POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Elettronica

Master's Degree Thesis

Design and implementation of a Convolution event-based neural network with Offline Learning

Supervisor
prof. Maurizio Martina

Candidate
Luigi Massari

Academic year 2020 - 2021

Contents

List of Figures

List of Tables

Abstract

Nowadays, Convolutional Neural Networks (CNNs) are exploited to solve different tasks, but their increasing complexity means an increase in the power consumption of these architectures which limits their applications.
The introduction of Spiking Neural Networks (SNNs) is an important step to overcome this limit. They work in the same way as the behaviour of our brain and they are organized in layers of biological neurons, which receives spikes, elaborate them and solve the task. Spikes mean a reduction of the complexity of the operation, due to the substitution of the Multiply and Accumulate operation with a simple Select and Accumulate, which is traduced into a reduction of the computational power.
The work is focused on the implementation VHDL of a convolution event-based neural network with offline learning based on a script PyTorch, used to recognize handwritten digits based on MNIST dataset. The architecture is organized with a convolutional layer, which extracts the features of the input, a max pooling layer, which reduces the noise and the image dimension, and two layers of Izhikevich Neuron used to classify the digits from 0 to 9.
To train the architecture, a PyTorch script has been written. To describe an event-drive convolutional neural network, PyTorch is extended with Norse library. First, the architecture is tested with this software script, in order to train it and to find weights of the fully connected layer and kernel of the convolutional one and then, with the obtained values, the hardware has been tested in order to verify the correctness of the described architecture.

1. Introduction

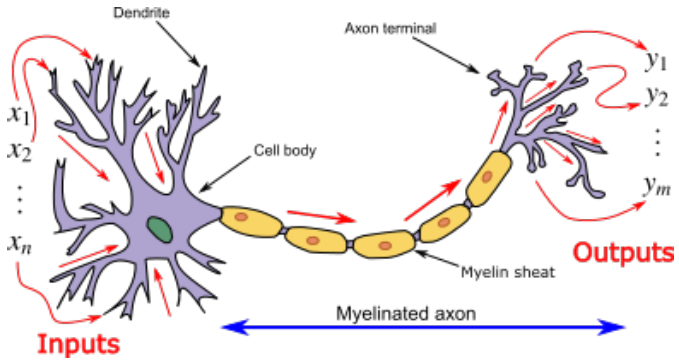


Figure 1.1: *Biological example of a neuron [Figure from Wikipedia]*

A Neural Network (NN) is a hardware, software or mathematical computational model made up of artificial neurons interconnected with each other. The neuron model is realized starting from how the neurons in our brain work. Each neuron processes a certain received signal and transmits it to subsequent nodes.
These signals are real numbers and each neuron calculates a non-linear function of the sum of its inputs. The connections are called synapses and have weights that are regulated by a learning process.
The tasks of neural networks are varied. Here are presented some of the most important applications:

- pattern recognition (image recognition, object recognition, face identification...);

- general game playing;
- system control (vehicle control, trajectory prediction);
- medical diagnosis.

In literature, it is possible to find different types of neural networks: from the simplest Artificial Neural Networks (ANNs), which include only neurons and weights, to the Convolutional Neural Networks (CNNs), analyzed in details in section 2.1, which implement the convolution operation to extract information, up to the modern Spiking Neural Networks, described in section 2.2, that work with trains of spikes in order to reduce the power consumption.
To try to exploit the advantages of both classes of neural networks, it was implemented a CEDNN, stand for Convolution Event-driven Neural Network, mainly experienced for handwritten digits. As explained in the following chapters, the architecture is implemented at the HW level in VHDL and each component is also tested through SW scripts, generally in C.
The advantage of combining the convolution operation with the spikes is that the convolution operation, which is expensive at the HW level due to the multipliers present to carry out the operation, is simplified because the spikes amplitude is always unitary, therefore the multiplier operator is simply reduced to a multiplexer.
The excellent result of the architecture can also be seen from the offline training, carried out through a PyTorch script, which also reaches values of accuracy of 94% with 10 epochs and 96%, with only 20 epochs.
In the following chapters the whole implementation of the architecture is described step by step, both as regard to the SW side and the HW side.
All the VHDL and PyTorch code are available at the Google Drive repository

`https://drive.google.com/drive/folders/`
`1UJXRC-tboHmG7j7Mrd6joLpYhxhRds9v?usp=sharing`

2. Convolution Neural Network and Spiking Neural Network

2.1 Convolution Neural Network

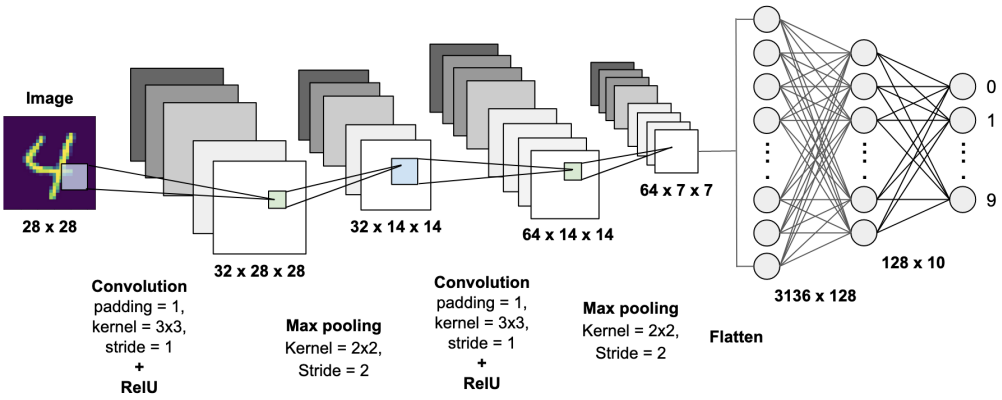


Figure 2.1: *Schematic representation of a Convolution Neural Network*

Convolution Neural network is a class of Neural Network mainly used for image and video recognition and image classification. Its name derives from the principal operation that is done in this architecture: the convolution. An example of this architecture is represented in the Figure 2.1. As it is possible to notice from the Figure 2.1, the basic building blocks of a CNN are:

- Convolutional Layer

- Pooling Layer
- Activation function Layer
- Fully Connected Layer

2.1.1 Convolutional Layer

Convolution is an operation where the input, that can be consider as a 2D grey-scale image with dimension ($height_{image} \times width_{image}$) after passing the layer, becomes abstracted to a feature map. In some most elaborated cases, the image can be considered as a 3D image, where, in addition to height and width, we consider another input channel that corresponds to the characteristic of colour.
Each convolution layer has its own characteristics:

- a set learnable kernels $K \times K$ defined by its hyper-parameters
- the number of input channels and output channels
- the hyperparameters like padding and stride size.

Mathematically, to described the convolution operation between two matrices, it is necessary firstly to define what padding and stride size is:

- stride indicates the number of steps we are moving in each step of the convolution operation;
- padding is a process of adding zeros to the input matrix symmetrically: it is used because at the output of the convolution operation, the size of output becomes smaller that input.

Then, to do an example, we can consider two matrices:

$$K(i,j);\ 0 \leq (i,j) \leq 4 \tag{2.1}$$

as a 5x5 convolution kernel and

$$I(z,q);\ 0 \leq (z,q) \leq 31 \tag{2.2}$$

as the 32x32 input matrix, which represents the input image.
If we consider $stride = 1$ and $padding = 0$, at the output we have a matrix

$$O(d,p);\ 0 \leq (d,p) \leq 28 \tag{2.3}$$

The output dimension is determined by the following formula:

$$[(weight - size_{Kernel} + 2 \times padding)/stride] + 1 = \tag{2.4}$$

$$= [(32 - 5 + 2 \times 0)/1] + 1 = 28 \tag{2.5}$$

Each location of the O matrix is calculated with the following formula:

$$O(d,p) = \sum_{(i,j)=0}^{K-1} K(i,j) * I(d+i, p+j) \tag{2.6}$$

2.1.2 Pooling Layer

The idea of the Pooling layer is to reduce the power required to process the data through a dimensional reduction.
In literature, there are two types of Pooling: Average Pooling and Max Pooling. Average Pooling returns the average of all the values from the portion of the output matrix of the convolutional layer. Instead, Max Pooling returns the maximum value from the portion of the output matrix.

2.1.3 Fully connected layer

After the Pooling Layer, there is a defined number of Fully Connected Layers which connect each neuron of a layer to all neurons of the next layer through

different weighted connections called synapses. The flattened matrix, obtained from the Pooling Layer, goes through the fully connected layers to classify the images.
Before the layer of neurons, a layer of linear activation function can be present: it is a linear function that maps the weighted inputs before the neuron elaboration. The most common used activation functions are:

- ReLU function
$$f(x) = max(0, x) \tag{2.7}$$

- Hard Sigmoid function
$$f(x) = max(0, min(1, \frac{x+1}{2}) \tag{2.8}$$

- Hard Sigmoid function
$$f(x) = \begin{cases} 1 \text{ if } x > 1 \\ -1 \text{ if } x < 1 \\ x \text{ otherwise} \end{cases} \tag{2.9}$$

2.2 Spiking Neural Network

Spiking Neural Networks represent a fundamental innovation in the world of Neural Network. They are a special class of Artificial Neural Networks where neurons exchange information via spikes, so they incorporate the concept of time in addition to neuron and synapse that characterized the NN.
It is possible to summarize the behaviour of a spiking neural network as it is described in the Figure 2.2.

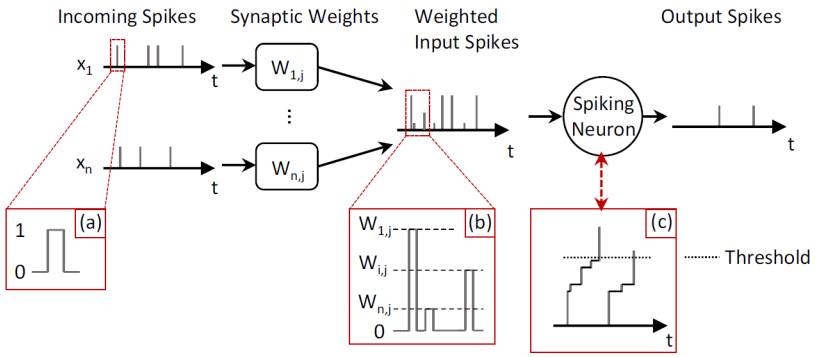


Figure 2.2: *Schematic representation of spiking neuron [Image from [1]]*

When spikes arrives from its pre-layer neurons into the synapse, which is the weighted connection between the two layers, they will be multiplied by the synaptic weight and they are summed up each other; this quantity goes into the neuron, which makes its elaboration, and the output quantity represents the membrane potential. If the potential overcomes a threshold, the neuron fires, so it emits a spike.

2.2.1 Spike Encoding

In SNN, information is encoded into spikes: in literature, we can find two types of encoding methods, rate or time coding. In Rate coding, the information is encoded by the number of spikes per second (frequency of the spike train will be proportional to intensity) while in the Time encoding, the information is encoded in the time of arrival of a spike (the time is inversely proportional to the pixel's intensity). The Figure 2.3 represents an example on how three pixels of a grey scale image are coded into spike with the two different techniques.

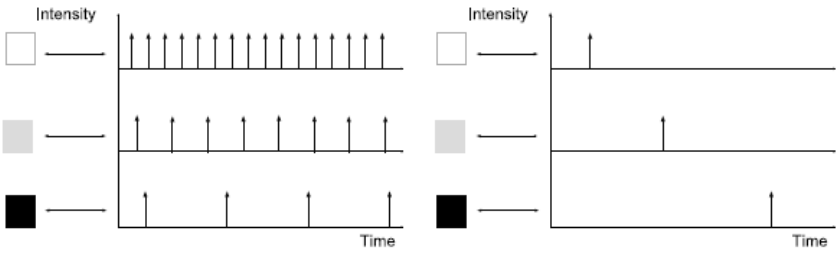


Figure 2.3: *Example of a Rate coding technique (on the left) and a Time coding technique (on the right) [Image from [2]]*

2.2.2 Type of Neurons

In literature, there are different types of neurons: from simpler models, as the IF (Integrate and Fire) neuron represented into the Figure 2.2, to more complex models as the Hodgkin-Huxkley.
In the list below, it is reported a small description of some of the most used neurons in literature:

- IF neuron (Integrate-and-Fire): this is the simplest model of neuron, in which the spikes are only multiplied for their weight and integrated. If the integrated value, which corresponds to the membrane potential, overcomes the threshold, the IF neuron fires;

- LIF neuron (Leaky-integrate-and-Fire): this model is very similar to the previous one, but in this case the membrane potential decreases continuously due to the leak between two input spikes;

- Hodgkin-Huxkley neuron: it is a biological plausible mathematical model that describes the process of decreasing the absolute value of the membrane potential of a neuron. This model is described by a set of 10 equations, where 4 of them are nonlinear differential equations, that approximate the electrical characteristics of the neuron;

- Izhikevich neuron: this model is a simplification of the Hodgkin-Huxkley one, because the neuron is described by a set of 2 nonlinear differential equation. So it combines the biologically plausibility of Hodgkin-Huxley model and the computational efficiency of IF neuron.

2.2.3 Advantages

Neural networks are constantly evolving and the most important trend is to reduce computational operation in order to maintain a good degree of neural network complexity without increasing too much the energy consumption. The most important advantage of the Spiking Neural Network is that the Multiply and Accumulate operation of the CNN will be substituted by Comparison and Accumulate operation of the SNN, which results more efficient in term of hardware required and power consumption.

2.2.4 Training of a Spiking Neural network

The most used technique to train the neural network is the back-propagation of the gradient, which results not to be feasible in the SNN due to the non-continuity of the equation of spiking neurons.
So in literature, to train SNN, different solutions are present:

- a conversion of a trained ANN into a SNN;
- an unsupervised learning technique, as the Spike Timing Dependent Plasticity (STDP), where the weight update depends only on the relative timings of pre- and post- synaptic neuron spike;
- a supervised learning technique, as the ReSUme and Chronotron.

It is also important to underline the difference between on-chip and off-chip learning. On-chip learning includes a dedicated hardware to train the neural network before using it; while the off-chip learning is done by other

architecture, in general software solution. If the architecture is a general accelerator for machine learning, the learning of the neural network must be on-chip to adapt to the different situation. On the other hand if the purpose is to perform a unique machine learning task on embedded low power hardware, an off-chip learning can be a good solution because power consumption is reduced respect to an on-chip learning.

2.3 SPOON architecture

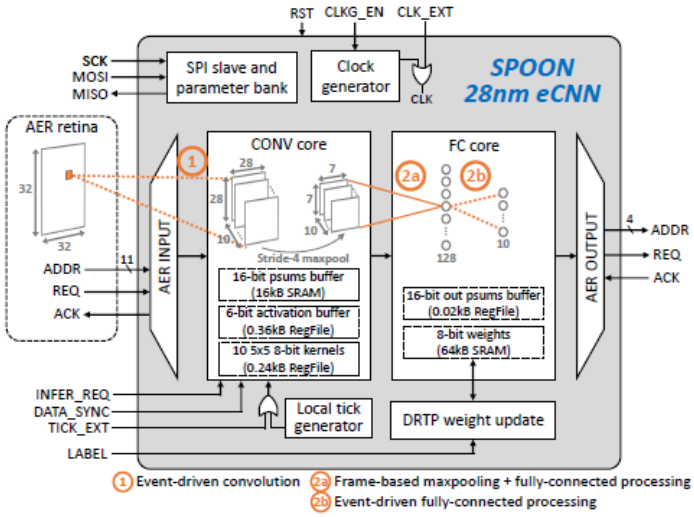


Figure 2.4: *Building blocks of the SPOON Architecture [3]*

In literature, there are a lot of architectures which combine the idea of spiking and convolutional neural network in order to exploit the advantage of the operation of the convolution with the advantage of low computing power of the spiking neuron. An example with very good results, that it has been analyzed in detail, is the SPOON (spiking online-learning convolutional neuromorphic processor) architecture [3].
The architecture, represented in the Figure 2.4, is an event-driven CNN for adaptive edge computing with an online learning structure. The architecture

is trained and tested with the MNIST dataset, described in the section 4.1, and with the N-MNIST, which is a DVS camera derivation of MNIST. It is composed by a convolution layer, which receives the data input from a FIFO, a max pooling layer and two layers of neurons, a hidden layer composed by 128 neurons and an output layer composed of 10 neurons, which have respectively a hardtanh function layer and a hardsigmoid function layer as activation functions. In the Table 2.1, there is a resume of the most important characteristics of the SPOON architecture.

Topology	C5×5@10 − $FC128 - FC10$
Max clock frequency	$150MHz$
Online-learning technique	Stochastic DRTP, 8bit weigths
Offline-learning	PyTorch script
Accuracy Online-learning, MNIST	92.8%
Accuracy Online-learning, N-MNIST	93.8%
Accuracy offline-learning, MNIST	97.5%

Table 2.1: *Most important characteristic of SPOON architecture*

3. Description of the architecture

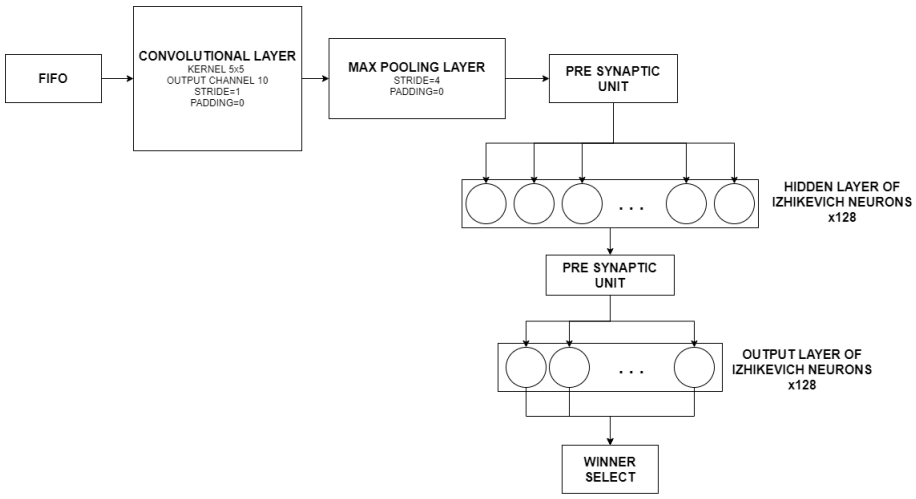


Figure 3.1: *Building block of the implemented architecture*

The idea of this thesis work is to implement CEDNN (standing for Convolutional Event Driven Neural Network), which is an architecture similar to the SPOON one, described in section 2.3, but using a more complex neuron: the *Izhikevich neuron*.

Hence, the chosen architecture is organized with an event-driven convolution layer, which receives the input data, in the form of spikes, from a FIFO, and a Fully Connected Layer, composed by two layers of 128 and 10 Izhikevich Neurons. The most important building blocks of the implemented architecture are represented into the Figure 3.1.

At the beginning of the work, it has been studied the input data. It has been chosen a Spike Latency encoding, therefore the intensity information is encoded into the arrival time of the spike. More information regarding this encoding method is reported in chapter 4.
The architecture receives a 10-bit address, which covers the coordinates x, y for images 32×32, concatenated to one bit which represents the polarity *ON/OFF*.
Since Izhikevich neurons are real biological models that depend on time and that are realized through the differential equations d/dv, it was necessary to choose an integration time T and an integration step dt. It has been decided to use $T = 100$ and $dt = 1\ ms$. The first value is due to the fact that it is possible to reach optimal accuracy values through the offline script with this value; the second one derives from the timing on which the spikes are generated.

3.1 Convolutional and Max Pooling Layer

The Convolutional layer is organized with a 10 5×5 programmable Kernels with stride-1 and padding-0, followed by a Max Pooling Layer with stride-4 and padding-0. It receives the input data, organized as described before, and elaborates them.
Consequently, as explained in section 2.1, at the output of the convolutional layer there will be 10 matrices 28x28 and at the output of the max pooling layer there will be 10 matrices 7x7.
Firstly, to implement the Convolutional Layer, it has been written a C code: it is used to understand how the algorithm of the event-driven convolution operation and of the max pooling operation work, to test the HW implementation and to compare the results. A part of this code is reported in the Appendix A.
The code is structured with an infinite *while loop*, which at each cycle it

requires to insert the input data with the following organization:

$$polarity, x_{address}, y_{address} \tag{3.1}$$

and then it computes the convolution and max pooling operations. The code works with two matrices: a Convolutional Matrix 28×28, a Kernel Matrix 5×5 and it is organized in three main parts:

- the first is the multiplication of the polarity bit for the content of the kernel matrix. Each obtained value is placed in a product matrix in a overturned position with respect to both axes;
- the second is two *for loops*, where the product matrix, calculated into the previous part, will be summed one-to-one with a sub-section of the convolutional matrix. The two matrices are summed putting the point $(4, 4)$ of the product matrix in the position $(x_{address}, y_{address})$, received from the input, of the convolution matrix. If there are points which exceed the dimension of the convolution matrix, the correspond values will be discarded;
- the third part is contained into the second *for loop* of the previous part and and it is organized with two other *for loops* used to perform the Max Pooling operation and to store them in another matrix;

In the Figures 3.2, 3.3, 3.4, 3.5 are reported the output of the different steps of the C Code. The same values used in this simulation will be used to test the VHDL code.

```
KERNEL:
-87 -93 62 4 97
-20 86 46 -46 16
-55 113 113 59 105
107 51 38 91 -68
7 -116 -66 25 -92
```

Figure 3.2: *First of all, the C Code generate a random matrix which contains the Kernel*

```
Insert x coordinate: 24
Insert y coordinate: 8
```

Figure 3.3: *Secondly, it asks to the user to insert the input*

```
-87 (8 24) - (248)
-93 (8 23) - (247)
62 (8 22) - (246)
4 (8 21) - (245)
97 (8 20) - (244)
-20 (7 24) - (220)
86 (7 23) - (219)
46 (7 22) - (218)
-46 (7 21) - (217)
16 (7 20) - (216)
-55 (6 24) - (192)
113 (6 23) - (191)
113 (6 22) - (190)
59 (6 21) - (189)
105 (6 20) - (188)
107 (5 24) - (164)
51 (5 23) - (163)
38 (5 22) - (162)
91 (5 21) - (161)
-68 (5 20) - (160)
7 (4 24) - (136)
-116 (4 23) - (135)
-66 (4 22) - (134)
25 (4 21) - (133)
-92 (4 20) - (132)
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -92 25 -66 -116 7 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -68 91 38 51 107 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 105 59 113 113 -55 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16 -46 46 86 -20 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 97 4 62 -93 -87 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 3.4: *Then, the code reports to the output the results of the convolution operation (on the left) with the following notations: value to store, x address of the Matrix, y address of the Matrix, Convolution Register File Address. After the previous operation, the code prints the Convolutional Matrix (on the right) after the computation*

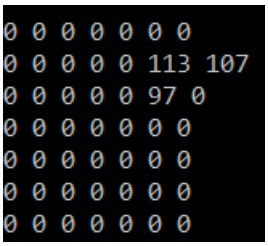


Figure 3.5: *At the end, the code reports to the output the result Max Pooling Matrix and the principal loop restarts*

HW implementation

At hardware level, the Convolutional layer is organized with 10 identical "pages". Each page, which is represented in details in the Figure 3.6, computes the convolution operation between the input data and one kernel matrix and it stores the result into a "personal" register file, called Page Convolution Register File. Therefore, the following description is referred to the single page of the Convolutional Layer.
The input data, which comes from the last FIFO memory location, is broken down into the polarity, which goes to the Convolutional Core, and the x, y addresses, which goes to the Address Decode Block.
The Convolutional Core contains one multiplexer: polarity is the selector of the multiplexer, while the two inputs are a sequence of $'0'$ and the output of the Kernel Register File which, through a multiplexer driven by a counter, sends a single kernel location to the multiplier. This operation emulates a multiplication between the input and the data present in the Kernel matrix. Hence, the product between polarity and kernel is stored in an accumulation register.
At the same moment, the Address Decoder Block receives the x, y addresses and computes the location of the Page Convolution Register File where the data should be stored.
The content of the accumulation register and the content of the Page Convolution Register File at the location calculated with the Address Decoder

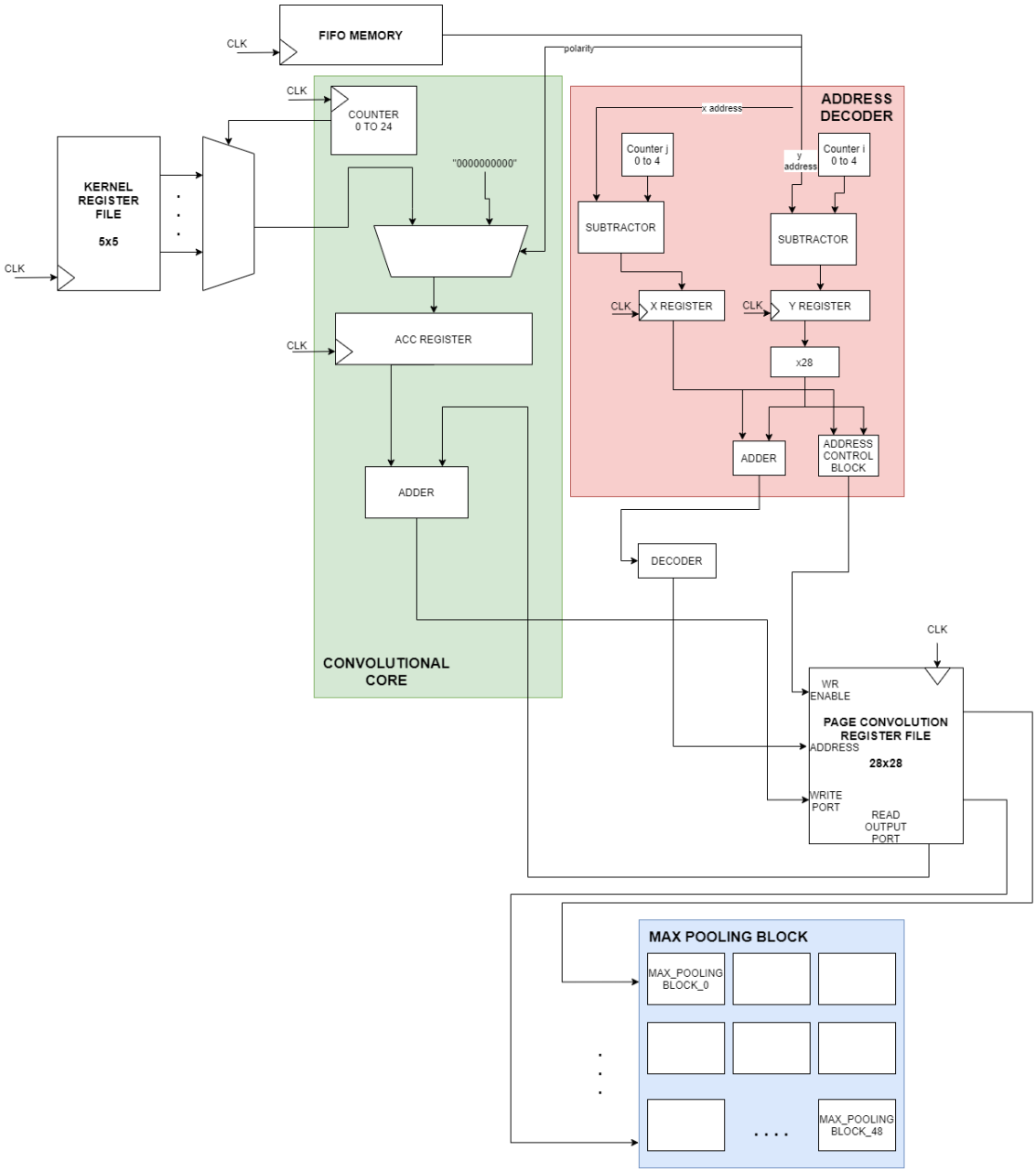


Figure 3.6: *Datapath scheme of a single page of the Convolutional Layer*

block, go in the input of an adder which computes the sum and finally the result is stored in the same location of the Page Convolution Register File. This operation is done for each location of the Kernel Register File and for each input received.
All these operations are controlled by a Finite State Machine, which is reported in Appendix B.
After the Convolution Layer, there is the Max Pooling Layer, which receives the output of the 10 Convolution Registers File and, with a combinatorial block, computes the Max Pooling operation. Then, this matrix is flattened and stored into the Max Pooling Register File.
The Max Pooling Layer is a set of 49 comparators, for each page, organized to compare in the correct way, the different location of the Convolution Registers File following the specification of $stride = 4$.

HW Simulation

In order to verify the correctness of the architecture, it has been realized a testbench for the single page Convolutional Layer to simulate on Modelsim and to compare the results with the C code.
For the Kernel Matrix values, it has been used the random values generated from the C code; also the polarity and the x, y address has been chosen randomly.
These values will be used both in the C code simulation and in the VHDL Simulation. The results of the C code simulation is reported in the Figures 3.2, 3.3, 3.4, 3.5; while, in the Figure 3.7, there is the Modelsim simulation.

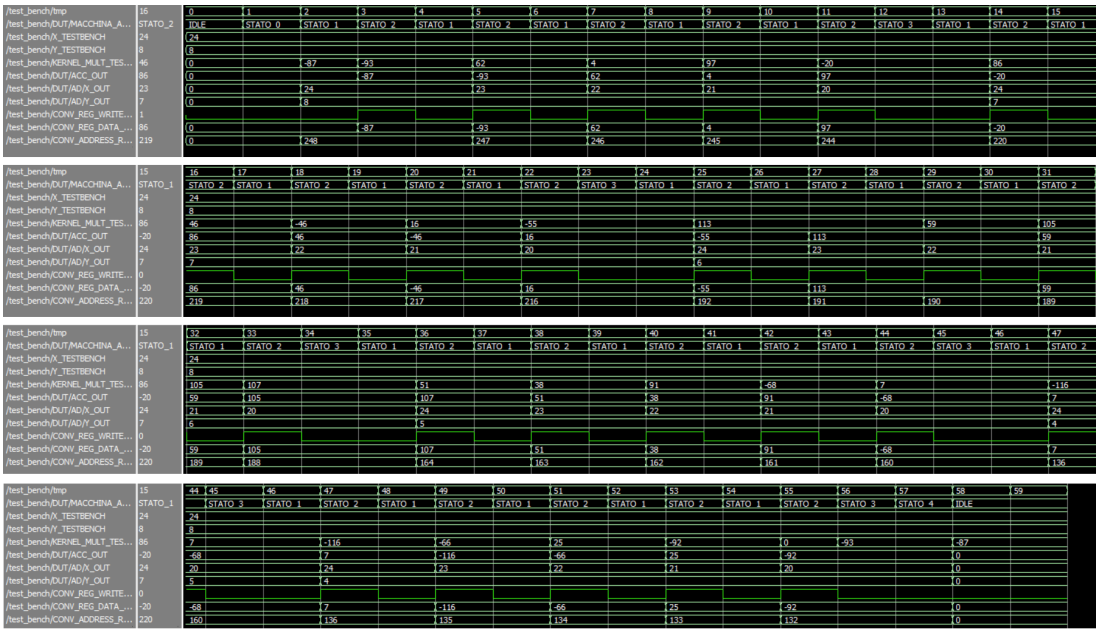


Figure 3.7: *Modelsim simulation of the behaviour of the Convolution Layer*

It is possible to notice that the values, which should be stored into the Convolutional Layer, are the same as the C code so the elaboration is correct. It can be also observed that, to elaborate a single input, the Convolutional Layer requires 58 clock cycles.

Total Convolutional Layer

As soon as the implementation of the single page of the Convolutional Layer has been completed, it has been realized the complete Convolutional Layer. The block is made up of the 10 identical pages of Convolutional Layer and the Max Pooling Block. In order to test the complete block, it has been realized a testbench which simulates the behavior of the FIFO memory, thus sending the data to be processed to the block with the appropriate notation. To simplify the simulation and verification phase, the same Kernel matrix was used for all pages of the Convolutional Layer and it is used the same random matrix of the Figure 3.2.

The inputs chosen are shown in the Figure 3.8.

Figure 3.8: *Input used for the simulation of the Convolutional and Max Pooling Layer*

To compare the correctness of the results, it has been used the same input of the VHDL code on the C Code. Finally the final Max Pooling Matrices produced by the two codes will be compared.
The output matrix of a single page of the Convolutional Layer is reported into the Figure 3.9, while the output of the Modelsim simulation, with the output of the Max Pooling Matrix, is reported Appendix C.

```
0 0 0 0 25 7 0
0 0 0 0 272 313 0
0 0 0 221 289 0 0
0 0 0 585 375 0 0
0 0 302 476 0 0 0
0 0 172 225 0 0 0
0 0 0 0 0 0 0
```

Figure 3.9: *Output matrix after Convolutional and Max Pooling operations*

3.2 Fully Connected Layer

As previously stated, the Fully Connected Layer is composed of two layers of 128 and 10 Izhikevich neurons.
Before studying at HW level how implement the Fully Connected layer and the neurons, it has been studied the behaviour of the Izhikevich neuron to understand its characteristics, its parameters and its functioning.

3.2.1 Izhikevich Neuron

Theory and Mathematical model of Izhikevich neuron

Izhikevich neuron model [4] is a simple spiking model which combines the biologically plausible of the Hodgkin-Huxley model with the computationally efficiency of the integrate-and-fire model.
The Izhikevich neuron model consists of two differential equations:

$$\begin{cases} v' = 0.04v^2 + 5v + 140 - u + I \\ u' = a(bv - u) \end{cases} \tag{3.2}$$

with the after-spike reset condition:

$$\text{if } v > 30mV, \text{ then } \begin{cases} v = c \\ u = u + d \end{cases} \tag{3.3}$$

The two equations in 3.2 are differential equations which represent respectively the variation of the membrane voltage v and the recovery function u respect to the time t. The recovery function u is a negative feedback to v which provides a better stability to v .
The value of I represents the input current, therefore the output of the previous neurons layers is contained in this variable.
In the equations, the parameter a, b, c and d represents the possible different behaviours of the firing pattern:

- the parameter a describes the time scale of the recovery function u
- the parameter b describes the sensitivity of the recovery function u to the sub-threshold fluctuations of the membrane potential v
- the parameter c describes the after-spike reset value of v
- the parameter d describes the after-spike reset value of u

An example of the different behaviour of output spike, with a constant current $I = 10$, changing the value of the parameters a, b, c and d is represented in Figure 3.10.

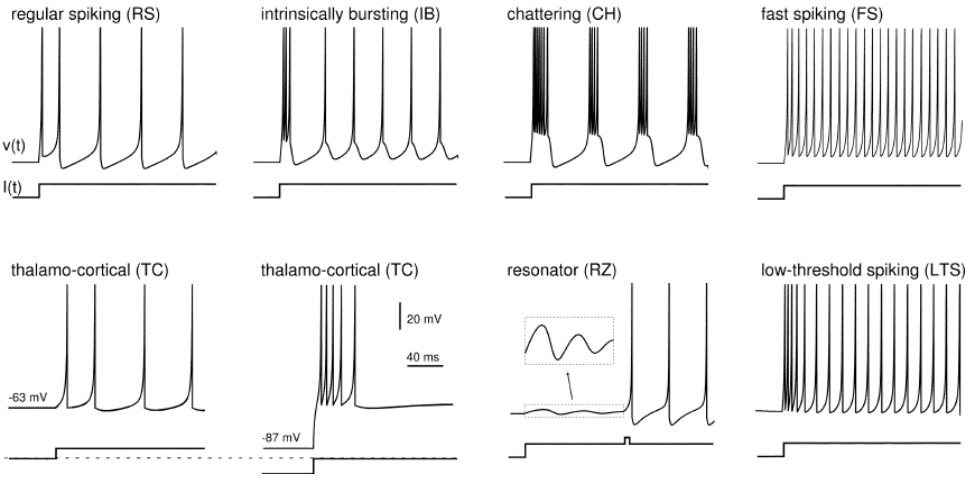


Figure 3.10: *Example of different spike firing of an Izhikevich Neuron, changing the value of a, b, c and d, with a current $I = 10$ [Figure from [4]]*

To solve the differential equations of Izhikevich model, it has been used the forward Euler integration method. The equations in 3.2 become:

$$\begin{cases} v(t+dt) = v(t) + dt(0.04v^2 + 5v + 140 - u + I) \\ u(t+dt) = u(t) + dt[a(bv(t) - u(t))] \end{cases} \tag{3.4}$$

Matlab implementation

The equation in 3.4 is composed by sum and multiplication: the second operation is too much expensive to implement in HW and so, it has been substituted the multiplication of the equation with shift operation.
In order to evaluate the error, it has been realized a Matlab script to compare the difference between the real model and the approximate one.
Due to the normalization used in PyTorch script described in chapter 5, all

operations and constants of the neurons are divided by 10.
The model implemented in Matlab is reported in 3.5 and 3.6.

$$\begin{cases} v(n+1) = v(n) + dt[2^{-8}v(n)^2 + 2^{-1}v + 14 - u(n) + I] \\ u(n+1) = u(n) + 2^{-9}dt[(2^{-6} + 2^{-7})v(n) - u(n)] \end{cases} \tag{3.5}$$

$$\text{if } v(n) > 3, \text{ then } \begin{cases} v = 6.5 \\ u = u + 0.6 \end{cases} \tag{3.6}$$

In the Figure 3.11, it has been reported a simulation of the approximated model with an input current of 10.
Firstly, it has been analyzed the amplitude error: therefore, it has been evaluated the difference between the trends of voltage in the approximated mode and in the real model. In the Figure 3.12, it has been reported the output graph, which show the trend of the difference.

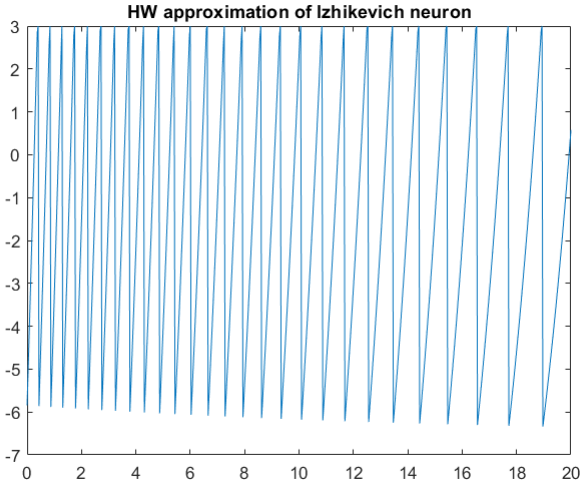


Figure 3.11: *HW approximated model of Izhikevich Neuron reported on Matlab*

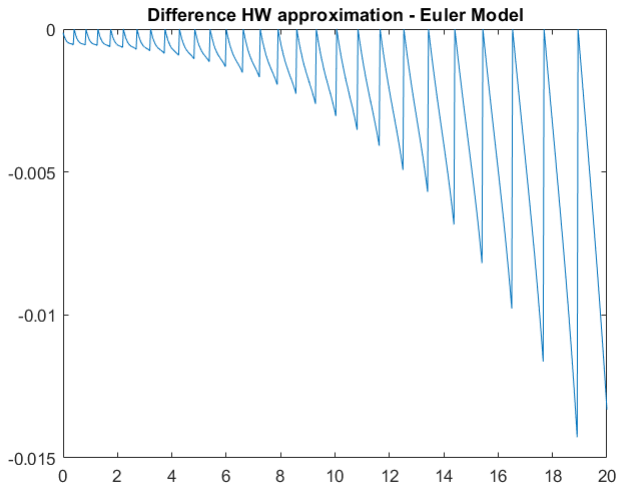


Figure 3.12: *Difference in amplitude between HW approximation and real model*

Secondly, it has been evaluated the value of the Mean Relative Error (MRE). As it is explained in [5], a timing evaluation is necessary due to the fact that the information is encoding in the timing of the spikes. The MRE is defined as:

$$MRE\% = \frac{\sum_{i=1}^{n} \frac{t_{APPROXIMATED} - t_{real}}{t_{real}}}{n} \times 100\% \tag{3.7}$$

The great advantage is that with the approximation introduced in 3.5, the MRE is zero and therefore there is no time difference between the real model and the approximate one, but only a slight difference in amplitude.

DFG of the neuron operation

In order to understand which are and how many the arithmetic operators are to be used in the HW implementation of the neuron, the Data Flow Graph (DFG) was created for both the voltage and the recovery function. The DFG of the voltage function is reported in Figure 3.13, while the recovery function one in Figure 3.14.

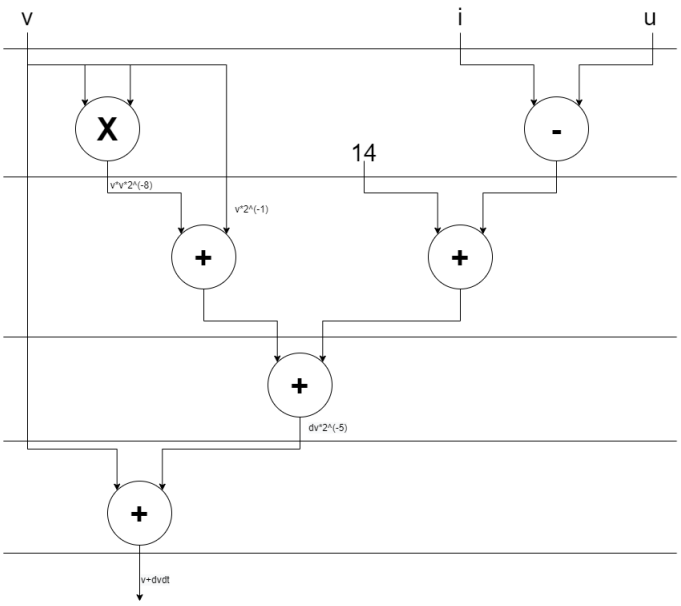


Figure 3.13: *Data Flow Graph of the voltage function of the Izhikevich Neuron*

How we can notice, with this configuration, it is possible to have the result in 4 clock cycle using:

- a multiplier, necessary to obtain the square of the voltage;
- two adders/subtractors, used for processing the voltage value;
- one adder/subtractor, used for processing the recovery value
- a series of shift operators used, as explained in the previous paragraph, to replace multiplications

HW implementation

The datapath simply represents a hardware translation of the operations present in the DFG and it is reported in Figure 3.15. It is composed by a voltage core, which compute the voltage update, with two adders and a

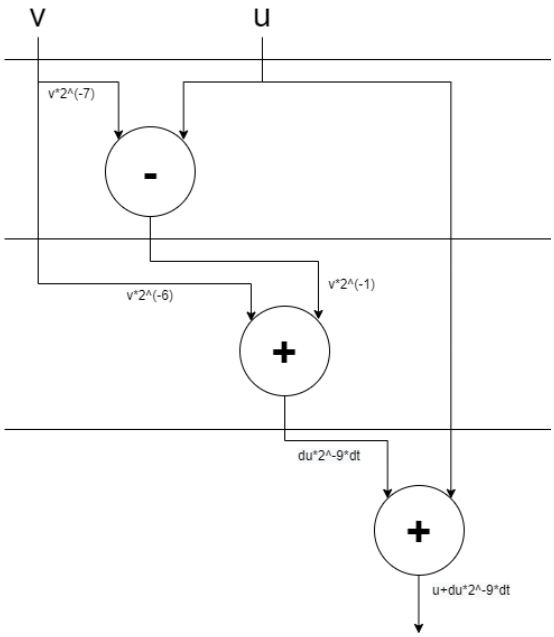


Figure 3.14: *Data Flow Graph of the recovery function of the Izhikevich Neuron*

multiplier, and a recovery core, which compute the recovery one with an adder.

In order to implement the HW architecture, it has been studied also the parallelism of the data: it has been chosen a floating point representation with 12 bit after the point, to have a precision of 2.44×10^{-4}. All the constants have therefore been converted using this format. Through the hardware simulation it was noticed that the error introduced due to the chosen notation does not lead to a deterioration in performance.Then, this notation will be applied to all blocks of the architecture.

To control all the signals of the neuron, a state machine has been implemented which is reported in Appendix D.

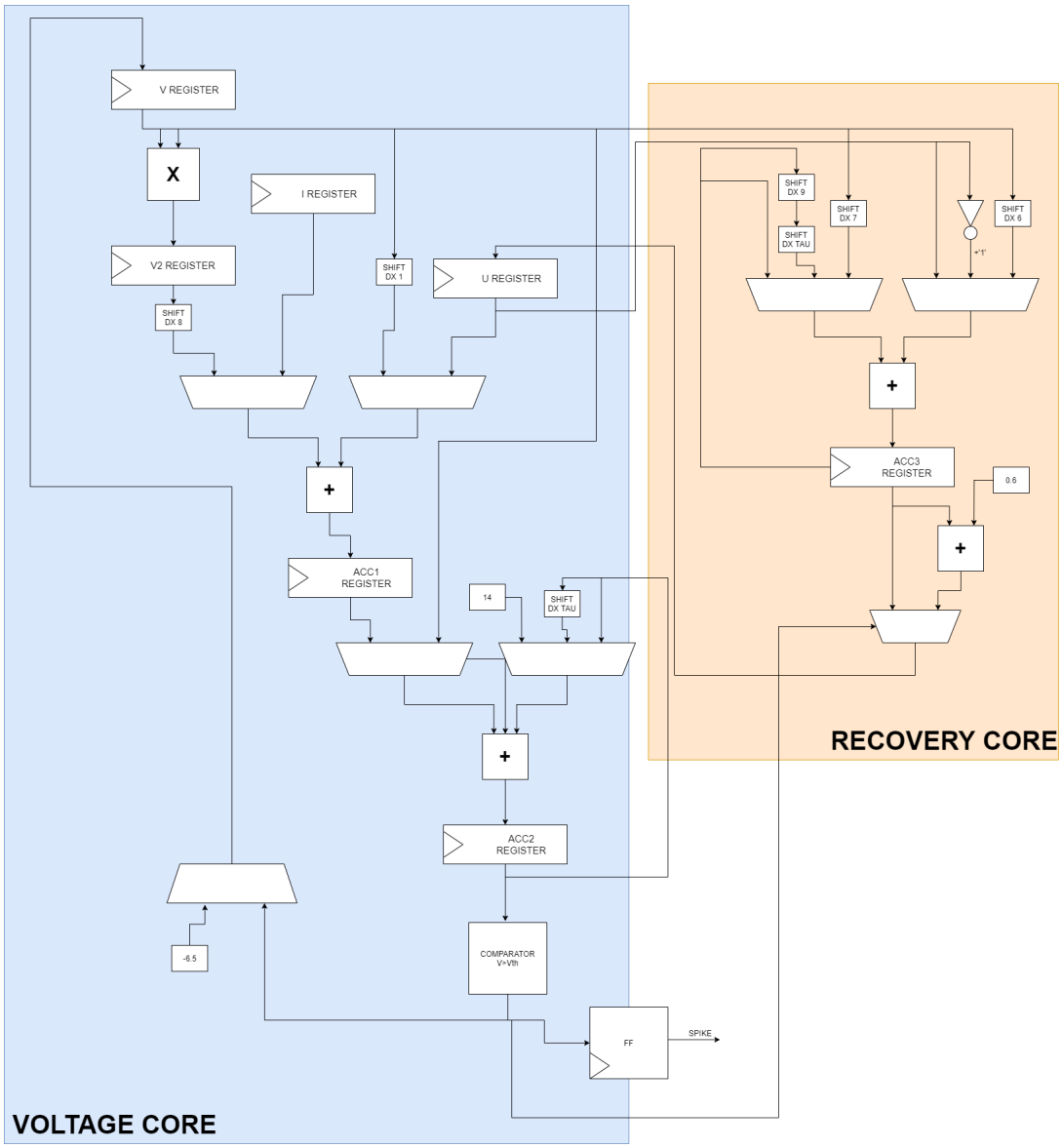


Figure 3.15: *Datapath scheme of the Izhikevich Neuron*

Modelsim Simulation of the Neuron

In order to simulate the behaviour of the neuron, it has been realized a VHDL testbench and the architecture is simulated with Modelsim, using a constant

current $I = 10$. In the Figures 3.16 and 3.17, it has been reported a piece of the overall simulation: in 3.16, it has been represent the first cycle of the neuron iteration, while in 3.17 it has been reported the cycle where a spike is emitted.

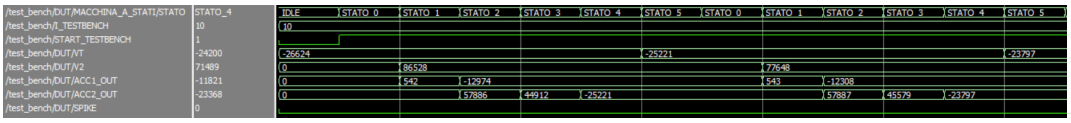


Figure 3.16: *Simulation Modelsim of a single cycle of Izhikevich neuron*

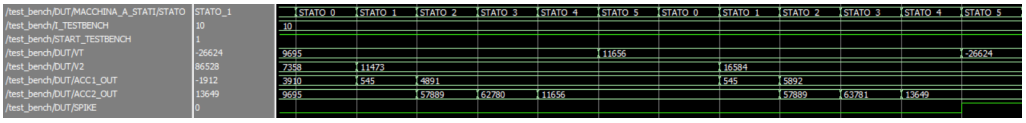


Figure 3.17: *Simulation Model of a neuron cycle where a spike is emitted*

The content of the register VT is compared to the Matlab script in order to verify the amplitude and time errors. As it is written before, the time error, represented by the MRE by the formula 3.7, in null, so the spikes are emitted in the same neuron cycle of both the Matlab code and the HW implementation; obviously there is an amplitude error introduced by the approximation due to the precision of the decimal digits, but it is considered negligible.

Integration Neuron

For the Output Layer of neuron, it has been used used an Izhikevich Neuron with some differences: the integration Izhikevich Neuron. Its task is to integrate the dv steps generated at the output of the neurons through an adder. The architecture and the FSM is exactly the same of the previous paragraphs; the only difference is that the sum of the various steps dv is reported at the end of each computation cycle.

At the end of the integration time, the neuron with the highest value will correspond to the digit to be recognized.

3.2.2 Pre-Synaptic Units

Before the layers of neurons, there are two pre-synaptic units: the hidden pre-synaptic unit, before the Hidden Layer of Neurons and the output pre-synaptic unit, before the Output Layer of Neurons. Their task is to compute the input current of the neurons combining linearly the weights of the layers with, respectively, the output of the Max Pooling Layer, for the hidden pre-synaptic unit, and the output spikes of the hidden layer, for the output pre-synaptic one.
In fact, the implementation of the two pre-synaptic units is profoundly different.
The general equation computed by the pre-synaptic units is described in 3.8.

$$out_{presyn,neuron-i} = \sum_{k=0}^{\#\ input} weight_{k,i} * input_k \tag{3.8}$$

Hidden Pre-Synaptic Unit

The Hidden Pre-Synaptic Unit, represented in the Figure 3.19 is organized with an array of 64 multipliers, an adder of tree and an accumulation register. Therefore, the data at the output of the Max Pooling Register is processed with a batch of 64, so to compute the input current of each neuron it requires 8 clock cycles. To compute all the input currents for the 128 neurons, it requires 1024 clock cycles.
The task of the Counter 3 bit is to drive the output multiplexer of the Max Pooling Register File, which is organized concatenating 64 output location of the register file for each input of the 8-multiplexer.
The Counter 7 bit is used to indicate which neuron the Pre-Synaptic Hidden Unit is computing the input current for.

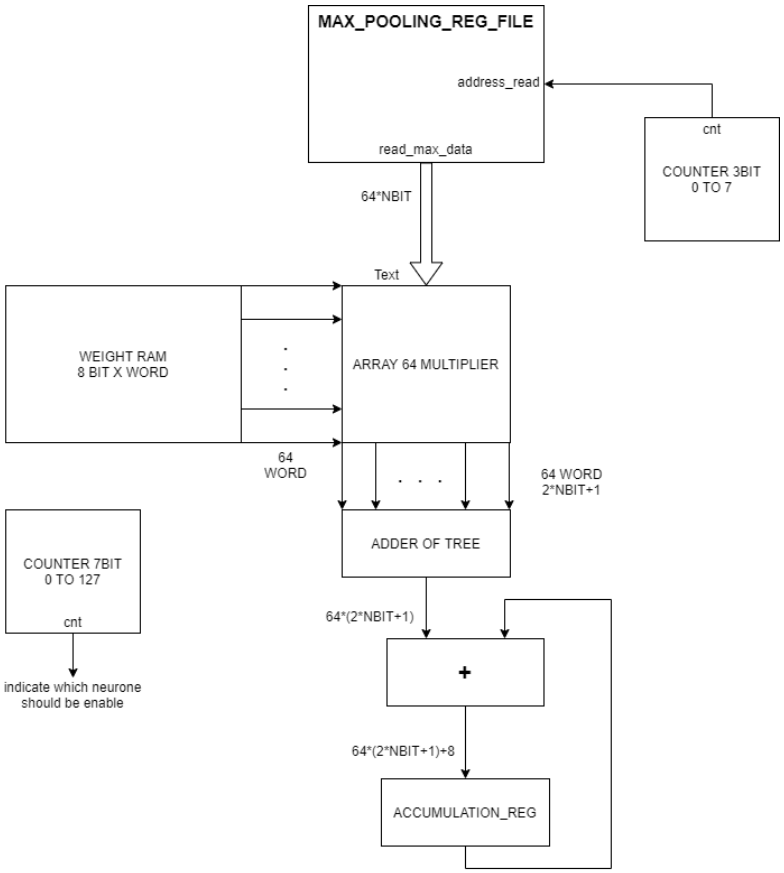
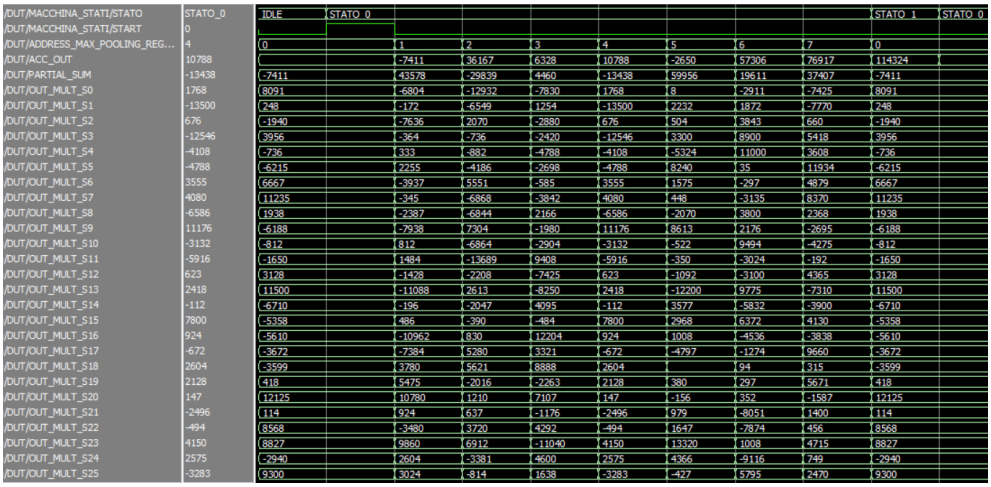


Figure 3.18: *Datapath of the Hidden Pre-Synaptic Unit*

It has been chosen this topology in order to reduce the number of multipliers to decrease the power consumption and the area.
The block is managed by a finite state machine, shown in Appendix E, which generates the various control signals. In this case, we consider the weight RAM with the different location always available, as in a Register File; if the RAM needs one more clock cycle to make the weights available to the output, it is necessary one more state.
To simulate the behaviour of the system, it has been realized a testbench with 490 random input values e 490×128 random weights and it has been used Modelsim to test. The output is compared to a C code realized with

the same function as the HW implemented.



/DUT/MACCHINA_STATI/STATO	STATO_0	IDLE	STATO_0								STATO_1	STATO_0
/DUT/MACCHINA_STATI/START	0											
/DUT/ADDRESS_MAX_POOLING_REG...	4	0	1	2	3	4	5	6	7	0		
/DUT/ACC_OUT	10788		-7411	36167	6328	10788	-2650	57306	76917	114324		
/DUT/PARTIAL_SUM	-13438	-7411	43578	-29839	4460	-13438	59956	19611	37407	-7411		
/DUT/OUT_MULT_S0	1768	8091	-6804	-12932	-7830	1768	8	-2911	-7425	8091		
/DUT/OUT_MULT_S1	-13500	248	-172	-6549	1254	-13500	2232	1872	-7770	248		
/DUT/OUT_MULT_S2	676	-1940	-7636	2070	-2880	676	504	3843	660	-1940		
/DUT/OUT_MULT_S3	-12546	3956	-364	-736	-2420	-12546	3300	8900	5418	3956		
/DUT/OUT_MULT_S4	-4108	-736	333	-882	-4788	-4108	-5324	11000	3608	-736		
/DUT/OUT_MULT_S5	-4788	-6215	2255	-4186	-2698	-4788	8240	35	11934	-6215		
/DUT/OUT_MULT_S6	3555	6667	-3937	5551	-585	3555	1575	-297	4879	6667		
/DUT/OUT_MULT_S7	4080	11235	-345	-6868	-3842	4080	448	-3135	8370	11235		
/DUT/OUT_MULT_S8	-6586	1938	-2387	-6844	2166	-6586	-2070	3800	2368	1938		
/DUT/OUT_MULT_S9	11176	-6188	-7938	7304	-1980	11176	8613	2176	-2695	-6188		
/DUT/OUT_MULT_S10	-3132	-812	812	-6864	-2904	-3132	-522	9494	-4275	-812		
/DUT/OUT_MULT_S11	-5916	-1650	1484	-13689	9408	-5916	-350	-3024	-192	-1650		
/DUT/OUT_MULT_S12	623	3128	-1428	-2208	-7425	623	-1092	-3100	4365	3128		
/DUT/OUT_MULT_S13	2418	11500	-11088	2613	-8250	2418	-12200	9775	-7310	11500		
/DUT/OUT_MULT_S14	-112	-6710	-196	-2047	4095	-112	3577	-5832	-3900	-6710		
/DUT/OUT_MULT_S15	7800	-5358	486	-390	-484	7800	2968	6372	4130	-5358		
/DUT/OUT_MULT_S16	924	-5610	-10962	830	12204	924	1008	-4536	-3838	-5610		
/DUT/OUT_MULT_S17	-672	-3672	-7384	5280	3321	-672	-4797	-1274	9660	-3672		
/DUT/OUT_MULT_S18	2604	-3599	3780	5621	8888	2604		94	315	-3599		
/DUT/OUT_MULT_S19	2128	418	5475	-2016	-2263	2128	380	297	5671	418		
/DUT/OUT_MULT_S20	147	12125	10780	1210	7107	147	-156	352	-1587	12125		
/DUT/OUT_MULT_S21	-2496	114	924	637	-1176	-2496	979	-8051	1400	114		
/DUT/OUT_MULT_S22	-494	8568	-3480	3720	4292	-494	1647	-7874	456	8568		
/DUT/OUT_MULT_S23	4150	8827	9860	6912	-11040	4150	13320	1008	4715	8827		
/DUT/OUT_MULT_S24	2575	-2940	2604	-3381	4600	2575	4366	-9116	749	-2940		
/DUT/OUT_MULT_S25	-3283	9300	3024	-814	1638	-3283	-427	5795	2470	9300		

Figure 3.19: *Modelsim simulation of the Hidden Pre-Synaptic Unit*

Once the system receives the start signal, it sends to the Max Pooling Register the address of the data to be elaborated, i.e. the output of the 3 bit counter, indicated in the Figure as *ADDRESS_MAX_POOLING_REG*. The input data will be multiplied and saved into the signal *OUT_MULT_S*; the signal *PARTIAL_SUM* is the behavioural description of the final sum of the adder tree. Finally, this signal is added to the output of the accumulation register and saved in the same accumulation register: the signal which represents the output is *ACC_OUT*.

ReLU Activation function

$$f(x) = max(0, x) \tag{3.9}$$

At the output of the Hidden Pre-Synaptic unit, it has been used a ReLU activation function. The shape of the function is represented in Figure 3.20. At HW level, the block was simply implemented in a behavioral way.

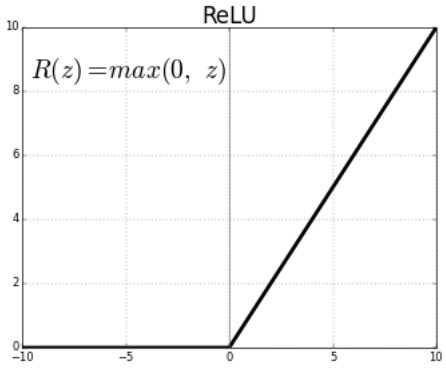


Figure 3.20: *ReLU function trend*

Output Pre-Synaptic Unit

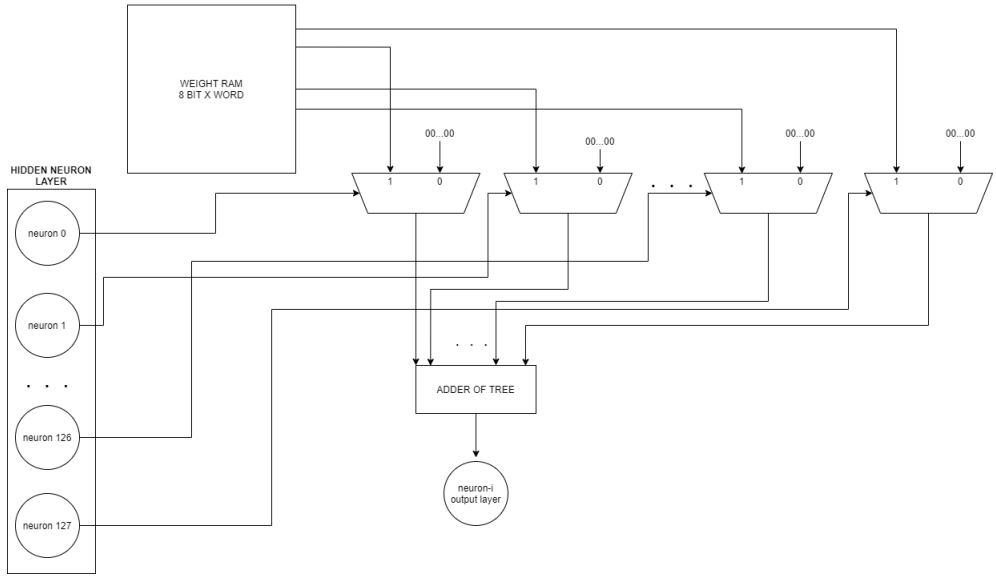


Figure 3.21: *Datapath of the Output Pre-Synaptic Unit*

The Output Pre-Synaptic Unit is realized in order to take the advantages described in the subsection 2.2.3.
In the system, there are 10 Output Pre-Synaptic Unit, one for each neuron of the output layer. The datapath of the Pre-Synaptic Unit is represented in

the Figure 3.21.
Multipliers are replaced by multiplexer due to the fact that the output of the neurons are sequence of spikes. Therefore, each multiplexer has in input the corresponding weight and a sequence of zeros: if the hidden neuron, driving the correspond multiplexer, fires, the multiplexer outputs the weight otherwise the sequence of zeros. The output of the multiplexers is summed through a sum tree and it becomes the input current of the output neuron. It is possible to note that the block is totally combinatorial.
Also for the verification of this block, it has been written a C code which generates random input for the multiplexers and random weights. Then, the C Code calculates the sum to compare with the results of the VHDL simulation.
It has been tested the HW implementation with Modelsim and the output waveform is reported in Figure 3.22.

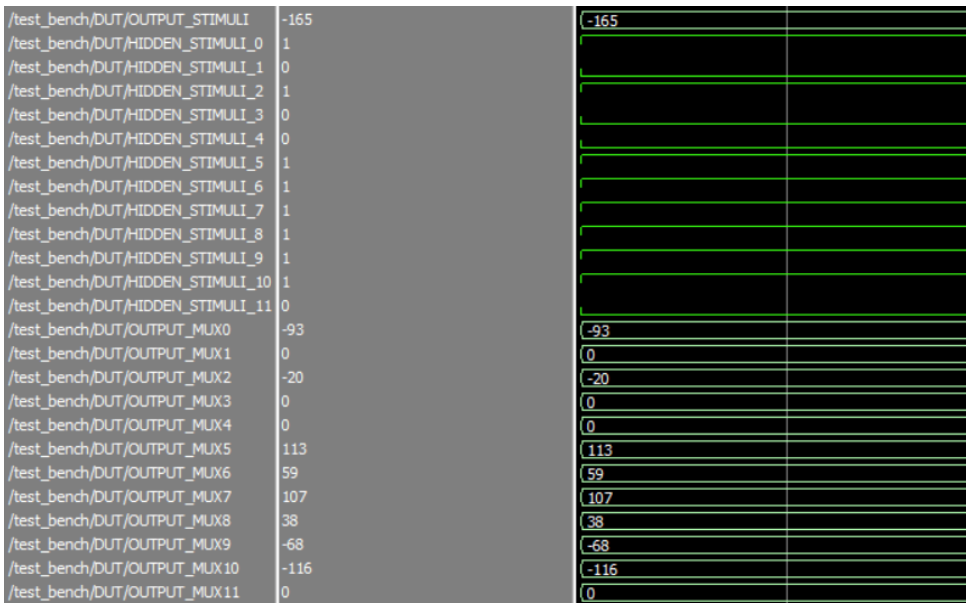


Figure 3.22: *Modelsim simulation of the Output Pre-Synaptic Unit*

In the Figure 3.22, it has been reported only a part of the overall simulation:

in the signals $HIDDEN\_STIMULI$, there are the output spikes of the neurons of the hidden layer, while in the signals $OUTPUT\_MUX$, there are the output of the multiplexers to be summed to have the input current of the output neurons.

3.2.3 Winner Selector

Finally, the outputs of the 10 Integration Neurons are sent to the block that takes care of deciding which is the correct output: the Winner Selector. The block simply compares the 10 different values together, and outputs the neuron with the highest value.
The index of the outgoing neuron will match the digit it identifies from the neural network.

4. Input Data

4.1 MNIST dataset

Figure 4.1: *Example of some handwritten digits from MNIST dataset*

The MNIST (Mixed National Institute of Standards and Technology) dataset [6] is a set of handwritten digits used to train and test various image processing system, as Neural Network. It is composed by 60.000 train images and 10.000 test images.

Original images were submitted to a preprocessing process. Firstly, the im-

ages will normalized to fit in a 20×20 pixel box while preserving the aspect ratio. Then, it has been applied an anti-aliasing filter, and as a result black and white images were effectively transformed into gray scale. Then a blank padding was introduced to fit the images in a larger 28×28 pixel box, so that the center of mass of the digit matched its centre.
An example of some figures of the dataset is represented in Figure 4.1, while single MNIST sample belonging to the digit "7" is in Figure 4.2.

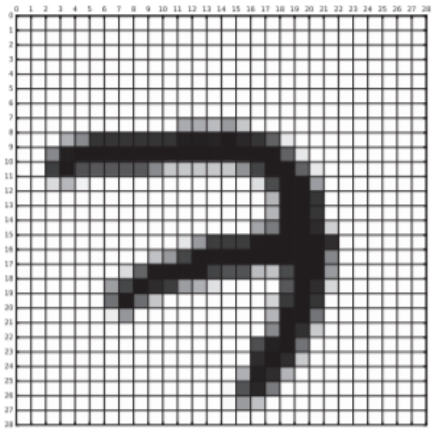


Figure 4.2: *Example of a single sample from MNIST dataset*

4.2 Spike Coding

In order to convert the dataset images into spikes, it has been used an appropriate PyTorch function, explained in details in the section 5.3. The chosen encoding method is the Spike Latency, present in the Norse library used for writing the PyTorch script.
This method encodes an input value by the time the first spike occurs, therefore for each pixel it is possible to have at most one spike. This train of spikes is composed by a vector with 3 location for each spikes: the timestamp (in ms) which corresponds to the instant of time when the pulse is emitted, the

x address and the y address, which are the pixel location of the pulse.
In the Figure 4.3, there is a Figure which represent an image of the MNIST dataset encoded into a spikes train.

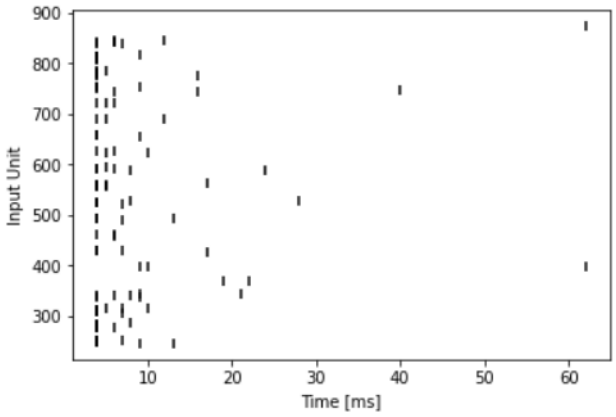


Figure 4.3: *Example of Spike Coding method*

5. Offline Learning

5.1 PyTorch and Norse

PyTorch introduction

PyTorch [7] is an open source library used for machine learning based on the Torch Library. It is used for developing and training neural network based deep learning model.
In literature, there are different libraries to model neural network: it has been chosen PyTorch for two reasons which make it particularly efficient for deep learning.
First of all, it provides accelerated computation using graphical processing units (GPUs), which a speedups of the calculation over 50x respect the same calculation with the CPU. Secondly, PyTorch provides structures which support numerical optimization on generic mathematical expressions, which deep learning uses for training.

Norse introduction

Norse [8] expands the PyTorch library with primitive biological components in order to develop and train event-driven neural network. It contains neuron models (Integrate and Fire and Leaky-Integrate-and-Fire), synapse dynamic, encoding and decoding algorithm, dataset integrations, tasks, and examples.

5.2 Extension of Norse: layer of Izhikevich

To implement the hidden layer and the output layer of Izhikevich neurons implemented in HW, the Norse library has been extended with two functions: IZHCell and IZHLinearCell.
For the implementation of the two functions, we started from the functions LIFCell and LILinearCell, which implement respectively a LIF neuron and a LI integrated neuron.

5.2.1 IZHCell

IZHCell is a module which computes a single euler-integration step of a Izhikevich neuron-model without recurrence.

class ***izhikevich_cell.IZHCell****(p=IZHParameters(a=tensor(0.0020), b=tensor(0.0200), c=tensor(-6.5000), d=tensor(0.6000), k=tensor(-1.3000), sq=tensor(0.0040), mn=tensor(0.5000), bias=tensor(14.), v_th=tensor(3.), tau_inv=tensor(31.25), method='super', alpha=100), dt=0.001)*

Parameters:

- p (IZHParameters) - Parameters of the IZH neuron model;
- dt (float) - Time step to use. Defaults to 0.001;

5.2.2 IZHLinearCell

The IZHLinearCell is a cell for a izhikevich-integrator with an additional linear weighting.

class ***izhikevich_integrator_module.IZHLinearCell****(hidden_size, input_size, p=IZHParameters(a=tensor(0.0020), b=tensor(0.0200), c=tensor(-6.5000), d=tensor(0.6000), k=tensor(-1.3000),*

sq=tensor(0.0040), mn=tensor(0.5000), bias=tensor(14.), v_th=tensor(3.), tau_inv=tensor(31.25), method='super', alpha=100), dt=0.001)

Parameters:

- input_size (int) - Size of the input layer;
- hidden_size (int) - Size of the hidden layer;
- p (IZHParameters) - Parameters of the IZH neuron model;
- dt (float) - Time step to use. Defaults to 0.001;

5.3 Description of the code

To implement the software training, it has been realized a PyTorch code using the library Norse and the realized extension functions. The code has been written and tested on Google Colab.
In this chapter, it has been reported the most significant part of the code.
First of all, it has been declare some of the library that the code needs:

```
import torch
import numpy as np
import matplotlib.pyplot as plt
!pip install --quiet norse
```

Then, it has been written the following portion of code in order to download and to extract the MNIST dataset and to adapt it to the input image 32×32.

```
import torchvision
BATCH_SIZE = 256
transform = torchvision.transforms.Compose(
    [
        torchvision.transforms.Resize(32),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.1307,), (0.3081,)
    ),
```

```
    ]
)
train_data = torchvision.datasets.MNIST(
    root=".",
    train=True,
    download=True,
    transform=transform,
)
train_loader = torch.utils.data.DataLoader(
    train_data,
    batch_size=BATCH_SIZE,
    shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST(
        root=".",
        train=False,
        transform=transform,
    ),
    batch_size=BATCH_SIZE
)
```

From the Norse library, it has been used the Spike Latency Encoder, which converts the input image of the MNIST dataset in a sequence of spikes.

```
from norse.torch import SpikeLatencyLIFEncoder

T = 100
example_encoder = SpikeLatencyLIFEncoder(T)

example_input = example_encoder(img)
example_spikes = example_input.reshape(T,32*32).to_sparse().
    coalesce()
t = example_spikes.indices()[0]
n = example_spikes.indices()[1]

plt.scatter(t, n, marker='|', color='black')
plt.ylabel('Input Unit')
```

```
plt.xlabel('Time [ms]')
plt.show()
```

Once the data is encoded into spikes, it has been described the convolutional event-based neural network implemented in HW, using the function described in the section 5.2. The neural network is composed by:

- $torch.nn.Conv2d(1, 10, 5, 1)$: a Convolutional Layer with 10 programmable Kernel 5×5, with $stride = 1$ and $padding = 1$;;
- $torch.nn.functional.max\_pool2d(z, 4, 4)$: a Max pooling Layer with $stride = 4$ and $padding = 4$;
- $torch.nn.Linear(490, 128, bias = False)$: a Fully Connected Layer, in order to applies a linear transformation to the incoming data
- $torch.nn.functional.relu(z)$: a layer of ReLu used as activation function
- $IZHCell(p = IZHParameters(method = method, alpha = alpha))$: a layer of Izhikevich neuron used to applied to the fully connected layer the Izhikevich neuron's formula;
- $IZHLinearCell(128, 10)$: a layer of integrated Izhikevich neuron to calculate the output

```
class ConvNet(torch.nn.Module):
    def __init__(self,  num_channels=1, feature_size=32,
    method="super", alpha=100):
        super(ConvNet, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 10, 5, 1, bias=False)
        self.fc1 = torch.nn.Linear(490, 128, bias=False)
        self.izh1 = IZHCell(p=IZHParameters(method=method,
    alpha=alpha))
        self.out = IZHLinearCell(128, 10)

```

```
    def forward(self, x):
        seq_length = x.shape[0]
        batch_size = x.shape[1]

        # specify the initial states
        s0 = s1 = s2 = so = None

        voltages = torch.zeros(
            seq_length, batch_size, 10, device=x.device,
    dtype=x.dtype)

        for ts in range(seq_length):
            z = self.conv1(x[ts, :])
            z = torch.nn.functional.max_pool2d(z, 4, 4)
            z = z.view(-1, 490)
            z = self.fc1(z)
            z = torch.nn.functional.relu(z)
            z = F.dropout(z, training=self.training)
            z, s1 = self.izh1(z, s1)
            v, so= self.out(z, so)
            voltages[ts, :, :] = v
        return voltages

conv_net=ConvNet()
print(conv_net)
```

The last step is to setup training and test code. This code does not depend on the fact that we are training a spiking neural network, therefore this last part is copied from the pytorch tutorial of training of a neural network. First of all, it has been defined the number of epochs, that is an hyperparameter which defines the number times that the learning algorithm will work through the entire training dataset. Hence, one epoch means that all the sample of the dataset can update one time the internal model weights and kernels.
Then, it has been described the *train* and *test* function in order to train and test the neural network using the 60.000 samples of the MNIST dataset.

Finally, the functions have been called up in a *for loop* which depends on the number of epochs.

```
from tqdm.notebook import tqdm, trange
EPOCHS  = 5 # Increase this number for better performance

def train(model, device, train_loader, optimizer, epoch,
    max_epochs):
    model.train()
    losses = []

    for (data, target) in tqdm(train_loader, leave=False):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = torch.nn.functional.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        torch.save({
          'state_dict': model.state_dict(),
          'optimizer' : optimizer.state_dict(),
        }, 'prova.pth.tar')

    mean_loss = np.mean(losses)
    return losses, mean_loss

def test(model, device, test_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += torch.nn.functional.nll_loss(
                output, target, reduction="sum"
```

```
            ).item()  # sum up batch loss
            pred = output.argmax(
                dim=1, keepdim=True
            )  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().
    item()

    test_loss /= len(test_loader.dataset)

    accuracy = 100.0 * correct / len(test_loader.dataset)

    return test_loss, accuracy

training_losses = []
mean_losses = []
test_losses = []
accuracies = []

for epoch in trange(EPOCHS):
    training_loss, mean_loss = train(model, DEVICE,
    train_loader, optimizer, epoch, max_epochs=EPOCHS)
    test_loss, accuracy = test(model, DEVICE, test_loader,
    epoch)
    training_losses += training_loss
    mean_losses.append(mean_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

print(f"final accuracy: {accuracies[-1]}")
```

5.4 Results of the simulation

After completing the drafting of the PyTorch code, it has been run several times using the dataset MNIST to evaluate the different accuracy results by modifying the number of epochs.

In the Table 5.1, it is reported a series of 5 simulations with 5 epochs with an integration time $T = 100$.

Iteration	**Accuracy**
1	92.39%
2	92.3%
3	93.22%
4	92.79%
5	93.08%

Table 5.1: *Result of Accuracy with 5 epochs*

Then, it has been increased the number of epochs, testing the software with 10, 20, 50 and 100 epochs. The results are reported in Table 5.2

Number of epochs	**Accuracy**
10	94.96%
20	96.63%
50	97.4%
100	97.55%

Table 5.2: *Result of Accuracy increasing the number of epochs*

It is possible to note that already with 5 epochs, the resulting accuracy is acceptable. Increasing the number of epochs, also the accuracy improved, reaching the optimum values of 97.4% with 50 epochs and 97.55% with 100 epochs. In the Table 5.3, it is possible to find the accuracy of other similar Convolutional Spiking Neural Network, with different configuration, which used an offline training.

Model	**Accuracy**
Diehl [9] (2015)	99.10%
Neil [10] (2016)	95.72%
Garbin [11] (2014)	94.00%
Frenkel [3] (2020)	97.5%

Table 5.3: *Accuracy of other similar architecture*

The accuracy of the implemented neural network is similar and acceptable

compared to the other one found in literature.
In the Figure 5.1, three test simulations of the PyTorch script are shown. On the left side there is the image to be decoded, with its correct label, while on the right side there is the trend, as a function of time, of the 10 output integration neurons: the result of the decoding corresponds to the neuron with a higher integration value.

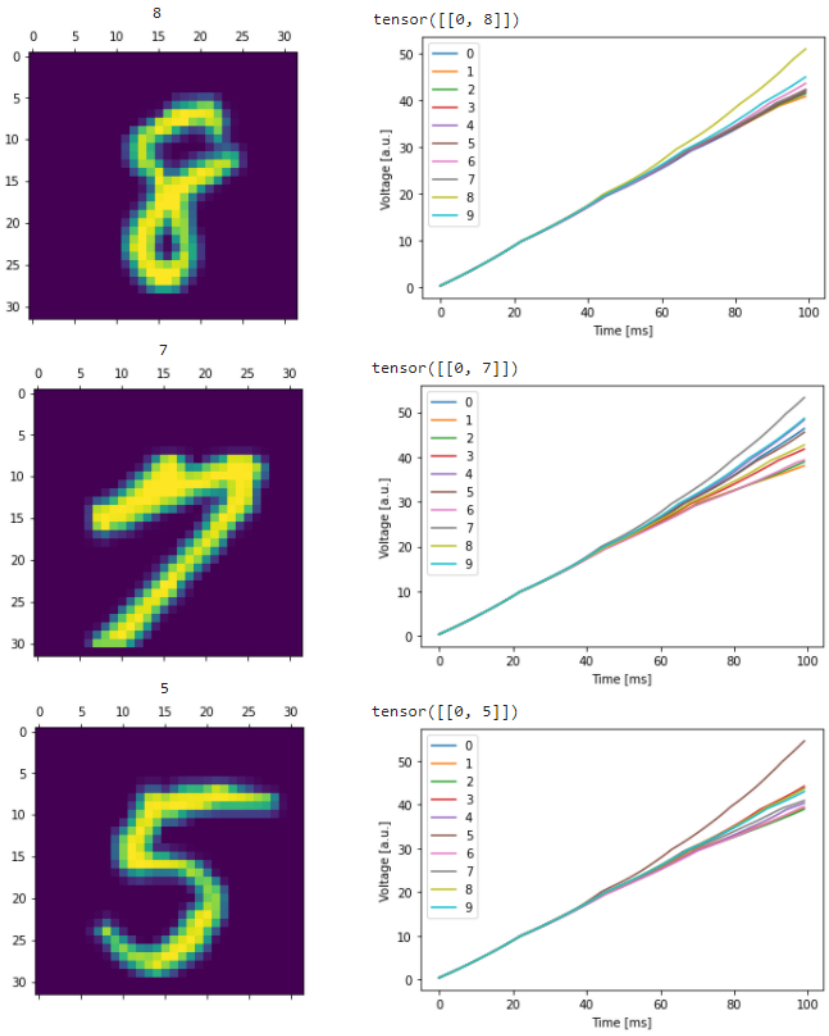


Figure 5.1: *Example of some simulations of the PyTorch code*

6. HW Simulation and Synthesis

6.1 Modelsim Simulation

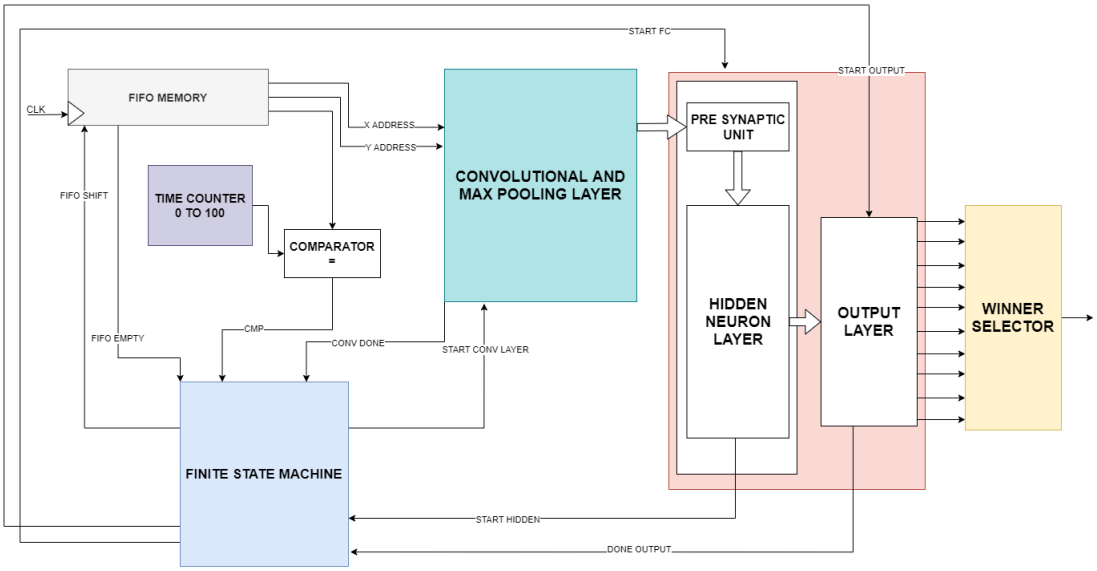


Figure 6.1: *Datapath of the whole architecture*

After completing the offline script, the whole architecture was implemented, combining the various blocks created in chapter 3, and was tested through Modelsim.
The overall architecture is represented in Figure 6.1; detailed control signals and detailed signal of the blocks are not shown for clarity.

The FIFO memory is modeled through a text file containing the input signals and a VHDL read function. The input data in the file is organized using this format: *polarity, arrival time, x address, y address.*
The integration time is counted by a counter and it is used with the arrival time to drive the whole architecture.
All the control signals, used to drive the start and the other control signals of the other blocks of the architecture, are generated though a Finite State Machine which is reported in Appendix F.
To test the architecture, it has been used Modelsim and to compare the output of the different blocks, it has been modified the PyTorch script implemented in chapter 5 in order to load a file containing the neural network model (i.e. its kernels and weights) and to print the output of different layers of the architecture.
In the Appendix G, it has been reported a Modelsim simulation of the whole architecture, described step by step. In the following, it has been reported only the final results of the architecture with some considerations.
The input used for this simulation is the sample 1521 of the dataset MNIST; it represents the digit 1 and it is reported in Figure 6.2. In the Figure 6.3, it

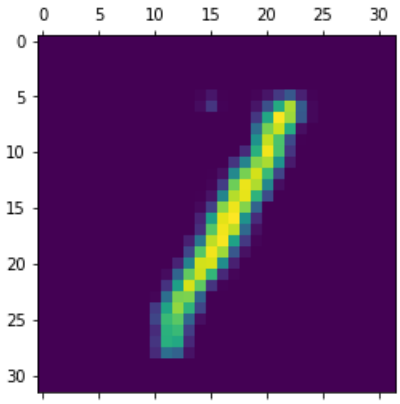


Figure 6.2: *Digit example chooses for the reported simulation*

has been reported the final part of the simulation: the signals into the Figure

represents the 10 output value of the Izhikevich integrated neurons.

Figure 6.3: *Simulation of a part of the Final part of the whole architecture*

Exactly as expected, the value of the signal *OUTPUT_VALUE1*, which represents the output of the neuron 1, has the highest value: this means that the neural network correctly identifies the input digit.
In the Table 6.1, it has been reported the final value of the 10 output neuron in PyTorch anche in Modelsim simulations. The values of the Modelsim simulation are normalized respect 2^{-12}.

Neuron	PyTorch results	Modelsim results
0	21.603	20.517
1	31.3439	33.053
2	24.4150	23.586
3	20.1959	20.876
4	21.4549	24.278
5	21.0834	17.605
6	21.5325	18.608
7	23.4512	24.649
8	20.7712	20.719
9	20.9425	20.873

Table 6.1: *Comparison between the PyTorch output and the Modelsim output*

As expected, the values are very similar, but not perfectly equal: this error is due to the various approximations that the hardware inevitably introduces. The architecture has been tested with several input digits and has always worked correctly: in future works, we could look for an efficient way to test the HW architecture with all 60000 samples present in the MNIST dataset and evaluate their accuracy, comparing to the value obtained from the PyTorch script.

6.2 Synthesis

Finally, at the design level, a synthesis of the different blocks that make up the architecture was carried out to study timing, power and area. Due to the complexity of the overall architecture, it has been divided into blocks.
The synthesis was carried out through Synopsys and the results are reported in the following sections.

6.2.1 UMC65

The chosen technology is UMC's 65nm Low-K Low leakage RVT process. The choice fell on this typology since, being the architecture organized in blocks that do not work simultaneously, but in succession, it is preferable to use a low leakage technology, so that the leakage current, and therefore the static power, in the blocks off is nothing.
In the Table 6.2, it has been reported the physical specification of for the standard cell library of the UMC65.

Characteristic	**Specification**
Cell height	$1.8\mu m$
Drawn gate lenght	$0.06\mu m$
Vertical routing track	9 $tracks$
Vertical pin grid	$0.2\mu m$
Horizontal pin grid	$0.2\mu m$
POWER/GND rail width	$0.3\mu m$

Table 6.2: *Physical specifications*

6.2.2 Neuron

In the Table 6.3, it has been reported the characteristics of a single neuron, synthesized with Synopsys.

Minimum Clock Period	$3.7ns$
Max Clock Frequency	$270.27MHz$
Combinational area	9866.52
Non Combinational area	2601.00
Buf/Inv area	1098.72

Table 6.3: *Synthesis of a single neuron*

6.2.3 Convolutional and Max Pooling Layer

In the Table 6.4, it has been reported the characteristics of a a single page of the Convolutional Layer, including its register File of 784 locations, the 49 Max Pooling operators and the correspond locations of the Max Pooling Register File.

Minimum Clock Period	$6.37ns$
Max Clock Frequency	$156.98MHz$
Combinational area	249485.04
Non Combinational area	252666.00
Buf/Inv area	38219.76

Table 6.4: *Synthesis of the Convolutional and Max Pooling Layer*

6.2.4 Pre-Synaptic Unit

In the Table 6.5, it has been reported the synthesis of the Hidden Pre-Synaptic Unit, without considering the weight RAM.

Minimum Clock Period	$2.14ns$
Max clock Frequency	$467.29MHz$
Combinational area	235204.19
Non Combinational area	1414.08
Buf/Inv area	8701.92

Table 6.5: *Synthesis of the Hidden Pre-Synaptic Unit*

6.2.5 Final Consideration

From the Tables in the previous sections, it can be seen that the bottleneck of the architecture area is represented by the area of the Convolutional Layer and the Hidden Pre-Synaptic Unit.
As far as the Convolutional Layer is concerned, the elevated area is justified by the fact that different Register File locations are also considered inside this synthesis, both of the Convolutional Layer and of the Max Pooling Layer; instead in the Hidden Pre-Synaptic unit, where the RAM containing the weights is not considered, the excessive area value is given by the presence of the array of 64 multipliers.
For a possible implementation on a FPGA, it is necessary to carefully evaluate the available area and the size of the integrated RAM: one possibility could be to use the integrated RAM both to store the different weights, and to store the outputs of the Max Pooling Register Block and the Convolutional Layer, replacing the register files now present.

7. Conclusion

In this thesis, it has been shown the potential of combining spiking neural network with Izhikevich Neuron, with the convolution operation. The excellent final result of the accuracy demonstrates how the Izhikevich neurons can be interfaced with the convolution operation without deteriorating the performance.
The next step will be to make the architecture more efficient in terms of power, area and speed: a possible idea is to work on the optimization of the Hidden Pre-Synaptic unit by using more efficient multipliers such as those based on Dadda-tree architectures. Another possibility could be to use the neurons also within the Convolutional Layer, in order to pass in the latter from a frame-based approach to a totally spike-base approach: this would lead to the Hidden Pre-Synaptic unit to eliminate totally the multipliers by replacing them with multiplexers, as done for the Output Pre-Synaptic Unit. Finally, it might be interesting to study the accuracy of the architecture using the N-MNIST dataset, which is the spiking translation of the MNIST, and testing everything with a DVS camera.

Appendix

A. Convolution C Code

```
for(i=k-1; i>=0; i--){
    for(j=k-1; j>=0; j--){
        kernel_product[i][j]=polarity*kernel[q][w];
        w++;
    }
} ......
for (i=0; i<k; i++){
    for(j=0; j<k; j++){
     if((x-j)>=0 && (y-i)>=0 && (x-j)<N-k+1 && (x-i)<N-k+1){
        acc=timestamp*kernel[i][j];
        convolutional_matrix[y-i][x-j]=convolutional_matrix[y
    -i][x-j]+acc;
        address=(y-i)*28+(x-j);
        for(z=0; z<4; z++)
           for(t=0; t<4; t++){
                if(part_max_pooling<convolutional_matrix[((y-
    i)/4)*4+z][((x-j)/4)*4+t]){
                    part_max_pooling=convolutional_matrix[((y
    -i)/4)*4+z][((x-j)/4)*4+t];
                }
           }
        }
        max_pooling_matrix[((y-i)/4)][((x-j)/4)]=
    part_max_pooling;
      }
    } }
```

B. **FSM Convolutional Layer**

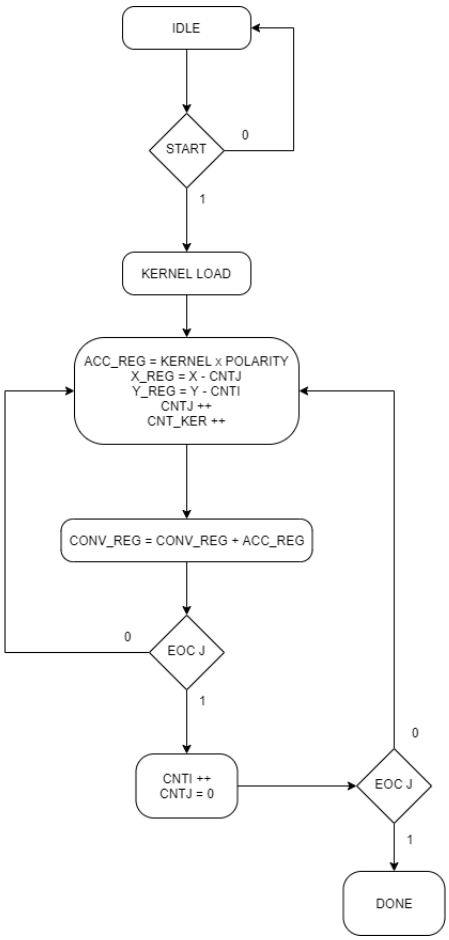


Figure B.1: *FSM of the Convolutional Layer*

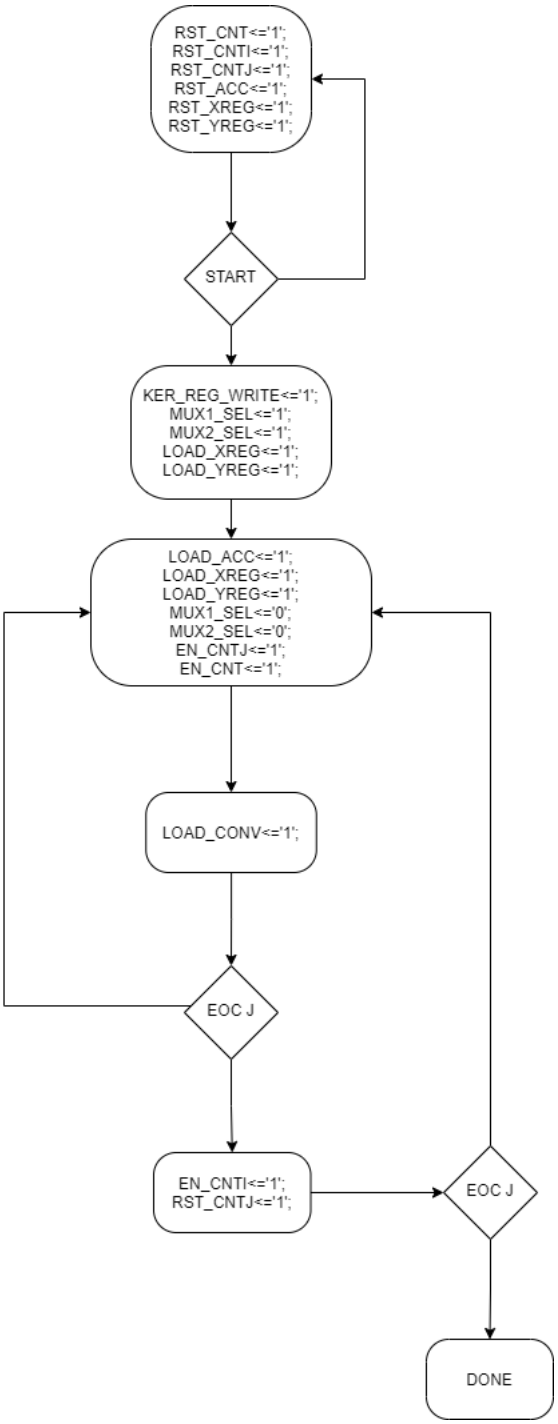


Figure B.2: *Detailed FSM of the Convolutional Layer*

C. Modelsim Simulation Conv.Layer

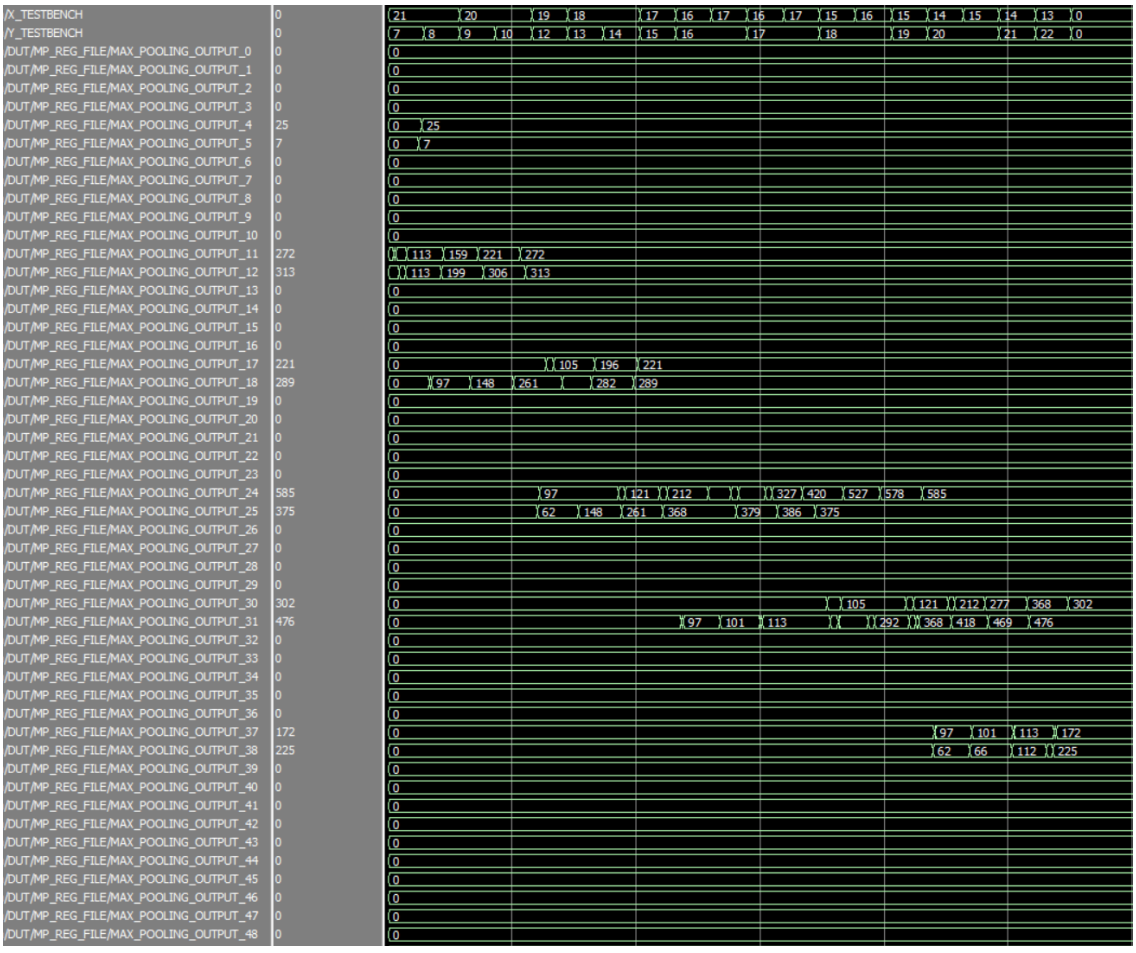


Figure C.1: *Modelsim simulation Convolutional and Max Pooling Layer*

D. **FSM Izhikevich Neuron**

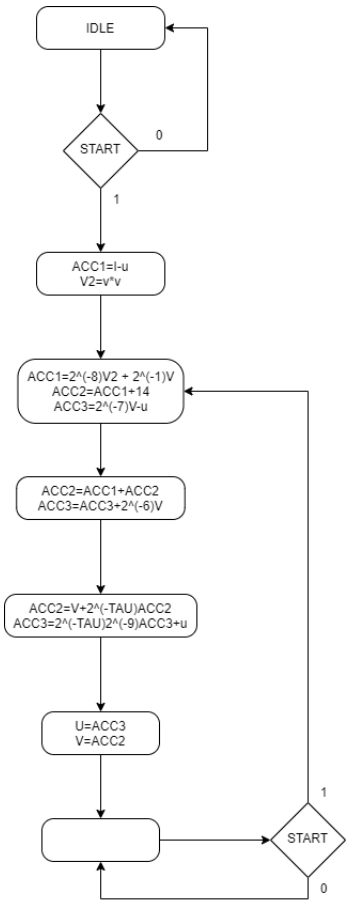


Figure D.1: *FSM of the HW implementation of the Izhikevich Neuron*

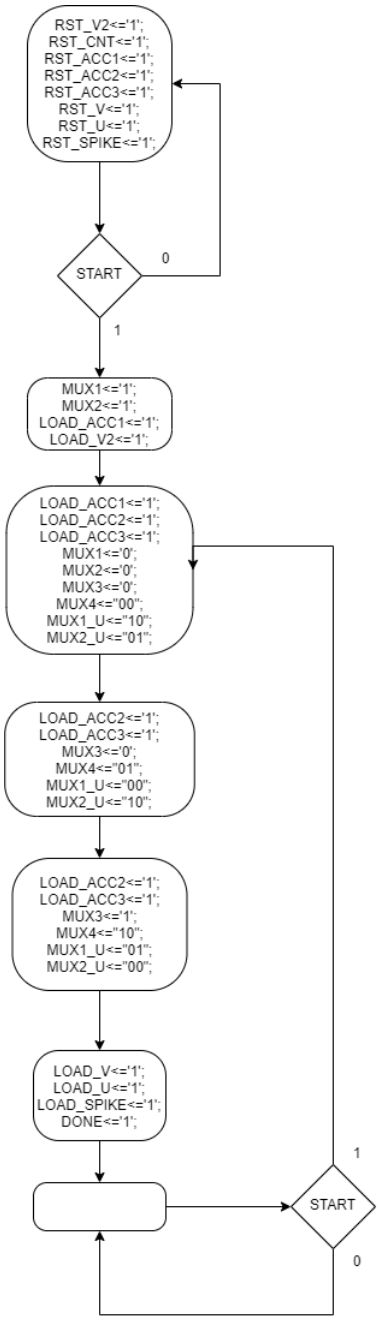


Figure D.2: *Detailed FSM of the HW implementation of the Izhikevich Neuron*

E. **FSM Hidden Pre-Synaptic Unit**

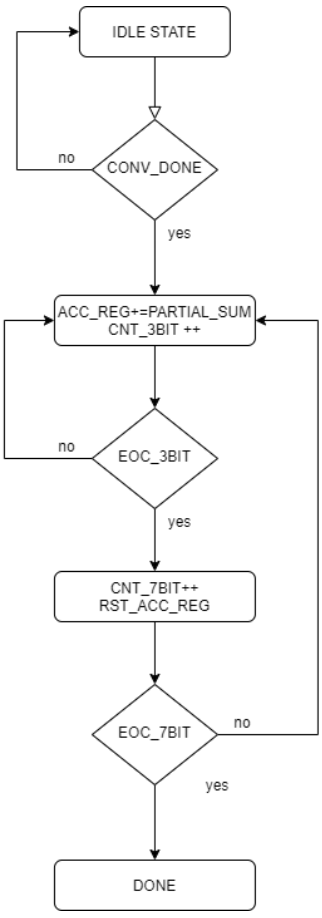


Figure E.1: *FSM of the Hidden Pre-Synaptic Unit*

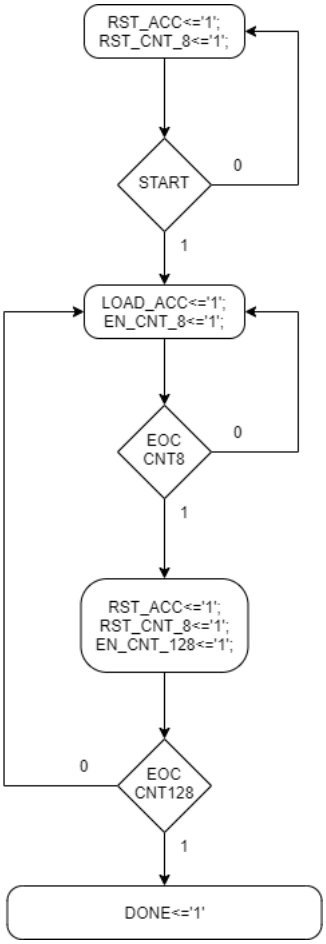


Figure E.2: *Detailed FSM of the Hidden Pre-Synaptic Unit*

F. FSM CEDNN

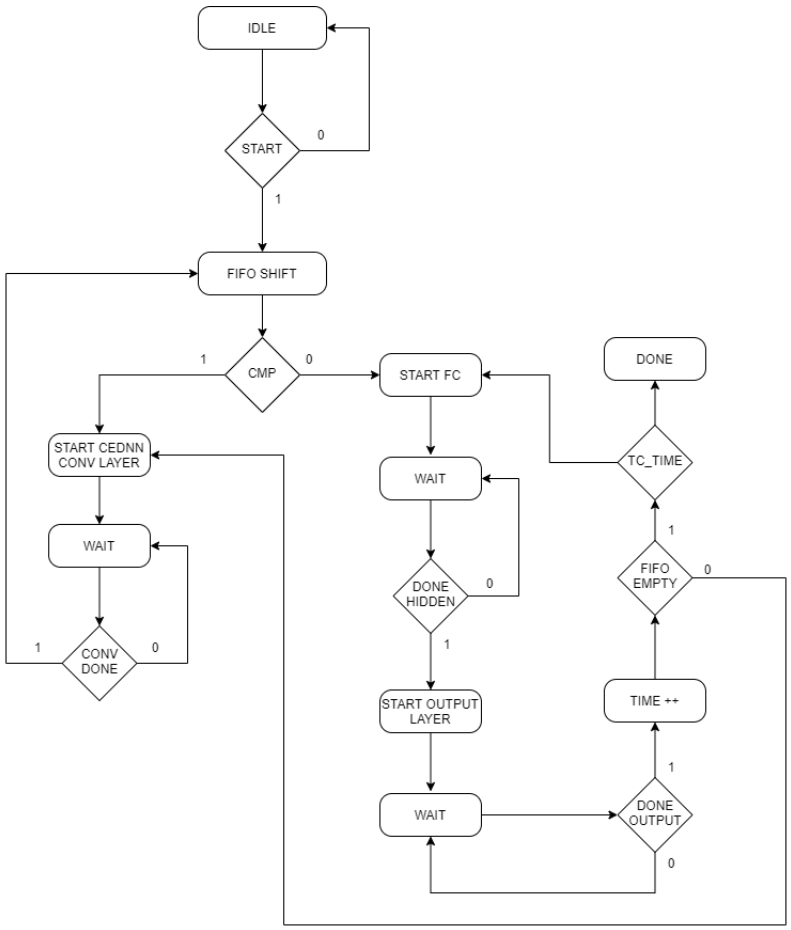


Figure F.1: *FSM of the whole architecture*

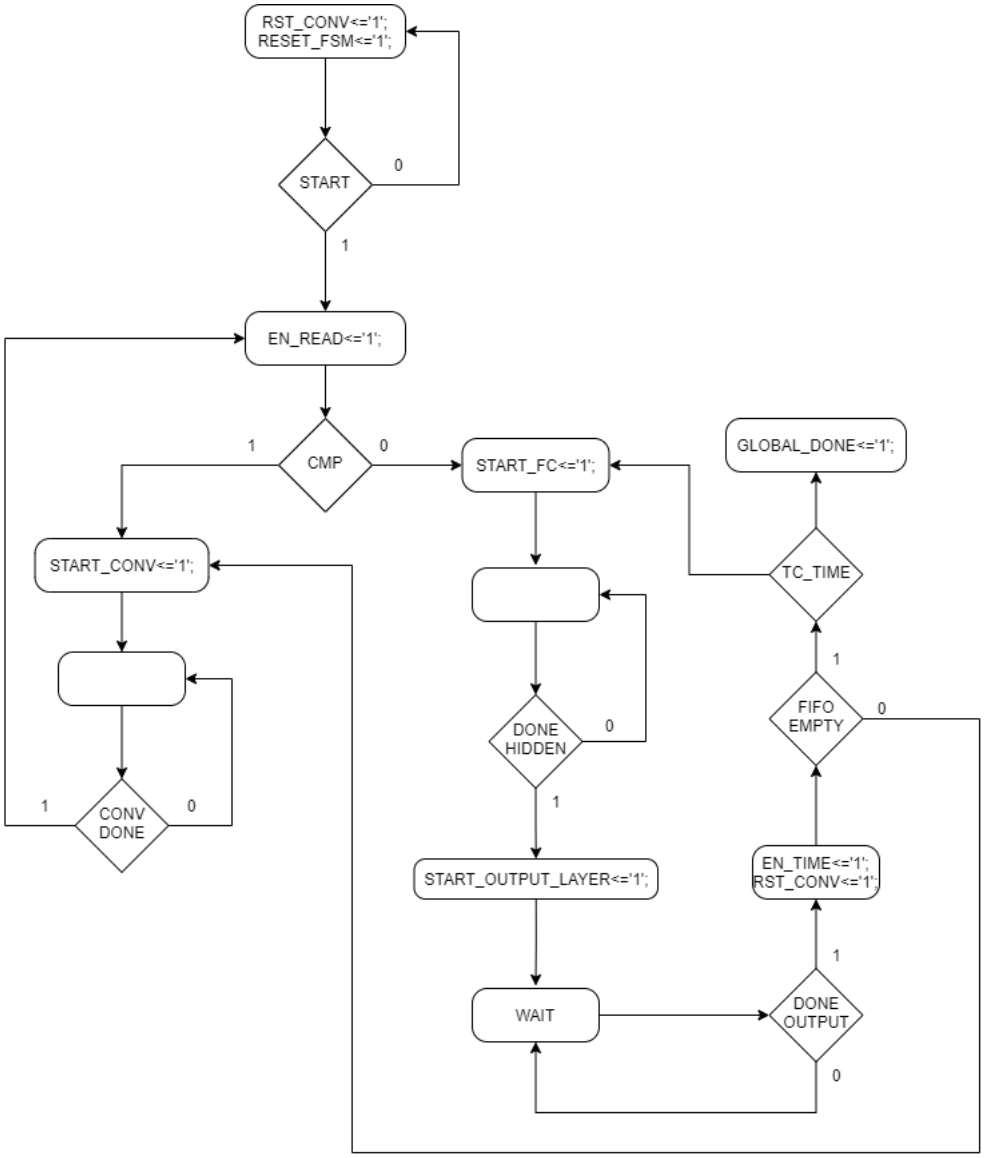


Figure F.2: *Detailed FSM of the whole architecture*

G. CEDNN Simulation

In Figure G.1, it has been represented a part of the Convolution operation of the whole architecture. It receives the x and y addresses, from the FIFO, and taking data from the Convolutional Register File, calculates the convolution results and store them into the Convolution Register File. In all the simulations, the variable *STATO* refers to the current state of the FSM of the whole architecture, reported in Appendix F.

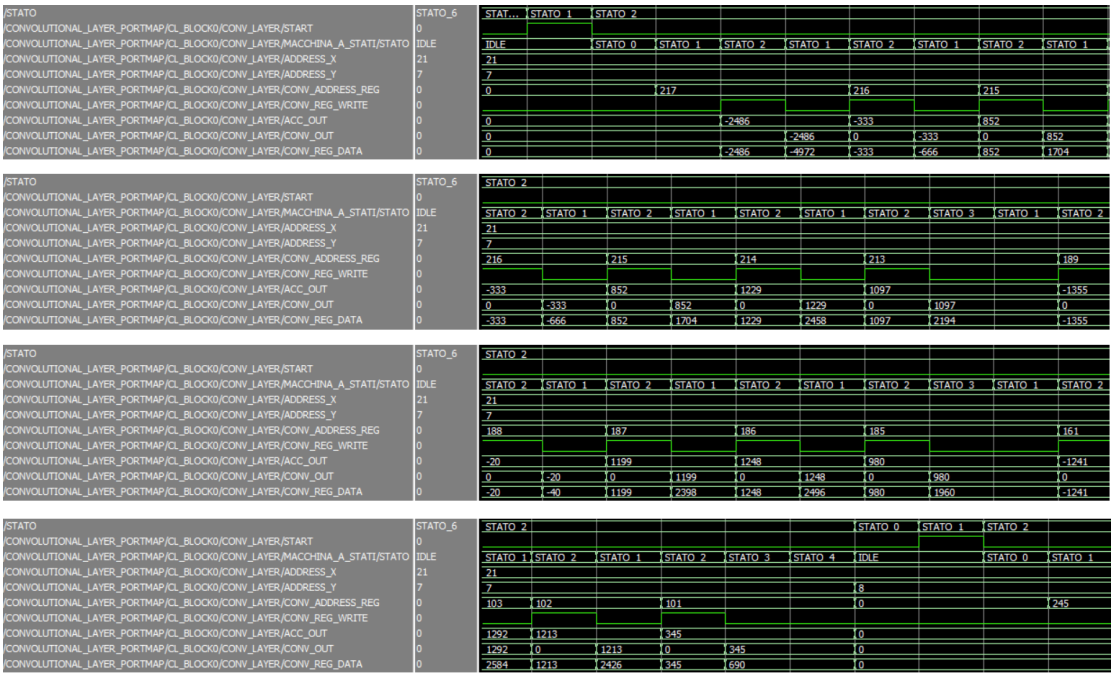


Figure G.1: *Simulation of the Convolutional Layer in the whole architecture*

In Figure G.2, it has been represented the Max Pooling operation. In a combinatorial way, it is applied the max pooling operation to the data at the output of the Convolutional Register File.

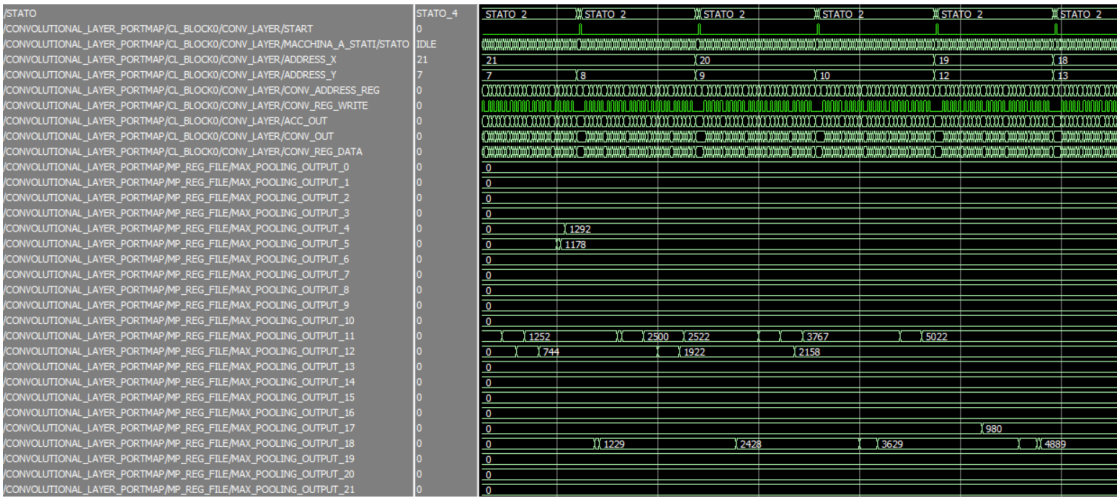


Figure G.2: *Simulation of the Max Pooling Layer in the whole architecture*

In Figures G.3 and G.4, it has been represented a part of the simulation of the Pre-Synaptic Layer. It receives the 490 data at the output of the Max Pooling register, it processes them with the respective weight and the final output becomes the input current of the neuron. Specifically, the Figure G.3 shows the calculation of the input current of neuron 0, which becomes zero due to the block of ReLU; while, in the Figure G.4, the calculation of the input current to neuron 1 is represented, which is different from 0 as it is a positive value.

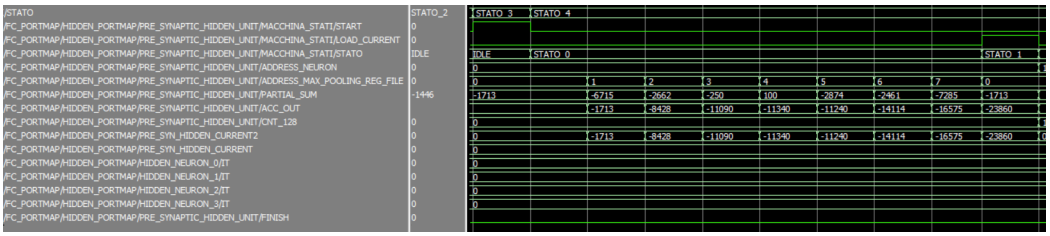


Figure G.3: *Simulation of the Pre Synaptic Layer in the whole architecture*

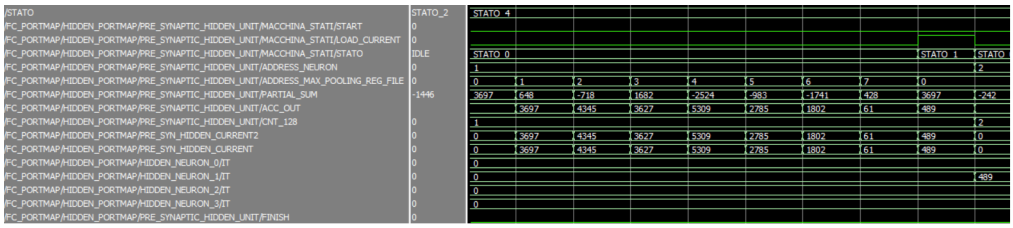


Figure G.4: *Simulation of the Pre Synaptic Layer in the whole architecture*

As soon as the Hidden Pre-Synaptic unit has calculated the input currents to all neurons, the 128 hidden neurons start its computation cycle. In the Figure G.5, it has been reported the trend of the VT variable of the first 5 neurons over several computation cycles, with the respective input currents and the emitted output spikes.

Figure G.5: *Simulation of a part of the Hidden Layer in the whole architecture*

After the computation cycle of the hidden neurons, the processing of the Output Layer starts . Firstly, the output current of the 10 neurons is computed with the Output Pre-Synaptic Unit, as it is explained in the previous

chapter. Then, the 10 output neurons start its computation cycle. In the Figure G.6, it has been reported the trend of 5 output neurons, with the input spikes received from the hidden layer and the respective current values.

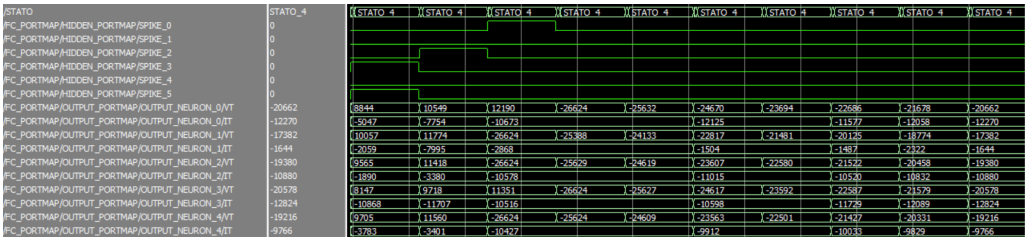


Figure G.6: *Simulation of a part of the Output Layer in the whole architecture*

Finally, when the value of the counter arrives to the integration time, the neural network makes its decision: through the Winner Selector, it controls the outputs of the 10 Integration Neurons and it concludes which the decoded digit is. In the Figure G.7, it has been represented the evolution of the output of the integration neurons in the last computational cycles and the final decision of the neural network, which is stored into the Winner Register, represented by the signal *WINNER_DIGIT*. The Winner Register is reset with the value 15, so that it is not a possible output of the architecture.

Figure G.7: *Simulation of a part of the Final part of the whole architecture*

Bibliography

[1] Maxence Bouvier et al. "Spiking Neural Networks Hardware Implementations and Challenges". In: *ACM Journal on Emerging Technologies in Computing Systems* 15.2 (June 2019), pp. 1–35. ISSN: 1550-4840. DOI: 10.1145/3304103. URL: http://dx.doi.org/10.1145/3304103.

[2] Nassim Abderrahmane, Edgar Lemaire and Benoıˆt Miramond. "Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence". In: *CoRR* abs/1910.01010 (2019). arXiv: 1910.01010. URL: http://arxiv.org/abs/1910.01010.

[3] Charlotte Frenkel, Jean-Didier Legat and David Bol. *A 28-nm Convolutional Neuromorphic Processor Enabling Online Learning with Spike-Based Retinas*. 2020. arXiv: 2005.06318 [cs.NE].

[4] E. M. Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572. DOI: 10.1109/TNN.2003.820440.

[5] A. J. Leigh, M. Mirhassani and R. Muscedere. "An Efficient Spiking Neuron Hardware System Based on the Hardware-Oriented Modified Izhikevich Neuron (HOMIN) Model". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12 (2020), pp. 3377–3381. DOI: 10.1109/TCSII.2020.2984932.

[6] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[7] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[8] Christian Pehle and Jens Egholm Pedersen. *Norse - A deep learning library for spiking neural networks*. Version 0.0.6. Documentation: https://norse.ai/docs/. Jan. 2021. DOI: 10.5281/zenodo.4422025. URL: https://doi.org/10.5281/zenodo.4422025.

[9] P. U. Diehl et al. "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280696.

[10] D. Neil and S. Liu. "Effective sensor fusion with event-based sensors and deep network architectures". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 2282–2285. DOI: 10.1109/ISCAS.2016.7539039.

[11] D. Garbin et al. "Variability-tolerant Convolutional Neural Network for Pattern Recognition applications based on OxRAM synapses". In: *2014 IEEE International Electron Devices Meeting*. 2014, pp. 28.4.1–28.4.4. DOI: 10.1109/IEDM.2014.7047126.

Ringraziamenti

Grazie a tutto il collegio docenti del DET del Politecnico di Torino, in particolare al mio relatore, il prof. Maurizio Martina, e al dott. Alberto Marchisio per avermi accompagnato in questo percorso di tesi, per la disponibilità che mi hanno concesso, per la pazienza che hanno avuto e per il supporto dato in questo momento conclusivo della mia carriera accademica.