UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

LiveSNN: a new ecosystem

for HEENS architecture

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Josep Àngel Oltra Oltra

In partial fulfilment

of the requirements for the degree of

MASTER IN ELECTRONIC ENGINEERING

Advisor: Jordi Madrenas

Barcelona, September 2020

Title of the thesis: LiveSNN: a new ecosystem for HEENS architecture

Author: Josep Àngel Oltra

Advisor: Jordi Madrenas

Abstract

This project proposal and development of a several tools, a communication protocol and an embedded program for giving a neural network called HEENS that it is currently developed by the group ISSET from UPC the capability of being controlled remotely, and to extract the neural . This will provide a faster development, user friendly tools for the development and analysis of spiking neural networks for emulation and verification of biological neural networks and neural models.

As such, three new pieces of software are developed called: LiveSNN protocol, which communicates the HEENS architecture to a remote PC for Supervisory Control And Data Acquisition (SCADA) operations, LiveSNN program, which manages the conections and supervises the activity of the HEENS, and HEENS Toolchain Suite (HTS), which is an upgrade of the previous synthesis and assembler tools in order to allow the implementation of more sophisticated neural networks being easy and be capable of optimise the assembler model of the neuron for the architecture.

With those developments, the time to generate spiking neural networks, to debug them, and emulate them is increased in addition to the reduction of possible human errors due to the increased automation workflow that those programs give to the end user.

Thanks to all the people who helped me and make my day.

「亀の甲より年の功」

(*Kame no kou yori toshi no kou*)

(*Years know more than books*)

Japanese proverb

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecos BCN

Acknowledgements

Thanks to the help from Jordi Madrenas, who has supervised, give feedback and tested the work that has been done, Diana Mata and Josep Maria Sanchez, who give feedback on the aesthetics and the user interface of the tools, Mireya Zapata and Bernardo Vallejo who have given feedback on the new netlist and the assembler formats.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecos BCN

Revision history and approval record

Revision	Date	Purpose
0	04/05/2020	Document creation
1	18/06/2020	General format
2	27/07/2020	Content review
3	23/08/2020	Compression test

Written by:		Reviewed and approved by:	
Date	04/05/2020	Date	02/09/2020
Name	Josep Àngel Oltra Oltra	Name	Jordi Madrenas
Position	Project Author	Position	Project Supervisor

Table of contents

1. Introduction....15
1.1. Preliminary concepts....15
1.1.1. Neural Networks....15
1.1.2. Artificial Neural Networks....17
1.1.3. Spiking Neural Networks....21
1.2. Requirements....22
1.3. Objectives....22
1.4. Organization....23
2. State of the art of SNN hardware implementations....24
2.1. TrueNorth....24
2.2. SpiNNaker....25
2.3. Loihi....26
2.4. HEENS....27
2.4.1. Bus AER....27
2.4.2. Sequencer....28
2.4.3. Processing Elements (PE)....28
2.4.3.1. Synapse and Neuron RAM (SN RAM memory)....29
2.4.3.2. Spike decoder (LCL RAM memory)....29
2.4.3.3. Arithmetic Logic Unit (ALU)....31
2.4.3.4. Auxiliary peripherals....31
3. Methodology / project development....32
3.1. LiveSNN protocol....32

3.1.1. Protocol structure selection....32

3.1.1.1. Register based protocol....32

3.1.1.2. Command based protocol....33

3.1.1.3. Summary....33

3.1.2. Protocol structure....34

3.1.3. Description of the fields....34

3.1.3.1. Transfer ID....34

3.1.3.2. DataLength field....35

3.1.3.3. Command field....35

3.1.3.4. Payload field....35

3.1.4. Table of commands....36

3.2. LiveSNN program....37

3.2.1. Selection of the program architecture....37

3.2.1.1. Bare metal....37

3.2.1.2. Real Time Operating System (RTOS)....38

3.2.1.3. Linux Operating System (Petalinux)....38

3.2.1.4. Summary....38

3.2.2. Program architecture....40

3.2.2.1. FreeRTOS....41

3.2.2.2. Context handler....41

3.2.2.3. Tasks....42

3.2.2.4. Shell....45

3.3. HEENS Toolchain Suite....46

3.3.1. Netlist V1....46

3.3.2. Assembler V1....47

3.3.3. Problems....47

3.4. Proposed solution: HTS....48

3.4.1. HEENS Toolchain Suite (HTS)....48

3.4.2. HEENS Neural Synthesis (HNS)....49

3.4.2.1. Configuration section....49

3.4.2.2. Netlist section....51

It is smaller and easier to specify the value of the synapse....51

3.4.2.3. Parameters section....52

3.4.3. HEENS Code Assembler (HCA)....53

3.4.4. HEENS High Level Neural Synthesis (HLNS)....55

3.4.4.1. Configuration section....55

3.4.4.2. Netlist section....55

3.4.4.3. Parameters section....56

3.5. HTS design flow....57

3.5.1. HLNS design flow....57

3.5.1.1. Reading the netlist file....57

3.5.1.2. Seeking and load the chips....57

3.5.1.3. Place-out: map the synapses to chips....57

3.5.1.4. Generation of the specific netlists....57

3.5.2. HNS design flow....58

3.5.2.1. Reading the netlist file....58

3.5.2.2. Generation of the memory files and network summary....58

3.5.3. HCA design flow....59

3.5.3.1. Reading the code....59

3.5.3.2. Load the network summary....59

3.5.3.3. Optimization process....59

3.5.3.4. Generation of the memory files and network summary....60

4. Results....61

4.1. LiveSNN protocol....61

4.2. LiveSNN program....68

4.3. HEENS Toolchain Suite....70

5. Budget....72

6. Environment Impact....73

7. Conclusions and future development....74

7.1. Conclusions....74

7.2. Future work....74

8. Appendices....78

8.1. Protocol format structure....78

8.1.1. NULL....78

8.1.2. STATUS....79

8.1.3. CONFIGURATION....81

8.1.4. DESCRIPTOR....82

8.1.5. START....83

8.1.6. STOP....84

8.1.7. STEP....85

8.1.8. RESET....86

8.1.9. UPLOAD FIRMWARE....87

8.1.10. DOWNLOAD FIRMWARE....88

8.1.11. SPIKE REPORT....89

8.1.12. RASTER REPORT....90

8.1.13. OTHER PROTOCOL....91

8.1.14. ERROR....93

8.1.15. HEARTBEAT....94

8.1.16. GET SPIKE.....96

8.2. SEND SPIKE data format.....97

8.3. Example of the old netlist.....98

8.4. Example of the old code.....100

8.5. Example of the new high level netlist.....104

8.6. Example of the new netlist.....105

8.7. Example of the new code.....106

List of Figures

Figure 1: Neuron cell diagram form [1]....15

Figure 2: Synapse diagram between an axon and a dendrite from [14]....16

Figure 3: Structure of a simple Artificial Neural Network from [12]....17

Figure 4: Model of a neuron from [9]....19

Figure 5: Spiking neuron model drawn in inkscape....21

Figure 6: TrueNorth board from [13]....24

Figure 7: SpiNNaker IC from [4]....25

Figure 8: Loihi silicon floorplan from [10]....26

Figure 9: Block diagram of the HEENS architecture....27

Figure 10: Simple diagram of the Processing Element....28

Figure 11: Neural virtualization in the processing element....30

Figure 12: ZYNQ 7 family architecture from [11]....40

Figure 13: Task creation order, where main is the entry point of the ARMs program....42

Figure 14: HEENS Toolchain Suite (HTS) diagram....48

Figure 15: Communication between PC and ARMs via ethernet....61

Figure 16: Frame of the “Get Descriptor” command....62

Figure 17: Error response result....63

Figure 18: Frame of the “Heartbeat” command....64

Figure 19: Response of the “Heartbeat” command....65

Figure 20: Frame of the “*Get* Spikes” command....66

Figure 21: Response of the “*Get* Spikes” command....67

Figure 22: Setup of the test of the code deployed on the ARMs in the ZC706 with the console and the TELNET connections open....69

List of Tables

Table 1: Example of activation functions....19

Table 2: Comparison between protocol schemes....33

Table 3: Frame structure of the proposed protocol....34

Table 4: List of commands that will support the first revision of the protocol....36

Table 5: Comparison between the different approaches to program the ARM processor 38

Table 6: Fields of the context handler structure....41

Table 7: List of supported commands in the terminal....44

Table 8: Fields of a entry in the old netlist....46

Table 9: Entry format of the low level *configuration*....49

Table 10: Fields of the configuration section....50

Table 11: Field description of the synapse connection in the low level netlist....51

Table 12: Field description of the parameter declaration in the low level netlist....52

Table 13: Keywords reserved to the HCA....53

Table 14: Field description of the mnemonic instruction set for the HCA....54

Table 15: Field description of the synapse declaration for the high level netlist....55

Table 16: Field description for the parameters of the neurons in high level netlist....56

Table 17: Comparison between the code generated by the old and the new toolchain....70

Table 18: Relative difference of the code generated by the different toolchains....70

Table 19: Budget of the project....72

Table 20: LiveSNN NULL command request frame....78

Table 21: LiveSNN *STATUS* command request frame....79

Table 22: LiveSNN *STATUS* command response frame....79

Table 23: LiveSNN status bit format....80

Table 24: LiveSNN CONFIGURATION command request frame....81

Table 25: LiveSNN CONFIGURATION command response frame....81

Table 26: LiveSNN *DESCRIPTOR* command request frame....82

Table 27: LiveSNN *DESCRIPTOR* command response frame....82

Table 28: LiveSNN *START* command request frame....83

Table 29: LiveSNN *START* command response frame....83

Table 30: LiveSNN S*TOP* command request frame....84

Table 31: LiveSNN S*TOP* command response frame....84

Table 32: LiveSNN *STEP* command request frame....85

Table 33: LiveSNN *STEP* command response frame....85

Table 34: LiveSNN *RESET* command request frame....86

Table 35: LiveSNN *RESET* command response frame....86

Table 36: LiveSNN *UPLOAD FIRMWARE* command request frame....87

Table 37: LiveSNN *UPLOAD FIRMWARE* command response frame....87

Table 38: LiveSNN *DOWNLOAD FIRMWARE* command request frame....88

Table 39: LiveSNN *DOWNLOAD FIRMWARE* command response frame....88

Table 40: LiveSNN *SPIKE REPORT* command request frame....89

Table 41: LiveSNN *SPIKE REPORT* command response frame....89

Table 42: LiveSNN *RASTER REPORT* command request frame....90

Table 43: LiveSNN *RASTER REPORT* command response frame....90

Table 44: LiveSNN OTHER PROTOCOL command request frame....91

Table 45: LiveSNN OTHER PROTOCOL command response frame....91

Table 46: Peripheral ID definitions....92

Table 47: LiveSNN *ERROR* command response frame....93

Table 48: LiveSNN *HEARTBEAT* command request frame....94

Table 49: LiveSNN *HEARTBEAT* command response frame....94

Table 50: LiveSNN *GET SPIKE* command request frame..96

Table 51: LiveSNN *GET SPIKE* command response frame..96

Table 52: Serialization format of the spike data for LiveSNN..97

Glossary

- ALU: Arithmetic Logic Unit
- CAN: Controller Area Network
- JSON: JavaScript Object Notation
- HCA: HEENS Code Assembler
- HLNS: HEENS high Level Netlist Synthesiser
- HNS: HEENS Netlist Synthesiser
- HTC: HEENS Toolchain Compiler
- HTS: HEENS Toolchain Suite
- I2C: Inter-Integrated Circuit
- LiveSNN: Live Spiking Neural Network
- LFSR: Linear Feedback Shift Register
- LSB: Least Significant Bit
- MSB: Most Significant Bit
- NN: Neural Network
- PC: Personal Computer
- PE: Processing Element
- OS: Operating System
- OSI: Open Systems Interconnection
- RLE: Run Length Encoding
- RTOS: Real Time Operating System
- SCADA: Supervisory Control And Data Acquisition
- SNN: Spiking Neural Network
- SoC: System on Chip
- TGF: Trivial Graph Format
- WIP: Work In Progress

1. Introduction

In this chapter, a preliminary knowledge will be given as well as the current situation about the interface between neural networks and machines

1.1. Preliminary concepts

In recent years, the idea of emulating a neural system like our brains has been increasing steadily due to the growth of the processing power of the machines and the birth of the big data. In order to process this information where there are governed by complex relationships, neural networks are a good alternative.

These networks rely on the fact that can behave as systems of neurons, and could potentially behave as a brain. This, with techniques of machine learning, neural networks can 'learn' patterns and make decisions even if the case presented is new for the network.

The other important property of neural networks is that could be used as a way of brains and network of biological neurons without the need of having an actual functional brain.

1.1.1. Neural Networks

A neural network is a collection of elements called 'neurons' as it is shown in fig [1].

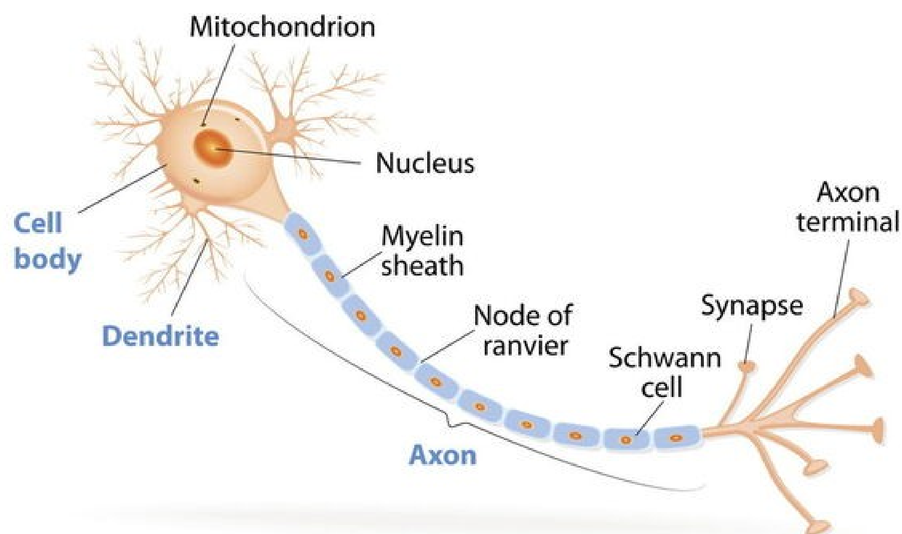


Figure 1: Neuron cell diagram form [1]

A neuron is a cell found in animals. This type of cells in charge of sending and process information in a fast and very controlled way. They use sodium and potassium ions in

order to convert the chemical signals into ion currents and electro-chemical potentials, and vice-versa.

The neurons can be divided into three main parts:

- Dendrites: transforms the chemical signals into ion currents.
- Cell body: process the incoming ion currents and generates an output.
- Axon: transports this output to the input of the other neurons and transforms the ion current into chemical signals.

In the interface between neurons and other cells where the information is passed by the chemical signals is called synapse. The operation of one neuron is as follows:

The signals are transform in ion currents. Depending on the type of the chemical used (also called neurotransmitter), the current is increased or decreased. This process is shown in fig. [2].

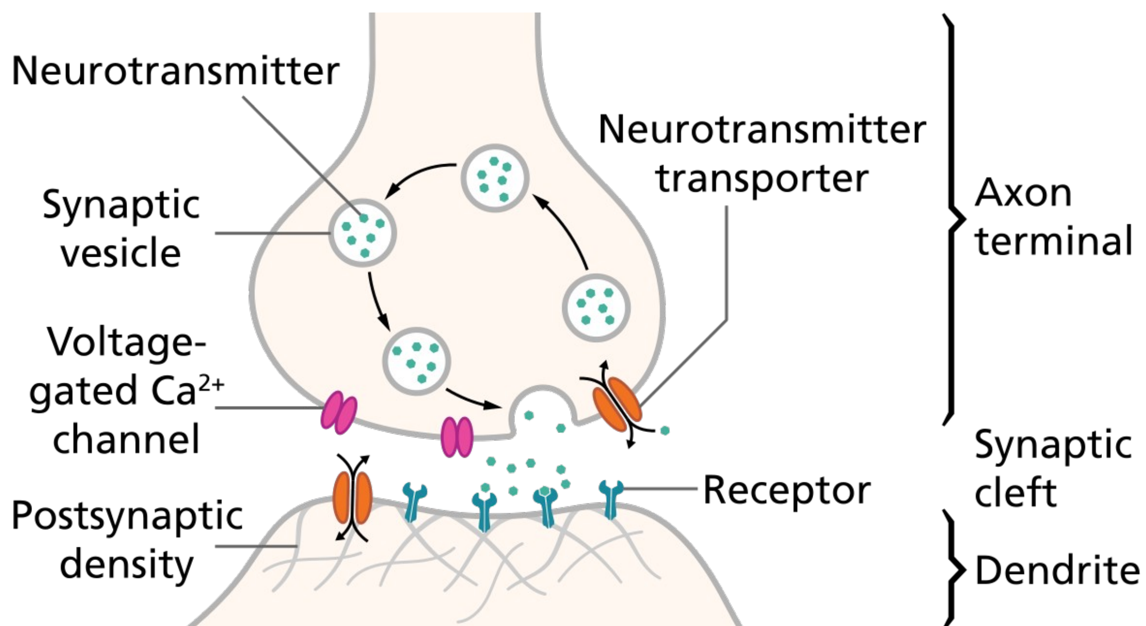


Figure 2: Synapse diagram between an axon and a dendrite from [14]

Then, all the ion currents converge in the cell body and accumulate, resulting a voltage called neuron potential. This voltage can rise until a threshold is reached, where the ion

channels start to generate a strong output signal and is propagated via to the axon. At the end of the axon, there are the synapses to the other cells.

There are several types of neural networks, but in this document the most common ones will be discussed.

The proposed system tries to copy the structure of the biological neural networks in a non biological implementation. The benefits are in computation speed and simulations of models related to neurons. As it is an artificial, it can be configured as the application needs with a high response. There are several implementations of neural networks, due to the range of applications that they try to solve. In the following points some examples are explained.

1.1.2. Artificial Neural Networks

In this type of networks, the neuron is a mathematical function that has as many inputs as a synapses, and an output that is transmitted to other neurons.

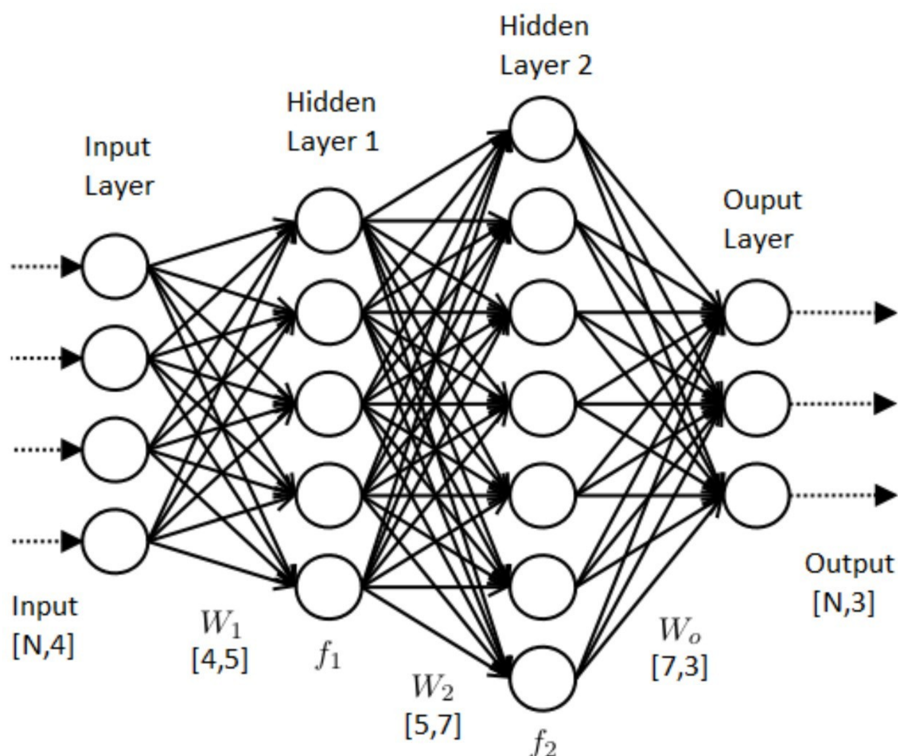


Figure 3: Structure of a simple Artificial Neural Network from [12]

Figure [3] shows an example of artificial neural network. It consists on an Input layer, whose neurons contain the input information. Then the information is passed to the hidden layers 1 and 2 for further processing. Those layers are called hidden because the neurons are internal, i.e., not connected to the input or the output, so they are connected

to other neurons. Finally the data is sent to the output neurons in order to produce the result from the neural network.

The neuron is modelled as shown in Fig. [4]:

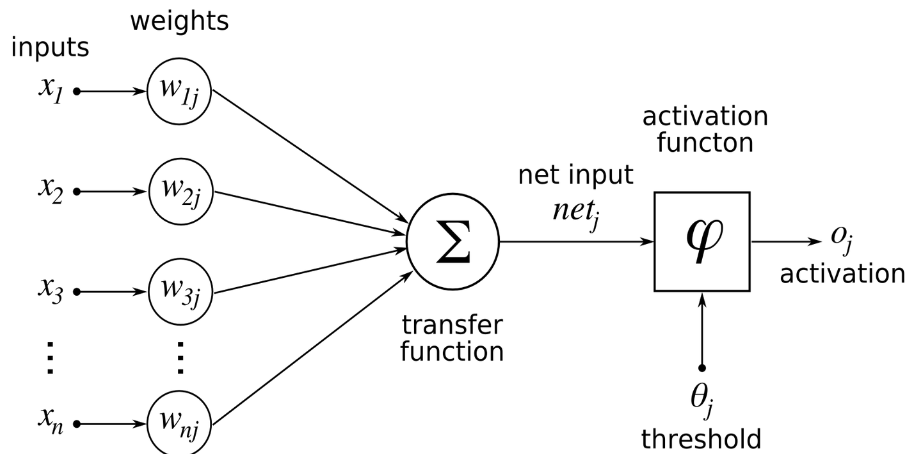


Figure 4: Model of a neuron from [9]

It consists on input connections that are multiplied by some weights, and sum all the values and passed into an activation function.

Typically, the synapses are represented as a matrix. This matrix contains all the weights for all the synapses of each neuron on a given layer. This representation is suitable to run neural networks with processors that work with matrices, like GPUs.

The activation functions that are used on the neural network are well behaved and has a very easy to compute derivative. This is important to be able to teach the network to do a task. The commonly used functions are:

Function	Expression	Derivative
Logistic	$f(x)=\frac{1}{1+e^x}$	$f'(x)=f(x)(1-f(x))$
Hyperbolic tangent	$f(x)=\tanh(x)$	$f'(x)=1-f^2(x)$
Rectified Linear Unit (ReLU)	$f(x)=max(0,ax)$	$f'(x)=\begin{cases} a, & if \quad x>0 \\ 0, & otherwise \end{cases}$

Table 1: Example of activation functions

This type of neural networks is used in all situations, and in conjunction with other types of neural networks. One example is the use on convolutional neural networks (CNN) as the last processing step on the data, as it is used as a classifier layer to sort the different features, like faces, traffic signals, that the CNN have extracted from the image.

In addition, each layer can be viewed as a vector space, and the all the weights from one layer to the next layer as a linear map. Then, the idea of neural networks is to transform the input vector space and group the vectors following a criteria, and then the output layer needs to classify the grouped values. So, for each hidden layer it is added, the neural network is more capable to sort the data and facilitates the training process.

1.1.3. Spiking Neural Networks

This type of neural network is the most similar network that emulates a biological network. It works via spikes, like the real neurons. Each neuron contains a model to transform these spikes into analog information that is used to generate the next spike.

The spike represents a neuron transmitting a signal like the biological counterpart. The neuron input spikes are accumulated increasing the membrane potential until it surpasses a threshold. Then an output spike is generated and propagated. An example is as shown in figure [5]:

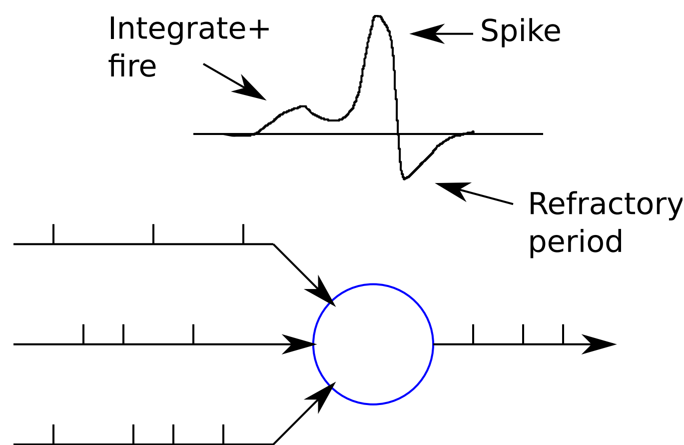


Figure 5: Spiking neuron model drawn in inkscape

The behaviour of the neuron depends on the model that it is loaded into each neuron. For example, a simple model is the “Integrate and Fire” model, where the spikes add or subtract a given amount of potential until it reaches the threshold potential. Then a spike is generated and transmitted to the neural network.

This type of neural network is the most similar to a biological one, where the information is encoded in a impulse-frequency basis, mimicking the biological counterpart. The utility of those types of neural networks are for research and emulation of biological systems, due to their similarities, but recently, there have been several attempts to use them in real-world applications.

One example of implementation of this type of neural network is the HEENS architecture, which it is being developed at the Universitat Politècnica de Catalunya.

1.2. Requirements

The HEENS architecture is divided into several subprojects and tools to generate the required files to do a simulation. So the requirements to mitigate time consuming efforts are:

- A protocol that needs to be able to exchange information from and to the HEENS
- A software that can run in embedded processors to control the HEENS
- Back compatibility as much as possible with the older tools.

1.3. Objectives

This project tries to accelerate and develop the infrastructure required to operate the HEENS in a more user friendly, yet with a high performance. So the objectives are:

- Develop a communication protocol that will transmit information between the HEENS and the PC via remote access, with a modular structure for quick upgrades that will be appearing.
- Generate a program that take advantage of the ZYNQ Z706 embedded ARM processor units to be able to communicate with the protocol and the HEENS, with analytic capabilities and remote control of the architecture, that in the future will be able to upgrade the system.
- Enhance existing tools to be modular, efficient and easy to use, without reducing the existing capabilities. In addition, automate and optimise (when enabled) the code to be executed. Those tools will be able to be cross -platform.

1.4. Organization

This project is divided in three main sections:

1. Define the protocol scheme, which will be set some requirements on the implementation of the code in the ARMs, and specifically the way of handling the messages.

2. Implement the code on the ARMs, where most of the time will be consumed by the debugging and validating the code.

3. Generate the second generation of the toolchain, that will have the same procedure as the point 2.

2. State of the art of SNN hardware implementations

Nowadays, there are different implementations done by different entities like IBM, the university of Manchester,etc. Those implementations are have it's own characteristics in order to test different approaches to how to create a SNN in silicon. So, among all the different implementations, four of them will be explored:

- TrueNorth from DARPA and IBM
- SpiNNaker from University of Manchester
- Loihi from Intel
- HEENS from Universitat Politècnica de Catalunya

2.1. TrueNorth

This neural network is designed by DARPA[2] and IBM[3]. It is a chip with a 4096 cores with a total neuron emulation of 1 million neurons and 256 million synapses. It claims to be more efficient and faster due to they use a custom architecture whereas the traditional Von Neumann architecture. A picture of the TrueNorth is shown in the Fig. [6].

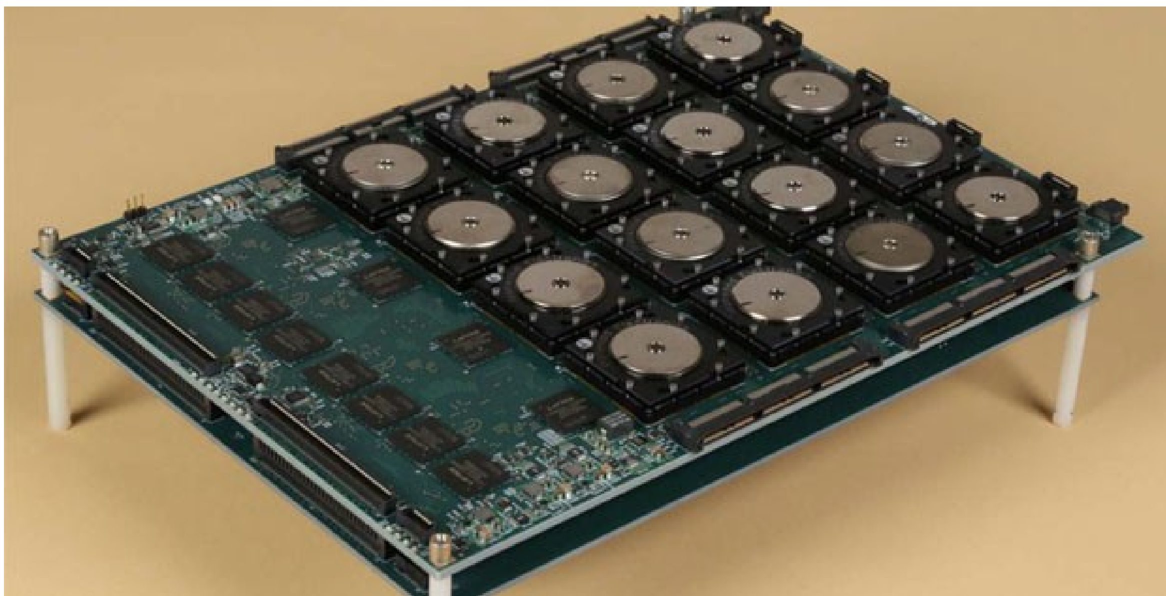


Figure 6: TrueNorth board from [13]

2.2. SpiNNaker

This neural network is designed by School of Computer Science at the University of Manchester in UK[4]. It is build in a chip and it has 16 cores to preform the execution on the neural network, 1 for task management, and 1 for fault or backup processor, giving a total of 18 cores. This type of processing emulates in software the neurons in the spiking neural network. It has been scaled up a large SNN unit with 518.400 SpiNNaker processors. The layout of the SpiNNaker is shown in the Fig. [7].

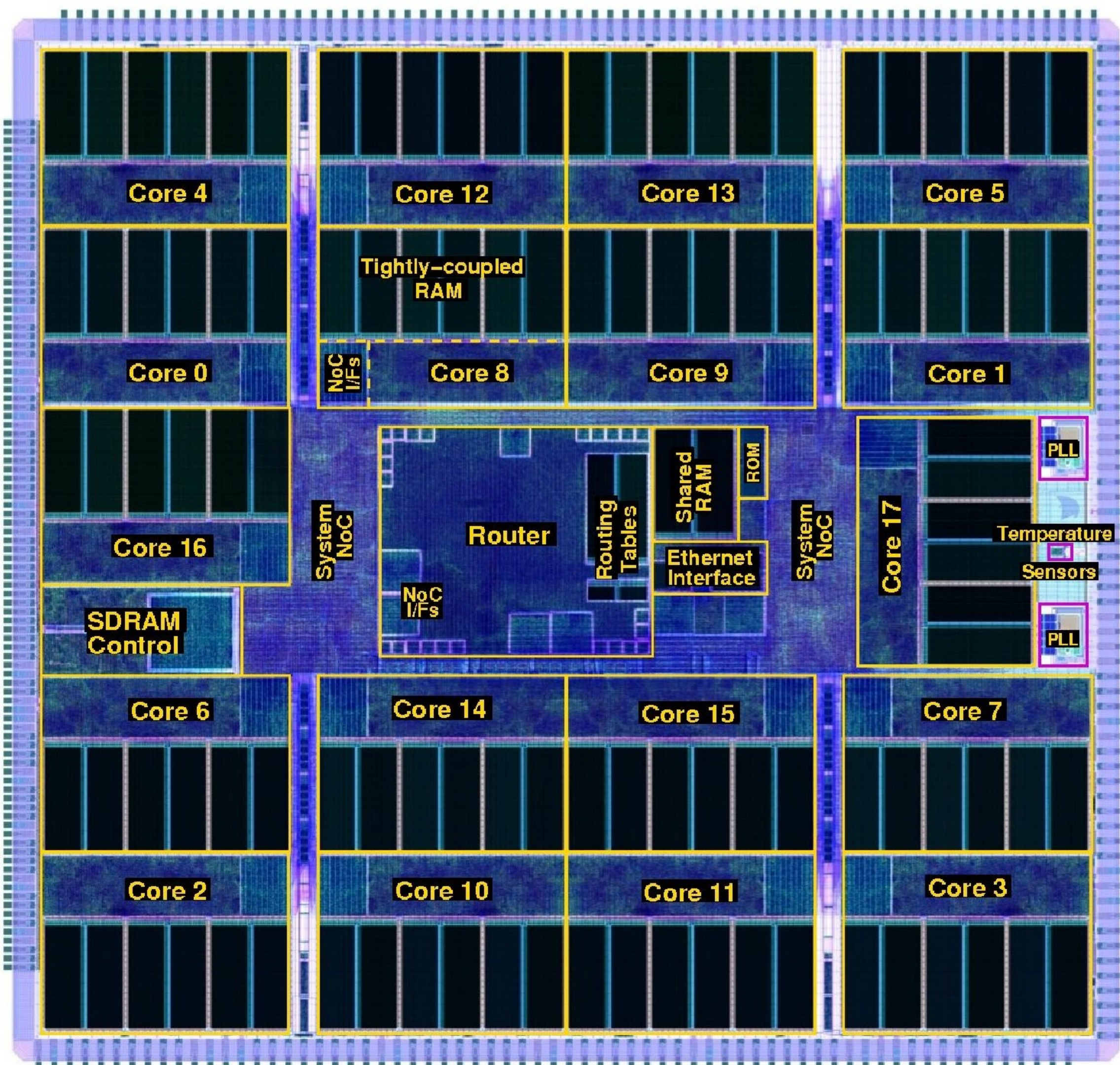


Figure 7: SpiNNaker IC from [4]

2.3. Loihi

This neural network is designed by Intel. It is a custom neuromorphic core with the learning process included in the core itself[5]. It is a 128 neuromorphic cores with three x86 processors in order to monitor and configure the neural network. It is divided into bundles of spikes, that then are passed through a tree of neurons. This makes this neural network to have a reduce interconnect between neurons, and a careful planning is needed in order to take advantage of this architecture. A layout of the Loihi is shown in the Fig. [8].

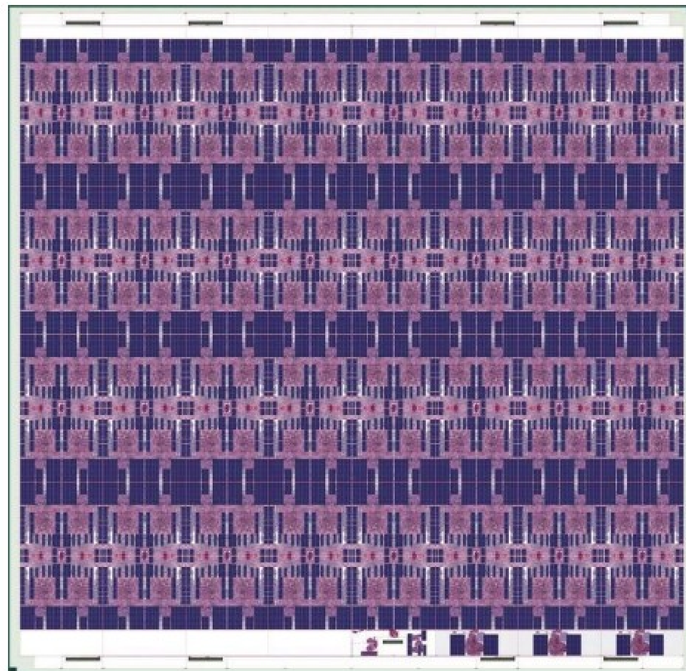


Figure 8: Loihi silicon floorplan from [10]

2.4. HEENS

As the world of neural networks advances, there are necessities in the development on more sophisticated architectures. In the niche of the spiking neural networks, the main developments are on software and hardware implementations for quick verification and analysis of the biological neural networks and neurons. The HEENS architecture is a hardware implementation of a spiking neural network. It is deployed on System on Chip of by the Xilinx manufacturer.

HEENS is a spiking neural network implemented by the ISSET research group of the UPC. It is implemented in VHDL and can be deployed in FPGAs and simulators like QuestaSim. This neural network has to work properly a binary file that contains the neuron model to be executed and the binary files containing the synapses. It is divided into 2 main blocks called “sequencer” and “Processing Element array” (PE array). A block diagram is shown in Fig. [9].

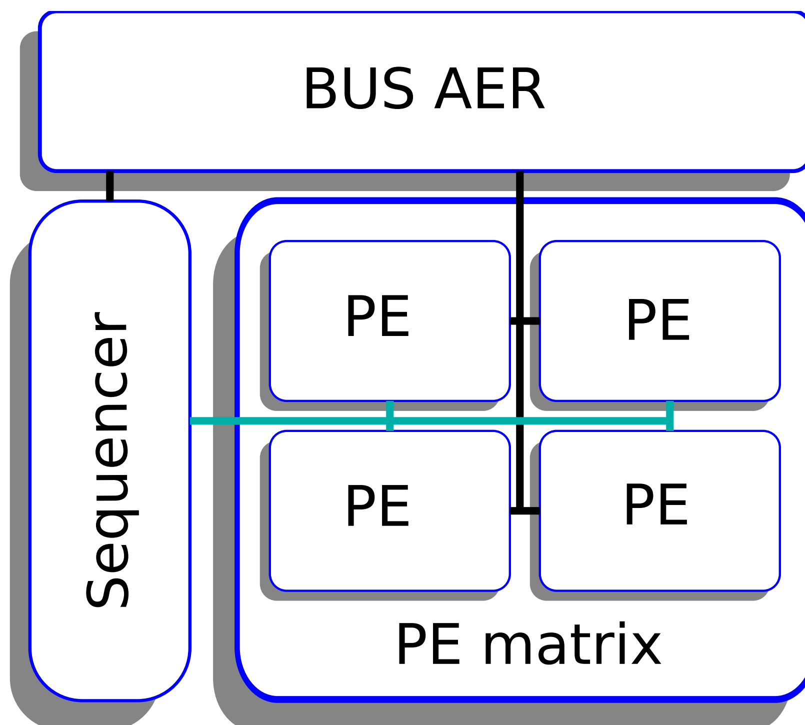


Figure 9: Block diagram of the HEENS architecture

2.4.1. Bus AER

This bus is used to interconnect different boards that implement the HEENS architecture. As such, it moves the spike information between boards. These spikes are called global spikes, and uses the ID number of the HEENS to sort these spikes.

2.4.2. Sequencer

The sequencer is the part that executes the neuron model and controls in the PEs. It consists on a control unit, with a memory that contains the instructions of the model and several register and memory cells to implement several functions, such as:

This bus is used to interconnect different boards that implement the HEENS architecture. As such, it moves the spike information between boards. These spikes are called global spikes, and uses the ID number of the board to sort these spikes.

- Counter to have value about the current iteration of a loop
- Stack LIFO memory for storing the stack pointers
- Instruction memory witch contains the model and common data values that are shared among all the PEs.

Also, it contains a bus that connects the sequencer to the PE array, which it enables to share the same information with all the PEs. In addition to that, the sequencer contains a pipeline which increases the performance of the neural network.

2.4.3. Processing Elements (PE)

The Processing Element (PE) is the distributed part of the HEENS. It consist on a dedicated Arithmetic Logic Unit (ALU) to execute the spiking model, and a routing logic in order to decode the spikes form the network. A block diagram of the PEis shown in Fig. [10]:

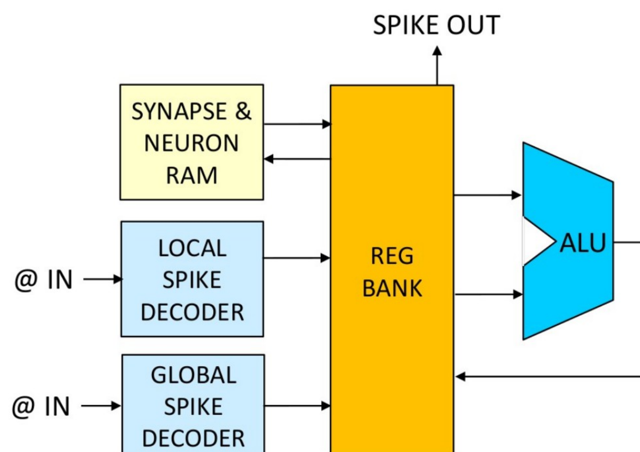


Figure 10: Simple diagram of the Processing Element

2.4.3.1. Synapse and Neuron RAM (SN RAM memory)

This memory is used to store the synaptic and neuronal parameters. It is used as a distributed memory for the PE, which can read specific data related to the instructions in the sequencer. It contains information like synaptical weight, initial seeds for the pseudo random generators, etc.

2.4.3.2. Spike decoder (LCL RAM memory)

The spike decoder is a circuit that decodes the incoming spike into an address that the model can access. This is implemented via memories.

The PEs are arranged into a matrix like format, where it contains r rows and c columns. As the memories are normally bigger than the PEs, virtualization layers are added in order to increase the effective number of neurons. So, the neurons with a given row and column are processed in the same PE, in order of the virtual layer number assigned to them. With this, the effective number of neurons are:

$$N = R \cdot C \cdot V \tag{1}$$

Where:

- R is the number of rows of the PE array.
- C is the number of columns of the PE array.
- V is the number of virtual layers that a PE can execute.

This can be viewed graphically as shown in Fig. [11]:

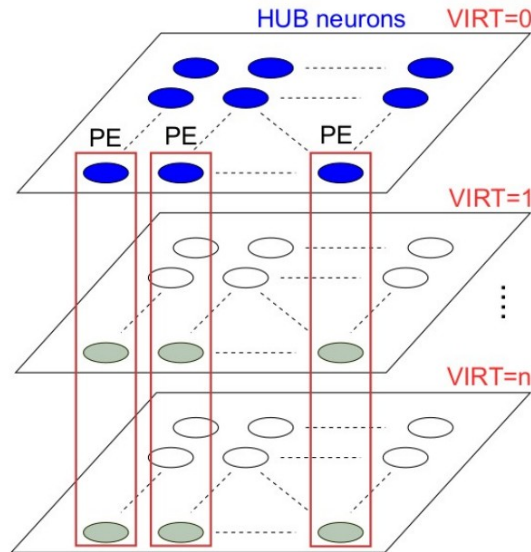


Figure 11: Neural virtualization in the processing element

So, with this, if the SNN is desired to be increased, more area can be dedicated to the PEs to increase the number of PEs, and more time can be spent in order to execute more neurons per PE.

The incoming spike is a number that codifies the spike into 3 parts:

- The virtual layer of the destination neuron.
- The column on the PE array of the destination neuron.
- The row on the PE array of the destination neuron.

This codification as a number and as a an array gives the connexion to the address to store the spike into the memory. If an spike has been produced, it will be stored in the destination PE array memory, in the following address A:

$$A = c + R \cdot (r + V \cdot v) \quad (2)$$

For spikes that come from external sources, like other chips with the HEENS, there is a dedicated memory to translate the id from the current board alongside the spike information in order to route the spike in the correct destination neuron.

This information is stored in the LCL-RAM memory, where the synapse number is stored in the address corresponding to the position of the source neuron.

2.4.3.3. Arithmetic Logic Unit (ALU)

The ALU is a circuit that can the arithmetic and logic operations. It calculates 16-bit operations as it has good trade-off between area and precision.

2.4.3.4. Auxiliary peripherals

The PE also contains two register banks of 8 registers each one for intermediate variable storage and modules that help to accelerate certain operations as well as generating useful patterns.

The pattern generator is a Linear Feedback Shift Register (LFSR). This is a series of flip-flops that are connected in a daisy chain configuration with some linear operation with the previous state. Moreover, it has 16 registers divided into 2 banks. This is useful in situations like storing the state when a function is called.

3. Methodology / project development

This project is called Live Spiking Neural Network (LiveSNN). It is called that way because it will give the HEENS SNN the capability of remote access for SCADA, the way it will communicate with the embedded ARM processors and the code that will be hable to understand this new protocol. It will has three phases:

- Develop of the LiveSNN protocol
- Develop the programs running on the ARM's processors (LiveSNN program)
- Develop advanced toolchain suites for the HEENS architecture

3.1. LiveSNN protocol

The neural network needs to communicate with a remote computer to supervise and control the activity of the HEENS processor. As it is a neural network processor, the protocol needs to be designed into the needs of the HEENS, as well as allowing bridge communication between the ARM processor and another electronic devices like sensors, memories, etc, for quick diagnosis and full supervision of the system.

3.1.1. Protocol structure selection

There are different ways to implement the LiveSNN protocol.

3.1.1.1. Register based protocol

This type of protocol scheme uses write and read memory in order to transmit information. It consist on the following sections:

- Address to be read from or written to.
- Read/Write identifier (control bit).
- Data stream

It is very easy to implement in hardware and software, and usually the control bit is combined with another field, like an identification number, in order to have more devices sharing the same bus. This also requires almost no logic at all and it is pretty fast due to it's simplicity.

A few examples of buses using this type of commands are Controller Area Network (CAN), Inter-Integrated Circuit (I2C), etc.

3.1.1.2. Command based protocol

This type of protocols has the flexibility to convey actions in a part of the frame. These actions are usually called commands. Each command executes the action coded in the command bytes will the help of the optional arguments. This achieves a good development time thanks to executing different things in separate messages with the penalty of having extra logic to decode the frames.

This type of protocol is a superset of the Register based one, due to the fact that it is possible to implement the commands that the Register based protocol have.

3.1.1.3. Summary

The protocol will have an identifier field in order to be able to distinguish among the different neural networks. If c is the number of bytes that the command field has, the following table is build:

	Register based	Command based
Frame overhead	2	2
Number of commands	2	256^c
Programming effort	Simple	Complex

Table 2: Comparison between protocol schemes

So, the format chosen is the command based, which will encode different actions in a small, compact form.

3.1.2. Protocol structure

This protocol needs to interact with the SNN in different ways, in a command-based protocol. It has different commands to manage the several functions in an efficient way. All the fields are stored in big endian (the most significant byte is the first, whereas the least significant byte is last). The message contains metadata in order to send the information. The messages have the following format:

Byte	Name	Description
0	TransferID	Identification of the frame
1 - 4	DataLength	Length of the frame
5	Command	Action to be executed
6 - (n-1)	Payload	Data related to the command

Table 3: Frame structure of the proposed protocol

An error detection algorithm is not included because this protocol will be in the session layer in the Open System Interconnection (OSI) model. So, all the problems relating with data routing, and the data integrity are handled by the other layers. In case of the implementation on the ZYNQ SoC, the LiveSNN protocol will use TCI/IP ethernet communication.

The commands are explained in Annex 8.1.

3.1.3. Description of the fields

In this section, the different fields that the protocol implements are explained.

3.1.3.1. Transfer ID

It is a message identifier in order to know the arrival of the messages. It is also used as a error checking due to the fact that this number always increments by one (until it overflows). As such, if the transferID does not coincide with the next one, an error is raised. In case that the number is bigger than 255, it simply wraps around and continues from 0. In case of using TCP/IP it is redundant.

3.1.3.2. DataLength field

This field holds the length of the whole frame, which it is used to pre-allocate a buffer for storing the frame. It is counted in bytes, and the possible valid range is between 6 and $2^{32} – 1$ bytes. If this value is less than 6, an error is raised as a bad frame has been read. The maximum value is the theoretical upper limit, and depending on the physical layers the real upper limit will be much lower.

3.1.3.3. Command field

This byte encodes the action to be executed to the neural network. Depending of the value, the ARM processor needs to perform an action on the HEENS processor, or perform an action on itself, like rebooting, or just forward the payload into another physical media in order to interact with the environment and get information related to the system. Those actions are classified in 128 values. The MSB is reserved for the protocol, and it shows if the message has been processed. This makes the message self contain, where within itself the protocol knows in every moment if a message is a request (MSB = 0) or a response (MSB = 1).

3.1.3.4. Payload field

Those bytes contain the information carried by the protocol specified by the command. It could be 0 or the entire firmware of the system running on the HEENS

3.1.4. Table of commands

The commands are encoded in one byte, so as mention before, the range is 256 different commands. As the MSB is for detecting if the frame is a request or a response, the effective number of commands are 128. So, the following commands are proposed in order to have control, with room for implementing extra functionalities.

Value	Command	Description
0x00	NULL	Empty message
0x01	STATUS	Sends the current status
0x02	CONFIGURATION	Sends the current configuration
0x03	DESCRIPTOR	Sends the descriptor
0x04	START	Starts the Neural Network (NN)
0x05	STOP	Stops the NN
0x06	STEP	Executes n steps of the NN
0x07	RESET	Resets the NN
0x08	UPLOAD FIRMWARE	Uploads a new firmware to the NN
0x09	DOWNLOAD FIRMWARE	Downloads the current firmware of the NN
0x0A	SPIKE REPORT	Downloads the spike report of the NN
0x0B	RASTER REPORT	Downloads the raster report of the NN
0x0C	OTHER PROTOCOL	Send a communication frame to the ports
0x0D	ERROR	An error has occurred
0x0E	HEARTBEAT	Sends a heart beat signal to test the com.
0x0F	GET SPIKE	Gets the new spike that has been generated
0x10 - 0x7F	Reserved	Reserved for future revisions

Table 4: List of commands that will support the first revision of the protocol

3.2. LiveSNN program

The ZYNQ is a family of System On Chip (SoC) where it contains 2 ARM processor units with a high preformance FPGA, where is capable of designing hardware accelerators to help the ARMS and hardware designs where the ARMS can be used as a monitor and control for the hardware.

As explained before, the HEENS in the ZYNQ has several ARM Cortex processors. With those cores, a diagnostic and SCADA control could be implemented in order to assist the neural network. This logic needs to solve several challenges due to the necessities of the architecture. Those needs are:

- Monitor the HEENS processors
- Update memory banks of the HEENS architecture for updates, debugging, etc
- Supervise and control the architecture
- Bring connectivity from/to the HEENS to a remote computer

As such, and to maintain compatibility between the ZYNQ and the ZYNQ ultrascale+, two cores are used to implement the software.

3.2.1. Selection of the program architecture

The ZYNQ can be programmed in different modes: Linux OS, Real Time OS and bare metal. Each framework have its own characteristics

3.2.1.1. Bare metal

This framework is the most simple one. It is programmed on C, C++ and assembler. In this framework, the code is deterministic because the program follows a sequential flow. It also may contain interrupts, where the sequential execution is broken in order to generate a response from a external signal, that is usually is high priority. For example, in Real Time Operating Systems uses a internal timer called SysTick where it triggers an interruption when certain time has passed in order to execute the scheluder, a piece of code that manages the current active tasks.

3.2.1.2. Real Time Operating System (RTOS)

This type of operating system is focused on having two main characteristics:

- Give processor the capability to execute multiple threads with an scheluder.
- The threads need to be deterministic in time, in order to ensure that the critical tasks are being executed in the required time slots.

One example of this type of RTOS is FreeRTOS[6]. This RTOS is very small, optimized for embedded projects. Some key aspects of this is that it is free and open source, giving this RTOS an advantage among the rest with the huge forum and documentation pages.

3.2.1.3. Linux Operating System (Petalinux)

This type of OS is focused on having the same user experience as a normal Linux distro as much as possible taking into account the limitations of the ZYNQ SoC. Petalinux[7] is the OS that the company Xilinx have adapted to their products.

3.2.1.4. Summary

By examining the characteristics of each solution for the code in the ARM, the following table is implemented:

Parameter	Petalinux	FreeRTOS	Baremetal
Memory overhead	Biggest	Medium	None
Time overhead	Biggest	Small	None
Developing time	Shortest	Normal	Longest
Code already imp. (libs)	Most	Some	Little
Memory and Execution time determinism	None	Controlled	Yes
Speed execution	Slowest	Normal	Fastest
Programming languages	All	C++/C/ASM	C++/C/ASM

Table 5: Comparison between the different approaches to program the ARM processor

For the characteristics of this project, the FreeRTOS is chosen. It is true that Petalinux has a lot of things readily available, but it uses both cores. Also, it has the biggest penalty on the memory and time overhead, due to all the code that it is being executed. The ZYNQ SoC has two ARM cores, so the second one will be dedicated to the HEENS SNN in order to monitor and transfer information between the core 0 and core 1. As the code needs to be fast and it needs to be fully deterministic, the baremetal framework is chosen.

3.2.2. Program architecture

As the ZYNQ has two cores, the tasks will be split by role. The different tasks between the control of the HEENS and the communications, giving an extra layer of security.

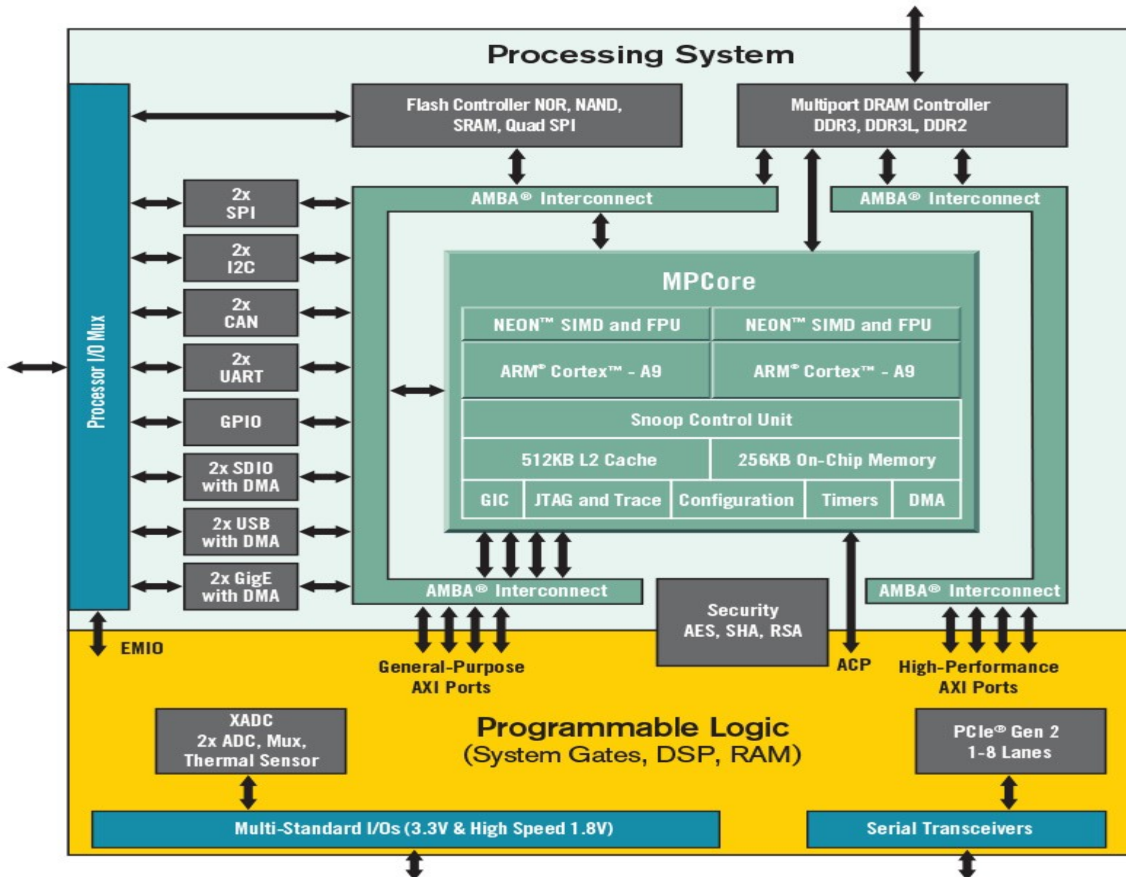


Figure 12: ZYNQ 7 family architecture from [11]

The Fig. [12] shows the diagram of the ZYNQ 7 family with the two ARM cortex A9.

The master core is in charge of the communications. As mentioned before, this core will be implementing the FreeRTOS as working framework. Also, as mentioned before, the slave core is in charge to supervise and control the neural network. As such, it will be programmed in baremetal framework, where there is no operating system nor any compatibility layer between the code and the core itself. This splits the control of the neural network by using specific methods and registers, giving the extra security mentioned before.

With this, the software architecture to be developed follows an asymmetric architecture.

3.2.2.1. FreeRTOS

As indicated before, the RTOS used is FreeRTOS[6] due to characteristics. It is a small, fully featured, low level and free RTOS. The communication program doesn't need a lot of features due to the low level processing that needs to be done. As such, the only RTOS elements that will be used are mutex, semaphores and tasks to be able to give service to the different communication protocols that will be handled.

3.2.2.2. Context handler

The context handler is a data structure with pointers to send and receive methods that is commonly used for the different tasks of the main core. The important thing of this structure is that it generalizes the communication ports for STD_IN and STD_OUT for code reuse, as the context handler has the information of where the information is coming from and where is leaving to. This produces a layer of isolation between two tasks that communicates to the external world.

A clear example is the shell class. It contains the methods to use a basic shell in some communication port. The TELNET and the serial port uses this class. So, without the context handler, the information is shared among all the instances, and the information of the TELNET session leaks into the serial. That means that the things that are written in the TELNET is also shown in the serial. The problem is also valid in the other way around. The serial can leak information to the TELNET shell.

With context handler, the output and input functions for the data are defined in the context, so the input and output comes with the context, so there is no data leak from telnet to the serial port, because to do that the data must go from one context handler to the other, and this is very difficult, due to the fact that the context handlers are used and instantiated independently in different classes.

Name	Type	Describe
word	string	Stores a word from the command stored in str
str	string	Stores the command that has been received
lastStr	string	Stores the previous command that has been executed
ix	uint16_t	Current position of the word in str
handler	int32_t	ID for the send and receive methods
Sendfnc	sendfn_t	Send function in order to send the responses.

Table 6: Fields of the context handler structure

3.2.2.3. Tasks

The program needs to handle several scenarios. It is divided in several tasks with the mission to handle each scenario. In addition, a logging system is developed in order to monitor the state of the tasks form the main core and shared to the slave core. With this approach, the programmer will have full state information with only listening one core.

In order to have uniformity between the different tasks, a set of interfaces has been developed in order to keep at minimum code duplication and generalize and add layers of abstraction.

The ITask interface has the following methods to be implemented:

- InitTask(void *p): This function will be called when the task is started
- Task(void *p): This function will be called periodically. If the function returns a number different to 0, the task will be finished.
- EndTask(void *p): This function will be called when the task is finished

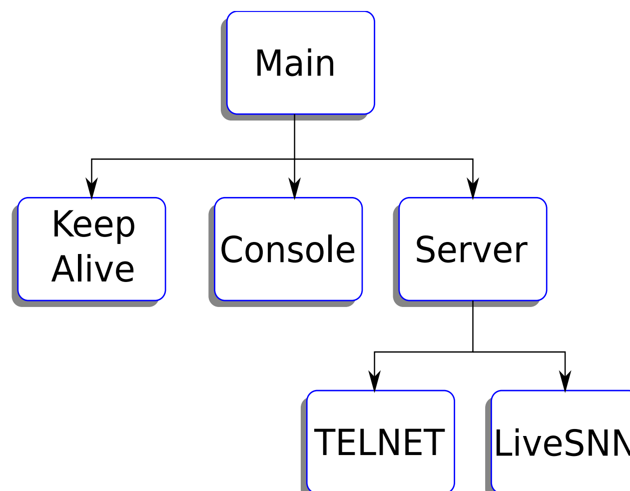


Figure 13: Task creation order, where main is the entry point of the ARMs program

As shown in the Fig. [13], the main block is the entry point of the program, and it creates three different tasks:

- The KeepAlive task, where checks the integrity of the ARM processors
- The Console task: which manages the serial console for debugging
- The Server task: where manages the ethernet connections and spawns the tasks Telnet and LiveSNN tasks to manage the connections from the different ports.

3.2.2.3.1. KeepAlive

This task is in charge of monitoring the SoC and make sure that the cores and the HEENS architecture do not stall or have any problem that will lead to the stall of the system. In order to address this problem, the ARMs watchdog is enabled and giving the running signal for the cores and checks the HEENS state if the sequencer has stopped. In addition, for visual inspection, a blinking led is programmed.

In case of partial stall of a core, the watchdog resets the stalled core and notifies the system into recovery mode, where the other core helps to configure the stalled core. In case of the sequencer is stalled, the slave core starts to get the sequencer info and executes a sequencer reset. Then it loads the contents of the sequencer to resume normal operation.

If any error occurs, it is logged into the terminal and in a file in a non volatile storage system like an SD card or USB for further diagnostics.

3.2.2.3.2. Console

The mission of this task is to have a low level interaction level between the ZYNQ with a PC via a serial connection. It manages the commands described in the shell section. It also outputs the log data in the command to be able to follow the program execution and have basic diagnostic tools in real time.

The implemented commands are:

Command	Name	Description
deviceInfo	Device Information	Gets the device descriptor
gsd	Generate Spike Data	Gets the spike data buffer and decodes it
help	Help	Prints all the available commands
ipconf	IP Configuration	Gets the IP configuration
rb	Read Byte	Read 1 byte in the RAM memory of the ARMs
reboot	Reboot	Performs a full system reset
rm	Read Memory	Read memory with the specified length
tasks	Tasks	Print the Task status
tasksInfo	Tasks Information	Show current task info
ver	Version	Print software version
wm	Write Memory	Write memory
ww	Write Word	Write 1 word in the memory

Table 7: List of supported commands in the terminal

The only commands that require arguments are the read and write operations.

- For the read, the address and then the length (only for the rm) is required.
- For the write is the same as the read, with the last argument being the data to be written.

3.2.2.3.3. Server

This task is the most complex of all. It is in charge of setting up the Ethernet stack, manage the connections and sort them between the different ports. In addition, it spawns the TELNET and LiveSNN tasks when a successful connection is detected. As the current configuration, only one connection per port is allowed.

The Ethernet stack is provided by the Lightweight Internet Protocol (LwIP) library provided by Xilinx, but minor bugs have been detected that do not compromise the execution of the task.

3.2.2.3.4. TELNET

This task allows the remote access of the console via TELNET protocol. So, the access of the console does not need to be in physical proximity, the only thing is to have Ethernet connection to the SoC. The only drawback of this method is the startup sequence of the processors is not shown due to the Ethernet initialization procedure. The used port is 23.

3.2.2.3.5. LiveSNN

This task manages all the LiveSNN protocol. It has the protocol built in and generalized for any physical media. As such, with the proper context instance, the protocol could be used. The used port is the 8080.

3.2.2.4. Shell

For an easy debug of the cores and the neural network, a lightweight shell is programmed in the software. This shell gives low level control for the ARMs and the neural network.

This shell is context-free, so multiple users can access the shells and work without any information being leaked. The tasks automatically use a mutex in order to protect the different commands that the different users could send.

3.3. HEENS Toolchain Suite

In this chapter, the toolchain developed the HEENS architecture is explained. In the previous version, a common python file contains the properties of the HEENS spiking processor, a assembly code which contains the neuron model, a netlist file containing the synapses in a grid based identification and a CSV with the neural parameters.

3.3.1. Netlist V1

The previous version of the netlist synthesis reads two input files, a netlist file and a memory CSV data descriptor.

The netlist contains every synapse of the network. Each row in the netlist is a synapse entry. It specifies everything that is related to that synapse. It is grouped into 4 groups, arranged in a continuous line separated by commas, like an CSV file:

0, 1, 1, 1, 1, 0, 1, 1, 1, 4131127296

1, 0, 1, 1, 0, 2, 0, 1, 2, 4131127296

Each entry has the following format:

Type	Position	Format	Example
Source neuron	0 - 3	i, v, r, c	1, 0, 1, 1
Dest. neuron	4 - 7	i, v, r, c	0, 2, 0, 1
Synapse number	8	s	2
R0 and R1 data	9	data	4131127296

Table 8: Fields of a entry in the old netlist

The output of the netlisting process takes the synapses and generates the data for the LCL-RAM memories, which preform the input spike decoding.

The other file used in the netlist is a CSV file, where is a memory dump to be loaded in the SN-RAM memories, that contain the local synaptic and neural parameters.

A file example is shown in Annex 8.3.

3.3.2. Assembler V1

The assembler used only compiles the code with a given instruction set and generates the compiled version. This process is done in a two step process, where the first step detects the labels, and the second step translates the instruction. In addition, it uses the parameter file to configure the assembler to work properly with the current configuration. The instruction set is described in a CSV file, which it allows to add, modify and remove the instructions in a fast way.

NOP,0b000000,0

LDALL,0b000001,1

A file example is shown in Annex 8.4.

3.3.3. Problems

The problem arises on the description of the neural network, that as in this process has as an input 3 different files. Those files contain information that depends on the others. In addition, the assembler code also needs the information of the network in order to operate correctly. So, depending on the application, if a modification is desired, it is probable that a parameter in one of the files has not been updated.

In addition, the debug of the code and/or the network is obfuscated by the fragmented information of the several files used.

3.4. Proposed solution: HTS

The problem is the definition of the assembly code and the netlist. The proposal is a redefinition of both files in order to synthesize the same information in a less, more general format. With this, tree toolchains have been develop: the HEENS Code Assembler (HCA), HEENS Neural Synthesis (HNS) and the HEENS high Level Neural Synthesis (HLNS). All the files support comments via the token '\#' and numbering is supported for negative and positive integers, with base 10 and base 16, with the prefix '0x'.

3.4.1. HEENS Toolchain Suite (HTS)

This tool is the basic compiler for the HEENS architecture. As input, a netlist descriptor and an assembly code of the model is used. Then, it generates all the memory information for the system. Keywords are used in the assembly to determine all the data related to the network. The toolset organization is shown in Fig. [14].

The HTS process starts with the HEENS Neural Synthesis (HNS), which takes the netlist of the neural network to be synthetized and generates the memory dumps in order to be able to simulate. Then generates tcl scripts to load this information into the vivado project and finally generates a summary for the HEENS Code Assembler (HCA) specifying the necessary info to compile the assembly code into a binary file.

The HEENS High Level Netlist Synthesis (HLNS) is an add-on feature for high-level synthesis that will be explained later.

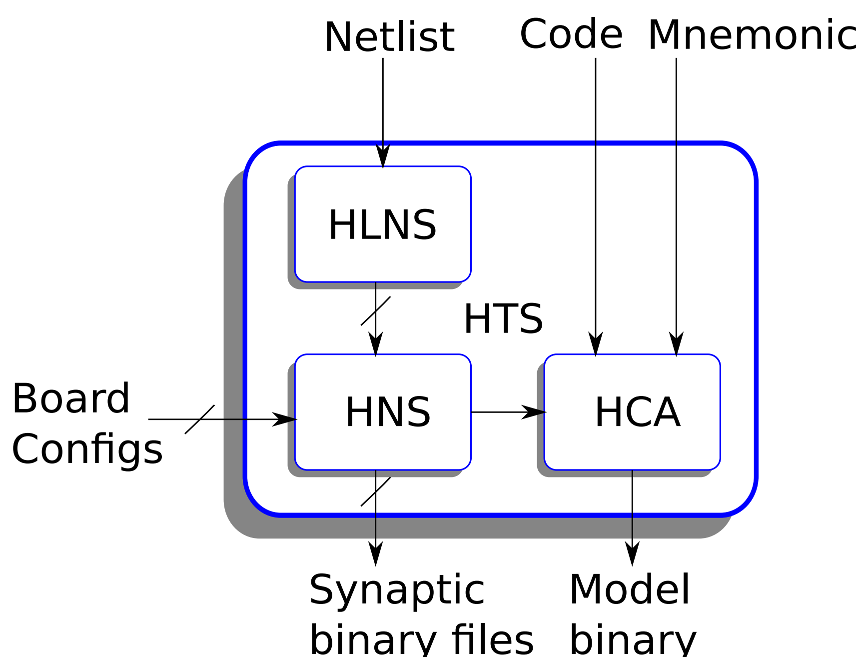


Figure 14: HEENS Toolchain Suite (HTS) diagram

3.4.2. HEENS Neural Synthesis (HNS)

The new proposed format for the netlisting has the same information as the three files. As such, a marker is introduced to differentiate the parts. It is divided into 3 main sections:

- @Config: several rows with the characteristics of the board that is being synthesized with.
- @Netlist: the netlist of the network, with similar structure as the previous version.
- @Params: the parameters that will be inserted into the SN-RAM, with meta information for the HCA tool.

Each section contains the related data in each row. Those rows are called entries.

3.4.2.1. Configuration section

This section of the netlist defines the parameters of the architecture version for the neural network to be deployed. It specifies all memories and registers. Each entry has the following format:

Type	Example
Variable name	ID
Separator	=
Variable data	2

Table 9: Entry format of the low level configuration

With this format, it is easy to check and edit the configuration of the architecture to be deployed. All the parameters needed to be specified are:

Field	Example	Description
Id	Id = 0	Id of the chip to be synthesized
Rows	Rows = 4	Number of rows of the PE
Cols	Cols = 4	Number of columns of the PE
VLay	VLay = 8	Number of virtual layers
RegS	RegS = 16	Register size of the PE in bits
RegN	RegN = 8	Number of registers in the PE
SneS	SneS = 32	Size of the SN-RAM in bits
SneA	SneA = 1024	Address depth of the SN-RAM
LclS	LclS = 7	Size of the LCL-RAM in bits
LclA	LclA = 128	Address depth of the LCL-RAM
InsS	InsS = 16	Size of the INS-RAM in bits
InsA	InsA = 1024	Address depth of the INS-RAM
DlyS	DlyS = 5	Size of the DLY-RAM in bits
DlyA	DlyA = 256	Address depth of the DLY-RAM
CnvS	CnvS = 7	Size of the CNV-RAM in bits
CnvA	CnvA = 2048	Address depth of the CNV-RAM
CodS	CodS = 5	Size of the COD-RAM in bits
CodA	CodA = 512	Address depth of the COD-RAM

Table 10: Fields of the configuration section

3.4.2.2. Netlist section

The new netlist shares the same definition as the old one, but with two modifications. The first one is that every chip will have it's own netlist file, so the chip ID of the destination neuron is irrelevant due to the fact that the chip id is defined in the *@Config* section.

The other one is the data part, which holds the synapse. In the old format, the data is merged into a 32bit numeric, containing in the upper half word the R1 value and the lower part the R0 value. In the current models, the synaptic weight is stored in the R1 register, giving a difficult number to deal with. In the proposed format, the register values are kept separated into two fields. The first one contains the R0 register value, where the second one contains the R1 register value. With this, both values are stored in a friendly format, with the capability to use positive and negative numbers, without the fear to override any value in the other register.

Type	Position	Format	Example
Source neuron	0 - 3	i, v, r, c	1, 0, 1, 1
Dest. neuron	4 - 6	v, r, c	2, 0, 1
Synapse number	7	s	2
R0 and R1 data	8 - 9	data	0, -2500

Table 11: Field description of the synapse connection in the low level netlist

As an example, a entry will be:

1, 0, 1, 1, 2, 0, 1, 2, 0, -2500

It is smaller and easier to specify the value of the synapse.

3.4.2.3. Parameters section

In the last section of the netlist, it appears the parameter definition. The format is simple, it only contains several instances of parameters. Each instance contains a header and a data dump.

The header always starts with a '.' marker, giving a easy way of detecting the start of each instance, followed by the metadata separated by '/'.

Field	Example	Description
Address	0x3E3	Address to store the data
Encoding	16	Expected data size
Varname	NEUR	Associated variable name

Table 12: Field description of the parameter declaration in the low level netlist

With this, a possible example of a header will be:

.0x3E3/16/NEUR

Then, the data to be loaded will be represented in a CSV format, where:

- The number of columns shall match with the number of PEs in the codification of the SneS parameter, with the PEs in ascending order (if size is 32bit, codification of 16bit in a 4x4 array, a list of 32 16-bit numbers is mandatory)
- Each row specifies the address offset, starting with the specified address in the header.

In this way, the user has control on the data on the registers and the address allocation, with the columns giving the PE number and the row giving the offset of the base address.

There are some exceptions for the name of the variables, which are data that generates according common parameters. The excluded words are:

- The mnemonics of the instructions
- NVL: stores the Number of Virtual Layers used
- S_X: stores the number of Synapses of the layer X

A file example is shown in Annex 8.6.

3.4.3. HEENS Code Assembler (HCA)

The HEENS Code Assembler is specifically design from scratch with the same specifications that the previous one has, with some extra functionalities:

- Dynamic instruction set, which it is specified in a CSV file.
- Macro instructions to be defined also in the CSV to be as flexible as possible.
- Control on the memory placement of the code and variables.
- Optimization levels for efficiency enhancement.
- Virtual variables specified in the netlist.
- Keywords that contain specific data about the neural network.

In order to accomplish those goals, keywords and directives are added to help some functionalities. The directives are the text that starts with a '.'.

Keyword	Example	Description
Vx	LOOPV V0	Gives the size of the virtual layer x
NVL	LOOP NVL	Gives the effective virtual layers
.org	.org 0x10	Sets the origin to start placing the data or code
.data	.data 0x04	Sets the current address for data
.code	.code	Sets the current address for code

Table 13: Keywords reserved to the HCA

The instruction set is defined in an external CSV file, with 4 elements per row. The codification is as follows:

Function	Example	Description
Mnemonic	LOOPV	Instruction name
Encoding	0b011101	Instruction code
Instruction class	0	Argument class for encoding
Macro expansion	"LDALL \$0; LOOPV"	Alternate use of the mnemonic

Table 14: Field description of the mnemonic instruction set for the HCA

As an example, a entry will be:

LOOPV, 0b011101, 0, "LDALL \$0; LOOPV"

For the macros, the parameters given with the prefix *$* and the position of the argument to be used, starting with 0. The code of the macro is the assembly code. This code is separated by ";", that denotes a new line. So, if in the assmebler the following line appears:

LOOPV 5

the tool replaces the code with:

LDALL 5

LOOPV

In this way, the compiler has the ability to adapt to the instruction set, and adapt to the newer architectures. The code contains the directives for allocating all the data and instructions for the sequencer RAM. Then, the program can use labels to generate subroutines and jumps to create loops. With those 2 files, and the summary from the HNS the HCA generates the bin file for the sequencer.

A file example is shown in Annex 8.7.

3.4.4. HEENS High Level Neural Synthesis (HLNS)

The HEENS High Level Neural Synthesis is a intermediate tool to generate netlist for the HNS in a user friendly format. The drawbacks of the netlist required for the HNS to work is to edit a given neural network. If it is big enough, the file is hard to work with. So, this tool transforms a easier, more general netlist to the format for the HNS. It also contains the same sections as the HNS, but the idea is to have a unique netlist that is easy to generate and work with.

3.4.4.1. Configuration section

In this section, the only thing to have is the names of the chips that will be used, ordered by chip id. This name is the same as a set of JavaScript Obejct Notation (JSON) files with the description of the architectures.

3.4.4.2. Netlist section

The netlist is greatly simplified in terms of parameters required. As the neural network only needs to know the synapse to synthesize into the chips, the entry only needs to know the id of the source neuron, the id of the destination neuron and the synaptic weight in the following format:

Type	Position	Format	Example
Source neuron id	0	id	1
Dest. neuron id	1	id	4
Synap. weight	2	data	-2500

Table 15: Field description of the synapse declaration for the high level netlist

As an example, a entry will be:

1, 4, -2500

In this format, it is easier to implement a neural network. The only parameters are the origin and the destination neurons, with its synapse weight.

3.4.4.3. Parameters section

The biggest drawback of the netlist required in the HNS is the description of the parameters in each PE and row, so a new approach of defining the parameters is designed. It is also based in a header, but the only thing that it is necessary to indicate is the exception values for a given neuron. The structure of the header is as follows:

Field	Example	Description
Address	0x3E3	Address to store the data
Encoding	16	Expected data size
Varname	NEUR	Associated variable name
Number of params.	$NVL	Number of parameters
Default value	0, -6000	Default value for the entry

Table 16: Field description for the parameters of the neurons in high level netlist

With this, a possible example of a header will be:

.0x3E3/16/NEUR/$NVL/0, -6000

As for the special cases, the codification is the neuron id and the value, as shown in the example:

0, 0, -4000

In this case, the neuron with ID 0 will have the value 0 and -4000 for registers R0 and R1 in the SN-RAM memory in a concatenated form.

In addition, recently the HLNS supports also the Trivial Graph Format (.TGF) file format for developing neural networks with existing graph tools like yEd.

A file example is shown in Annex 8.5.

3.5. HTS design flow

In this section, the operations of the HEENS Tool Suite are explained in a high level format. Those tools can be executed on any platform, as it is programmed in python3.

3.5.1. HLNS design flow

The HLNS process is straightforward, as the "only" work to do is to distribute the neural network and place the neurons in the chips. In this TFM, a simple place algorithm is implemented. It is done in four steps:

- Reading of the top netlist
- Seek and load the instantiated chips in the config section
- Place-out: map the synapses to chips
- Generate the required sub netlists specific for each chip

3.5.1.1. Reading the netlist file

The first part is to read the netlist and check if any error is present in the file. Depending on the section, the reading process starts to decode the data accordingly. As such, when the file has been read, all the necessary data has been checked and pre-processed.

3.5.1.2. Seeking and load the chips

With all the data in memory, the HLNS starts to search the JSON files containing the board descriptor of the chips, in order to prepare to the place-out of the neurons in the selected chips.

3.5.1.3. Place-out: map the synapses to chips

Then the tool places the neurons into the chips, with a specified algorithm. For now, a 1 to 1 until fill strategy (one neuron in one PE until it's filled completely) is implemented. Then when all PEs are filled, the tool starts to fill the next chip and so on. Capacity checks are tested in order to check that the network could be deployed in virtualization, but not for the maximum number of synapses. The synapses are more tricky to check because the memory area is also shared with the synaptic parameters. For this reason, it is a working in progress (WIP).

3.5.1.4. Generation of the specific netlists

With the neurons all placed, the final step is to write down the netlist for each chip. It is done with the format specified in the HNS tool.

3.5.2. HNS design flow

The HNS process is designed to analyse and synthesize the neural network, and producing the values related to the neural processors with a network report for adapting the model to the network to be executed. It is done in two steps:

- Reading of the netlist
- Generate the required memory files and report for HCA

3.5.2.1. Reading the netlist file

As the netlist is a self contained, fully defined description, the tool only needs to read this file. Using the same process used on HLNS tool, the file is digested and the data is processed depending on the current section. If an error is detected, the execution is notified and the tool aborts the process.

The data is in memory, the tool starts to process the neurons and classify the synapses as local or as global. Then the information of the neural network is computed. In the end, the memory template is generated and prepared for generating the memory files and the network summary.

3.5.2.2. Generation of the memory files and network summary

The final step is to generate the output files. It is formatted in the specification of questa simulator and also in a TCL script for Vivado, in order to be able to load the memory contents into the chip. In addition, a summary of the netlist containing the number of virtual layers used for the synthesis process and a description of the variables in the SN-RAM is generated. This last summary is required to compile the code in the HCA tool.

3.5.3. HCA design flow

The HCA tool is an assembler with a dynamic instruction set. The instruction set is provided in a CSV file (currently with the name mnemonic.csv) with support on custom macro expansion on every instruction. Together with the code, the instruction set file and the network summary generated by the HNS tool, it generates the binary data for the sequencer. It also has selectable optimization levels for giving the user the choice of performance or one to one output binary to the code. It is done in four steps:

- Reading the code
- Load the network summary
- Optimize the code
- Assemble the binary and generate the output binary

3.5.3.1. Reading the code

It uses a similar process in the HLNS and the HNS tools. It reads the code and starts to pre-process the data in order to detect any syntax error. When this process is finished, the code is separated into main code, subroutine table, data table and a label table.

3.5.3.2. Load the network summary

The next step is to load the summary in order to verify that the variables used on the code are defined in the summary and update the data table with the information stored in the netlist.

3.5.3.3. Optimization process

As the code and data is fully loaded, then the HCA tool starts to optimize the code with the level specified by the user. Those optimizations are only for size and speed optimization at the same time, whereas the speed vs size optimization is not implemented. There are three optimization levels:

- O0: No optimization is performed
- O1: Remove unused data and subroutines
- O2: O1 level + inlining functions that gain performance and reduce memory space

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

<u>O0 optimization level</u>

It doesn't preform any modification into the code nor data.

<u>O1 optimization level</u>

It searches the code and counts the number of references of the subroutines and data labels. If it is 0, then the data or subroutine is removed.

<u>O2 optimization level</u>

It applies the O1 optimization and checks if the subroutine gains speed and reduces program size if it is inlined.

An inlined subroutine is the code of the subroutine that is written where the call instruction references that subroutine. If we define:

- c as the code size of the subroutine without the return call
- n as the number of calls of the subroutine

Then the code is inlined if the following condition is met:

$$nc \leqslant c+1+n \tag{3}$$

$$0 \leqslant c+1+n-nc=c+1+n(1-c)=c-1+1+1-n(c-1)=2+(1-n)(c-1) \tag{4}$$

$$-(1-n)(c-1)=(n-1)(c-1) \leqslant 2 \tag{5}$$

$$(n-1)(c-1) \leqslant 2 \tag{6}$$

So, the tool checks this inequality. If it is true, then the subroutine is inlined into the code. In the end, a small report is displayed in order to see the effects of the optimization.

3.5.3.4. Generation of the memory files and network summary

In this final step, the code is assembled and written into the areas defined by the directives .org, .data and .code. If there is no call to main, the code auto-detects main and generates the jump call.

A debug file is also generated to help debug the assemble process, where the address, the assembled data and the instruction are written in the same file. This is done if the tool have introduced an error.

4. Results

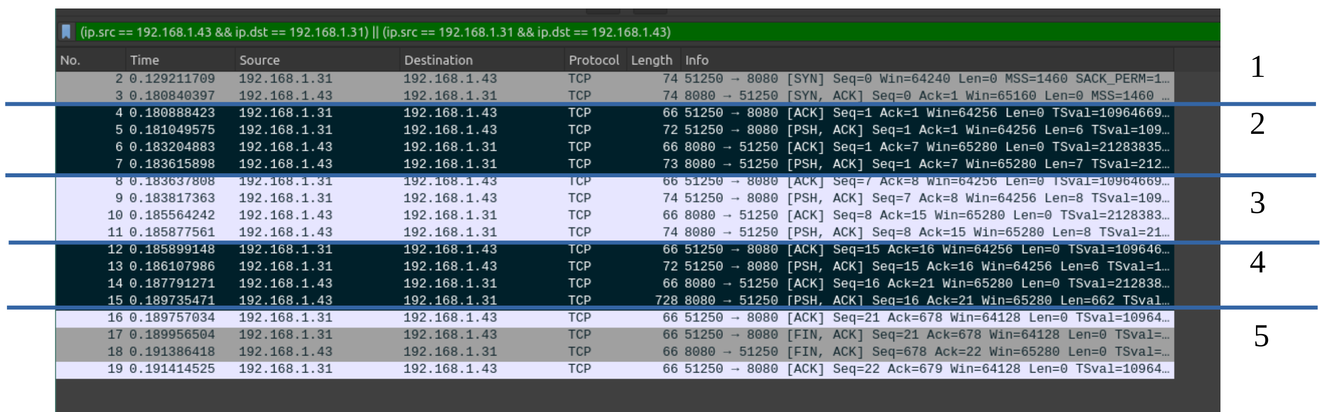
In this section, the results of this project will presented divided in the same sections as section 3. The order is as same as section 3.

4.1. LiveSNN protocol

The protocol is checked using wireshark. A test is done in order to verify the communication is working properly. The test is done by sending 3 commands:

- Get descriptor command configured to send an error in order to check the error message
- The heartbeat command to see if the connection is still open (although in TCP is redundant)
- Send spikes command to see a considerable data packet.

The results shown in the Fig. [15] show 5 regions:


(ip.src == 192.168.1.43 && ip.dst == 192.168.1.31) || (ip.src == 192.168.1.31 && ip.dst == 192.168.1.43)

No.	Time	Source	Destination	Protocol	Length	Info
2	0.129211709	192.168.1.31	192.168.1.43	TCP	74	51250 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1…
3	0.180840397	192.168.1.43	192.168.1.31	TCP	74	8080 → 51250 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 …
4	0.180888423	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=10964669…
5	0.181049575	192.168.1.31	192.168.1.43	TCP	72	51250 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=6 TSval=109…
6	0.183204883	192.168.1.43	192.168.1.31	TCP	66	8080 → 51250 [ACK] Seq=1 Ack=7 Win=65280 Len=0 TSval=21283835…
7	0.183615898	192.168.1.43	192.168.1.31	TCP	73	8080 → 51250 [PSH, ACK] Seq=1 Ack=7 Win=65280 Len=7 TSval=212…
8	0.183637808	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [ACK] Seq=7 Ack=8 Win=64256 Len=0 TSval=10964669…
9	0.183817363	192.168.1.31	192.168.1.43	TCP	74	51250 → 8080 [PSH, ACK] Seq=7 Ack=8 Win=64256 Len=8 TSval=109…
10	0.185564242	192.168.1.43	192.168.1.31	TCP	66	8080 → 51250 [ACK] Seq=8 Ack=15 Win=65280 Len=0 TSval=2128383…
11	0.185877561	192.168.1.43	192.168.1.31	TCP	74	8080 → 51250 [PSH, ACK] Seq=8 Ack=15 Win=65280 Len=8 TSval=21…
12	0.185899148	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [ACK] Seq=15 Ack=16 Win=64256 Len=0 TSval=109646…
13	0.186107986	192.168.1.31	192.168.1.43	TCP	72	51250 → 8080 [PSH, ACK] Seq=15 Ack=16 Win=64256 Len=6 TSval=1…
14	0.187791271	192.168.1.43	192.168.1.31	TCP	66	8080 → 51250 [ACK] Seq=16 Ack=21 Win=65280 Len=0 TSval=212838…
15	0.189735471	192.168.1.43	192.168.1.31	TCP	728	8080 → 51250 [PSH, ACK] Seq=16 Ack=21 Win=65280 Len=662 TSval…
16	0.189757034	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [ACK] Seq=21 Ack=678 Win=64128 Len=0 TSval=10964…
17	0.189956504	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [FIN, ACK] Seq=21 Ack=678 Win=64128 Len=0 TSval=…
18	0.191386418	192.168.1.43	192.168.1.31	TCP	66	8080 → 51250 [FIN, ACK] Seq=678 Ack=22 Win=65280 Len=0 TSval=…
19	0.191414525	192.168.1.31	192.168.1.43	TCP	66	51250 → 8080 [ACK] Seq=22 Ack=679 Win=64128 Len=0 TSval=10964…

Figure 15: Communication between PC and ARMs via ethernet

1. Connection handshake of the TCP protocol
2. Get descriptor communication
3. Heartbeat communication
4. Send spikes communication
5. Disconnection of the TCP protocol

First, the request packet of the get descriptor is shown:

```
  3 0.180840397   192.168.1.43   192.168.1.31   TCP    74 8080
  4 0.180888423   192.168.1.31   192.168.1.43   TCP    66 51250
  5 0.181049575   192.168.1.31   192.168.1.43   TCP    72 51250
  6 0.183204883   192.168.1.43   192.168.1.31   TCP    66 8080
  7 0.183615898   192.168.1.43   192.168.1.31   TCP    73 8080
  8 0.183637808   192.168.1.31   192.168.1.43   TCP    66 51250
  9 0.183817363   192.168.1.31   192.168.1.43   TCP    74 51250
 10 0.185564242   192.168.1.43   192.168.1.31   TCP    66 8080
 11 0.185877561   192.168.1.43   192.168.1.31   TCP    74 8080
 12 0.185899148   192.168.1.31   192.168.1.43   TCP    66 51250
 13 0.186107986   192.168.1.31   192.168.1.43   TCP    72 51250
 14 0.187791271   192.168.1.43   192.168.1.31   TCP    66 8080
 15 0.189735471   192.168.1.43   192.168.1.31   TCP   728 8080
 16 0.189757034   192.168.1.31   192.168.1.43   TCP    66 51250
 17 0.189956504   192.168.1.31   192.168.1.43   TCP    66 51250
 18 0.191386418   192.168.1.43   192.168.1.31   TCP    66 8080
 19 0.191414525   192.168.1.31   192.168.1.43   TCP    66 51250

    Window size value: 502
    [Calculated window size: 64256]
    [Window size scaling factor: 128]
    Checksum: 0x83c7 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  + Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  + [SEQ/ACK analysis]
  + [Timestamps]
    TCP payload (6 bytes)

0000  64 5d 86 ba 0b 71 d8 cb  8a cb 25 c8 08 00 45 00   d]···q·· ··%···E·
0010  00 3a 7c 49 40 00 40 06  3a da c0 a8 01 1f c0 a8   ·:|I@·@· :·······
0020  01 2b c8 32 1f 90 d5 58  e3 3e bc 04 44 7d 80 18   ·+·2···X ·>··D}··
0030  01 f6 83 c7 00 00 01 01  08 0a 41 5a c1 e3 0c af   ········ ··AZ····
0040  a7 d3 05 00 00 00 06 03                            ········
```

Figure 16: Frame of the "Get Descriptor" command

As it is shown in Fig. [16], the frame is correctly formatted. The selected bytes are formatted correctly as the protocol says, where the TransferID is 5, message length is 6 bytes and the command is the number 3, so it is a Get Descriptor command.

Then, the programmed error response is received:

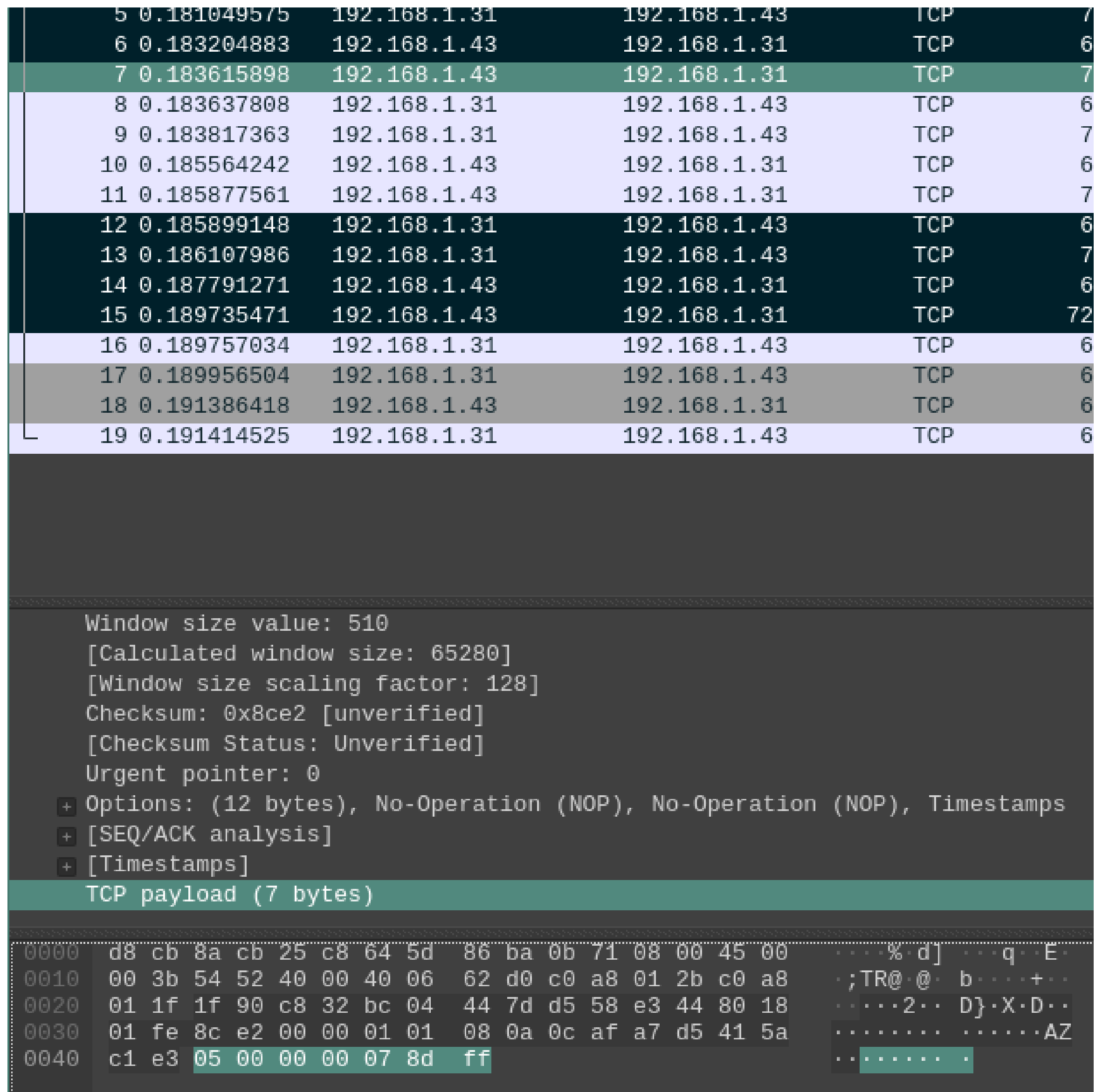


Figure 17: Error response result

As it is shown in Fig. [17], then the error response is received as expected. The TransferID is also 5, the message length is 7 bytes, that is the header frame length plus one byte of data. The command is 0x8D, so it is a response frame thanks to the MSB that it is set, and the command is 0x0D, which it is an error command. The data is 0xFF. The error codes are programmed in the ZYNQ as negative numbers, so that means -1. In this case -1 shows a generic error.

Following the test, the heartbeat command is sent:

No.	Time	Source	Destination	Protocol	Length	Info
2	0.129211709	192.168.1.31	192.168.1.43	TCP	74	51250 → 808
3	0.180840397	192.168.1.43	192.168.1.31	TCP	74	8080 → 5125
4	0.180888423	192.168.1.31	192.168.1.43	TCP	66	51250 → 808
5	0.181049575	192.168.1.31	192.168.1.43	TCP	72	51250 → 808
6	0.183204883	192.168.1.43	192.168.1.31	TCP	66	8080 → 5125
7	0.183615898	192.168.1.43	192.168.1.31	TCP	73	8080 → 5125
8	0.183637808	192.168.1.31	192.168.1.43	TCP	66	51250 → 808
9	0.183817363	192.168.1.31	192.168.1.43	TCP	74	51250 → 808
10	0.185564242	192.168.1.43	192.168.1.31	TCP	66	8080 → 5125
11	0.185877561	192.168.1.43	192.168.1.31	TCP	74	8080 → 5125
12	0.185899148	192.168.1.31	192.168.1.43	TCP	66	51250 → 808
13	0.186107986	192.168.1.31	192.168.1.43	TCP	72	51250 → 808
14	0.187791271	192.168.1.43	192.168.1.31	TCP	66	8080 → 5125
15	0.189735471	192.168.1.43	192.168.1.31	TCP	728	8080 → 5125
16	0.189757034	192.168.1.31	192.168.1.43	TCP	66	51250 → 808
17	0.189956504	192.168.1.31	192.168.1.43	TCP	66	51250 → 808
18	0.191386418	192.168.1.43	192.168.1.31	TCP	66	8080 → 5125
19	0.191414525	192.168.1.31	192.168.1.43	TCP	66	51250 → 808

```
Window size value: 502
[Calculated window size: 64256]
[Window size scaling factor: 128]
Checksum: 0x83c9 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[SEQ/ACK analysis]
[Timestamps]
TCP payload (8 bytes)
```

```
0000  64 5d 86 ba 0b 71 d8 cb  8a cb 25 c8 08 00 45 00   d]···q·· ··%···E·
0010  00 3c 7c 4b 40 00 40 06  3a d6 c0 a8 01 1f c0 a8   ·<|K@·@· :·······
0020  01 2b c8 32 1f 90 d5 58  e3 44 bc 04 44 84 80 18   ·+·2···X ·D··D···
0030  01 f6 83 c9 00 00 01 01  08 0a 41 5a c1 e6 0c af   ········ ··AZ····
0040  a7 d5 06 00 00 00 08 0e  00 00                     ········ ··
```

Figure 18: Frame of the "Heartbeat" command

As it is shown in Fig. [18], the format of the data is still correct. The TransferID has been incremented by one, and the command send is the Heartbeat (0x0E). It requires two parameters, a and b, which are of 8-bit unsigned integers. Those are for checking that the ARMs are still connected and they are running. As such, in this case the pair is (0, 0) the expected response is he pair (b, a+1), so (0, 1).

Then, the heartbeat response is received:

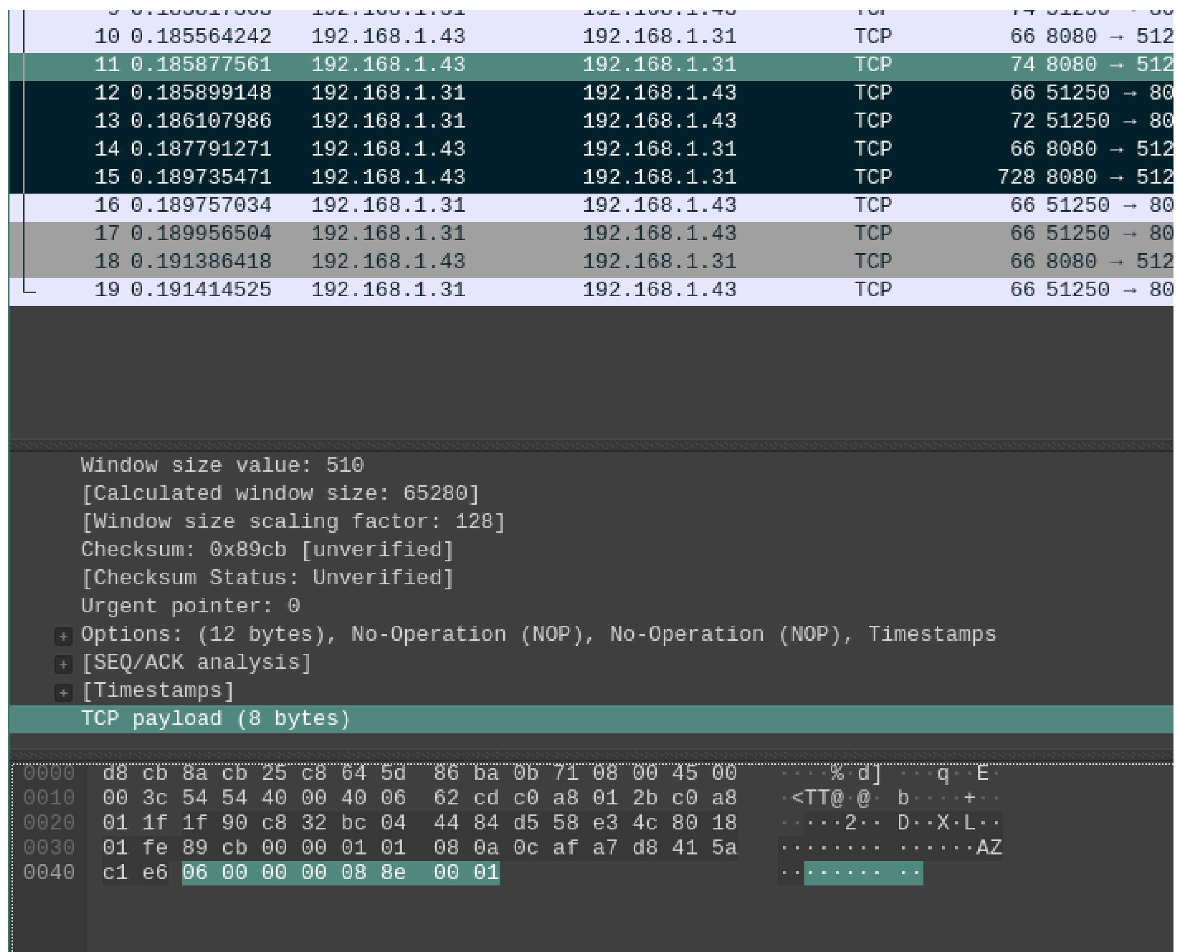


Figure 19: Response of the “Heartbeat” command

As it is shown in Fig. [19], the response arrives, and the data still has the correct frame. The TransferID is still 6, and the command is 0x8E, showing that it is a response and the command still is the Heartbeat. The received data is (0, 1), so the ARMs are still running.

For Ethernet with TCP/IP it does not make sense, but if this protocol is deployed, for example, in a SPI bus, then this command is useful to have in order to check the status of the processors.

Finally, the send spikes command is sent:

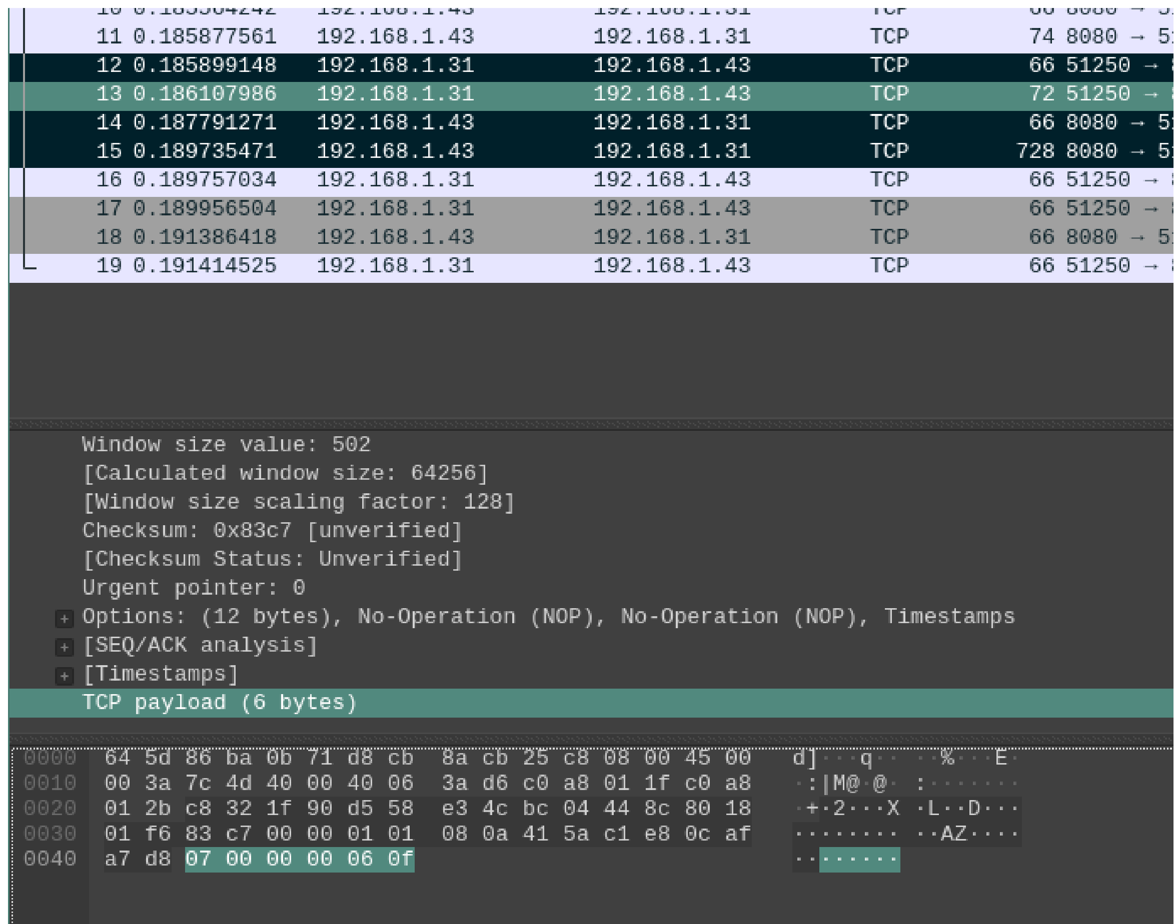


Figure 20: Frame of the “Get Spikes” command

As it is shown in Fig. [20], the format of the data is still correct. The TransferID has been incremented by one, giving a result of 7, and the command send is the Get Spikes (0x0F). So the response should have the information of the spikes generated from the last Get Spikes command.

At last, the send spike response is received:

No.	Time	Source	Destination	Protocol	Length
11	0.185877561	192.168.1.43	192.168.1.31	TCP	74
12	0.185899148	192.168.1.31	192.168.1.43	TCP	66
13	0.186107986	192.168.1.31	192.168.1.43	TCP	72
14	0.187791271	192.168.1.43	192.168.1.31	TCP	66
15	0.189735471	192.168.1.43	192.168.1.31	TCP	728
16	0.189757034	192.168.1.31	192.168.1.43	TCP	66
17	0.189956504	192.168.1.31	192.168.1.43	TCP	66
18	0.191386418	192.168.1.43	192.168.1.31	TCP	66
19	0.191414525	192.168.1.31	192.168.1.43	TCP	66

```
Window size value: 510
[Calculated window size: 65280]
[Window size scaling factor: 128]
Checksum: 0x0f06 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[SEQ/ACK analysis]
[Timestamps]
TCP payload (662 bytes)
```

```
0000  d8 cb 8a cb 25 c8 64 5d  86 ba 0b 71 08 00 45 00   ····%·d] ···q··E·
0010  02 ca 54 56 40 00 40 06  60 3d c0 a8 01 2b c0 a8   ··TV@·@· `=···+··
0020  01 1f 1f 90 c8 32 bc 04  44 8c d5 58 e3 52 80 18   ·····2·· D··X·R··
0030  01 fe 0f 06 00 00 01 01  08 0a 0c af a7 db 41 5a   ········ ······AZ
0040  c1 e8 07 00 00 02 96 8f  00 00 00 00 00 00 00 0a   ········ ········
0050  00 00 03 0d 00 00 01 35  00 00 02 8a 00 00 03 1f   ·······5 ········
0060  00 00 01 35 00 00 02 d8  00 00 02 65 00 00 02 93   ···5···· ···e····
0070  00 00 03 01 00 00 02 38  00 00 00 01 00 00 00 16   ·······8 ········
0080  00 00 00 01 00 00 01 92  00 00 02 ea 00 00 01 b0   ········ ········
0090  00 00 01 c0 00 00 00 d3  00 00 02 36 00 00 00 3a   ········ ···6···:
00a0  00 00 00 3c 00 00 02 5c  00 00 02 9b 00 00 01 ec   ···<···\ ········
```

Figure 21: Response of the “Get Spikes” command

As it is shown in Fig. [21], the response to the “get spikes” has the correct size and data. The transmission of the spikes is a very simple one, but has a lot of wasted space. So the next version is to incorporate a compression algorithm. It is currently being studied, but the Run Length Encoding scheme for now is the optimal one. It will be implemented similar to the one implemented in the video game “*Pokemon Red/Blue*” in the sprite compression for the pokemon’s battle sprites.

As the preliminary tests, the compression ratio achieved in python in order to simulate the algorithm was 662/290 = 2.2875.

4.2. LiveSNN program

In this section, the program is deployed into the board. The compiler only shows two warnings, where those are referring in the auto-code of Vivado. Omitting those warnings, the rest of the code written is free of errors and warnings.

The following compiler configuration is used:

- g++: for compiling C++ code.
- -wextra: add extra warnings in order to check the robustness of the code.
- -wall: enables all the warnings for getting the last possible misunderstanding of the compiler from the source code.
- -pedantic: enables strict rules of programming in order to ensure that the programmer is not leaving things that may cause a misbehaviour on the code. For example a integer that does not fit inside the size of the variable that it is writing to.
- -werror: treats every warning as an error.

By this rules, the code is compiled and when the phase of correcting any possible errors, the debugging process is started.

On the Fig. [22] the setup is shown. The screen contains 3 windows.

The background is the Vivado Software Development Kit (SDK), where the code is typed and then programmed on the ZYNQ.

Then the black screen on the left shows the serial port of the ZYNQ, where the log is printed out.

Finally to the right there is another terminal. In this case it is the TELNET connection, where for testing purposes, the command help is executed. This prints all the commands implemented in the console class.

On the bottom, there is the ZC706 board containign the ZYNQ 7045 SoC, where the code is running. It is not seen clearly, but there are green LEDs that are blinking in order to check if the ARMs are hang up in a way that it is easy to see.

So By programming in the board ZC706 the Telnet port can be accessed, the serial port it is also operative:

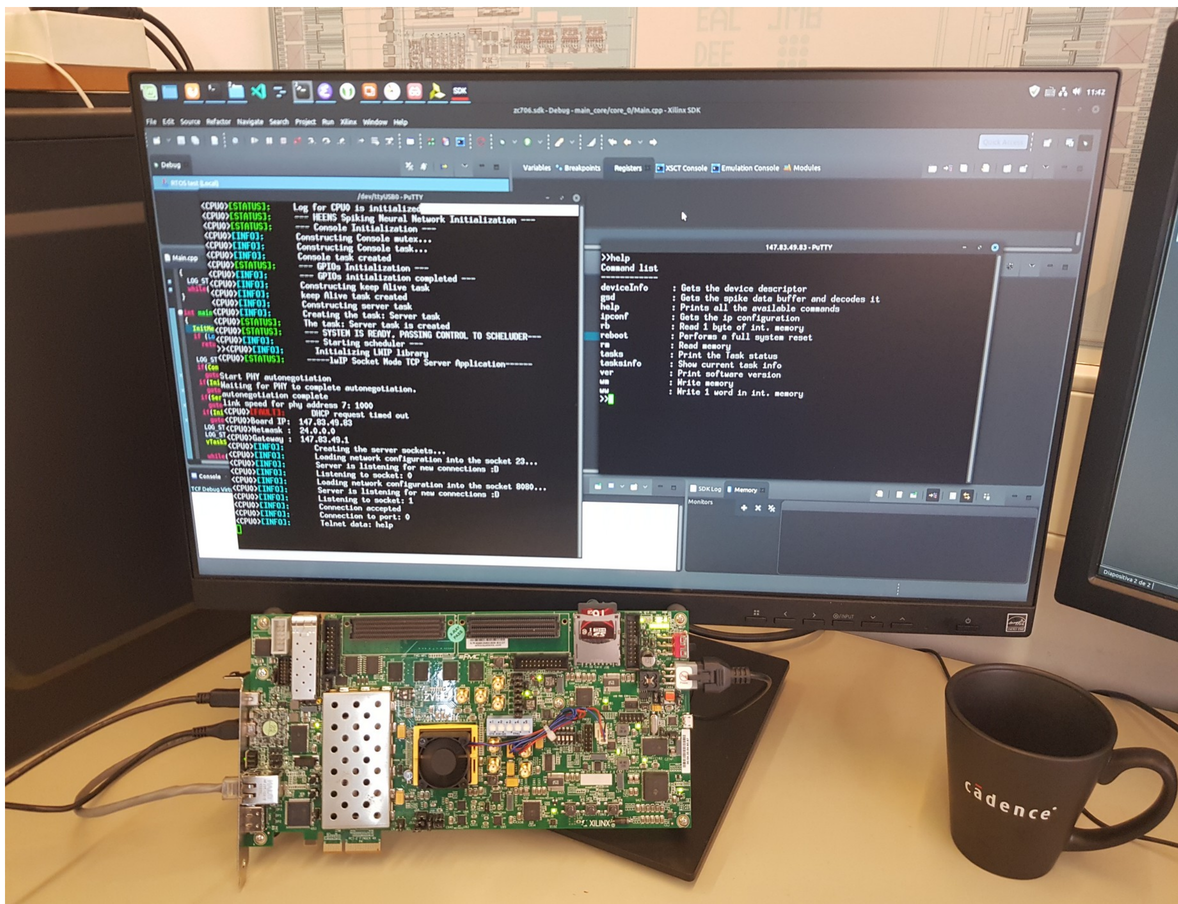


Figure 22: Setup of the test of the code deployed on the ARMs in the ZC706 with the console and the TELNET connections open

4.3. HEENS Toolchain Suite

In this section, the HTS is tested with the comparison with the code generated by the old toolchain. For the netlist synthesis there is no change between files, due to the fact that the neural network is the same.

For the code generator the different cases are tested, depending on the optimization level. Both of them uses the model included in the appendices 8.4 and 8.7. It is a Leaky Integrate and Fire model, and both versions have the exact same configuration and instruction count in order to study the performance of the assemblers. The results are:

OPT LEVEL	Previous version [bytes]			HTS [bytes]		
	code	data	total	Code	data	total
0	103	16	119	82	18	100
1				82	10	92
2				68	10	78

Table 17: Comparison between the code generated by the old and the new toolchain

In order to show in a easier form the performance of the tools, the relative differences are computed in order to know the advantages of the toolchains. The reference of the differences are the old toolchain for the first column group, and the O0 level from the proposed toolchain.

OPT LEVEL	DIFF BETWEEN TOOLS [%]			DIFF BETWEEN OP LEVELS [%]		
	code	data	total	Code	data	total
0	-20.39	12.50	-15.97	-	-	-
1	-20.39	-37.50	-22.69	00.00	-44.44	-08.00
2	-33.98	-37.50	-56.43	-17.07	-44.44	-22.00

Table 18: Relative difference of the code generated by the different toolchains

The results show an improvement on the size of the binary generated, and, in consequence, it could support bigger models, except the O0 level, which the data section has increased 12.5%. This is due to the fact that all the keywords are declared on the code even if some of them are not used.

5. Budget

Element	Quantity[u]	Price [€/u]	Cost [€]
Hardware			
ZYNQ ZCU706	1	2667.00	2667.00
Computer	1	1000.00	1000.00
USB communication cables	3	3.01	9.03
Subtotal			**3676.03**
Software			
OS from the computer (Linux MINT)	1	0.00	0.00
RTOS for the ZYNQ SoC (FreeRTOS)	1	0.00	0.00
Vivado (WebPACK)	1	0.00	0.00
Code editor (EMACS)	1	0.00	0.00
Subtotal			**0.00**
Development			
Development cost	384h	12.50	4800.00
Subtotal			**4800.00**
Total			**8476.03**

Table 19: Budget of the project

6. Environment Impact

This project is not focused on solving environmental impacts, as it consists in several tools to develop and test different models and neural networks on a simulator and on chip via a SoC as commented in previously chapters.

Nonetheless as this tools tries to extract more performance out of the HEENS architecture, the cost of running this tools increases a little bit, but the whole deployment of new neural networks has been accelerated.

So in overall, those changes try to minimize the time required, an this translates into less energy consumption. The kg of CO2 per kWh from Spain is 0,2654[8]. As such, the tools automatize and prevents some human errors, leading a reduction between a 20% to a 80% , from 0.21232 kgCO2/kWh to 0.05308 kgCO2/kWh respectively, depending on the required task and neural network to implement. This numbers are a loosely estimation in order to study the CO2 reduction.

This approximation shows that it reduces the CO2 footprint but depending on the task it will be a marginal reduction.

7. Conclusions and future development

In this final chapter, the conclusions of this project will be discussed and give some future directions in order to continue the work done on this project.

7.1. Conclusions

This is a big project that several people work on it. As the HEENS is a prototype, there are things that are in the scope of the project but cannot be implemented yet to the software.

The LiveSNN protocol has been implemented in a basic way that could be extensible enough to be able to communicate the ARMs to the PC. The basic needs for the HEENS to operate are covered, as well as extra functionality that gives this protocol the capability to update and read information while the network is operating. As such, the debugging could be done remotely. This has it's advantages and disadvantages. The advantage is the remote access to the HEENS, but the problem is that there is a lack of cybersecurity.

The LiveSNN program has developed the core functionalities. Although the communications are tested and the code is written, part of the code that manages the HEENS part is missing due to the waiting on the update on the interface between the ARMs and the HEENS. So it preformed well, but it misses this part, and the design on secondary parts of the code, like firmware updates.

The HTS preforms very well with the new netlist design, although minor design decisions need to be discussed. These details are the variable instantiation on each neural processor and the equivalent name on the model file. Despite of this, the ability to work at high and low level gives the user the fast deployment without sacrificing the opportunity to work in a low level environment.

7.2. Future work

As mentioned in the conclusions, there are things that are in the scope of the project but cannot be implemented yet to the software, so the following points will describe the next steps to do that may provide new TFG and TFM thesis in the future.

For the LiveSNN protocol, some commands, like the START, STOP, STEP, are not implemented yet due to the current status of the HEENS architecture which does not support the functionality or is being implemented. So, when it is finished the modifications, the rest of the protocol needs to be implemented.

For the LiveSNN program, it is only implemented the code related to the communications between the computer and the ARMS, so the only parts missing is security on the communications via Ethernet and the firmware management unit, that it is on second

plane. In addition, the protocol may use a compression algorithm as said in the result section (4.1) in order to have better transmission efficiency.

Finally, for the HEENS Toolchain Suite, it has been checked and tested to preform some easy neural networks in order to validate. The HEENS high Level Neural Synthesis uses a one to one neural mapping in the processors for a quick test and deployment of neural networks in only one board. As such, one possible project could be to implement different placing routines for the tool, to allow smart placing and a better resource management.

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH telecos BCN

Bibliography

[1] "Neurons and synapses - Ms. Frost A world of biology....." https://aworldofbiology.weebly.com/neurons-and-synapses.html (accessed Sep. 02, 2020).

[2] DARPA, "SyNAPSE Program Develops Advanced Brain-Inspired Chip." https://www.darpa.mil/news-events/2014-08-07 (accessed Jul. 13, 2020).

[3] Dharmendra S. Modha, "IBM Research: Brain-inspired Chip." https://www.research.ibm.com/articles/brain-chip.shtml (accessed Jul. 13, 2020).

[4] APT Research Group, "Research Groups: APT - Advanced Processor Technologies (School of Computer Science - The University of Manchester)." https://apt.cs.manchester.ac.uk/projects/SpiNNaker/architecture/ (accessed Jul. 13, 2020).

[5] M. Davies *et al.*, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018, doi: 10.1109/MM.2018.112130359.

[6] FreeRTOS team, "RTOS - Free professionally developed and robust real time operating system for small embedded systems development." https://freertos.org/RTOS.html (accessed Jul. 13, 2020).

[7] Xilinx, "PetaLinux Tools Documentation Reference Guide," 2020. [Online]. Available: www.xilinx.com.

[8] EEA, "CO2 emission intensity — European Environment Agency." https://www.eea.europa.eu/data-and-maps/daviz/co2-emission-intensity-5#tab-chart_2 (accessed Jul. 13, 2020).

[9] "ArtificialNeuronModel_english.png (PNG Image, 1682 × 799 pixels)." https://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png (accessed Sep. 02, 2020).

[10] "File:loihi floorplan.png - WikiChip." https://en.wikichip.org/wiki/File:loihi_floorplan.png (accessed Sep. 02, 2020).

[11] "zynq-mp-core-dual.png (PNG Image, 800 × 900 pixels)." https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png (accessed Sep. 02, 2020).

[12] "ANNDiagram.png (PNG Image, 1250 × 1057 pixels) - Scaled (86%)." https://st4.ning.com/topology/rest/1.0/file/get/2808361999?profile=original (accessed Aug. 24, 2020).

[13] "DARPA_SyNAPSE_16_Chip_Board.jpg (JPEG Image, 1169 × 648 pixels)." https://upload.wikimedia.org/wikipedia/commons/9/9c/DARPA_SyNAPSE_16_Chip_Board.jpg (accessed Sep. 02, 2020).

[14] "Synapse-QBI-brain-neuroscience.jpg (JPEG Image, 1937 × 1066 pixels) - Scaled (85%)." https://qbi.uq.edu.au/files/7758/Synapse-QBI-brain-neuroscience.jpg (accessed Aug. 24, 2020).

8. Appendices

This section contains the different examples and extra information relating to the project.

8.1. Protocol format structure

In this subsection, the different command frames will be explained with a transmission example:

8.1.1. NULL

This message is an empty message. If send, or detected, the message has been corrupted. This message has a empty payload.

Request arguments: None

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x03	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x00	NULL command

Table 20: LiveSNN NULL command request frame

- Response: None

In this case, a NULL message is sent. As it is a empty frame, there is no payload and the receiver should ignore this message. If this frame is ever transmitted, is a clear sign of a bug in the implementation of the protocol.

8.1.2. STATUS

This message requests the current status of the neural network. The request has no arguments whereas the response returns a uint32_t status register.

Request arguments: None

Response arguments: uint32_t status

Example:

- Request:

Byte	Value	Description
0x00	0x07	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x01	STATUS command

Table 21: LiveSNN STATUS command request frame

- Response:

Byte	Value	Description
0x00	0x07	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x81	STATUS command
0x06	0xXX XX XX XX	Current status

Table 22: LiveSNN STATUS command response frame

The computer wants to know the current state of the device. As such, it sends a STATUS message to the ARM processor. It replies with it's configuration word. This configuration is a uint32_t word.

Bit	Name	Description
31 - 1	Reserved	Not used
0	IsSnnRunning	NN execution status (STOP, RUNNING)

Table 23: LiveSNN status bit format

8.1.3. CONFIGURATION

NOT IMPLEMENTED YET

This message requests the current configuration of the neural network.

Request arguments: None

Response arguments: uint32_t configuration

Example:

- Request:

Byte	Value	Description
0x00	0x12	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x02	CONFIGURATION command

Table 24: LiveSNN CONFIGURATION command request frame

- Response:

Byte	Value	Description
0x00	0x12	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x82	CONFIGURATION command
0x06	0xXX XX XX XX	Current status

Table 25: LiveSNN CONFIGURATION command response frame

The computer wants to know the current configuration of the device. As such, the computer sends a CONFIGURATION message, which the device responses with it's configuration.

8.1.4. DESCRIPTOR

NOT IMPLEMENTED YET

This message requests the current descriptor of the chip.

Request arguments: None

Response arguments: string descriptor

Example:

- Request:

Byte	Value	Description
0x00	0x21	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x03	DESCRIPTOR command

Table 26: LiveSNN DESCRIPTOR command request frame

- Response:

Byte	Value	Description
0x00	0x21	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x83	DESCRIPTOR command
0x06	0xXX XX XX XX	Descriptor data

Table 27: LiveSNN DESCRIPTOR command response frame

The computer wants to know the what are the properties of the device. As such, the computer sends a DESCRIPTOR message, which the device responses with it's descriptor.

8.1.5. START

This message requests the NN to start. There is no arguments used.

Request arguments: None

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x02	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x04	START command

Table 28: LiveSNN START command request frame

- Response:

Byte	Value	Description
0x00	0x02	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x84	START command

Table 29: LiveSNN START command response frame

The computer wants to start the NN, so sends the START message. With this, the NN starts it's operation. If not, an ERROR message will be sent.

8.1.6. STOP

This message requests the NN to stop. There is no arguments used.

Request arguments: None

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x50	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x05	STOP command

Table 30: LiveSNN STOP command request frame

- Response:

Byte	Value	Description
0x00	0x50	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x85	STOP command

Table 31: LiveSNN STOP command response frame

The computer wants to stop the NN, so sends the STOP message. With this, the NN starts it’s operation.

8.1.7. STEP

This message requests the NN to execute n cycles.

Request arguments: uint32_t steps

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x0A	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x06	STEP command
0x06	0x00 00 01 00	Number of steps

Table 32: LiveSNN STEP command request frame

- Response:

Byte	Value	Description
0x00	0x0A	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x86	STEP command

Table 33: LiveSNN STEP command response frame

The computer wants to do 256 execution steps on the NN, so sends the STEP message with an argument of 256. With this, the NN tries to execute that number of steps. If not, an ERROR message will be sent.

8.1.8. RESET

This message requests the SoC to reset. There is no arguments used.

Request arguments: None

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x0F	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x07	RESET command

Table 34: LiveSNN RESET command request frame

- Response:

Byte	Value	Description
0x00	0x0F	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x87	RESET command

Table 35: LiveSNN RESET command response frame

The computer wants to restart the hole SoC due to a glich, so the REBOOT message is sent. Then the SoC reboots itself.

NOTE: This command will kill the communication between the computer and the SoC with the TELNET or LiveSNN protocols (Ethernet), but not the serial port.

8.1.9. UPLOAD FIRMWARE

This message uploads a new firmware for the NN.

Request arguments: uint8_t* newFirmware

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0x1F	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x08	UPLOAD FIRMWARE command
0x06	0xXX XX XX XX	New firmware

Table 36: LiveSNN UPLOAD FIRMWARE command request frame

- Response:

Byte	Value	Description
0x00	0x1F	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x88	UPLOAD FIRMWARE command

Table 37: LiveSNN UPLOAD FIRMWARE command response frame

The computer wants to update the NN to a newer version, so it sends the UPLOAD FIRMWARE to the device. Then the SoC installs the new firmware to the NN and reboots itself.

NOTE: This command will kill the communication between the computer and the SoC with the TELNET or LiveSNN protocols (Ethernet), but not the serial port.

8.1.10. DOWNLOAD FIRMWARE

This message downloads the current firmware that has the NN.

Request arguments: None

Response arguments: uint8_t* currentFirmware

Example:

- Request:

Byte	Value	Description
0x00	0x20	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x09	DOWNLOAD FIRMWARE command

Table 38: LiveSNN DOWNLOAD FIRMWARE command request frame

- Response:

Byte	Value	Description
0x00	0x20	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x89	DOWNLOAD FIRMWARE command
0x06	0xXX XX XX XX	Current firmware

Table 39: LiveSNN DOWNLOAD FIRMWARE command response frame

The computer wants to keep a copy of the current firmware, so it sends the DOWNLOAD FIRMWARE to the SoC. Then the SoC will send the NN firmware stored in the static memory.

8.1.11. SPIKE REPORT

This message requests the spike report of the NN.

Request arguments: None

Response arguments: uint8_t* spikeReport

Example:

- Request:

Byte	Value	Description
0x00	0xF1	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x0A	SPIKE REPORT command

Table 40: LiveSNN SPIKE REPORT command request frame

- Response:

Byte	Value	Description
0x00	0x03	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x00	SPIKE REPORT command
0x06	0xXX XX XX XX	Spike report

Table 41: LiveSNN SPIKE REPORT command response frame

The computer wants to know how many spikes per second is generating the NN, so it sends the SPIKE REPORT message. The SoC will send back the activity of the neurons on average.

8.1.12. RASTER REPORT

This message requests the raster report of the NN.

Request arguments: None

Response arguments: uint8_t* rasterReport

Example:

- Request:

Byte	Value	Description
0x00	0x2C	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x0B	RASTER REPORT command

Table 42: LiveSNN RASTER REPORT command request frame

- Response:

Byte	Value	Description
0x00	0x2C	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x8B	RASTER REPORT command
0x06	0xXX XX XX XX	Raster report

Table 43: LiveSNN RASTER REPORT command response frame

The computer wants to know the neurons that have fired, so it sends the RASTER REPORT message. The SoC will send back an array of neurons IDs and time stamps, where the neurons that appeared have fired at that timestamp.

8.1.13. OTHER PROTOCOL

This message uses the LiveSNN protocol as a gateway for other protocols, like SPI, I2C, CAN, etc.

Request arguments: uint8_t peripheral, uint8_t* frame

Response arguments: uint8_t peripheral, uint8_t* frameResponse

Example:

- Request:

Byte	Value	Description
0x00	0x5A	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x01	op command
0x06	0x01	Peripheral for forwarding
0x07	0x11 03 006B 0003 7687	Modbus RTU frame

Table 44: LiveSNN OTHER PROTOCOL command request frame

- Response:

Byte	Value	Description
0x00	0x5A	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x00	NULL command
0x06	0x01	Peripheral for forwarding
0x07	0x11 03 06 AE41 5652 4340 49AD	Modbus RTU frame

Table 45: LiveSNN OTHER PROTOCOL command response frame

In this example, the computer sends a Modbus RTU frame over the LiveSNN protocol in order to ask a sensor (ID of 11) some data (Registers 40108, 40109 and 40110), and the ARM replies with the data that the sensor has sent (0xAE41, 0x5652 and 0x4340, respectively).

The peripheral ID table is the following:

ID	Peripheral
0x00	UART A
0x01	UART B
0x02	CAN A
0x03	CAN B
0x04	SPI A
0x05	SPI B
0x06	I2C A
0x07	I2C B
0x08	USB A
0x09	USB B
0x0A	Ethernet A
0x0B	Ethernet B
0x0C	SD A
0x0D	SD B

Table 46: Peripheral ID definitions

As it is seen in the table, the Least Significant Bit (LSB) of the code determines the which port (A or B) is forwarded the message.

8.1.14. ERROR

This message alerts that an error has occurred

Request arguments: uint8_t error

Response arguments: uint8_t error

Example:

- Request: None

- Response:

Byte	Value	Description
0x00	0xC4	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x8D	ERROR command

Table 47: LiveSNN ERROR command response frame

This message will send as a request when there is a problem or as a response if a problem has occurred while the it was executing a LiveSNN request.

8.1.15. HEARTBEAT

This message sends a heart beat message for checking the quality of the communication.

Request arguments: uint8_t a, uint8_t b

Response arguments: uint8_t na, uint8_t nb

Example:

- Request:

Byte	Value	Description
0x00	0xD1	Transfer ID
0x01 - 0x04	0x00 00 00 08	Message length
0x05	0x01	STATUS command
0x06	0x01	a parameter
0x07	0x01	b parameter

Table 48: LiveSNN HEARTBEAT command request frame

- Response:

Byte	Value	Description
0x00	0xD1	Transfer ID
0x01 - 0x04	0x00 00 00 08	Message length
0x05	0x00	NULL command
0x06	0x01	na parameter
0x07	0x02	nb parameter

Table 49: LiveSNN HEARTBEAT command response frame

For checking the connection, the HEARTBEAT message is sent. It sends 2 parameters: a and b. For example, the sender sends a and b being equal to 1.

Then, the receiver reads the 2 parameters and increment b by 1, and sends back the response.

Finally, the sender will check that a is not modified and b is incremented by 1.

If not, there is a problem in the communication and/or the receiver.

8.1.16. GET SPIKE

This message sends the new spike generated by the NN.

Request arguments: SpikeData data

Response arguments: None

Example:

- Request:

Byte	Value	Description
0x00	0xB2	Transfer ID
0x01 - 0x04	0x00 00 00 0A	Message length
0x05	0x0F	*GET SPIKE* command
0x06	0xXX XX XX XX	Spike data

Table 50: LiveSNN GET SPIKE command request frame

- Response:

Byte	Value	Description
0x00	0xB2	Transfer ID
0x01 - 0x04	0x00 00 00 06	Message length
0x05	0x8F	*GET SPIKE* command

Table 51: LiveSNN GET SPIKE command response frame

The NN has generated new spikes. In order to generate a real time raster plot of the neurons, the SEND SPIKE command is sent with the current spikes that have just fired. The computer then acknowledges the message with the response message.

8.2. SEND SPIKE data format

The communication between the computer and the SoC will be using the LiveSNN protocol. As for the spike data for the commands "SEND SPIKE" and "RASTER REPORT", the payload is serialized as follows:

Byte	Name	Description
0x00 - 0x03	TimeStamp 1	First timestamp
0x04 - 0x07	nSpikes	Number of spikes
0x08 - 0x0B	Spike 1	Spike 1
0x0C - 0x0F	Spike 2	Spike 2
...	...	...
n - n+3	TimeStamp N	Nth timestamp
n+4 - n+7	nSpikes	Number of spikes
n+8 - n+11	Spike 1	Spike 1
n+12 - n+15	Spike 2	Spike 2
...	...	...

Table 52: Serialization format of the spike data for LiveSNN

8.3. Example of the old netlist

This example is a ring oscillator that forms a 4x4 square.

netlist.lst:

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

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

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

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

In the following example, the memory layout of the neurons are configured. Each column is a PE, and each line is a different address of the memory

neuron.csv:

@0x3E3
-4000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000 -6000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
-7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000 -7000
@0x3FD 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
790397107 281151190 3309506875 1263568161 2343084726 1445520166 3956335112 346198339 3024860000 3146000210 6789335 139614675 560173392 829721 2009585225 1008007293
1522134871 2478058071 4125695476 2020487816 3716890928 1203286407 537884939 1705442851 1158594759 225757523 1855561354 3838380997 411379302 137305429 1315299481 49213318
@1023 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
# # # # # # # # # # # # # # # #

8.4. Example of the old code

This code is a functional Leaky Integrate and Fire model for the HEENS architecture in the old format.

LIF.asm:

```
; GOTO CODE
;
; Integrate and fire. Both non-virtual and virtual
; DEFAULT operation without virtual layers
; REMOVE: 'semicolon%VIRT ' for operation with virtual layers

; Network definitions

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

.DATA

; Virtual layers

V0 = "0000000F" ; Number of assigned synapses to the main layer
;%VIRT V1 = "00000004" ; Number of assigned synapses to virtual layer 1
;%VIRT V2 = "00000006" ; Number of assigned synapses to virtual layer 2
;%VIRT V3 = "00000008" ; Number of assigned synapses to virtual layer 3
;%VIRT V4 = "00000010" ; Number of assigned synapses to virtual layer 4
;%VIRT V5 = "00000005" ; Number of assigned synapses to virtual layer 5
;%VIRT V6 = "00000004" ; Number of assigned synapses to virtual layer 6
;%VIRT V7 = "00000003" ; Number of assigned synapses to virtual layer 7
;%VIRT VLAYERS="00000007" ; Number of virtual layers.

; Membrane potential parameters common to all neurons
VREST="FFFFE4A8"    ; Resting potential -70 mV = -7000 in tens of of uV
VTHRES="FFFFEA84"   ; Threshold voltage -55 mV = -5500
VDEPOL="FFFFE0C0"   ; Depolarization voltage -80 mV = -8000
VACT = "00001771"   ; Action potential +10 mV = +1000
;
; Synapse parameters common to all neurons come here
; TBD
;
; Neural and Synaptic RAM addresses
SEEDH_ADDR = "00000200" ; Address of noise seed in NSBRAM
SEEDL_ADDR = "00000201" ;
NEU_ADDR="00000100"   ; First address of Neural parameters in NSBRAM
SYN_ADDR="00000000"   ; First address of Synaptic parameters in NSBRAM.
;
; General constants
;THAU_MEM="00007F00"   ; Membrane time constant decay (inverse value). To be tuned
;THAU_MEM="00007EE0"   ; Membrane time constant decay (inverse value). To be tuned. Thau = 113.78
```

```
; THAU_MEM = 32768/(1-1/tau)
THAU_MEM="0000799A"   ; Membrane time constant decay (inverse value). To be tuned. Thau =
20
NOISE_MSK="0000001F"  ; Noise mask. To be tuned

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

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

.CODE
;
GOTO MAIN      ; Jump to main program
;
; *************************** PROCEDURES BEGIN ***************************
;
.RANDOM_INIT            ; Uses R0 and R1
LOADBP SEEDH_ADDR
LOADSN
SEED                            ; High seed
LOADBP SEEDL_ADDR
LOADSN
SEED                            ; Low seed
RET
;
.LOAD_NEURON                    ; Uses R0, R1, R2 and R3
READMPV NEU_ADDR ; Address of real neuron + virt (valid also for non-virtual)
LOADBP ; NSBRAM pointer to currently processed neuron
LOADSN                  ; Load Neural parameters from NSBRAM to R1 & ACC
MOVR R2                 ; Move Vmem from ACC to R2
RET
;
.MEMBRANE_DECAY  ; Uses R0, R4
MOVA R2                          ;     TEMPORARY WHILE MULS has problems. REWRITE when
it works
LDALL R4 VREST
SUB R4
LDALL R1, THAU_MEM
MULS R1                 ; Calculate decay
SHLAN 1                 ; Shift one bit left because we multiply by n-1 bits (positive value in 2's
complement)
ADD R4
MOVR R2                 ; Back to R2 where membrane potential is stored
RET
;
.ADD_NOISE ; Uses R0, R2 and R5
RANDON              ; LFSR ON
LLFSR ; Noise to ACC
MOVR R5
LDALL ACC, NOISE_MSK
AND R5
SHRN 1
RANDOFF              ; LFSR OFF. Arbitrarily here
FREEZENC
```

```
   MOVR R5
   RST ACC
   SUB R5              ; Generate signed noise without the negative bias of two's complement
UNFREEZE
   MOVSR ACC                                  ;   TO MONITOR THE NOISE
ADD R2          ; Add to Vmem
MOVR R2 ; Back to R2
RET
;
.DETECT_SPIKE          ; Uses R0 and R2
LDALL ACC, VTHRES
SUB R2                 ; Compare Vth - Vmem
SHLN 1          ;subtraction sign to C flag
RST ACC
FREEZENC        ; If positive, spike
SET ACC
LDALL R2 VREST ; Vmem to resting potential
UNFREEZE
STOREPS                ; Push spikes
RET

.STORE_NEURON ; uses R0 and R1
MOVA R2                ; Move Vmem from R2 to ACC
READMPV NEU_ADDR ; Address of real neuron + virt (valid also for non-virtual)
LOADBP ; NSBRAM pointer to currently processed neuron
STORESP  ; Store Vmem to NSBRAM
RET

; **************************** PROCEDURES END ******************************

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

; Initial instructions
GOSUB RANDOM_INIT; For noise initialization

.EXEC_LOOP ; Execution loop

;LOOP V0 ; Neuron loop for virtual operation
GOSUB LOAD_NEURON
GOSUB MEMBRANE_DECAY ; Calculate membrane potential decay
GOSUB ADD_NOISE
LOADBP SYN_ADDR ; Initial position for addresses
   LOOP tot_synapses  ; synaptic loop
;%VIRT LOOPV V0  ; synaptic loop. Reads number of current-layer synapses
      LOADSP           ; Load Synaptic parameters and spike to R1 & ACC
      SHRN 1           ; Move spike to flag C
      FREEZENC
MOVA R1 ; Synaptic parameter to ACC
;                LDALL ACC, JUMP_MV   ; Replaces previous instruction. Jumps a constant
voltage on Sj=1
         ADD R2
         MOVR R2       ; Save Neural parameter in R2
      UNFREEZE
      RST ACC
```

```
    STORESP        ; Stores synaptic parameter and increases BP for next synapse processing
  ENDL
; Compare and eventually spike
GOSUB DETECT_SPIKE
GOSUB STORE_NEURON
INCV
ENDL
NOP  ; Empty pipeline wait NOPs
NOP
NOP
SPKDIS ; Distribute spikes
GOTO EXEC_LOOP ; Execution loop
```

8.5. Example of the new high level netlist

This example is a ring oscillator that forms a 4x4 square in a high level view.

netlist_top.lst:

```
#------------------------------------------------------
#| High level netlist for HEENS processors
#| Neural circuit name: 4x4 ring oscillator
#| Description:
#------------------------------------------------------

@Config
ZC706_4x4

@Netlist
#neurons from/to chips
# From|To|Synaptical weight
# Distributing neurons

0,  1,  2500
1,  2,  2500
2,  3,  2500

3,  4,  2500
4,  5,  2500
5,  6,  2500

6,  7,  2500
7,  8,  2500
8,  9,  2500

9,  10, 2500
10, 11, 2500
11, 0,  2500

@Params
# Addr/Size/Name/Entries/default (empty for random)
.0x3E3/16/NEUR/$NVL/0, -6000
#Neuron, params
0, 0, -4000

.0x3FD/32/SEED/2/
5, 10
```

8.6. Example of the new netlist

This example is a ring oscillator that forms a 4x4 square specific for one board. Also, it is the output of the tool HLNS. netlist_top_b0.lst:

```
@Config
#This section configures the HTC for the synth of the network
Id   =     0 # Identification number of the chip
Rows =     4 # Number of rows of the processing elements
Cols =     4 # Number of columns of the processing elements
VLay =     8 # Number of virtualization layers
RegS =    16 # Register bit size of the processing element
RegN =     8 # Number of active registers on the processing element
SneS =    32 # Word size of the SN memory
SneA =  1024 # SN memory addresses length
LclS =     7 # Bit size of the LCL memory
LclA =   128 # LCL word size
InsS =    16 # Instruction size
InsA =  1024 # Instruction memory size
DlyS =     5 # Delay word size
DlyA =   256 # Delay address length
CnvS =     7 # Conversion word size
CnvA =  2048 # Conversion address length
CodS =     5 # Codification word size
CodA =   512 # Codification address length

@Netlist
# Source   |  dest  | | Registers R0, R1
#i,v, r, c, v, r, c, s, R0, R1
0, 0, 0, 0, 0, 0, 1, 1, 0, 2500
0, 0, 0, 1, 0, 0, 2, 1, 0, 2500
0, 0, 0, 2, 0, 0, 3, 1, 0, 2500
0, 0, 0, 3, 0, 0, 4, 1, 0, 2500
0, 0, 0, 4, 0, 0, 5, 2, 0, 2500
0, 0, 0, 5, 0, 0, 6, 1, 0, 2500
0, 0, 0, 6, 0, 0, 7, 2, 0, 2500
0, 0, 0, 7, 0, 0, 8, 1, 0, 2500
0, 0, 0, 8, 0, 0, 9, 1, 0, 2500
0, 0, 0, 9, 0, 0, 10, 1, 0, 2500
0, 0, 0, 10, 0, 0, 11, 1, 0, 2500
0, 0, 0, 11, 0, 0, 0, 1, 0, 2500

@Params
#Addr/Cod/Varname
.0x3E3/16/NEUR
0, -4000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0,
-6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000, 0, -6000

.0x3FD/32/SEED
640053022, 3233463630, 3184370176, 2752528616, 1457963001, 10, 4075445273, 63789517,
2770718219, 3256546841, 1511408732, 256106765, 2961578236, 3545250858, 3565134408,
1491381318
2181241349, 3583072137, 1111602253, 891597181, 3554535619, 3185413299, 818177162,
520518205, 2798381813, 1234887563, 139415931, 3980153862, 2854953035, 2936803993,
164569107, 2454684651
```

8.7. Example of the new code

This code is a functional Leaky Integrate and Fire model for the HEENS architecture in the new format.

LIF_V2.asm:

```
;;; ------------------------------------------------------------------------
;;; Integrate and fire. Both non-virtual and virtual
;;; DEFAULT operation without virtual layers

;;; keywords:
   ;; NVL  = Number of Virtual Layers
   ;; TGS  = Total number of Global Synapses
   ;; TLS  = Total number of Local Synapses
   ;; Vx   = Virtual layer number "x"
   ;; data map in netlist.lst
   .org 0x010
   .data

;;; Membrane potential parameters common to all neurons
VREST   = -7000          ; Resting potential -70 mV = -7000 in tens of of uV
VTHRES  = -5500 ; Threshold voltage -55 mV = -5500
VDEPOL  = -8000 ; Depolarization voltage -80 mV = -8000
VACT    =  1000 ; Action potential +10 mV = +1000

;;; Synapse parameters common to all neurons come here
;;; TBD
;;;
;;; Neural and Synaptic RAM addresses
                 ; THAU_MEM = 32768/(1-1/tau)
THAU_MEM  = 0x799A  ; Membrane time constant decay (inverse value). Thau = 20
NOISE_MSK = 0x001F  ; Noise mask. To be tuned

;;; Constants for debug
JUMP_MV   = 0x0100  ; Jump 2.56 mV on spike
LFSR_VAL  = 0xAAAA
LFSR_VAL2 = 0x5555

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

   .code

GOTO MAIN         ; Jump to main program
;;;
;;; **************************** PROCEDURES BEGIN ****************************
;;;
RANDOM_INIT:          ; Uses R0 and R1
LOADBP  SEED_0
LOADSN
SEED             ; High seed
LOADBP  SEED_1
LOADSN
SEED             ; Low seed
```

```
   RET

LOAD_NEURON:           ; Uses R0, R1, R2 and R3
READMPV NEUR_0      ; Address of real neuron + virt
                        ; (valid also for non-virtual)
LOADBP              ; NSBRAM pointer to currently processed neuron
LOADSN                                ; Load Neural parameters from NSBRAM to R1 & ACC
MOVR    R2              ; Move Vmem from ACC to R2
   RET

MEMBRANE_DECAY:        ; Uses R0, R4
MOVA    R2          ; TEMPORARY WHILE MULS has problems.
                        ; REWRITE when it works
LDALL   R4, VREST
SUB     R4
LDALL   R1, THAU_MEM
MULS    R1                      ; Calculate decay
SHLAN   1                       ; Shift one bit left because we multiply by n-1 bits
                        ; (positive value in 2's complement)
ADD     R4
MOVR    R2                      ; Back to R2 where membrane potential is stored
   RET

ADD_NOISE:             ; Uses R0, R2 and R5
RANDON              ; LFSR ON
LLFSR               ; Noise to ACC
MOVR    R5
LDALL   ACC, NOISE_MSK
AND     R5
SHRN    1
RANDOFF             ; LFSR OFF. Arbitrarily here
FREEZENC
   MOVR    R5
   RST     ACC
   SUB     R5      ; Generate signed noise without the negative bias of two's complement
UNFREEZE
   MOVSR   ACC        ; TO MONITOR THE NOISE
ADD     R2          ; Add to Vmem
MOVR    R2          ; Back to R2
   RET

DETECT_SPIKE:          ; Uses R0 and R2
LDALL   ACC, VTHRES
SUB     R2          ; Compare Vth - Vmem
SHLN    1           ;subtraction sign to C flag
RST     ACC
FREEZENC            ; If positive, spike
SET     ACC
LDALL   R2, VREST ; Vmem to resting potential
UNFREEZE
STOREPS                         ; Push spikes
   RET

STORE_NEURON:          ; uses R0 and R1
MOVA    R2          ; Move Vmem from R2 to ACC
```

```
READMPV NEU_ADDR    ; Address of real neuron + virt
                    ; (valid also for non-virtual)
LOADBP              ; NSBRAM pointer to currently processed neuron
STORESP             ; Store Vmem to NSBRAM
   RET

;;; **************************** PROCEDURES END ******************************

;;; **************************** MAIN PROGRAMME BEGIN ***********************
MAIN:

;;; Initial instructions
   GOSUB RANDOM_INIT        ; For noise initialization

EXEC_LOOP:           ; Execution loop

   LOOP    NVL       ; Neuron loop for virtual operation
      GOSUB   LOAD_NEURON
      GOSUB   MEMBRANE_DECAY ; Calculate membrane potential decay
      GOSUB   ADD_NOISE
      LOADBP  SYN_ADDR    ; Initial position for addresses
      LOOP    V0         ; synaptic loop
         LOADSP                  ; Load Synaptic parameters and spike to R1 & ACC
         SHRN    1               ; Move spike to flag C
         FREEZENC
            MOVA    R1  ; Synaptic parameter to ACC
            LDALL   ACC, JUMP_MV  ; Replaces previous instruction. Jumps a constant voltage on
Sj=1
            ADD     R2
            MOVR    R2   ; Save Neural parameter in R2
         UNFREEZE
         RST     ACC
         STORESP                 ; Stores synaptic parameter and increases BP for next synapse
processing
         ENDL

;;; Compare and eventually spike
      GOSUB DETECT_SPIKE
      GOSUB STORE_NEURON
      INCV
      ENDL
   NOP              ; Empty pipeline wait NOPs
   NOP
   NOP
   SPKDIS            ; Distribute spikes
   GOTO    EXEC_LOOP   ; Execution loop
```