Neural and Synaptic modelling on bio-inspired hardware

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
and to the Department of
Electronics and Telecommunication Engineering of
Politecnico di Torino
by

Geremia Muccioli

In partial fulfillment
of the requirements for the master in
ELECTRONIC ENGINEERING

Advisor: Jordi Madrenas
Advisor: Claudio Passerone
Barcelona, Date 25/10/2023

Abstract

The presented thesis proposes to explore the implementation of different neural applications, in particular, the *Adaptive Exponential Integrate and Fire* (aEIF) neural model on a neuromorphic device called *HEENS*, and a simulation of a Spiking Neural Network with a Reservoir topology, along with the comparison of the results with an analogue neural counterpart, implemented in CMOS technology. For doing so, initially, some basic concepts about neuron's modeling and Spiking Neural Network are exposed, and then *HEENS* multiprocessor is introduced, both in the architecture and its software support. Afterwards, the focus is moved toward four different spiking neural models, explaining some theory and their equations, and for one of them, also the *HEENS* implementation. Lastly, a comparison between an analogue and a digital technologies implementing the same model over a reservoir network topology is discussed, presenting similarities and differences of the two approaches.

Contents

1 Introduction

During the last few decades, technological progresses made possible the manufacturing of sophisticated biologically inspired electronic devices, that propose to emulate the behaviour of the organical world and supported also by the mathematical models that have been developed during the years. In particular, the human brain and its capabilities have attracted lot of attention from the scientific world, and because of its wide amount of unique capabilities, many efforts have been made in order to reproduce the behaviour of brain cells.

In this domain, it can be found *HEENS*, acronym for *Hardware Emulator of Evolved Neural System*, a custom digital multi-processor device designed to simulate Spiking Neural Networks, featuring high parallelism capabilities, a dedicated instruction set and a communication systems that allows for the scaling to bigger networks.

Due to their biologically-inspired approach to information processing, Spiking Neural Networks (SNNs) have emerged as a valid alternative to the traditional Artificial Neural Networks (ANNs), because, unlike ANNs that rely on continuous activation functions, SNNs emulate the behavior of biological neurons by transmitting information through discrete spikes, that are SNNs fundamental units of computation, and that represent the firing of a neuron.

The structure of the thesis, after a brief introduction of some minimal concepts, is to present *HEENS* multi-processor, its main features and its software support, keeping the focus on the aspects that have been more involved in developing the applications.

Once exposed the device, some neural models are reviewed, as an introduction for the implementation of one of them, the Adaptive Exponential Integrate and Fire (aEIF).

Lastly, a 16 neurons Spiking Neural Network is simulated on *HEENS*, and the results are compared with an analogue device made in CMOS technology, also able to reproduce neural models.

1.1 Biological Neurons

A biological neuron, or nerve cell, is an electrically excitable cell able to receive and transmit electric signals toward other cells by means of various components, among which there are dendrites, axons and synapses.

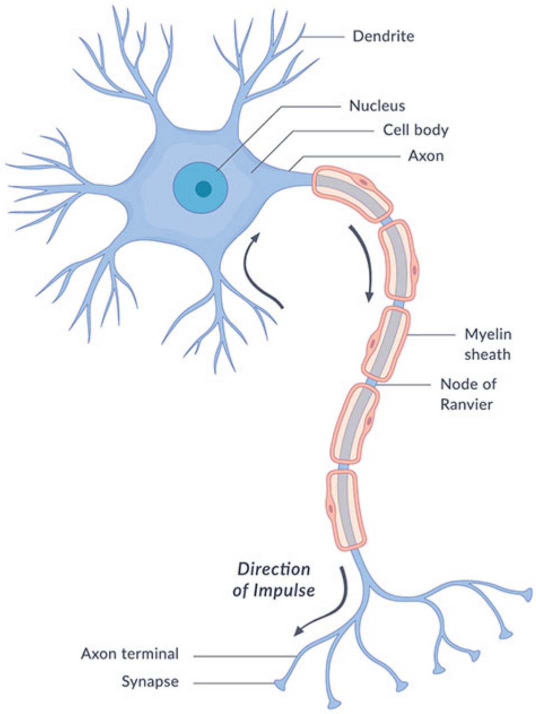

Figure 1: Neuron Structure [2]

Dendrites are extensions of the neuron that receive incoming electrochemical signals from other neurons or the external environment. These incoming signals are then transported toward the cell body where they can be processed and integrated.

In contrast, axons are extensions of the neuron that serve as electrical conductors for outgoing signals, but while many dendrites can be found in a cell, only a single axon can be present. As the axon extends, it may branch into multiple terminal regions, each of which ends with a synapse, that is the actual joint between a cell and the dendrites of the successive ones.

Inside the synapses can be found voltage-gated channels, that through complex biochemical reactions are able to be opened or closed depending on the concentration of some chemicals called neurotransmitters, in particular Sodium and Potassium ions.

The concentration of these elements in the neuron creates a voltage difference across the cell's membrane, which, in turn, regulates the generation of the electrical pulses: indeed, when this difference overcomes a biological threshold, a flow of ions is allowed to pass through the channels in the synapses, that are also able to modulate the strength of the signal, and effectively transmitting the pulses to the connected cells.

The variation over time of the voltage difference in the cell's membrane, referred to as membrane potential, has a deeply non-linear behaviour, which is a key requirement for providing functional capabilities, and an example of that behaviour can be seen in figure 2.

In absence of stimuli, the value of the voltage difference across the cell tries to reach its resting state, or resting potential. When incoming spikes are received by the neuron, the membrane potential is increased or decreased, depending on the reactions happening in the synapses, and when the value overcomes a voltage threshold, a spike is generated by the cell: for this reason, a neuron fires only when stimulated enough, otherwise the membrane potential return to its resting state, as indicated in figure 2 as "failed initiations".

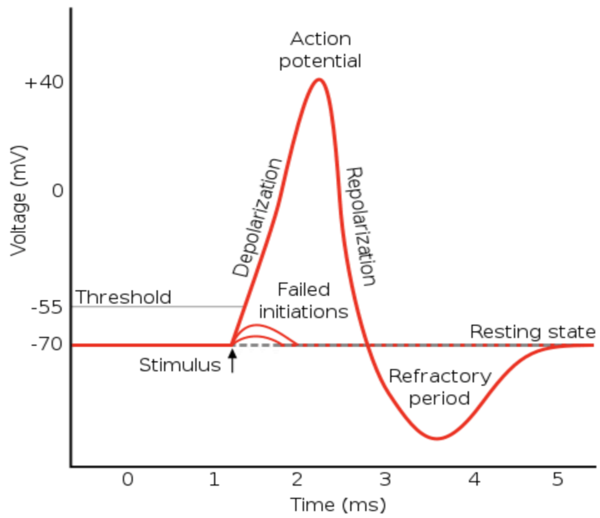

Figure 2: Membrane voltage of a neuron over time [16]

After the firing, the neuron enters in a refractory period, during which, due to other biological processes, it is harder for the cell to generate a subsequent spike and thus the activity of the cell is limited. Finally, after some time, the neuron returns to its resting state.

Furthermore, the amount of increment or decrement in the membrane potential associated to a received spike is dynamically controlled through the synapses, and this phenomenon, referred as synaptic plasticity, is directly associated to the learning process of the networks and their ability to recognize input patterns. [1]

1.2 Artificial Neurons

Artificial neurons aim to replicate the behaviour of biological neurons through the evaluation of non-linear mathematical functions applied to a variable known as the membrane potential, which is the parameter that reproduces the electrical charge difference across the cell's membrane. The non-linear function is also referred to as activation function, reminding to the ability to generate an output signal only if its variable, i.e. the membrane potential, exceeds a certain threshold. [1]

For traditional neural network applications, some common activation functions are, for

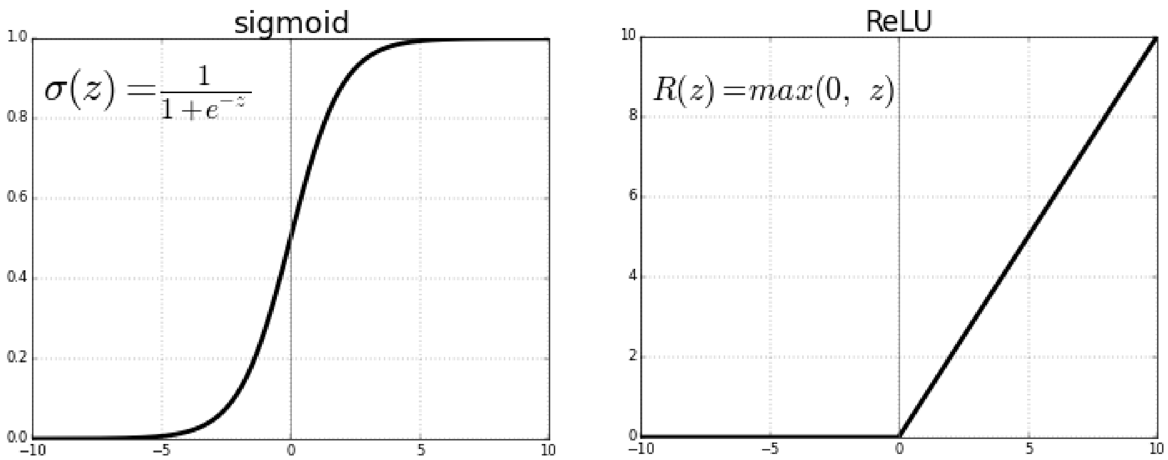

Figure 3: Sigmoid and Relu functions

example, the Sigmoid or the Relu (figure 3), both non-linear, and that are very effective when applied to solve tasks such as pattern recognition or classification, but lacking the actual mimicry of biological neural processes, and indeed, the output of such neurons does not generate a sequence of spikes, but it's associated to a continuous value.

On the contrary, neural models for SNNs applications have been developed in order to produce firing pattern outputs, but the complexity of their implementation scales with the biological accuracy desired.

Differently from the activation functions cited before, the output of spiking neuron is a sequence of pulses in which every vertical line is a generated spike. [2]

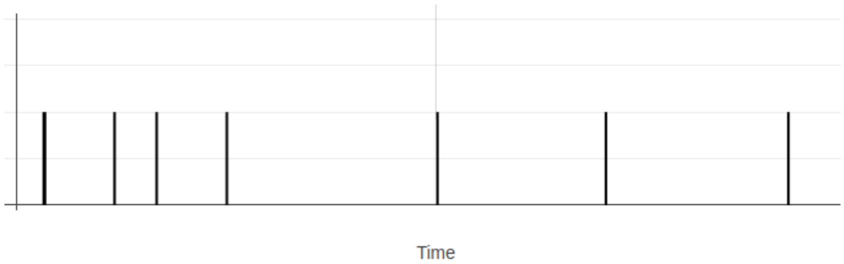

Figure 4: Example of a spiking output pattern

On the other side, synapses are modeled as weighted connections between two neurons, both for ANNs and SNNs applications, and the learning process is performed by adjusting these weights.

Even tough some models exist for modifying these values, such as *Spike-timing-dependent plasticity*, but since they are not strictly related to the scope of this work, they won't be treated.

1.3 Neural Networks and Reservoir Computing

A neural network can be defined as a graph in which the nodes are the neurons and the weighted interconnections are the synapses. The key feature of a neural network is to output a value that can be associated to a specific input, allowing the network to perceive relationships within data.

This ability to recognize patterns is achieved through the tuning of the synaptic weights, and by adjusting them during a learning process, the neural network can accurately map inputs to outputs. This mechanism enables the network to generalize from examples and recognize input patterns, making it capable of solve tasks like classification, image recognition, language processing and others.

In fact, before testing a neural network for a specific task, it first needs to be trained with known sets of data, in order to correctly adjust the weights for the particular application. One common way of implementing a neural network is to arrange neurons in a multi-layered structure, forming what is referred to as a Feedforward Neural Network. This configuration comprehends an input layer, one or more hidden layers, and an output layer. Neurons within each layer are interconnected, with connections typically unidirectional, flowing from the input to the output layer. [2]

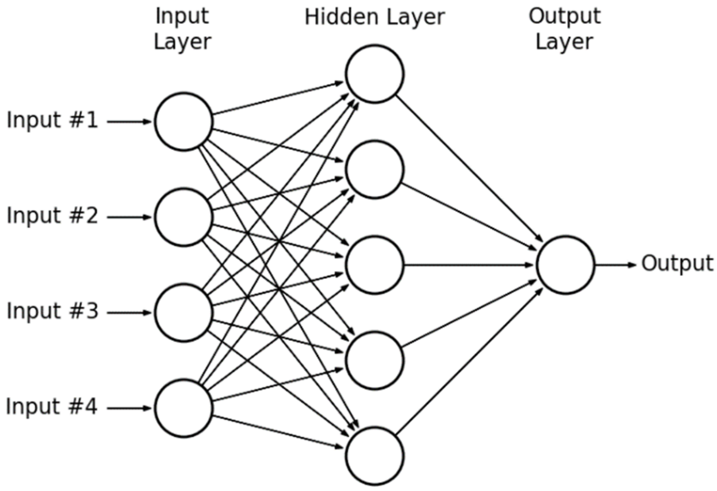

Figure 5: Traditional multi-layered network topology

This architecture facilitates the propagation of information through successive layers, where each neuron processes input data and contributes to the network's overall computation.

Among the different techniques for implementing a neural network, the one used for the simulation in chapter 5 belongs to the family of Reservoir Computing.

This particular class of networks can be seen as a multi-layer network, in which, the hidden layer, is a complete Spiking Neural Network with fixed synapses and synaptic weights. In

this way, the training phase involves less neurons and requires less computational efforts, and the task to be solved can be easily changed rearranging only the output weights.
Whereas in a traditional network the focus lies in training the synapses to accurately produce the solution for a given problem, Reservoir Computing takes a distinct approach, for that the network is free to produce an unsupervised output, that is then interpreted by a fully interconnected output layer, which is the only layer with trained connections, that, if correctly tuned, allows to whole system to perform tasks.

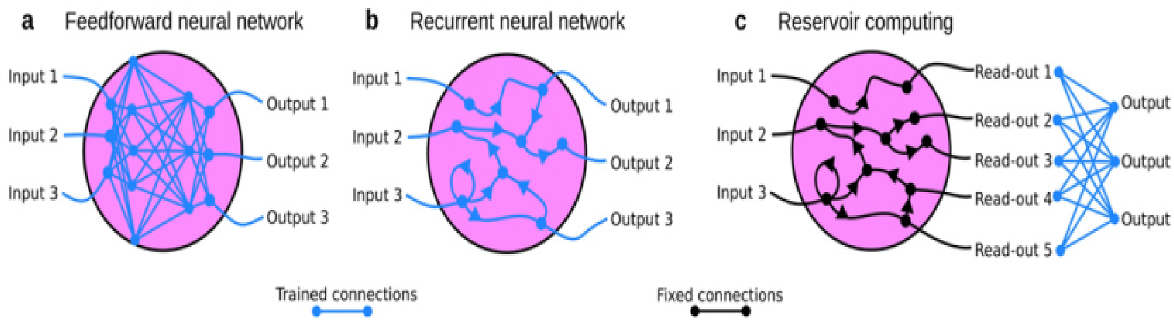

Figure 6: Comparison of different network's structure [3]

Usually, the neurons for the output layer are modeled differently with respect to the network's ones, with the purpose of linearizing the firing pattern in order to get more meaningful results.[3] This linearization can be performed in multiple ways, for example counting the total received spikes by an output neuron in a certain amount of time and use it as a probability for classification or activate only the output neuron that received the highest number of spikes.
For example, in figure 7, three different input patterns (black, orange and green) are fed to three neurons in the reservoir, and if the output weights are correctly tuned, after some time (3ms in this specific case), only one of the three output neurons will be active.

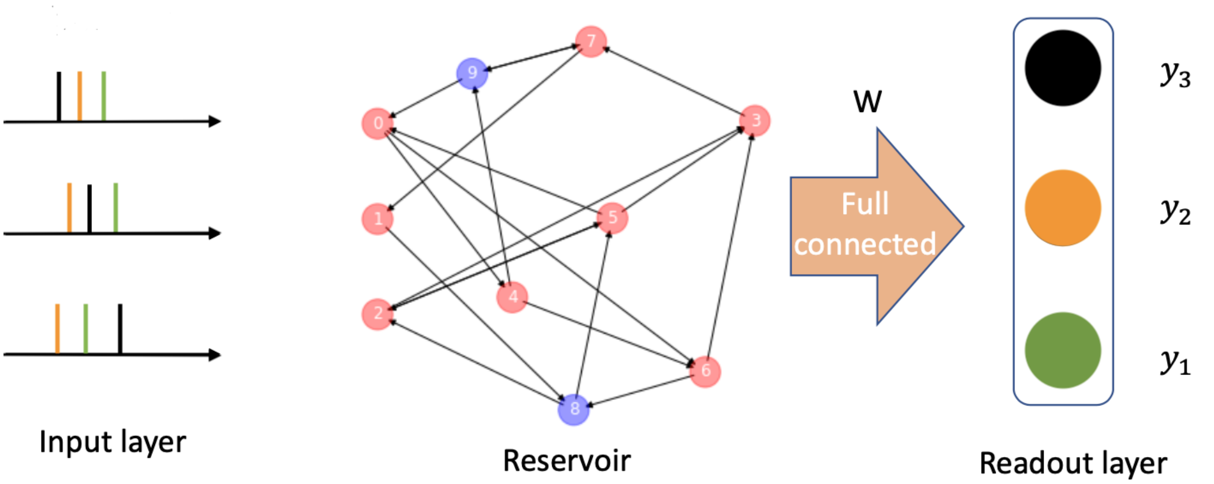

Figure 7: Example of a reservoir network

2 HEENS Architecture and Software Support

HEENS, acronym for *Hardware Emulator of Evolved Neural System*, is an electronic multi-processor chip with biological inspired features, able to simulate Spiking Neural Networks for real time applications.

The device is implemented on a FPGA, in order to be easily reprogrammed and to allow for the connection with a computer or even more boards.

One of its main characteristics is the high parallelism capability, achieved through a Single Instruction Multiple Data architecture, for that, a single sequencer sends the same instruction to an array of processing elements (PEs), and each of them, provided with its own memory and arithmetical unit, executes the algorithm at the same time. [4]

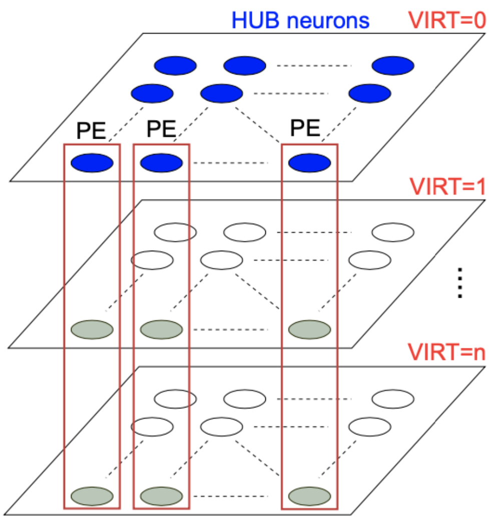

Figure 8: Virtualization mechanism for the PE's array [4]

Moreover, a single PE, by means of a virtualization mechanism, can represent and simulate more neurons within the same network, reducing the actual physical units and allowing for a larger topology. The idea behind this mechanism is to retrieve the state of a network's neuron, i.e. the membrane potential, compute the variables for the current time step, save back the new state, and jump to the next layer until all virtual layers have been evaluated. Usually, a single PE can simulate up to 8 different neurons, while maintaining an overall real-time behaviour, but this number can be scaled together with the hardware.

For example, an array of 4 PEs with 4 virtualization layers, can simulate an SNN of 16 neurons, where the first processing element holds the state for the neurons number 1, 5, 9, 13.

Furthermore, *HEENS* is provided with a serial communication system for spike distribution, *Address Event Representation over Synchronous Serial Ring*, or AER-SRT, which allows to connect all the neurons within a single board, but also more *HEENS* devices into a ring topology, in order to enlarge the neural network's size without real time functionality losses. In this configuration, one of the boards assumes the role of Master Chip (MC), controlling the traffic of the data over the communication bus, while the other boards are configured as Slave Chips or Neuromorphic Chips (NCs). These roles apply only to the communication system, and do not affect the behaviour or the simulation of the network. [4]

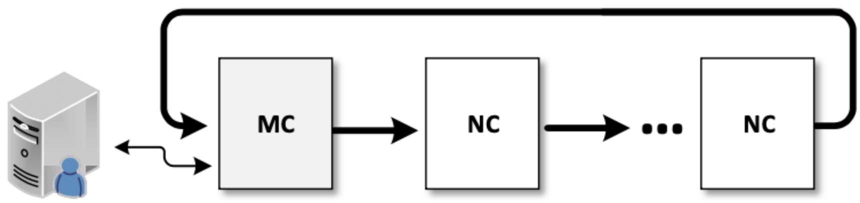

Figure 9: HEENS's ring topology [4]

Despite the existence of other neuromorphic architectures with larger sizes and higher computational capacities, *HEENS*'s strength lies in its flexibility and in the possibility to program the neuron freed from physical implementations, making it suitable for small topologies or for verification of networks and neural models.

2.1 Architecture

As can be seen in figure 10, *HEENS* is composed of three macro blocks, the control unit, an array of processing elements and the controller for the communication system.

The parallelism of the architecture is 16 bits, excepts for the machine instructions that are encoded with 32 bits.

The number are represented as integers or in fixed point numbers, and a floating point unit is not present in the processors.

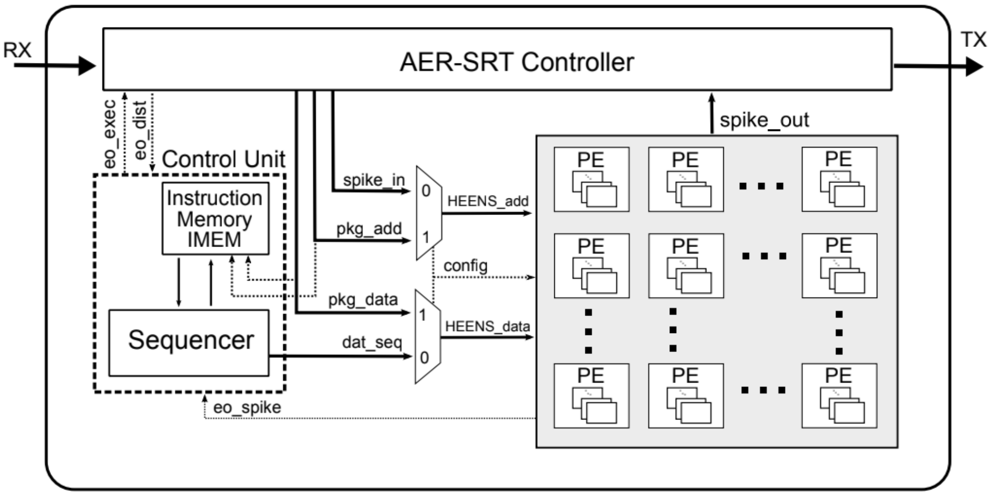

Figure 10: HEENS block diagram [4]

2.1.1 Control Unit

The control unit is in charge of managing the flow of execution of the program, and it's made of an Instruction Memory (IMEM) and a Sequencer.

The IMEM is used to store both machine instructions, i.e. the neural model's algorithm, and global constant variables, and it can be addressed with some specific instructions in order to retrieve those memory cells.

Since the instructions size is on 32 bits but the constants stored in the memory are on 16 bits, all the fetched constants are taken in couples, and this fact has to be taken into account when developing the algorithms.

The sequencer, instead, is the unit that actually manages the execution of the algorithms stored in the memory, and sends simultaneously data to the whole array of PEs. Also, it's in control of the configuration of the PEs, and it's capable of generating the signals for synchronizing and managing the communication bus towards the other possible connected devices.

2.1.2 Processing Element

Each PE inside the array is a digital processor, composed of some logic, a memory, a register file and an arithmetical unit. A single neuron in the SNN is addressed by a triplet of integers (v, r, c), where r,c represents a position in the PE array and v is the virtual layer. In figure 11 it is shown the architecture of a single Processing Element within the array:

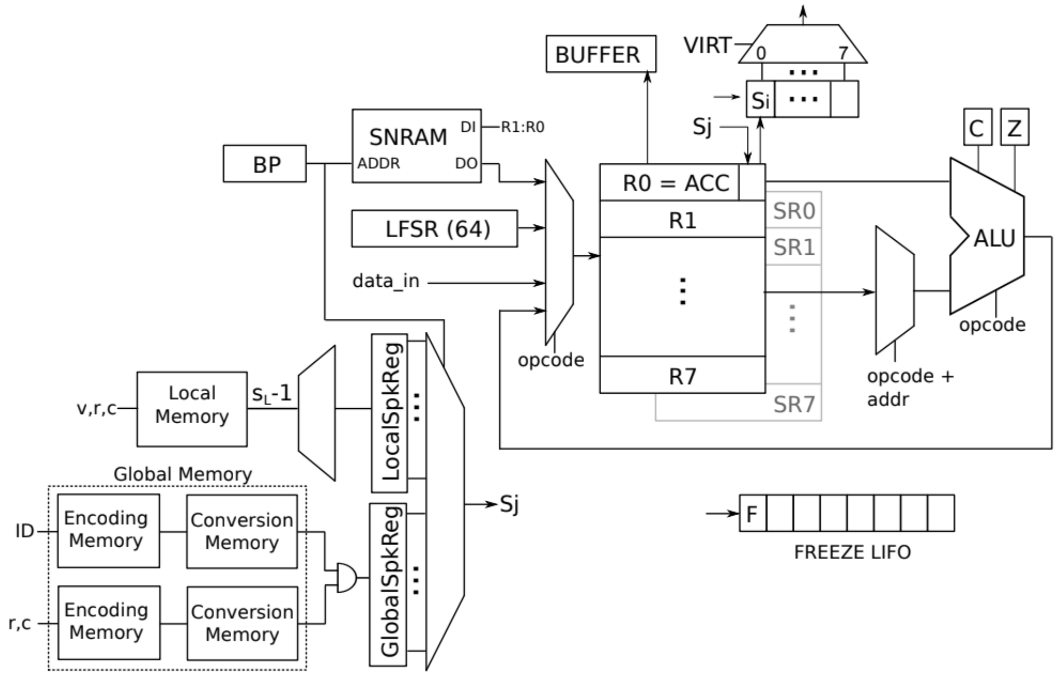

Figure 11: Processing Element Architecture [4]

- *Local Memory (SNRAM)*: Each PE is provided with its own *Synaptic and Neural memory*, used for storing local variables, and among which are found the membrane potential and all the incoming synaptic weights for each neuron's connections.

All the addresses in the memory keeps two words, i.e. 32 bits, thus the values stored or retrieved always involve two registers, R0 and R1. The memory is addressed through the *BP* pointer, that can be loaded with specific instructions.

This module has a central role also in virtualization because it keeps the data for all the neurons represented by the processing element, and indeed, the virtualization mechanism works by addressing differently this memory, as can be seen in the next table.

SNRAM Address	**16 MSBs**	**16 LSBs**
NEU_0 + 0	var1	var2
NEU_0 + 1	var3	var4
NEU_1 + 0	var1	var2
NEU_1 + 1	var3	var4
...	...	...
...	...	...
SYN_0 + 0	$synaptic\_weight_{00}$	S_{00}
SYN_0 + 1	$synapti\_weight_{10}$	S_{10}
SYN_0 + 2	$synaptic\_weight_{20}$	S_{20}
SYN_0 + 3	$synaptic\_weight_{30}$	S_{30}
SYN_1 + 0	$synaptic\_weight_{01}$	S_{01}
SYN_1 + 1	$synaptic\_weight_{11}$	S_{11}
...	...	...
...	...	...
NOISE SEED 0	seed MSBs	seed LSBs
NOISE SEED 1	seed MSBs	seed LSBs

Table 1: SNRAM configuration

All the virtual neurons in a PE have the same number of neural variables, because they depend on the mathematical model, but the same does not happen with synapses, that depend on the network's topology and whose amount can vary among neurons.

After the neural parameters, the synaptic weights can be found, stored in MSBs of the memory cell, while the LSBs contains the information of incoming spikes from that synapse. This information is stored as a logical value in the first bit of the LSBs.

Indeed, for example, if neuron number 0 receives a spike from neuron number 1, then S_{10} will be equal to 1, 0 otherwise.

Differently from neural variables, the number of synapses for a single neuron can differ from a neuron in another virtual layer, but the number of addresses used in the memory by each PE must be the same. In fact, accessing the memory for retrieving the synaptic weights for the current virtual layer is done by implementing a software loop over a constant defined at compile time, which is the same for all PEs, and that is calculated as the maximum number of synapses for the particular layer among all PEs. For example, if PE_0 needs 10 synaptic weights in the first virtual layer and PE_1 needs only 2, the memory slots used by both for that layer is equal to 10, and, for PE_1, eight addresses will be filled with weights equal to 0. In this way, the parallelism in the execution flow for all the PEs in the array is preserved.

Lastly, at the very end of the memory, some 32-bits seeds for noise generation are stored.

The main limitation is the size of this memory, indeed, the number of neurons in the network depends on the capacity of the single PE to store all the variables needed to execute the algorithm, i.e. the SNRAM must be large enough to contain all variables and synaptic weights for all virtual layers. This number can be large, and it scales rapidly with the number of neurons.

- *Register file*: The register file is composed of 8 registers with direct access, R0 to R7, and 8 shadow registers, SR0 to SR7, that cannot be directly addressed by the ALU or the sequencer, but they can only exchange data with their standard counterpart. Moreover, the register R0, also called Accumulator, is the main one in the register file, due to the fact that is always addressed as one of the inputs by the arithmetical unit, and it also stores every time the result of the operation. Even the *SNRAM* always involves register R0, because, along with R1, are loaded with the values retrieved from the memory.

Another very important feature of the register file is the possibility to be frozen (i.e. inactive) depending on the value of some flags present in the ALU: this is the technique that enables to differentiate the flow of the algorithms and makes possible performing *if statements*.

The current state of the register file is saved in a LIFO, and this operation can be nested at most 8 times.

- *Arithmetical Unit*: The ALU inside each processing element is able to perform 16 bits additions, subtractions, logical functions and multiplications. Divisions could be performed by multiplying for the reciprocal of the divisor.

  Each operation takes the value stored in R0 and a second register, and always saves the result back to R0.

  It's also important to highlight that sums saturate instead of going to overflow, and that the multiplications save the result in both R0 (MSBs of the result) and in R1 (LSBs of the result).

  In addition, the ALU is also provided with a *Zero* and a *Carry* flag, that can be used for controlling the algorithm's execution.

- *Spiking Logic*: From the point of view of a PE, the most important feature of the spike distribution system is that the information of a received spike is found, at each time step, in the *SNRAM*, in the first bit of the 16 LSBs of the synaptic memory cell, stored along the synaptic weight, as shown in table 1. When this value is read from the memory, it's saved in the LSB of R0, with the possibility of raising a flag when shifted. On the opposite, when a neuron is firing in the current cycle, the first bit of R0 is stored in a specific register and this value is distributed in the spike distribution phase.

- *Noise Generator (LSFR)*: Each processing element is also provided with a circuit for generating Gaussian noise, that has an important role in physical neural networks.

This circuit is a *Linear-feedback shift register*, that, starting from a seed, it's able to generate a very long sequence of pseudo-random numbers.

2.1.3 Communication system

The communication system in *HEENS* allows it to exchange data with other boards by means of the *AER-SRT Controller*, but also with a computer through an HDMI interface and it is programmed remotely by means of an Ethernet connection.
The HDMI protocol is used for visualizing results, and allows for the transmission of real time spikes from the device, with a dedicated user interface formed by the raster plot of the network, and the possibility to monitor up to 4 neurons. [6]
The Ethernet connections, instead, is used to remotely load the programs in the device, by specifying its IP address.

2.1.4 Operation phases

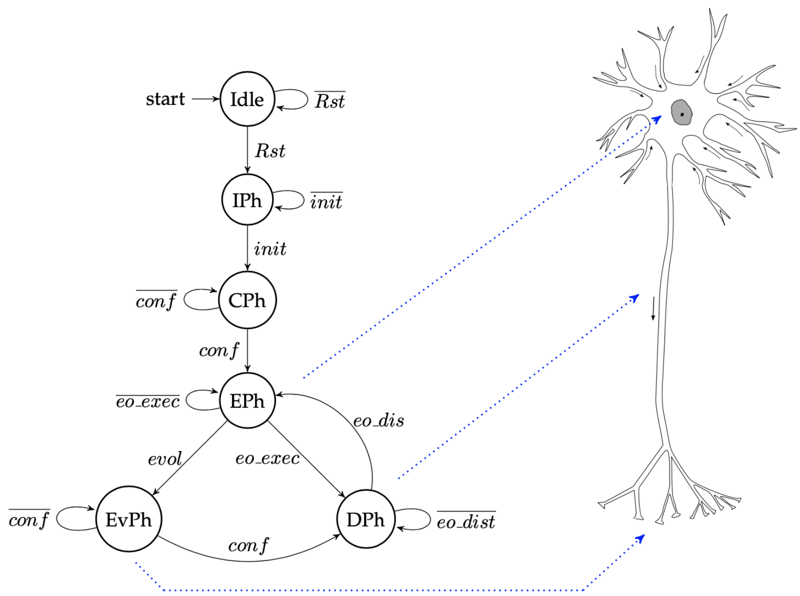

Figure 12: Operation Phases [5]

In order to simulate an SNN, after a first initialization and configuration phases, needed for both setting up the *HEENS* network and the SNN, the execution flow is divided in three different parts:

1. *Execution Phase (EPh)*: In this phase, each PE executes the neural algorithm for all the virtual layers, and evaluates the new values for the model's variables.

2. *Evolution Phase (EvPh)*: At the end of each cycle of execution a signal for evolution can be present, and the involved neurons are modified according to incoming information sent by the Master chip. This feature it's still under developing, but once completed, it should allow the network to dynamically rearrange its topology without the necessity of compiling again.

3. *Distribution Phase (DPh)*: The generated spikes from all the neurons are actually distributed through the communication bus, and the firing information is spread throughout the network.

Except for the initialization phases, *HEENS* is designed to simulate a network with a time step of 1ms, which is a constraint chosen for allowing the device to execute algorithms in a real time manner with biological plausibility. Therefore, a whole cycle of execution, evolution and distribution lasts that amount of time.
For this reason, the length of the executable program is limited to a maximum length, that can be derived from the clock frequency of the device. There are also some dedicated hardware modules able to synchronize the working frequency of the chip on 1ms when a single execution cycle is shorter, but this topic is beyond the purpose of this paper.

2.2 Software support

In order to develop *HEENS* applications and due to the custom nature of its architecture, it has also been developed a dedicated Instruction Set, a python compiler and two file formats to specify both the SNN topology and the neural model.

2.2.1 Instruction Set

The complete *HEENS* Instruction Set is shown in table 2, and each instruction belongs to one of the following categories:

- Sequencer: Instructions that are involved in the Sequencer or IMEM functioning, such as unconditional jumps.
- Register: Operations that can be performed on active registers, such as shifts or resets.
- Movement: Any operation involving the movement of data within registers or shadow registers. Also the noise configuration is part of this set.
- Flags: Instructions that can modify the value of the flags present in the ALU.
- Arithmetic: Instructions that involve the arithmetical unit.
- Logic: This class of instructions perform logical operations, such as ANDs or ORs.
- Conditional: These operations are used for freezing the register file and performing conditional operations.
- Others: Specific instructions implemented for specific functions in the chip, such as storing a spike, loading the SNRAM pointer or controlling the noise generator.

SEQ	REG	MOV	FLAG	ARITH	LOGIC	COND	OTHER
NOP	LDALL	LLSFR	SETZ	INC	AND	FREEZEC	LOADSP
LOOP	RST	MOVA	SETC	DEC	OR	FREEZENC	STOREB
LOOPV	SET	MOVR	CLRZ	ADD	INV	FREEZEZ	STORESP
ENDL	SHLN	SWAPS	CLRC	SUB	XOR	FREEZENZ	STOREPS
GOSUB	SHRN	MOVRS		MULU		UNFREEZE	LOADSN
RET	RTL	SEED		MULS			RANDON
HALT	RTR	MOVSR					RANDOFF
SPKDIS	SHLAN						LOADBP
READMP	SHRAN						SPMOV
RST_SEQ	BITSET						INCS
LAYERV	BITCLR						
GOTO							
INCV							
READMPV							
MARK							
SYNAPSE							

Table 2: *HEENS* instruction set

2.2.2 Network Netlist

The first of the two file formats developed for *HEENS* is for specifying the configuration of the neural network, its topology and the initial state of the SNRAM of processing element.

This file is composed of four sections:

1. **@Config:** needed for configuration, defines the board for uploading the *HEENS* architecture and the size of the SNN.

2. **@ParamSyn**: In this section are present the values that will be stored in the SNRAM in the positions pointed by each synapse.

3. **@Netlist**: Here can be found the synapses definition, in the form of pre-synaptic neuron and post-synaptic neuron separated by a comma, and with the possibility to also specify, after the second parameter, a specific value for that synapse, that will erase the one loaded in the previous section.

4. **@Params**: Used for initializing sequential memory cells in the neural section of the SNRAM, in order to store model variables. The addresses of these parameters are

saved inside the IMEM, and they can be retrieved through the apposite instructions.

As for the synapses, there is the possibility to differentiate the values received by each neuron, so that different kinds of neuron can be modeled in parallel.

This section should also define the seeds for initializing the noise generator, since they're also stored in the SNRAM memory.

Below, a general example of a netlist file, with different colors for each section, and its respective graph.

```
@Config     #board configuration
Zedboard_4x8   ;name of one possible board

@ParamSyn      #synaptic parameters definition
0, 1000        ;all the synaptic weights are set to 1000


@Netlist    #netlist definition
0 , 1            ; the same as 0, 1, 1000
1 , 2 , 2500
2 , 0 , 300


@Params     #neural parameters definition and seeds for noise generator
.0x1E3/16/NEU_VAR_1_2/$NVL/-7000, -1400
;All the neurons receive the values -7000 and -1400

.0x1E4/16/NEU_VAR_3_4/$NVL/1000, 0
0 , 2000, 500
1 , 0 , -500
;only neuron number 2 receives the values 1000 and 0

.0x3E0/32/SEED/2/ 100, 100
100, 100
```

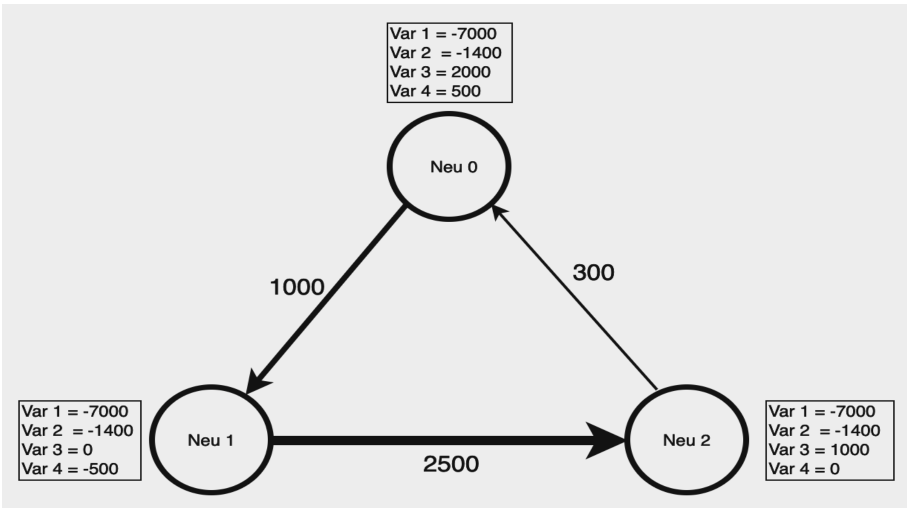

Figure 13: An example of a netlist file with its respective network's graph

2.2.3 Neural Model

The file format for the neural models is a typical assembly executable, with a section for global variables and another for the actual code, and it's entirely stored in the IMEM.
Despite the fact that each neural model is different, some recurrent structures should always be present for the correct behaviour of the neurons. In fact, before the starting point of the simulation, the virtual mechanism inside each PE must be initialized as shown in figure 14, and then, if required, the seeds should be given to the noise generator through the appropriate instructions.
Finally, the simulation loop is executed, and inside, the virtualization loop, that is repeated NVL times, along with the increment of the layer at the end by means of the instruction $INCV$, followed by the spike distribution (performed by $SPKDIS$) once the loop finishes. [5]

```
.org0x010
.data
GLOBAL_CONST = 1000

.org0x070
.code

GOTO MAIN

;;;;;;;;;;;;;;;;;;;;;;;;
; Functions definition ;
;;;;;;;;;;;;;;;;;;;;;;;;

MAIN:
    ; Virtual operation init
    LAYERV NVL          ; Init sequencer vlayers
    LDALL ACC, NVL      ; Load defined virtual layers to PE array
    SPMOV 0              ; VIRT <= ACC

    GOSUB NOISE_INIT    ; initialize noise gen.(optional)

; starting simulation
    SIM_LOOP:
    ;starting virtualization loop
       VIRT_LOOP:
           LOOP NVL
             ;;;;;;;;;;;;;;;;;;;;;;;
             ;neural model algorithm
             ;;;;;;;;;;;;;;;;;;;;;;;
             INCV    ;increment virtual layer
       ENDL    ;end of virtualization loop
       SPKDIS      ;spike distribution phase begin
       GOTO EXEC_LOOP  ; execute next time step
```

Figure 14: Neural model generic structure

2.2.4 Result Analysis

The analysis of *HEENS* results is done through a dedicated user interface, that, through an HDMI connection, can display on a screen the real time raster plot of the network and up to four other different parameters. [6]

A raster plot is a graph able to represent the firings of the neurons, by having on the x-axes the time and on the y-axes the number of the neurons, and when a neuron generates a spike, it's visualized a pulse on that instant of time for the given neuron.

By means of a FIFO, *HEENS* is able to communicate with the pc and send the necessary data for producing the raster plot of the net. In this way, it's possible to keep track of the behaviour of the network and up to four parameters, by storing them in the FIFO using the dedicated instruction *STOREB*.

In figure 15, an example of how the output is shown via HDMI. The implemented network has not a specific application, and it has been used only with the aim of presenting how the results are displayed.

The raster plot is the figure on top, but, since with the current implementation are always shown 256 neurons while only 16 have been used, the visualization is condensed and thus not much meaningful for this specific case. Anyway, *HEENS* has the possibility to generate a text file with the information of the raster plot, i.e. the neurons that fired at any given time, in order to check accurately the results when necessary.

The bottom graphs represent the membrane potentials over time of four different neurons, but any variable can be chosen and plotted here.

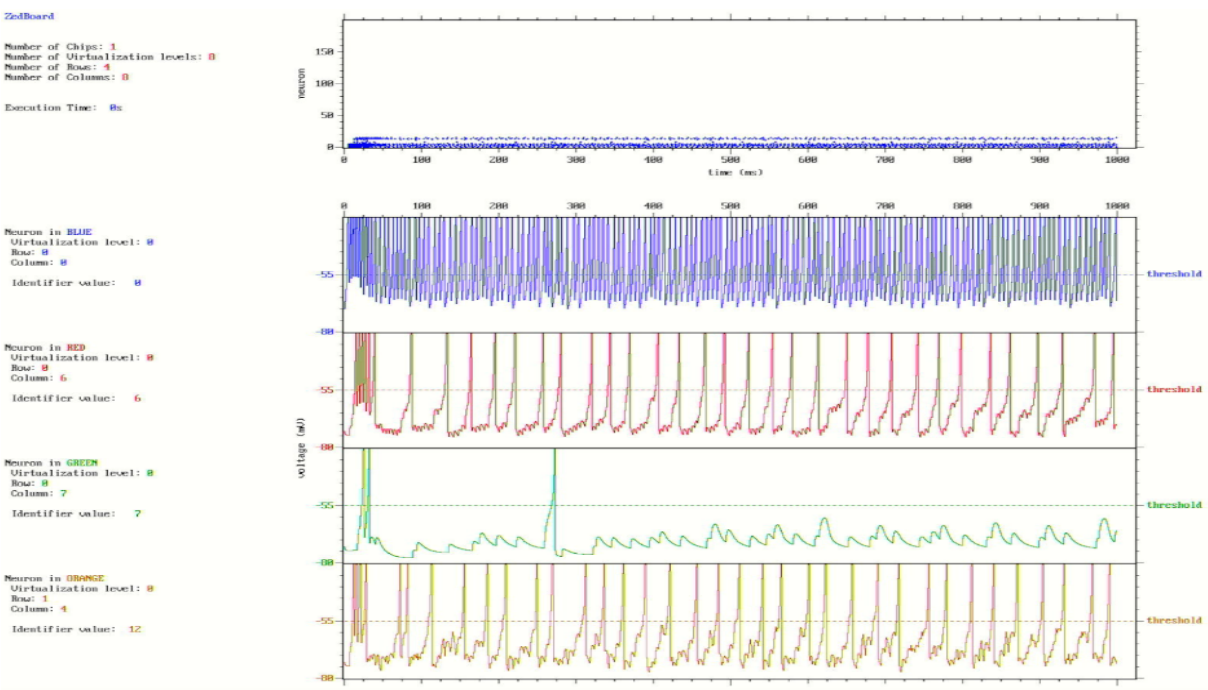

Figure 15: Example of HEENS visualization of results

3 Neural Models

Neural models are mathematical equations that reproduce the behavior of a biological neuron as a function of membrane potential and incoming spikes.
The amount of biological accuracy of the model drives the complexity of its implementation, in fact, in order to emulate the electrochemical reactions inside a neuron, lots of computational efforts are needed, slowing down the simulation and requiring lot of power. As a result, certain models opt for a simplified and cost-effective implementation, sacrificing some biological details. This, in turn, highlights the importance of choosing an appropriate model based on the specific requirements for the specific application.

3.1 Hodgkin-Huxley

Hodgkin-Huxley is one of the most important neural models that have been developed, describing the flow of Sodium and Potassium ions across a cell's membrane [7]. The strength and the weakness of the model lie together in its high biological accuracy, providing a realistic emulation of a neuron's behaviour, but with the drawback of a very demanding implementation. For this reason, is not commonly used for SNNs applications, where other simpler model are preferred.
The membrane potential evolution over time is described by a set of four differential equations:

$$C_m \frac{dV}{dt} = I_{Na} + I_K + I_{leak} + I_{ext} \tag{1a}$$

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V)m \tag{1b}$$

$$\frac{dh}{dt} = \alpha_h(V)(1-h) - \beta_h(V)h \tag{1c}$$

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n \tag{1d}$$

being:

$I_{Na} = g_{Na}m^3h(V - E_{Na})$ Sodium Current

$I_K = g_K n^4(V - E_K)$ Potassium current

$I_{leak} = g_{leak}(V - E_{leak})$ Leakage current

and with:

V : Membrane potential

C_m : Membrane capacitance

m : Variable for activation of Sodium gate

h : Variable for inhibition of Sodium gate

n : Variable for activation of Potassium gate

E_{Na}, E_K, E_{leak} : Reversal potentials

g_{Na}, g_K, g_{leak} : Channel conductances

α, β : Rate constants

In figure 16 are shown the dynamics of the variables of the model in response to an external step input current.

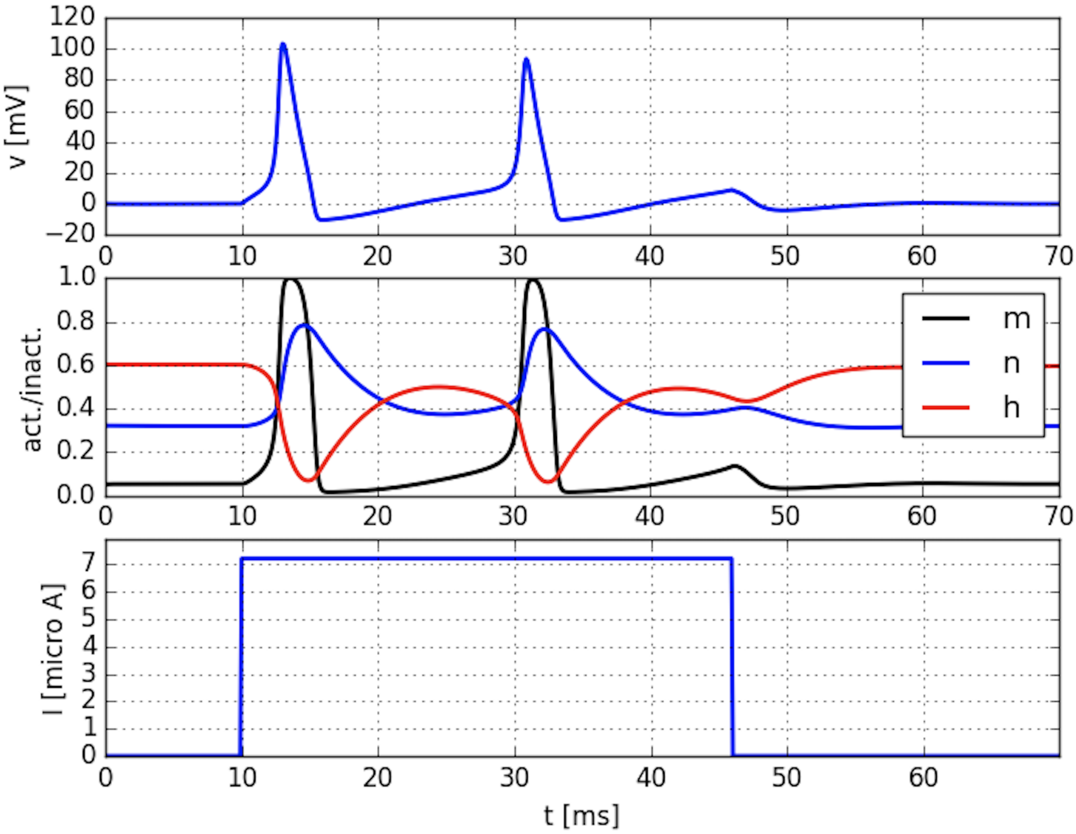

Figure 16: Dynamics of a Hodgkin-Huxley neuron [15]

3.2 Leaky Integrate and Fire

The Leaky Integrate and Fire (LIF), is the model with the simplest implementation, and its name reminds to the fact that the integration of received synaptic currents raises the membrane potential, but also that some current is leaking from the membrane. This duality makes the neuron firing only in presence of enough input spikes in a short period of time.
When the membrane voltage is above a threshold, the neuron fires and sends a spike, and then returns to its resting potential [8].

The equation describing this model is:

$$\tau_m \frac{dV}{dt} = -(V - V_{rest}) + I_{ext} \tag{2}$$

$$\text{if } V \geq V_{th} \text{ then } V = V_{rest}$$

and being:

τ_m : Decay constant

V : Membrane potential

V_{rest} : Resting potential

V_{th} : Threshold voltage

I_{ext} : External current

The dynamics of a LIF neuron (depicted in figure 17) are much simpler with respect to a Hodgkin-Huxley one, but still, the efficacy of its results makes it a reliable option for SNNs applications.

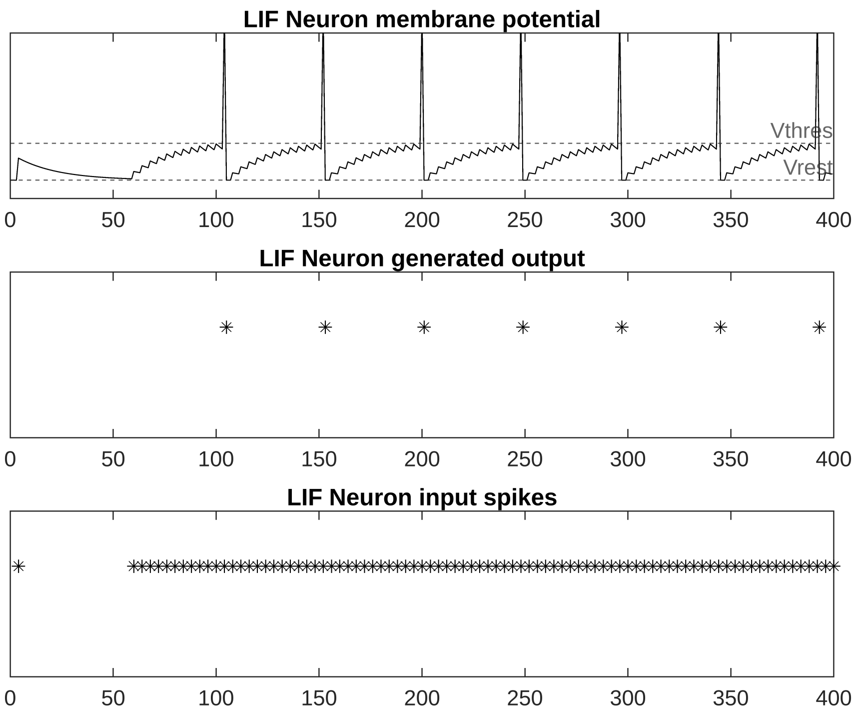

Figure 17: LIF neuron dynamics

Another important feature of this model is that a LIF neuron can be implemented in analogue with an RC circuit (figure 18), that has the key advantages of being very small and consuming very little power.

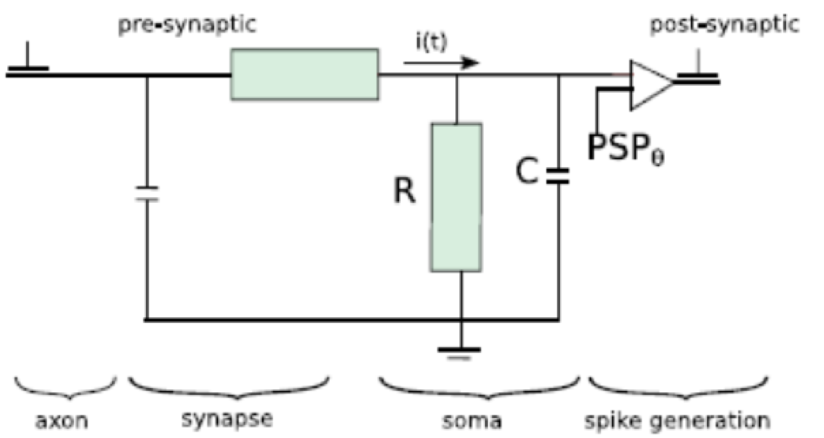

Figure 18: LIF neuron implemented as an RC circuit [5]

3.3 Izhikevich Model

The Izhikevich model is probably the most prominent neural model, proposing a very effective trade-off between implementation costs and biological accuracy.
This model has been obtained from the Hodgkin-Huxley one, it uses a set of two differential equations in order to mimic neural activity, and it's able to emulate different kinds of neural behaviours as shown in figure 19. [12]
For example, Regular Spiking, Intrinsically Bursting and Chattering are behaviour studied in cortical excitatory neurons, while Fast Spiking and Low-Threshold Spiking have been observed in cortical inhibitory neurons. The membrane potential is here a function not only of input spikes, but also of second state variable, called recovery potential, used for accounting also the recovery time needed for a neuron after a spike. [10]
The equations describing the model are:

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - U + I_{ext} \tag{3a}$$

$$\frac{dU}{dt} = a(bV - U) \tag{3b}$$

$$\text{if } V \geq V_{th} \text{ then } V = c \text{ and } U = U + d$$

and being:

V : Membrane potential

U : Recovery potential

a : Time scale of U

b : Sensitivity of U to subthreshold fluctuations of V

c : After-spike reset potential

d : After-spike increment of U

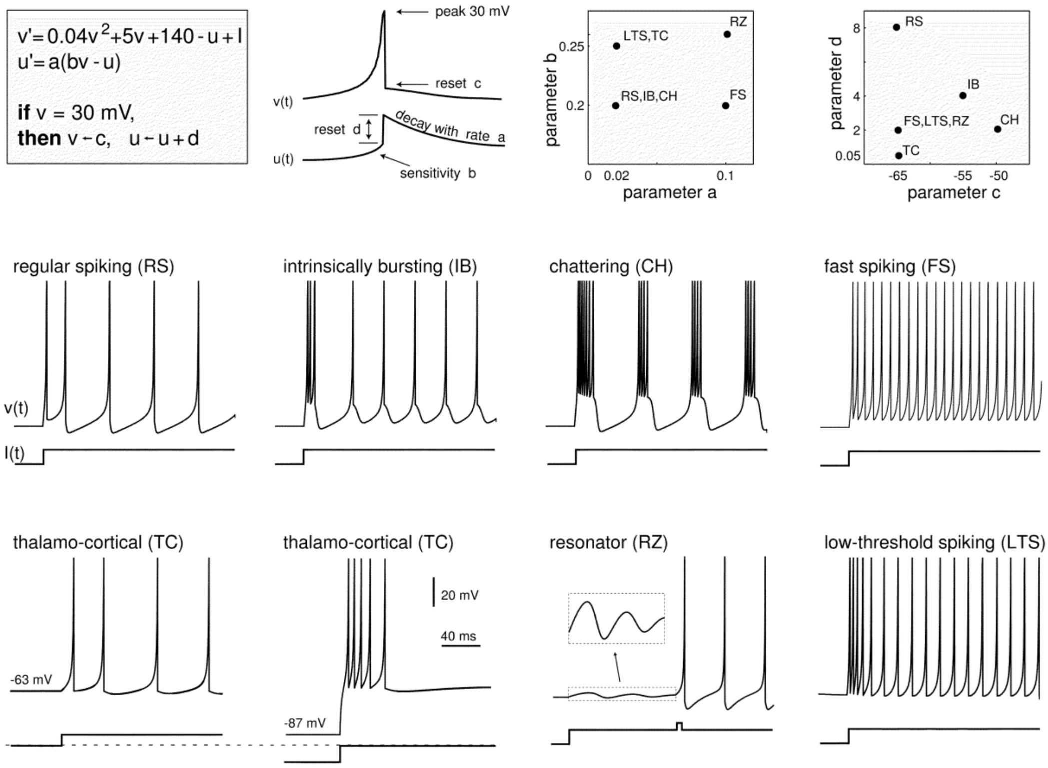

Figure 19: Dynamics of Izhikevich neurons [9]

Also for this model have been developed analogue circuits counterparts, as the one shown below in figure 20, in which the membrane and recovery potentials are modeled by the voltage difference across the two capacitors; besides, the increased complexity with regard to the RC-LIF analogue model (figure 18) is evident.

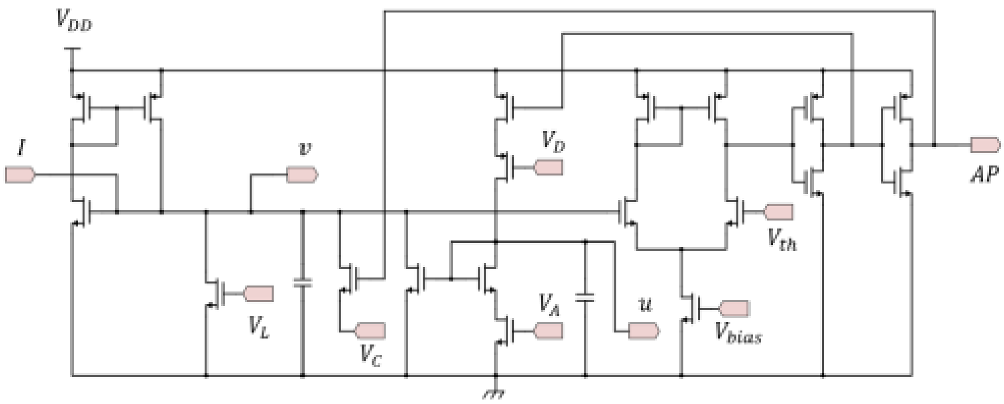

Figure 20: Analogue circuital implementation of an Izhikevich neuron [13]

3.4 Adaptive Exponential Integrate and Fire

The last model to be presented is the *Adaptive Exponential Integrate and Fire*, or aEIF, firstly presented by R. Brette and W. Gerstner in 2005, and which also utilizes a second variable for accounting the recovery time of the neurons. Differently from Izhikevich, an exponential term is introduced in the equation, allowing for a more realistic and smoother spike initialization region, i.e. when the membrane potential is close to the threshold voltage. [10]

Again, more biological details imply higher implementation complexity, in this case, embedded in the addition of the exponential function.

The set of differential equation is the following:

$$C\frac{dV}{dt} = -g_L(V - E_L) + g_L\Delta_T e^{\frac{(V-V_{th})}{\Delta_T}} - U + I_{ext} \tag{4a}$$

$$\tau_u \frac{dU}{dt} = a(V - E_L) - U \tag{4b}$$

$$\text{if } V \geq V_{th} \text{ then } V = E_L \text{ and } U = U + b$$

and being:

V	:	Membrane potential
U	:	Recovery potential
E_L	:	Leak reversal potential
V_{th}	:	Spike threshold
V_{peak}	:	Spike peak
C	:	Membrane capacitance
g_L	:	Leak conductance

Δ_T : Slope factor

a : Subthreshold adaptation

b : Spike-triggered adaptation

It is worth noticing that, in this case, there is an distinction between V_{th} and V_{peak}; the former, indeed, is the actual value to overcome for the membrane potential to trigger a spike, while the latter is the peak voltage of the spike reached by the neuron, ensuring that the membrane potential does not increase uncontrollably during a burst [11]. In the previous models this distinction is not present and does not affect the results.
Lastly, as for Izhikevich, this model can describe a various set of neural behaviours, depending on the values of its constant parameters, with the different dynamics shown in the next figure and briefly explained in section 4.1.1.
The main difference, is that even tough the Izhikevich model is sufficient to account for most types of firing patterns observed in the nervous system, the generated spikes appear with an unrealistic delay, while, with the introduction of an exponential term, the results match with direct measurements of biological neurons. [11]

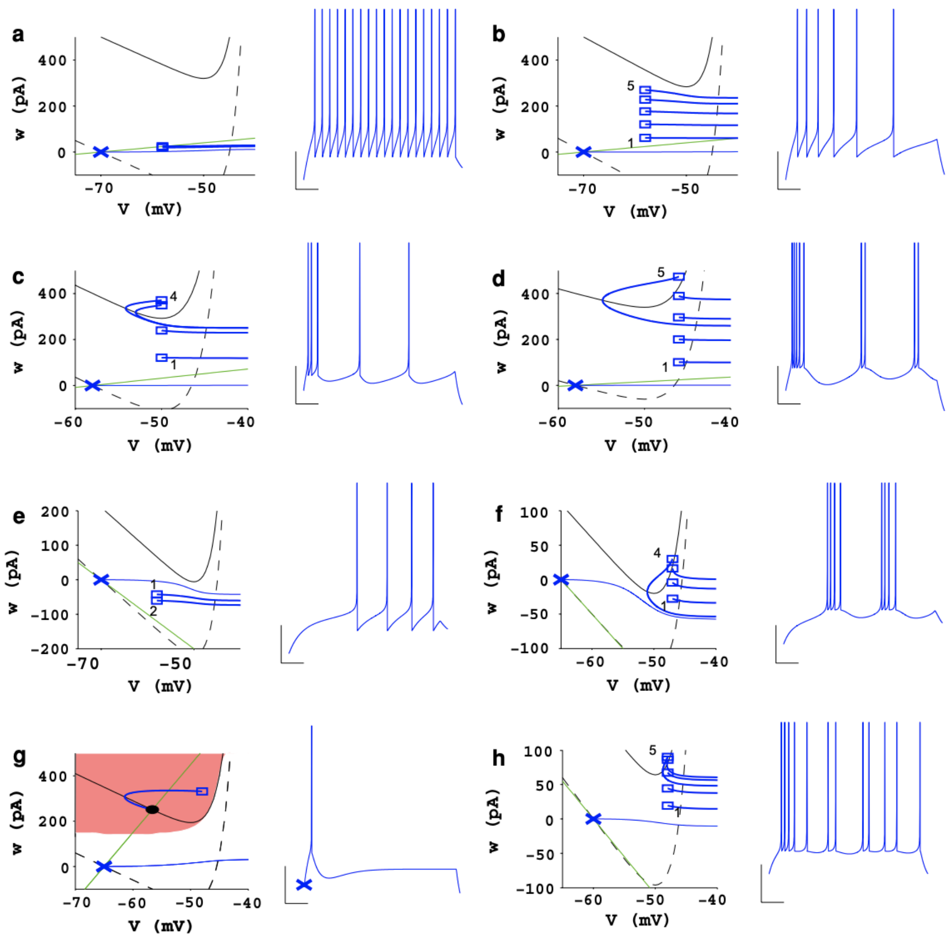

Figure 21: Dynamics of Adaptive Exponential Integrate and Fire neurons[11]

4 Implementation of the Adaptive Exponential Integrate and Fire model

In this chapter, it is presented the implementation of the aEIF model by showing four different neural behaviours (Regular Spiking, Spike Frequency Adaptation, Initial bursting and Tonic Bursting) in response to a DC current, firstly using MATLAB and then on *HEENS*, focusing on the problems that arose from the architecture's limitations and the methods used to solve them.
For sake of readability, only some extracts of the code are presented, but it can be found complete in the appendices.

4.1 General Information

4.1.1 Expected Results

The simulations portrayed in the following sections aim to depict four different type of neural behaviour, comparing the results with the ones illustrated in figure 21. These are: Regular Spiking (21.a), Spike Frequency Adaptation (21.b), Initial Bursting (21.c) and Tonic Bursting (21.d).
Regular Spiking (RS) is the simplest type of spiking pattern, generated by a regular discharge of action potentials, and it's the only firing pattern that a standard leaky or non-leaky integrate-and-fire model shows subject to constant current injection. It corresponds to the absence of spike-triggered adaptation and adaptation sensitivity to subthreshold voltage (a, b = 0).
Most neurons, however, have some level of spike-frequency adaptation (SFA), depicted in figure 21.b.[11]
Neurons with spike frequency adaptation are the most common in mammalian cortex. [9]
Initial Bursting (IB) behaviour denotes a group of spikes that were emitted at a frequency considerably greater than the steady-state frequency.

The main difference between IB and the previous spike patterns can be found in the after-spike resets: RS and SFA exhibits only sharp resets, meaning that the membrane potential increases monotonically after a spike, while IB shows sharp resets only at beginning, followed by broad resets, that can be recognized from a low curvature at all times after a firing.

Tonic Bursting (TB) is an alternation of sharp resets followed by a broad one.[11]

In figure 22 on the right side, it can be seen the difference between the two types of resets, above the sharp reset, below the broad one. On the left, a voltage-current graph of a neuron, reported for completeness but outside the scope of this work, and thus it won't be considered.

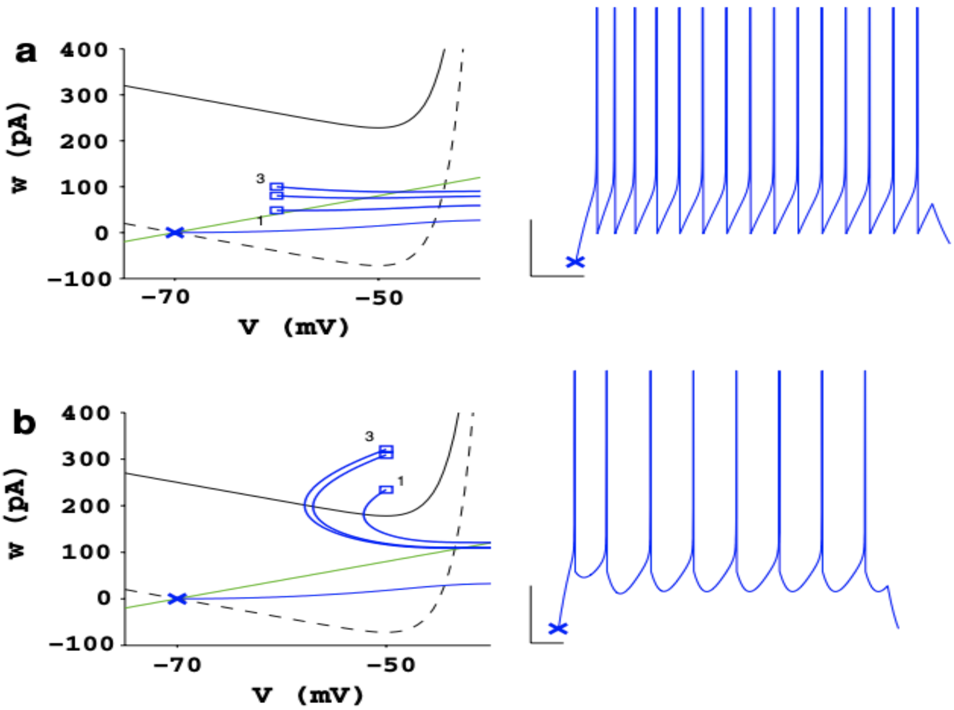

Figure 22: Sharp (a) and Broad (b) resets [11]

4.1.2 Neural Constants

In order to obtain the different output patterns, the constants for the model have been retrieved from the work of Brette and Gerstner[11] and have been set as in table 3.

	RS	SFA	IB	TB
$C(pF)$	200	200	130	200
$g_L(nS)$	10	12	18	10
$\Delta_T(ms)$	200	200	200	200
$V_{th}(mV)$	-50	-50	-50	-50
$V_{rst}(mv)$	-58	-58	-50	-46
$E_L(mv)$	-70	-70	-58	-58
$\tau_u(ms)$	30	300	150	120
$a(nS)$	2	2	4	2
$b(pA)$	0	60	120	100
$I_{in}(pA)$	500	500	400	210

Table 3: Model's constants

4.1.3 Ranges and Measurement Units

Due to the length of registers in *HEENS* and to the 2's complement representation of integer numbers, the range of possible values involved in the calculations is [-32768, 32767]; meantime, the membrane potentials involved in neural models are in the range of [-70, 30]mV.

These facts, combined with the needing of working with a reasonable resolution, led to the selection of $10uV$ as the unit for voltages, and consequently, all the voltages in the models are limited by the range [-327.68, 327.67]mV. This scaling allows for working with 2 decimal digits resolution.

The same happens for the currents, that are in the range [-327.68, 327.67]pA, with a single unit of $0.01pA$.

The time constants used in the models work with a resolution of $1ms$ and are not scaled.

4.1.4 Differential Equations and Time Resolution

The differential equations of the model are solved using *Euler's method*, for which, each variables next step in the simulation is evaluated as:

$$x(t+1) = x(t) + x'(t) * dt$$

Since *HEENS* operates with a $1ms$ resolution, for the calculations it has been selected $dt = 1ms$.

4.1.5 Multiplications and Divisions

Given N as the number of bits in a register, the result of a digital multiplication of the form $N * N$ bits can always be represented with at most $2N$ bits.
For this reason, multiplications in *HEENS* takes R0 and another register as input operands and the couple R0-R1 for storing the result.
In addition, divisions are made by multiplying for the reciprocal of the divisor, considering that $x : y = x * (y)^{-1}$.
The reciprocal value has to be calculated when writing the algorithm, and it must be treated as a constant number, since during the execution is not possible to evaluate it.

4.1.6 Code references

In table 4, a reference for some of the variables names used in the code.

Variable Name	**Description**
N	Number of neurons in the network
$exec\_cycles$	Length of the simulation
S	NxN matrix of synapses
v	Membrane potentital over time
u	Recovery potential over time
$firings$	Post-synaptic spikes over time
Iin	External DC current
fv	Function that combines the linear and the exponential term of the model's equation. Used for differentiating the sequence of operations done in *HEENS*
fv_ap	The approximated function of fv
fv_root	Value of voltage for which fv_ap equals 0
v_n	Current step membrane potential
u_n	Current step recovery potential
$Vmin,\ Vmax$	Limits of the registers

Table 4: Variables Description

4.2 Methods Involved

4.2.1 Method for handling fixed point numbers

Constant numbers belonging to the range of values (0,1) are transformed into integers by multiplying them for, usually, the value 2^N, with N being the registers length, and truncating the eventual remaining part of the mantissa.

In this way, when multiplied, the result itself is enlarged by a factor 2^{16}, but since it's stored within the couple R0-R1, by taking only the value stored in the MSBs (R0), the result is divided again by 2^{16}, restoring the correct result.

The drawback is that, when the mantissa is truncated, the numbers are approximated and the precision is decreased.

In table 5, an example of the division $900 : 200 = 4.5$, where in the last row, it can be seen the approximation of the digital result which is 5 instead of 4.5.

Also, is important to notice that the shifted reciprocal of the divisor, equal to 328 in this case, has to be evaluated before the execution, and it's the actual value given to *HEENS* for performing the divisions.

An actual example of this issue is presented in section 4.4.1, where, in the part of the file dedicated to the variables definition, the values of C and τ_u are already $\frac{2^{16}}{C}$ and $\frac{2^{16}}{\tau_u}$.

	Decimal Number	Digital Number	Digital Approximation
$Dividend$	900	00000011 10000100	900
$Divisor$	200	00000000 11001000	200
$\frac{1}{divisor}$	0.005	.00000001 01001000	0.0050048828125
$\frac{2^{16}}{divisor}$	327.68	00000001 01001000.	328
$Dividend * \frac{2^{16}}{divisor}$	294912	00000000 00000100 (R0) .10000000 00000000 (R1)	294912
$\frac{Dividend}{divisor}$	4.5	00000000 00000100 (R0)	5

Table 5: Division in *HEENS*

4.2.2 Method for controlling flow of execution

The conditional statements in *HEENS* are done by using subtractions and shiftings, by considering that $a > b \iff (a - b) > 0$.

By shifting by one to the left the result of a subtraction, the carry flag of the arithmetical unit is loaded with the sign's bit of the number, and it's possible to freeze the register file based on that bit's value. In this way, only some neurons will modify their status based on the operation enclosed in *freezing* instructions, differentiating effectively the execution flows.

4.2.3 Method for the approximation the exponential function

One of the main problems in the development of the model is the approximation of the exponential term in the equations. The linear and the exponential terms in the first equation at (4), have been joined in a new function referred to as *fv(V)*, and rewriting the equation, it becomes:

$$fv(V) = -g_L(V - E_L) + g_L \Delta_T e^{\frac{V - V_{th}}{\Delta_T}}$$

$$C\frac{dV}{dt} = fv(V) + I - U$$

The approximation of the exponential is therefore extended to the whole *fv* function, justified by the fact that, combined with hardware limitations, it has been found easier to implement and to get correct results from it. The technique used for approximating this approximation, referred to as *fv_ap* is to divide it in three operating regions, each with a different function:

$$fv\_ap = \begin{cases} (1) \quad -g_L(V - E_L) & V < V_{th} \\ (2) \quad \min(0, V^2 fv\_a + V fv\_b + fv\_c) & V_{th} \leq V < fv\_root \\ (3) \quad \max(0, 4(V^2 fv\_a + V fv\_b + fv\_c)) & fv\_root \leq V \end{cases}$$

(1) In the first case, when the membrane potential is below from the threshold voltage, the exponential term can be considered ≈ 0, and thus is neglected.

(2) The second range is the most critical one, because the exponential term is influencing the result but not enough to quickly trigger a spike. The solution found is a quadratic approximation of fv in the range $[V_{th}, fv\_root]$, with V_{th} being the spiking threshold while fv_root is the value of potential such that $fv(fv\_root) = 0$. Since this range is below the value of fv_root, all the positive results are discarded. The major drawback involving this method is the fact that for each set of model's constants, the quadratic function has to be found manually, first by calculating fv_root by solving the equation $fv(V) = 0$, than by finding the constants for a quadratic approximation of the function in that range.

Therefore, for every neural behaviour simulated the constants fv_root, fv_a, fv_b and fv_c are different.

These values have been extracted with a python script, found in appendix, but unfortunately, the results obtained using this method alone have been found not good enough, so a further manual tuning of the constants has been necessary, as reported in table 6.

	fv_root	**fv_a**	**fv_a (final)**	**fv_b**	**fv_b (final)**	**fv_c**	**fv_c (final)**
RS	-4494	32	32	1245	1241	11842	11950
SFA	-4494	39	39	1494	1491	14210	14175
IB	-4650	57	57	2222	2210	21404	21400
TB	-4650	21	21	802	810	7729	7796

Table 6: Constants for fv calculation

(3) The quadratic function used in previous paragraph is multiplied by a scaling factor for emulating the rapid increase of the exponential for voltage values above fv_root. As done for the previous equation, the negative results are discarded..

The error here is much bigger then in the other sections, but its influence in the result is minimal, as long as the approximation can trigger a spike within few execution cycles, as the original function achieves.

The multiplication factor has been chosen equal to 4 as a trade-off between implementation and approximation of the results.

The dynamics for the approximation can be seen in figure 23, in which the black line is the correct function, the purple lines represent the chosen approximation without precision losses, and finally the scattered red dots are the approximated version with also the roundings operated by the hardware.

To be noticed that the value of the purple and the red approximations need to be ≤ 0 for voltage values below the root, and thus in the figure they are not interrupted but forced to 0.

It can be seen that the error is less than 1mV before the function's root (every square is 0.5mV), and it becomes larger for voltage values far from that point.

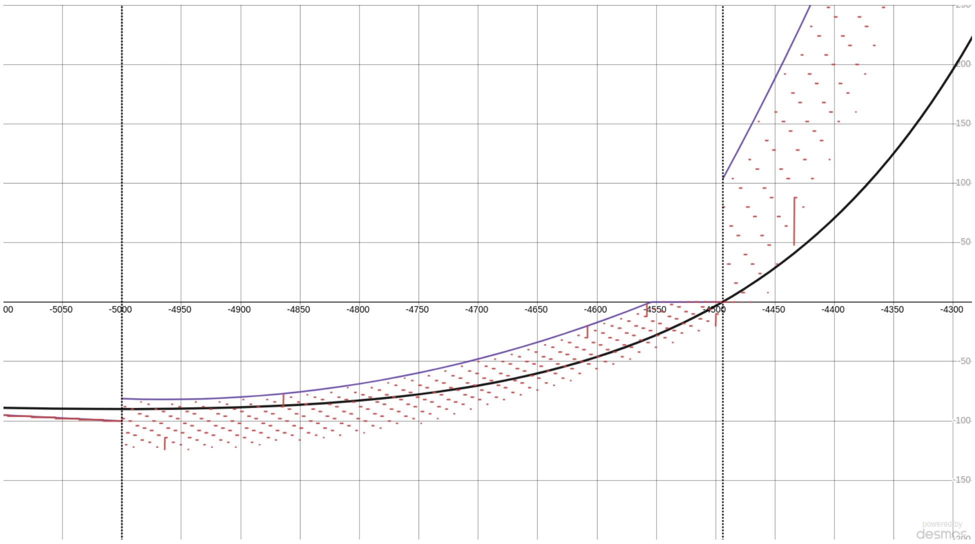

Figure 23: Dynamics of the approximation (in black the exact function, in purple the approximation, in red the implemented approximation with the precision losses due to hardware constraints)

Although also other techniques have been tried, such as a truncated Taylor series expansion or a simpler linear approximation, this solution has been the one whose results behaved as close as possible w.r.t figure 21.

4.3 MATLAB

MATLAB has been used for developing two different programs simulating an aEIF neuron, one for the actual model described by the equations at (4), and one emulating the features of *HEENS* architecture, for example, by limiting the values to the range $[-2^{15},\ 2^{15}-1]$ or representing decimal numbers in fixed point notation.

The network is composed of 4 different unconnected neurons, each with a different set of constants for emulating a different behaviour.

The main program contains the definition of the constants for the four different neural behaviours, than calls the two model functions, and plots the results on the same image.

4.3.1 aEIF High Accuracy Simulation

The implementation of the model from the equations given at (4) is pretty straightforward, and the structure of the simulation is divided in three main parts:

1. Spiking evaluation: all the neurons are checked, and, if their membrane potential is equal or greater than the peak, a spike is registered and the current generated by the neuron is stored inside variable I.

2. Model simulation: the step for the differential equations is calculated by computing the delta for both membrane and recovery potentials.

3. Variables update: the new value for the variables is updated and stored in v and u.

```
%simulating #cycles ms
    for cycle = 1:exec_cycles
        t = cycle*dt; % time in ms

        v_n = v(:, cycle); % current membrane pot
        u_n = u(:, cycle); % current recovery pot

```

```
         I = Iin(:, cycle)/C; % input current divided by C for later operations


        %firings and synaptic currents evaluation
        for i = 1:N  % for every neuron in the net
            if v_n(i) >= Vpeak      %if v > peak then fire
                firings = [firings; t, i-1];    %store the firing

                v_n(i) = Vrst;      %restore membrane potential
                u_n(i) = u_n(i) + b; %increment recovery potential

                for j = 1:N % for every synapse of the neuron
                    I(j) = I(j) + S(i, j);  %store outgoing current
                end
            end
        end

        %model simulation
        for i = 1:N
                %evaluate fv
                fv(i, cycle) = (-gl*(v_n(i) - El) + gl*delta_t.*exp((v_n(i)-Vt)/delta_t))/C;

                dv(i) = fv(i, cycle) - u_n(i)/C + I(i); %calculate dv
                dv(i) = dv(i)*dt;

                du(i) = a*(v_n(i) - El) - u_n(i);   %calculate du
                du(i) = du(i) / tau_u * dt;

        end

        %updating the neurons
        v(:, cycle+1) = v_n(:) + dv(:); %update membrane potential
        v(:, cycle +1) = min( max(v(:, cycle+1), -Vmin) , Vmax);

        u(:, cycle+1) = u_n(:) + du(:); %update recovery potential

    end
```

4.3.2 HEENS Emulation of the aEIF Model

This program aims to replicate the flow of operations done by *HEENS*, in particular, all the results are rounded the limited and the function fv_ap is calculated as discussed in 4.2.3 and in 4.4.2. Also here, the simulation loop is divided in three main phases, that can be seen in the code below, with the main differences found in the calculations for the variables step.

```
    %simulating #cycles ms
    for cycle = 1:exec_cycles
        t = cycle*dt; % ms with dt resolution
        v_n = v(:, cycle);
        u_n = u(:, cycle);
        I = Iin(:, cycle); %getting external current

        %firings and synaptic currents evaluation
        for i = 1:N  % for every neuron in the net
            if v_n(i) >= Vpeak      %if v > peak then fire
                firings = [firings; t, i-1];    %store the firing

                v_n(i) = Vrst;      %restore membrane potential
                u_n(i) = u_n(i) + b; %increment recovery potential
                u_n(i) = clip(Vmin, Vmax, u_n(i));

                for j = 1:N % for every synapse of the neuron
                    I(j) = I(j) + S(i, j);  %store outgoing current
                end
            end
        end
        %model execution
        for i = 1:N
            %evaluation of linear term always perfomed
                fv_ap(i) = El-v_n(i);
                fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                fv_ap(i) = fv_ap(i)*gl;     % +gl(El-v) == -gl(v-El)
                fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                fv_ap(i) = fv_ap(i)*C_div; gl(El-v)*C*2^16
                fv_ap(i) = floor(fv_ap(i)/2^16);    %fv_ap = gl(El-v)
                fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                %if  Vt < v_n
                if Vt - v_n(i) < 0
                    %evaluate quadratic term of approximation
                    fv_ap(i) = floor(v_n(i)^2/2^16);    % v^2/2^16
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
                    fv_ap(i) = fv_ap(i)*fv_a;       %fv_ap = fv_a*v^2
```

```
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                    %evaluate first order term of approximation
                    tmp = floor(v_n(i)/2)*fv_b;     %v/2 * fv_b *2^8
                    tmp = floor(tmp/2^8);   v/2*fv_b
                    tmp =clip(Vmin, Vmax, tmp);

                    fv_ap(i) = fv_ap(i) + tmp; %fv_a*v^2 + fv_b*v/2
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
                    fv_ap(i) = fv_ap(i) + tmp;  % fv_ap = fv_a*v^2+fv_b*v
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                    fv_ap(i) = fv_ap(i) + fv_c; %fv_ap = fv_a*v^2 + fv_b*v + fv_c

                    %perform min function and get result
                    tmp = 0;
                    if fv_ap(i) >= 0    %if fv_ap is positive, save in tmp but reset for later
                      tmp = fv_ap(i);   %only if fv_ap was >= 0
                      clip(Vmin, Vmax, tmp);
                      fv_ap(i) = 0;    %if fv_ap > 0, here put fv_ap = 0
                    end
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

                end
                %if v_n > root
                if root - v_n(i) < 0
                    fv_ap(i) = (4*tmp); %max func not needed because tmp >= 0
                    fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
                end

                dv(i) = I(i) - floor((u_n(i)* C_div)/2^16 ) ;
                dv(i) = dv(i)*dt  +  dt*fv_ap(i);   %get total dv

                du(i) = a*(v_n(i) - El) - u_n(i);
                du(i) = floor((du(i) * floor(2^16/tau_u))/2^16 ) * dt; %get du
        end
        %updating the neurons
        v(:, cycle+1) = v_n(:) + dv(:); %update membrane potential
        v(:, cycle +1) = clip(Vmin, Vmax, v(:, cycle+1) );

        u(:, cycle+1) = u_n(:) + du(:); %update recovery potential
        u(:, cycle +1) = clip(Vmin, Vmax, u(:, cycle+1) );
    end
```

4.3.3 MATLAB results

The MATLAB file have been tested using an input DC current, as done for the reference results in figure 21.

Below, figure 24 illustrates the firings pattern achieved by the exact model in black, and by it's approximation in red, while in figure 25 are represented the membrane voltages over time.

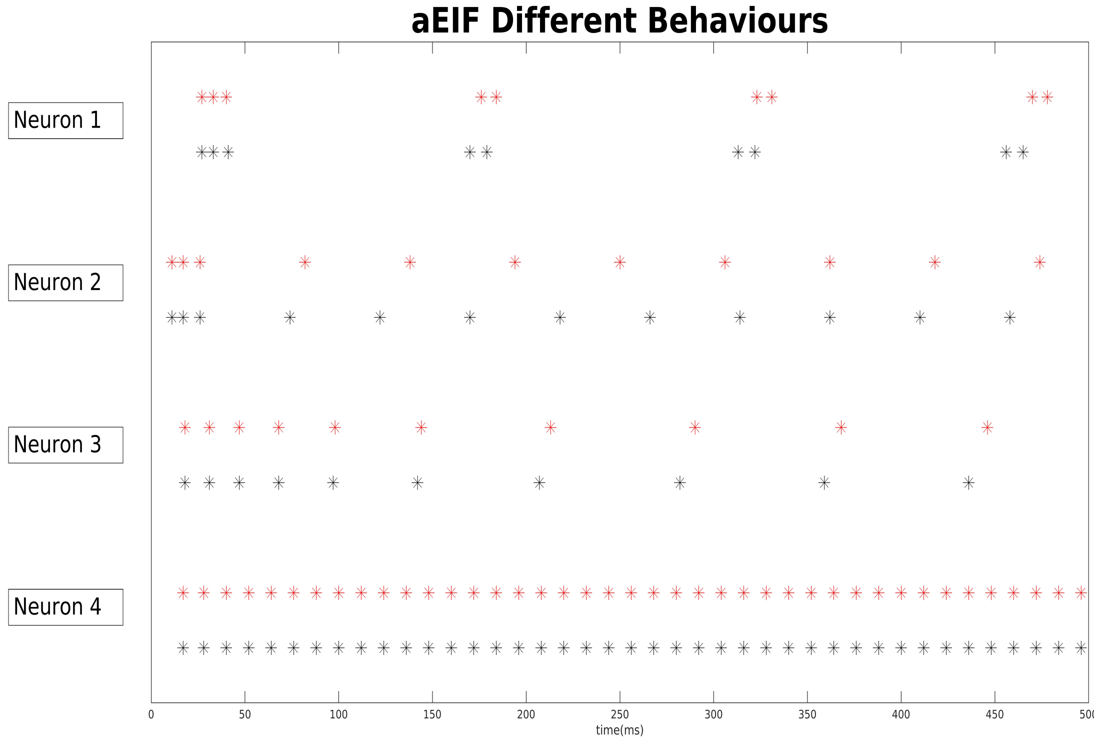

Figure 24: MATLAB simulation (in black the exact model, in red the approximated one)

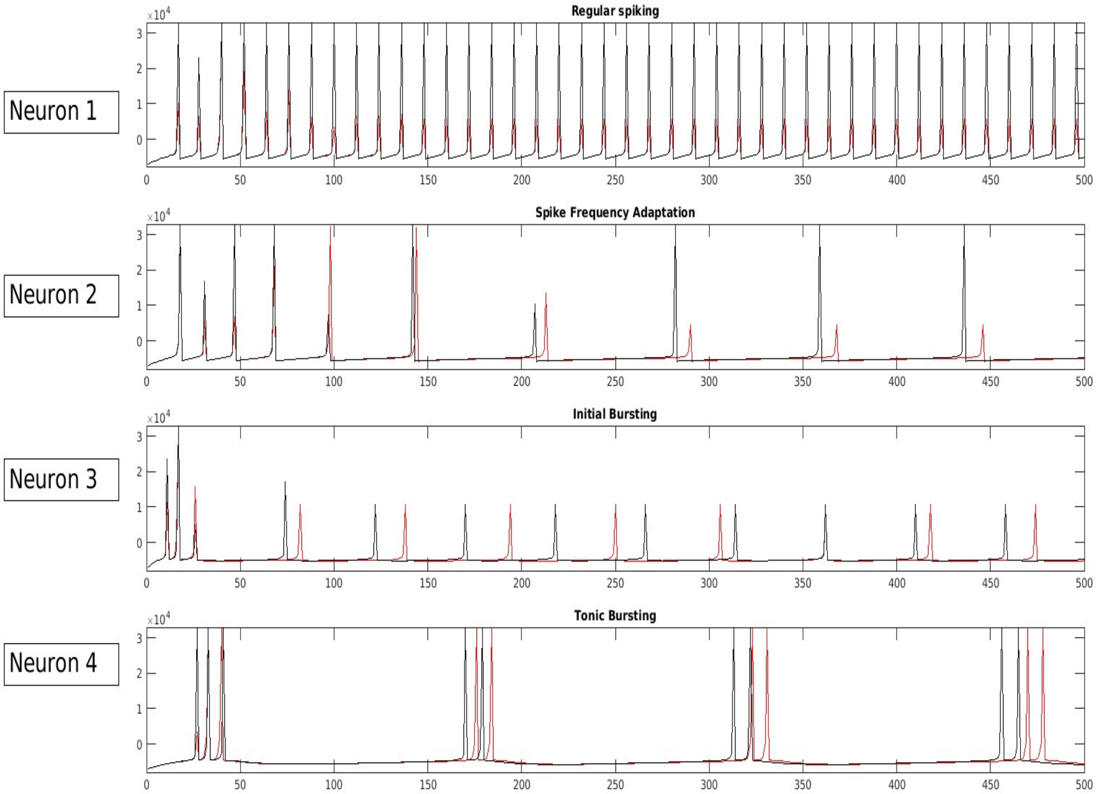

Figure 25: Membrane potentials over time (in black the exact model, in red the approximated one)

As can be seen, the black and red points follow the same behaviour, but the approximation of the mathematical expressions makes the frequencies of the neurons not exactly identical. For the same reason, for certain spikes the value of the black line get much higher values w.r.t the red one, but what's important is that the red value is above the threshold.

Lastly, in table 7, the errors committed by the drift of the approximations, evaluated as $\frac{\#red\ spikes}{\#black\ spikes}$ for a 20 seconds simulation.

	Red Spikes	**Black Spikes**	$\frac{\#red\ spikes}{\#black\ spikes}$
RS	1666	1666	1
SFA	260	264	0.985
IB	359	419	0.857
TB	273	281	0.972

Table 7: Spikes count in a 20s simulation

4.4 HEENS

Regarding *HEENS* implementation, the two necessary files have been developed, and the results are shown both with a QuestaSim simulation, and with a screenshot of *HEENS* user interface.

4.4.1 Netlist file

The network is composed of four unconnected neurons, but providing to each of them a different set of constants, summarized in table 8.

It's important to notice that the constants C, Δ_T and τ_u are not set to their value, but to the their multiplied reciprocal, in order to being used correctly in the calculations.

	RS	SFA	IB	TB
C_div	327	327	504	327
g_L	10	12	18	10
Δ_T	200	200	200	200
Δ_T_div	327	327	327	327
V_{th}	-50	-50	-50	-50
V_{rst}	-58	-58	-50	-46
E_L	-70	-70	-58	-58
τ_u	2184	218	436	546
a	2	2	4	2
b	0	60	120	100
$Const\_curr$	250	250	307	105
fv_root	-4494	-4494	-4650	-4650
fv_a	32	39	57	21
fv_b	1241	1491	2210	810
fv_c	11950	14175	21400	7796

Table 8: All constants used

Excepts for Δ_T and V_{th} that are global variables and thus are stored inside the IMEM, the others constants are proper of each neural behaviour, making necessary to store them in the SNRAM in the PE.

In this way, the maximum number of neuron in the network is reduced, because each neuron needs 7 different addresses in the SNRAM for storing the model's constants, plus the one for the potentials, fact that has to be accounted when simulating SNNs. Moreover, it can be noticed that the value of the input constant current is treated as a variable, and it's value is bigger then the size of registers, for this reason, in the netlist file, the current is already divided by C, in order to fit the in 16 bits registers.

```
@Config
Zedboard_4x8
@ParamSyn
# synaptic weights = 0
0, 0
@Netlist
0, 0
1, 1
2, 2
3, 3
@Params
#membrane potential and recovery variable at t = 0
.0x1E3/16/NEUR/$NVL/-7000, -1400

# all the model's variables
.0x3E0/16/EL_GL/$NVL/0 , 0
0, -7000, 10
1, -7000, 12
2, -5800, 18
3, -5800, 10
UNMAPPED, 0, 0

.0x3E1/16/V_RST_CONST_CURR/$NVL/0, 0
0, -5800, 250
1, -5800, 250
2, -5000, 307
3, -4600, 105
UNMAPPED, 0, 0

.0X3E2/16/C_DIV_TAU_U/$NVL/0, 0
0, 327, 2184
1, 327, 218
2, 504, 436
3, 327, 546
UNMAPPED, 0, 0

.0x3E3/16/NEU_A_B/$NVL/0, 0
0, 2, 0
```

```
1, 2, 6000
2, 4, 12000
3, 2, 10000
UNMAPPED, 0, 0

.0x3E4/16/FV_A_B/$NVL/0, 0
0, 32, 1241
1, 39, 1491
2, 57, 2210
3, 21, 810
UNMAPPED, 0, 0

.0x3E5/16/FV_C_ROOT/$NVL/0, 0
0, 11950, -4494
1, 14175, -4494
2, 21400, -4650
3, 7796, -4650
UNMAPPED, 0, 0

#seed for noise generation
.0x1FD/32/SEED/2/-6500, 800
5, 10
```

4.4.2 Neural Model

MAIN LOOP

The main algorithm is short and the flow is linear, and after some initial configuration, the state of the neuron in the first virtual layer is loaded inside the processor, then the overcoming of the threshold is checked, the synaptic currents and the constant DC input are summed and stored, and after the delta for the model are calculated.

Finally, the new value for the variables is updated,the neuron's state is stored back in the memory and the virtual layer is increased.

Since for this simulation only one layer is needed, the virtualization loop ends and the spike distribution phase begins.

```
EXEC_LOOP:              ; Execution loop

    LOOP    NVL         ; Virtualization loop
        SYNAPSE NLS_0
        GOSUB LOAD_NEURON   ; Loading current neuron

        GOSUB DETECT_SPIKE; ;check for spike

        RST R6              ;reset register for current
        READMPV LSA0_0      ;loads the address with curr layer synapses
        LOADBP
        LOOPV NLS_0         ; synaptic loop. Reads number of current-layer synapses
            GOSUB SYNAPSE_CALC  ;calculate synaptic currents
        ENDL

        GOSUB ADD_CONST_CURR    ;add DC const current

        GOSUB EVAL_DELTA_V;     ;evaluate dv/dt
        GOSUB EVAL_DELTA_U;     ;evaluate du/dt
        GOSUB CALC_STEP;        ; update v and u

        MOVA R2     ; R0 <= membrane potential
        STOREB      ; value of R0 in fifo for visualization of results

        GOSUB STORE_NEURON;     ;store back neuron state
```

```

        RST ACC
        RST R3
        RST R2
        INCV            ;increment virtual layer
    ENDL            ;end virtualization loop

SPKDIS        ; Distribute spikes
GOTO EXEC_LOOP ;
```

LOAD_NEURON

As an example of working with the SNRAM memory, it is presented the loading of a neuron inside the processor for each virtual layer:

```
LOAD_NEURON:        ;
        READMPV NEUR_0  ; Address of real neuron + virt (valid also for non-virtual)
        LOADBP                     ; SNRAM pointer to currently processed neuron
        LOADSN                    ; Load Neural parameters from SNRAM to R1<=u & ACC<=Vmem
        MOVR R2                    ; R2 <= v0
        MOVA R1                    ; ACC<=u0
        MOVR R3                    ; r3<=u
RET
```

The instructions *READMPV*, *LOADBP* and *LOADSN* are to be executed in this order, to provide the correct reading of the data from the memory inside R0-R1.

SYNAPSE_CALC

For this network synapses are not present, but the function is shown for illustrating an example of an *if statement*.

```
SYNAPSE_CALC:
    LOADSP                    ; Load Synaptic parameters and spike to R1 & ACC
    SHRN 1                    ; Move spike to flag
    FREEZENC
            MOVA R1              ; Synaptic parameter to ACC
            ADD R6
            MOVR R6
    UNFREEZE
    RST ACC
    STORESP                    ; Stores synaptic parameter and increases BP for
                    ; next synapse processing
    INCS
RET
```

The instruction *SHRN* loads the Carry flag with the LSB of R0, in which is stored the information of received spike, and the register file is freezed with the operation *FREEZENC* if that flag equals 0, meaning that a spike has not arrived.
If R0 contains the information for a received spike, the operations inside the freeze are executed as normally, summing the synaptic current to R6.

EVAL_DELTA_V

The evaluation of dv is done by firstly approximating the exponential approximation, and later by summing the currents and the recovery potential.
The most critical part is to calculate fv, for that a further explanation is needed.
As said in 4.2.3, the approximation is divided in three operating regions:

1. $V \ < V_{th}$: The calculations involving only the linear term are always performed, and the result is overwritten when not needed.
   It's interesting to notice that the division by C is done with a multiplication, with the constant $\frac{2^{16}}{C}$ defined in the netlist and stored in the SNRAM.
   The division by 2^{16} for getting the correct result is done by considering only register R0 and discarding R1.

```
EVAL_FV:
    ;1) evaluate always v < vt : fv = -gl(v - El) -> gl(el - v), R0 KEEPS EL, R1 KEEPS GL
        READMPV EL_GL_0    ;ADDRESS CONSTANTs gl AND El
        LOADBP
        LOADSN             ;R0 <= EL, R1 <= GL
        SUB R2        ; R0 <= EL - V
        MULS R1        ; GL*(EL - V) ; RESULT IS IN R1 BECAUSE LSB

        MOVSR R1        ;R1S TEMPORARY STORES THE VALUE GL(EL-V)

        READMPV C_DIV_TAU_U_0 ; ADDRESSES C_DIV AND TAU_U
        LOADBP
        LOADSN     ;2^16/C IN R0, 2^16/TAU_U IN R1
        MOVRS R1   ; R1 <= GL(EL -V)

        MULS R1 ; RESULT IS IN R0 BECAUSE MSB
        MOVR R1 ; R1 <= GL(EL - V)/C
        MOVSR R1 ; SR1 <= GL(EL-V)/C FINAL RESULT
```

2. $V_{th} < V < fv\_root$: in this region, its evaluated the quadratic approximation of the function fv, but due to hardware limitations and the constant values, some precautions have been taken.

   Indeed, because of the high range of values assumed by the different fv_b constants, it has not been possible to multiply it for 2^{16}, but only for 2^8, in order to not get an overflow from the multiplication. Still, this effort was not enough for avoiding too much bigger values, and the solution found has been to firstly divide by 2 the membrane and sum it two times, reproducing effectively a 9 position shift, but with the utilization of a saturated sum, which prevents the overflow.

   Lastly, after that fv_c has been summed, if the result is positive, it is saved into R5 for later otherwise it's discarded. In this way the values of dv got from this calculations are always negative, while if positive they're used for triggering the spike in the next part.

   Notice also that the register $SR1$ was storing the linear term, but it's then overwritten with the quadratic approximated value.

```
EVAL_FV:
    ;2) evaluate v - Vt > 0 : fv = fv_a*v^2 + fv_b*v + fv_c
    LDALL R0, VT
    SUB R2      ;VT - V
    SHLN 1      ;
    FREEZENC
        ;evaluate fv_a*v^2
        MOVA R2
        MULS R2 ; V^2 BUT TAKE V^2/2^16 CONSIDERING ONLY R0
        MOVR R5 ; V^2/2^16 IN R5
        READMPV FV_A_B_0
        LOADBP
        LOADSN
        MULS R5  ; R0 AND R1 KEEPS fv_a*V^2, result in R1
        MOVA R1     ; ACC <= R1
        MOVR R4     ; QUADRIC TERM IN R4

        MOVA R2     ; R0 <= VMEMB
        SHRAN 1     ; R0 <= VMEMB/2
        MOVR R5     ; R5 <= VMEMB/2
        READMPV FV_A_B_0    ;ADDRESS CONSTS FV_A, FV_B
        LOADBP
        LOADSN      ;R0 <= FV_A, R1 <= FV_B
        MOVA R1            ;R0 <= FV_B
        MULS R5        ;R0 <= V/2*FV_B AND RESULT IN R0[7:0] AND R1[15:8]
        SHLN 7         ;
        SHLN 1  ;R0 NOW KEEPS RESULT IN R0[15:8] AND R0[7:0] = 0
        MOVR R5        ; R5[15:8] KEEPS PARTIAL RESULTS, R5[7:0] = 0
        MOVA R1        ; R0 NOW KEEPS OTHER HALF OF RESULT IN R0[15:8]
        SHRN 7         ;
        SHRN 1  ; R0[15:8] = 0, R0[7:0] KEEPS PARTIAL RESULT
        OR R5         ; (R5[15:8] OR R0[15:8]=0), (R5[7:0]=0 OR R0[7:0]) ---> (2^8*FV_B*V/2^8) /

        ;summing  fv_b*v/2 + fv_a*Vmemb + fv_b*v/2
        MOVR R1     ;SAVE RESULT ON R1
        ADD R4      ; R0 = FV_A*VMEMB^2 + (2^8*FV_B*V/2^8) / 2
        ADD R1      ; R0 = FV_A*VMEMB^2 + (2^8*FV_B*V/2^8)
        MOVR R4        ; R4 KEEPS SUM OF FIRST AND SECOND ORDER TERMS

        READMPV FV_C_ROOT_0 ;ADDRESS FV_C AND FV_ROOT
        LOADBP
        LOADSN  ;R0 <= FV_C, R1 <= FV_ROOT
        ADD R4         ; R0 <= FINAL RESULT OF APPROXIMATION

        MOVR R1 ; R1 <= FINAL RESULTO
```

```
        RST R5   ; R5 TO 0
        SHLN 1   ; CHECK WHETHER FINAL RESULT < 0
        FREEZEC ;FREEZE IF RESULT IS NEGATIVE, OTHERWISE SAVE IT FOR LATER
            MOVA R1     ;GET BACK RESULT FROM R1
            MOVR R5     ;R5 != 0 ONLY WHEN RESULT OF FV IS POSITIVE
            RST R1      ;RESET R1
        UNFREEZE

        MOVSR R1        ; NOW R1S KEEPS THE TERM OF FV
    UNFREEZE
```

3. $fv\_root < V$: for the last part, the result of the previous section is retrieved from R5, and it is different from 0 only if it was positive, to avoid approximation errors. This value is then summed four times, in order to enlarge it and to prevent overflows.

```
;3)evaluate ROOT - V >= 0 : 4*fv of case 2)
READMPV FV_C_ROOT_0 ; ADDRESS FV_C AND FV_ROOT
LOADBP
LOADSN      ; R0 <= FV_C, R1 <= FV_ROOT
MOVA R1     ; R0 <= FV_ROOT
SUB R2  ; R0 <= FV_ROOT - VMEMB
SHLN 1  ; LOAD CARRY FLAG
FREEZENC  ; FREEZE WHEN ROOT - V >= 0
    MOVA R5 ; QUADRATIC APPROX RESULT, ONLY RESULTS > 0

    ADD R5
    ADD R5
    ADD R5  ; ADD INSTEAD OF SHLAN BECAUSE SATURATES

    MOVR R1 ;R1 <= FINAL RESULT
    MOVSR R1    ; OVERWRITE PREVIOUS RESULTS
UNFREEZE
```

4.4.3 HEENS Results

The results of the *HEENS* simulation are presented both in QuestaSim hardware simulations, in fig. 26-27, and with the actual *HEENS* user interface, in figure 28.
As expected, the results are identical and they're congruent with the MATLAB simulation done for the architecture shown in previous paragraphs.

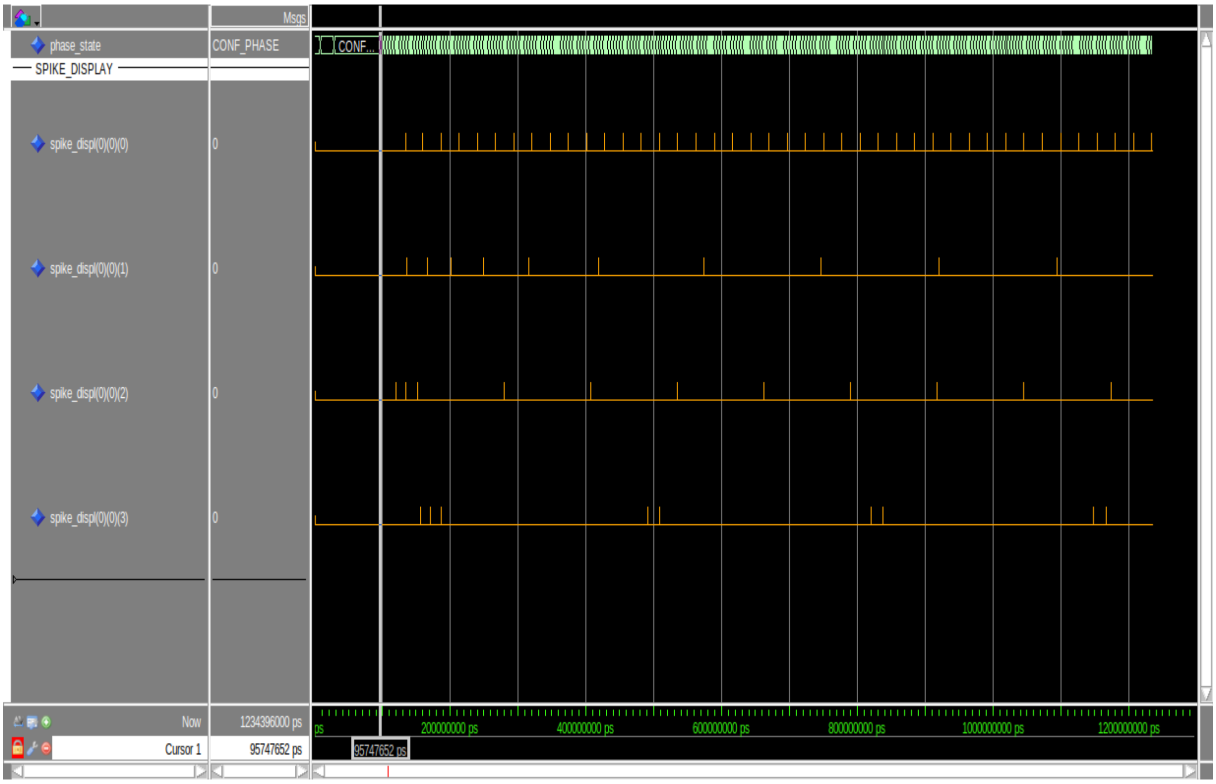

Figure 26: QuestaSim view of the output spikes

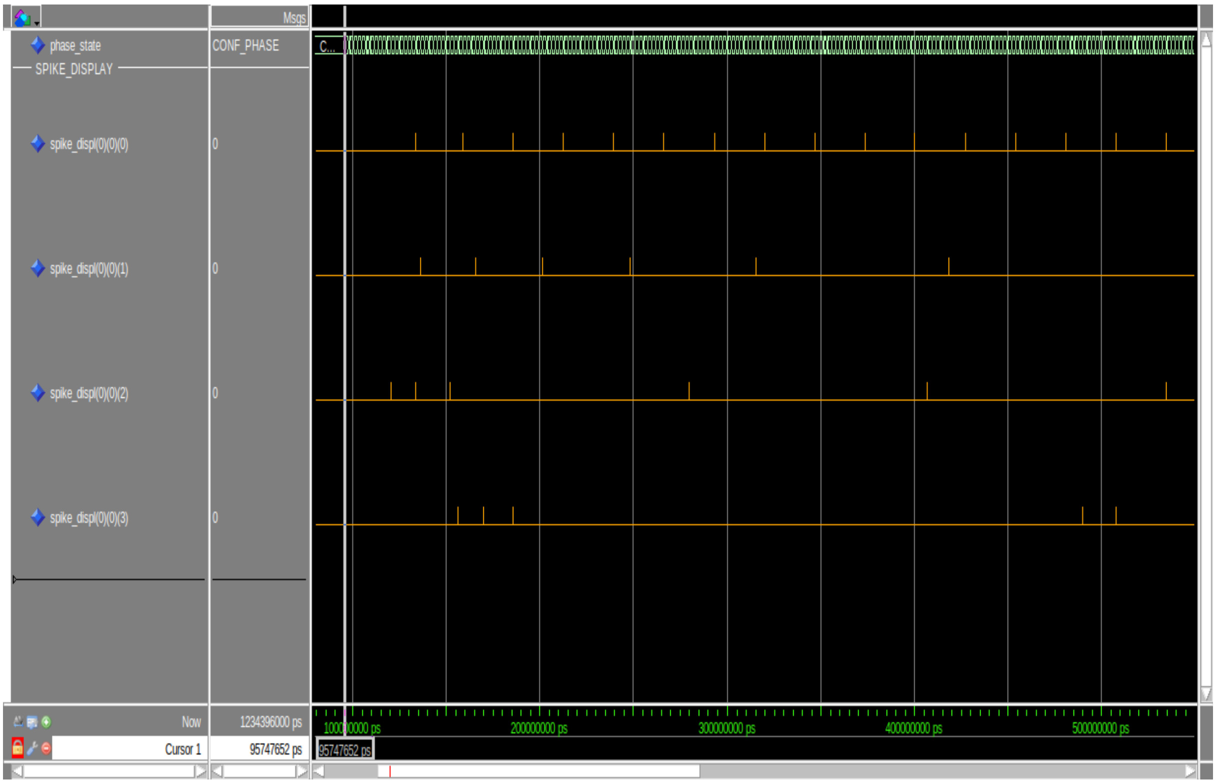

Figure 27: Closer view of Questasim output spikes

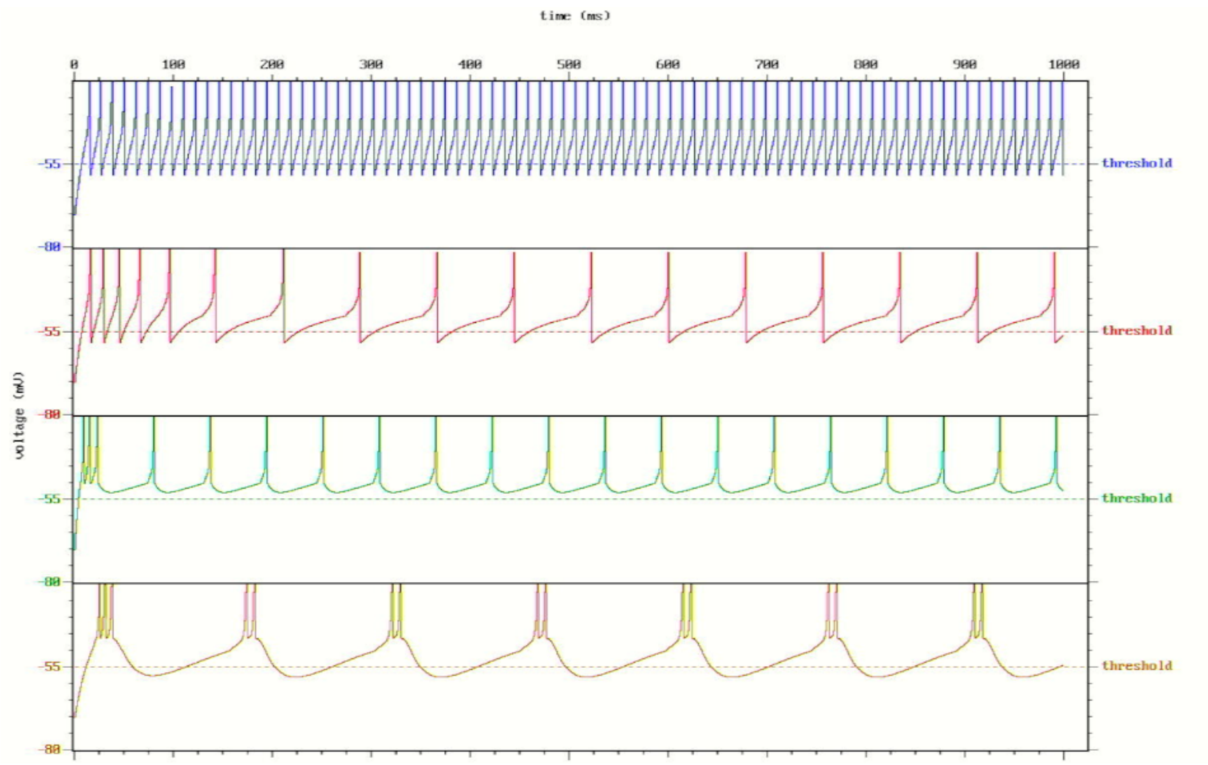

Figure 28: HEENS view of membrane potentials

5 Reservoir Network Simulation

The last part of this work is the simulation of a Spiking Neural Network, composed of 16 neurons, and implemented in a Reservoir topology.

The topology of the network and the data for the comparison of the results have been provided by professor S. Moriya of Tohoku University in Japan, whom we thank, and that is developing an analogue CMOS circuit able to implement Izhikevich equations.

We received only partial information regarding the network functionalities, mainly because they are still under research and because it's only a part of a bigger project. The final purpose of the network is to generate different firing patterns in response to inputs arriving to different neurons: in fact, this topology could be seen as a sub-network withing a bigger topology, and its inputs are the outputs from different nets, connected only to some neurons. In this way, depending on which neuron receive an input stimulation, the output layer should be able to classify it, but, again, the project is not completed and an output layer is yet not present.

Overall, the whole system aims to provide autonomous features for robotics applications. For these reasons, the network does not have an actual input nor an output, and the results are compared inspecting the whole dynamics.

The scope of this section is therefore to provide a digital *HEENS* simulation of the same network, in order to compare analogical and digital results, and proving their consistency.

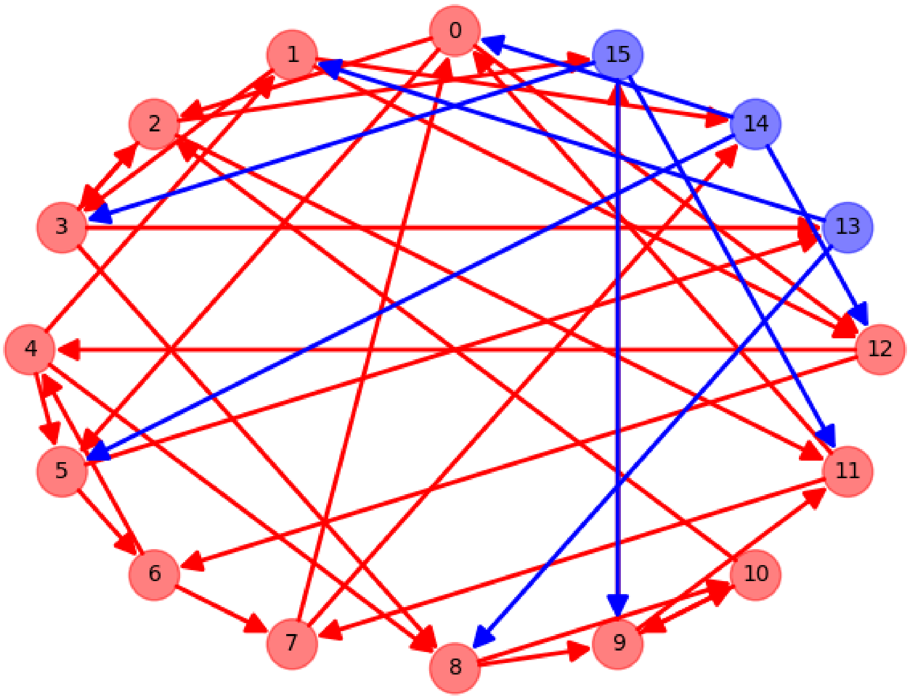

Figure 29: Reservoir Network Topology

5.1 Izhikevich Analogue Neuron

The mechanism behind this technology is to have CMOS transistors working in sub-threshold regions, arranged to behave following the equations described by the Izhikevich model.

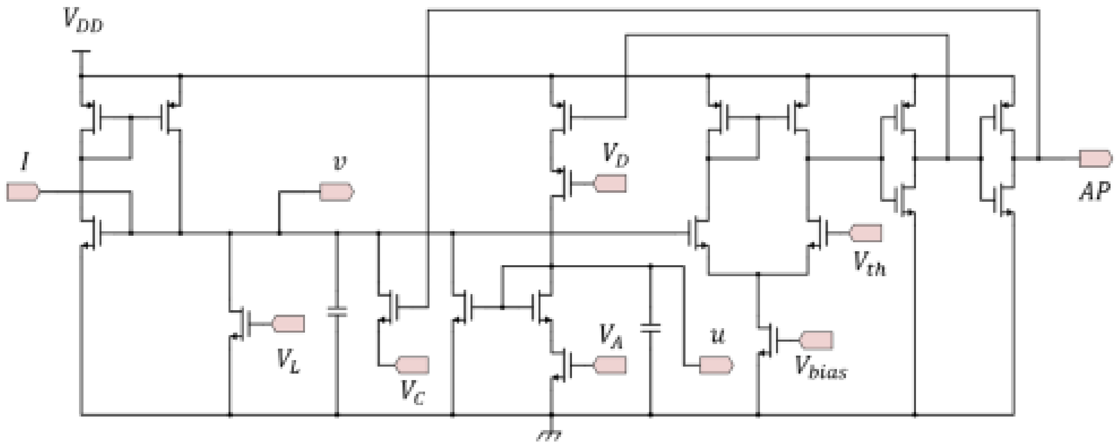

Figure 30: CMOS implementation of an Izhikevich neuron [13]

The circuit is divided in three parts, one subcircuit for membrane potential dynamics, one for the recovery potential, and the last is a comparator that triggers the spikes. The model's variables are represented by the voltage difference across the capacitors. [13]
Since analyzing such circuit is not part of this work, it won't be discussed further.

5.2 Network Topology and Neural Models

The network consists of 16 neurons, 13 of which (numbered 0 to 12) have outgoing positive synaptic weights and are referred as *excitatory*, and 3 (13 to 15) that have negative synaptic weights and are called *inhibitory*. Furthermore, inhibitory neurons have a different set of constants with respect to the excitatory ones, thus having a different behaviour. The inputs are constant currents for the neurons 0 to 5, whereas an actual output layer is missing, and the raster plot of the net is the final result.

The network connections are represented in figure 31 in the form of a Synaptic Matrix, for which the neurons in the rows are the pre-synaptic neurons, i.e. the ones firing, and the columns indicate the post-synaptic neurons, i.e. the ones receiving the spikes. Notice that there are no recurrent connections (the main diagonal) and all the weights are set to 1 or -1, meaning that they have same absolute value.

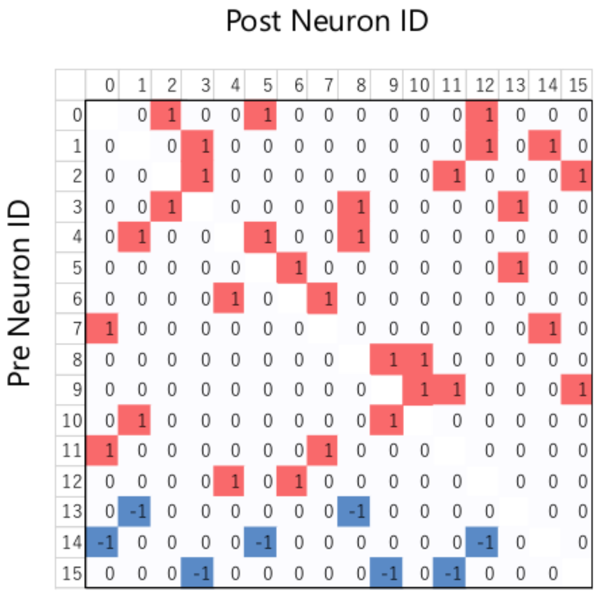

Figure 31: Synaptic Matrix for the Reservoir Network

The different set of constants for the two kind of neurons are shown in table 9.

	a	**b**	**c**	**d**
Excitatory neurons	0.015	0.15	-70	6
Inhibitory neurons	0.02	0.2	-70	2

Table 9: Excitarory and Inhibitory constants

Moreover, in order to get similar results to the one obtained with the analogue technology, both the DC input currents and the synaptic ones are modeled with an exponential decay, and thus, at each time step, instead of being reset, they are decreased with a factor $\tau_I = 20$.

$$I(t) = I(t-1) * e^{(-1/\tau_I)} + I_{in}(t) \tag{6}$$

5.3 HEENS Files

5.3.1 Netlist

The netlist file is simple and linear, with the only arrangements done for the synaptic weights, for the inhibitory constants and for the constant currents to input to neurons 0 to 5.

```
@Config
Zedboard_4x8

@ParamSyn
400, 0

@Netlist
#excitatory synapses definition
0        ,        2
...
12  ,   6

#inhibitory synapses
13  ,   1    , -400
...
15  ,   11   , -400

@Params
# Addr/Size/Name/Entries/default (empty for random) R0 / R1
.0x1E3/16/NEUR/$NVL/-7000, -1050

.0x3E0/16/IZH_A_B/$NVL/983 , 9830
13, 1310, 13107
14, 1310, 13107
15, 1310, 13107
UNMAPPED, 0, 0

.0x3E8/16/IZH_C_D/$NVL/-7000, 600
13, -7000, 200
14, -7000, 200
15, -7000, 200
UNMAPPED, 0, 0

.0x3F4/16/CONST_CURR/$NVL/0, 0
```

```
0, 400 , 0
1, 400 , 0
2, 400 , 0
3, 400 , 0
4, 400 , 0
5, 400 , 0
UNMAPPED, 0, 0

.0x1FD/32/SEED/2/-6500, 800
5, 10
```

5.3.2 Neural Model

The main loop in file for the Izhikevich neural model has the same structure of the one proposed for the aEIF, with the addition of the currents exponential decay and a slight difference in the membrane potential evaluation.

As the current in this case is decaying and should not be reset at every cycle, it important to store it in the SNRAM as done for the membrane and the recovery potential, and this is obtained with the routines *LOAD_CURR* and *STORE_CURR*. In addition, in order to have numerical stability, the step dv for the membrane is evaluated with a resolution of $\frac{dt}{2} = 0.5ms$ [9], hence is calculated two times as follows:

$$\overline{V} = V(t + \frac{dt}{2}) = V(t) + dV * \frac{dt}{2}$$
$$V(t + dt) = \overline{V} + d\overline{V} * \frac{dt}{2}$$

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

    ; Initial instructions
```

```
    GOSUB RANDOM_INIT        ; For noise initialization

EXEC_LOOP:           ; Execution loop
    LOOP NVL            ; Neuron loop for virtual operation
        GOSUB LOAD_NEURON   ;loading membrane and recovery potentials
        GOSUB LOAD_CURR     ;get current from last step
        GOSUB DETECT_SPIKE  ;check if v > Vth

        SYNAPSE NLS_0     ; configuring number of synapses
        READMPV LSA0_0    ; addressing the synapses in mem
        LOADBP              ;load pointer
        LOOPV NLS_0       ; synaptic loop. Reads number of current-layer synapses
           NOP          ;to prevent pipeline error
           GOSUB SYNAPSE_CALC   ;total current stored in SR1
        ENDL

        GOSUB ADD_CONST_CURR ;  add constant input
        GOSUB CURR_DECAY     ;  current exp decay


        SWAPS R1        ; take total current from SR1
        MOVA R1         ; move to acc
        SWAPS R1        ; move to SW1

        SHRAN 1         ; divide by 2 total current for later steps
        MOVR R5         ; R5 <= current/2

        LOOP 1          ; dt = 0.5
                GOSUB MEMBRANE_POTENTIAL ; Calculate membrane potential according izhikevic
                ;GOSUB ADD_NOISE               ; Noise not added
                ADD R5      ; add curr/2
            MOVR R2     ;store back membrane pot
        ENDL

         GOSUB RECOVERY_UPDATE  ;update recovery potential

         GOSUB STORE_NEURON     ;store neuron
         GOSUB STORE_CURR       ;store the current of this time step

         MOVA R2
         STOREB
         NOP      ;for preventing pipeline error, maybe not needed
         NOP

         RST ACC        ;reset r0
```

```
            MOVR R1              ;reset r1
            INCV       ;increment virtual layer

        ENDL
        NOP
        SPKDIS        ; Distribute spikes
    GOTO EXEC_LOOP ; Execution loop
```

```
LOAD_CURR:
    READMPV CONST_CURR_0    ;get address of const_curr
    LOADBP
    LOADSN              ; R0 <= const_curr, R1 <= Current from prev cycle
    MOVSR R1            ; SR1 <= curr for this cycle
RET

STORE_CURR:
    READMPV CONST_CURR_0    ;get address of const_curr
    LOADBP
    LOADSN          ; R= <= const_curr, R1 <= curr from prev cycle(to update)

    MOVRS R1        ; R1 <= SR1, SR1 store updated current
    STORESP     ; store back R0 and R1 to SNRAM
RET
```

Since the function for the membrane potential update has been retrieved from a previous work [5], it is not reported here but left complete in appendix.

Instead, below are illustrated the functions for adding the input current and the exponential decay, which are an example of the difference between retrieving data from the SNRAM (performed with three different operations) or from the IMEM (as done for the decay constant τ).

Again, multiplications with numbers < 1 are done by transforming them to integer, calculated the enlarged result, and then divide back for restoring the correct outcome.

```
CURR_DECAY:
    LDALL R0, TAU_I ;R0 <= tau_I from IMEM
    SWAPS R1    ;take total current
    MULS R1 ; R0-R1 <= I*e^(-1/20)*2^15
    ;dividing by 2^16 by discarding the part of the result stored in R1
    SHLN 1  ; shift R0 for
    MOVR R1 ; R1 <= total curr
    SWAPS R1 ; SR1 <= total curr
RET

ADD_CONST_CURR:
    READMPV CONST_CURR_0    ;read address for constant current in SNRAM
    LOADBP      ;load pointer
    LOADSN  ; R0 <= CONST_CURR , R1 <= TAU_I
    SWAPS R1  ;R1 <= TOTAL I
    ADD R1    ; R0 <= CONST_CURR + TOT_I
    MOVR R1
    SWAPS R1 ;  R1S <= TOTAL CURRENT
RET
```

5.4 Comparison of Results

In this last section are presented three raster plots, the reference one made with the CMOS neurons, a MATLAB simulation and *HEENS* results executed on QuestaSim.

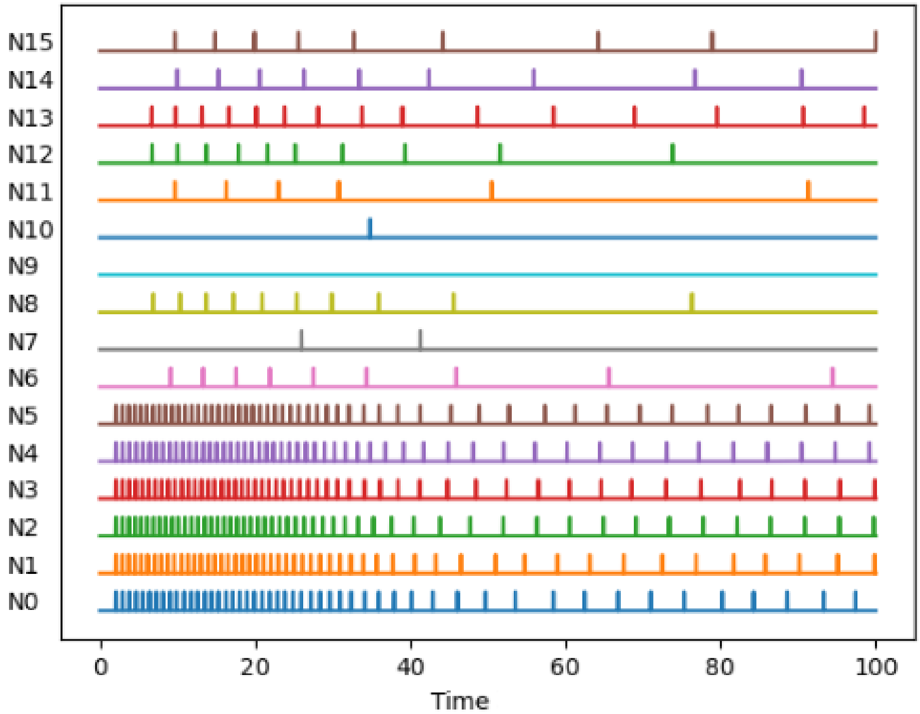

Figure 32: Analogue Results

Even tough the patterns are not identical, they somehow express the same behaviour, but with differences in the frequency of the spikes. This fact could be derived from the fact that a finer tuning of the model's parameters should be performed, possibly also differentiating each neuron's constants, but also from the limited resolution involved in the calculations.

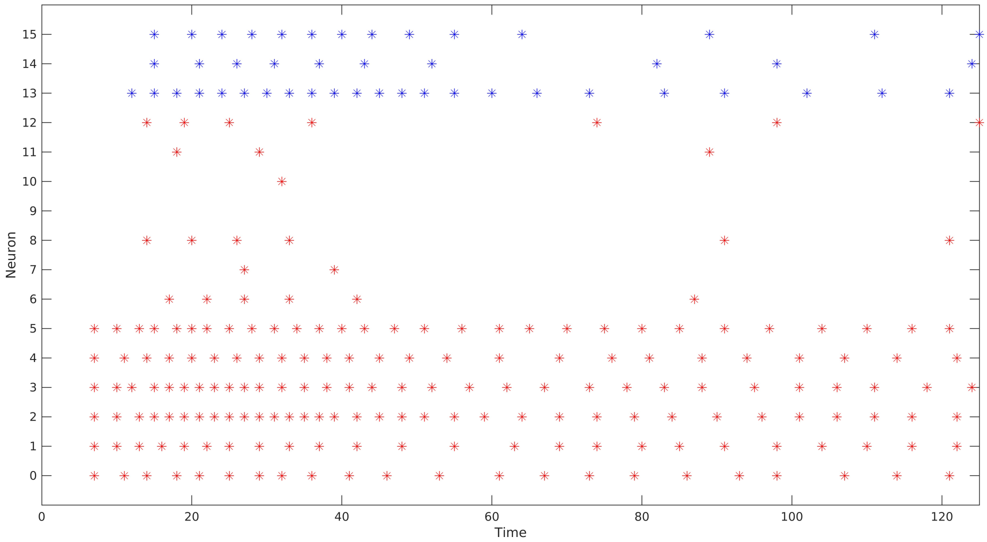

Figure 33: MATLAB Results

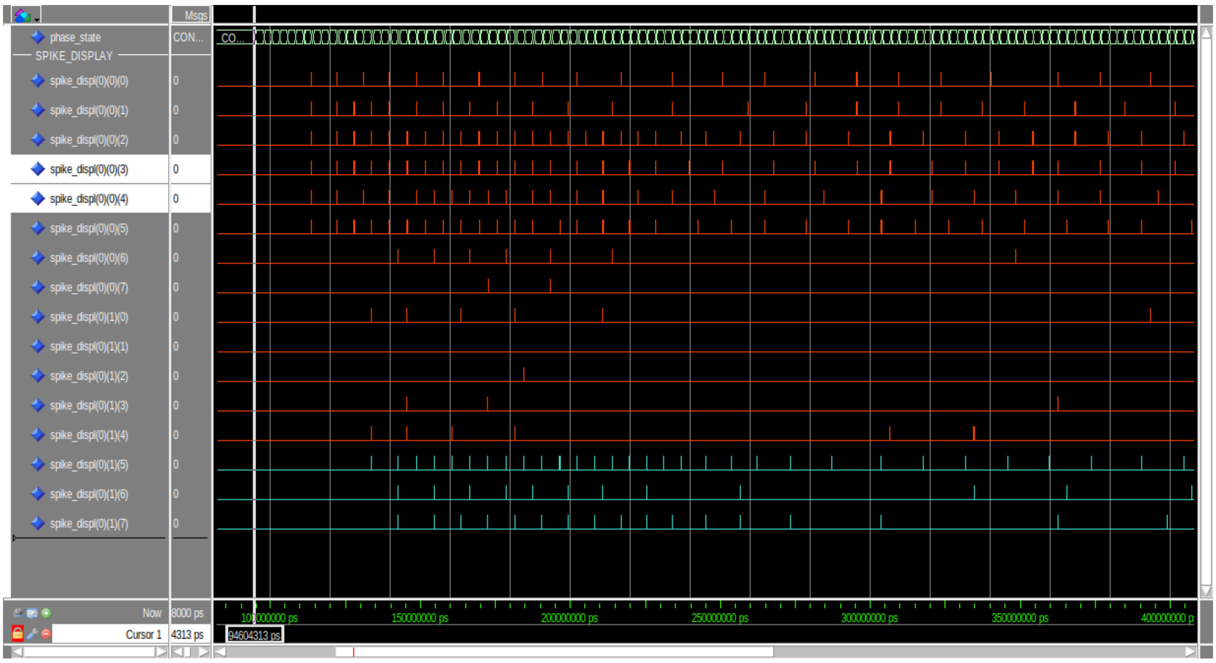

Figure 34: HEENS spiking output on QuestaSim

6 Conclusion and future work

This thesis has focused on the development of some neural models, their theory and their software implementation on the *HEENS* neuromorphic architecture, a device made with the aim of reproducing Spiking Neural Networks.

The example reported in the work is the implementation of the *Adaptive Exponential Integrate and Fire* model, which presents implementation issues in the memory usage and in the approximation of the results. While the number of the constants used in the model cannot be changed, and thus also the needed space in memory, there could be other ways to better calculate the exponential function, and thus reducing errors and limiting the frequency difference from the exact model.

Also, a simulation of a Spiking Neural Network has been proposed and compared with an analogue technology implementing the Izhikevich neural model. Qualitatively the obtained results reproduce the expected and desired behaviour, but finer modifications of the parameters are needed in order to get a better fit of the output firing pattern.

Originally, the net was designed to also have an output trained layer in order to recognize input patterns, so a possible future work is to develop the mentioned layer, define some input patterns and train the network to prove whether it could be able to solve classification tasks.

Acknowledgements

As a conclusion of a long path, I would like to thank here all the people that shared their support and love throughout the years, starting with my parents, my siblings and all my family.

A special thank also to professor Jordi Madrenas, supervisor of this work and that gave me this opportunity, and to its assistant Bernardo Vallejo, without whom the thesis would have been finished by the year 3023.

Last but not least, a thank to all my friends for standing me, for the company received in the infinite hours spent in libraries and for all the moments we still have to live.

"Gentlemen, it has been a privilege playing with you tonight." (Titanic, 1997)

References

[1] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity.* Cambridge University Press, 2002.

[2] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition.* Cambridge University Press, USA, 2014.

[3] Matteo Cucchi, Steven Abreu, Giuseppe Ciccone, Daniel Brunner, and Hans Kleemann. Hands-on reservoir computing: a tutorial for practical implementation. *Neuromorphic Computing and Engineering*, 2(3):032002, aug 2022.

[4] Mireya Zapata Rodriguez. *Arquitectura escalable SIMD con conectividad jerárquica y reconfigurable para la emulación de SNN.* PhD thesis, UPC, Departament d'Enginyeria Electrònica, Sep 2017.

[5] Antonio Caruso. *Izhikevich neural model and STDP learning algorithm mapping on spiking neural network hardware emulator.* PhD thesis, UPC, Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, Departament d'Enginyeria Electrònica, Nov 2020.

[6] Clément Nader. *Real-time display of a multiprocessor spiking neural network.* PhD thesis, UPC, Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, Departament d'Enginyeria Electrònica, Feb 2022.

[7] Hodgkin L. and Huxley F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The journal of Physiology*, 117:500–544, 1952.

[8] Gerstner W. and Werner M.K. *Spiking Neuron Models : Single Neurons Populations Plasticity.* Cambridge U.K: Cambridge University Press, 2002.

[9] E.M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003.

[10] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology*, 94(5):3637–3642, 2005. PMID: 16014787.

[11] Richard Naud, Nicolas Marcille, Claudia Clopath, and Wulfram Gerstner. Firing patterns in the adaptive exponential integrate-and-fire model. *Biol. Cybern.*, 99(4–5):335–347, nov 2008.

[12] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. The MIT Press, 07 2006.

[13] Shigeo Sato, Satoshi Moriya, Yuka Kanke, Hideaki Yamamoto, Yoshihiko Horio, Yasushi Yuminaka, and Jordi Madrenas. A subthreshold spiking neuron circuit based on the izhikevich model. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 177–181, Cham, 2021. Springer International Publishing.

[14] Brian2 Contributors. Brian2: Examples, brette_gerstner_2005 - brian2 simulator. URL: https://brian2.readthedocs.io/en/stable/examples/frompapers.Brette_Gerstner_2005.html (Online, accessed on September 01, 2023).

[15] Brian2 Contributors. Brian2: Examples, hodgkin_huxley_1952 - brian2 simulator. URL: https://brian2.readthedocs.io/en/stable/examples/compartmental.hodgkin_huxley_1952.htm (Online, accessed on September 01, 2023).

[16] Wikipedia Contributors. Biological neural model - wikipedia, the free enciclopedia. URL: https://en.wikipedia.org/wiki/Biological_neuron_model (Online, accessed on September 01, 2023).

A aEIF MATLAB Code

aEIF Simulation

```
clear
close all

S = zeros(1);
seconds_sim = 1;
exec_cycles = 1000 * seconds_sim;
dt = 1;
Vmax = 2^15-1;
Vmin = -2^15;
v0 = -7000;
u0 = -1400;
Vpeak = 3000;
scale_I_factor = 100;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%% REGULAR SPIKING %%%%%%%%%%%%%%%%
rs.gl = 10;
rs.C = 200;
rs.C_div = 327;
rs.delta_t = 200;
rs.Vt = -5000;
rs.El = -7000;
rs.Vrst = -5800;
rs.Vpeak = Vpeak;
rs.a = 2;
rs.b = 0;
rs.tau_u = 30;
rs.v0 = v0;
rs.u0 = u0;
rs.dt = dt;
rs.Vmax = Vmax;
rs.Vmin = Vmin;

rs.const_curr = 500 * scale_I_factor;
rs.I = rs.const_curr*ones(length(S), exec_cycles); %for exact model
rs.I_C = floor(rs.const_curr/rs.C)*ones(length(S), exec_cycles);  %I/C for heens model

rs.root = -4494;
rs.fv_a = 32;
rs.fv_b = 1241;
rs.fv_c = 11950;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% SPIKE FREQUENCY ADAPTATION %%%%%%%%%%
fa.gl = 12;
fa.C = 200;
fa.C_div = 327;
fa.delta_t = 200;
fa.Vt = -5000;
fa.El = -7000;
fa.Vrst = -5800;
fa.Vpeak = Vpeak;
fa.a = 2;
fa.b = 6000;
```

```
fa.tau_u = 300;
fa.v0 = v0;
fa.u0 = u0;
fa.dt = dt;
fa.Vmax = Vmax;
fa.Vmin = Vmin;

fa.const_curr = 500 * scale_I_factor;
fa.I = fa.const_curr*ones(length(S), exec_cycles) * scale_I_factor;
fa.I_C = floor(fa.const_curr/fa.C)*ones(length(S), exec_cycles);

fa.root = -4494;
fa.fv_a = 39;
fa.fv_b = 1491;
fa.fv_c = 14175;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% INITIAL BURSTING %%%%%%%%%%%%%%%%%%
ib.gl = 18;
ib.C = 130;
ib.C_div = 504;
ib.delta_t = 200;
ib.Vt = -5000;
ib.El = -5800;
ib.Vrst = -5000;
ib.Vpeak = Vpeak;
ib.a = 4;
ib.b = 12000;
ib.tau_u = 150;
ib.v0 = v0;
ib.u0 = u0;
ib.dt = dt;
ib.Vmax = Vmax;
ib.Vmin = Vmin;

ib.const_curr = 400 * scale_I_factor;
ib.I = ib.const_curr * ones(length(S), exec_cycles) * scale_I_factor;
ib.I_C = floor(ib.const_curr/ib.C) * ones(length(S), exec_cycles);

ib.root = -4650;
ib.fv_a = 57;
ib.fv_b = 2210;
ib.fv_c = 21400;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% TONIC BURSTING %%%%%%%%%%%%%%%%%%%%
tb.gl = 10;
tb.C = 200;
tb.C_div = 327;
tb.delta_t = 200;
tb.Vt = -5000;
tb.El = -5800;
tb.Vrst = -4600;
tb.Vpeak = Vpeak;
tb.a = 2;
tb.b = 10000;
tb.tau_u = 120;
tb.v0 = v0;
tb.u0 = u0;
tb.dt = dt;
tb.Vmax = Vmax;
```

```
tb.Vmin = Vmin;

tb.const_curr = 210 * scale_I_factor;
tb.I = tb.const_curr * ones(length(S), exec_cycles) * scale_I_factor;
tb.I_C = floor(tb.const_curr/tb.C) * ones(length(S), exec_cycles);

tb.root = -4650;
tb.fv_a = 21;
tb.fv_b = 810;
tb.fv_c = 7796;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%% APPROXIMATED SIMULATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[v1_approx, u1_approx, firings1_approx] = aEIF_HEENS(S, rs.I_C, exec_cycles, rs);

[v2_approx, u2_approx, firings2_approx] = aEIF_HEENS(S, fa.I_C, exec_cycles, fa);

[v3_approx, u3_approx, firings3_approx] = aEIF_HEENS(S, ib.I_C, exec_cycles, ib);

[v4_approx, u4_approx, firings4_approx] = aEIF_HEENS(S, tb.I_C, exec_cycles, tb);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% EXACT SIMULATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[v1, u1, firings1] = aEIF_exact(S, rs.I, exec_cycles, rs);

[v2, u2, firings2] = aEIF_exact(S, fa.I, exec_cycles, fa);

[v3, u3, firings3] = aEIF_exact(S, ib.I, exec_cycles, ib);

[v4, u4, firings4] = aEIF_exact(S, tb.I, exec_cycles, tb);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% RASTER PLOT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure(1)

plot(firings1_approx(:,1), 15+firings1_approx(:,2),'*r');
title('aEIF Different Behaviours', 'FontSize',30)
xlabel("time(ms)")

hold on
ylim([5 65])
yticks([])
plot(firings1(:,1), 10+firings1(:,2),'*k');

plot(firings2_approx(:,1), (30+firings2_approx(:,2)),'*r');
plot(firings2(:,1), (25+firings2(:,2)),'*k');

plot(firings3_approx(:,1), (45+firings3_approx(:,2)),'*r');
plot(firings3(:,1), (40+firings3(:,2)),'*k');

plot(firings4_approx(:,1), (60+firings4_approx(:,2)),'*r');
plot(firings4(:,1), (55+firings4(:,2)),'*k');

annotation('textbox',[0.012 0.05 .05 .2], 'String',"Neuron 4",'FitBoxToText','on','FontSize',10)
annotation('textbox',[0.012 0.25 .05 .2], 'String',"Neuron 3",'FitBoxToText','on','FontSize',10)
annotation('textbox',[0.012 0.45 .05 .2], 'String',"Neuron 2",'FitBoxToText','on','FontSize',10)
annotation('textbox',[0.012 0.65 .05 .2], 'String',"Neuron 1",'FitBoxToText','on','FontSize',10)
```

```matlab

%%%%%%%%%%%%%%%%%%%% MEMBRANE POTENTIALS PLOT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure(2)
dim = [0 500 -8000 33000];

subplot(4,1,1)
plot(v1_approx, 'r')
hold on
plot(v1, 'k');
axis(dim)
title('Regular spiking')

subplot(4,1,2)
plot(v2_approx, 'r')
hold on
plot(v2, 'k')
axis(dim)
title('Spike Frequency Adaptation')

subplot(4,1,3)
plot(v3_approx, 'r')
hold on
plot(v3, 'k')
axis(dim)
title('Initial Bursting')

subplot(4,1,4)
plot(v4_approx, 'r')
hold on
plot(v4, 'k')
axis(dim)
title('Tonic Bursting')

annotation('textbox',[0.012 0.05 .05 .2], 'String',"Neuron 4",'FitBoxToText','on','FontSize',20)
annotation('textbox',[0.012 0.25 .05 .2], 'String',"Neuron 3",'FitBoxToText','on','FontSize',20)
annotation('textbox',[0.012 0.45 .05 .2], 'String',"Neuron 2",'FitBoxToText','on','FontSize',20)
annotation('textbox',[0.012 0.65 .05 .2], 'String',"Neuron 1",'FitBoxToText','on','FontSize',20)
```

aEIF exact model

```
function [v, u, firings] = aEIF_exact (S, Iin, cycles, constants)
    N = length(S);

    %constants definition
    C = constants.C;
    gl = constants.gl;
    El = constants.El;
    Vt = constants.Vt;
    Vpeak = constants.Vpeak;
    Vrst = constants.Vrst;
    delta_t = constants.delta_t;
    tau_u = constants.tau_u;
    a = constants.a;
    b = constants.b;
    v0 = constants.v0;
    u0 = constants.u0;
    dt = constants.dt;
    Vmax = constants.Vmax;
    Vmin = constants.Vmin;

    exec_cycles = cycles/dt; %length of simulation

    v = zeros(N, exec_cycles); %membrane potential evolution in time
    u = zeros(N, exec_cycles); %recovery potential evolution in time
    firings = [];       %output firings

    v(:,1) = v0; %v init
    u(:,1) = u0; %u init

    v_n = v0*ones(N,1); %temp variable for storing membrane potential
    u_n = u0*ones(N,1); %temp variable for storin recovery potential
    fv_ap = zeros(N, 1);     %approximation of the exponential function

    I = zeros(N, 1);            %external current at each cycle

    %simulating #cycles ms
    for cycle = 1:exec_cycles
        t = cycle*dt; % time in ms

        v_n = v(:, cycle); % current membrane pot
        u_n = u(:, cycle); % current recovery pot

         I = Iin(:, cycle)/C; % input current divided by C for later operations


        %firings and synaptic currents evaluation
        for i = 1:N  % for every neuron in the net
            if v_n(i) >= Vpeak      %if v > peak then fire
                firings = [firings; t, i-1];    %store the firing

                v_n(i) = Vrst;      %restore membrane potential
                u_n(i) = u_n(i) + b; %increment recovery potential

                for j = 1:N % for every synapse of the neuron
                    I(j) = I(j) + S(i, j);  %store outgoing current
```

```
                end
            end
        end

        %model simulation
        for i = 1:N

                fv(i, cycle) = (-gl*(v_n(i) - El) + gl*delta_t.*exp((v_n(i)-Vt)/delta_t))/C;    %evaluate fv

                dv(i) = fv(i, cycle) - u_n(i)/C + I(i); %calculate dv
                dv(i) = dv(i)*dt;

                du(i) = a*(v_n(i) - El) - u_n(i);   %calculate du
                du(i) = du(i) / tau_u * dt;

        end

        %updating the neurons
        v(:, cycle+1) = v_n(:) + dv(:); %update membrane potential
        v(:, cycle +1) = min( max(v(:, cycle+1), -Vmin) , Vmax);

        u(:, cycle+1) = u_n(:) + du(:); %update recovery potential

    end
```

aEIF approximated model

```
function [v, u, firings] = aEIF_HEENS(S, Iin, cycles, constants)
    N = length(S);

    %constants definition
    C = constants.C;
    C_div = constants.C_div;
    gl = constants.gl;
    El = constants.El;
    Vt = constants.Vt;
    Vpeak = constants.Vpeak;
    Vrst = constants.Vrst;
    delta_t = constants.delta_t;
    tau_u = constants.tau_u;
    tau_u_div = floor(2^16/tau_u);
    a = constants.a;
    b = constants.b;
    v0 = constants.v0;
    u0 = constants.u0;
    dt = constants.dt;
    Vmax = constants.Vmax;
    Vmin = constants.Vmin;

    %approximated function constants
    root = constants.root;
    fv_a = constants.fv_a;
    fv_b = constants.fv_b;
    fv_c = constants.fv_c;

    exec_cycles = cycles/dt; %length of simulation

    v = zeros(N, exec_cycles); %membrane potential evolution in time
    u = zeros(N, exec_cycles); %recovery potential evolution in time
    firings = [];       %output firings

    v(:,1) = v0; %v init
    u(:,1) = u0; %u init

    v_n = v0*ones(N,1); %temp variable for storing membrane potential
    u_n = u0*ones(N,1); %temp variable for storin recovery potential

    fv_ap = zeros(N, 1);     %approximation of the exponential function
    I = zeros(N, 1);           %external current at each cycle

    %simulating #cycles ms
    for cycle = 1:exec_cycles
        t = cycle*dt; % ms with dt resolution
        v_n = v(:, cycle);
        u_n = u(:, cycle);
        I = Iin(:, cycle); %getting external current

        %firings and synaptic currents evaluation
        for i = 1:N  % for every neuron in the net
            if v_n(i) >= Vpeak      %if v > peak then fire
                firings = [firings; t, i-1];    %store the firing

```

```
        v_n(i) = Vrst;      %restore membrane potential
        u_n(i) = u_n(i) + b; %increment recovery potential
        u_n(i) = clip(Vmin, Vmax, u_n(i));

        for j = 1:N % for every synapse of the neuron
            I(j) = I(j) + S(i, j);  %store outgoing current
        end
    end
end
%model execution
for i = 1:N
    %evaluation of linear term always perfomed
        fv_ap(i) = El-v_n(i);
        fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

        fv_ap(i) = fv_ap(i)*gl;     % +gl(El-v) == -gl(v-El)
        fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

        fv_ap(i) = fv_ap(i)*C_div; gl(El-v)*C*2^16
        fv_ap(i) = floor(fv_ap(i)/2^16);    %fv_ap = gl(El-v)
        fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

        %if  Vt < v_n
        if Vt - v_n(i) < 0
            %evaluate quadratic term of approximation
            fv_ap(i) = floor(v_n(i)^2/2^16);    % v^2/2^16
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
            fv_ap(i) = fv_ap(i)*fv_a;       %fv_ap = fv_a*v^2
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

            %evaluate first order term of approximation
            tmp = floor(v_n(i)/2)*fv_b;     %v/2 * fv_b *2^8
            tmp = floor(tmp/2^8);   v/2*fv_b
            tmp =clip(Vmin, Vmax, tmp);

            fv_ap(i) = fv_ap(i) + tmp; %fv_a*v^2 + fv_b*v/2
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
            fv_ap(i) = fv_ap(i) + tmp;  % fv_ap = fv_a*v^2+fv_b*v
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

            fv_ap(i) = fv_ap(i) + fv_c; %fv_ap = fv_a*v^2 + fv_b*v + fv_c

            %perform min function and get result
            tmp = 0;
            if fv_ap(i) >= 0    %if fv_ap is positive, save in tmp but reset for later
              tmp = fv_ap(i);   %only if fv_ap was >= 0
              clip(Vmin, Vmax, tmp);
              fv_ap(i) = 0;    %if fv_ap > 0, here put fv_ap = 0
            end
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));

        end
        %if v_n > root
        if root - v_n(i) < 0
            fv_ap(i) = (4*tmp); %max func not needed because tmp >= 0
            fv_ap(i) = clip(Vmin, Vmax, fv_ap(i));
        end

        dv(i) = I(i) - floor((u_n(i)* C_div)/2^16 ) ;
```

```
                dv(i) = dv(i)*dt  +  dt*fv_ap(i);   %get total dv

                du(i) = a*(v_n(i) - El) - u_n(i);
                du(i) = floor((du(i) * floor(2^16/tau_u))/2^16 ) * dt; %get du
        end
        %updating the neurons
        v(:, cycle+1) = v_n(:) + dv(:); %update membrane potential
        v(:, cycle +1) = clip(Vmin, Vmax, v(:, cycle+1) );

        u(:, cycle+1) = u_n(:) + du(:); %update recovery potential
        u(:, cycle +1) = clip(Vmin, Vmax, u(:, cycle+1) );
    end
```

B aEIF HEENS Code

Netlist file

```
@Config
Zedboard_4x8

@ParamSyn
# synaptic weights = 0
0, 0
@Netlist
#empty netlist
0, 0

@Params
#membrane potential and recovery variable at t = 0
.0x1E3/16/NEUR/$NVL/-7000, -1400

# all the model's variables
.0x3E0/16/EL_GL/$NVL/0 , 0
0, -7000, 10
1, -7000, 12
2, -5800, 18
3, -5800, 10
UNMAPPED, 0, 0

.0x3E1/16/V_RST_CONST_CURR/$NVL/0, 0
0, -5800, 250
1, -5800, 250
2, -5000, 307
3, -4600, 105
UNMAPPED, 0, 0

.0X3E2/16/C_DIV_TAU_U/$NVL/0, 0
0, 327, 2184
1, 327, 218
2, 504, 436
3, 327, 546
UNMAPPED, 0, 0

.0x3E3/16/NEU_A_B/$NVL/0, 0
0, 2, 0
1, 2, 6000
2, 4, 12000
3, 2, 10000
UNMAPPED, 0, 0

.0x3E4/16/FV_A_B/$NVL/0, 0
0, 32, 1241
1, 39, 1491
2, 57, 2210
3, 21, 810
UNMAPPED, 0, 0

.0x3E5/16/FV_C_ROOT/$NVL/0, 0
0, 11950, -4494
1, 14175, -4494
```

```
2, 21400, -4650
3, 7796, -4650
UNMAPPED, 0, 0

#seed for noise generation
.0x1FD/32/SEED/2/-6500, 800
5, 10
```

aEIF Neural Model

```
;;;;;;;;;;;;;;;;REGISTERS USAGE;;;;;;;;;;;;;;;;;;;
;  R0: CALCULATIONS                     SR0: UNUSED
;  R1: CALCULATION                      SR1: TMP STORAGE
;  R2: STORING MEMBRANE POTENTIAL       SR2: STORING dV
;  R3: STORING RECOVERY POTENTIAL       SR3: STORING dU
;  R4: TMP STORAGE                      SR4: UNUSED
;  R5: TMP STORAGE                      SR5: UNUSED
;  R6: TOTAL SYNAPTIC CURRENT           SR6: UNUSED
;  R7: UNUSED                           SR7: UNUSED
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .org 0x010
    .data
VPEAK        = 3000
VT           = -5000
DELTA_T      = 200
DELTA_T_DIV  = 327 ; 2^16/DELTA_T

    .org 0x70
    .code

GOTO MAIN

;;;;;;;;;;;;;;;;;;;;;; FUNCTIONS ;;;;;;;;;;;;;;;;;
RANDOM_INIT:                 ; Uses R0 and R1
        LOADBP SEED_0
        LOADSN
        SEED                                ; High seed
        LOADBP SEED_1
        LOADSN
        SEED                                ; Low seed
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

LOAD_NEURON:          ;
        READMPV NEUR_0  ; Address of real neuron + virt (valid also for non-virtual)
        LOADBP                    ; SNRAM pointer to currently processed neuron
        LOADSN                   ; Load Neural parameters from SNRAM to R1<=u & ACC<=Vmem
        MOVR R2                   ; R2 <= v0
        MOVA R1                   ; ACC<=u0
        MOVR R3                   ; r3<=u
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SYNAPSE_CALC:
    LOADSP                    ; Load Synaptic parameters and spike to R1 & ACC
    SHRN 1                    ; Move spike to flag
    FREEZENC
            MOVA R1              ; Synaptic parameter to ACC
            ADD R6
            MOVR R6
    UNFREEZE
    RST ACC
    STORESP                    ; Stores synaptic parameter and increases BP for
                     ; next synapse processing
```

```
    INCS
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ADD_CONST_CURR:
    READMPV V_RST_CONST_CURR_0
    LOADBP
    LOADSN
    MOVA R1
    ADD R6          ;R6 is total syn current + const_curr
    MOVR R6
    RST ACC
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
EVAL_FV:
    ;1) evaluate below vt : fv = -gl(v - El) -> gl(el - v), R0 KEEPS EL, R1 KEEPS GL
        READMPV EL_GL_0
        LOADBP
        LOADSN
        SUB R2        ; EL - V
        MULS R1        ; GL*(EL - V) ; RESULT SHOULD BE IN R1 BECAUSE LSB

        MOVSR R1        ;R1S SAVES THE VALUE GL(EL-V)
        READMPV C_DIV_TAU_U_0 ; 1/C*2^16 IN R0, TAU_U IN R1
        LOADBP
        LOADSN
        MOVRS R1
        MULS R1 ; RESULT IS IN R0 BECAUSE MSB
        MOVR R1 ; R1 <= GL(EL - V)/C
        MOVSR R1 ; GL(EL-V)/C -> FV RESULT IF V < VT

    ;2) evaluate v - Vt > 0 : fv = fv_a*v^2 + fv_b*v + fv_c
    LDALL R0, VT
    SUB R2      ;VT - V
    SHLN 1      ;
    FREEZENC
        ;evaluate fv_a*v^2
        MOVA R2
        MULS R2 ; V^2 BUT TAKE V^2/2^16 CONSIDERING ONLY R0
        MOVR R5 ; V^2/2^16 IN R5
        READMPV FV_A_B_0
        LOADBP
        LOADSN
        MULS R5  ; R0 AND R1 KEEPS fv_a*V^2, result in R1
        MOVA R1     ; ACC <= R1
        MOVR R4     ; QUADRIC TERM IN R4

        ;evaluate 2 times fv_b*v/2
        MOVA R2
        SHRAN 1
        MOVR R5
        READMPV FV_A_B_0
                LOADBP
                LOADSN
                MOVA R1        ; R0 KEEPS FV_B
                MULS R5        ; V/2*FV_B AND RESULT IN R0[7:0] AND R1[15:8]
                SHLN 7        ; R0 KEEPS RESULT IN R0[15:8] AND R0[7:0] = 0
        SHLN 1
                MOVR R5        ; R5 USED AS TMP REGISTER
                MOVA R1        ; R0 NOW KEEPS RESULT IN R0[15:8]
```

```
                SHRN 7        ; R0[7:0] AND R0[15:8] = 0
        SHRN 1
                OR R5                 ; R5[15:8] OR R0[15:8]=0 --- R5[7:0]=0 OR R0[7:0] ---> SHOULD BE  2^8*FV_B

        MOVR R1
                ADD R4
        ADD R1
                MOVR R4        ; R4 KEEPS FIRST AND SECOND ORDER TERMS

                READMPV FV_C_ROOT_0
                LOADBP
                LOADSN
                ADD R4        ; NOW R0 KEEPS THE RESULT OF TOTAL FV APPROXIMATED

        MOVR R1 ; SAVES RESULT ON R1
        RST R5  ; R5 TO 0
        SHLN 1  ; SET CARRY FLAG
        FREEZEC ;FREEZE IF RESULT IS NEGATIVE, OTHERWISE SAVE STIT FOR LATER
            MOVA R1     ;GET THE POS RESULT STORE IN R1
            MOVR R5     ;R5 != 0 ONLY WHEN RESULT OF FV IS POSITIVE
            RST R1      ;RESET R1 TO PERFORM MIN(0, FV)
        UNFREEZE

                MOVSR R1        ; NOW R1S KEEPS THE TERM OF FV

        UNFREEZE ; ENDS 2)

    ;3)evaluate ROOT - V >= 0 : 4*fv of case 2)
    READMPV FV_C_ROOT_0
    LOADBP
    LOADSN
    MOVA R1
    SUB R2  ; ROOT - V
    SHLN 1
    FREEZENC  ; ROOT - V >= 0 FREEZE, SO DONT FREEZE WHEN V > ROOT
        MOVA R5 ;GET MAX(0, FV)

        ADD R5  ; 2*FV
        ADD R5  ; 3*FV
        ADD R5  ; 4*FV, USED ADD BECAUSE SATURATES
        MOVR R1
        MOVSR R1    ; PUT IN R1S RESULT
    UNFREEZE

RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
EVAL_DELTA_V:

    MOVSR R2    ; R2S now keeps the membrane val maybe not needed
    MOVSR R3    ; R3S now keeps the recovery pot maybe not needed
    MOVSR R6    ; R6S now keeps total incoming current

    GOSUB EVAL_FV   ; R1S now keeps the value of fv/c

    MOVRS R2
    MOVRS R3
    MOVRS R6

    READMPV C_DIV_TAU_U_0
```

```
    LOADBP
    LOADSN
    MULS R3                ; U/C WITH RESULT IN R0 BECAUSE MSB
    MOVR R1
    MOVA R6
    SUB R1                 ; ACC <= TOTAL_I - U/C

    MOVR R1      ; R1 <= U/C

    MOVRS R1     ;RETRIEVE FV VALUE
    ADD R1       ;DV IN R0

    MOVR R2     ; R0S KEEPS DV
    SWAPS R2     ; R2 Vmemb, R2S dV
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
EVAL_DELTA_U:
        READMPV EL_GL_0
        LOADBP
        LOADSN
        MOVR R1  ; R1 KEEPS EL
        MOVA R2        ; ACC <= VMEMB
        SUB R1        ; ACC <= VMEMB - EL
        MOVR R4        ; R4 USED AS TMP STORAGE

        READMPV NEU_A_B_0
        LOADBP
        LOADSN
        MULS R4        ; A*(VMEMB - EL) WITH RESULT IN R1 BECAUSE LSB
        MOVA R1
        SUB R3        ; ACC <= A(VMEMB-EL) - U
        MOVR R4

        READMPV C_DIV_TAU_U_0
        LOADBP
        LOADSN
        MOVA R1        ; ACC <= TAU_U
        MULS R4        ; ACC&R1 <= (A(VMEMB-EL) - U )/TAU_U WITH RESULT IN R0 BECAUSE MSB
        MOVSR R3        ;SAVES U INTO R3S
        MOVR R3        ; R3 <= DU
        SWAPS R3        ; R3 <= U, R3S <= DU

RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DETECT_SPIKE:
    LDALL R0, VPEAK
    SUB R2                 ; Vthres - Vmemb
    SHLN 1                 ; if MSB = 1 then Vmemb > Vthres so it fired
    RST ACC
    FREEZENC
        SWAPS R2
        SWAPS R3
        MOVR R2 ; DV = 0
        MOVR R3 ; DU = 0
        SWAPS R2
        SWAPS R3

        READMPV V_RST_CONST_CURR_0
        LOADBP
```

```
        LOADSN
        MOVR R2  ; R2 <= V_RST

        READMPV NEU_A_B_0
        LOADBP
        LOADSN
        MOVA R1
        ADD R3
        MOVR R3  ; R3 <= U + B

        SET ACC ; put a 1 in R0 LSB for tx a spike
   UNFREEZE

    STOREPS
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
CALC_STEP:
    MOVA R2      ; ACC = Vmemb
    SWAPS R2     ; R2 = dV
    ADD R2       ; ACC = Vmemb + dV
    MOVR R2      ; R2 = Vmemb + dV

    MOVA R3      ; same for R3 and U-dU
    SWAPS R3
    ADD R3
    MOVR R3
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
STORE_NEURON:
    MOVA R3      ; acc = U
    MOVR R1      ; r1 = U
    MOVA R2      ; acc = Vmemb

    READMPV NEUR_0  ; Address of real neuron + virt (valid also for non-virtual)
    LOADBP          ; SNRAM pointer to currently processed neuron

    STORESP         ; Store u and Vmem to SNRAM
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;; MAIN PROGRAM ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MAIN:
    ; Virtual operation init
    LAYERV NVL          ; Init sequencer vlayers. It is 0 for non-virtual operation
    LDALL ACC, NVL      ; Load defined virtual layers to PE array
    SPMOV 0                    ; VIRT <= ACC

    ; Initial instructions
    GOSUB RANDOM_INIT        ; For noise initialization

EXEC_LOOP:              ; Execution loop

    LOOP    NVL         ; Virtualization loop
        SYNAPSE NLS_0
        GOSUB LOAD_NEURON   ; Loading current neuron

        GOSUB DETECT_SPIKE; ;check for spike

        RST R6              ;reset register for current storing
```

```
        READMPV LSA0_0      ; needed for configuration
        LOADBP
        LOOPV NLS_0         ; synaptic loop. Reads number of current-layer synapses
            GOSUB SYNAPSE_CALC  ;calculate synaptic currents
        ENDL

        GOSUB ADD_CONST_CURR    ;add DC const current

        GOSUB EVAL_DELTA_V;     ;evaluate dv/dt
        GOSUB EVAL_DELTA_U;     ;evaluate du/dt
        GOSUB CALC_STEP;        ; update v and u

        MOVA R2 ; R0 <= membrane potential
        STOREB      ; value of R0 in fifo for visualization of results

        GOSUB STORE_NEURON;     ;store back neuron state

        RST ACC
        RST R3
        RST R2
        INCV            ;increment virtual layer
    ENDL            ;end virtualization loop

SPKDIS        ; Distribute spikes
GOTO EXEC_LOOP ; Execution loop

```

C Reservoir Network Code

MATLAB Simulation

```
%indexes for neurons
neu_0 = 1; neu_1 = 2; neu_2 = 3; neu_3 = 4; neu_4 = 5; neu_5 = 6;
neu_6 = 7; neu_7 = 8; neu_8 = 9; neu_9 = 10; neu_10 = 11; neu_11 = 12;
neu_12 = 13; neu_13 = 14; neu_14 = 15; neu_15 = 16;
Ne=13;   % # of excitatory neurons
Ni=3;   % # ofinhibitory neurons

Vrest = -70;    %resting potential
Vthres = 30;    %threshold potential

%coefficient for exitatory neurons
a_ex = 0.015;
b_ex = 0.15;
c_ex = -70;
d_ex = 6;
%coefficient for inhibitory neurons
a_in = 0.02;
b_in = 0.2;
c_in = -70;
d_in = 2;

%%%%%%%%%%%%%%%% SYNAPSIS MATRIX FROM FILE %%%%%%%%%%%%%%%%%%%%%%%%%%%
fileID = fopen('reservoir_net.txt', 'r');
sizeA = [3 inf];
file_array = fscanf(fileID, '%d, %d, %d', sizeA);
fclose(fileID);

S_dim = Ne + Ni;
S = zeros(S_dim);

for j = 1:length(file_array)
    %dividing by 100 because the weigths in the file are in uV
    S(file_array(1,j)+1 , file_array(2, j)+1) = file_array(3, j)/100; % /100 because weights in uV in the netli
end
clear fileID sizeA file_array;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Init v and u
v=Vrest*ones(Ne+Ni,1);              % Initial values of v

u(1:Ne,1)=b_ex.*v(1:Ne);                % Initial values of u for excitatory
u(Ne+1 : Ne+Ni,1) = b_in.*v(Ne+1 : Ne+Ni);            % Initial values of u for inhibitory

%init current
I = zeros(Ne+Ni,1);
const_curr = 4;  %The value of the const current that goes to neurons 0 to 5
I_noise = 0;    %current noise set to 1
tau_I = 20;     %decay constant for the current

%init plot variables and exec time
exec_cycle = 125;

```

```
neu2plot = neu_0;   %neuron that will be plotted (its membrane pot)
memb2plot = [];     %plot-related vars
u2plot = [];

memb2plot = [1, v(neu2plot)];        % init plot-relateted vars
u2plot = [1, u(neu2plot)];
all_firings=[];                      % spike timings for raster plot

%%%%%%%%%%%%%%%%% SIMULATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simulation of exec_cycle time stamps
for t=1:exec_cycle
    fired = [];
    x_fired = zeros(Ne+Ni,1);

    %constant current to apply to neuron 0 to 5
    I = I*exp(-1/tau_I);
    I(neu_0:neu_5) = const_curr + I(neu_0:neu_5) ;

    for i=1:Ne         %for every exitatory neuron
        if v(i) >= Vthres    %if fired
           all_firings = [all_firings; t, i-1];
           fired = [fired, i];

           v(i) = c_ex;              %update fired neuron params
           u(i) = u(i) + d_ex;

           for j = 1:length(S)   %evaluate its spikes
                I(j) = I(j) + S(i,j); %
           end
        end
    end

    for i = Ne+1:(Ne+Ni)       %for every inhibitory neuron
        if v(i) >= Vthres   %if fired
           all_firings = [all_firings; t, i-1];
           fired = [fired, i];

           v(i) = c_in;              %update neuron params
           u(i) = u(i) + d_in;

           for j = 1:length(S)   %evaluate neuron spikes
                I(j) = I(j) + S(i,j) ;
           end
        end
    end

    for i = 1:(Ne+Ni)  %for every neuron update v and u
            %calculate dv in two steps
            v(i) = v(i) + 0.5.*(0.04*v(i).^2 + 5.*v(i) + 140 - u(i) + I(i));
            v(i) = v(i) + 0.5.*(0.04*v(i).^2 + 5.*v(i) + 140 - u(i) + I(i));

            %calculate u depending on the variables
            if i <= Ne
                u(i) = u(i) + a_ex.*(b_ex.*v(i) - u(i));
            else
                u(i) = u(i) + a_in.*(b_in.*v(i) - u(i));
            end

            if v(i) > Vthres %for limiting the values
```

```
                v(i) = Vthres;
            end
        end

        %for plots
        memb2plot = [memb2plot; t, v(neu2plot)];
        u2plot = [u2plot; t, u(neu2plot)];
end

%%%%%%%%%%%% PLOTS AND FIGURES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
set(groot, 'defaultLineMarkerSize', 8)
n_fig = 1;

figure(n_fig) %All firings for all neurons in the execution time
n_fig = n_fig+1;
exc_firing = find(all_firings(:,2) < Ne);
inh_firing = find(all_firings(:,2) >= Ne);

plot(all_firings(exc_firing, 1), all_firings(exc_firing,2),'*r');
hold on
plot(all_firings(inh_firing,1), all_firings(inh_firing,2),'*b');
xlabel('Time')
ylabel('Neurons')
axis([0 exec_cycle -1 (Ne+Ni)])

%Membrane potential of neuron to be plotted (neu2plot)
figure(n_fig)
n_fig = n_fig+1;
plot(memb2plot(:,1), memb2plot(:,2))

figure(n_fig)
n_fig = n_fig+1;
plot(u2plot(:,1), u2plot(:,2))
```

HEENS Netlist file

```
@Config
Zedboard_4x8
@ParamSyn
#Single synaptical weight|R0
400, 0

@Netlist
0   ,   2
0   ,   5
0   ,   12

1   ,   3
1   ,   12
1   ,   14

2   ,   3
2   ,   11
2   ,   15

3   ,   2
3   ,   8
3   ,   13


4   ,   1
4   ,   5
4   ,   8

5   ,   6
5   ,   13

6   ,   4
6   ,   7

7   ,   0
7   ,   14

8   ,   9
8   ,   10

9   ,   10
9   ,   11
9   ,   15

10  ,   2
10  ,   9

11  ,   0
11  ,   7

12  ,   4
12  ,   6

13  ,   1    , -400
13  ,   8    , -400
```

```
14  ,   0    , -400
14  ,   5    , -400
14  ,   12   , -400

15  ,   3    , -400
15  ,   9    , -400
15  ,   11   , -400

@Params
# Addr/Size/Name/Entries/default (empty for random) R0 / R1
.0x1E3/16/NEUR/$NVL/-7000, -1050

.0x3E0/16/IZH_A_B/$NVL/983 , 9830
13, 1310, 13107
14, 1310, 13107
15, 1310, 13107
UNMAPPED, 0, 0

.0x3E8/16/IZH_C_D/$NVL/-7000, 600
13, -7000, 200
14, -7000, 200
15, -7000, 200
UNMAPPED, 0, 0

.0x3F4/16/CONST_CURR/$NVL/0, 0
0, 400 , 0
1, 400 , 0
2, 400 , 0
3, 400 , 0
4, 400 , 0
5, 400 , 0
UNMAPPED, 0, 0

.0x1FD/32/SEED/2/-6500, 800
5, 10
```

HEENS Izhikevich Neural Model

```
   .org 0x010
    .data
;; Membrane potential parameters common to all neurons
VTHRES        = 3000       ; Threshold voltage -25 mV
N70           = 7000        ; 70mV, used as 140/2 for the model
N0002         = 26844       ; 0.0002*2^27, constant for the model
TAU_I         = 31170        ; e^(-1/20)*2^15
NOISE_LIMIT   = 0x3FF; noise for test is 10mV, 1280

    .org 0x70
    .code
GOTO MAIN         ; Jump to main program

;
; **************************** PROCEDURES BEGIN ****************************
;
RANDOM_INIT:                ; Uses R0 and R1
        LOADBP SEED_0
        LOADSN
        SEED                                ; High seed
        LOADBP SEED_1
        LOADSN
        SEED                                ; Low seed
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LOAD_NEURON:        ; Uses R0, R1, R2, R3, R5
        READMPV NEUR_0  ; Address of real neuron + virt (valid also for non-virtual)
        LOADBP                 ; SNRAM pointer to currently processed neuron
        LOADSN                ; Load Neural parameters from SNRAM to R1<=u & ACC<=Vmem
        MOVR R2                ; Move Vmem from ACC to R2
        MOVA R1                ; ACC<=u
        MOVR R3                ; r3<=u
        MARK
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MEMBRANE_POTENTIAL:         ;Uses R0,R4,R7  32808
        MOVA R2
        MULS R0                ; v^2*2^12 (CHANGED MUL IN MULS!!)
        NOP                    ; Check if needed
                        ; Shift ROR1 4 positions left
        SHLN 4                ; Shift Accumulator 2^4
        MOVR R4
        MOVA R1                ; Move LS part (R1) to R0 (2^16)
        SHRN 4
        SHRAN 4
        SHRAN 4                ; 2^16/2^12 = 2^4
        ADD R4                ; Combine and obtain v^2/2^12
        LDALL R4 N0002        ; 0.0002*2^27 is in R4
        MULS R4                ; v^2*2^(-12)*0.0002*2^27/2^16 = 0.0002*v^2*2^(-1)
                    ;(CHANGED MUL IN MULS!!)
        NOP                    ; Check if needed
        SHLN 1                ; Shift Accumulator 2^1
        MOVR R4
        MOVA R1                ; Move LS part (R1) to R0 (2^16)
        SHRN 5
```

```
        SHRAN 5
        SHRAN 5                   ; 2^16/2^15 = 2^1
        ADD R4                   ; Combine and obtain 0.0002*v^2
        MOVR R7
        MOVA R2         ; ACC<=Vinit
        SHRAN 2         ; ACC<=0.25*Vinit;
        ADD R2          ; ACC<=ACC+Vinit=1.25*Vinit
        SHLAN 1         ; ACC<=2*ACC =2.5*Vinit
        ADD R7
        LDALL R4 N70    ;          R4<=70
        ADD R4
        MOVR R7
        RST ACC
        SUB R3          ;          ACC=- u
        SHRAN 1
        ADD R7
        ADD R2          ;         ACC=ACC+Vinit
        MOVR R2         ;         Back to R2 where membrane potential is stored
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ADD_NOISE:              ; Uses R0, R2 and R5
        RANDON              ; LFSR ON
        LLFSR               ; Noise to ACC
        MOVR    R5
        LDALL   ACC, NOISE_LIMIT
        AND     R5
    RANDOFF             ; LFSR OFF. Arbitrarily here
        SHRN    1
        FREEZENC
            MOVR    R5
            RST     ACC
            SUB     R5      ; Generate signed noise without the negative bias of two's complement
        UNFREEZE
    MOVSR   ACC         ; TO MONITOR THE NOISE
        ADD     R2          ; Add to Vmem
        MOVR    R2          ; Back to R2
    RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
SYNAPSE_CALC:
    LOADSP                    ; Load Synaptic parameters and spike to R1 & ACC
    SHRN 1                    ; Move spike to flag C
    FREEZENC
            MOVA R1                ; Synaptic parameter to ACC
        SWAPS R1
            ADD R1
            MOVR R1
            SWAPS R1
    UNFREEZE
    RST ACC
    STORESP                    ; Stores synaptic parameter and increases BP for
                          ; next synapse processing
    INCS
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
RECOVERY_UPDATE: ;uses R3,R5,R6

    READMPV IZH_A_B_0
    LOADBP
    LOADSN  ; R0 <= A, R1 <= B
```

```
    MOVR R6   ;R6 <= A
    MOVA R1
    MOVR R5 ; R5 <= B
        MOVA R2               ;ACC<=Vinit
        MULS R5        ;ACC<=R5*ACC=B*Vinit
        SUB R3         ;ACC<= ACC-R3= B*VMEMB-U

        MULS R6        ;ACC<=A*ACC;
        ADD R3         ;ACC<=ACC+Uinit
        MOVR R3        ;Back to R3 where recovery value is stored
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DETECT_SPIKE:            ; Uses R0,R3 and R2
        LDALL ACC, VTHRES
        SUB R2                   ; Compare Vth - Vmem
        SHLN 1                   ;subtraction sign to C flag
    RST ACC
        FREEZENC            ; If positive, freeze
        READMPV IZH_C_D_0
        LOADBP
        LOADSN  ;R0 <= C, R1 <= D
        MOVR R2 ; VMEMB = C

                MOVA R1     ; R0 <= D
                ADD R3      ; ACC<= u+d
                MOVR R3     ;        u<= u+d
                SET ACC
        UNFREEZE
        STOREPS                   ; Push spikes
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
STORE_NEURON:       ; uses R0,R3 and R1
        MOVA R3                   ;move u from R3 to acc
        MOVR R1                   ;move u from ACC to R1
        MOVA R2                   ; Move Vmem from R2 to ACC
        READMPV NEUR_0  ; Address of real neuron + virt (valid also for non-virtual)
        LOADBP          ; SNRAM pointer to currently processed neuron
        STORESP         ; Store u&Vmem to SNRAM
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ADD_CONST_CURR:
    READMPV CONST_CURR_0    ;read address for constant current in SNRAM
    LOADBP      ;load pointer
    LOADSN  ; R0 <= CONST_CURR , R1 <= TAU_I
    SWAPS R1  ;R1 <= TOTAL I
    ADD R1    ; R0 <= CONST_CURR + TOT_I
    MOVR R1
    SWAPS R1 ;  R1S <= TOTAL CURRENT
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
CURR_DECAY:
    LDALL R0, TAU_I ;R0 <= tau_I from IMEM
    SWAPS R1    ;take total current
    MULS R1 ; R0-R1 <= I*e^(-1/20)*2^15
    ;dividing by 2^16 by discarding the result store in R1
    SHLN 1  ; shift R0 for
    MOVR R1 ; R1 <= total curr
    SWAPS R1 ; SR1 <= total curr
RET
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
LOAD_CURR:
    READMPV CONST_CURR_0    ;get address of const_curr
    LOADBP
    LOADSN              ; R0 <= const_curr, R1 <= Current from prev cycle
    MOVSR R1            ; SR1 <= curr for this cycle
RET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
STORE_CURR:
    READMPV CONST_CURR_0    ;get address of const_curr
    LOADBP
    LOADSN          ; R= <= const_curr, R1 <= curr from prev cycle(to update)

    MOVRS R1       ; R1 <= SR1, SR1 store updated current
    STORESP     ; store back R0 and R1 to SNRAM
RET

; **************************** PROCEDURES END ******************************
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; **************************** MAIN PROGRAMME BEGIN ************************
MAIN:
    ; Virtual operation init
    LAYERV NVL          ; Init sequencer vlayers. It is 0 for non-virtual operation
    LDALL ACC, NVL      ; Load defined virtual layers to PE array
    SPMOV 0                    ; VIRT <= ACC

    ; Initial instructions
    GOSUB RANDOM_INIT        ; For noise initialization

EXEC_LOOP:          ; Execution loop
    LOOP NVL            ; Neuron loop for virtual operation
        GOSUB LOAD_NEURON   ;loading membrane and recovery potentials
        GOSUB LOAD_CURR     ;get current from last step
        GOSUB DETECT_SPIKE  ;check if v > Vth

        SYNAPSE NLS_0     ; configuring number of synapses
        READMPV LSA0_0    ; addressing the synapses in mem
        LOADBP              ;load pointer
        LOOPV NLS_0       ; synaptic loop. Reads number of current-layer synapses
           NOP          ;to prevent pipeline error
           GOSUB SYNAPSE_CALC   ;total current stored in SR1
        ENDL

        GOSUB CURR_DECAY     ;  current exp decay
        GOSUB ADD_CONST_CURR ;  add constant input

        SWAPS R1        ; take total current from SR1
        MOVA R1         ; move to acc
        SWAPS R1        ; move to SW1

        SHRAN 1         ; divide by 2 total current for later steps
        MOVR R5         ; R5 <= current/2

        LOOP 1          ; dt = 0.5
                GOSUB MEMBRANE_POTENTIAL ; Calculate membrane potential according izhikevic
                ;GOSUB ADD_NOISE            ; Noise not added
                ADD R5      ; add curr/2
            MOVR R2      ;store back membrane pot
        ENDL
```

```
         GOSUB RECOVERY_UPDATE  ;update recovery potential

         GOSUB STORE_NEURON     ;store neuron
         GOSUB STORE_CURR       ;store the current of this time step

         MOVA R2    ; ACC <= Vmemb
         STOREB     ; used for sending Vmemb to the pc for displaying

         RST ACC        ;reset r0
         MOVR R1              ;reset r1
         INCV       ;increment virtual layer

    ENDL
    NOP
    SPKDIS        ; Distribute spikes
GOTO EXEC_LOOP ; Execution loop
```

D Python Code

Constant generator for FV approximation

```python
import numpy as np
from scipy.optimize import fsolve

def exp_func(v, gl, El, delta_t, vt, C): #exact function
    return (-gl*(v - El) + gl*delta_t*np.exp( (v-vt)/delta_t) )/C


def main():
    #constants for the neural model
    rs_gl = 10 ; rs_el = -7000 ; rs_delta_t = 200 ; rs_vt = -5000 ; rs_C = 200
    fa_gl = 12 ; fa_el = -7000 ; fa_delta_t = 200 ; fa_vt = -5000 ; fa_C = 200
    ib_gl = 18 ; ib_el = -5800 ; ib_delta_t = 200 ; ib_vt = -5000 ; ib_C = 130
    tb_gl = 10 ; tb_el = -5800 ; tb_delta_t = 200 ; tb_vt = -5000 ; tb_C = 200

    #################### regular spikign ###########################
    #get fv_root with the give set of constants
    root = fsolve(exp_func, x0 = rs_vt/2, args=(rs_gl, rs_el, rs_delta_t, rs_vt, rs_C) )
    print("rs_root = ", root[0] )

    #calculate the exact function
    x = np.array(np.linspace(rs_vt, round(root[0]), num=10000))
    print(x)
    y = exp_func(x, rs_gl, rs_el, rs_delta_t, rs_vt, rs_C)
    #get the coefficient for quadratic approx
    coefficients = np.polyfit(x, y, 2)
    a,b,c = coefficients

    #print the coefficient multiplied by 2^16 and 2^8
    print("rs_a, rs_b, rs_c = ", a, ' ', b, ' ', c)
    print("rs_a, rs_b, rs_c = ", a*2**16, ' ', b*2**8, ' ', c)
    print("rs_a, rs_b, rs_c = ", round(a*2**16), ' ', round(b*2**8), ' ', round(c))
    print("rs_a, rs_b, rs_c = ", round(a*2**16)/2**16, ' ', round(b*2**8)/2**8, ' ', c)
    print()

    #################### frequency adaptation ###########################
    root = fsolve(exp_func, x0 = fa_vt/2, args=(fa_gl, fa_el, fa_delta_t, fa_vt, fa_C) )
    print("fa_root = ", root[0] )

    x = np.array(np.linspace(rs_vt, round(root[0]), num=10000))
    y = exp_func(x, fa_gl, fa_el, fa_delta_t, fa_vt, fa_C)

    coefficients = np.polyfit(x, y, 2)

    a,b,c = coefficients

    print("fa_a, fa_b, fa_c = ", a, ' ', b, ' ', c)
    print("fa_a, fa_b, fa_c = ", a*2**16, ' ', b*2**8, ' ', c)
    print("fa_a, fa_b, fa_c = ", round(a*2**16), ' ', round(b*2**8), ' ', round(c))
    print("fa_a, fa_b, fa_c = ", round(a*2**16)/2**16, ' ', round(b*2**8)/2**8, ' ', c)
    print()

    #################### initially bursting ###########################
    root = fsolve(exp_func, x0 = ib_vt/2, args=(ib_gl, ib_el, ib_delta_t, ib_vt, ib_C) )
```

```
    print("ib_root = ", root[0] )

    x = np.array(np.linspace(rs_vt, round(root[0]), num=10000))
    y = exp_func(x, ib_gl, ib_el, ib_delta_t, ib_vt, ib_C)

    coefficients = np.polyfit(x, y, 2)

    a,b,c = coefficients

    print("ib_a, ib_b, ib_c = ", a, ' ', b, ' ', c)
    print("ib_a, ib_b, ib_c = ", a*2**16, ' ', b*2**8, ' ', c)
    print("ib_a, ib_b, ib_c = ", round(a*2**16), ' ', round(b*2**8), ' ', round(c))
    print("ib_a, ib_b, ib_c = ", round(a*2**16)/2**16, ' ', round(b*2**8)/2**8, ' ', c)
    print()

    #################### tonic bursting ###########################
    root = fsolve(exp_func, x0 = tb_vt/2, args=(tb_gl, tb_el, tb_delta_t, tb_vt, tb_C) )
    print("tb_root = ", root[0] )

    x = np.array(np.linspace(rs_vt, round(root[0]), num=10000))
    y = exp_func(x, tb_gl, tb_el, tb_delta_t, tb_vt, tb_C)

    coefficients = np.polyfit(x, y, 2)

    a,b,c = coefficients

    print("tb_a, tb_b, tb_c = ", a, ' ', b, ' ', c)
    print("tb_a, tb_b, tb_c = ", a*2**16, ' ', b*2**8, ' ', c)
    print("tb_a, tb_b, tb_c = ", round(a*2**16), ' ', round(b*2**8), ' ', round(c))
    print("tb_a, tb_b, tb_c = ", round(a*2**16)/2**16, ' ', round(b*2**8)/2**8, ' ', c)
    print()

    return

if __name__ == "__main__":
    main()
```

Netlist Display

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Crea un grafo diretto
G = nx.DiGraph()

# Aggiungi 16 neuroni come nodi
num_neurons = 16
G.add_nodes_from(range(num_neurons))

# Leggi il file delle connessioni e aggiungi gli archi con colori e direzione appropriati
with open("log/Netlist.lst", "r") as file:
    for line in file:
        neu0, neu1 = line.strip().split(", ")
        neu0, neu1 = int(neu0), int(neu1)

        # Determina il colore e la direzione in base al neurone di origine
        if neu0 < 13:
            edge_color = 'red'
            G.add_edge(neu0, neu1, color=edge_color, directed=True)  # Imposta 'directed=True' per archi direzi
        else:
            edge_color = 'blue'
            G.add_edge(neu0, neu1, color=edge_color, directed=True)

# Estrai i colori dei nodi e degli archi e la loro direzione
edge_colors = [G[u][v]['color'] for u, v in G.edges()]
node_colors = [(1, 0, 0, 0.5) if node < 13 else (0, 0, 1, 0.5) for node in G.nodes()]

#define the positions of a circular layout
radius = 2.0
pos = {}
for node in range(num_neurons):
    theta = 2 * np.pi * node / num_neurons + np.pi/2
    x = radius * np.cos(theta)
    y = radius * np.sin(theta)
    pos[node] = (x, y)

#draw the net
nx.draw(G, pos, with_labels=True, node_size=500, font_size=10, font_color="black", node_color=node_colors, edge

plt.title("Grafo della Rete Neurale con Connessioni Direzionate")
plt.axis("off")
plt.show()  #show the net
```