitrarily here
105  FREEZENC
106       MOVR R5
107       RST ACC
108       SUB R5     ; Generate signed noise without the negative bias of two's complement
109  UNFREEZE
110    MOVSR ACC                                                  ;     TO MONITOR THE
       NOISE
111  ADD R2  ; Add to Vmem
112  MOVR R2 ; Back to R2
113 RET
114 ;
115 .SYNAPSE_CALC
116    LOADSP     ; Load Synaptic parameters and spike to R1 & ACC
117    SHRN 1     ; Move spike to flag C
118    FREEZENC
119      MOVA R1 ; Synaptic parameter to ACC
120        ADD R2
121        MOVR R2 ; Save Neural parameter in R2
122    UNFREEZE
123    RST ACC
124    STORESP    ; Stores synaptic parameter and increases BP for next synapse
       processing
125 RET
```

```
126 ;
127 .DETECT_SPIKE ; Uses R0 and R2
128   LDALL ACC, VTHRES
129   SUB R2     ; Compare Vth - Vmem
130   SHLN 1     ;subtraction sign to C flag
131   RST ACC
132   FREEZENC   ; If positive, spike
133     SET ACC
134     LDALL R2 VREST ; Vmem to resting potential
135   UNFREEZE
136   STOREPS    ; Push spikes
137 RET
138 ;
139 .STORE_NEURON ; uses R0 and R1
140   MOVA R2    ; Move Vmem from R2 to ACC
141   READMPV NEU_ADDR0 ; Address of real neuron + virt (valid also for non-virtual)
142   LOADBP ; SNRAM pointer to currently processed neuron
143   STORESP  ; Store Vmem to SNRAM
144 RET
145 ;
146 ; *************************** PROCEDURES END ****************************
147
148 ; *************************** MAIN PROGRAMME BEGIN ***********************
149 .MAIN
150 ;
151
152 ; Virtual operation init
153 LAYERV virtual_layers ; Init sequencer vlayers. It is 0 for non-virtual operation
154 LDALL ACC, VLAYERS ; Load defined virtual layers to PE array
155 SPMOV 0 ; VIRT <= ACC
156
157 ; Initial instructions
158 GOSUB RANDOM_INIT ; For noise initialization
159
160 .EXEC_LOOP ; Execution loop
161
162 ; ---------------------- UNCOMMENT AND CHECK FOR GLOBAL SYNAPSES
163 ; LAYER 0 NEURON
164 ; Global synapses (layer 0)
165 ; GOSUB LOAD_NEURON
166 ; GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
167 ; GOSUB ADD_NOISE
168 ;
169 ; LOADBP GSYN_ADDR
170 ; LOOP gsynapses
171 ;      NOP
172 ;      GOSUB SYNAPSE_CALC
173 ; ENDL
174 ; End of global synapses
175 ; ---------------------- END UNCOMMENT AND CHECK FOR GLOBAL SYNAPSES
176 ;
177 LOOP virtual_layers ; Neuron loop for virtual operation
178     NOP   ;to prevent pipeline error
179     GOSUB LOAD_NEURON
180     GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
181 ;    GOSUB ADD_NOISE
182     READMPV SYN_ADDR0
183     LOADBP
184       LOOPV V0  ; synaptic loop. Reads number of current-layer synapses
185             NOP   ;to prevent pipeline error
186             GOSUB SYNAPSE_CALC
```

```
187           ENDL
188       ; Compare and eventually spike
189       GOSUB DETECT_SPIKE
190       GOSUB STORE_NEURON
191       INCV
192       ENDL
193 .FINISH
194 NOP   ; Empty pipeline wait NOPs
195 NOP
196 NOP
197 SPKDIS ; Distribute spikes
198 GOTO EXEC_LOOP ; Execution loop
```

B.3 Algorithm with monitoring instruction

```
   breakatwhitespace
1  LOOP virtual_layers ; Neuron loop for virtual operation
2       NOP    ;to prevent pipeline error
3       GOSUB LOAD_NEURON
4       GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
5       GOSUB ADD_NOISE
6       READMPV SYN_ADDR0
7       LOADBP
8         LOOPV V0   ; synaptic loop. Reads number of current-layer synapses
9               NOP    ;to prevent pipeline error
10              GOSUB SYNAPSE_CALC
11        ENDL
12      ; Compare and eventually spike
13      GOSUB DETECT_SPIKE
14      GOSUB STORE_NEURON
15      LOADBP PEID
16      LOADSN
17      MOVR R3
18      MOVA R0
19      STOREB
20      MONIT R3
21      INCV
22      ENDL
23 .FINISH
24 NOP   ; Empty pipeline wait NOPs
25 NOP
26 NOP
27 SPKDIS ; Distribute spikes
28 GOTO EXEC_LOOP ; Execution loop
```

Appendix C

Netlist

C.1 Delay line 4x4 (no virtualization)

```
breakatwhitespace
# presyn postsyn
#i v r c|i v r c s ph pl
# N delay line 1
 0 0 0 0 0 0 0 1 1 1800 0
 0 0 0 2 0 0 0 1 2 1800 0
 0 0 0 3 0 0 0 1 3 -1800 0
 0 0 0 1 0 0 0 2 1 1800 0
 0 0 0 3 0 0 0 2 2 -1800 0
 0 0 0 2 0 0 0 3 1 200 0
# N delay line 2
 0 0 0 3 0 0 1 1 1 1800 0
 0 0 1 2 0 0 1 1 2 1800 0
 0 0 1 3 0 0 1 1 3 -1800 0
 0 0 1 1 0 0 1 2 1 1800 0
 0 0 1 3 0 0 1 2 2 -1800 0
 0 0 1 2 0 0 1 3 1 150 0
# Time base
 0 0 0 0 0 0 2 1 1 1800 0
 0 0 2 2 0 0 2 1 2 1800 0
 0 0 2 1 0 0 2 2 1 1800 0
 0 0 2 2 0 0 2 3 1 300 0
```

C.2 Oscillator 4x4 (no virtualization)

```
breakatwhitespace
# presyn postsyn
#i v r c i v r c s ph pl
 0 0 0 0 0 0 0 1 1 2500 0
 0 0 0 1 0 0 0 2 1 2500 0
 0 0 0 2 0 0 0 3 1 2500 0
 0 0 0 3 0 0 1 0 1 2500 0
 0 0 1 0 0 0 1 1 1 2500 0
 0 0 1 1 0 0 1 2 1 2500 0
```

```
 0  0  1  2  0  0  1  3  1  2500  0
 0  0  1  3  0  0  2  0  1  2500  0
 0  0  2  0  0  0  2  1  1  2500  0
 0  0  2  1  0  0  2  2  1  2500  0
 0  0  2  2  0  0  2  3  1  2500  0
 0  0  2  3  0  0  0  0  1  2500  0
```

C.3 Oscillator (with virtualization)

breakatwhitespace

```
# presyn  postsyn
#i  v  r  c  i  v  r  c  s  ph    pl
 0  0  0  0  0  1  0  0  1  2500  0
 0  1  0  0  0  2  0  0  1  2500  0
 0  2  0  0  0  3  0  0  1  2500  0
 0  3  0  0  0  4  0  0  1  2500  0
 0  4  0  0  0  5  0  0  1  2500  0
 0  5  0  0  0  6  0  0  1  2500  0
 0  6  0  0  0  7  0  0  1  2500  0
 0  7  0  0  0  0  0  0  1  2500  0
```

Appendix D

VHDL source files

D.1 PE_ARRAY

breakatwhitespace

```
----------------------------------------------------------------------------------
-- Project Name:  HEENS
-- Design Name:   PE_array.vhd
-- Module Name:   PE_array - connection
--
-- Creator: Sergi Juan
-- Modified: Roberto Gattuso
-- Modified: Corrado Bonfanti
--
-- Company: Universitat Politecnica de Catalunya (UPC)
--
--
-- Description:
-- Array of PE rows
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);
          col_pe         : in  std_logic_vector(4 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_monit       : in  std_logic;
```

```
38          block_monit    : in  std_logic;
39          vlayers_count  : in  std_logic_vector(vlayer_bits downto 0);
40          start_monit    : out std_logic;
41          eo_spike       : out std_logic;
42          spike_valid    : out std_logic;
43          row_sp         : out std_logic_vector(4 downto 0);
44          col_sp         : out std_logic_vector(4 downto 0);
45          virt_sp        : out std_logic_vector(2 downto 0);
46          monit_out      : out monit_type
47          );
48 end PE_array;
49
50 architecture interconnect of PE_array is
51
52     component PE_row is  -- Processing Element Row
53     generic(
54           row_number: integer;
55           LoadInitFile: integer
56           );
57     port (-- Ports of PE row
58           clk              : in  std_logic;
59           reset            : in  std_logic;
60           reset_spike      : in  std_logic;
61           next_virt        : in  std_logic;
62           vlayers          : in  std_logic_vector(vlayer_bits downto 0);
63           BRAMD_seq        : in  std_logic_vector(31 downto 0);
64           BRAMA_spike      : in  std_logic_vector(17 downto 0);
65           config           : in  std_logic;
66           AM_on            : in  std_logic;
67           sp_IntExt        : in  std_logic;
68           enable_x         : in  std_logic_vector(size_x_1 downto 0);
69           enable_y         : in  std_logic;
70           next_PE_row      : in  std_logic;
71           next_monit_row   : in  std_logic;
72           monit_block      : in  std_logic;
73           spike_inR        : in  std_logic_vector(size_x_1 downto 0);
74           spike_outR       : out std_logic_vector(size_x_1 downto 0);
75           monit_data_in    : out monit_type;
76           monit_data_out   : out monit_type
77           );
78     end component;
79     --
80     -- other signals of the architecture ....
81     --
82
83     -- Monitoring signals
84     type      monit_typeA is array(size_y_1 downto 0) of monit_type;
85     signal    monit_inA          : monit_typeA;
86     signal    monit_outA         : monit_typeA;
87     signal    PE_count_monit       : integer range 0 to size_y;
88     signal    next_row_monit        : std_logic;
89     signal    next_row_monit_aux    : std_logic;
90     signal    en_monit_del       : std_logic;
91     signal    en_monit_aux       : std_logic;
92
93 begin -- interconnect of net_addr
94
95  -- Array of PE_row (Arrays of PE) port map generation
96     gPEi: for i in 0 to size_y_1 generate    -- rows
97         PEi: entity work.PE_row
98         generic map (i,
```

```
99          LoadInitFile)
100         port map(
101                 clk,
102                 reset,
103                 reset_spike,
104                 shift_sp,
105                 vlayers_count,
106                 BRAMD_seq,
107                 BRAMA_spike,
108                 config,
109                 AM_on,
110                 sp_IntExt,
111                 en_col(size_x_1 downto 0),
112                 en_row(i),
113                 next_PE_row,
114                 next_row_monit,
115                 block_monit,
116                 spike_inA(i)(size_x_1 downto 0),
117                 spike_outA(i)(size_x_1 downto 0),
118                 monit_inA(i)(size_x_1 downto 0),-- monit_data_in
119                 monit_outA(i)(size_x_1 downto 0)-- monit_data_out
120                 );
121     end generate gPEi;
122 --
123 -- Spikes connection and controller FSM ....
124 --
125
126 -- 1st Row start vector
127     gmonit_in0: for j in 0 to size_x_1 generate
128         monit_in0: monit_inA(0)(j)(15 downto 0) <= (others => '0');
129     end generate gmonit_in0;
130
131 -- Array Interconnect
132      gmonitA_coli: for i in 0 to size_x_1 generate
133         gmonitA_rowj: for j in 0 to (size_y_1 - 1) generate
134             monitAj: monit_inA(j+1)(i)(15 downto 0) <= monit_outA(j)(i)(15 downto 0);
135         end generate gmonitA_rowj;
136     end generate gmonitA_coli;
137 -- Monit data transfer FSM
138     monit_transfer_process: process(clk)
139     begin
140         if clk'event and clk = '1' then
141             if (reset = '1') then
142                 PE_count_monit <= size_y;
143                 next_row_monit_aux <= '0';
144             elsif (next_row_monit_aux = '1') then
145                 if (block_monit = '1') then
146                     next_row_monit_aux <= '1';
147                     PE_count_monit <= PE_count_monit;
148                 else
149                     if (PE_count_monit = 0) then
150                         PE_count_monit <= size_y;
151                         next_row_monit_aux <= '0';
152                     else
153                         PE_count_monit <= PE_count_monit - 1;
154                         next_row_monit_aux <= '1';
155                     end if;
156                 end if;
157             elsif (en_monit = '1') then
158                 PE_count_monit <= PE_count_monit - 1;
159                 next_row_monit_aux <= '1';
```

```
160             end if;
161         end if;
162     end process;
163 
164     start_monit <= next_row_monit_aux;
165     next_row_monit <= next_row_monit_aux and (not block_monit);
166 
167  -- output monitoring data assignment
168     monit_out_process: process(clk)
169     begin
170         if clk'event and clk = '1' then
171             if (reset = '1') then
172                 for i in 0 to size_x_1 loop
173                     monit_out(i) <= (others => '0');
174                 end loop;
175             elsif (next_row_monit = '1') then
176                 monit_out <= monit_outA(size_y_1);
177             end if;
178         end if;
179     end process;
180     --
181     -- Other logic for virtualization and debugging ....
182     --
183 
184 end architecture interconnect; -- of PE_array
```

D.2 PE_ROW

breakatwhitespace

```
1  ----------------------------------------------------------------------------
2  -- Project Name:  HEENS
3  -- Design Name:   PE_row.vhd
4  -- Module Name:   PE_row - connection
5  --
6  -- Creator: Sergi Juan
7  -- Modified: Roberto Gattuso
8  -- Modified: Corrado Bonfanti
9  -- Company: Universitat Politecnica de Catalunya (UPC)
10 library IEEE;
11 use IEEE.std_logic_1164.all;
12 use work.log_pkg.all;
13 use work.SNN_pkg.all;
14 use ieee.numeric_std.all;
15 
16 entity PE_row is
17         generic(
18               row_number: integer;
19               LoadInitFile: integer
20         );
21     port (
22           clk              : in  std_logic;
23           reset            : in  std_logic;
24           reset_spike      : in  std_logic;
25           next_virt        : in  std_logic;
26           vlayers          : in  std_logic_vector(vlayer_bits downto 0);
27           BRAMD_seq        : in  std_logic_vector(31 downto 0);
28           BRAMA_spike      : in  std_logic_vector(17 downto 0);
29           config           : in  std_logic;
```

```
30          AM_on            : in  std_logic;
31          sp_IntExt        : in  std_logic;
32          enable_x         : in  std_logic_vector(size_x_1 downto 0); --
33          enable_y         : in  std_logic;
34          next_PE_row      : in  std_logic;
35          next_monit_row   : in  std_logic;
36          monit_block      : in  std_logic;
37          spike_inR        : in  std_logic_vector(size_x_1 downto 0);
38          spike_outR       : out std_logic_vector(size_x_1 downto 0);
39          monit_data_in    : in  monit_type;
40          monit_data_out   : out monit_type
41      );
42  end PE_row;
43
44  architecture connection of PE_row is
45
46      component PE is  -- Processing Element cell
47      generic(
48              row_number  : integer;
49              col_number  : integer;
50              LoadInitFile: integer
51              );
52      port (
53          clk             : in  std_logic;
54          reset           : in  std_logic;
55          reset_spike     : in  std_logic;
56          next_virt       : in  std_logic;
57          vlayers         : in  std_logic_vector(vlayer_bits downto 0);
58          BRAMD_seq       : in  std_logic_vector(31 downto 0);
59          BRAMA_spike     : in  std_logic_vector(17 downto 0);
60          config          : in  std_logic;
61          AM_on           : in  std_logic;
62          sp_IntExt       : in  std_logic;
63          en_x            : in  std_logic;
64          en_y            : in  std_logic;
65          en_spike_tx     : in  std_logic;
66          en_monit_tx     : in  std_logic;
67          spike_input     : in  std_logic;
68          spike_output    : out std_logic;
69          data_in_monit   : in  std_logic_vector(15 downto 0);
70          data_out_monit  : out std_logic_vector(15 downto 0)
71      );
72      end component;
73
74   --
75   -- Other signals of the entity ....
76   --
77  begin
78
79   -- Row of PE port map generation
80      gPEj: for j in 0 to size_x_1 generate -- columns
81          PEj: entity work.PE
82          generic map(row_number,
83                      j,
84                      LoadInitFile
85                              )
86          port map(
87                  clk,
88                  reset,
89                  reset_spike_reg,
90                  next_virt,
```

```
91                  vlayers,
92                  BRAMD_seq_reg,
93                  BRAMA_spike_reg,
94                  config_reg,
95                  AM_on_reg,
96                  sp_IntExt_reg,
97                  enable_x_reg(j),
98                  enable_y_reg,
99                  next_PE_row,
100                 next_monit_row,-- en_monit_tx
101                 spike_inR(j),
102                 spike_outR(j),
103                 monit_data_in(j)(15 downto 0),-- data_in_monit
104                 monit_data_out(j)(15 downto 0)-- data_out_monit
105                 );
106     end generate gPEj;
107
108     --
109     -- Signal registering for timing ....
110     --
111
112 end architecture connection; -- of PE_row
```

D.3 PE

breakatwhitespace

```
1  ----------------------------------------------------------------------------------
2  -- Project Name:  HEENS
3  -- Design Name:   PE.vhd
4  -- Module Name:   PE - behavioral
5  --
6  -- Creator: Sergi Juan & Jordi Madrenas
7  -- Modified: Roberto Gattuso
8  -- Modified: Corrado Bonfanti
9  --
10 -- Company: Universitat Politecnica de Catalunya (UPC)
11 library IEEE;
12 use ieee.std_logic_1164.all;
13 use IEEE.STD_LOGIC_MISC.all;
14 use ieee.std_logic_unsigned.all;
15 use ieee.numeric_std.all;
16 use work.log_pkg.all;
17 use work.SNN_pkg.all;
18
19
20 entity PE is
21     generic(
22         row_number  : integer;
23         col_number  : integer;
24         LoadInitFile: integer
25     );
26     port(
27         clk             : in  std_logic;
28         reset           : in  std_logic;
29         reset_spike     : in  std_logic;
30         next_virt       : in  std_logic;
31         vlayers         : in  std_logic_vector(vlayer_bits downto 0);
32         BRAMD_seq       : in  std_logic_vector(31 downto 0);
```

```
33          BRAMA_spike     : in  std_logic_vector(17 downto 0);
34          config          : in  std_logic;
35          AM_on           : in  std_logic;
36          sp_IntExt       : in  std_logic;
37          en_x            : in  std_logic;
38          en_y            : in  std_logic;
39          en_spike_tx     : in  std_logic;
40          en_monit_tx     : in  std_logic;
41          spike_input     : in  std_logic;
42          spike_output    : out std_logic;
43          data_in_monit   : in  std_logic_vector(15 downto 0);
44          data_out_monit  : out std_logic_vector(15 downto 0)
45      );
46  end PE;
47
48  architecture behavioral of PE is
49
50      component SynapNeuralMemory is
51      generic(
52          row_number   : integer;
53          col_number   : integer;
54          LoadInitFile: integer
55      );
56      port(
57          clka   : in  std_logic;
58          ena    : in STD_LOGIC;
59          wea    : in  std_logic_vector(0 downto 0);
60          addra  : in  std_logic_vector(9 downto 0);
61          dina   : in  std_logic_vector(31 downto 0);
62          douta  : out std_logic_vector(31 downto 0)
63      );
64      end component;
65
66      component spike_RAMs is
67      generic(
68          row_number   : integer;
69          col_number   : integer;
70          LoadInitFile: integer
71      );
72      port(
73          clk          : in  std_logic;
74          reset        : in  std_logic;
75          en           : in  std_logic;
76          Sp_IntExt    : in  std_logic;
77          BRAMA_spike  : in  std_logic_vector(AER_RX_WIDTH - 1 downto 0);
78          BRAMD_seq    : in  std_logic_vector(GLOBAL_SYN - 1 downto 0);
79          Config       : in  std_logic;
80          AM_on        : in  std_logic;
81          reset_spikeReg   : in  std_logic;
82          GblSpike     : out std_logic_vector(GLOBAL_SYN - 1 downto 0);
83          LclSpike     : out std_logic_vector((LOCAL_SYN) - 2 downto 0)
84      );
85      end component;
86
87      component REG is
88      port(
89          clk        : in  std_logic;
90          reset      : in  std_logic;
91          regcode    : in  std_logic_vector(2 downto 0);
92          en         : in  std_logic_vector(7 downto 0);
93          data_in0   : in  std_logic_vector(15 downto 0);
```

```
94          data_in1  : in  std_logic_vector(15 downto 0);
95          data_in2  : in  std_logic_vector(15 downto 0);
96          data_in3  : in  std_logic_vector(15 downto 0);
97          data_in4  : in  std_logic_vector(15 downto 0);
98          data_in5  : in  std_logic_vector(15 downto 0);
99          data_in6  : in  std_logic_vector(15 downto 0);
100         data_in7  : in  std_logic_vector(15 downto 0);
101         data_out0 : out std_logic_vector(15 downto 0);
102         data_out1 : out std_logic_vector(15 downto 0);
103         data_out2 : out std_logic_vector(15 downto 0);
104         data_out3 : out std_logic_vector(15 downto 0);
105         data_out4 : out std_logic_vector(15 downto 0);
106         data_out5 : out std_logic_vector(15 downto 0);
107         data_out6 : out std_logic_vector(15 downto 0);
108         data_out7 : out std_logic_vector(15 downto 0)
109     );
110     end component;
111
112     component BUF is
113     port (
114         clk       : in  std_logic;
115         reset     : in  std_logic;
116         en        : in  std_logic;
117         data_in   : in  std_logic_vector(15 downto 0);
118         data_out  : out std_logic_vector(15 downto 0)
119     );
120     end component;
121
122     component ALU is
123     port(
124         clk       : in  std_logic;
125         reset     : in  std_logic;
126         InA       : in  std_logic_vector(15 downto 0);
127         InB       : in  std_logic_vector(15 downto 0);
128         OP_CODE   : in  std_logic_vector(5 downto 0);
129         OutCarry  : out std_logic;
130         OutZero   : out std_logic;
131         OutSolve  : out std_logic_vector(31 downto 0)
132     );
133     end component;
134
135 -- Operation Signals
136     signal data_in    : std_logic_vector(15 downto 0);
137     signal addr_reg   : std_logic_vector(2 downto 0);
138     signal addr_reg2  : std_logic_vector(3 downto 0);
139     signal opcode     : std_logic_vector(5 downto 0);
140     signal PE_en      : std_logic;
141
142 -- Register Bank Signals
143     signal regcode    : std_logic_vector(2 downto 0);
144     signal REG_en     : std_logic_vector(7 downto 0);
145     signal en_addr    : std_logic_vector(7 downto 0);
146     signal en_op      : std_logic_vector(7 downto 0);
147     signal data_in0   : std_logic_vector(15 downto 0);
148     signal data_in1   : std_logic_vector(15 downto 0);
149     signal data_in2   : std_logic_vector(15 downto 0);
150     signal data_in3   : std_logic_vector(15 downto 0);
151     signal data_in4   : std_logic_vector(15 downto 0);
152     signal data_in5   : std_logic_vector(15 downto 0);
153     signal data_in6   : std_logic_vector(15 downto 0);
154     signal data_in7   : std_logic_vector(15 downto 0);
```

```
155     signal data_out0 : std_logic_vector(15 downto 0);
156     signal data_out1 : std_logic_vector(15 downto 0);
157     signal data_out2 : std_logic_vector(15 downto 0);
158     signal data_out3 : std_logic_vector(15 downto 0);
159     signal data_out4 : std_logic_vector(15 downto 0);
160     signal data_out5 : std_logic_vector(15 downto 0);
161     signal data_out6 : std_logic_vector(15 downto 0);
162     signal data_out7 : std_logic_vector(15 downto 0);
163
164  -- Monitoring Buffer signals
165     signal en_buf_aux, en_buf: std_logic;
166     signal data_in_buf: std_logic_vector(15 downto 0);
167
168  --
169  -- Other signals of the entity ....
170  --
171 begin
172
173
174     data_in   <= BRAMD_seq(27 downto 12);
175     addr_reg  <= BRAMD_seq(8 downto 6);
176     addr_reg2 <= BRAMD_seq(9 downto 6);
177     opcode    <= BRAMD_seq(5 downto 0) when config = '0' else
178                  (others => '0');
179
180  --
181  -- All control and arithmetic hardware of the PE ....
182  --
183 ------------------------------------------------------------------------
184 ------------------------------------------------------------------------
185     with opcode select
186     en_buf_aux <= '1' when STOREB,
187                   '0' when others;
188
189     en_buf <= en_buf_aux or en_monit_tx;
190
191     data_in_buf <= data_out0 when (en_monit_tx ='0') else data_in_monit;
192 ------------------------------------------------------------------------
193 ------------------------------------------------------------------------
194
195     with opcode select
196     regcode <= "001" when RST,
197                "010" when SET,
198                "011" when SWAPS,
199                "100" when MOVSR,
200                "101" when MOVRS,
201                "000" when others; -- External write case
202
203  -- Register Enable vector
204     with opcode select
205     en_op <= en_addr when LDALL | MOVR | SWAPS | RST | SET | MOVRS | MOVSR,
206              "00000001" when LLFSR | SHLN | SHRN | RTL | RTR | INC | DEC | OP_ADD |
        OP_SUB | MULS | OP_AND | OP_OR | INV | OP_XOR | MOVA | SHLAN | SHRAN | BITSET |
        BITCLR,
207              "00000011" when LOADSP | LOADSN | MUL,
208              "00000000" when others;
209
210  -- Register enable vector by address
211     with addr_reg select
212     en_addr <= "00000001" when "000",
213                "00000010" when "001",
```

```
214                 "00000100" when "010",
215                 "00001000" when "011",
216                 "00010000" when "100",
217                 "00100000" when "101",
218                 "01000000" when "110",
219                 "10000000" when "111",
220                 "00000000" when others;
221
222 -- Auxiliary ALU_B operand
223     with addr_reg select
224     B_aux <= data_out0 when "000",
225              data_out1 when "001",
226              data_out2 when "010",
227              data_out3 when "011",
228              data_out4 when "100",
229              data_out5 when "101",
230              data_out6 when "110",
231              data_out7 when "111",
232             X"0000"    when others;
233
234     --
235     -- ....
236     --
237 -- Write R0 (ACC)
238     with opcode select
239     data_in0 <= data_in when LDALL,
240                 LFSR(15 downto 0) when LLFSR,
241                 ( SynNeuMem_DataOUT(15 downto 1) & spike_a ) when LOADSP | LOADSN,
242                 x"0000" when RST,
243                 x"FFFF" when SET,
244                 B_aux when MOVA,
245                 data_out0 when MOVR,
246                 ALU_Solve(15 downto 0) when others;
247
248     --
249     -- ....
250     --
251 -- Register port map
252     REG_inst: REG
253     port map(
254         clk        => clk,
255         reset      => reset,
256         regcode    => regcode,
257         en         => REG_en,
258         data_in0   => data_in0,
259         data_in1   => data_in1,
260         data_in2   => data_in2,
261         data_in3   => data_in3,
262         data_in4   => data_in4,
263         data_in5   => data_in5,
264         data_in6   => data_in6,
265         data_in7   => data_in7,
266         data_out0  => data_out0,
267         data_out1  => data_out1,
268         data_out2  => data_out2,
269         data_out3  => data_out3,
270         data_out4  => data_out4,
271         data_out5  => data_out5,
272         data_out6  => data_out6,
273         data_out7  => data_out7
274     );
```

```
275
276     --
277     -- ....
278     --
279 -- Buffer monitor port map
280     Buffer_inst: BUF
281     port map(
282         clk      => clk,
283             reset    => reset,
284             en       => en_buf,
285             data_in  => data_in_buf,
286             data_out => data_out_monit
287         );
288
289 end behavioral;
```

D.4 Z_AER_interface (first version)

breakatwhitespace

```
1  ----------------------------------------------------------------------------------
2  -- Project:  High Speed Serial AER interface for communicate SNN
3  -- Engineer:   Agosto 2016 -   Mireya Zapata
4
5  -- Module Name: Z_AER_INTERF.vhd
6  --
7  -- Description: Interface that allows the connection between HEENS and high speed AER
       module
8  -- Dependencies: SNN_PKG.vhd
9  -- modified: Corrado Bonfanti
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.all;
12 use ieee.std_logic_arith.all;
13 use IEEE.STD_LOGIC_MISC.all;
14 use IEEE.STD_LOGIC_UNSIGNED.all;
15
16 library work;
17 use work.SNN_pkg.all;
18
19 entity Z_AER_INTERFACE is
20     port(
21         reset              : in  std_logic;
22         -- AER side. -- AER_clk_in CLOCK DOMAIN --
23         AER_clk_in         : in  std_logic;
24         eo_tx_data         : in  std_logic;
25         eoconf_done        : in  std_logic;
26         dlyEmpty           : out std_logic;
27         -- tx data ---------------------------------------------------
28         En_ErrFifo         : in  std_logic;
29         OutFifoEn          : in  std_logic;
30         OutFifoData        : out std_logic_vector(0 to AER_TX_WIDTH - 1);
31         OutFifoEmpty       : out std_logic;
32         OutFifoValid       : out std_logic;
33         -- mon data ---------------------------------------------------
34         MonFifoEn          : in  std_logic;
35         MonFifoData        : out std_logic_vector(0 to 15);
36         MonFifoEmpty       : out std_logic;
37         MonFifoValid       : out std_logic;
38         -- -- rx data ---------------------------------------------------
```

```
39          AER_rx_data_out   : in std_logic_vector(AER_RX_WIDTH-1 downto 0);
40          AER_rx_valid_out  : in std_logic;
41          AER_rst_spikes_in: in std_logic;
42          -- ------------ HEENS_Side ------------------------------------------
43          HEENS_clk         : in  std_logic;
44          -- rx side
45          aer_addr_out      : out std_logic_vector(AER_RX_WIDTH-1 downto 0);
46          monit_block       : out std_logic;
47          ------  Configuration data ----------------------
48          ph_conf           : in  std_logic;
49          ph_dist           : in  std_logic;
50          enFIFO            : in  std_logic;
51          ------------Own Data  INPUT FIFO  ------------------
52          Z_ownCnfData      : in  STD_LOGIC_VECTOR(31 downto 0);
53          Z_ownCnfWr        : in  std_logic;
54          -- ------------------ HEENS  TO AER_SRT ----------------------------
55          row_sp            : in  std_logic_vector(4 downto 0);
56          col_sp            : in  std_logic_vector(4 downto 0);
57          virt              : in  std_logic_vector(2 downto 0);
58          spike_valid       : in  std_logic;
59          eo_exec           : in  std_logic;
60          monit_data_in     : in  monit_type;
61          start_monit       : in  std_logic;
62          -- ------------------AER_SRT  TO HEENS ----------------------------
63          AM_on             : out std_logic;
64          BRAMA_spike       : out std_logic_vector(AER_RX_WIDTH-1 downto 0);
65          AddConf           : out std_logic_vector(31 downto 0);
66          DataConf          : out std_logic_vector(31 downto 0);
67          Sp_IntExt         : out std_logic
68      );
69  end entity;
70
71  architecture arc of Z_AER_INTERFACE is
72
73      component mem_zdelay IS
74          PORT(
75              clka   : IN  STD_LOGIC;
76              ena    : IN  STD_LOGIC;
77              wea    : IN  STD_LOGIC_VECTOR(0 DOWNTO 0);
78              addra  : IN  STD_LOGIC_VECTOR(10 DOWNTO 0);
79              dina   : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
80              douta  : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
81          );
82      END component;
83
84      component monitFIFO is
85          PORT (
86              rst            : IN STD_LOGIC;
87              wr_clk         : IN STD_LOGIC;
88              rd_clk         : IN STD_LOGIC;
89              din            : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
90              wr_en          : IN STD_LOGIC;
91              rd_en          : IN STD_LOGIC;
92              dout           : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
93              full           : OUT STD_LOGIC;
94                 almost_full   : OUT STD_LOGIC;
95              empty          : OUT STD_LOGIC;
96              almost_empty   : OUT STD_LOGIC;
97              valid          : OUT STD_LOGIC
98          );
99      END component;
```

```
100     --
101     -- Other signals of the entity ....
102     --
103
104     -- Monitoring controller ------------------------------------------------------
105     type mon_fsm is (mon_IDLE, mon_WRITE );
106     signal mon_state : mon_fsm;
107
108     signal dIn_monitfifo     : std_logic_vector(15 downto 0);
109     signal MonitFifoData_i   : std_logic_vector(15 downto 0);
110     signal MonitFifoEmpty_i  : std_logic;
111     signal MonitFifoFull_i   : std_logic;
112     signal MonitFifoValid_i  : std_logic;
113     signal rd_monitfifo      : std_logic;
114     signal wr_monitfifo      : std_logic;
115     signal wr_mon_en         : std_logic;
116     signal monit_block_i     : std_logic;
117     signal monit_count       : std_logic_vector( (log2_size_x_1 - 1) downto 0 );
118 begin
119     -- ================================================================================
120     -- ------------------------------- OUTPUT_FIFO   HEENS TO AER -----
121     -- ================================================================================
122     -- ================================================================================
123     -- ------------------------------- INPUT FIFO   ( AER -> HEENS )
124     -- ================================================================================
125     -- ================================================================================
126     -- ------------------------------------------- READ CONFIGURATION DATA
127     -- ================================================================================
128     -- ================================================================================
129     -- ------------------------------------------- DELAY CONTROLLER   -------
130     -- ================================================================================
131     -- ================================================================================
132     -- ------------------------------- ERROR DETECTION   ------------------
133     -- ================================================================================
134     -- ================================================================================
135     -- ------------------------------------------- MONITORING CONTROLLER  -
136     -- ================================================================================
137
138     monit_fifo_inst : monitFIFO
139         PORT map(
140             rst           => reset,
141             rd_clk        => AER_clk_in,
142             wr_clk        => HEENS_clk,
143             din           => dIn_monitfifo,
144             wr_en         => wr_monitfifo,
145             rd_en         => rd_monitfifo,
146             dout          => MonitFifoData_i,
147             full          => MonitFifoFull_i,
148             almost_full   => OPEN,
149             empty         => MonitFifoEmpty_i,
150             almost_empty  => OPEN,
151             valid         => MonitFifoValid_i
152         );
153
154     MonFifoEmpty <= MonitFifoEmpty_i;
155     MonFifoValid <= MonitFifoValid_i;
156     MonFifoData  <= MonitFifoData_i;
157     wr_monitfifo <= wr_mon_en and (not MonitFifoFull_i);
158     rd_monitfifo <= MonFifoEn;
159     monit_block  <= monit_block_i or MonitFifoFull_i;
160
```

```
161     process(HEENS_clk)
162     begin
163         if (rising_edge(HEENS_clk)) then
164             case mon_state is
165                 when mon_IDLE =>
166                     if (start_monit = '1' and MonitFifoFull_i = '0') then
167                         mon_state <= mon_WRITE;
168                     else
169                         mon_state <= mon_IDLE;
170                     end if;
171                 when mon_WRITE =>
172                     if (start_monit = '1' or monit_block_i = '1' or MonitFifoFull_i =
        '1') then
173                         mon_state <= mon_WRITE;
174                     else
175                         mon_state <= mon_IDLE;
176                     end if;
177                 when others =>
178                     mon_state <= mon_IDLE;
179             end case;
180         end if;
181     end process;
182 
183     -- Output depends solely on the current state
184     process(mon_state)
185     begin
186         case mon_state is
187             when mon_IDLE =>
188                 wr_mon_en    <= '0';
189             when mon_WRITE =>
190                 wr_mon_en    <= '1';
191             when others =>
192                 wr_mon_en    <= '0';
193         end case;
194     end process;
195 
196     -- Process to let the FIFO loads all the elements of "monit_data"
197   -- input (there are "size_x" data to load)
198 
199     monit_count_process: process(HEENS_clk)
200     begin
201         if (rising_edge(HEENS_clk)) then
202             if (reset = '1') then
203                 monit_block_i <= '0';
204                 monit_count   <= (others => '0');
205             elsif (MonitFifoFull_i = '1') then
206                 monit_block_i <= monit_block_i;
207                 monit_count   <= monit_count;
208             elsif (monit_block_i = '1') then
209                 if (monit_count = size_x_1 - 1) then
210                     monit_block_i <= '0';
211                     monit_count   <= monit_count + 1;
212                 else
213                     monit_block_i <= '1';
214                     monit_count   <= monit_count + 1;
215                 end if;
216             elsif (start_monit ='1') then
217                 monit_block_i <= '1';
218                 monit_count   <= (others => '0');
219             end if;
220         end if;
```

```
221     end process;
222
223     dIn_monitfifo <= monit_data_in(conv_integer(unsigned(monit_count)))(15 downto 0);
224
225     -- ============================================================================
226     -- ------------------------------------    Clock  sync  ----------------------
227     -- ============================================================================
228
229 end arc;
```

D.5 Z_AER_tx (first version)

breakatwhitespace

```
1  ----------------------------------------------------------------------------------
2  -- Project:  High Speed Serial AER interface for communicate SNN
3  -- Engineer: Taho Dorta
4  --           Mireya Zapata
5  -- Create Date: Mayo 2013
6  -- Design Name: AER_top
7  -- Module Name: AER_tx.vhd
8  -- Modified: Corrado Bonfanti
9  library IEEE;
10 use ieee.std_logic_1164.all;
11 use IEEE.STD_LOGIC_UNSIGNED.all;
12
13 library work;
14 use work.SNN_pkg.all;
15
16 entity Z_AER_tx is
17   port(
18     user_clk          : in  std_logic;
19     reset             : in  std_logic;
20     -- bypass fifo if
21     f_data_bp         : in  std_logic_vector(0 to 15);
22     f_valid_bp        : in  std_logic;
23     f_empty_bp        : in  std_logic;
24     f_rd_bp           : out std_logic;
25     -- Aurora tx if
26     tx_d_o            : out std_logic_vector(0 to 15);
27     tx_src_rdy_n_o    : out std_logic;
28     tx_dst_rdy_n_i    : in  std_logic;
29     CHANNEL_UP        : in  std_logic;
30     -- inputs
31     chip_id_in     : in std_logic_vector(CHIP_ID_WIDTH - 1 downto 0);
32     eo_exec_in     : in std_logic;
33     en_monit_in    : in std_logic;
34     AER_on_in         : in std_logic;
35     AER_eo_distrib    : in std_logic;
36     AER_eo_Mon        : in std_logic;
37     rst_spikes_o      : out std_logic;
38     -- tx data
39     OutFifo_data_r_in : in  std_logic_vector(0 to AER_TX_WIDTH - 1);
40     OutFifoValid      : in  std_logic;
41     OutFifo_empty_in  : in  std_logic;
42     OutFifo_rd_out    : out std_logic;
43     eo_tx_data        : out std_logic;
44     eoconf_done       : out std_logic;
45     -- mn data
```

```
46      MonFifo_data_r_in : in  std_logic_vector(0 to 15);
47      MonFifoValid      : in  std_logic;
48      MonFifo_empty_in  : in  std_logic;
49      MonFifo_rd_out    : out std_logic;
50      monit_busy        : out std_logic;
51      monit_bp          : out std_logic;
52      -- ------------------- zynq signals ---------------
53      En_ErrFifo        : out std_logic;
54      st_initconf       : in  std_logic;
55      st_init           : in  std_logic;
56      en_config         : in  std_logic
57    );
58    attribute KEEP_HIERARCHY : string;
59    attribute KEEP_HIERARCHY of Z_AER_tx : entity is "YES";
60  end entity;
61
62  architecture arc of Z_AER_tx is
63
64    -- ****************Parameter Declarations**************************
65
66    constant DLY : time := 1 ns;
67
68    -- *************************Internal Register Declarations********
69
70    signal reset_c : std_logic;
71
72    signal dly_data_xfer  : std_logic;
73    signal channel_up_cnt : std_logic_vector(4 downto 0) := "00000";
74
75    -- TX CONTROLLER fsm ____________________________________________
76
77    type tx_fsm is (ST_TX_INIT_IDLE, ST_TX_INIT, TX_DATA_1,
78            ST_TX_EOINIT, ST_TX_CONF_IDLE,
79                    ST_TX_CONFIG, ST_TX_DATACONF,
80            ST_TX_EOCONFIG, TX_IDLE_1, TX_SYNC_1,
81                    TX_SYNC_2, TX_BP_1, TX_START_1, TX_DATA_2,
82            TX_FINISH_1, TX_BP_2, TX_BP_2_XT,
83                    TX_IDLE_2, TX_START_MON, TX_FINISH_MON,
84            TX_MON1, TX_MON2, TX_MON2_WAIT);
85
86    signal tx_state : tx_fsm;
87    signal mux_i    : std_logic_vector(3 downto 0);
88    -- BYPASS _____________________________________________________
89    -- finite state machine to read fifo and send data to aurora
90    signal en_bypass        : std_logic;
91    type bp_fsm is (BP_IDLE, BP_READ, BP_READ_WRITE,
92    BP_WAIT, BP_WAIT1, BP_EMPTY);
93    signal bp_state         : bp_fsm;
94    signal fifoB_rd_ena     : std_logic;
95    signal tx_src_rdy_n_bp  : std_logic;
96    signal tx_d_bp          : std_logic_vector(0 to 15);
97    signal bp_oip           : std_logic;
98    signal ready_tx         : std_logic;
99    -- START monitoring ____________________________________________
100   signal tx_src_rdy_n_sm  : std_logic;
101   signal tx_d_sm          : std_logic_vector(0 to 15);
102   signal en_start_mon     : std_logic;
103   signal sm_pck_valid     : std_logic;
104   type sm_fsm is (sm_IDLE, sm_SEND, sm_DONE_SM);
105   signal sm_state         : sm_fsm;
106   signal sm_done          : std_logic;
```

```
107
108   -- MONITORING ____________________________________________________________
109   signal tx_src_rdy_n_mn : std_logic;
110   signal tx_d_mn         : std_logic_vector(15 downto 0);
111   signal en_monit        : std_logic;
112   signal MonFifo_rd_en   : std_logic;
113   type   mn_fsm is (mn_IDLE, mn_READ, mn_READ_WRITE,
114   mn_WAIT, mn_WAIT1, mn_EMPTY);
115   signal mn_state        : mn_fsm;
116   signal wait_docc_mn    : std_logic;
117   signal mn_oip          : std_logic;
118   signal mn_oip2         : std_logic;
119   signal monit_busy_int  : std_logic;
120   signal monit_bp_int    : std_logic;
121   signal MonFifoValid_in : std_logic;
122
123     -- FINISH monitoring_____________________________________________________
124   signal tx_src_rdy_n_fm : std_logic;
125   signal tx_d_fm         : std_logic_vector(0 to 15);
126   signal en_finish_mon   : std_logic;
127   signal fm_pck_valid    : std_logic;
128   type fm_fsm is (fm_IDLE, fm_SEND, fm_DONE_st);
129   signal fm_state        : fm_fsm;
130   signal fm_done         : std_logic;
131
132   signal tx_mux       : std_logic_vector(3 downto 0);
133   signal rst_spikes_i : std_logic;
134   signal data_ready   : std_logic := '0';
135
136    --
137    -- Other signals of the entity ....
138    --
139 begin
140
141   -- ============================================================================
142   -- ------------------------- READY TX SIGNAL generation ---------------------
143   -- ============================================================================
144   --
145   -- ready tx signal generation. Active HIGH. It indicates when it is possible to
      write data in the BUS
146   --
147
148   ready_tx <= not tx_dst_rdy_n_i;
149   -- ============================================================================
150   -- ------------------------------------------------------- BYPASS FIFO ------
151   -- ============================================================================
152
153   -- MOORE FSM
154   -- Logic to advance to the next state
155   process(user_clk)
156   begin
157     if (rising_edge(user_clk)) then
158       case bp_state is
159         when BP_IDLE =>
160           if (ready_tx ='1' and f_empty_bp ='0' and en_bypass ='1') then
161             bp_state <= BP_READ;
162           else
163             bp_state <= BP_IDLE;
164           end if;
165         when BP_READ =>
166           bp_state <= BP_IDLE;
```

```
167           if (ready_tx = '1') then
168             bp_state <= BP_READ_WRITE;
169           end if;
170         when BP_READ_WRITE =>
171           bp_state <= BP_EMPTY;
172           if ready_tx = '1' and f_empty_bp = '0' then
173             bp_state <= BP_READ_WRITE;
174           end if;
175         when BP_WAIT =>
176           bp_state <= BP_WAIT;
177           if ready_tx = '1' then
178             bp_state <= BP_WAIT1;
179           end if;
180         when BP_WAIT1 =>
181           bp_state <= BP_READ;
182         when BP_EMPTY =>
183           bp_state <= BP_IDLE;
184           if ready_tx = '0' then
185             bp_state <= BP_WAIT;
186           end if;
187         when others =>
188           bp_state <= BP_IDLE;
189       end case;
190     end if;
191   end process;
192
193   -- Output depends solely on the current state
194   process(bp_state)
195   begin
196     case bp_state is
197       when BP_IDLE =>
198         fifoB_rd_ena <= '0';
199         bp_oip       <= '0';
200         wait_docc_bp <= '1';
201       when BP_READ =>
202         fifoB_rd_ena <= '1';     -- --read
203         bp_oip       <= '1';
204         wait_docc_bp <= '1';
205       when BP_READ_WRITE =>
206         fifoB_rd_ena <= '1';     -- --read
207         bp_oip       <= '1';
208         wait_docc_bp <= '1';
209       when BP_WAIT =>
210         fifoB_rd_ena <= '0';
211         bp_oip       <= '1';     -- ---done
212         wait_docc_bp <= '1';
213       when BP_WAIT1 =>
214         fifoB_rd_ena <= '0';
215         bp_oip       <= '1';     -- ---done
216         wait_docc_bp <= '0';
217       when BP_EMPTY =>
218         fifoB_rd_ena <= '0';
219         bp_oip       <= '1';     -- ---done
220         wait_docc_bp <= '1';
221       when others =>
222         fifoB_rd_ena <= '0';
223         bp_oip       <= '1';
224         wait_docc_bp <= '1';
225     end case;
226   end process;
227
```

```
228  -- fifo data bypass between fifo and tx_d
229  f_rd_bp          <= fifoB_rd_ena and ready_tx;
230  tx_d_bp          <= f_data_bp;
231  tx_src_rdy_n_bp <= (f_valid_bp and wait_docc_bp) or tx_dst_rdy_n_i;
232  -- ================================================================
233  -- ------------------------------------ START MONITORING PACKET ----------
234  -- ================================================================
235
236  tx_d_sm <= CTRL_HEAD & START_MON_HEAD & "0110" & CHIP_ID_in;
237
238  -- MOORE FSM
239  -- Logic to advance to the next state
240  process(user_clk)
241  begin
242    if (rising_edge(user_clk)) then
243      if (reset_c = '1') then
244        sm_state <= sm_IDLE;
245      else
246        case sm_state is
247          when sm_IDLE =>
248            sm_state <= sm_IDLE;
249            if (ready_tx = '1' AND en_start_mon = '1') then
250              sm_state <= sm_SEND;
251            end if;
252          when sm_SEND =>
253            sm_state <= sm_IDLE;
254            if (ready_tx = '1') then
255              sm_state <= sm_DONE_SM;
256            end if;
257          when sm_DONE_SM =>
258            sm_state <= sm_IDLE;
259          when others =>
260            sm_state <= sm_IDLE;
261        end case;
262      end if;
263    end if;
264  end process;
265
266  -- Output depends solely on the current state
267  process(sm_state)
268  begin
269    case sm_state is
270      when sm_IDLE =>
271        sm_pck_valid <= '1';
272        sm_done      <= '0';
273      when sm_SEND =>
274        sm_pck_valid <= '0';     -- > send
275        sm_done      <= '0';
276      when sm_DONE_SM  =>
277        sm_pck_valid <= '1';
278        sm_done      <= '1';     -- > done
279      when others =>
280        sm_pck_valid <= '1';
281        sm_done      <= '0';
282    end case;
283  end process;
284
285  tx_src_rdy_n_sm <= sm_pck_valid or (not ready_tx);
286
287
288  -- ================================================================
```

```
289  -- ------------------------------------- Monitoring PACKET ------------------
290  -- ========================================================================
291
292  -- moore fsm
293  -- Logic to advance to the next state
294  process(user_clk)
295  begin
296    if (rising_edge(user_clk)) then
297      case mn_state is
298        when mn_IDLE =>
299          mn_state <= mn_IDLE;
300          if(ready_tx ='1'and MonFifo_empty_in ='0'and en_monit ='1')then
301            mn_state <= mn_READ;
302          end if;
303        when mn_READ =>
304            mn_state <= mn_IDLE;
305            if (ready_tx = '1') then
306              mn_state <= mn_READ_WRITE;
307            end if;
308        when mn_READ_WRITE =>
309            mn_state <= mn_EMPTY;
310            if ready_tx = '1' and MonFifo_empty_in = '0' then
311              mn_state <= mn_READ_WRITE;
312            end if;
313        when mn_WAIT =>
314            mn_state <= mn_WAIT;
315            if ready_tx = '1' then
316              mn_state <= mn_WAIT1;
317            end if;
318        when mn_WAIT1 =>
319            mn_state <= mn_READ;
320        when mn_EMPTY =>
321            mn_state <= mn_IDLE;
322            if ready_tx = '0' then
323              mn_state <= mn_WAIT;
324            end if;
325        when others =>
326            mn_state <= mn_IDLE;
327      end case;
328    end if;
329  end process;
330
331  -- Output depends solely on the current state
332  process(mn_state)
333  begin
334    case mn_state is
335      when mn_IDLE =>
336
337          MonFifo_rd_en <= '0';
338          mn_oip        <= '0';
339          wait_docc_mn  <= '1';
340      when mn_READ =>
341          MonFifo_rd_en <= '1';    -- --read
342          mn_oip        <= '1';
343          wait_docc_mn  <= '1';
344      when mn_READ_WRITE =>
345          MonFifo_rd_en <= '1';    -- --read
346          mn_oip        <= '1';
347          wait_docc_mn  <= '1';
348      when mn_WAIT =>
349          MonFifo_rd_en <= '0';
```

```
350           mn_oip          <= '1';
351           wait_docc_mn    <= '1';
352         when mn_WAIT1 =>
353           MonFifo_rd_en <= '0';
354           mn_oip          <= '1';
355           wait_docc_mn    <= '0';
356         when mn_EMPTY =>
357           MonFifo_rd_en <= '0';
358           mn_oip          <= '1';    -- ----done
359           wait_docc_mn    <= '1';
360         when others =>
361           MonFifo_rd_en <= '0';
362           mn_oip          <= '0';
363           wait_docc_mn    <= '1';
364       end case;
365     end process;
366
367     process(user_clk)
368     begin
369       if (rising_edge(user_clk)) then
370         mn_oip2 <= not mn_oip;
371       end if;
372     end process;
373
374     MonFifoValid_in <= not MonFifoValid;
375     MonFifo_rd_out  <= MonFifo_rd_en and ready_tx;
376     tx_d_mn         <= MonFifo_data_r_in;
377     tx_src_rdy_n_mn <=(MonFifoValid_in and wait_docc_mn)or tx_dst_rdy_n_i;
378
379
380     -- ================================================================================
381     -- ------------------------------- FINISH MONITORING PACKET -----------------------
382     -- ================================================================================
383
384     tx_d_fm <= CTRL_HEAD & EOMON_HEAD & "0110" & CHIP_ID_in;
385
386     -- MOORE FSM
387     -- Logic to advance to the next state
388     process(user_clk)
389     begin
390       if (rising_edge(user_clk)) then
391         if (reset_c = '1') then
392           fm_state <= fm_IDLE;
393         else
394           case fm_state is
395             when fm_IDLE =>
396               fm_state <= fm_IDLE;
397               if (ready_tx = '1' AND en_finish_mon = '1') then
398                 fm_state <= fm_SEND;
399               end if;
400             when fm_SEND =>
401               fm_state <= fm_IDLE;
402               if (ready_tx = '1') then
403                 fm_state <= fm_DONE_ST;
404               end if;
405             when fm_DONE_ST =>
406               fm_state <= fm_IDLE;
407             when others =>
408               fm_state <= fm_IDLE;
409           end case;
410         end if;
```

```
411     end if;
412   end process;
413
414   -- Output depends solely on the current state
415   process(fm_state)
416   begin
417     case fm_state is
418       when fm_IDLE =>
419         fm_pck_valid <= '1';
420         fm_done      <= '0';
421       when fm_SEND =>
422         fm_pck_valid <= '0';     -- > send
423         fm_done      <= '0';
424       when fm_DONE_ST =>
425         fm_pck_valid <= '1';
426         fm_done      <= '1';     -- > done
427       when others =>
428         fm_pck_valid <= '1';
429         fm_done      <= '0';
430     end case;
431   end process;
432
433   tx_src_rdy_n_fm <= fm_pck_valid or (not ready_tx);
434   -- ================================================================
435   -- ------------------------------  TX_MAIN CONTROLLER ---------------
436   -- ================================================================
437
438   process(user_clk)
439   begin
440     if (rising_edge(user_clk)) then
441       if (reset_c = '1') then
442         tx_state <= ST_TX_INIT_IDLE;
443       else
444         case tx_state is
445       --
446       -- other states ....
447       --
448           when TX_IDLE_1 =>
449             tx_state <= TX_IDLE_1;
450             if (en_monit_in = '1') then
451               tx_state <= TX_START_MON;
452             elsif (eo_exec_in = '1') then
453               tx_state <= TX_SYNC_1;
454             end if;
455
456           -- Monitoring states
457
458           when TX_START_MON =>
459             tx_state <= TX_START_MON;
460             if (sm_done = '1') then
461               tx_state <= TX_MON1;
462             end if;
463
464           when TX_MON1 =>
465             tx_state <= TX_MON1;
466             if(MonFifo_empty_in ='1'and mn_oip ='0'and mn_oip2 ='0')then
467               tx_state <= TX_FINISH_MON;
468             end if;
469
470           when TX_FINISH_MON =>
471             tx_state <= TX_FINISH_MON;
```

```
472             if (fm_done = '1') then
473               tx_state <= TX_MON2;
474             end if;
475
476           when TX_MON2 =>
477             tx_state <= TX_MON2;
478             if (AER_eo_Mon = '1') then
479               tx_state <= TX_MON2_WAIT;
480             end if;
481
482           when TX_MON2_WAIT =>
483             tx_state <= TX_MON2_WAIT;
484             if (f_empty_bp = '1' and bp_oip = '0') then
485               if (MonFifo_empty_in = '0' or en_monit_in = '1') then
486                 tx_state <= TX_START_MON;
487               else
488                 tx_state <= TX_IDLE_1;
489               end if;
490             end if;
491
492       --
493       -- other states ....
494       --
495         end case;
496       end if;
497     end if;
498   end process;
499
500   output_tx_fsm : process(tx_state)
501   begin
502     en_sync         <= '0';
503     en_bypass       <= '0';
504     en_start        <= '0';
505     en_data         <= '0';
506     en_finish       <= '0';
507     en_idle         <= '0';
508     en_init         <= '0';
509     en_eoinit       <= '0';
510     en_conf         <= '0';
511     en_eoconf       <= '0';
512     tx_mux          <= "0000";
513     rst_spikes_i    <= '0';
514     En_ErrFifo      <= '0';
515     monit_busy_int <= '0';
516     monit_bp_int    <= '0';
517     en_monit        <= '0';
518       en_start_mon   <= '0';
519       en_finish_mon  <= '0';
520
521     case tx_state is
522   --
523     -- other states ....
524   --
525
526       when TX_IDLE_1 =>
527         en_idle     <= '1';
528         tx_mux      <= "0110";
529         En_ErrFifo <= '1';
530
531       when TX_START_MON =>
532         en_start_mon    <= '1';
```

```
533           tx_mux          <= "1100";
534           monit_busy_int <= '1';
535
536         when TX_MON1 =>
537
538           en_monit        <= '1';
539           tx_mux          <= "1011";
540           monit_busy_int <= '1';
541
542         when TX_FINISH_MON =>
543           en_finish_mon   <= '1';
544           tx_mux          <= "1101";
545           En_ErrFifo      <= '1';
546           monit_busy_int <= '1';
547
548         when TX_MON2 =>
549           en_bypass       <= '1';
550           tx_mux          <= "0001";
551           monit_bp_int    <= '1';
552
553         when TX_MON2_WAIT =>
554           en_bypass       <= '1';
555           tx_mux          <= "0001";
556           monit_bp_int    <= '1';
557
558         --
559         -- other states ....
560         --
561
562       end case;
563     end process;
564
565     rst_spikes_o <= rst_spikes_i;
566     eo_tx_data   <= en_finish;
567     monit_busy   <= monit_busy_int;
568     monit_bp     <= monit_bp_int;
569
570     -- ============================================================================
571     -- ------------------------------MUXES (output assignment)---------------------
572     -- ============================================================================
573
574     -- the mux enable port came from external config
575     -- or from tx controller fsm
576
577     mux_i <= tx_mux;
578
579     -- tx_src_rdy_n_o assignment
580     process(mux_i, tx_src_rdy_n_bp, tx_src_rdy_n_sy, tx_src_rdy_n_st,
581     tx_src_rdy_n_dt, tx_src_rdy_n_fi, tx_src_rdy_n_id, tx_src_rdy_n_conf,
582     tx_src_rdy_n_eoconf, tx_src_rdy_n_init, tx_src_rdy_n_eoinit,
583     tx_src_rdy_n_sm, tx_src_rdy_n_mn, tx_src_rdy_n_fm)
584       variable TEMP : std_logic;
585     begin
586       case mux_i is
587         when "0001" => TEMP := tx_src_rdy_n_bp; -- bypass
588         when "0010" => TEMP := tx_src_rdy_n_sy; -- AER SYNC
589         when "0011" => TEMP := tx_src_rdy_n_st; -- Start Packet
590         when "0100" => TEMP := tx_src_rdy_n_dt; -- Data Packet
591         when "0101" => TEMP := tx_src_rdy_n_fi; -- Finish Packet
592         when "0110" => TEMP := tx_src_rdy_n_id; -- Tx Idle
593         when "0111" => TEMP := tx_src_rdy_n_init; -- Tx Init Packet
```

```
594         when "1000" => TEMP := tx_src_rdy_n_eoinit; -- Tx EoInit Packet
595         when "1001" => TEMP := tx_src_rdy_n_conf; -- Tx Conf Packet
596         when "1010" => TEMP := tx_src_rdy_n_eoconf; -- Tx EoConf Packet
597         when "1011" => TEMP := tx_src_rdy_n_mn; -- Tx Monitoring Packet
598         when "1100" => TEMP := tx_src_rdy_n_sm; -- Start Monitoring Packet
599         when "1101" => TEMP := tx_src_rdy_n_fm; --Finish Monitoring Packet
600         when others => TEMP := '1';
601       end case;
602       tx_src_rdy_n_o <= TEMP after DLY; -- DLY is ignored by synth
603     end process;
604
605     -- tx_d_o assignment
606     process(mux_i, tx_d_bp, tx_d_sy, tx_d_st, tx_d_dt, tx_d_fi, tx_d_id,
607     tx_d_conf, tx_d_eoconf, tx_d_init, tx_d_eoinit, tx_d_mn, tx_d_sm,
608     tx_d_fm)
609       variable TEMP : std_logic_vector(0 to 15);
610     begin
611       case mux_i is
612         when "0001" => TEMP := tx_d_bp; -- TX bypass
613         when "0010" => TEMP := tx_d_sy; -- SYNC packet
614         when "0011" => TEMP := tx_d_st; -- Start packet
615         when "0100" => TEMP := tx_d_dt; -- Data packet
616         when "0101" => TEMP := tx_d_fi; -- Finish Packet
617         when "0110" => TEMP := tx_d_id; -- Tx IDLE
618         when "0111" => TEMP := tx_d_init; -- Tx Init Packet
619         when "1000" => TEMP := tx_d_eoinit; -- Tx EoInit Packet
620         when "1001" => TEMP := tx_d_conf; -- Tx Conf Packet
621         when "1010" => TEMP := tx_d_eoconf; -- Tx EoConf Packet
622         when "1011" => TEMP := tx_d_mn; -- Tx Monitoring Packet
623         when "1100" => TEMP := tx_d_sm; -- Start Monitoring Packet
624         when "1101" => TEMP := tx_d_fm; -- Finish Monitoring Packet
625         when others => TEMP := (others => '0');
626       end case;
627       tx_d_o <= TEMP after DLY;        -- DLY is ignored by synth
628     end process;
629
630 end arc;
```

D.6 Z_AER_rx

breakatwhitespace

```
1  ----------------------------------------------------------------------------------
2  -- Project:  High Speed Serial AER interface for communicate SNN
3  -- Engineer: Taho Dorta
4  --
5  -- Create Date: Mayo 2013
6  -- Design Name: AER_top
7  -- Module Name: aer_rx.vhd
8  -- Modified: Corrado Bonfanti
9  library IEEE;
10 use ieee.std_logic_1164.all;
11 use IEEE.STD_LOGIC_UNSIGNED.all;
12 use ieee.std_logic_arith.all;
13
14 library work;
15 use work.SNN_pkg.all;
16 use work.log_pkg.all;
17
```

```
18 entity Z_AER_rx is
19   port(
20     user_clk            : in  std_logic;
21     reset               : in  std_logic;
22     AER_ConfReady       : out std_logic;
23     -- aurora rx if
24     rx_src_rdy_n_i      : in  std_logic;
25     rx_d_i              : in  std_logic_vector(0 to 15);
26     -- bypass fifo if
27     f_data_w_o          : out std_logic_vector(0 to 15);
28     f_wr_o              : out std_logic;
29     -- second monitoring fifo if
30     f2_data_w_o          : out std_logic_vector(0 to 15);
31     f2_wr_o              : out std_logic;
32     -- frame_check
33     CHANNEL_UP     : in  std_logic;
34     -- parameters
35     chip_id_i      : in std_logic_vector(CHIP_ID_WIDTH - 1 DOWNTO 0);
36     ring_size_i    : in std_logic_vector(RING_SIZE_WIDTH - 1 DOWNTO 0);
37     -- AER status
38     AER_on_o            : out std_logic;
39     --AER_done_o          : out std_logic;
40     data_valid_o        : out std_logic;
41     -- AER RX interface (Connects to multiprocessor system)
42     AER_rx_data_out : out std_logic_vector(AER_RX_WIDTH - 1 downto 0);
43     AER_rx_valid_out: out std_logic;
44     AER_eo_distrib  : out std_logic;
45     eo_init         : out std_logic;
46     eo_config       : out std_logic;
47     AER_eo_Mon      : out std_logic
48   );
49   attribute KEEP_HIERARCHY : string;
50   attribute KEEP_HIERARCHY of Z_AER_rx : entity is "YES";
51 end entity;
52
53 architecture arc of Z_AER_rx is
54
55   -- **********************Parameter Declarations**********************
56
57   constant DLY : time := 1 ns;
58
59   -- ***************Internal Register Declarations*********************
60   -- SLACK registers
61   signal RX_D_SLACK            : std_logic_vector(0 to 15);
62   signal RX_SRC_RDY_N_SLACK    : std_logic;
63   signal RX_D_SLACK_2          : std_logic_vector(0 to 15);
64   signal RX_SRC_RDY_N_SLACK_2 : std_logic;
65
66   signal AER_eo_distrib_i   : std_logic;
67   signal init_detected_c    : std_logic;
68   signal eoinit_detected_c  : std_logic;
69   signal conf_detected_c    : std_logic;
70   signal eoconf_detected_c  : std_logic;
71   signal reset_c            : std_logic;
72   signal data_valid_c       : std_logic;
73   signal AER_eo_Mon_i   : std_logic;
74
75   -- detect packets
76   signal idle_detected_c        : std_logic;
77   signal idle_detected_r        : std_logic;
78   signal sync_detected_c        : std_logic;
```

```
79   signal sync_detected_r        : std_logic;
80   signal sync_detected_r1       : std_logic;
81   signal sync_detected_r2       : std_logic;
82   signal start_detected_c       : std_logic;
83   signal start_mon_detected_c   : std_logic;
84   signal finish_mon_detected_c  : std_logic;
85   signal finish_mon_detected_r  : std_logic;
86   signal start_detected_r       : std_logic;
87   signal finish_detected_c      : std_logic;
88   signal finish_detected_r      : std_logic;
89   signal finish_detected_r1     : std_logic;
90   signal finish_detected_r2     : std_logic;
91   signal own_ctrl_detected_c    : std_logic;
92   signal own_ctrl_detected_r    : std_logic;
93   signal data_detected_c        : std_logic;
94   signal data_detected_r        : std_logic;
95   signal own_data_detected_c    : std_logic;
96   signal own_data_detected_r    : std_logic;
97   signal control_bp             : std_logic;
98   -- MONITORING
99   signal en_Mon              : std_logic := '0';
100  signal finish_mon_dt       : std_logic := '0';
101  signal cont_finish_mon : std_logic_vector(log2_MON_SIZE-1 downto 0);
102  signal cont_finish_mon_dt :std_logic_vector(log2_n_PE_1 - 1 downto 0);
103   --
104   -- Other signals of the entity ....
105   --
106
107 begin
108
109  -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++  -- ++ -- ++ --
110  -- ________________________________AER PROTOCOL STARTS HERE _________________
111  -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ -- ++ --
112
113  -- SLACK registers
114  process(USER_CLK)
115  begin
116    if (USER_CLK'event and USER_CLK = '1') then
117      RX_D_SLACK_2          <= rx_d_i after DLY;
118      RX_D_SLACK            <= RX_D_SLACK_2;
119      RX_SRC_RDY_N_SLACK_2 <= rx_src_rdy_n_i after DLY;
120      RX_SRC_RDY_N_SLACK    <= RX_SRC_RDY_N_SLACK_2;
121    end if;
122  end process;
123
124  -- Generate RESET signal when Aurora channel is not ready
125  reset_c <= RESET;
126
127  -- ___________________________________ Capture incoming data _______________
128  -- Data is valid when RX_SRC_RDY_N is asserted
129  data_valid_c <= not RX_SRC_RDY_N_SLACK;
130
131  -- ___________________________________ DETECT PACKETS ____________________________
132
133  idle_detected_c       <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
       & IDLE_HEAD & "0") and (not en_Mon));
134  sync_detected_c       <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
       & SYNC_HEAD & "0") and (not en_Mon));
135  start_detected_c      <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
       & START_HEAD & "0") and (not en_Mon));
```

```
136   own_ctrl_detected_c   <= (data_valid_c and std_bool(RX_D_SLACK(0) = CTRL_HEAD) and
        std_bool(RX_D_SLACK(9 to 15) = chip_id_i) and (not en_Mon));
137   data_detected_c       <= (data_valid_c and std_bool(RX_D_SLACK(0) = DATA_HEAD) and
        (not en_Mon));
138   finish_detected_c     <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
        & FINISH_HEAD & "0") and (not en_Mon));
139
140
141
142   start_mon_detected_c  <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
        & START_MON_HEAD & "0") and (not en_Mon));
143   finish_mon_detected_c <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD
        & EOMON_HEAD & "0") and finish_mon_dt);
144
145   -- **--**--**--**--**--**--* ZYNQ SIGNALS **--**--**--**--**--**--**--
146
147   -- signal for detecting master chip_id
148   master_detected_c <= (data_valid_c and std_bool(RX_D_SLACK(0) = CTRL_HEAD) and
        std_bool(RX_D_SLACK(9 to 15) = CHIP_ID_BROADCAST) and (not en_Mon));
149   -- ---new packages
150   init_detected_c   <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD &
        INIT_HEAD & "0") and (not en_Mon));
151   eoinit_detected_c <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD &
        EOINIT_HEAD & "0") and (not en_Mon));
152   conf_detected_c   <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD &
        CONF_HEAD & "0") and (not en_Mon));
153   eoconf_detected_c <= (data_valid_c and std_bool(RX_D_SLACK(0 to 5) = CTRL_HEAD &
        EOCONF_HEAD & "0") and (not en_Mon));
154   MasterID_detected <= (data_valid_c and std_bool(RX_D_SLACK(5 to 8) = ID_DISCOVER)
        and (not en_Mon));
155
156   --////////////////////////////////////////////////////////////////
157   --_____________ generate BYPASS FIFO signals (F I L T E R) _____________
158
159   -- combinational filter
160   control_bp <= (own_ctrl_detected_c OR own_data_detected_c OR idle_detected_c OR
        MasterID_detected) and (not start_mon_detected_c);
161
162   f_wr_o     <= '0' when (control_bp = '1') OR ((start_mon_detected_c = '1' or en_Mon
        = '1') and (not (cont_finish_mon < (MON_SIZE - 1)))) else data_valid_c;
163   f_data_w_o <= RX_D_SLACK;
164
165   -- _____________ generate Sink FIFO signals (F I L T E R) _____________
166
167   -- combinational filter
168   -- SinkFIFO write enable
169
170   f2_wr_o     <= data_valid_c when ( (start_mon_detected_c = '1') or
171   en_Mon = '1') else '0';
172
173   f2_data_w_o <= RX_D_SLACK;
174
175    --
176    -- Other logic of the entity ....
177    --
178   -- ------------------------------------------------------------------------
179   -- ------------------------------ COUNTERS ------------------------------
180   -- ------------------- cont_finish_mon. Local -------------------------
181   process(user_clk)
182   begin
183     if (rising_edge(user_clk)) then
```

```
184       if (reset_c = '1' or AER_eo_Mon_i = '1') then
185         cont_finish_mon <= (others => '0');
186       elsif (CHANNEL_UP = '1') then
187         if (finish_mon_detected_c = '1') then
188           cont_finish_mon <= cont_finish_mon + 1;
189         end if;
190       end if;
191     end if;
192   end process;
193
194   -- ------------------- cont_finish_mon_dt. Local ------------------------
195   process(user_clk)
196   begin
197     if (rising_edge(user_clk)) then
198       if (reset_c = '1' or finish_mon_detected_c = '1') then
199         cont_finish_mon_dt <= (others => '0');
200       elsif (CHANNEL_UP = '1') then
201         if (data_valid_c = '1' and en_Mon = '1') then
202           cont_finish_mon_dt <= cont_finish_mon_dt + 1;
203         end if;
204       end if;
205     end if;
206   end process;
207
208   process(user_clk)
209   begin
210     if (rising_edge(user_clk)) then
211       if (reset_c = '1' or finish_mon_detected_c = '1') then
212         finish_mon_dt <= '0';
213       elsif (CHANNEL_UP = '1') then
214         if (cont_finish_mon_dt = n_PE_tot_1 and data_valid_c = '1') then
215           finish_mon_dt <= '1';
216         end if;
217       end if;
218     end if;
219   end process;
220
221   -- ------------------- mon_FSM. Local-----------------------------------
222
223   process(user_clk)
224   begin
225     if (rising_edge(user_clk)) then
226       if (reset_c = '1') then
227         en_Mon   <= '0';
228       elsif (CHANNEL_UP = '1') then
229         if (en_mon = '1') then
230           if (finish_mon_detected_c = '1' and finish_mon_dt = '1') then
231             en_Mon   <= '0';
232           end if;
233         elsif (start_mon_detected_c = '1') then
234           en_Mon   <= '1';
235         end if;
236       end if;
237     end if;
238   end process;
239
240    --
241    -- Other counters and logic of the entity ....
242    --
243   -- ------------------------ AER EO MONIT GENERATION ---------------------
244   --
```

```
245   --
246   process(user_clk)
247   begin
248     if (rising_edge(user_clk)) then
249       if ((finish_mon_detected_r = '1') and (cont_finish_mon = MON_SIZE)) then
250         AER_eo_Mon_i <= '1';
251       else
252         AER_eo_Mon_i <= '0';
253       end if;
254     end if;
255   end process;
256
257   -- ____________________________ output assignment ____________________________
258   --
259   data_valid_o        <= data_valid_c;
260   AER_on_o            <= AER_on;
261   AER_eo_distrib <= AER_eo_distrib_i;
262   AER_eo_Mon      <= AER_eo_Mon_i;
263
264 end arc;
```

D.7 AER_OneBoard (first version)

breakatwhitespace

```
1  -- ============================================================================
2  -- ------------------------ MONITORING CONTROLLER ------------------------------
3  -- ============================================================================
4
5  monit_fifo_inst : SinkFIFO_SB
6   PORT map(
7     clk           => user_clk,
8     rst           => reset,
9     din           => dIn_monitfifo,
10    wr_en         => wr_monitfifo,
11    rd_en         => rd_monitfifo,
12    dout          => MonitFifoData_i,
13    full          => MonitFifoFull_i,      -- to check
14    almost_full   => open,
15    empty         => MonitFifoEmpty_i,
16    almost_empty  => open,
17    valid         => MonitFifoValid_i
18   );
19
20  wr_monitfifo <= wr_mon_en and (not MonitFifoFull_i);
21  rd_monitfifo <= MonFifo_rd_en and (not MonitFifoEmpty_i);
22  monit_block  <= monit_block_i or MonitFifoFull_i;
23
24  -- This two following processes (and "resume_mon") are used to set
25     -- to '1' the signal "mn_valid_out" when "MonFifo_rd_en" is set
26  -- to '1' again, in order to signal that the last value
27  -- (before "MonFifo_rd_en" was set to '0') is now valid
28
29  mn_rd_cntrl_proc: process(user_clk)
30  begin
31   if (rising_edge(user_clk)) then
32    if (reset = '1') then
33     cntrl_mn_rd <= '0';
34    elsif (MonFifo_rd_en = '1') then
```

```
35      if (MonitFifoEmpty_i = '1') then
36       cntrl_mn_rd <= '0';
37      else
38       cntrl_mn_rd <= '1';
39      end if;
40     else
41      cntrl_mn_rd <= cntrl_mn_rd;
42     end if;
43    end if;
44  end process mn_rd_cntrl_proc;
45
46  mn_rd_delay_proc: process(user_clk)
47  begin
48   if (rising_edge(user_clk)) then
49    if (reset = '1') then
50     MonFifo_rd_en_d <= '0';
51    else
52     MonFifo_rd_en_d <= MonFifo_rd_en;
53    end if;
54   end if;
55  end process mn_rd_delay_proc;
56
57 resume_mn_read <= MonFifo_rd_en and(not MonFifo_rd_en_d)and cntrl_mn_rd;
58
59  process(user_clk)
60  begin
61   if (rising_edge(user_clk)) then
62    case mon_state is
63     when mon_IDLE =>
64      if (start_monit = '1' and MonitFifoFull_i = '0') then
65       mon_state <= mon_WRITE;
66      else
67       mon_state <= mon_IDLE;
68      end if;
69     when mon_WRITE =>
70      if (start_monit='1'or monit_block_i='1'or MonitFifoFull_i ='1')then
71       mon_state <= mon_WRITE;
72      else
73       mon_state <= mon_IDLE;
74      end if;
75     when others =>
76      mon_state <= mon_IDLE;
77    end case;
78   end if;
79  end process;
80
81  -- Output depends solely on the current state
82  process(mon_state)
83  begin
84   case mon_state is
85    when mon_IDLE =>
86     wr_mon_en     <= '0';
87     monit_busy    <= '0';
88    when mon_WRITE =>
89     wr_mon_en     <= '1';
90     monit_busy    <= '1';
91    when others =>
92     wr_mon_en     <= '0';
93     monit_busy    <= '0';
94   end case;
95  end process;
```

```
96
97  -- Process to let the FIFO loads all the element of monit_data input
98  monit_count_process: process(user_clk)
99  begin
100   if (rising_edge(user_clk)) then
101    if (reset = '1') then
102     monit_block_i    <= '0';
103     monit_count      <= (others => '0');
104    elsif (MonitFifoFull_i = '1') then
105     monit_block_i <= monit_block_i;
106     monit_count      <= monit_count;
107    elsif (monit_block_i = '1') then
108     if (monit_count = size_x_1 - 1) then
109      monit_block_i <= '0';
110      monit_count   <= monit_count + 1;
111     else
112      monit_block_i <= '1';
113      monit_count  <= monit_count + 1;
114     end if;
115    elsif (start_monit ='1') then
116     monit_block_i    <= '1';
117     monit_count      <= (others => '0');
118    end if;
119   end if;
120  end process;
121
122  dIn_monitfifo <= monit_data_in(conv_integer(unsigned(monit_count)))(15 downto 0);
123  MonFifoData   <= MonitFifoData_i;
124  mn_valid_out  <= (MonitFifoValid_i or resume_mn_read)and MonFifo_rd_en;
125  MnFIFO_Empty  <= MonitFifoEmpty_i;
```

D.8 PE_array (second version)

breakatwhitespace

```
1  ------------------------ Monit data transfer FSM ------------------------
2  -------------------------------------------------------------------------
3    arr_mon_fsm_ns: process(clk)
4    begin
5      if (rising_edge(clk)) then
6        case mn_state is
7          when mn_IDLE =>
8            if (en_monit = '1') then
9              if (block_monit = '1') then
10               mn_state <= mn_WAIT;
11             else
12               mn_state <= mn_START;
13             end if;
14           else
15             mn_state <= mn_IDLE;
16           end if;
17         when mn_START =>
18           if (block_monit = '1') then
19             mn_state <= mn_WAIT;
20           elsif (PE_count_monit < 4) then
21             mn_state <= mn_FINISH;
22           else
23             mn_state <= mn_START;
24           end if;
```

```
25          when mn_WAIT =>
26            if (block_monit = '1') then
27              mn_state <= mn_WAIT;
28            elsif (PE_count_monit < 4) then
29              if (PE_count_monit = 0) then
30                mn_state <= mn_IDLE;
31              else
32                mn_state <= mn_FINISH;
33              end if;
34            else
35              mn_state <= mn_START;
36            end if;
37          when mn_FINISH =>
38            if (block_monit = '1') then
39              mn_state <= mn_WAIT;
40            elsif (PE_count_monit < 4) then
41              if (PE_count_monit = 1) then
42                if (en_monit = '1') then
43                  mn_state <= mn_START;
44                else
45                  mn_state <= mn_IDLE;
46                end if;
47              else
48                mn_state <= mn_FINISH;
49              end if;
50            end if;
51        end case;
52      end if;
53    end process arr_mon_fsm_ns;
54
55    arr_mon_fsm_ps: process(mn_state)
56    begin
57      case mn_state is
58        when mn_IDLE =>
59          next_row_monit_aux <= '0';
60          busy_monitPE       <= '0';
61        when mn_START =>
62          next_row_monit_aux <= '1';
63          busy_monitPE       <= '1';
64        when mn_WAIT =>
65          next_row_monit_aux <= '0';
66          busy_monitPE       <= '1';
67        when mn_FINISH =>
68          next_row_monit_aux <= '1';
69          busy_monitPE       <= '0';
70        when others    =>
71          next_row_monit_aux <= '0';
72          busy_monitPE       <= '0';
73      end case;
74    end process arr_mon_fsm_ps;
75
76    PE_count_process: process(clk)
77    begin
78      if (clk'event and clk = '1') then
79            if (reset = '1') then
80          PE_count_monit <= conv_std_logic_vector(size_y, log2_size_y);
81        else
82          if (next_row_monit_aux = '1' and block_monit = '0') then
83            if (PE_count_monit = 1) then
84              PE_count_monit <= conv_std_logic_vector(size_y, log2_size_y);
85            else
```

```
86              PE_count_monit <= PE_count_monit - 1;
87            end if;
88          end if;
89        end if;
90      end if;
91    end process PE_count_process;
92
93    write_monit     <= next_row_monit_aux;
94    next_row_monit <= next_row_monit_aux and (not block_monit);
95    ------------------------------------------------------------------------
96    ------------------------------------------------------------------------
```

D.9 Z_AER_interface (second version)

breakatwhitespace

```
1  -- ============================================================================
2  -- ------------------------- MONITORING CONTROLLER ----------------------------
3  -- ============================================================================
4
5    gen_monit_fifo: for i in 0 to size_x_1 generate
6    monit_fifo_inst : monitFIFO
7      PORT map(
8        rst     => reset,
9        rd_clk  => AER_clk_in,
10       wr_clk  => HEENS_clk,
11       din     => dIn_monitfifo(i),
12       wr_en   => wr_monitfifo(i),
13       rd_en   => rd_monitfifo(i),
14       dout    => MonitFifoData_i(i),
15       full    => MonitFifoFull_i(i),
16       empty   => MonitFifoEmpty_i(i),
17       valid   => MonitFifoValid_i(i)
18     );
19   end generate gen_monit_fifo;
20
21   MonitFifoFull <= or_reduce(MonitFifoFull_i);
22
23   gen_MonEmpty: for i in 0 to size_x_1 generate
24     Empty: MonFifoEmpty(i) <= MonitFifoEmpty_i(i);
25   end generate gen_MonEmpty;
26
27   gen_MonValid: for i in 0 to size_x_1 generate
28     Valid: MonFifoValid(i) <= MonitFifoValid_i(i);
29   end generate gen_MonValid;
30
31   gen_MonData_out: for i in 0 to size_x_1 generate
32     Data_out: MonFifoData(i)  <= MonitFifoData_i(i);
33   end generate gen_MonData_out;
34
35   gen_MonWrite: for i in 0 to size_x_1 generate
36     Write:    wr_monitfifo(i) <= wr_mon_en and (not MonitFifoFull);
37   end generate gen_MonWrite;
38
39   gen_MonRead: for i in 0 to size_x_1 generate
40     Read: rd_monitfifo (i)<= MonFifoEn(i);
41   end generate gen_MonRead;
42
43   monit_block <= MonitFifoFull;
```

```
44
45  dIn_monitfifo(0) <= std_logic_vector( resize(unsigned(reg_Mon), dIn_monitfifo(0)'
      length) ) when reg_mon_flag = '1' else monit_data_in(0);
46
47  gen_dinMonFifo: for i in 1 to size_x_1 generate
48    dIn_monitfifo(i) <= (others => '0') when reg_mon_flag = '1' else monit_data_in(i)
      ;
49  end generate gen_dinMonFifo;
50
51  process(HEENS_clk)
52  begin
53    if (rising_edge(HEENS_clk)) then
54      case mon_state is
55        when mon_IDLE =>
56          if (start_monit = '1') then
57            mon_state <= mon_WRITE_reg;
58          else
59            mon_state <= mon_IDLE;
60          end if;
61        when mon_WRITE_reg =>
62          if (MonitFifoFull = '1') then
63            mon_state <= mon_WAIT1;
64          else
65            mon_state <= mon_WRITE;
66          end if;
67        when mon_WAIT1 =>
68          if (write_monit = '0') then
69            mon_state <= mon_WAIT1;
70          else
71            mon_state <= mon_WRITE_reg;
72          end if;
73        when mon_WRITE =>
74          if (MonitFifoFull = '1') then
75            mon_state <= mon_WAIT2;
76          elsif (start_monit = '1') then
77            mon_state <= mon_WRITE_reg;
78          elsif (write_monit = '0') then
79            mon_state <= mon_IDLE;
80          else
81            mon_state <= mon_WRITE;
82          end if;
83        when mon_WAIT2 =>
84          if (write_monit = '0') then
85            mon_state <= mon_WAIT2;
86          else
87            mon_state <= mon_WRITE;
88          end if;
89        when others =>
90          mon_state <= mon_IDLE;
91      end case;
92    end if;
93  end process;
94
95  -- Output depends solely on the current state
96  process(mon_state)
97  begin
98    case mon_state is
99      when mon_IDLE =>
100       wr_mon_en     <= '0';
101       reg_mon_flag <= '0';
102     when mon_WRITE_reg =>
```

```
103           wr_mon_en     <= '1';
104           reg_mon_flag <= '1';
105         when mon_WAIT1 =>
106           wr_mon_en     <= '0';
107           reg_mon_flag <= '0';
108         when mon_WRITE =>
109           wr_mon_en     <= '1';
110           reg_mon_flag <= '0';
111         when mon_WAIT2 =>
112           wr_mon_en     <= '0';
113           reg_mon_flag <= '0';
114         when others     =>
115           wr_mon_en     <= '0';
116           reg_mon_flag <= '0';
117       end case;
118     end process;
```

D.10 Z_AER_tx (second version)

breakatwhitespace

```
1  -- ============================================================================
2  -- ------------------------- START MONITORING PACKET --------------------------
3  -- ============================================================================
4
5    tx_d_sm <= CTRL_HEAD & START_MON_HEAD & reg_Mon & CHIP_ID_in;
6
7    reg_Mon <= '0' & MonFifo_data_r_in(0)(2 downto 0);
8
9    -- MOORE FSM
10   -- Logic to advance to the next state
11   process(user_clk)
12   begin
13     if (rising_edge(user_clk)) then
14       if (reset_c = '1') then
15         sm_state <= sm_IDLE;
16       else
17         case sm_state is
18           when sm_IDLE =>
19             sm_state <= sm_IDLE;
20             if (en_start_mon = '1') then
21               if (MonFifo_empty_int = '0') then
22                 sm_state <= sm_reg_Mon;
23               else
24                 sm_state <= sm_WAIT_reg;
25               end if;
26             end if;
27           when sm_WAIT_reg =>
28             if (MonFifo_empty_int = '0') then
29               sm_state <= sm_reg_Mon;
30             else
31               sm_state <= sm_WAIT_reg;
32             end if;
33           when sm_reg_Mon =>
34             if (ready_tx = '1') then
35               sm_state <= sm_SEND;
36             else
37               sm_state <= sm_WAIT;
38             end if;
```

```
39            when sm_WAIT =>
40              if (ready_tx = '1') then
41                sm_state <= sm_SEND;
42              else
43                sm_state <= sm_WAIT;
44              end if;
45            when sm_SEND =>
46              sm_state <= sm_WAIT;
47              if (ready_tx = '1') then
48                sm_state <= sm_DONE_SM;
49              end if;
50            when sm_DONE_SM =>
51              sm_state <= sm_IDLE;
52            when others =>
53              sm_state <= sm_IDLE;
54          end case;
55        end if;
56      end if;
57    end process;
58
59    gen_MonRD_out: for i in 0 to size_x_1 generate
60      MonFifo_rd_out(i) <= MonFifo_rd_out_int(i) or MonFifoRd_sm;
61    end generate;
62
63    -- Output depends solely on the current state
64    process(sm_state)
65    begin
66      case sm_state is
67        when sm_IDLE =>
68          sm_pck_valid <= '1';
69          sm_done      <= '0';
70          MonFifoRd_sm <= '0';
71        when sm_WAIT_reg =>
72          sm_pck_valid <= '1';
73          sm_done      <= '0';
74          MonFifoRd_sm <= '0';
75        when sm_reg_Mon =>
76          sm_pck_valid <= '1';
77          sm_done      <= '0';
78          MonFifoRd_sm <= '1';
79        when sm_WAIT =>
80          sm_pck_valid <= '1';
81          sm_done      <= '0';
82          MonFifoRd_sm <= '0';
83        when sm_SEND =>
84          sm_pck_valid <= '0';     -- > send
85          sm_done      <= '0';
86          MonFifoRd_sm <= '0';
87        when sm_DONE_SM  =>
88          sm_pck_valid <= '1';
89          sm_done      <= '1';     -- > done
90          MonFifoRd_sm <= '0';
91        when others =>
92          sm_pck_valid <= '1';
93          sm_done      <= '0';
94          MonFifoRd_sm <= '0';
95      end case;
96    end process;
97
98    tx_src_rdy_n_sm <= sm_pck_valid or (not ready_tx);
99
```

```
100
101 -- ============================================================================
102 -- ------------------------------ Monitoring PACKET ---------------------------
103 -- ============================================================================
104
105   -- Mealy fsm
106   -- Logic to advance to the next state
107   process(user_clk)
108   begin
109     if (rising_edge(user_clk)) then
110       case mn_state is
111         when mn_IDLE =>
112           mn_state <= mn_IDLE;
113           if (ready_tx = '1' and MonFifo_empty_int = '0' and en_monit = '1' and
      end_mon_packet = '0') then
114             mn_state <= mn_READ;
115           end if;
116         when mn_READ =>
117           if (ready_tx = '1' and end_mon_packet = '0') then
118             mn_state <= mn_READ_WRITE;
119           else
120             mn_state <= mn_IDLE;
121           end if;
122         when mn_READ_WRITE =>
123           if ready_tx = '1' and end_mon_packet = '0' then
124             mn_state <= mn_READ_WRITE;
125           else
126             mn_state <= mn_EMPTY;
127           end if;
128         when mn_WAIT =>
129           mn_state <= mn_WAIT;
130           if ready_tx = '1' then
131             mn_state <= mn_WAIT1;
132           end if;
133         when mn_WAIT1 =>
134           if (end_mon_packet = '1') then
135             mn_state <= mn_IDLE;
136           else
137             mn_state <= mn_WAIT2;
138           end if;
139         when mn_WAIT2 =>
140           if (ready_tx = '0') then
141             mn_state <= mn_WAIT;
142           elsif (end_mon_packet = '1') then
143             mn_state <= mn_IDLE;
144           else
145             mn_state <= mn_READ;
146           end if;
147         when mn_EMPTY =>
148           mn_state <= mn_IDLE;
149           if ready_tx = '0' then
150             mn_state <= mn_WAIT;
151           end if;
152         when others =>
153           mn_state <= mn_IDLE;
154       end case;
155     end if;
156   end process;
157
158   -- Process to set "end_mon_packet", that is used to stop the monitoring
159   -- transmission relative to one instruction, after the right number of
```

```
160   -- data have been already transmitted
161   end_mon_process: process(user_clk)
162   begin
163     if (rising_edge(user_clk)) then
164       if (reset_c = '1' or mn_oip = '0') then
165         end_mon_packet <= '0';
166       else
167         if (tx_src_rdy_n_mn = '0' and (monFifo_count = (size_x_1) ) and (
      monArray_count = size_y_1) ) then
168           end_mon_packet <= '1';
169         end if;
170       end if;
171     end if;
172   end process;
173
174   -- Read enable signal assignment to each FIFO
175   gen_dec_mon: for i in 0 to size_x_1 generate
176     MonFifo_rd_out_int(i) <= '1' when (monFifo_count = i) and (end_mon_packet = '0')
      and (MonFifo_rd_en = '1') else '0';
177   end generate gen_dec_mon;
178
179   MonFifo_empty_int <= and_reduce(MonFifo_empty_in);
180   MonFifoValid_int  <= not MonFifoValid(conv_integer(unsigned(monFifo_count_1)));
181
182   process(user_clk)
183   begin
184     if (rising_edge(user_clk)) then
185       if (reset_c = '1') then
186         monFifo_count_1 <= (others => '0');
187       elsif (monCount_en = '1') then
188         monFifo_count_1 <= monFifo_count;
189       end if;
190     end if;
191   end process;
192
193
194   -- process to count an index used to switch from one monit. FIFO to another
195   MonFIFO_count_process: process(user_clk)
196   begin
197     if (rising_edge(user_clk)) then
198       if (reset_c = '1') then
199         monFifo_count <= (others => '0');
200       elsif (monCount_en = '1') then
201         if (monFifo_count < size_x_1) then
202           monFifo_count <= monFifo_count + 1;
203         else
204           monFifo_count <= (others => '0');
205         end if;
206       end if;
207     end if;
208   end process;
209
210   ArrayCount_en <= '1' when tx_src_rdy_n_mn = '0' and (monFifo_count = (size_x_1) )
      else '0';
211
212   -- Process to keep track of the number of the monitoring data transmitted (of each
      fifo)
213   Array_counter_process: process(user_clk)
214     begin
215       if (rising_edge(user_clk)) then
216         if (reset_c = '1') then
```

```
217           monArray_count <= (others => '0');
218         elsif (ArrayCount_en = '1') then
219           if (monArray_count < size_y_1) then
220             monArray_count <= monArray_count + 1;
221           else
222             monArray_count <= (others => '0');
223           end if;
224         end if;
225       end if;
226     end process;
227
228   -- Mealy machine
229   process(mn_state, ready_tx, end_mon_packet)
230   begin
231     case mn_state is
232       when mn_IDLE =>
233         monCount_en    <= '0';
234         MonFifo_rd_en <= '0';
235         mn_oip         <= '0';
236         wait_docc_mn   <= '1';
237       when mn_READ =>
238         if (ready_tx = '1' and end_mon_packet = '0') then
239           monCount_en <= '1';
240         else
241           monCount_en <= '0';
242         end if;
243         MonFifo_rd_en <= '1';    -- --read
244         mn_oip         <= '1';
245         wait_docc_mn   <= '1';
246       when mn_READ_WRITE =>
247         if (ready_tx = '1' and end_mon_packet = '0') then
248           monCount_en <= '1';
249         else
250           monCount_en <= '0';
251         end if;
252         MonFifo_rd_en <= '1';    -- --read
253         mn_oip         <= '1';
254         wait_docc_mn   <= '1';
255       when mn_WAIT =>
256         monCount_en    <= '0';
257         MonFifo_rd_en <= '0';
258         mn_oip         <= '1';
259         wait_docc_mn   <= '1';
260       when mn_WAIT1 =>
261         if (end_mon_packet = '1') then
262           monCount_en <= '0';
263         else
264           monCount_en <= '1';
265         end if;
266         MonFifo_rd_en <= '0';
267         mn_oip         <= '1';
268         wait_docc_mn   <= '0';
269       when mn_WAIT2 =>
270         if (end_mon_packet = '0' and ready_tx = '1') then
271           monCount_en <= '1';
272         else
273           monCount_en <= '0';
274         end if;
275         MonFifo_rd_en <= '1';
276         mn_oip         <= '1';
277         wait_docc_mn   <= '0';
```

```
278         when mn_EMPTY =>
279           monCount_en    <= '0';
280           MonFifo_rd_en <= '0';
281           mn_oip         <= '1';    -- ----done
282           wait_docc_mn   <= '1';
283         when others =>
284           monCount_en    <= '0';
285           MonFifo_rd_en <= '0';
286           mn_oip         <= '0';
287           wait_docc_mn   <= '1';
288       end case;
289     end process;
290
291     tx_d_mn           <= MonFifo_data_r_in(conv_integer(unsigned(monFifo_count_1)));
292     tx_src_rdy_n_mn   <= (MonFifoValid_int and wait_docc_mn) or tx_dst_rdy_n_i;
```

D.11 AER_OneBoard (second version)

breakatwhitespace

```
1   -- ================================================================================
2   -- ------------------------- MONITORING CONTROLLER -------------------------------
3   -- ================================================================================
4
5   gen_monit_fifo: for i in 0 to size_x_1 generate
6   monit_fifo_inst : monitFIFO_SB
7     PORT map(
8       rst          => reset,
9       clk          => user_clk,
10      din          => dIn_monitfifo(i),
11      wr_en        => wr_monitfifo(i),
12      rd_en        => rd_monitfifo(i),
13      dout         => MonitFifoData_i(i),
14      full         => MonitFifoFull_i(i),
15      almost_full  => open,
16      empty        => MonitFifoEmpty_i(i),
17      almost_empty=> open,
18      valid        => MonitFifoValid_i(i)
19    );
20  end generate gen_monit_fifo;
21
22  MonitFifoFull <= or_reduce(MonitFifoFull_i);
23
24  gen_MonWrite: for i in 0 to size_x_1 generate
25    Write:    wr_monitfifo(i) <= wr_mon_en and (not MonitFifoFull);
26  end generate gen_MonWrite;
27
28  dIn_monitfifo(0) <= conv_std_logic_vector(0, 13) & reg_Mon when mon_reg_flagWR =
      '1' else monit_data_in(0);
29
30  gen_dinMonFifo: for i in 1 to size_x_1 generate
31    dIn_monitfifo(i) <= (others => '0') when mon_reg_flagWR = '1' else monit_data_in(
      i);
32  end generate gen_dinMonFifo;
33
34  monit_block <= MonitFifoFull;
35
36  ------------------------- WRITING FSM ------------------------
37  process(user_clk)
```

```
38  begin
39    if (rising_edge(user_clk)) then
40      case mn_state_wr is
41        when mon_IDLE =>
42          if (start_monit = '1') then
43            mn_state_wr <= mon_WRITE_reg;
44          else
45            mn_state_wr <= mon_IDLE;
46          end if;
47        when mon_WRITE_reg =>
48          if (MonitFifoFull = '1') then
49            mn_state_wr <= mon_WAIT1;
50          else
51            mn_state_wr <= mon_WRITE;
52          end if;
53        when mon_WAIT1 =>
54          if (write_monit = '0') then
55            mn_state_wr <= mon_WAIT1;
56          else
57            mn_state_wr <= mon_WRITE_reg;
58          end if;
59        when mon_WRITE =>
60          if (MonitFifoFull = '1') then
61            mn_state_wr <= mon_WAIT2;
62          elsif (start_monit = '1') then
63            mn_state_wr <= mon_WRITE_reg;
64          elsif (write_monit = '0') then
65            mn_state_wr <= mon_IDLE;
66          else
67            mn_state_wr <= mon_WRITE;
68          end if;
69        when mon_WAIT2 =>
70          if (write_monit = '0') then
71            mn_state_wr <= mon_WAIT2;
72          else
73            mn_state_wr <= mon_WRITE;
74          end if;
75        when others =>
76          mn_state_wr <= mon_IDLE;
77      end case;
78    end if;
79  end process;
80
81  -- Output depends solely on the current state
82  process(mn_state_wr)
83  begin
84    case mn_state_wr is
85      when mon_IDLE =>
86        wr_mon_en       <= '0';
87        mon_reg_flagWR <= '0';
88      when mon_WRITE_reg =>
89        wr_mon_en       <= '1';
90        mon_reg_flagWR <= '1';
91      when mon_WAIT1 =>
92        wr_mon_en       <= '0';
93        mon_reg_flagWR <= '0';
94      when mon_WRITE =>
95        wr_mon_en       <= '1';
96        mon_reg_flagWR <= '0';
97      when mon_WAIT2 =>
98        wr_mon_en       <= '0';
```

```
99          mon_reg_flagWR <= '0';
100       when others     =>
101         wr_mon_en       <= '0';
102         mon_reg_flagWR <= '0';
103     end case;
104   end process;
105
106   ------------------------ READING FSM ------------------------
107
108   -- Set the constant flag basing on the number of PE (odd or even)
109   size_flag_odd: if ( (size_x mod 2) = 1 ) generate
110     constant size_PE_odd : std_logic := '1';
111     begin
112     -- ...
113   end generate size_flag_odd;
114
115   size_flag_even: if ( (size_x mod 2) = 0 ) generate
116     constant size_PE_odd : std_logic := '0';
117     begin
118     -- ...
119   end generate size_flag_even;
120
121   -- Mealy fsm
122   -- Logic to advance to the next state
123   process(user_clk)
124   begin
125     if (rising_edge(user_clk)) then
126       case mn_state_rd is
127         when mn_IDLE =>
128           mn_state_rd <= mn_IDLE;
129           if (MonFifo_rd_en = '1' and MonFifo_empty_int = '0' and end_mon_packet =
      '0') then
130             mn_state_rd <= mn_READ_reg;
131           end if;
132         when mn_READ_reg =>
133           if (MonFifo_rd_en = '1') then
134             mn_state_rd <= mn_READ;
135           else
136             mn_state_rd <= mn_WAIT_reg;
137           end if;
138         when mn_WAIT_reg =>
139           if (MonFifo_rd_en = '1') then
140             mn_state_rd <= mn_READ;
141           else
142             mn_state_rd <= mn_WAIT_reg;
143           end if;
144         when mn_READ =>
145           if (MonFifo_rd_en = '1' and end_mon_packet = '0') then
146             mn_state_rd <= mn_READ_WRITE;
147           else
148             mn_state_rd <= mn_IDLE;
149           end if;
150         when mn_READ_WRITE =>
151           if MonFifo_rd_en = '1' and end_mon_packet = '0' then
152             mn_state_rd <= mn_READ_WRITE;
153           else
154             mn_state_rd <= mn_EMPTY;
155           end if;
156         when mn_WAIT =>
157           mn_state_rd <= mn_WAIT;
158           if MonFifo_rd_en = '1' then
```

```
159             mn_state_rd <= mn_WAIT1;
160           end if;
161         when mn_WAIT1 =>
162           if (end_mon_packet = '1') then
163             mn_state_rd <= mn_IDLE;
164           else
165             mn_state_rd <= mn_WAIT2;
166           end if;
167         when mn_WAIT2 =>
168           if (MonFifo_rd_en = '0') then
169             mn_state_rd <= mn_WAIT;
170           elsif (end_mon_packet = '1') then
171             mn_state_rd <= mn_IDLE;
172           else
173             mn_state_rd <= mn_READ;
174           end if;
175         when mn_EMPTY =>
176           mn_state_rd <= mn_IDLE;
177           if MonFifo_rd_en = '0' then
178             mn_state_rd <= mn_WAIT;
179           end if;
180         when others =>
181           mn_state_rd <= mn_IDLE;
182       end case;
183     end if;
184   end process;
185
186   mon_cond   <= '1' when (monFifo_count =(size_x_1)or monFifo_count =(size_x_1 - 1))
187   and (monArray_count = size_y_1) else '0';
188   mon_cond_d <= '1' when (monFifo_count_d =(size_x_1) or monFifo_count_d =
189   (size_x_1 - 1))and(monArray_count = size_y_1) else '0';
190   mon_cond1  <= '1' when (monFifo_count = (size_x_1) )and(monArray_count = size_y_1)
191   else '0';
192
193  -- Process to set "end_mon_packet", that is used to stop the monitoring
194  -- transmission relative to one instruction after the right number of
195  -- data have been already transmitted
196   end_mon_process: process(user_clk)
197   begin
198     if (rising_edge(user_clk)) then
199       if (reset = '1' or monit_read_stop = '0' ) then
200         end_mon_packet <= '0';
201       else
202         if (tx_mn_valid = '0' and mon_cond = '1') then
203           end_mon_packet <= '1';
204         end if;
205       end if;
206     end if;
207   end process;
208
209   rst_counters <= '1' when tx_mn_valid = '0' and mon_cond_d ='1' else '0';
210
211   -- Mon read enable assignment
212   gen_dec_mon: for i in 0 to size_x_1 generate
213     rd_monitfifo_int(i) <= '1' when
214   (monFifo_count = i or (monFifo_count1 =  i and (mon_cond1 = '0')))
215   else '0';
216   end generate gen_dec_mon;
217
218   MonFifo_empty_int <= and_reduce(MonitFifoEmpty_i);
219   MonFifoValid_int  <= not MonitFifoValid_i(conv_integer(unsigned(monFifo_count_d)));
```

```
220   monit_busy          <= not(MonFifo_empty_int);
221   MnFIFO_Empty        <= MonFifo_empty_int;
222
223   -- "mon_reg_flag" is necessary in order to set to '1' all "read_enable"
224   -- signals when the number of the monitored register (the data in the
225   -- first column and first row) needs to be transmitted
226   gen_rd_MonFIFO: for i in 0 to size_x_1 generate
227     rd_monitfifo(i) <= (rd_monitfifo_int(i) and
228   (not end_mon_packet)and MonFifo_rd_int)or(mon_reg_flag and wait_docc_mn);
229   end generate gen_rd_MonFIFO;
230
231   process(user_clk)
232   begin
233     if (rising_edge(user_clk)) then
234       if (reset = '1' or rst_counters = '1') then
235         monFifo_count_d  <= (others => '0');
236         monFifo_count1_d <= conv_std_logic_vector(1,log2_size_x_1);
237       elsif (monCount_en = '1') then
238         monFifo_count_d  <= monFifo_count;
239         monFifo_count1_d <= monFifo_count1;
240       end if;
241     end if;
242   end process;
243
244   -- Process to count an index used to switch from one monitoring FIFO to another one
245   MonFIFO_count_process: process(user_clk)
246   begin
247     if (rising_edge(user_clk)) then
248       if (reset = '1' or rst_counters = '1') then
249         monFifo_count  <= (others => '0');
250         monFifo_count1 <= conv_std_logic_vector(1,log2_size_x_1);
251       elsif (monCount_en = '1') then
252         if(monFifo_count < size_x_1 - 1) then
253           monFifo_count <= monFifo_count + 2;
254         else
255           monFifo_count <= 1 - (size_x_1 - monFifo_count);
256         end if;
257         if(monFifo_count1 < size_x_1 - 1) then
258           monFifo_count1 <= monFifo_count1 + 2;
259         else
260           monFifo_count1 <= 1 - (size_x_1 - monFifo_count1);
261         end if;
262       end if;
263     end if;
264   end process;
265
266   ArrayCount_en <= '1' when tx_mn_valid = '0' and (monFifo_count_d = (size_x_1) or
        monFifo_count_d = (size_x_1 - 1)) else '0';
267
268   -- Process to keep track of the number of the monitoring data transmitted (of one
        fifo)
269   Array_counter_process: process(user_clk)
270     begin
271       if (rising_edge(user_clk)) then
272         if (reset = '1' or rst_counters = '1') then
273           monArray_count <= (others => '0');
274         elsif (ArrayCount_en = '1') then
275           if (monArray_count < size_y_1) then
276             monArray_count <= monArray_count + 1;
277           else
278             monArray_count <= (others => '0');
```

```
279           end if;
280         end if;
281       end if;
282     end process;
283
284   -- Mealy machine
285   process(mn_state_rd, MonFifo_rd_en, end_mon_packet)
286   begin
287     case mn_state_rd is
288       when mn_IDLE =>
289         monCount_en      <= '0';
290         monit_read_stop <= '0';
291         mon_reg_flag     <= '0';
292         MonFifo_rd_int   <= '0';
293         wait_docc_mn     <= '1';
294       when mn_READ_reg =>
295         monCount_en      <= '0';
296         monit_read_stop <= '1';
297         mon_reg_flag     <= '1';
298         MonFifo_rd_int   <= '0';
299         wait_docc_mn     <= '1';
300       when mn_WAIT_reg =>
301         monCount_en      <= '0';
302         monit_read_stop <= '1';
303         mon_reg_flag     <= '1';
304         MonFifo_rd_int   <= '0';
305         wait_docc_mn     <= '0';
306       when mn_READ =>
307         if (MonFifo_rd_en = '1' and end_mon_packet = '0') then
308           monCount_en <= '1';
309         else
310           monCount_en <= '0';
311         end if;
312         monit_read_stop <= '1';
313         MonFifo_rd_int   <= '1';
314         mon_reg_flag     <= '0';
315         wait_docc_mn     <= '1';
316       when mn_READ_WRITE =>
317         if (MonFifo_rd_en = '1' and end_mon_packet = '0') then
318           monCount_en <= '1';
319         else
320           monCount_en <= '0';
321         end if;
322         MonFifo_rd_int   <= '1';
323         monit_read_stop <= '1';
324         mon_reg_flag     <= '0';
325         wait_docc_mn     <= '1';
326       when mn_WAIT =>
327         monCount_en      <= '0';
328         monit_read_stop <= '1';
329         MonFifo_rd_int   <= '0';
330         mon_reg_flag     <= '0';
331         wait_docc_mn     <= '1';
332       when mn_WAIT1 =>
333         if (end_mon_packet = '1') then
334           monCount_en <= '0';
335         else
336           monCount_en <= '1';
337         end if;
338         monit_read_stop <= '1';
339         MonFifo_rd_int   <= '0';
```

```
340           mon_reg_flag     <= '0';
341           wait_docc_mn     <= '0';
342         when mn_WAIT2 =>
343           if (end_mon_packet = '1' or MonFifo_rd_en = '0') then
344             monCount_en <= '0';
345           else
346             monCount_en <= '1';
347           end if;
348           monit_read_stop <= '1';
349           MonFifo_rd_int  <= '1';
350           mon_reg_flag     <= '0';
351           wait_docc_mn     <= '0';
352         when mn_EMPTY =>
353           monCount_en      <= '0';
354           monit_read_stop <= '1';
355           MonFifo_rd_int  <= '0';
356           mon_reg_flag     <= '0';
357           wait_docc_mn     <= '1';
358         when others =>
359           monCount_en      <= '0';
360           monit_read_stop <= '0';
361           MonFifo_rd_int  <= '0';
362           mon_reg_flag     <= '0';
363           wait_docc_mn     <= '1';
364       end case;
365     end process;
366
367     -- Mux in order to transmit all zeros when: 1) the number of the
368     -- monitored register (the data in the first column and first row)
369     -- needs to be transmitted and 2) when there is an odd number of
370     -- column and so the less significant 16 bits of "MonFifoData" need
371     -- to be set to '0'
372     MonFifoData_out1(15 downto 0)   <= MonitFifoData_i(conv_integer(unsigned(
          monFifo_count_d)));
373     MonFifoData_out2(15 downto 0)   <= (others=>'0') when (end_mon_packet = '1'and
          monFifo_count_d =(size_x_1))or(mon_reg_flag = '1')
374                                     else MonitFifoData_i(conv_integer(unsigned(
          monFifo_count1_d)));
375     MonFifoData(15 downto 0 )         <= MonFifoData_out1(15 downto 0);
376     MonFifoData(31 downto 16)         <= MonFifoData_out2(15 downto 0);
377
378     tx_mn_valid <= (MonFifoValid_int and wait_docc_mn)or(not MonFifo_rd_en) or
          mn_stop_tx;
379     mn_valid_out <= not(tx_mn_valid);
```

Bibliography

[1] A. Sripad, G. Sanchez, M. Zapata, V. P., Taho Dorta, S. Cambria, A. Marti, K. Krishnamourthy, and J. Madrenas. «SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture». In: *Neural networks* 97 (Nov. 2018), pp. 28–45 (cit. on pp. 1, 2, 12–14, 22, 33).

[2] N. Zheng and P. Mazumder. *Learning in Energy-Efficient Neuromorphic Computing*. IEEE Press., 2020 (cit. on pp. 1, 2, 5–9, 11–13).

[3] J. L. Lobo, J. Del Ser, A. Bifet, and N. Kasabov. «Spiking Neural Networks and online learning: An overview and perspectives». In: *Neural networks* 121 (2020), pp. 88–100 (cit. on pp. 1, 5–11).

[4] M. Zapata, T. Dorta, J. Madrenas, and G. Sánchez. «AER-SRT: Scalable spike distribution by means of synchronous serial ring topology address event representation». In: *Neurocomputing* 171 (2016), pp. 1684–1690 (cit. on pp. 2, 22).

[5] M. Zapata. «Arquitectura Escalable SIMD con Conectividad Jera'rquica y Reconfigurable para la Emulacio'n de SNN». 2016 (cit. on pp. 3, 14–18, 20, 22, 25–27).

[6] W. Gerstner and W. M. Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge University Press., 2002 (cit. on pp. 3–5, 9–11).

[7] A. L. Hodgkin and A. F. Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». In: *The Journal of Physiology* 171 (Aug. 1952), pp. 500–544 (cit. on p. 6).

[8] E.M. Izhikevich. *Dynamical systems in neuroscience*. MIT Press., 2007 (cit. on p. 7).

[9] B. Petro, N. Kasabov, and R. M. Kiss. «Selection and Optimization of Temporal Spike Encoding Methods for Spiking Neural Networks». In: *IEEE transactions on neural networks and learning systems* 31(2) (Feb. 2020), pp. 358–370 (cit. on pp. 8, 9).

[10] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley, 1949 (cit. on p. 11).

[11] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. «A wafer-scale neuromorphic hardware system for large-scale neural modeling». In: *IEEE International Symposium on Circuits and Systems (ISCAS)* (2010), pp. 1947–1950 (cit. on p. 12).

[12] A.W.Smith, L.J.McDaid, and S.Hall. «A compact spike-timing-dependent-plasticity circuit for floating gate weight implementation». In: *Neurocomputing* 124 (Jan. 2014), pp. 210–217 (cit. on p. 12).

[13] H. Ekkehard, M. Diesmann M.-O. Gewaltig, and A. Morrison. «NEST: The Neural Simulation Tool». In: *Encyclopedia of Computational Neuroscience* (Apr. 2013), pp. 1–4 (cit. on p. 13).

[14] R. Hoang, D. Tanna, L. Jayet Bray, S. Dascalu, and F. Harris. «novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling». In: *Frontiers in Neuroscience* 7 (2013), p. 19 (cit. on p. 13).

[15] S. W. Moore, P. J. Fox, S. J. Marsh, A. Mujumdar, and al. «Bluehive-a fieldprogramable custom computing machine for extreme-scale real-time neural network simulation». In: *In 2012 IEEE 20th annual international symposium on, field programmable custom computing machines. (FCCM)* (2012), pp. 133–140 (cit. on p. 13).

[16] C. Zamarreno-Ramos, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco. «Multicasting Mesh AER: A Scalable Assembly Approach for Reconfigurable Neuromorphic Structured AER Systems. Application to ConvNets». In: *IEEE Transactions on Biomedical Circuits and Systems* 7(1) (June 2012), pp. 82–102 (cit. on p. 13).

[17] A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodriguez, A. Jiménez, M. Rivas, and G. Jiménez. «On the AER convolution processors for FPGA». In: *Proceedings of 2010 IEEE international symposium on circuits and systems* (2010), pp. 4237–4240 (cit. on p. 13).

[18] M. Davies, N. Srinivasa, T.H. Lin, and al. «Loihi: a neuromorphic manycore processor with on-chip learning». In: *IEEE Micro* 38(1) (2018), pp. 82–99 (cit. on p. 13).

[19] J. Madrenas. «HEENS_info_v0.1». In: *Private document* (2020) (cit. on pp. 18, 20, 31, 43).